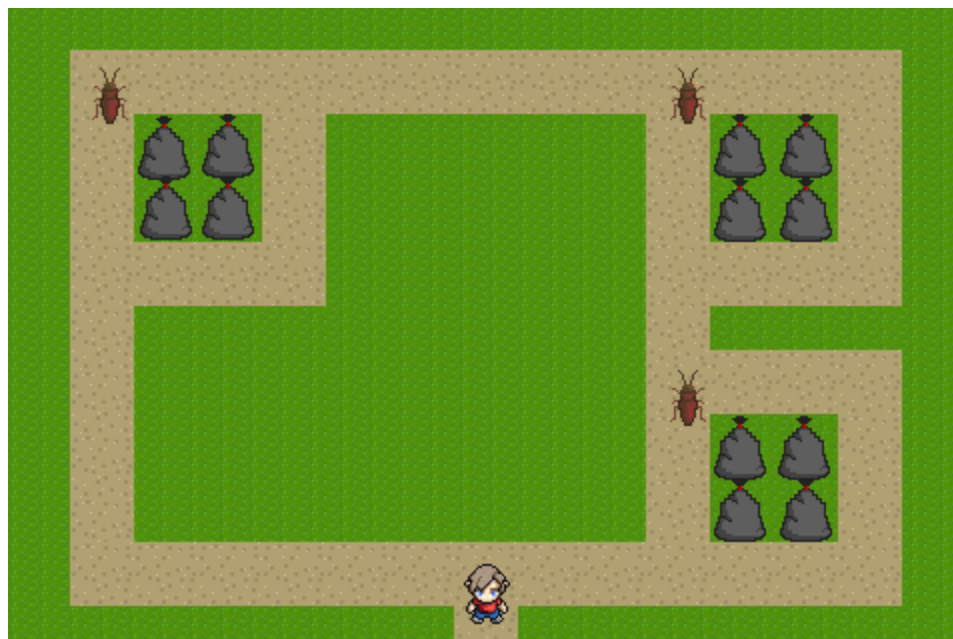


How to Make “Operation Green Clean” in GameSalad

by J. Matthew Griffis

*Note: this is an Intermediate-level tutorial. It is recommended, though not required, to read the separate PDF **GameSalad Basics** and go through the Beginner-level tutorials before continuing.*



In Operation Green Clean, you're on a quest to clean up the park by picking up the garbage, but watch out for the vermin drawn to the stench! The game is simple but provides an introduction to different movement types and thinking about creating interesting obstacles for the player. Most games give the player “enemies” to fight or avoid, and making them interesting is essential.

In this tutorial, you'll learn how to recreate Operation Green Clean. Make sure to download the folder of Resource Files for the game, and play the game on the website to see it in action.

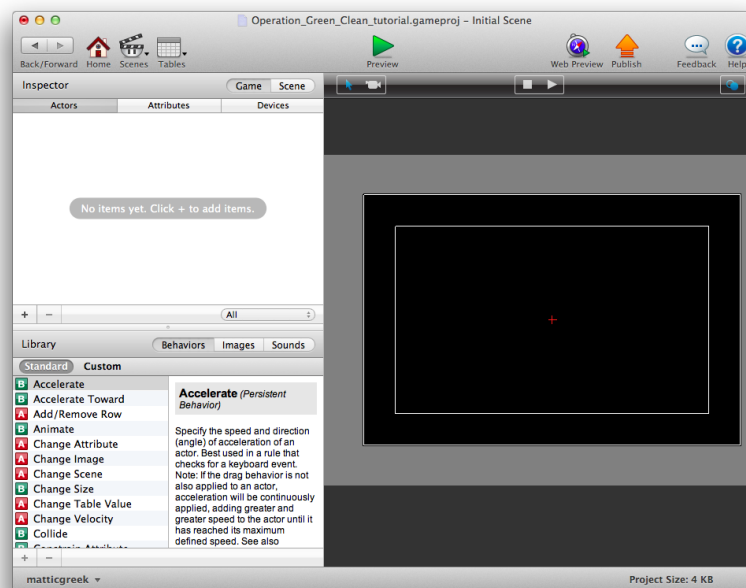
OK! Let's get started.

Create a blank project and fill in the Project Info, making sure to select “GameSalad Arcade” as the Platform if you want to publish the game online. If you don’t, feel free to select another Platform but be advised that your screen may be a different size than the one in this tutorial.

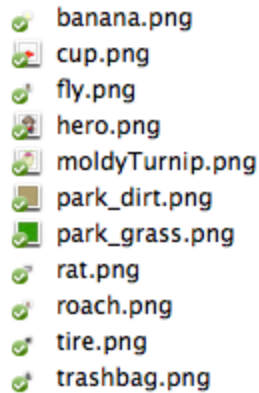
Save your work. Do this often. Saving is your friend.

Alright, let’s make some stuff and put it in our game! This will be a game with three *levels*, or scenarios for the player to play through. GameSalad divides its game settings into “Scenes,” so it makes the most sense to create a Scene for each level.

For now, we’ll work on the first Scene. Click the *Scenes* tab, then double-click on “Initial Scene” to see this:



An empty stage, waiting for us to transform it into an exciting show. Let’s start with hiring...sorry, *creating* some Actors, the elements that will make up the game environment and characters. To do so, look at the Library in the lower-left of the interface and click on the *Images* tab. Now open the Resource Files folder you downloaded and open the Sprites folder. You’ll find many images:



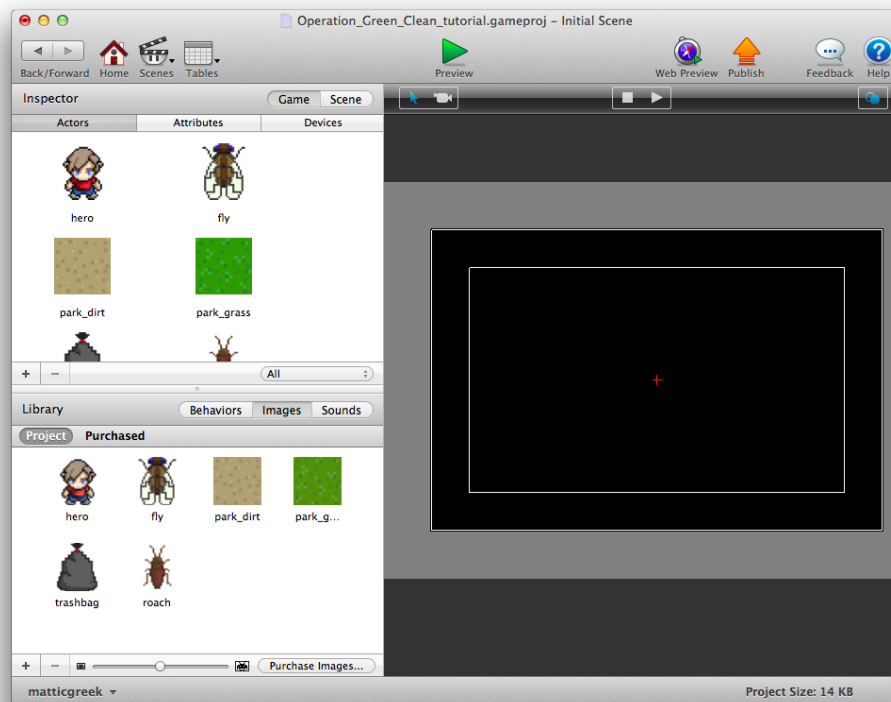
We are not going to use *all* of these in the tutorial, but they are in the folder if you want to use them. Drag the following six into the *Images* box:

- *park_grass.png
- *park_dirt.png
- *hero.png
- *trashbag.png
- *roach.png
- *fly.png

GameSalad will import the sprites. Finally, drag all the images from the Library into the *Actors* box just above in the Inspector. Poof! Six brand spanking new Actors with images, ready to rock and roll.

What have we just done? We've imported images into GameSalad so we can use them for our game. Then we used those images to create Actors, which are the actual elements that make up a game in GameSalad. An image can't be used in a game without being attached to an Actor, and an Actor without an image is just a blank white box.

Your screen should now look something like this:



Go ahead and click the *Behaviors* tab in the Library; we won't be needing to add any further Images, but we will be needing to add *so many* Behaviors. Oh yes.

We'll need to build a *level*. Generally speaking a level is an environment or specific scenario that the player must successfully navigate/overcome. It has a definite start point and a definite end point. What constitutes a level is pretty flexible and varies from game to game.

In this game we move a boy (the hero) about a park, picking up trash bags and avoiding pests. The hero, trash bags and pests all start in a specific place, and there are a specific number of trash bags to pick up and pests to avoid. In this game, then, a level is a single "area" of the park. Picking up all the trash bags in the area wins the level, while running into a pest loses it.

It's tempting to start making the level right away. However, if you played the game on the website, you probably noticed that it uses a unique control scheme. Instead of moving the hero directly via keyboard input, we click on a specific location, prompting the boy to move there on his own. This style of control is common to certain game genres such as point-and-click adventure games and real-time strategy games. It is more challenging to implement, so we should take care of that before anything else. Fortunately we are up to the task!

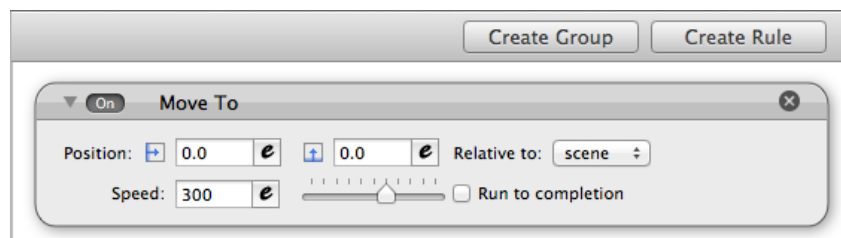
First things first. Right now there is nothing in our Scene. We need a hero! Drag the hero Actor into the Scene to create an instance of the hero. Unlike the Actors in the Inspector (known as “prototypes”), instances will actually appear in the game. Remember to place the hero within the bounds of the black rectangle. That is the area of the game screen. Your result should look something like this:



He won't do anything on his own, so let's give him some movement Behavior. Return to the Inspector and double-click on the hero *prototype* to open its details. Any Behavior we assign to the prototype will apply to all the instances, including the one we created in the Scene.

