

Little Town

A GameMaker Studio 2™ tutorial

GameMaker Studio 2™ is a trademark of YoYo Games Ltd. © YoYo Games Ltd. 2021

Table of Contents

1	About this tutorial.....	7
2	How to use this tutorial.....	8
2.1	Setup and included components	8
2.2	Downloading the art assets.....	9
2.3	Downloading the AudioHero sound assets.....	9
3	Topics covered.....	11
4	Session 1	12
4.1	Getting started	12
4.2	Making our first Object	13
4.3	About Sprite Strips.....	16
4.3.1	Converting a Sprite Strip manually	17
4.4	Changing a Sprite's speed and origin.....	18
4.5	Applying a Sprite to an Object.....	20
4.6	Setting up a Room	20
4.7	Running the game.....	22
4.8	Making things happen with Events.....	23
4.9	Writing our first GameMaker Language (GML) code	25
4.10	Using keyboard Events to move right.....	26
4.11	Controlling our player with a Step Event	26
4.12	Moving our player Object left	29
4.13	Adding vertical movement.....	30
4.14	Changing all our movement Events to code.....	32
4.15	Changing our player Sprites.....	34
4.16	Preparing Tile Sets	37
4.17	Designing our town	39
4.18	Placing Tiles	41
4.19	A Room with a view	42
4.20	Adding Objects to our town	45
4.21	Simple depth sorting	47
4.22	Object parenting for easy editing.....	47

4.23	End of Session 1.....	49
5	Session 2	50
5.1	Creating solid Objects with collisions	50
5.2	Collision masks.....	51
5.3	Creating the other environmental details.....	53
5.3.1	A note about obj_fountain01	57
5.4	Filling out the town.....	57
5.4.1	Scaling and flipping placed Instances	59
5.4.2	Adjusting Tiles	60
5.5	Keeping the player in bounds.....	63
5.6	Creating the Baker character	66
5.7	Editing Sprite animations	67
5.8	Thinking ahead with parent Objects.....	68
5.9	Variable Definitions	70
5.10	Controlling animation with Alarms.....	71
5.11	Our first Alarm.....	72
5.11.1	A quick note on steps and game speed	72
5.11.2	Back to our Alarm	73
5.12	Using those Variable Definitions	74
5.13	Populate the town	76
5.14	Detecting the NPCs	81
5.15	Using debug messages	84
5.16	Updating our NPC Collision Masks	85
5.16.1	A note on collision Events	87
5.17	Sound and music	89
5.18	Using control Objects.....	91
5.19	Say hello with sound	95
5.20	Using variables to maintain control	97
5.21	3D positional audio	98
5.22	Moving the Audio Listener with the player.....	99
5.23	Adding an Audio Emitter	100
5.24	End of Session 2.....	104
6	Session 3	105

6.1	Setting up our textbox	105
6.2	Drawing things manually	106
6.3	Working with fonts.....	108
6.4	A more advanced textbox	111
6.5	Calling a textbox when we need one.....	113
6.6	Talking to our NPCs.....	115
6.7	Giving our NPCs a voice of their own.....	116
6.8	Taking control away from the player	122
6.9	Closing our textbox	123
6.10	Adding effects to our textbox.....	125
6.11	Adding a sound effect to the textbox	129
6.12	Adding a visual prompt	129
6.13	Creating our own functions within Script Assets.....	131
6.14	Writing a custom function.....	133
6.14.1	How functions work.....	133
6.15	Returning the value of a function.....	134
6.16	Using the scr_showPrompt function.....	134
6.17	Adding effects to the prompt icon.....	137
6.18	Making the prompt go away	141
6.19	Bonus: animating the prompt	143
6.20	End of Session 3.....	144
7	Session 4	146
7.1	Making a basic item Object.....	146
7.2	Create our first item	146
7.3	Recognizing the items.....	149
7.4	States and enums.....	153
7.5	Setting up our player states	154
7.6	Creating a simple array	157
7.7	Using an array to change Sprites	161
7.8	Applying states to items	161
7.9	Picking up an item.....	162
7.10	Returning results from functions.....	169
7.11	Putting the item back down	174

7.12	Limiting when an item prompt can show	178
7.13	Affecting the player's speed	178
7.14	Creating the other items.....	179
7.15	Adding the ability to run	182
7.16	Adding a dust cloud effect.....	184
7.17	Making the dust cloud effect work.....	186
7.18	End of Session 4.....	189
8	Session 5	190
8.1	Getting our Sequence assets ready	190
8.2	Preparing our new Baker assets	193
8.3	Starting our first Sequence.....	195
8.4	Setting up the Sequence Canvas.....	196
8.5	Adding our first track	197
8.6	Turning off automatic keyframes	198
8.7	Editing the first track	199
8.8	Adding parameter tracks and keyframes.....	200
8.9	Adding a background to our Sequence	203
8.10	Extending our Sequence.....	206
8.11	Bringing in our Baker	206
8.12	Making our Baker dance	211
8.13	Completing the Baker Sequence.....	216
8.14	Copying and pasting elements for quick editing	218
8.15	Adding music to our Sequence	221
8.16	Broadcast Messages	222
8.17	Creating our "sad Baker" Sequence.....	223
8.18	Create the Sequences for the Teacher	225
8.19	Create the Sequences for the Grocer.....	227
8.20	Preparing to control a Sequence.....	230
8.21	Expanding the control object with Sequences	230
8.22	Testing our Sequence.....	232
8.23	Using Broadcast Messages to listen to Sequences.....	234
8.24	Using states to control Sequences.....	237
8.25	Controlling town audio when Sequences are playing	239

8.26	Adjusting 3D positional audio when a Sequence is playing	242
8.27	End of Session notes	245
9	Session 6	246
9.1	Adding our core gameplay loop.....	246
9.2	Changing what the NPCs say	246
9.2.1	Updating the Baker	248
9.2.2	Updating the Teacher	249
9.2.3	Updating the Grocer.....	250
9.3	Checking for which item the Player has	251
9.4	Playing a Sequence via the Textbox Object.....	254
9.5	Telling the Textbox which Sequence to play	256
9.6	Aligning Sequence emitters	258
9.7	Removing “correct” Objects once given	259
9.8	Setting up NPC states.....	260
9.9	Marking an NPC as “done”.....	261
9.10	Switching our NPCs state	263
9.11	Setting the NPCs “done” Sprites	264
9.12	Changing interaction when an NPC is “done”.....	267
9.13	Removing the “S” key test Event.....	268
9.14	Designing a “Game Over” Sequence	269
9.15	Setting up the “Game Over” Sequence	272
9.16	Creating a function to play Sequences.....	275
9.17	Playing the actual “Game Over” Sequence	278
9.18	Using Moments to call function in Sequences	280
9.19	Creating a title screen	281
9.20	Setting the Room Order	284
9.21	Making a title logo.....	287
9.22	Creating a prompt Sequence	287
9.23	Adding control to the title screen	292
9.24	End of session notes.....	297
10	Bonus Session	298
10.1	Bonus 1: Add shadows to the player	298
10.2	Challenge 1: add a shadow to the NPCs.....	300

10.3	Bonus: add pick-up and put-down animations	301
10.4	Challenge 2: redesign the prompt and textbox Sprites	304
10.5	Challenge 3: design and add more items to the game	304
10.6	Challenge 4: make the game more challenging	305
10.7	Export your game	305
10.8	End of Session.....	305

1 About this tutorial

Little Town is a tutorial meant to teach the core functionality of GameMaker Studio 2 in a natural, fun way. It has three key goals:

1. Teach GameMaker Studio 2 fundamentals to give a great foundation from which to build awesome games
2. Encourage self-driven learning and experimentation
3. Allow for both synchronous and asynchronous participation for in-class and remote learning

This is ideal for starting out with game development courses, or learning about game art. It covers several major features of GameMaker Studio 2 in a holistic way, and is intended to run you through common workflows to prepare them to create their own games.

You do not need to know programming beforehand, but programming is a core component of this tutorial. The sessions contained here teach GameMaker Studio 2's own language (GML) alongside its more visually-driven features.

In *Little Town*, players control a child in an idyllic small town. They must give the town's three characters — the Baker, Teacher and Grocer — the item each of them most wants.

As a player, you must find the correct items and bring them to the right person. As a developer, you can choose where to hide each item and write their own hints and dialogue. They are also encouraged (see the Bonus Session) to provide their own art and music assets afterward to transform *Little Town* into their own creation.

2 How to use this tutorial

The contents of *Little Town* are organized into six Sessions and broken down into multiple *sections* per Session. Each section in this document is labeled and numbered (e.g. “4.6 Setting up a Room”) for easy reference.

The video versions of this tutorial are labeled in accordance with the section numbers in this document; **the contents of this document and the videos are the same.**

2.1 Setup and included components

This tutorial comes with the following components:

- 137 tutorial videos (organized into six Sessions, labeled by section)
- YoYo Games art assets package
- AudioHero sound assets package
- Example final GameMaker Studio 2 project file (.YYZ) with annotated code
- In-progress GameMaker Studio 2 project files (.YYZ) for sessions 2–6

Prior to beginning this course, the instructor should:

- Familiarize themselves with the course content
- Organize the necessary art assets (see [Downloading the art assets](#), below) for delivery to the you in a folder called Sprites
- Provide the AudioHero redemption instructions (see [Downloading the AudioHero sound assets](#), below)
- Make the tutorial videos as necessary (e.g. on a Session-by-Session basis)

Prior to beginning this course, you should:

- Download the newest version of GameMaker Studio 2 and install it
- Open GameMaker Studio 2 at least once to sign in with their YoYo account and perform any required updates
- Create a project folder on their hard drive (e.g. “Little Town Tutorial”)
- Download and unzip the provided art assets package into the project folder (see [Downloading the art assets](#), below)
- Sign up for an AudioHero account and download the sound assets (see [Download the AudioHero sound assets](#), below)

2.2 Downloading the art assets

You will need to be provided the art assets that came with this document by the instructor alongside the video tutorials. The assets come as a folder called Sprites, and include four folders filled with image files:

-  Sprites
 -  Backgrounds
 -  Characters and Items
 -  Sequences
 -  User Interface

The included art assets

2.3 Downloading the AudioHero sound assets

You will also need to download the required sound assets from AudioHero.com for this tutorial.

Redemption and signup link:

<https://yoyogames.audiohero.com/signup.php?forward=https%3A%2F%2Fyoyogames.audiohero.com>

1. You needs to click the redemption link above and fill out the registration (name, email, and password)
 - a. If they already have an Audio Hero account, they can click “Log In” and log in instead)
2. Click “Sign Up,” and then you will be requested by email to confirm their account
3. Then, once they have confirmed their account by email, they should click on the redemption link again and log in with their details
4. they will then have full access to a list of sound assets for this tutorial, which they can download for free
5. You should download all supplied sound assets (15 in total) to their hard drive
 - a. If asked to choose a file format, you should choose “Download WAV”
 - b. They should create a folder called Sounds and place all sound assets in this folder
 - c. They should move the Sounds folder into the same location on their hard drive as the Sprites folder (as described in [Downloading the art assets](#))

📁 Sounds

- 🔊 snd_fountain
- 🔊 snd_gameOver
- 🔊 snd_greeting01
- 🔊 snd_itemPickup
- 🔊 snd_itemPutDown
- 🔊 snd_pop01
- 🔊 snd_pop02
- 🔊 snd_seq_bad01_BGM
- 🔊 snd_seq_bad02_BGM
- 🔊 snd_seq_bad03_BGM
- 🔊 snd_seq_good01_BGM
- 🔊 snd_seq_good02_BGM
- 🔊 snd_seq_good03_BGM
- 🔊 snd_townAmbience
- 🎵 snd_townBGM

The complete sound assets after being downloaded from AudioHero

If you need to re-download sound assets after creating an account, they can do so by going to
<https://yoyogames.audiohero.com/>

3 Topics covered

Little Town is intended to walk you through the most commonly used functions of GameMaker Studio 2 in a practical way, simulating a real, professional project. Please note that this tutorial uses GameMaker Studio 2's GML project type, so you will learn to code. Throughout the tutorial you will learn the following features:

- **General features:**
 - GameMaker Studio 2's Integrated Development Environment (IDE) and user interface (UI)
 - Creating and editing Objects, Sprites and Sounds
 - Coding player movement
 - Using Alarms to time actions
 - Using Draw and Draw GUI for animating and displaying graphics
 - Creating and editing Rooms
 - Designing levels with Tiles, Sprites and Objects
 - Creating and editing Cameras and Viewports
 - Playing sound effects and music
 - Creating 3D audio with Emitters and Listeners
 - Tracking collision between Objects
- **Coding (GML) specifics:**
 - Variables, global variables and booleans
 - Simple arrays for storing data
 - For loops, switch statements and with statements
 - Functions and Script Assets
 - States and enums
- **Sequences**
 - UI and functionality overview
 - Creative use of the timeline
 - Using Broadcast Messages to communicate with other assets in the game
 - Using a control Object to listen to and affect Sequences

4 Session 1

Welcome to the course! In this session, we're going to go through the basics of the GameMaker Studio 2 interface and jump right into making our game.

