**Shooter - Stencyl instructions**

*A quick note: statements in this guide will be shown in screenshots, however they will not detail where they are to be found. This is to aid you in learning where certain components would be found. If you require assistance, ask your tutor.*

Begin by importing the required game file(s). This will be covered by the tutor at the beginning of the session.

Before we begin coding, we will be adding some game attributes. Navigate to “Settings” at the top of the window, then select “Attributes” on the left side of the pop-up window. From here, select “Create New”. There are 3 attributes we will be adding. The information you will be required to input for each is in the table below:

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Category** | **Type** | **Initial value** |
| isPlaying | Default | Boolean | False |
| spawnX | Default | Number | 0.0 |
| spawnY | Default | Number | 0.0 |

**Player movement (part 1a)**

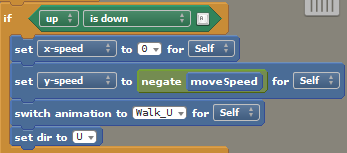
Create a new actor behaviour, named “playerBehaviour”.

Add an updating event.

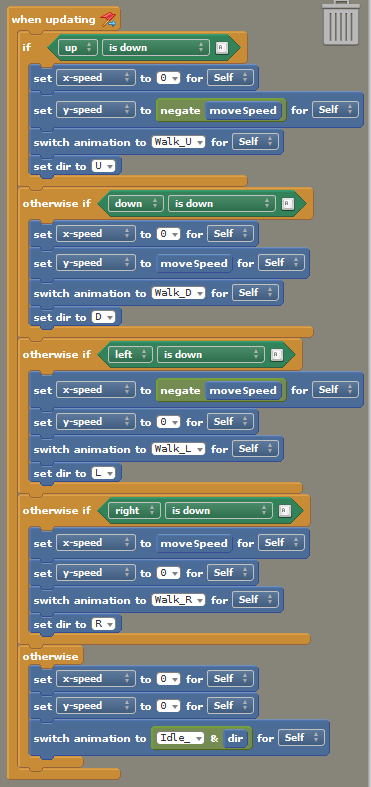
Create 4 attributes as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Hidden?** | **Default value** |
| Actor | Actor | Yes |  |
| moveSpeed | Number | No | 0.0 |
| dir | Text | Yes |  |
| isAlive | Boolean | Yes | False |

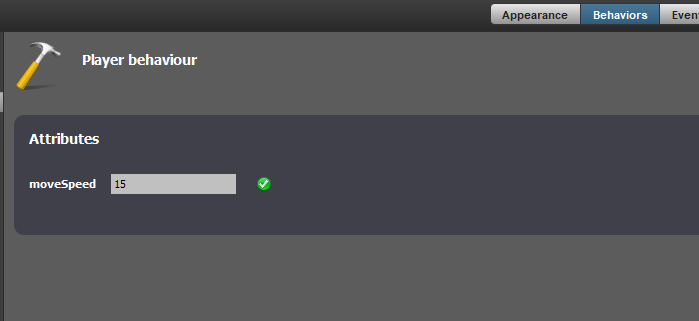
From here, we then code in the statements responsible for moving and facing our character. This is done using otherwise-if statements, which will perform an action depending on what key is being pressed. We also code the animations for our character in this step.

*An example of one block of code*

*The completed block of code for “updating”*



Now return to your actor, and under “Behaviors” attach *playerBehaviour*. You will be required to input an integer for *moveSpeed*. I used 15 as an example.



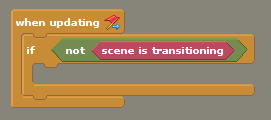
Now try testing movement in the scene. Reminder that the actor of that type must be put into the scene for it to appear. Click on “Test Scene” in the top right when viewing your scene to begin the game.



**Scene preparation (part 1b)**

Now we will code the additional scenes that the character will be placed in. First create a new scene behaviour and attach it to the scene you created at the beginning of the class. Call it “sceneManager”. Once open, add a “when updating” event.