OK. Let's think about what we want to happen. We want to be able to click somewhere onscreen and have the hero move to where we clicked.

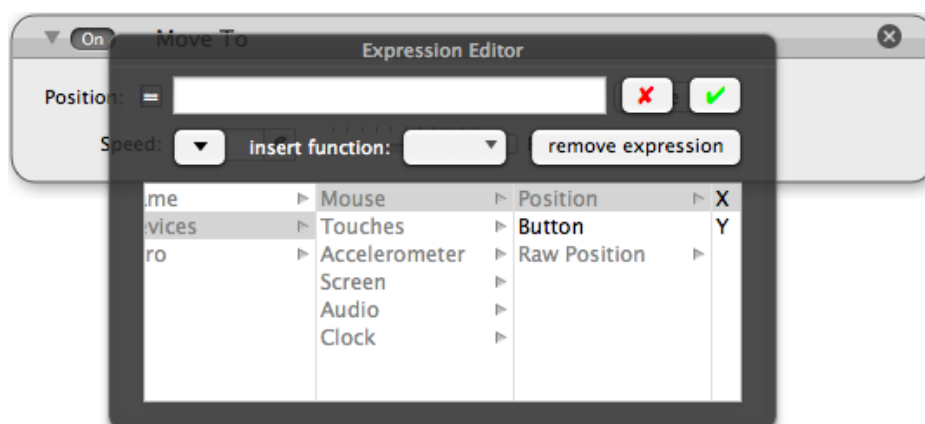
First, let's see if we can get the hero to move to the position of the mouse (not worrying about clicking). GameSalad makes this easy. We'll use the “Move To” Behavior, which lets us set a destination. Find “Move To” in the Library and drag it into the hero's empty Behavior box:



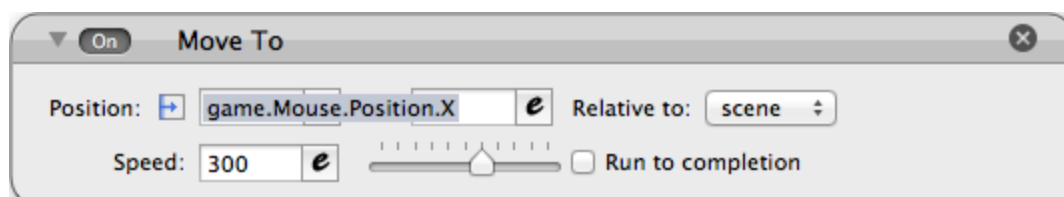
The first Position field is for the x-axis (note the arrow pointing to the right); the second is for

the y-axis (note the arrow pointing up). To the right of each is an “e.” Click on an “e” to open the Expression Editor for that field. This lets you set special values.

We need to tell the hero to move to the position of the computer mouse, but we don’t know what that is! Fortunately, GameSalad tracks the mouse position for us and stores its data so we can use it. Open the Expression Editor for the x-axis and click the down-facing arrowhead on the lower-left. This gives you a list of Attributes, special containers that hold changing values, like the position of the mouse. The Attributes are divided into three categories: Game, which has to do with the settings of the game itself; Devices, which tracks the information from the computer hardware; and hero, which includes the Attributes unique to this Actor. The mouse is a piece of hardware, so choose Devices → Mouse → Position → X:

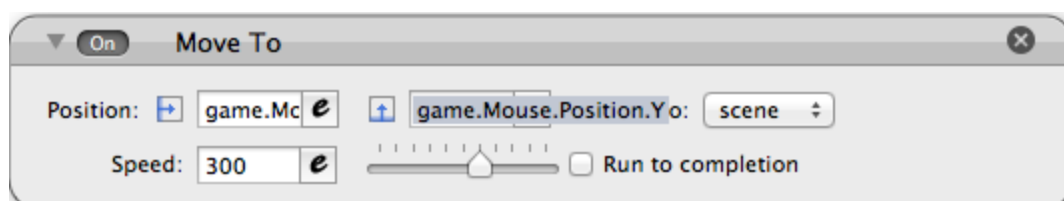


Double-click on “X” to fill the Expression Editor with your selection, then click the green checkmark to save it. Now Move To should look like this:



(If the text is truncated you can hover over it to see the full version.)

Open the Expression Editor for the y-axis and choose the mouse’s y-position the same way:



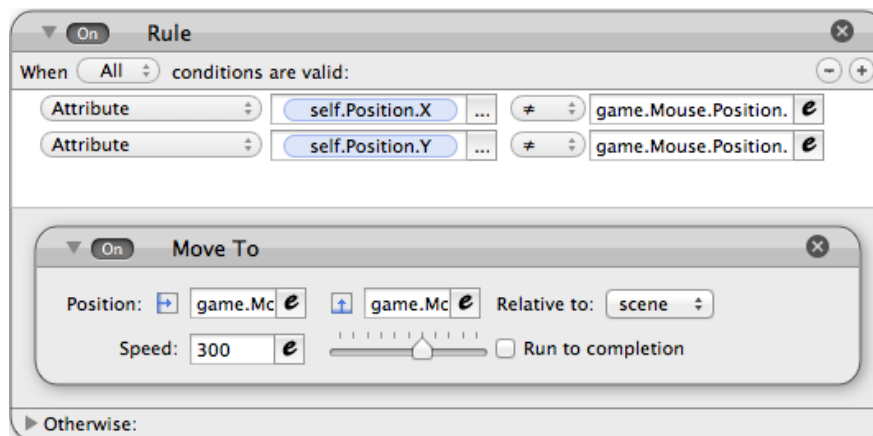
Great work! The hero's default Behavior is to move to the mouse's x- and y-position. Click the large green Preview arrow at the top of GameSalad to run the game. Move the mouse around.

It works! Sort of. That boy sure zips around, doesn't he? You'll notice that as soon as he "touches" the mouse, he stops moving, and he doesn't start again even if you move the mouse away. Why not?

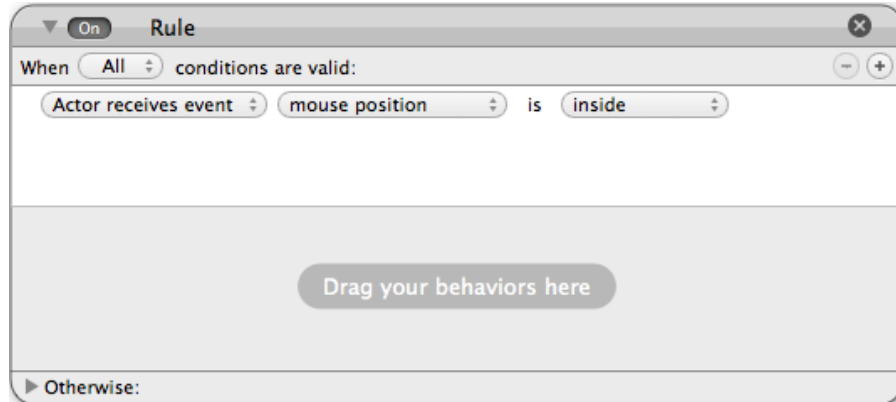
The reason is that the Behavior we created tells the hero to do only one thing: reach the mouse. *This is not the same thing as telling him to follow the mouse.* As we move the mouse around and the hero rushes to catch up, the x- and y-coordinates of the goal change...but as soon as he reaches the mouse, he has completed the objective. Even if the mouse position changes after that, it doesn't affect the hero because he already did what we told him to do.

So, how do we fix this? Well, we could ask the player to complete the game without ever letting the hero touch the mouse. But that seems rather...impractical. Instead, we can create a Rule. A Rule is a conditional statement that says that if something is the case, take action.

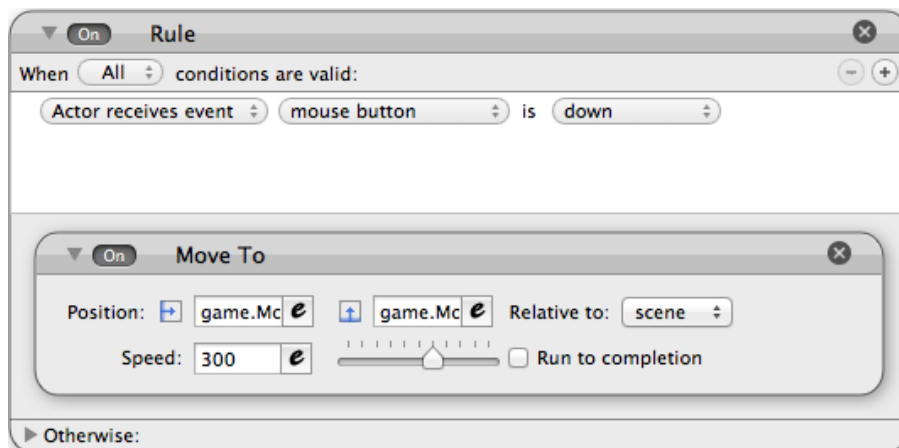
We could create various Rules here. For instance, we could check whether the hero's position is the same as the mouse position, and if not, move him there. (*This is the same thing as telling him to follow the mouse.*) The Rule looks like this (you don't need to create it):



This works just fine, setting a new movement goal every time the positions of the mouse and the hero differ (i.e. repeatedly telling him to move to the mouse). However, we want to be able to move the mouse without moving the hero, and only trigger the hero's movement when we click. Click "Create Rule," which is in the upper-right of the hero's Behavior box. You'll see this:



Change “mouse position” to “mouse button” and make sure the setting is “down.” Drag the Move To we already created into the new Rule. The result should look like this:



If you Preview the game, you’ll see that this works as often as you press the mouse button, but it *only* works if you press the mouse button. As soon as you lift the button, the hero stops moving. We did nothing wrong; it’s just that the Rule only applies the Move To behavior when the mouse is clicked, even if stopping the Behavior means that the hero doesn’t reach the goal.