Little Town is a simple 2D game set in a quaint small town that you get to design. The player can find and bring items to the three characters in our town and see those characters' humorous reactions as a result.

Some general terms we may use:

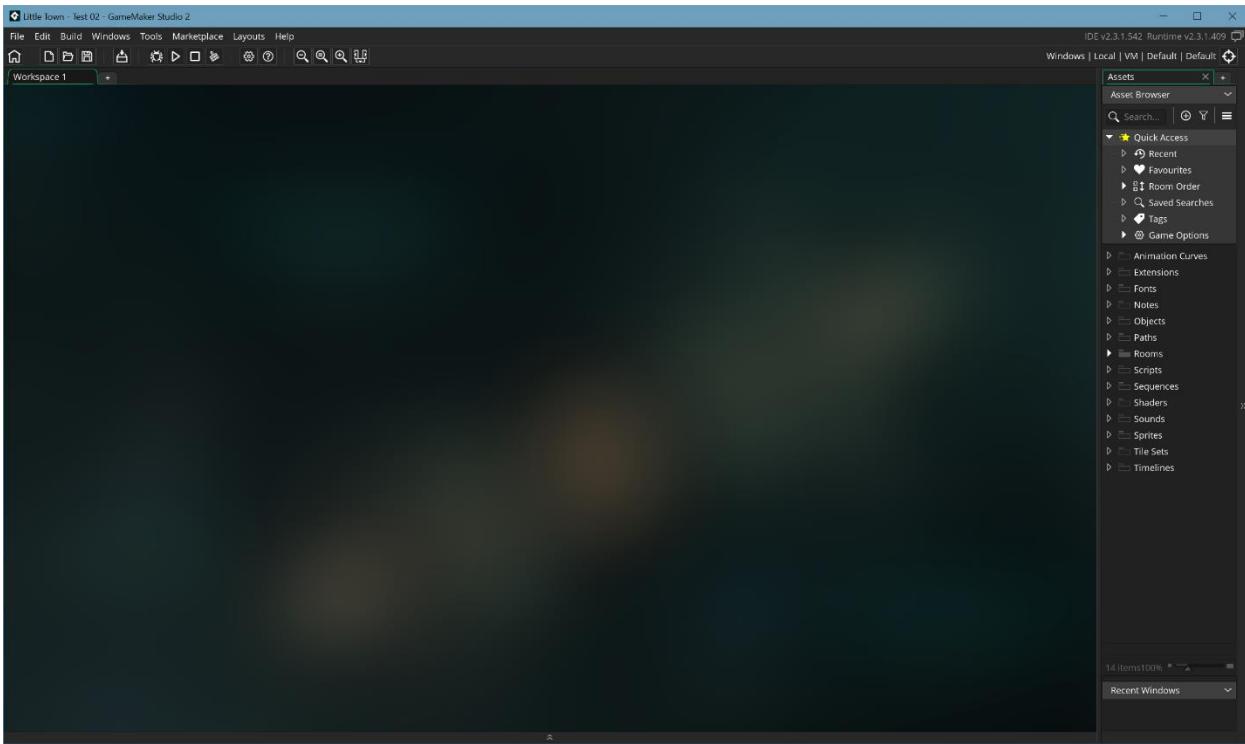
- NPC: a “non-player character” is just that; a character in our game that isn’t controllable by the player. In *Little Town*, this is the Baker, Teacher and Grocer
- IDE: the “integrated development environment,” or GameMaker Studio 2 interface
- GML: “GameMaker Language,” the programming language used in GameMaker Studio 2
- Sprite: a two-dimensional image or graphic used in a game. This could be a static image of a tree, or a series of animated frames to show the player is walking

4.1 Getting started

With GameMaker Studio 2 open, you'll see the welcome screen. Click on “New Project” and click “GameMaker Language” to create our project. You'll be asked to save your project right away, so navigate to your project folder and name this *Little Town Tutorial*.

Click “Save,” and just like that, you’re in the main GameMaker Studio 2 interface, also known as the *Integrated Development Interface*, or *IDE*. Here’s where the magic happens.

That big area you see is called the *Workspace*; it’s like a big desk in which you’ll view assets you’re working on. That big column on the right is your *Asset Browser*; this is where you’ll organize all your Sprites, Sounds, Objects and other assets.



The GameMaker Studio 2 Integrated Development Environment (IDE). The blank area is the Workspace and the list on the right is the Asset Browser.

You may see by default that your Asset Browser has several folder icons with labels. These folders are called *Groups*, and they exist to help you organize commonly used assets (Sprites, Sounds, Objects, etc.). You can also add new groups as you need them or remove ones you don't (but we'll experience this later on).

4.2 Making our first Object

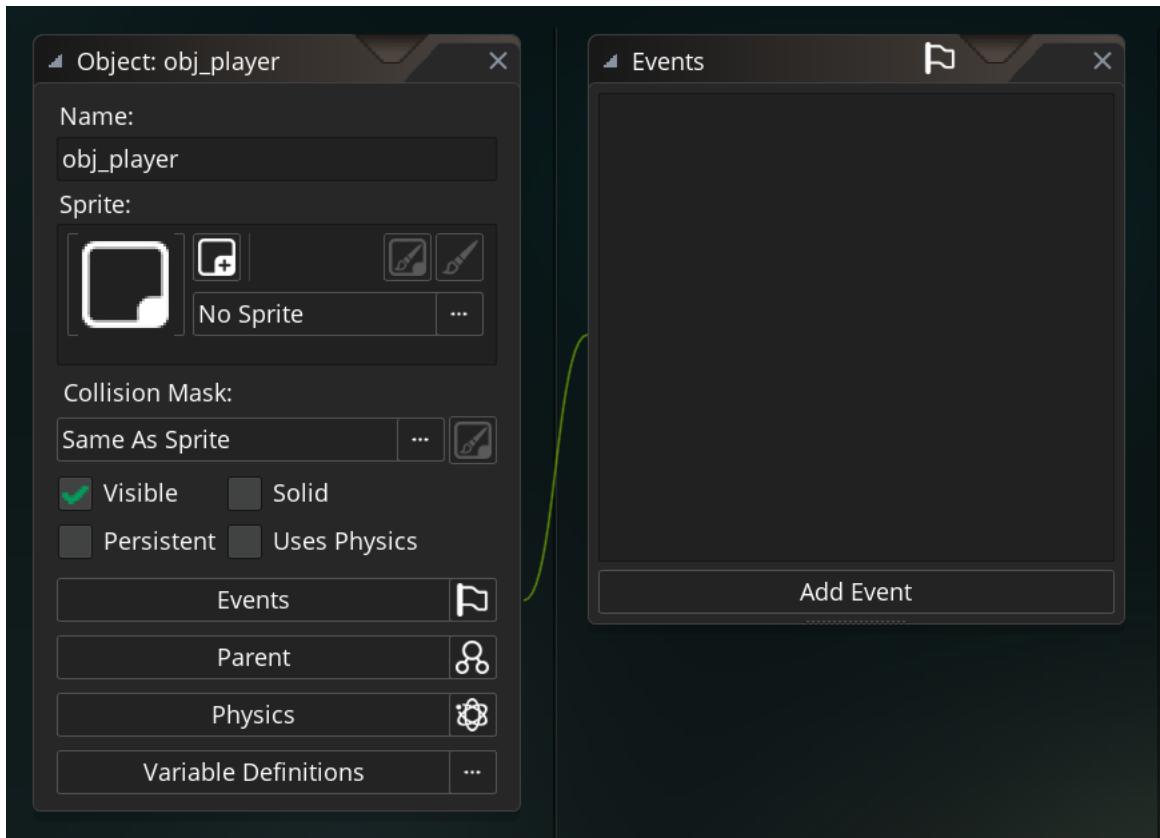
Objects are the heart of GameMaker Studio 2. They can be players, enemies, items, obstacles, but they can also be health bars, textboxes, title screens and even invisible stuff that keeps track of our game behind the scenes. In short, if something must *do* something, it's probably going to be an Object.

So, let's make our first!

Right-click on the Objects group in the Asset Browser and choose Create > Object. (If you don't have an Objects group, you can right-click in the Asset Browser and choose Create Group.)

Let's name this `obj_player`. Keep in mind that asset names can't have spaces or special characters in them; for our project, we'll be using this naming convention (*asset prefix – underscore – name*) for all our assets to stay organized.

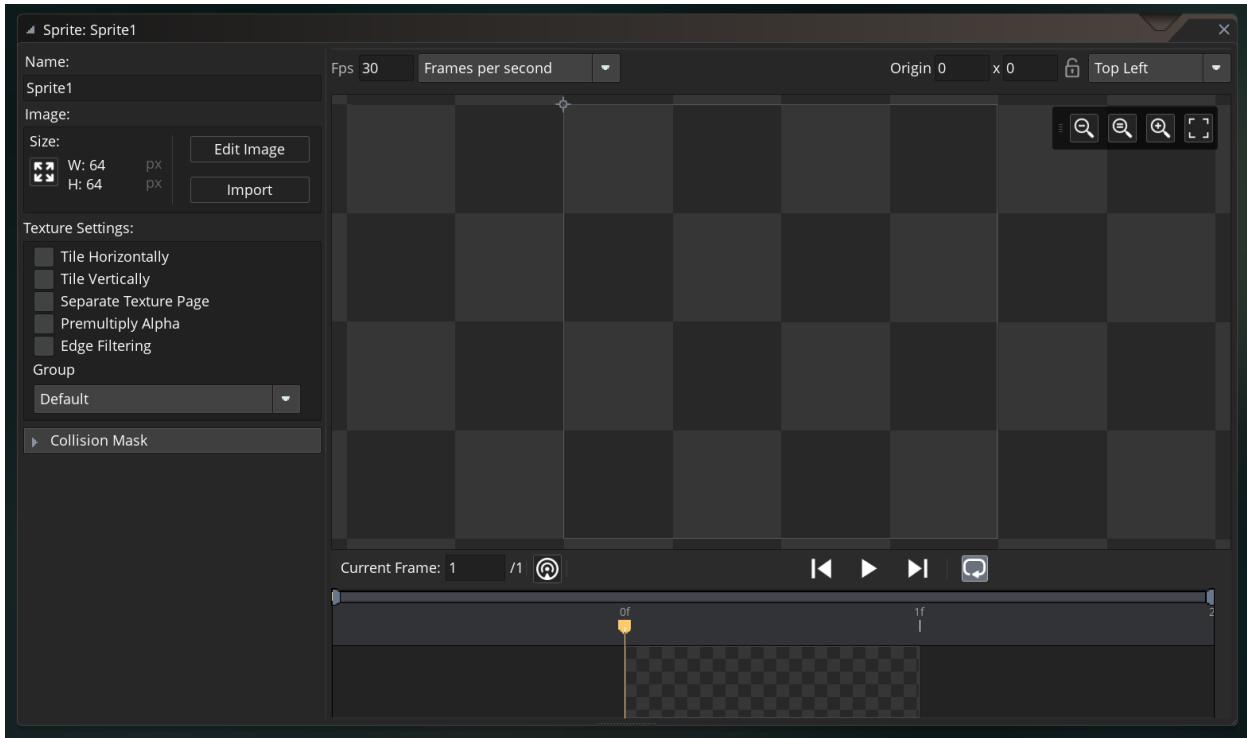
You'll see our new player Object has appeared in the Workspace and now we can edit it. The interface we see when we open an Object is called the *Object Editor*.