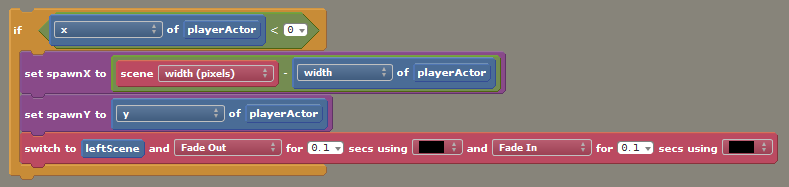
Our game will be using a transitioning effect to change from one scene to another. To make sure it does not break our game or loop infinitely, we need to make a condition that only executes the transitioning code once until it is complete, before executing again. To do this, we use another if-statement block, as shown below.



As before, create attributes for this behaviour file:

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Hidden?** | **Default value** |
| playerActor | Actor | Yes |  |
| upScene | Scene | No |  |
| downScene | Scene | No |  |
| leftScene | Scene | No |  |
| rightScene | Scene | No |  |

From here, we can code a block that will handle the transitioning of a scene to the left of the one being currently viewed.



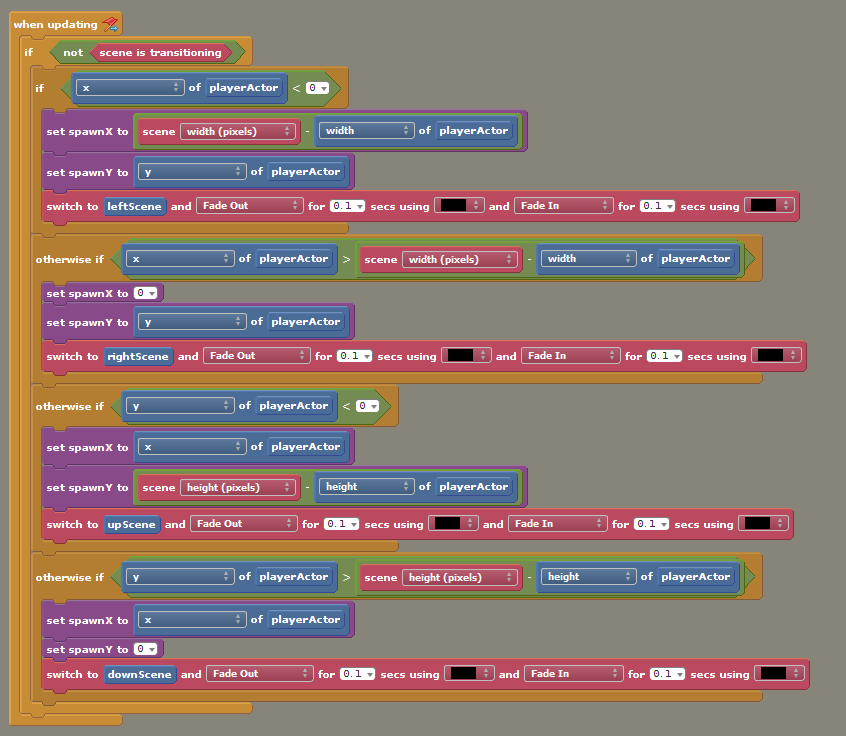
What the above block does is return the x and y values of the character each frame. When the character moves to the far left of the current scene (shown by the top code, x of playerActor < 0), the code beneath it executes (which is responsible for placing the character in the coordinates determined). It also switches the scene for us, in this case to *leftScene*, which we will then set up soon.

The reason we set *spawnX* and *spawnY* is to ensure that the character is far enough into the scene on the left to prevent them from accidentally triggering a change back to the original scene (which would be to the far right of the new scene).

Additional attributes such as fading effects and time taken are for you to decide.

We will now repeat the above for the other directions. Scenes above and below are optional.

*The complete code block*



If you only want a world that travels left-to-right, ignore the bottom 2 otherwise-if blocks, as they deal with the player’s y-position.

We will now create another 2 scenes to place alongside our current one. Try to make them in a way that allows objects such as ‘paths’ or ‘terrain’ to intersect accordingly. Below is an image showing my 6 scenes placed in a way that shows my entire world.

*My scenes*



*My game “world”*

**

If you have opted for a left-to-right format for your game world, the same concept still applies.

Now create another scene, this time calling it “blankScene”. Make it, well, blank.

This new scene will serve us in the coming steps.

Now we will attach the *sceneManager* behaviour to our scenes, and define (for each scene) the scene to the left, and to the right of the one we are currently viewing. (If applicable, we will also be attaching those above and below too).