You can check the “Run to completion” box to make the Move To behavior play out all the way, even if the Rule doesn’t apply anymore (i.e. the mouse button is no longer down). The result is progress: the hero only starts moving when we click, he moves to the mouse’s position even if we release the mouse button, and even after he reaches the mouse and stops moving, we can click to set a new destination and get him moving again. Excellent work.

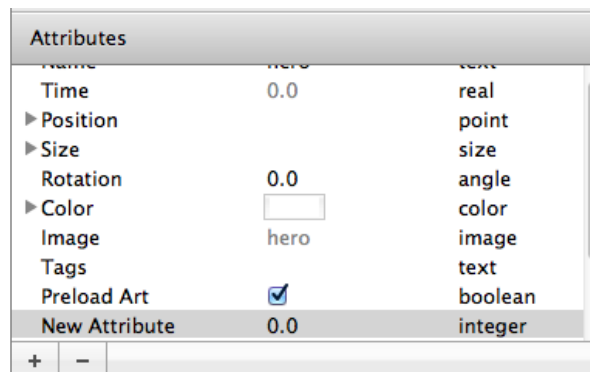
The movement is still not quite right, though. As you’ve surely noticed, the hero changes direction mid-movement to follow the location of the mouse if you move it after clicking, even though you haven’t clicked again to set a different destination. It’s the same situation we had the first time: we’ve issued a single instruction (“move to the mouse’s location”) and then

changed that location while the hero's en route. The hero, being an obedient fellow, shifts accordingly.

In other words, clicking does not really “set” a location. It simply tells the hero to get moving. But we want to click somewhere to make the hero move *there*, regardless of how we move the mouse afterward. We want to set a fixed location.

How do we do this? It's no big deal. We still need to use the “Mouse.Position.X” and “Mouse.Position.Y” Attributes. However, those will change whenever we move the mouse. There's nothing we can do about that. What we need then is a way to record the x- and y-position of the mouse when we click, and store them in something that will not change. We need to create new Attributes.

These will only apply to the hero, so let's create them in the hero prototype. In the Attributes section, just below the hero's sprite, click the plus button in the lower-left. You'll get a pop-up asking what kind of Attribute you want to create. We'll be storing simple numbers, so choose “integer.”

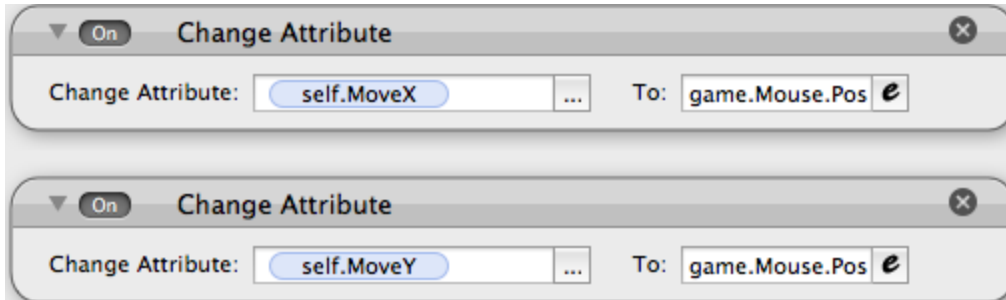


Attributes		
Name	Value	Type
Time	0.0	real
▶ Position		point
▶ Size		size
Rotation	0.0	angle
▶ Color		color
Image	hero	image
Tags		text
Preload Art	<input checked="" type="checkbox"/>	boolean
New Attribute	0.0	integer

Poof! There it is! Double-click on “New Attribute” and rename it “MoveX.” We'll need a second one, so create another integer and rename it “MoveY.”

Preload Art	<input checked="" type="checkbox"/>	boolean
MoveX	0.0	integer
MoveY	0.0	integer

Great! We have our containers. Now to store things in them. Return to the hero's Behavior. Drag the “Change Attribute” behavior from the Library into our “mouse button is down” Rule. We'll change MoveX (“hero → MoveX”) to the mouse's x-position (“Devices → Mouse → Position → X”). Drag Change Attribute into the Rule a second time and set it to change “MoveY” to the mouse's y-position the same way.

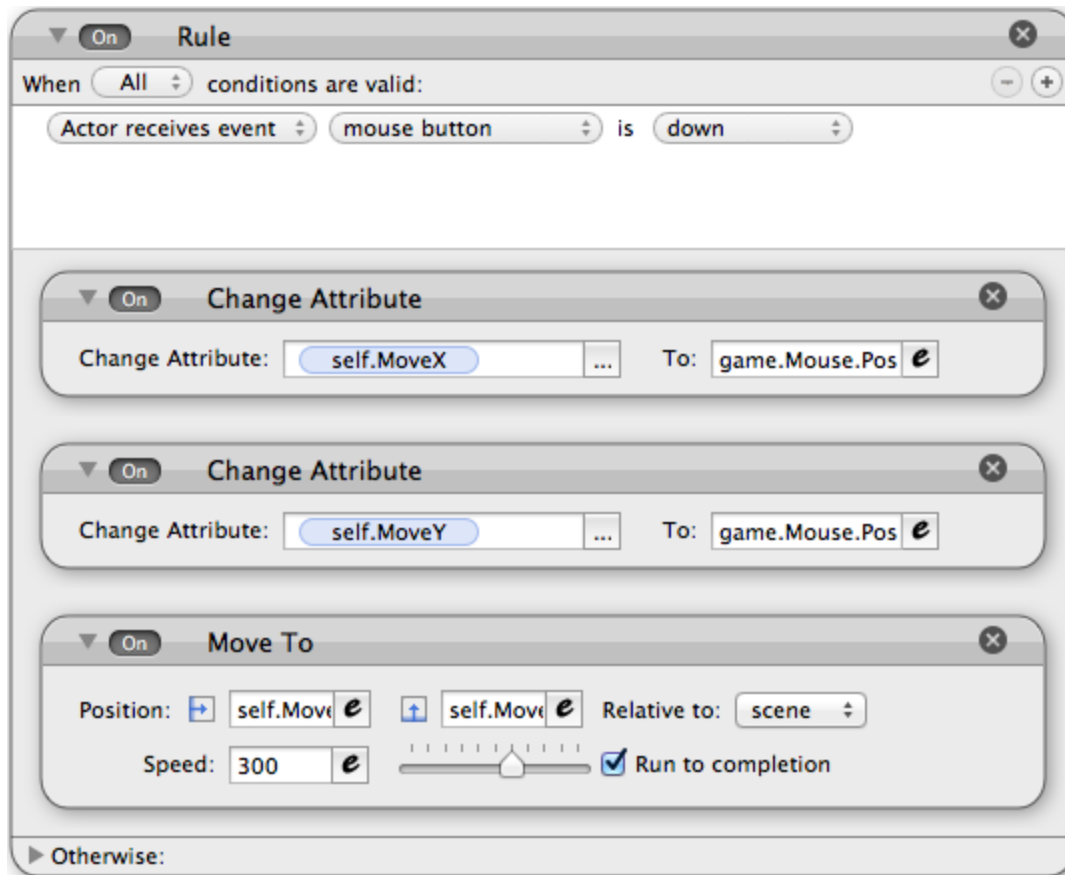


What have we just done? Don't be confused when we talk about "changing" our new Attributes to the position of the mouse. What we're actually doing is changing our Attributes' values; i.e. copying the current value of the mouse's x-position into our MoveX Attribute, and the current value of the mouse's y-position into our MoveY Attribute, all with a click. That means that MoveX and MoveY store the position of the mouse, but only at the moment that we click; MoveX and MoveY don't change any further even if the mouse's position does. Unless, of course, we click again and store a new set of fixed values.

This won't make any difference if we don't update our Move To behavior. Change the destination from the mouse's x- and y-position to MoveX and MoveY.

Note: Make sure you place the Move To behavior below the two Change Attribute behaviors. Otherwise you'll get an odd teleportation effect, where the hero instantly appears wherever you click. That happens because the Rule starts the hero moving toward MoveX and MoveY (which have their original values) and then immediately changes them to the position of the mouse, all within the same event. If that's confusing to you, think how it makes the computer feel! By changing the values first and then starting the hero moving, we avoid confusion.