Our new `obj_player` Object in the Object Editor

In order to continue, let's get some Sprite assets into our project. A *Sprite* is a two-dimensional image that represents a character, enemy, item and so on. They can be static, or they can be animated.

So, right-click on the Sprites group in the Asset Browser and choose Create > Sprite. You'll create a new Sprite, which will open automatically in the *Sprite Editor*.

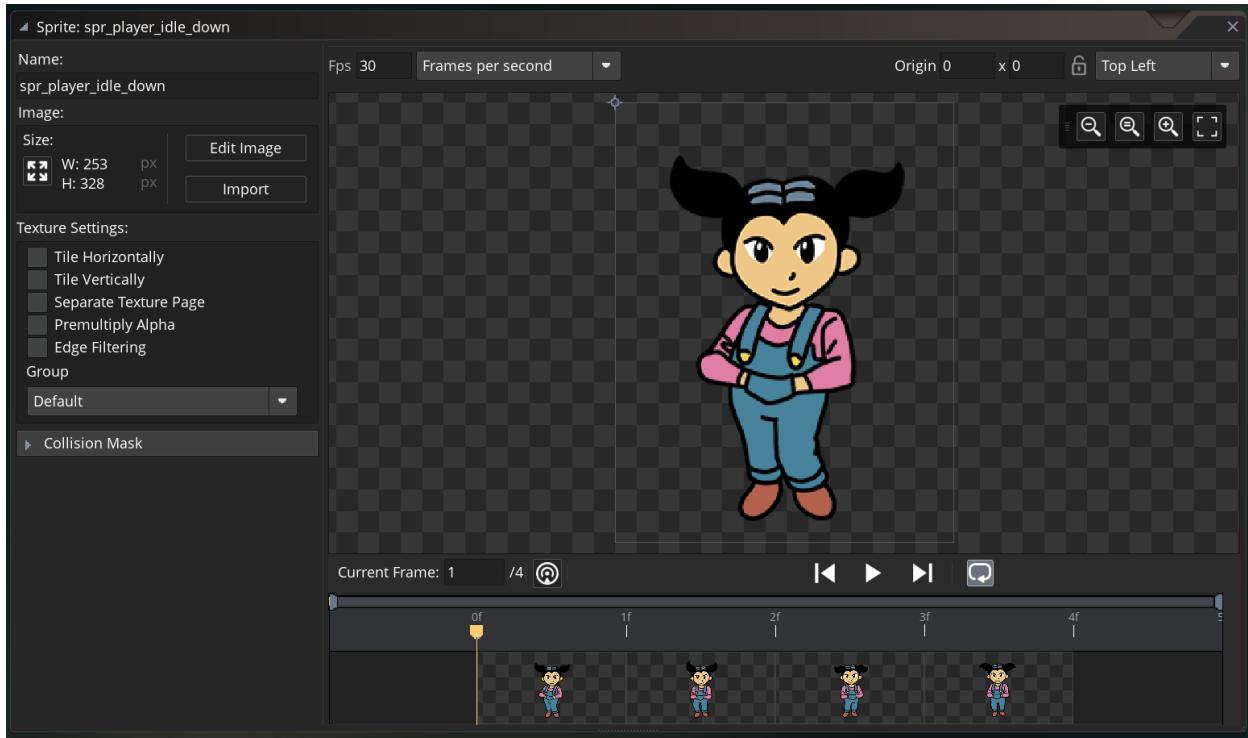


Our brand-new Sprite in the Sprite Editor

Click in the top-left where it says “Name:” and rename this Sprite `spr_player_idle_down`.

Next, click the Import button. In the dialogue box that pops up, navigate to the Assets folder you downloaded for this course. Open the Sprites folder and then the Characters and Items folder.

Choose the image called `spr_player_idle_down_strip04` and click “Open” to select that Sprite. What you’ll likely see is this:



The first player Sprite, converted to from a Sprite Strip to animation

The image we've imported is called a *Sprite Strip*. It contains several frames of animation in a single image file. And due to some intelligence on GameMaker Studio 2's behalf, that image file should have been automatically converted into an animation!

4.3 About Sprite Strips

So how does GameMaker Studio 2 know to convert the image we just imported into separate frames of animation? Well, it uses two criteria:

- Does the image file have a `_stripXX` suffix? (where XX is a number)
- Can the pixel width of the image file be evenly divided by whatever number is in the `_stripXX` suffix?

For example, the image we just imported is 1012 pixels wide, and the file name says `_strip04`. GameMakerStudio 2 checks to see if $1012 \div 4 =$ a whole number (which it does: 253).

4.3.1 Converting a Sprite Strip manually

If an image you've imported hasn't converted to frames of animation correctly, here's what you can do to fix that.

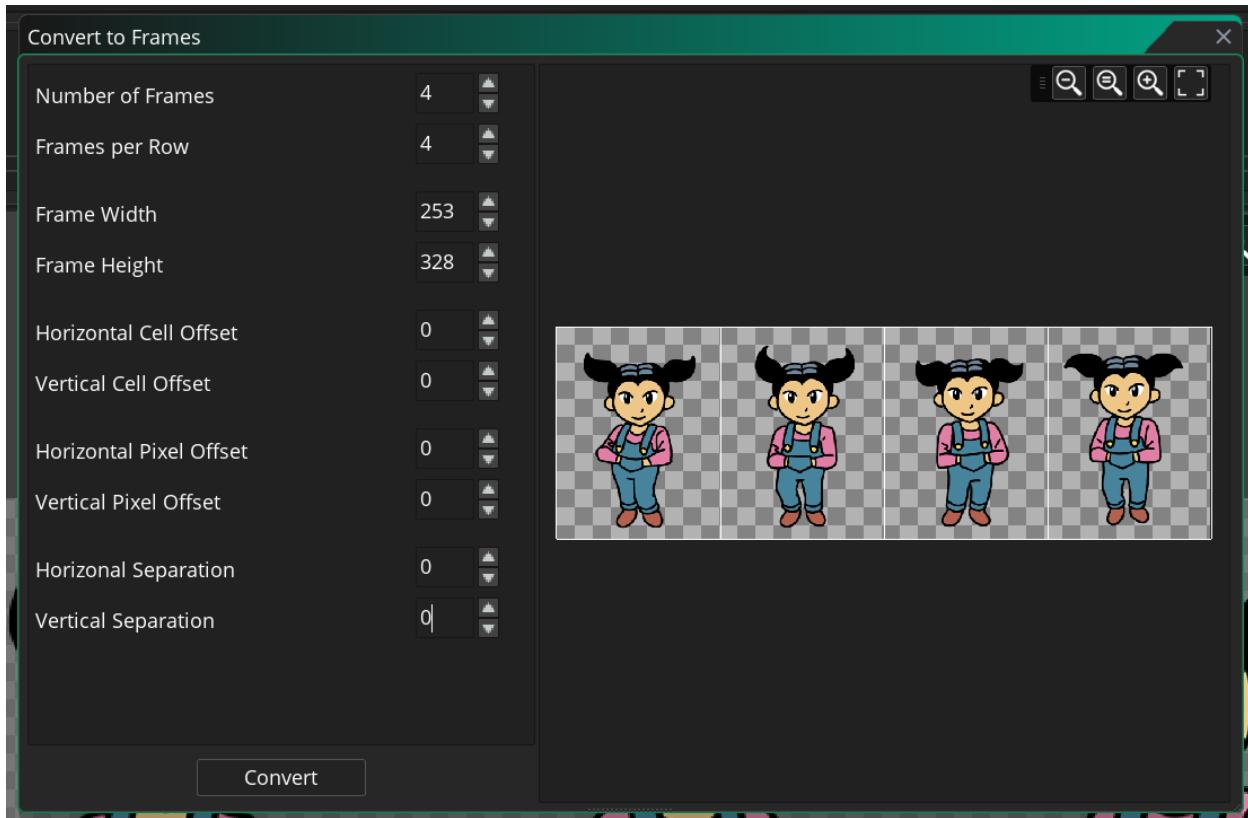
	<p>Tip: If the Sprite Editor displays that you have four frames in your Sprite already, you can skip ahead to Changing a Sprite's speed and origin.</p>
---	--

In the Sprite Editor, click the `Edit Image` button. You'll be taken to a large version of your Sprite with several drawing and selection tools.

Note that doing this has opened a new tab at the top of the IDE. You can click the X on this tab to close this at any time, or you can click on the "Workspace" tab to return to where we were.

In the menu bar at the top of the IDE, click `Image > Convert to Frames`. This new window will allow us to "slice" our Sprite into individual frames.

To do so, change the values like so:



The Convert to Frames window, with the correct values entered

- Number of Frames: 4
- Frames per Row: 4
- Frame Width: 253
- Frame Height: 328

Click Convert to close this window. If you see a dialogue that warns that you are about to commit changes, click OK. You'll see that our player Sprite has now been split into four frames.

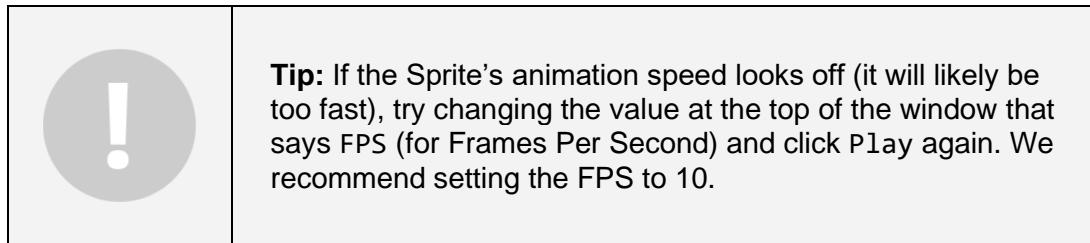
Click the X on the spr_player_idle_down tab at the top of the IDE to close this Sprite.

4.4 Changing a Sprite's speed and origin

Look at the spr_player_idle_down sprite again in your Workspace. If you don't see it, find this Sprite in your Asset Browser and double-click it to open it again.

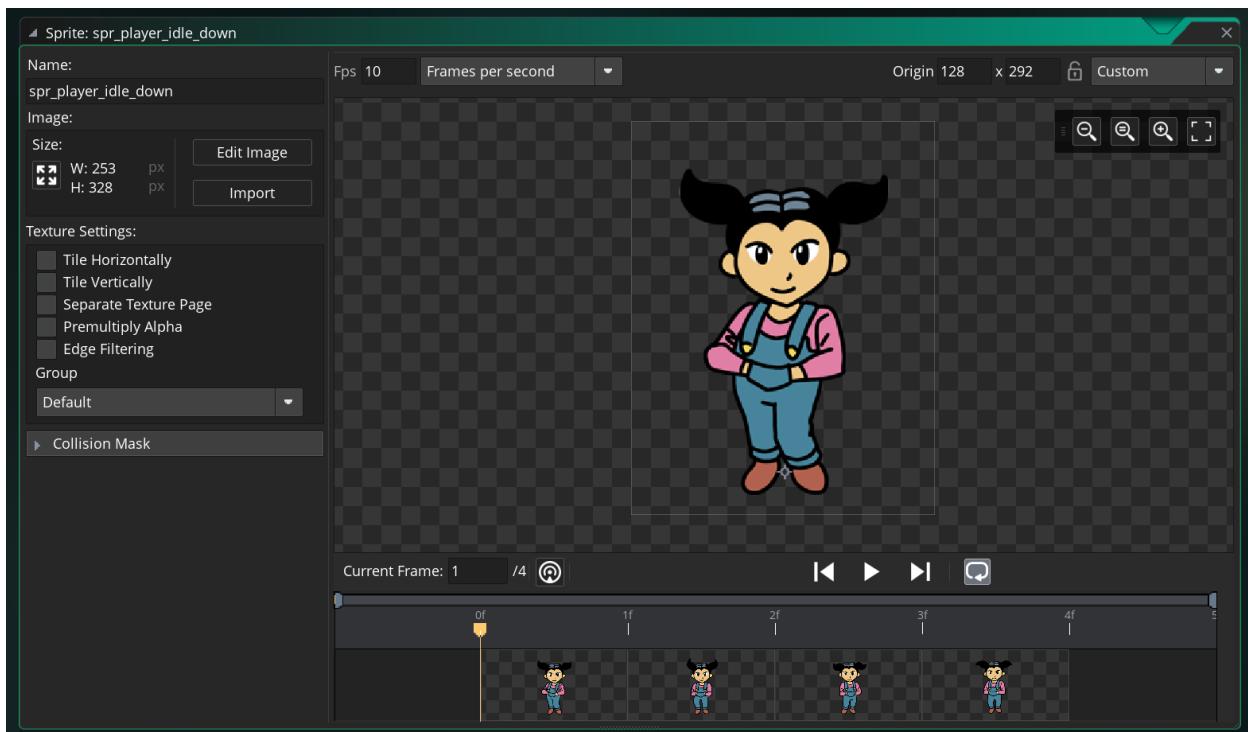
Look at the row of animation frames at the bottom of the Sprite Editor.