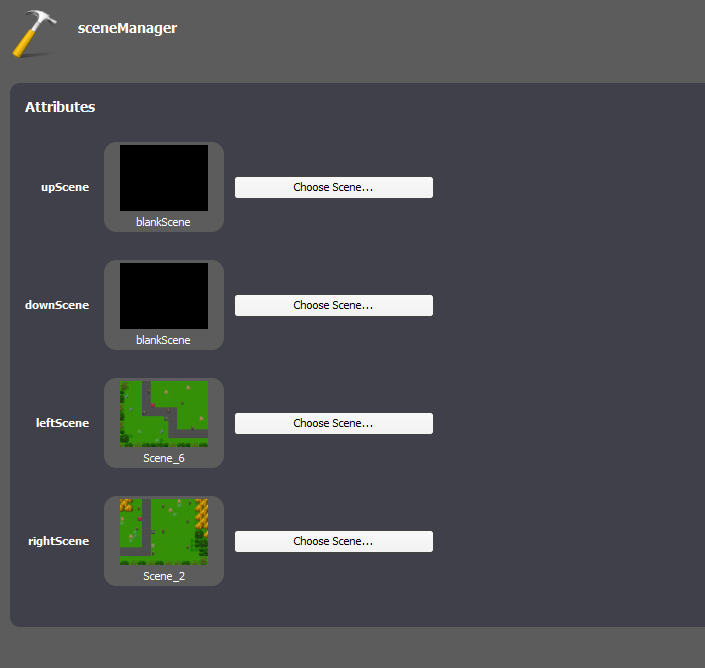
Select your first scene and attach the *sceneManager* behaviour to it. You will now be required to select the scene to the left, right, above and below of the one you have selected. This is entirely dependent on the way you have made your scenes. Also note that if your game is left-to-right, all of your scenes will have *blankScene* for above and below. I will include my inputs below as reference.

*Example 1*

I will begin with Scene\_1.



When I add the *sceneManager* behaviour to scene 1 (under the “Behaviors” tab on Scene\_1), I am given the below (which I have already filled out, yours will be blank):



*upScene* represents the scene above my current one.

*downScene* the one below.

*leftScene* and *rightScene* the same thing, both scenes either side of the current one (in this case, Scene\_1).

If I remember the layout of my world, I know that Scene\_6 is to the left of Scene\_1, and Scene\_2 is to the right. I decided that I wouldn’t join Scene\_1 to Scene\_4 above, as I have obstacles in the way. Plus, I want the player to explore around the world. Despite this, I cannot leave any values empty (or *null*), so I instead add *blankScene*. I do the same for below.

I then repeat the steps above for the remaining scenes. Those of you with left-to-right worlds will have an easier time doing these steps.

*<End of example>*

Remember that for each new scene you create, you have to put an actor of your player character into the scene. Our code doesn’t actually CREATE the entity, rather it repositions an existing one when we enter another part of our world.

**Enemies (part 2)**

We will now be adding NPCs (Non-Player Characters).

These little dudes will wander around your world aimlessly.

Begin by creating or selecting an actor to represent your NPCs. Now create a new behaviour. Call it “npcWander”.

Create the following attributes:

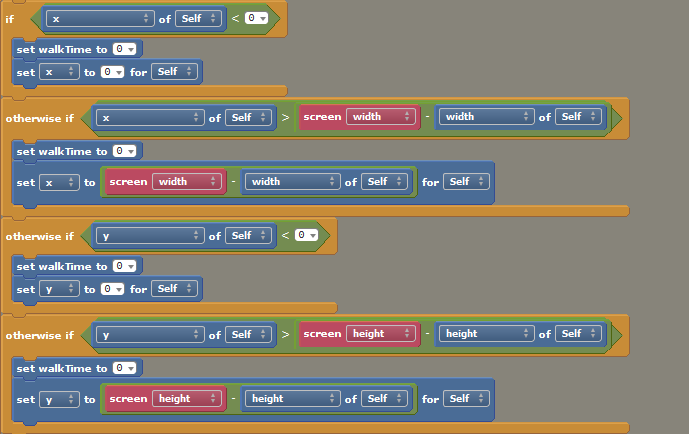
|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Hidden?** | **Default value** |
| Actor | Actor | Yes |  |
| walkTime | Number | Yes | 0.0 |
| dir | Text | Yes |  |
| walkSpeed | Number | No | 0.0 |
| randomNumber | Number | Yes | 0.0 |
| newDir | Text | Yes |  |
| waitTime | Number | Yes | 0.0 |

Now add a “when created” event, and insert the following:



Don’t worry about *generateDir* yet.