The Rule now looks like this:



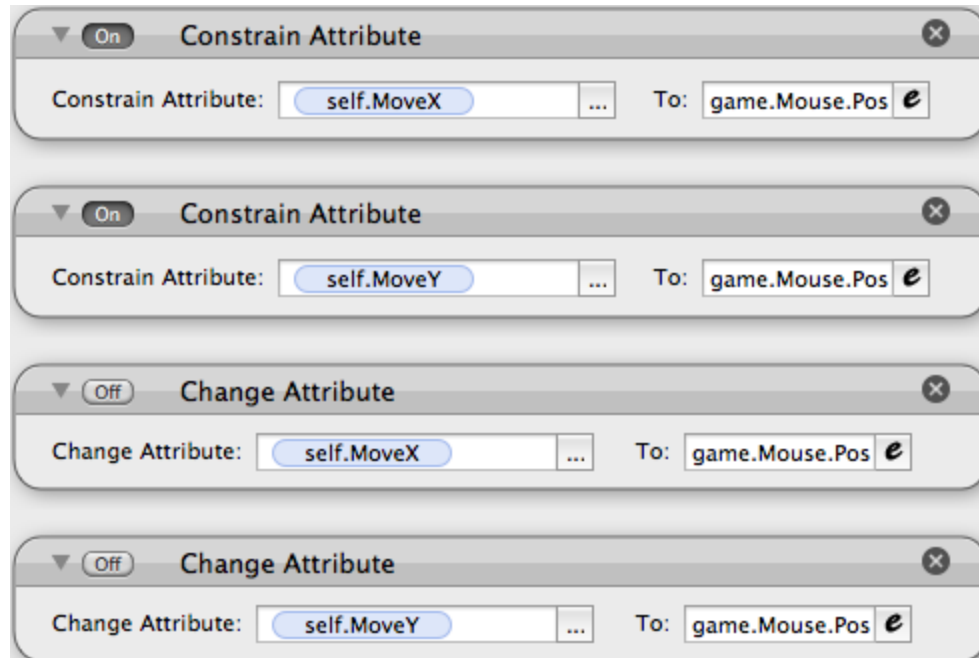
Preview the game. Alright! The hero moves to wherever we last clicked the mouse, even if we move the mouse afterward. Nice work!

We could leave it there, and it's OK to do so. But what if we want to be able to hold the mouse button down and keep updating the hero's destination? Think of it as combining the best parts of all the previous versions of his movement. If we click and release, he'll head towards the click, but if we hold the button down, he'll follow the mouse as we move it about, and if he touches the mouse cursor and then we move it away from him, he'll start moving again, without our having to release and then press the mouse button a second time. This is the control the website version of the game uses. If you're interested, read on; if you've had enough of complex mouse controls, feel free to skip over the following italicized section and jump ahead to the rest of the game.

Still here? Great! It only takes a few more steps to achieve our advanced movement goal. If you Preview the game and click somewhere, then move the mouse elsewhere while the hero is still moving, you'll see that he doesn't change his path to reach the new mouse position, even if you held the button down. Of course, that's exactly what we just set out to achieve. It's working!

But the Rule says to change MoveX and MoveY to the mouse's x- and y-position when the mouse button is down. Why doesn't it keep changing MoveX and MoveY as you move the mouse around with the button held down? The reason is that GameSalad considers when the mouse button is down to be a single event--which ends when the mouse button is released--and the Change Attribute behavior occurs only once per event. You'd have to click again to reactivate it.

What to do? There is another type of Behavior called "Constrain Attribute." Like Change Attribute, Constrain Attribute changes the value of an Attribute to something else, but unlike Change Attribute, it does so repeatedly. Drag two copies of Constrain Attribute into the "mouse button is down" Rule and set them up to change MoveX and MoveY just like you did the two Change Attribute behaviors. Once you've done so, you no longer need the Change Attribute behaviors, so click the "x" in the upper-right to remove them, or click "On" to turn them off.

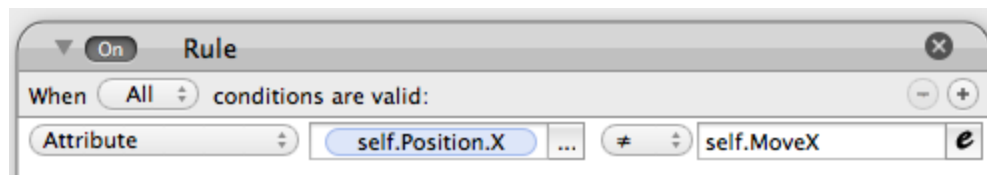


Preview the game and you'll see that we're mostly there, but we have the problem we had before in which the hero stops moving once he touches the mouse, even if we're still holding the button and we move the mouse again. The reason is the same as before: when we click we tell the hero to do one thing, which is to move to a certain point, and when he reaches it, that is that. Even though we may update his destination by moving the mouse while he is en route, he is still following a single order, and until we release the mouse button and click again, there is no new order.

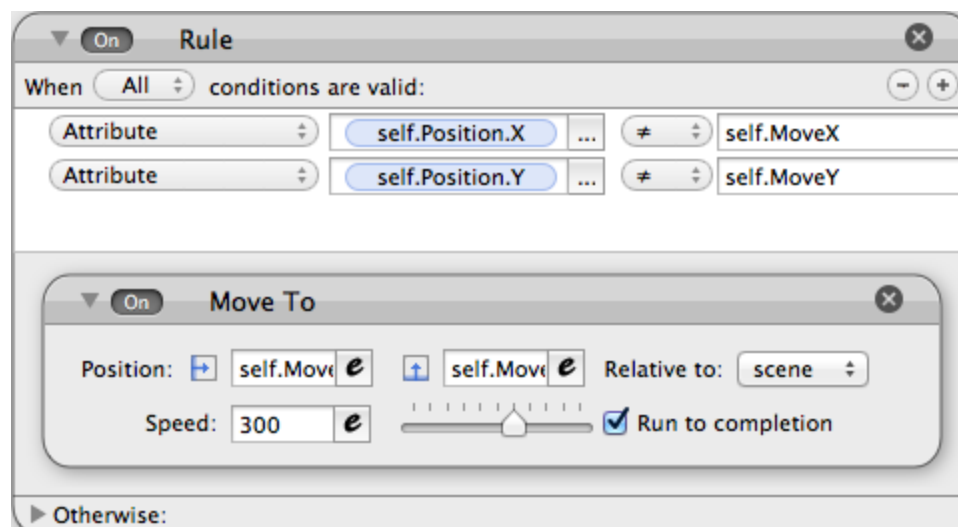
This is just like what happened with Change Attribute, but there's no equivalent to "Constrain Attribute" ("Move To Repeatedly"?) to help us out this time. Fortunately we don't need one.

Instead, we use another Rule. Remember that a Rule is always checking whether something is the case. We will check whether the hero's current x- and y-positions are the same as MoveX and MoveY, and if not, we'll trigger the Move To behavior.

Returning to the hero prototype's Behavior, click "Create Rule." Drag the new Rule into the "mouse button is down" Rule and place it just beneath the two "Constrain Attribute" behaviors. Change "Actor receives event" to "Attribute"; choose "hero → Position → X" as the Attribute; click the equals sign and change it to "does not equal" (looks like an equals sign with a diagonal slash through it); finally, click on the "e" to open the Expression Editor, click the down-arrow on the left and choose "hero → MoveX." Whew!



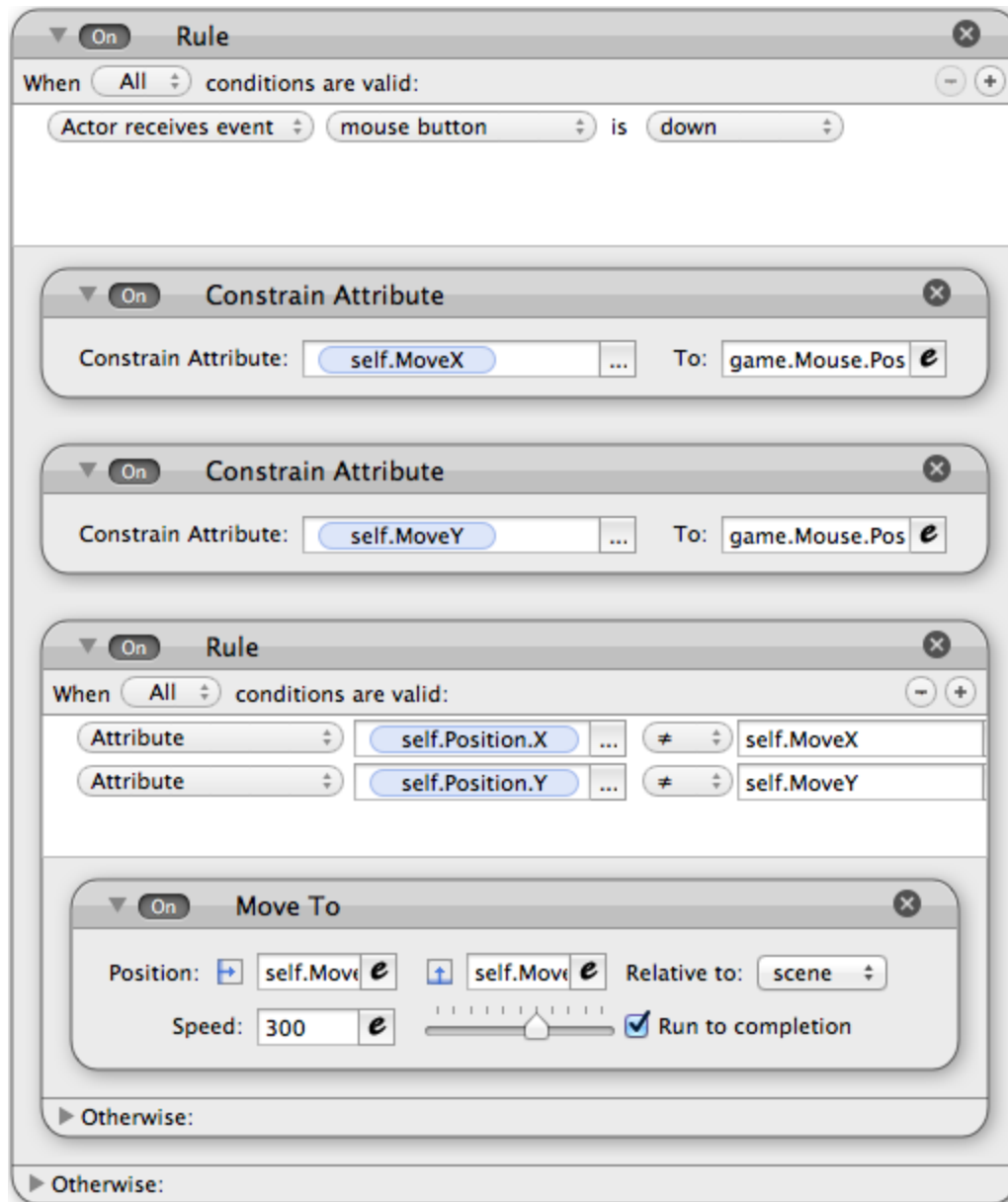
We're not done! Click the plus in the upper-right to create an additional condition. Set it to check if the hero's y-position is not equal to MoveY. Finally, note that it says "When All conditions are valid" at the top. We want the hero to move even if only one of the two conditions is not true, so click "All" and change it to "Any." We're still not done! Right now the Rule doesn't do anything! Drag our Move To behavior into the Rule. The final result looks like this:



You may recognize this as a modified version of the example from earlier in the tutorial. It wasn't the best solution then but now we're putting it to use!