You can click on each frame to view it, or you can click the Play ▶ button above the row of frames to play the animation.



Now, we just need to do one last thing: see how our player Sprite has a little grey target in the upper left corner? This is its *origin* — the point from which we'll be checking for objects and tracking the position of the player.

For now, just click and drag that origin point to somewhere between the player's feet.



The first player sprite, correctly split into frames, with its origin point moved

With this complete, you can close the Sprite by clicking the X at the top-right of the Sprite Editor window. (Note that GameMaker Studio 2 automatically saves your assets as you work on them.)

If you haven't already done so, click our new spr_player_idle_down Sprite in the Asset Browser and drag it into the Sprites group to keep things tidy.

4.5 Applying a Sprite to an Object

Go back to your player Object (`obj_player`); if it's not already open in your Workspace, just double-click it in the Asset Browser. You'll see it has a generic white box for an image and a drop-down menu that says, "No Sprite." Click on this menu and choose the new `spr_player_idle_down` Sprite we just made.

Voila! Our player now has a Sprite attached to it. (Otherwise, it would just be an invisible nothing and that's no fun.)

	<p>Tip: if you want to clean up your Workspace at any time, you can right-click on a blank space in the Workspace and choose Windows > Close All. You can always re-open an asset by double-clicking on it in the Asset Browser.</p>
--	--

4.6 Setting up a Room

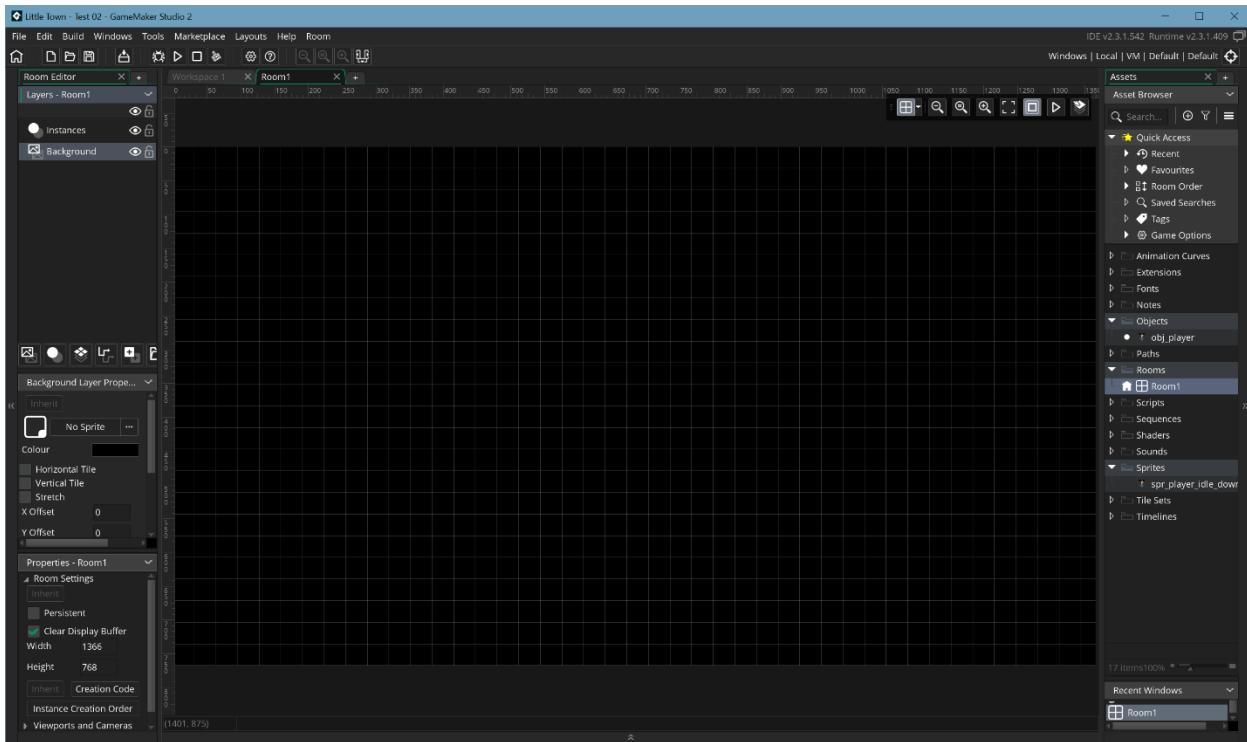
Now that we have a player object we can see, let's put it in a Room! Luckily, we already have one to use because GameMaker Studio 2 always creates one Room for us by default.

In the Asset Browser, open the folder called Rooms and double-click Room1. It will open our first Room, which is, I'll admit, pretty boring.

A *Room* in GameMaker Studio 2 is like a level or scene in your game. But it's more than that; a Room can be a title screen, a loading screen, a credits screen, or almost anything else. But like the name implies, it's a *place where stuff happens*.

Right now, though, nothing of interest is happening. So, let's change a few things. On the left is your Room Editor, which has all sorts of important things. From top to bottom you'll see:

- Room Layers: this is where we'll place Objects, background images and more
- Layer Properties: this lets you change all sorts of stuff for a particular layer
- Room Settings: here we can change the Room size and edit Cameras, which we'll do later



The Room Editor and our boring, blank Room

For now, click on the **Background** layer and look at its **Properties** in the section on the lower-left. (If things look squished or out of the way, you can adjust the size of the different sections here.) Click on that black void beside **Colour** and pick any colour you like. Wow! Something interesting!

Now, click on the **Instances** layer in the **Layers** section. Then look at your **Asset Browser** (it should still be on the right of the IDE). Find our **obj_player** Object and drag it onto the room to place it.

	<p>Tip: If you get an error when trying to drag obj_player into the Room, make sure you have the Instances layer selected.</p>
--	---

Though we created a player *Object*, by dragging it into our Room we've created an *instance* of that Object in the Room. This is why the layer's default name is **Instances**. You can rename this layer to anything you like, but for this tutorial, let's leave it as is.

Also, do you notice the double-circle icon on the layer to the left of its name? This shows you at a glance what *type* of layer it is. Different layer types support different kinds of assets, so this is important.

(You can see below the layer list icons that represent other layer types.)

One last thing before we continue; we should rename our Room so it's easier to track later. In the Asset Browser, right-click on Room1 (or whatever your lone Room is called) and choose Rename. Name it `rm_gameMain`.

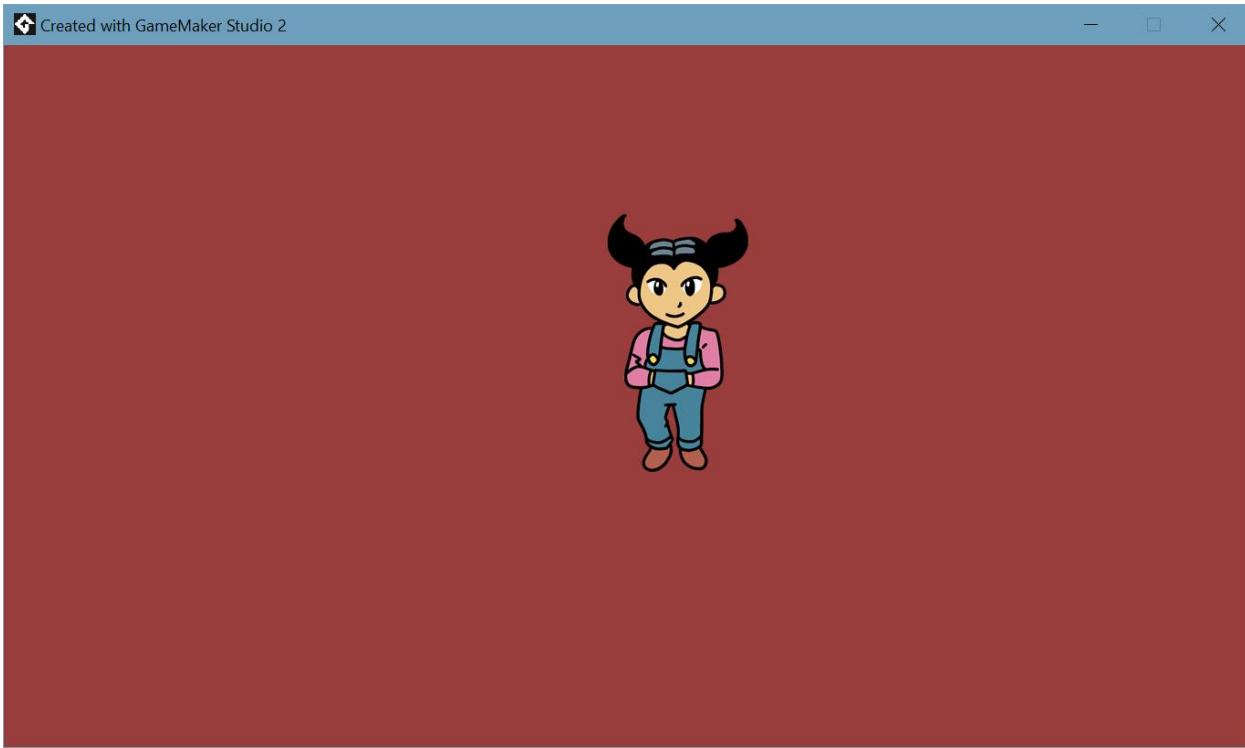
4.7 Running the game

Okay, let's run our game for the first time and see what's what.

(First, if you haven't been doing so already, make sure to save your project as you work on this course. Go to File > Save Project in the menu bar or use the keyboard shortcut appropriate to your platform.)

Click the Run ▶ button at the top of the IDE and watch as our "game" springs to life! You should see our player doing its thing in the Room. Note that the speed at which she animates right now is based on that FPS option in that first Sprite we imported.

Amazing. Once your jaw is back in place, close the game window and go back to GameMaker Studio 2.



Our “game,” running for the first time.

4.8 Making things happen with Events

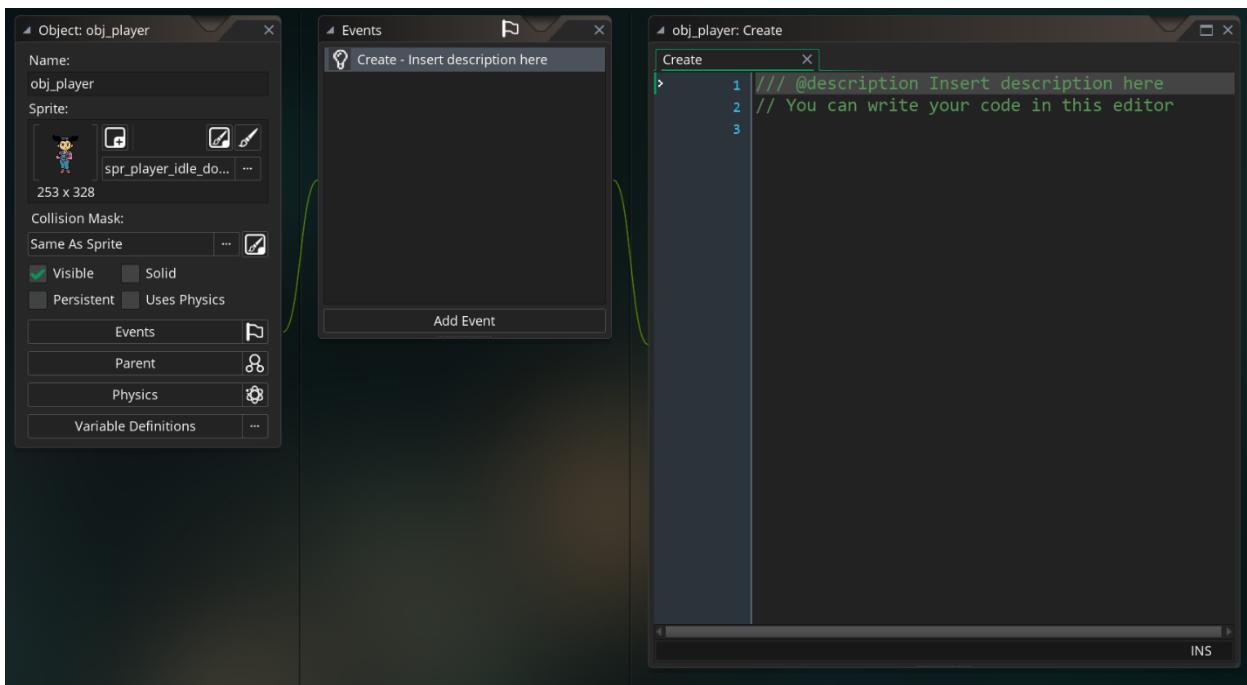
Okay, now we need to make our player *do* something. Go to your Workspace again (remember, you can see tabs at the top of the IDE; one should be called Workspace 1 or something similar).