Add another event, this time “when updating”. We will be using code similar to the player’s character to determine the direction the actor should be facing. Before we do this, let’s ensure that our NPCs don’t walk outside of the map

Clip the above into the *when updating* segment, and move on.

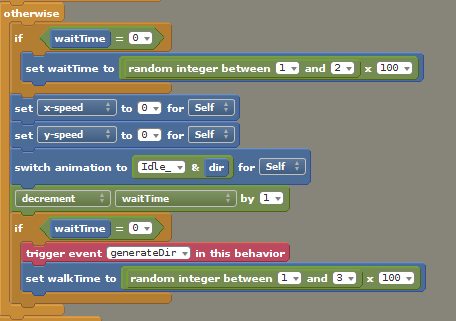
Now we determine the direction that our NPCs will move.



We only want this to happen when *walkTime* is a value other than zero. To do this, we place the code into an if-statement, and add a statement that decreases *walkTime* by 1 every time the NPC moves.

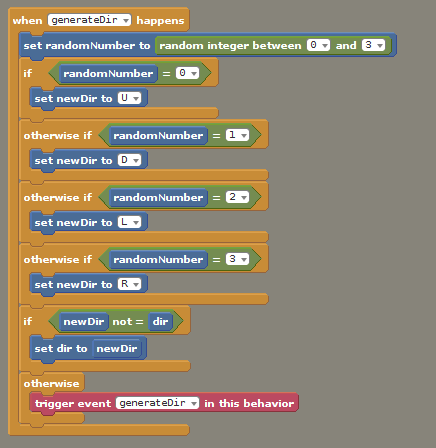


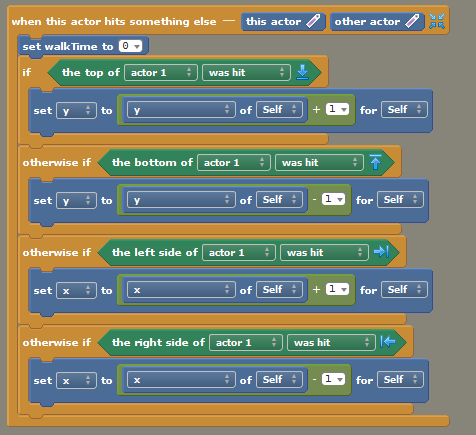
Now we will create a block that generates random numbers for the above code, so the movement and direction of the NPCs is randomised. We will put these blocks in an otherwise statement, so that it executes when *walkTime* is zero.





Your completed block should look like this:

Next, we will add a custom event (Add Event > Advanced > Custom Event) to define “generateDir” for the NPC behaviour.



Finally, we will add collision to the NPCs, so that they do not phase through items.

Go to Add Event > Collisions > Something Else, and complete the following:

**Bullets (part 3a)**

We will now be adding a basic ability for our character to fire projectiles at NPCs. First, create or assign an actor to be a bullet, then create a new actor behaviour called “fireBullet”.

Create the following attributes:

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Type** | **Hidden?** | **Default value** |
| Actor | Actor | Yes |  |
| fire | Boolean | Yes | False |
| fireDir | Number | Yes | 0.0 |
| bulletAlive | Number | Yes | 0.0 |
| wait | Boolean | Yes | False |
| useMouse | Boolean | No | True |
| useControls | Boolean | No | True |
| fireControl | Control | No |  |
| bulletType | Actor Type | No |  |
| bulletSpeed | Number | No | 50.0 |
| dirMode | Text | No |  |
| dir | Number | No | 0.0 |
| upAnim | Text | No |  |
| downAnim | Text | No |  |
| leftAnim | Text | No |  |
| rightAnim | Text | No |  |
| offset | Number | No | 0.0 |
| rateOfFire | Number | No | 3.0 |
| bulletAliveLimit | Boolean | No | False |
| bulletAliveMax | Number | No | 1.0 |
| useAmun | Boolean | No | False |
| maxAmun | Number | No | 5.0 |
| currentAmun | Number | No | 5.0 |