Take a moment and look at the entirety of the hero's Behavior. It consists of a single Rule, checking if the mouse button is down. If the button is down, GameSalad stores the mouse's x-

and y-positions in the MoveX and MoveY Attributes we created, then checks to see if the hero's current position (x, y) is different than the destination (MoveX, MoveY). If there is a difference, the hero starts moving towards the destination, and keeps moving until he gets there, even if the mouse button is released in the meantime. Here's the whole thing:



Preview the game. Success! The hero moves just the way we wanted. Nice work! You've implemented a sophisticated movement system. Pat yourself on the back, then continue.

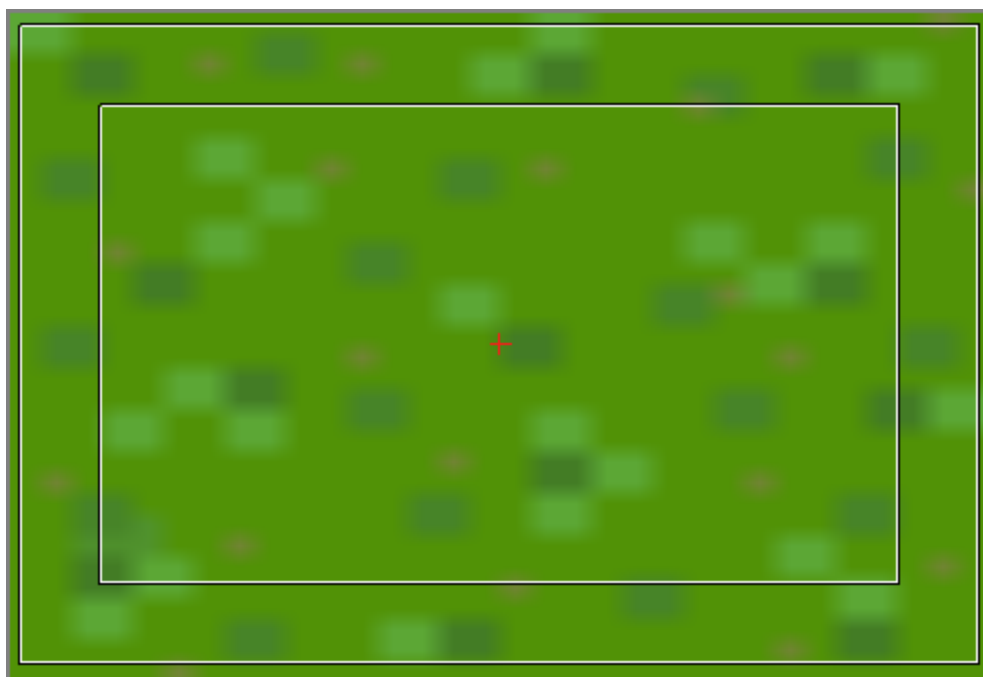
Alright! The hero moves! But he moves through...nothing. Nothing but a dark void. It's rather existential, and too heavy for our game! He's supposed to be in a park, so let's make a park.

First, the grass. Drag the “park_grass” prototype into the Scene. You will almost certainly notice immediately that the grass square is so tiny that it’s useless. Imagine filling the Scene with that! Don’t worry, you don’t have to. Delete that instance, then double-click on the prototype. Click the down-arrow to the left of “Size.” The actual grass image is 32 x 32, so let’s make the Actor that size, too. Double-click on the Width and Height numbers and change them to 32 as needed. Take a moment to do the same thing with the “park_dirt” prototype. We’ll need it shortly.

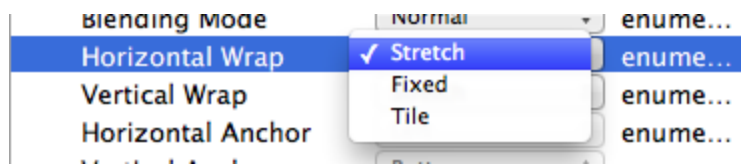
▼ Size		size
Width	32	real
Height	32	real

Now if you drag the grass into the Scene, you’ll see it’s a much more workable size, though still far smaller than the Scene. What we have is what’s called a “tile”: a rectangle (typically a square) of a repeating pattern that we can use over and over again to build a larger image (of whatever shape we like) featuring the same pattern. That process is called “tiling,” and it’s a very common way to create a level, especially a background like sky or...grass.

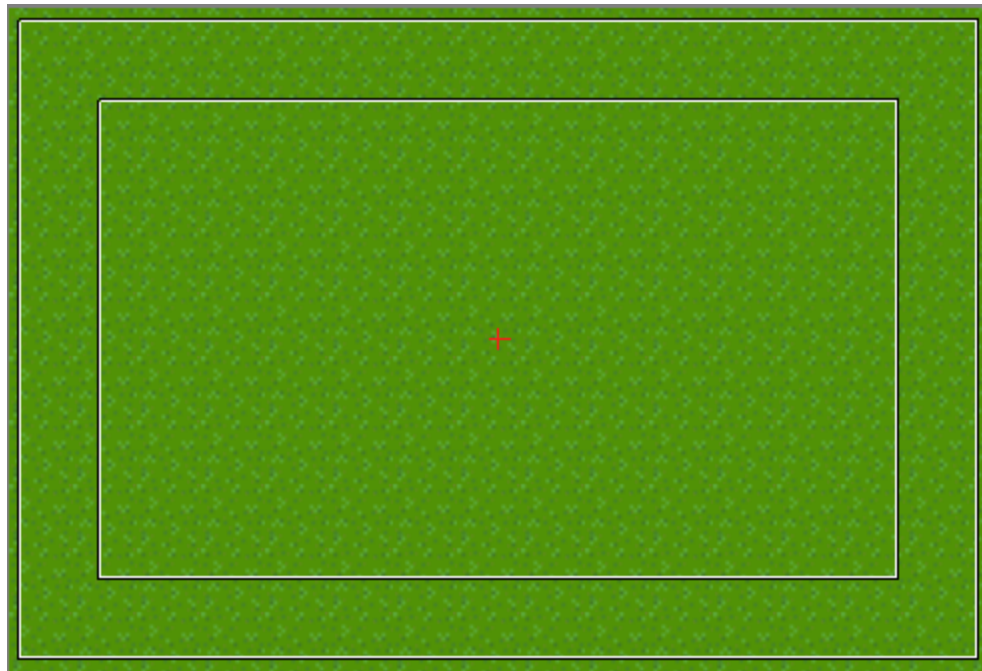
We could create many instances and use them to cover the Scene with grass, but there’s an easier way. Click on the instance we already placed to select it, then drag the circles at its edges to stretch it until it fills the whole Scene. Don’t worry about lining it up exactly with the edges of the black rectangle; remember that the rectangle is only the limit of what the game screen shows, so it’s fine to drag the grass edges beyond the rectangle, to ensure the entire game screen is covered.



“Hang on,” you say. “The grass looks terrible! It’s all blurry and low-resolution.” You’re right. That’s because we stretched a 32 x 32 image to fill a space many times that size. But remember that the Actor is not the *same* as the image. The image is a “cover” for the Actor, nothing more. And that means we have options for *how* the image covers the Actor. Double-click on the grass prototype again and scroll down through the list of its Attributes. At the very bottom, you’ll see “Graphics,” “Motion” and “Physics.” Click the arrow to the left of “Graphics” to expand that category, and scroll down until you come to “Horizontal Wrap” and “Vertical Wrap.” These control how the image covers the Actor. By default, they are set to “Stretch,” which means a single copy of the image stretches to fill the full dimensions of the Actor. That’s fine if image and Actor are the same size. But if you click on “Stretch”...



Do any of those look familiar? Yes! Set both Horizontal Wrap and Vertical Wrap to “Tile.” Click the Back button to return to the Scene view.



So much better! The image repeats horizontally and vertically as much as it needs to in order to fully cover the Actor. Tiling is a great technique so don’t be afraid to use it.

You may be wondering what happened to the hero. The answer is that we covered him with

the grass. How cruel! This is because by default GameSalad displays things in the order you placed them in the Scene, with newer things appearing in front of older things. But we can change this. Right-click on the grass (on Mac, click to select the grass, then click again while holding Control) and choose “Send To Back.” This moves the grass to the back of the *layer*.

Think of a *layer* as a single sheet of transparent paper. You could draw a tree and a house on one layer and a family on another, then put one layer atop the other to see the combined picture. Within the first layer, you might decide to draw the tree in front of the house, or vice-versa, after which you’re more or less stuck. Fortunately, with a digital layer, you can change your mind later and move the house in front of the tree (or vice-versa)!