If it's not already opened, find `obj_player` again in the Asset Browser and double-click it to open it.

On the left of the Object editor, you'll see some main options and on the right you'll see Events, which is currently empty. (If you don't see Events, just click the Events button on the left.)

Everything in GameMaker Studio 2 happens in an Event; for example, when a player presses a key or clicks a mouse button, that's an Event. There are special Events for all sorts of things, many of which we'll get to play with in this course.

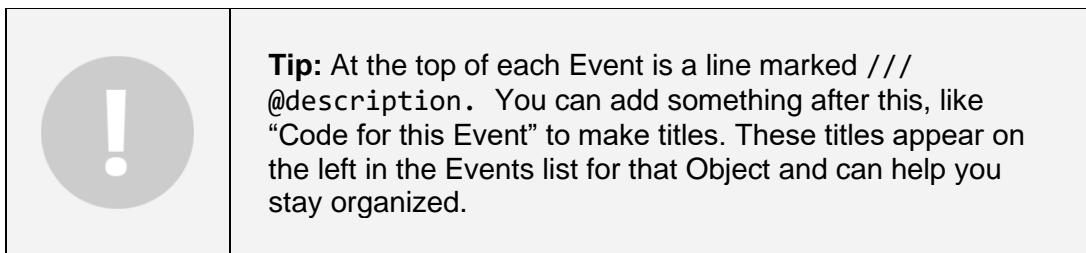
Let's add our first Event to `obj_player` by clicking the Add Event button. You'll see a big list of different Events and even some in sub-lists. For now, just choose Create.



Our brand new Create Event. Note that you can change the description to label the Event.

The Create Event happens when an Object is “created” within the game — in other words, when it first appears. Think of a space shooter where mashing a button shoots little bullets; each of those bullets is an Object and as soon as they appear, they each run their Create Event.

Since we’ve placed an instance of obj_player into our first Room, this Create Event will happen as soon as that Object appears in the Room — which is to say, as soon as we see the Room in our game.



4.9 Writing our first GameMaker Language (GML) code

GameMaker Studio 2 uses its own programming language called *GameMaker Language*, or *GML*. This course won't be going into every detail of GML but will instead teach you what you need to know as we proceed.

	<p>Important: It is <i>highly recommended</i> that you read the GameMaker Studio 2 manual entries for the code we use if you want more information on what we're doing throughout the course. You can access the manual at any time by choosing Help > Open Manual in the menu bar at the top of the IDE.</p>
---	---

For now, let's enter our first code in obj_player's new Create Event, underneath the @description line:

```
// Variables  
walkSpeed = 16;  
vx = 0;  
vy = 0;  
dir = 3;  
moveRight = 0;  
moveLeft = 0;  
moveUp = 0;  
moveDown = 0;
```

These are all *variables*. A variable is a way to store information, called a *value*. GameMaker Studio 2 has built-in variables (i.e., things that always exist), but we can also make our own, like magic. Here, we're creating several variables and assigning values to them for later use.

We can assign different kinds of values to our variables, including numbers (16, 0.2, etc.), strings (e.g. myName = "Jimmy";) and more, but for now, these simple variables will do.

	<p>Tip: A line that starts with 2 or more forward slashes (such as // This is a comment) is "commented out," meaning it is not code that will run. You'll see this a lot in this course; use it to keep notes for yourself if need be as you work.</p>
---	---

4.10 Using keyboard Events to move right

The first thing we want to do is be able to move our player Object around by using the keyboard, so let's add another Event. With obj_player open in the Object Editor, click Add Event and choose Key Down > Right.

You'll notice there are several Key events. They work like this:

Event	What it means
Key Down	Occurs if that key is being held down
Key Pressed	Occurs only when that key is first pressed down
Key Up	Occurs when that key is released

In the new Key Down – Right Event, write the following code:

```
| moveRight = 1;
```

As you can see, we're changing the *value* of the variable moveRight to 1.

In the Object Editor, click “Add Event” again and choose Key Up > Right.

In this new Key Up – Right Event, add this line of code:

```
| moveRight = 0;
```

So here we're resetting moveRight back to 0 when we release the right arrow key. Simple enough, right? However, right now all we're doing is setting the value moveRight — it's what we do next that will actually allow our player to move around.

4.11 Controlling our player with a Step Event

Let's add yet another Event to our player Object — the Step Event. In the Object Editor, click Add Event and choose Step > Step. Note that there are *multiple* Step Events, but we want the one just called Step.

The Step Event happens *every single frame* of the game, so we must use it wisely. If our game is 60 FPS, then our player Object is going to run whatever code is in here 60 times every second.

In this new Event, add the following code:

```
// Calculate movement  
vx = (moveRight * walkSpeed);  
  
// If Idle  
if (vx == 0) {  
    // do nothing for now  
}  
  
// If moving  
if (vx != 0) {  
    x += vx;  
}
```

You'll notice in the second two blocks we're asking some questions here about the value of vx. If you want to check the value of a variable you can do it like so:

```
if ([variable] == [value]) {  
    // do something  
}
```

This is called an *if statement*. In GML, == means "is equal to?" and != means "is *not* equal to?"

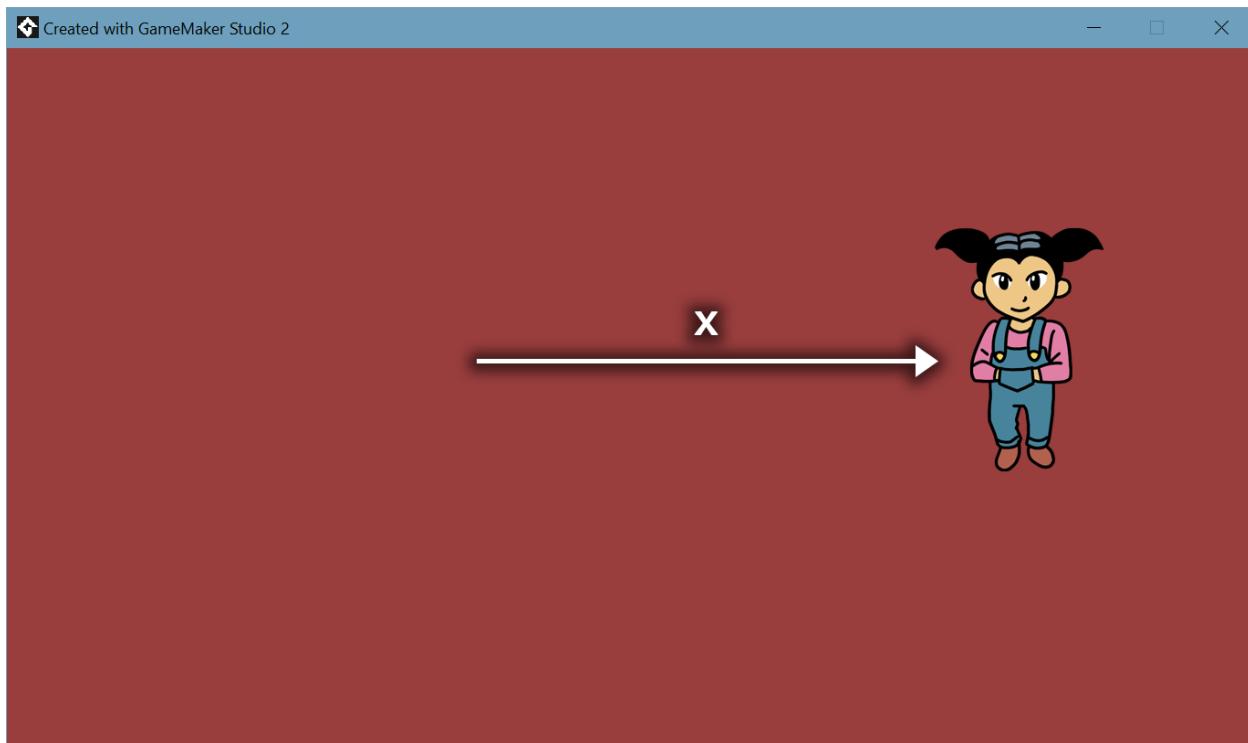
Just to be clear, let's break this down:

Code	Notes
<pre>// Calculate movement vx = (moveRight * walkSpeed);</pre>	<p>We're calculating the value of vx based on two other values: moveRight and walkSpeed:</p> <ul style="list-style-type: none">- We set moveRight to 1 when we're holding down the right arrow key (in the Key Down - Right Event)- We reset moveRight to 0 when we release the right arrow key (in the Key Up - Right Event)- In obj_player's Create Event, we set walkSpeed to 16 <p>Therefore, vx can be either 16 (1 x 16) or 0 (0 x 16)</p>

<pre>// If Idle if (vx == 0) { // do nothing for now }</pre>	If vx <i>is equal to</i> 0, do something. (Right now, we're not doing anything in particular, so we just have a comment here)
<pre>// If moving if (vx != 0) { x += vx; }</pre>	If vx <i>is not equal to</i> 0, then <i>add</i> the value of vx to obj_player's x position. (In GML, += means "add to". Not unexpectedly, -= means "subtract from.")

Let's test our game again so you can see what's happening: click the Run button at the top of the IDE.

When your game window opens, press the right arrow key; the player should move to the right. Release the right arrow key, and the player should stop moving.



The player Object moves to the right

Close the game and return to GameMaker Studio 2. Now that we understand the basics of movement using Events, we can add the other directions.

4.12 Moving our player Object left

First, make sure obj_player is opened. In the Object Editor, click Add Event and choose Key Down > Left.

In this new Event, add this code:

```
| moveLeft = 1;
```

Next, add another Event and choose Key Up > Left.

In this new Event, add this code:

```
| moveLeft = 0;
```

(This is the same as what we did for the right arrow key.)

Finally, open obj_player's Step Event again and update the // Calculate movement code block like so (changes we've made are **bold blue**).

```
| // Calculate movement
| vx = ((moveRight - moveLeft) * walkSpeed);
```

If you're wondering how this math works to make the player Object move left, consider:

vx =	((moveRight - moveLeft) * walkSpeed);	
	((1 - 0) * 16) = 16	(if pressing right arrow key)
	((0 - 1) * 16) = -16	(if pressing left arrow key)

And since in the Step event we use this code to actually move the player Object:

```
| // If moving
| if (vx != 0) {
|   x += vx;
| }
```

We're adding vx to obj_player's x. If the player Object starts at 0, for example, and we hold down the right arrow key, its x will be 16, then 32, then 48 and so on. If we hold the left arrow key when the player Object starts at 0, its x would be -16, -32, -48 and so on.

Run the game again and use the right and left arrow keys; you'll be able to move the player Object back and forth, horizontally.