When creating *dirMode*, we will require a dropdown menu. To do this, select the “Attributes” tab at the bottom of the window, select *dirMode* from the list and select “Dropdown”. From here, add:

Relative=Relative

Absolute=Absolute

Based on Animations=Based on Animations

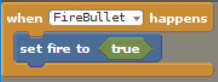
Alternatively, for this specific exercise, you can simply make the default value of *dirMode* the following, as it appears:

Based on Animations

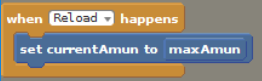
**Creating the behaviour (part 3b)**

Firstly, begin with Add Event > Advanced > Custom Event.

Construct the block as below:

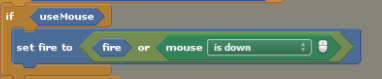


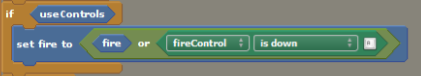
Create another custom event, and add the following:

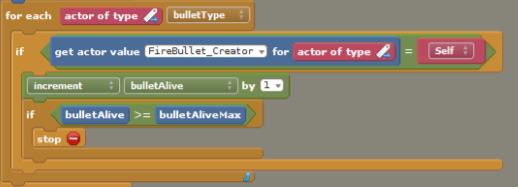


Now we can add a new “when updating event”.

Below is an explanation of the main elements of the code block, followed by the completed code block.

*Used to fire the bullet on a mouse press.*

*Another command for firing, this time when fireControl is triggered.*

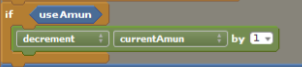
*Counts the number of bullet entities and limits them, if the user desires.*

*Defines the rate of fire by comparing the bullets on the screen to the maximum number allowed at any one time.*

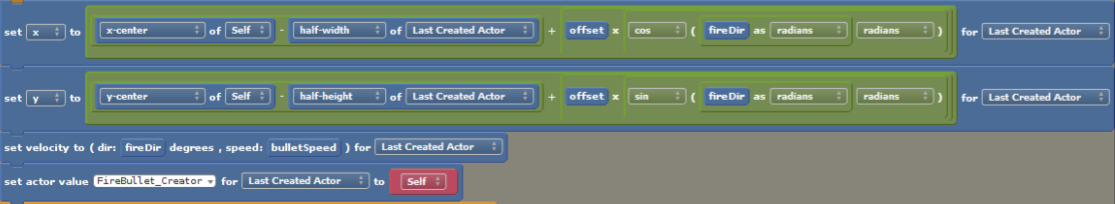


*This block first determines if the user has defined for bullets to be fired in a direction based on actor animation, or if it is to be either absolute or relative. The code uses user-defined parameters for determining what direction bullets go based on the orientation of the character.*