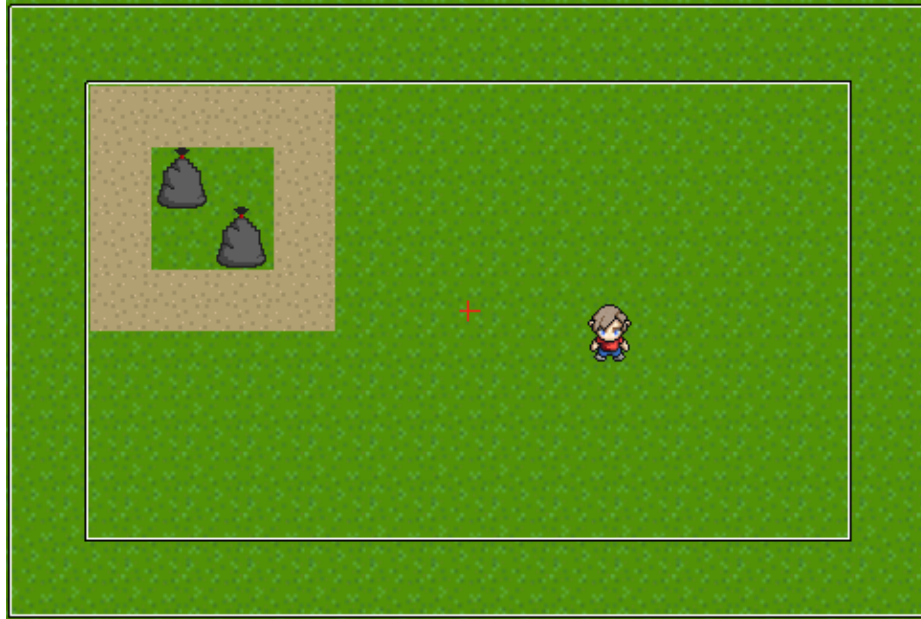
*Note: any GameSalad project starts out with a single layer. Anything you add to the Scene will be added to that layer. However, you can create new layers and distribute your Actors among them. We will not do so for this tutorial, but feel free to play around by going to the Scene tab in the Inspector, then clicking the Layers subtab. See **GameSalad Basics** for more on layers.*

Let’s take a moment to think about what else we need to add to the Scene. We need some trash bags, which we’ll make the hero collect. We need some dirt paths surrounding those trash bags. And we need some pests to travel the dirt paths and threaten the hero.

Let’s start with the trash bags and dirt paths. Place some trash bags in the Scene. Now is the time for you to put on your level designer cap, because the details are up to you! Just remember that one trash bag is roughly the size of one dirt tile (you changed the dirt to 32 x 32, right?), and that we want rectangular dirt paths--anything diagonal or curved would be too difficult to make the pests follow. As for the dirt, you can use many individual instance of the tile, or stretch a single tile (remember to change the Horizontal and Vertical Wrap settings to “Tile” if you do).

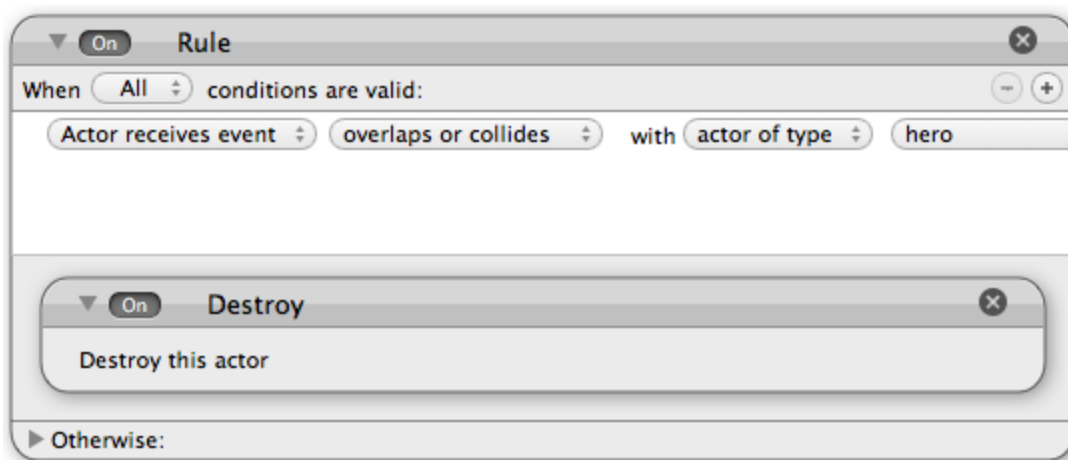
A quick note about the grass. You’ll probably notice that it is *really* easy to select the grass when you mean to select something else. This is unavoidable since the grass is everywhere. For this reason you may want to move the grass off to the side (outside the bounds of the Scene) and build the rest of the level, then drag the grass back into place.

Here’s an example layout that’ll do for our needs; please make yours more interesting than this:



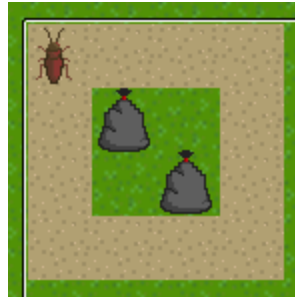
The dirt paths don't need to do anything; like the grass, they're just part of the background. We have to define behavior for the trash bags though, since the hero is supposed to collect them. What should happen when the hero runs into a trash bag? Eventually we'll want to subtract from a tally of total trash bags (to determine if he's collected all the trash and won the level), and if we were going to put this game out for release we'd probably want to add other feedback such as sound effects and maybe a point bonus. For now though, let's just make the bag disappear.

This is simple as could be. Double-click on the trash bag prototype, then create a Rule. Leave it set to "Actor receives event" but change "mouse position" to "overlaps or collides" and make sure the "actor of type" is set to "hero." Finally, drag the Destroy behavior into the Rule. Voilà:



If you Preview the game and run the hero into a trash bag, you'll see it disappears.

We're close to done at this point! Two tasks remain: creating the pests, and setting the win/lose condition(s) for the level(s). Since we need the pests to make losing the level possible, it makes sense to tackle them first. So! First things first. Let's add a pest! Drag the roach Actor into the Scene, and remember that it is going to travel the dirt path around the trash bags, like a guard:



Alongside making the controls for the hero, setting the pests' movement patterns is one of the real challenges in creating this kind of game. In fact, if there's one thing that characterizes Operation Green Clean, it's complex movement (just be glad the trash bags don't move). Don't be alarmed! What we're going to do is still not that hard. But it requires a little more forethought, calculation, and good old trial-and-error.

In many games, the enemies have very simple movement systems. They only go in one direction, or they bump into an obstacle and reverse direction, or they patrol one platform, back and forth. The pests in Operation Green Clean are pretty dumb--they don't have any special behavior to attack the hero or anything. But they do travel an actual path, "following" the dirt trails we created. Of course, they're not really following the dirt; they have no awareness of it!¹ But we make it look like that's what they're doing.

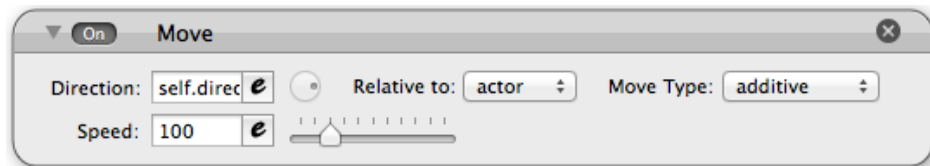
How do we do this? As always, there are many possible solutions. Regardless of our preferred implementation, if we want a pest to "patrol" the dirt square around the trash bags, it is going to need to change direction four times, i.e. every time it comes to a corner of the square.

Now, we could place invisible obstacles (i.e. Actors with no sprites) at the edges of the square, and set the pest's behavior to change direction upon collision with those obstacles. Or we could use other methods. The simplest for our purpose is probably to use time. That is, we'll say that the roach should change direction every so often. The advantage of using a square path is that, since every side is the same length, the amount of time between directional changes is constant.

¹ Although, there's nothing stopping us from *giving* them an awareness of the dirt. If we really wanted to get fancy we could make the roach only move in the direction of dirt, but that's beyond the scope of the tutorial.

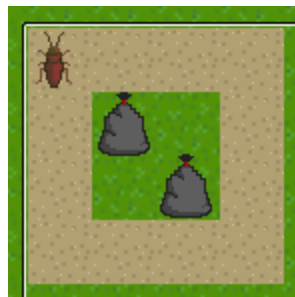
The first thing to do is to get the roach in motion. Open up the roach prototype and give it the Move behavior. If we just leave it at that and run the game, the roach moves. But it just moves in one direction (the default right) and keeps going, right off the screen. Even if we change the direction in the settings of the Move behavior, we'll have the same problem, because the direction is fixed. We need to use a direction with a value that we can change during gameplay, which means we need to create an Attribute.

We do so the same way we did earlier for the hero. Look on the left under the roach's sprite at its list of Attributes, and click the plus button in the lower-left. Choose "integer" from the Attribute Type menu, then double-click on "New Attribute" and rename it to "direction." Inside the roach's Move behavior, open the Expression Editor to the right of the "Direction" field and choose our new Attribute ("roach → direction"). You may wish to take this opportunity to slow down the roach's movement speed, and maybe also that of the hero. You don't want anyone moving too quickly across a small level. Now the roach's Behavior should look like this:



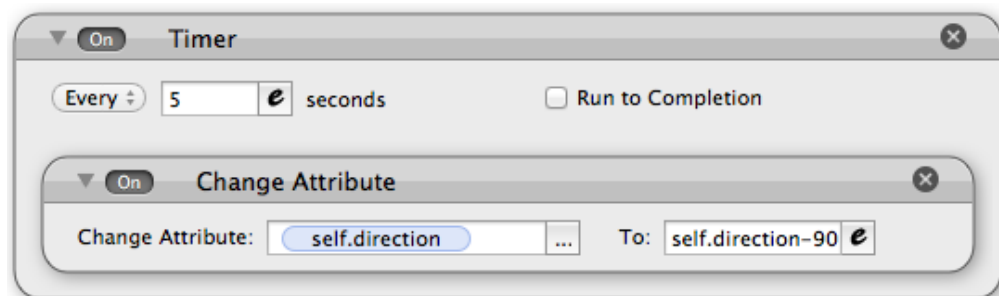
Of course, the roach still moves in just the one direction, but now it's based on an Attribute, which means we can change the direction. Think of the roach's direction as an *angle*, measured against the x-axis. Remember from geometry that angles go from zero to 360 degrees, moving counter-clockwise. Straight right is zero degrees. Straight up is 90 degrees. Straight left is 180 degrees. Straight down is 270 degrees. Straight right (again) is 360 degrees, functionally the same as zero degrees.