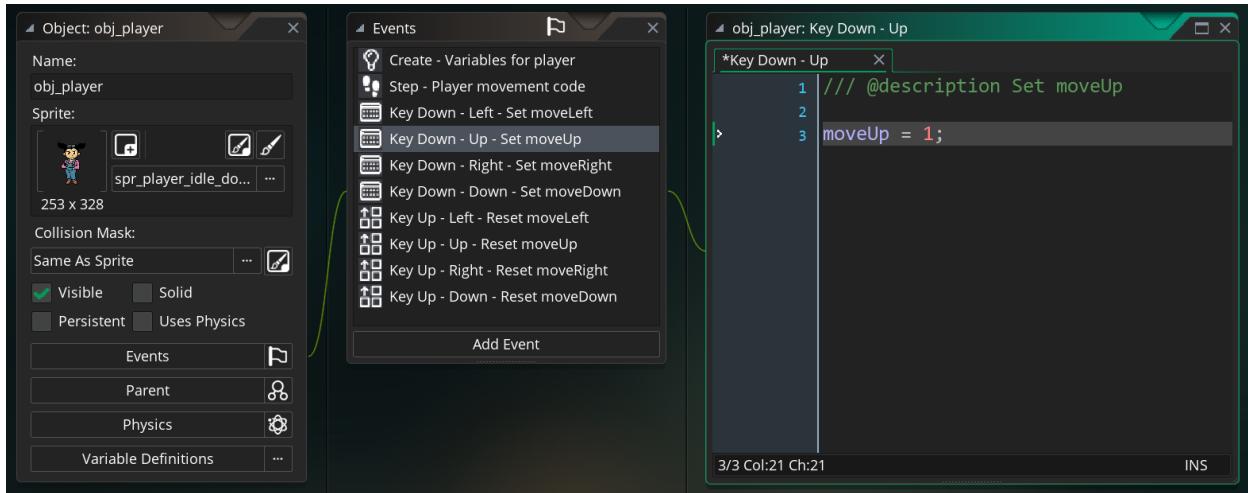
Once you're done having fun, close the running game window and return to GameMaker Studio 2.

4.13 Adding vertical movement

Now that we have these principles in place, we can make our player Object move up and down too.

With obj_player open, use the Add Event button to add the following Events, and enter the code shown here for each:

Event	Code to type
Key Down - Down	<code>moveDown = 1;</code>
Key Up - Down	<code>moveDown = 0;</code>
Key Down - Up	<code>moveUp = 1;</code>
Key Up - Up	<code>moveUp = 0;</code>



Our player Object will all four key Events, in both Down and Up variants. Note how adding a Description to the top line of each event lets us read the Event list more easily.

When you have these Events in place, open the Step event again, and edit the three code blocks we've written there like so:

```
// Calculate movement
vx = ((moveRight - moveLeft) * walkSpeed);
vy = ((moveDown - moveUp) * walkSpeed);

// If Idle
if (vx == 0 && vy == 0) {
    // do nothing for now
}

// If moving
if (vx != 0 || vy != 0) {
    x += vx;
    y += vy;
}
```

You can see we've added vertical movement to our player with these additions. We've also added checks for vy in the second and third blocks.

With this done, run your game again and try all four arrow keys. You should be able to move the player Object around in all four directions!

4.14 Changing all our movement Events to code

It's important to understand how GameMaker Studio 2's Events work, and to grasp what Key Down and Key Up Events are. But what if I told you we could replace all eight of the Events we just created with only four lines of code?

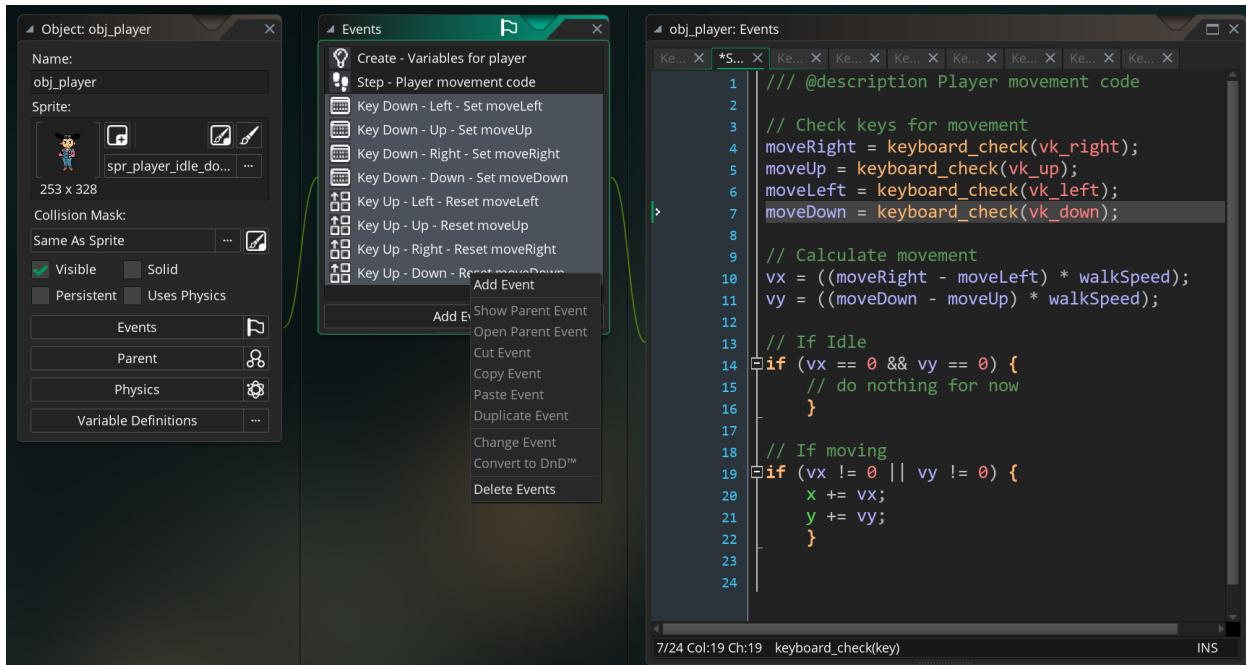
If it's not opened in your Workspace already, open obj_player and its Step event. Enter the following new code above the // Calculate movement code block:

```
// Check keys for movement
moveRight = keyboard_check(vk_right);
moveUp = keyboard_check(vk_up);
moveLeft = keyboard_check(vk_left);
moveDown = keyboard_check(vk_down);
```

Now, in the Object Editor, we're going to delete all eight of those Key Events, because we don't need them anymore.

You can right-click on each Key Down and Key Up event and choose Delete Event. You can also shift-click a group of Events all at once and use the same right-click menu to delete them all in one go.

You'll receive a warning when you do this (and yes, you can undo if you mess up).



Now that we have our new movement code, we can delete all eight of these Events

Once you've deleted these Events, run your game again and try the arrow keys on the keyboard. Our player Object should be moving just like before!

When you're ready, close your running game and return to GameMaker Studio 2.

So how does this work? Let's take the first line of our new code as an example:

```
| moveRight = keyboard_check(vk_right);
```

What's on the right of the = sign (`keyboard_check(vk_right);`) is called a *function* — a key element of programming in GML and is how we'll accomplish most tasks. Functions usually work like this:

```
| functionName(argument);
```

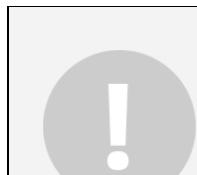
The function `keyboard_check()` requires a single *argument* — a detail it needs to do its job; in this case, the key we want our game to check. Here, we're checking the right arrow key (called `vk_right` in GML).

And, as it turns out, the `keyboard_check()` function does the exact same thing as the Key Down Events we used at first! So, `keyboard_check(vk_right)` is the same as creating a Key Down – Right Event.

Now, what's new here is that on the left of the = sign, we're storing the *result* of that function in the variable `moveRight`, like so:

moveRight	=	keyboard_check(vk_right)
Update the variable moveRight...		... with the result we get by checking if the player is pressing the right arrow key

And since this code is in the Step Event, 60 times a second, `obj_player` is asking, "Did you press the right-arrow key?"



Tip: If the user is pressing the right arrow key, then `moveRight = 1`. If they're not, then `moveRight = 0`. Think of "1" being "yes" or "true," and "0" being "no" or "false."

Now that we have this updated movement code, we're ready to do a whole lot more. Make sure to save your project, and let's move on.

4.15 Changing our player Sprites

We're now going to do two things:

1. Change the player Object's Sprite based on whether it's idle (standing still) or walking
2. Do so based on its direction (up, down, left, right)

First, let's get all our other assets ready. Using File Explorer (Windows) or Finder (Mac), navigate to the Assets folder provided with this course. Open the Sprites > Characters and Items folder and drag these Sprites into the Sprite group in GameMaker Studio 2's Asset Browser:

- sprPlayer_idle_up_strip04
- sprPlayer_idle_right_strip04
- sprPlayer_idle_left_strip04
- sprPlayer_walk_up_strip04
- sprPlayer_walk_right_strip04
- sprPlayer_walk_down_strip04
- sprPlayer_walk_left_strip04



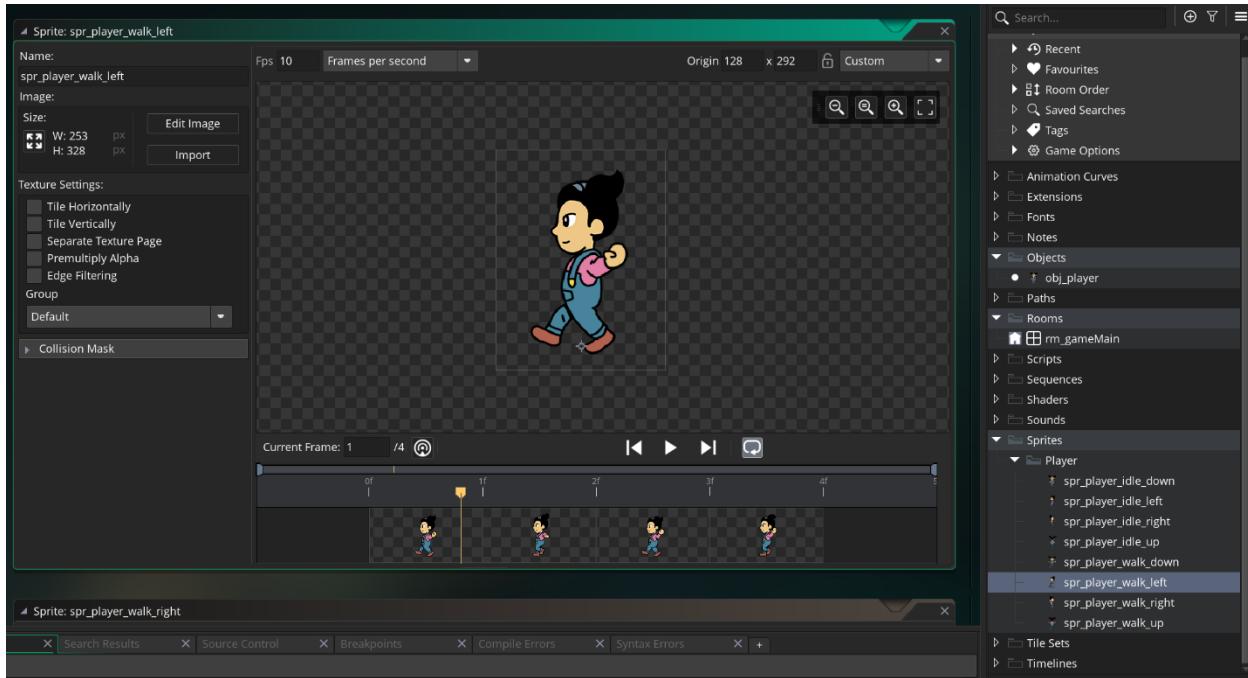
Tip: you can organize the Asset Browser however you like. For example, you could create a Group in Sprites called Player and put all these player-specific assets in there. To create a group, just right-click in the Asset Browser and choose Create Group.

Open each Sprite and follow the same steps we took in [Converting a Sprite Strip manually](#) to turn these strips into correctly animating Sprites (if they have not been automatically converted for you).

Once each new Sprite Strip has correctly been converted into a Sprite with frames, you can remove its _stripXX suffix.

Make sure the origin point for each of these new Sprites is the *same* as what you else things will get weird. You can see that first Sprite (spr_player_idle_down)'s origin x and y values by double-clicking the Sprite in the Asset Browser; in the top-right of the Sprite Editor window are two Origin fields.