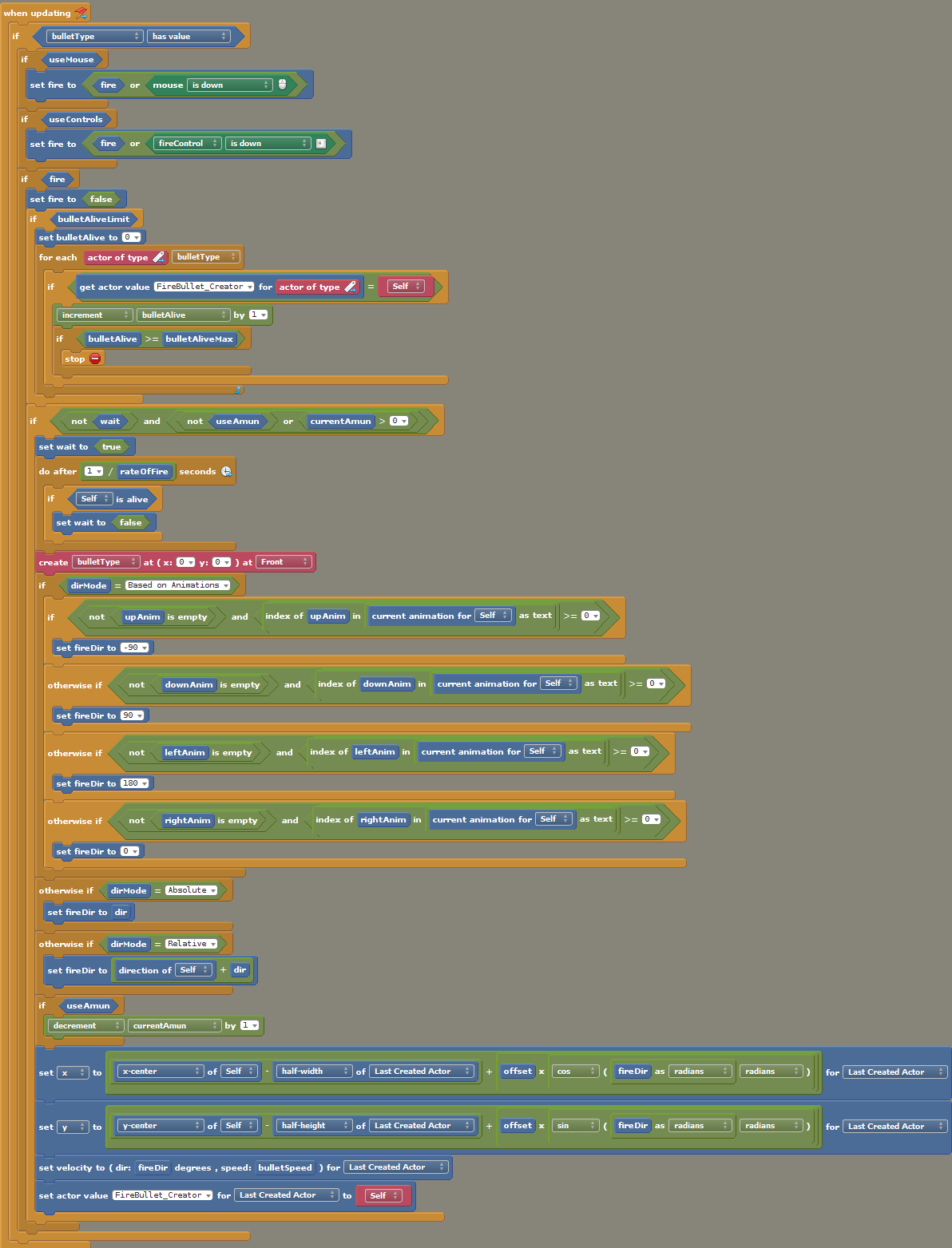
*If-statement that reduces the value of ‘currentAmun’ by 1 for each fire bullet, if ‘useAmun’ returns true.*



*Render all information collected previously (namely the orientation of the bullets) and processes the act of firing the actor ‘bullet’ across the screen.*



With some additional statements, the final block should then look like the following:



To apply our code to our playable character, we will then attach it to the respective actor. Return to Stencyl and select your character from the Actor section, then under “Behavior” add *fireBullet*. Then, fill out the respective fields to ensure the code runs correctly. Specifically, you will need to at least select a fire method (mouse or fireControl), the bullet actor and the dirMode (covered more next).

For *dirMode*, we will make them based on animations. Select the option from the dropdown menu (or if you chose the alternative method, skip this step).

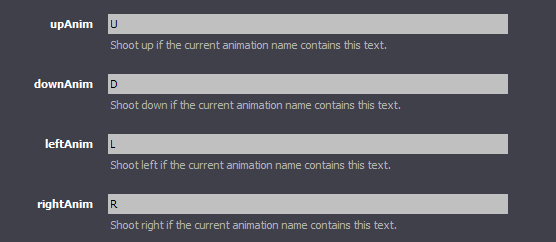
Now we will fill in the fields that tell *fireBullet* which animations to look out for. When we created our character actor in part 1, we simply used letters to represent the direction that our character was facing. Hence, we will use the same terminology here so that *fireBullet* can find it.

For *upAnim*, input “U”.

For *downAnim*, input “D”.

For *leftAnim*, input “L”.

For *rightAnim*, input “R”.



The remaining values are for you to decide.

Now feel free to try it out!

**Additional exercises**

1. Add additional scenes to world
2. Add an ammo counter to your game
3. Add descriptions to your major attributes, to make them easier to understand
4. Create a scene that displays to the user when their character has been killed