Perhaps you can see where this is going. To make the roach turn at a right angle, we simply have to modify "direction" by 90. We can add or subtract. Furthermore, if we want the roach to move clockwise instead, we can use negative numbers (-90 is the same as 270; they both mean straight down). So, let's figure out how the math should work. Here, again, is what we have:



By default the roach starts moving to the right (zero degrees). When it reaches the upper-right corner of the square, it should start moving down (-90 degrees). When it reaches the lower-right corner, it should start moving left (-180 degrees). When it reaches the lower-left corner, it should start moving up (-270 degrees). And when it reaches the upper-left corner, it should start moving right again (-360 degrees, i.e. zero degrees). So, every so often, we should subtract 90 from “direction.” We just have to figure out *how* often.

This is where the trial-and-error comes in. First, we need to set a timer. Give the Timer behavior to the roach prototype, and make sure not to put it inside any other Behavior or Rule. Now, drag the Change Attribute behavior into the Timer. Select “direction” as the Attribute to change, and open the Expression Editor to the right of the “To” field. Select “direction” again, and then type “-90.” The result should look something like this:

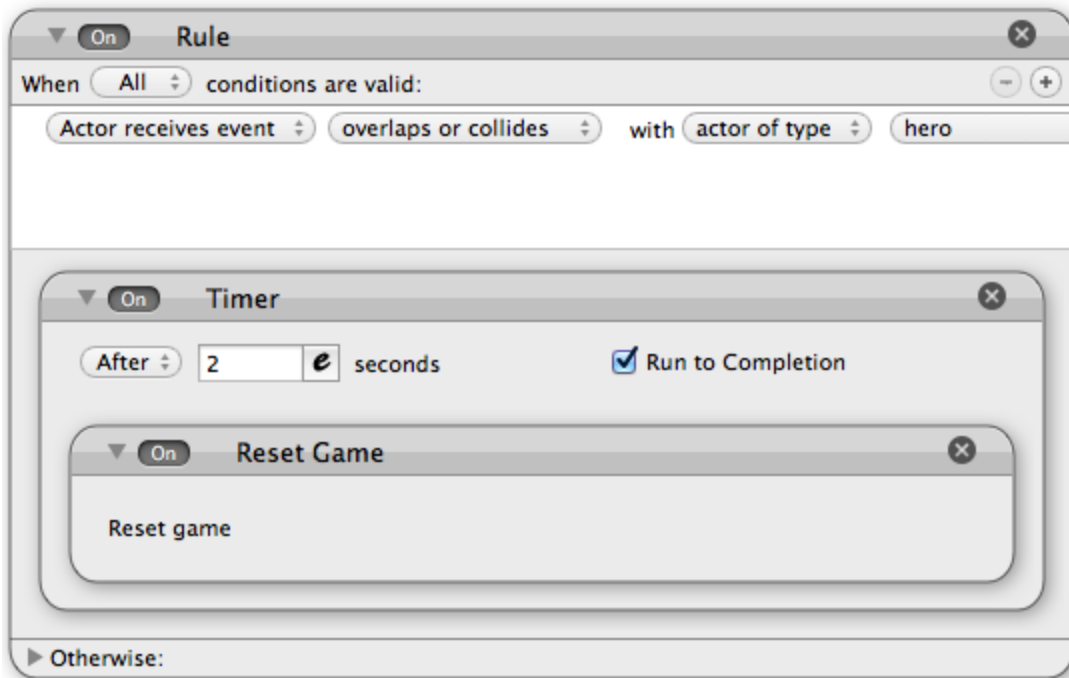


In other words, this says that every five seconds GameSalad subtracts 90 from “direction.”

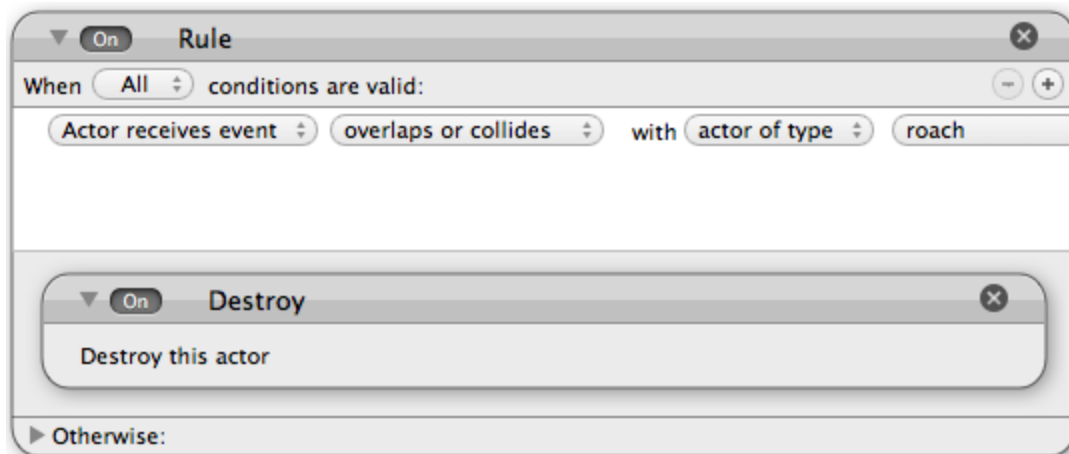
Five seconds is almost certainly far too long, of course. If you made a layout like the one above, 1 second will be just about perfect. Change the interval to some small value like 1 second so you can make sure the roach is changing direction correctly, then play with the interval (and the roach’s speed) to get the timing just the way you want it. The end result should be that the roach moves perpetually around the dirt path. Nice work! You’ve made a patrolling enemy!

The “enemy” doesn’t do anything to the hero as yet, though. We’d better fix that. Add another Rule to the roach and set it to check for a collision with the hero. Give it the Reset Game behavior. Pests have no mercy!

Note that if the game restarts immediately upon the hero’s colliding with an enemy, the player may not have time to realize why he or she lost. For that reason it’s good to put in a slight delay. You could put another copy of the Timer behavior inside the Rule for collision with the hero, have it wait a couple of seconds (change the dropdown from “Every” to “After”), and only then reset the game. If you do that, make sure to check “Run to completion”; otherwise the Rule will lose effect as soon as there’s no longer a collision. The result:



Now open the hero prototype and create a Rule to destroy the Actor on collision with the roach.



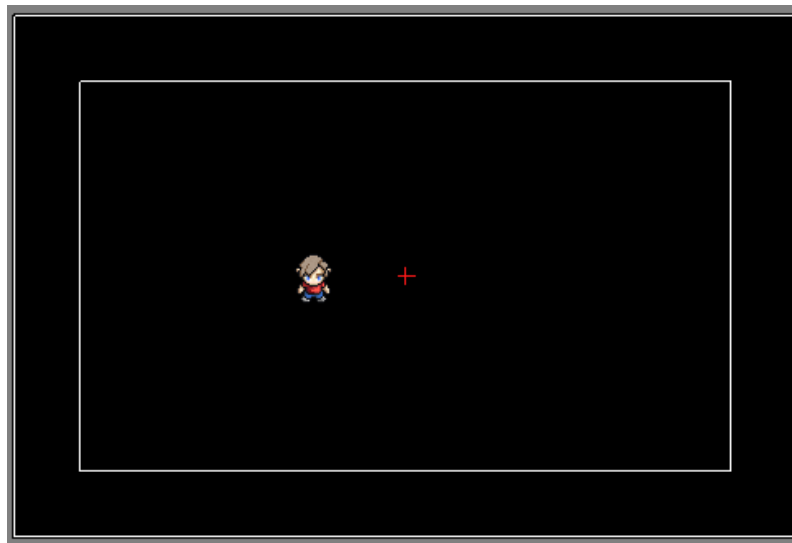
Try the game out. Now the hero should disappear as soon as he collides with the roach, and then a couple of seconds later the game should restart.

Excellent work! Consider what we've accomplished. We have a playable character. We have an environment for him to navigate. We have objects for him to collect. And we have enemies on patrol for him to avoid. These are (almost) all the ingredients of a game!

We even have a lose condition (touching an enemy restarts the game). The last thing to add is

a win condition. After all, the hero isn't really "collecting" the trash bags, because there's no system to count how many he's picked up and reward him for reaching a certain total. We need to make something happen when he collects them all. Let's make him move to the next level.

Of course, we don't have a next level. We'd better create one. Click the Home button, then select the *Scenes* tab. Click the plus button in the lower-left to create a new Scene. Double-click on the Scene to enter Scene view, then add some content to it. For now, here's the minimum:

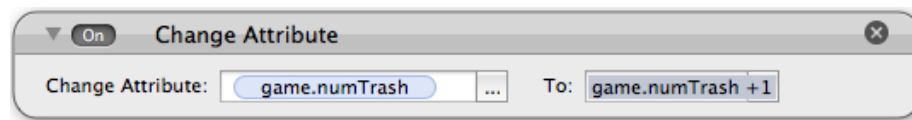


Yes, back in the void. Well, it'll do for our immediate purpose. Return to the initial Scene. We need to count how many trash bags there are. Or, rather, we need to tell GameSalad to count. That means we need to create another Attribute. This count should exist independently of any individual Actor instance, and we'll likely want to use it in multiple Scenes, so we should create it as a Game Attribute. In the Inspector, make sure the *Game* tab is selected, then click on "Attributes." Click the plus button in the lower-left, choose "integer," and rename it "numTrash."