Adjust each Sprite's FPS value to match.



Here, we've converted one of the new player Sprites into frames; set its FPS to 10 to match the original sprite; changed its Origin to match as well; and organized these new player Sprite assets within the Asset Browser

Once you have all the new Sprites dealt with, open obj_player and its Step Event again.

Update the // If moving code block like so:

```
// If moving
if (vx != 0 || vy != 0) {
    x += vx;
    y += vy;

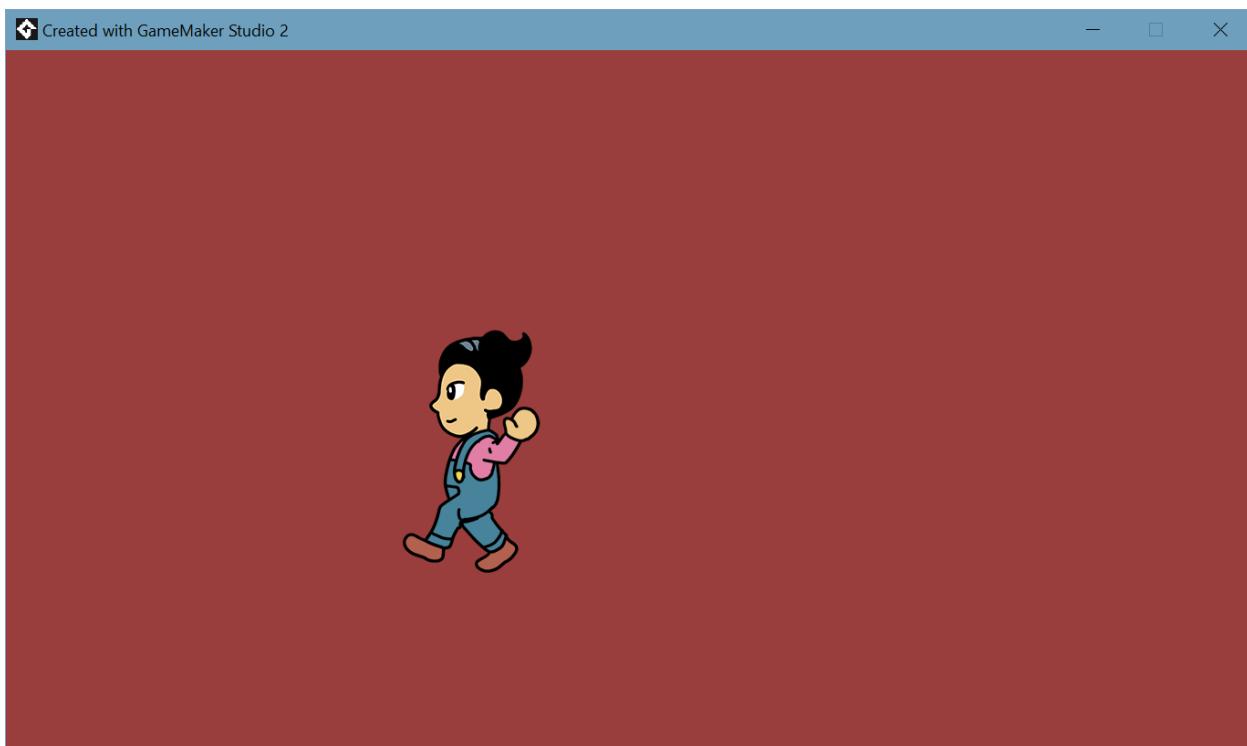
    // Change walking Sprite based on direction
    if (vx > 0) {
        sprite_index = spr_player_walk_right;
        dir = 0;
    }
    if (vx < 0) {
        sprite_index = spr_player_walk_left;
        dir = 2;
    }
    if (vy > 0) {
        sprite_index = spr_player_walk_down;
```

```
    dir = 3;
}
if (vy < 0) {
    sprite_index = spr_player_walk_up;
    dir = 1;
}
}
```

Since we're already using vx and vy for movement, we can check it here with a bunch of if statements. It's not the most efficient way to do this, but it totally works. We also set dir to a different value in each case so we can use it in the next bit.

(If you check obj_player's Create Event, you'll see that we initialized the dir variable back in [Writing our first GameMaker Language \(GML\) code](#).)

Run your game and try it out! You should see your player moving in all four directions, with the correct Sprites for each direction. (Remember, if the Sprite appears to "skip around" as you change direction, you might need to check the origin point of each Sprite.)



Our player Object now looks like it's walking in all four directions.

You may have noticed one problem, however; once we stop moving, our player Object keeps walking, which isn't what we want. Let's close the game window, return to GameMaker Studio 2 and fix that.

Back in obj_player's Step Event, let's replace the "If idle" block with new code:

```
// If Idle
if (vx == 0 && vy == 0) {
    // Change idle Sprite based on last direction
    switch dir {
        case 0: sprite_index = spr_player_idle_right; break;
        case 1: sprite_index = spr_player_idle_up; break;
        case 2: sprite_index = spr_player_idle_left; break;
        case 3: sprite_index = spr_player_idle_down; break;
    }
}
```

This new code is called a *switch function*. It's a handy way to combine a bunch of if statements into something easier to read. (You can read more about how this works in the GameMaker Studio 2 manual.)

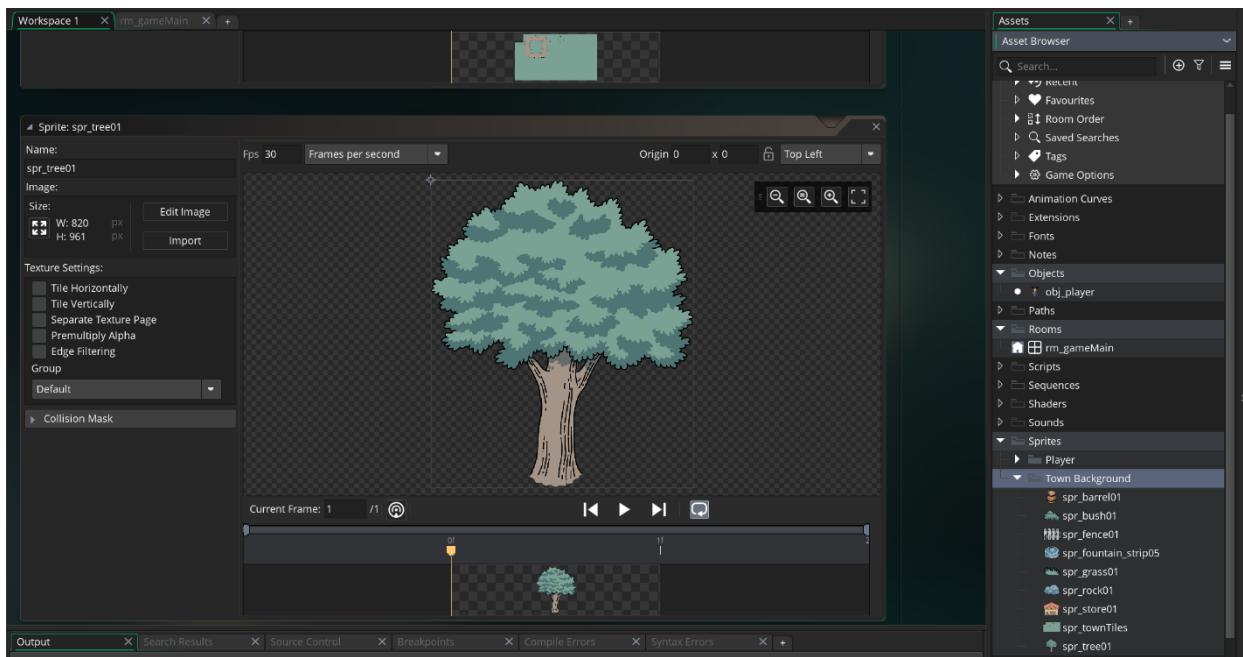
Here, we're checking the `dir` variable (which we set whenever we're moving) and applying the correct idle Sprite for the last direction that `obj_player` was moving.

Run the game again and see the difference; now our player correctly moves and stops!

4.16 Preparing Tile Sets

Okay, now it's time to build our game's little town. We're going to cover a lot of functionality regarding Rooms, so let's get started.

Using File Explorer (Windows) or Finder (Mac), navigate to the Assets folder provided with this course. Go to Sprites > Backgrounds and drag the files there into the Asset Browser. As mentioned before, you can organize these assets if you want to (for example, by putting them in a Town Background group within the Sprites group.).



Our newly imported town assets. We've organized these new assets into a Group named "Town Background."

You'll see a bunch of Sprites for things like trees and bushes and a big image called spr_townTiles. Open this image from the Asset Browser.

Most games build levels and areas with reusable assets that designers can place and nudge to create a scene. In GameMaker Studio 2, we can use *Tiles* to do this. They are just as the name suggests; tiled graphics that we can use over and over. This file (spr_townTiles), is going to let us design a town by creating a *Tile Set*.

Tile Sets need two pieces: a Tile Set asset and a Sprite attached to that asset. We already have the Sprite (spr_townTiles) so we just need to create the asset.

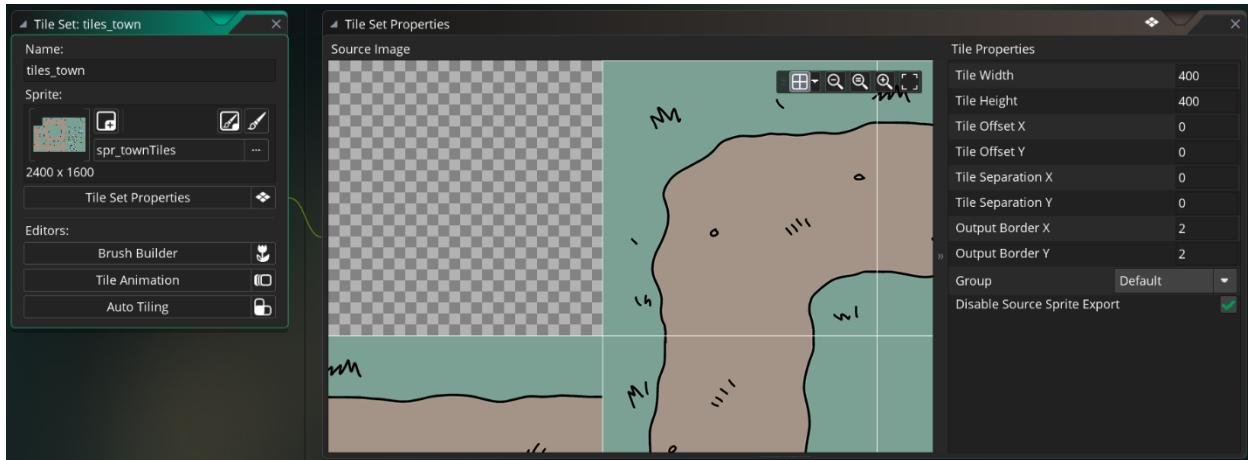
In the Asset Browser, you'll see a pre-existing group called *Tile Sets*. You can right-click on this and choose *Create > Tile Set*. (If you don't see this Group, you can make one yourself.)

In the Tile Set editor that pops up, name this tiles_town,

Click the *Select Sprite* menu to choose the Sprite you just imported (spr_townTiles). You'll see your Tile Set now has a grid and probably doesn't make much sense.

That grid is the tile grid and as you can see, it's way too small. So, on the right side of the Tile Set editor, change both the *Tile Width* and *Tile Height* to 400. Now the grid should look correct and you'll see a blank space in the top-left.

(For more details on the Tile Set editor, you can check out the GameMaker Studio 2 manual.)



Our Tile Set with Sprite attached and Tile Width and Height set.