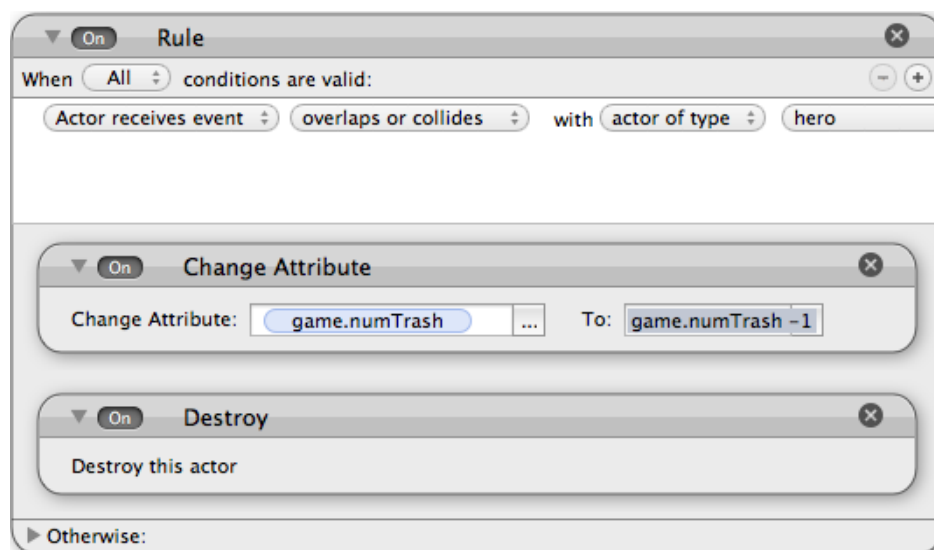
Inspector			Game	Scene
Actors	Attributes	Devices		
Name	default name		text	
Time	0.0		real	
► Display Size			size	
Actor Tags			text	
numTrash	0.0		integer	

Next, click away from "Attributes" onto "Actors" and open up the trash bag prototype. Adding one to the count is as simple as giving the trash bag the Change Attribute behavior (outside of

any other Behavior or Rule) and setting it to change numTrash to numTrash+1. This Behavior runs once, immediately upon the trash bag's creation in the game:



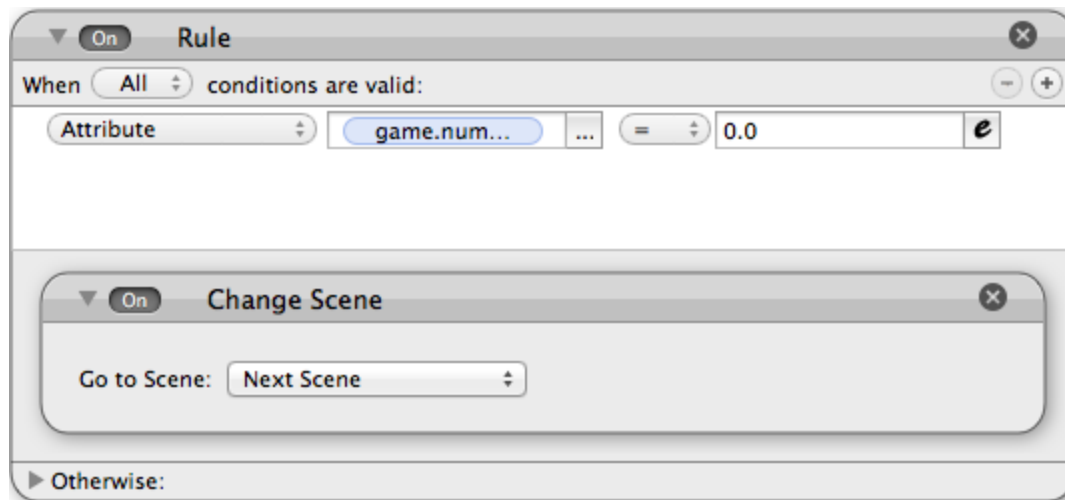
Finally, we need to subtract one from the count whenever the hero picks up a trash bag. This is very easy. Simply add the exact same Change Attribute behavior to the trash bag's Rule about collision with the hero, except that it subtracts one from numTrash instead of adding one:



Here is an important note about Attributes like this. They don't necessarily reset to their original value automatically. Because we created a Game Attribute, if we only restarted the Scene (rather than the game) upon collision with an enemy, numTrash would not reset to zero. So if you started playing the Scene and numTrash counted up to two (for the two trash bags) and then you collected one trash bag (changing numTrash to one), and then collided with an enemy, causing the Scene to restart, numTrash would start out at one instead of zero, because it kept its value from when you last played the Scene. But both trash bags would reappear, bumping numTrash up to three...and thereby making it impossible to win, since there wouldn't be enough trash bags to get numTrash to zero. As such, we would need to reset numTrash to zero ourselves whenever we restarted the Scene. However, since we restart the entire game, we avoid this problem.

Finally, we need to tell GameSalad to move to the next level (i.e. Scene) when numTrash reaches zero. We can add this Behavior to the hero. Open up the hero prototype and create a new Rule. Set it to check when numTrash is equal to zero, and give it the Change Scene

behavior, which by default goes to the next Scene:



Test the game out. Now when you collect all the trash bags, the game should load the next level, if there is one.

That's it! You now have a working game. Of course, it's not very fun. The game so far is extremely simplistic, with only one enemy type and the bare minimum of level design. There's hardly any challenge, and no sense of progression. Fortunately you have all the tools you need to improve upon this design and create a fun, challenging game. Here are some suggestions:

***Make more levels!** Perhaps you already built more complex layouts than those in the tutorial pictures. Keep going! If you're using the GameSalad Arcade setting, then the game screen is not very big, but even within that small area you can create a variety of interesting layouts. Remember that, while gameplay is king, aesthetics play a big part in making levels different and interesting. The dirt paths have no practical impact whatsoever on gameplay, but they serve as a visual cue for where enemies are likely to go (an important psychological role, which you can exploit), and they break up the monotony of the grass field, so change up the path layouts in each level! Of course you can--and should--vary the number of enemies and trash bags too.

***Create tension.** One slow-moving enemy is not very threatening, because the entire rest of the level is safe, and players can choose the perfect moment to make their move. But four slow-moving enemies, all starting at different points on the level with only narrow channels of grass separating them? Suddenly the player must move much more carefully and be prepared to think (and dodge) fast. The enemies aren't any more dangerous, at least individually, but their quantity and proximity to one another greatly change the dynamic of the

level.

If that's not enough you can do things like putting players in danger right at the start of the level, so if they don't move out of the way immediately they'll lose. Be careful, though. You don't want the design to be unfair. There should be sufficient time for them to react if they're paying attention but not so much that they can afford to be lazy.

***Escalate the difficulty.** Of course this and all the following points are related to creating tension, but they deserve their own special mention. Generally speaking you want to start the game off easy and gradually increase the difficulty. If you make the game too hard right up front, the player will get frustrated and quit, but if you start off easy and never make the game harder, the player will get bored and quit. Increasing the quantity of items to collect and the quantity and types of enemies as well as the complexity of their movement paths are good strategies.

Note that you can do this from level to level but also within the same level. The longer the player spends on a single level, the worse the prospect of losing becomes, so if you include some easy collectibles for the player to grab first and then make the remainder much harder to reach, the tension will be that much greater for the player's having spent time collecting the easy ones.

***Create new enemy types.** Create variants of existing enemies, like roaches that move faster, or create entirely new enemies, like flies with more complex movement patterns. Remember that you can edit instances of an Actor (although they'll lose their connection to the prototype) or copy a prototype to make a new one and then edit it. Introducing new enemies as the player gets further in the game is a great way to keep the game interesting and challenging.

***Implement a time limit.** This isn't reflected in the version of the game on the website, but you could give each level a mandatory time limit, or reward the player for completing the level within a certain time.

*If you're stuck, look to the game on the website for inspiration. It includes many of these suggestions as well as some extra polish not covered in the tutorial.

Your challenge is to create a three-level version of the game.

Great job! You've made Operation Green Clean!

Think about how you might improve the game further. Sound effects and music would certainly add a lot. How about many more levels? New hazards? Different types of

collectibles? Bad campers who generate trash? A dumpster for the hero to get rid of the trash he picked up? Recycling? You could combine this game with Dropcycle! (see the **Dropcycle** tutorial)

With this tutorial you've learned how to make a multi-level game in GameSalad, utilizing complex movement systems. In particular, making NPCs (non-player characters, including enemies) move in interesting and useful ways is one of the biggest challenges in game design and development. You've learned a pretty basic but entirely functional way of doing so, using timers and directional changes, but there are many other ways to do movement, including fairly sophisticated methods that let the game itself take into account the layout of the level (this is called "pathfinding"). If that's your interest, there is much more to learn.

Another important lesson is that the control system you use for the player-character affects the kind of gameplay that makes sense for your game. The click-to-set-a-location system we implemented here works well for our game but would not work well for an action or platform game that demanded precision. In the latter cases you'd want direct control. As such, be careful when creating your level layouts that they don't demand greater precision from the player than the control system can support.

Feel free to build this game further, work through the other tutorials in the curriculum, or even start your own project. The possibilities are limitless!