4.17 Designing our town

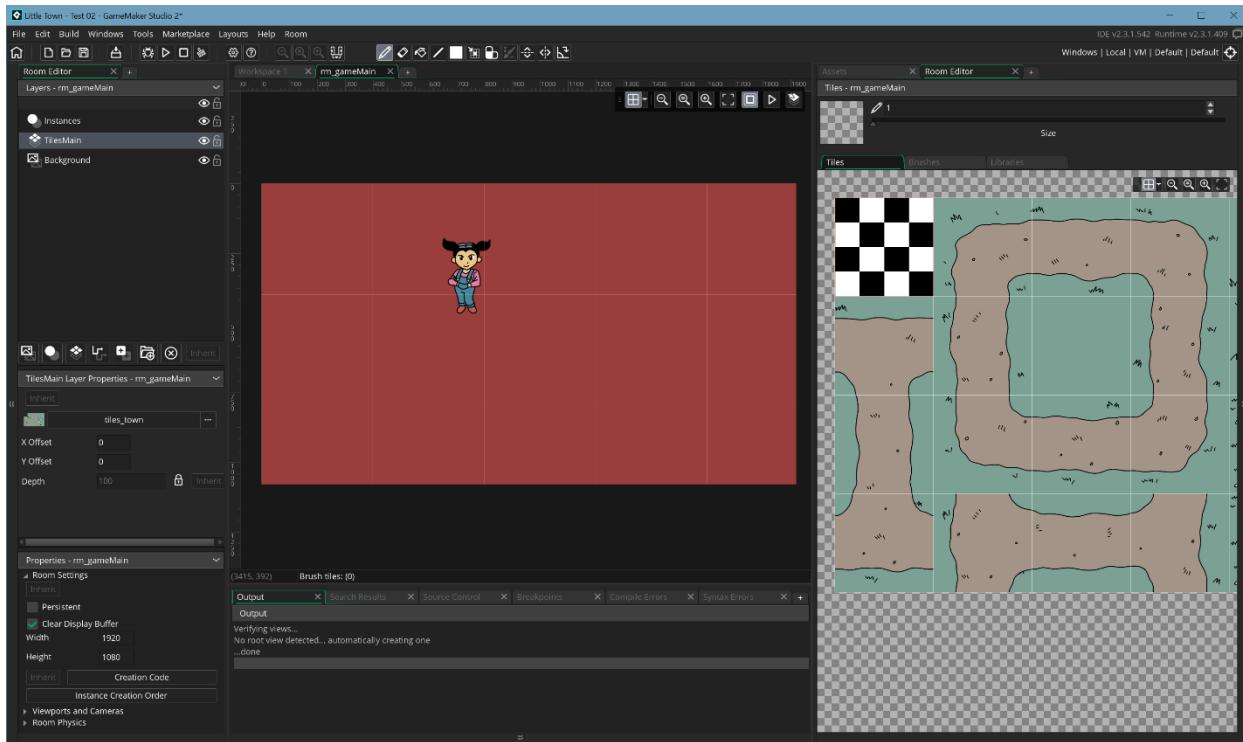
Open our one-and-only Room (`rm_gameMain`) and make sure you can see the Room Editor (the tab that shows us layers like Instances and Background. At the bottom of this tab are those layer buttons we mentioned; click on the Create New Tile Layer (button).

This will make a new layer called `Tiles_1`. Right-click on the new layer and choose Rename; rename it to `TilesMain`.

Drag the layer to place it between Instances and Background.

Select `TilesMain` in the Layers section so we can edit it. Below, under Layer Properties, you'll see a menu that says, "No Tile Set." Click on this and choose the `TilesTown` Tile Set we just made.

You'll see a new tab open called Room Editor and it will show our new Tile Set. If it looks huge or you can't see it properly, you can use the little magnifying glass tools to zoom in and out.



The `rm_gameMain` Room with the *TilesTown* Tile Set, ready for placement

You might have noticed that our little Room is way too small to do much with, so let's change that. In the Properties tab (by default on the lower left of the IDE), you'll see a few sub-sections you can show and hide.

The first is **Room Settings**; if it's hidden, click the little arrow to show its contents and change these values:

```
Width: 4000
Height: 2400
```

Now we've got some space to play in! In the main Room view, you can use the magnifying glass tools at the top to zoom in and out and you can hold the space bar while clicking and dragging with your mouse to pan around. Make sure you can see the whole Room.

4.18 Placing Tiles

You'll notice a grid in the Room that corresponds to the size of our Tile Set's Tile Width and Tile Height (in this case, 400). If you click on the Grid icon at the top right on of the Room tab, you can turn the grid on and off and check the grid size.

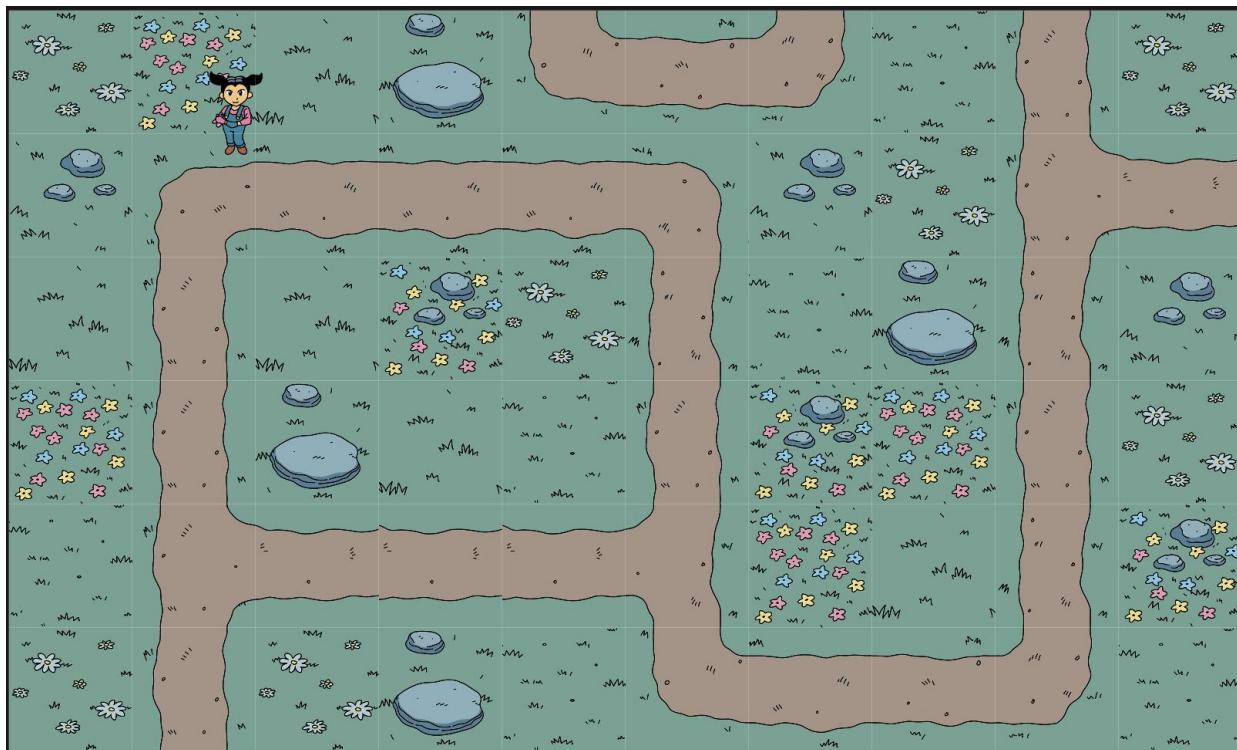


The Room toolbar. L-R: Grid options; zoom out; reset zoom; zoom in; centre fit; show views; play animation; select from any layer

For now, start placing Tiles in your Room! You can “stamp” them one by one by clicking on a tile in the Room Editor tab, or you can click and drag on the Tile Set to select and stamp a series of Tiles all at once.

(If you want to remove a tile, you can hover your cursor over it and right-click.)

Design the base for your town anyway you like. Fill up every tile space in the Room and have fun!



A simple implementation of our Tile Set

4.19 A Room with a view

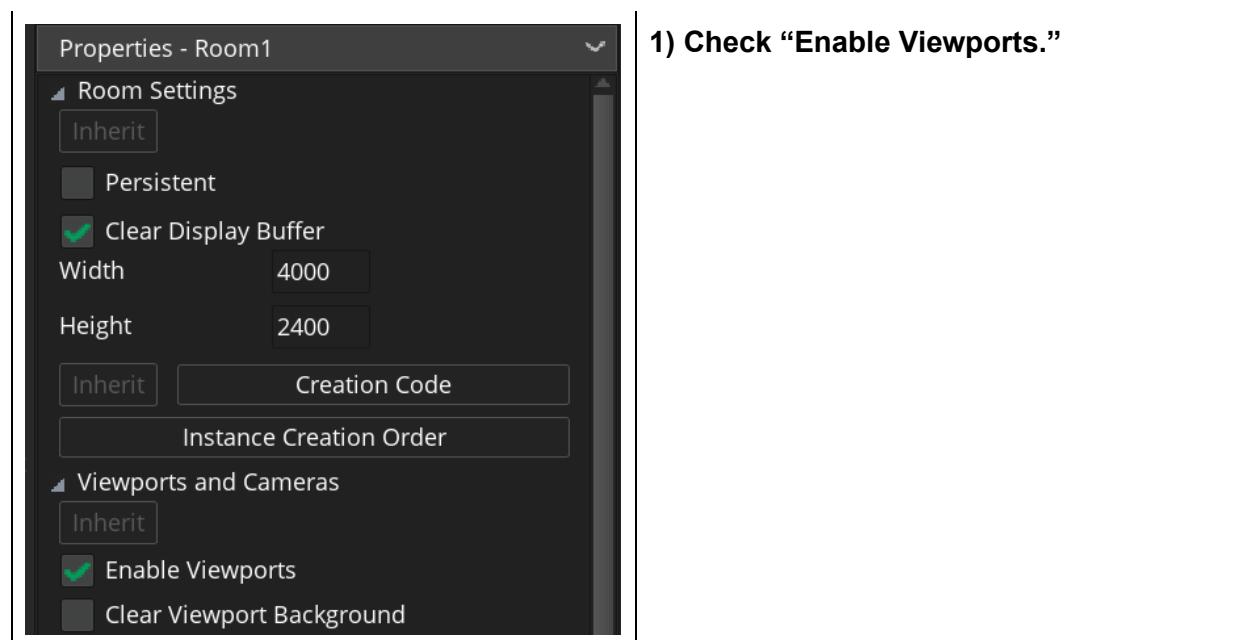
Before we continue with populating our lovely town, we need to make a quick detour to talk about *Cameras* and *Viewports*. (You can read all about this in the GameMaker Studio 2 manual.)

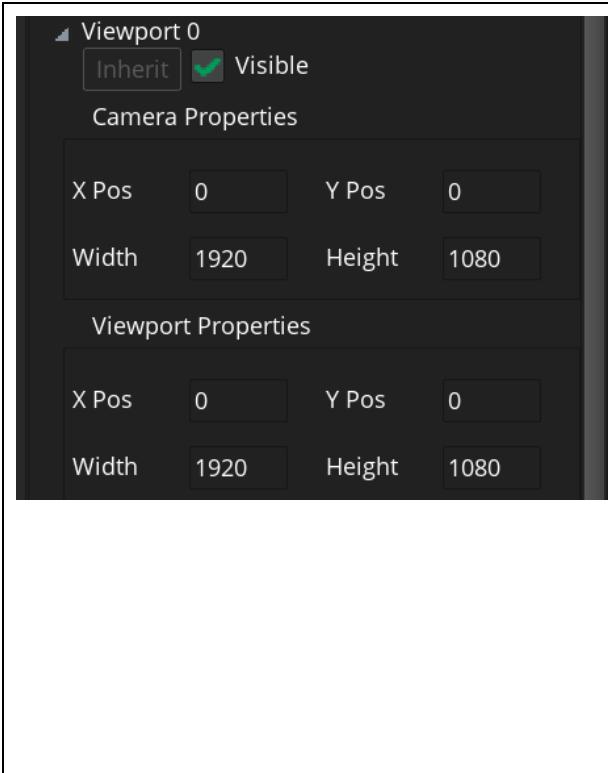
We're not going to show the entire town at once in the game window, because it's large. Think about a classic platforming game or RPG, where you explore a large world or level and the game "scrolls" as you play.

To recreate this effect, we need to set up a *Viewport* that will follow our player Object as it moves throughout the town.

With `rm_gameMain` still open, look for the `Room Properties` section in the `Room Editor` tab (where we changed the size of the Room).

You'll see a section you can expand called `Viewports and Cameras`. Click it to expand its contents and change its properties like so:



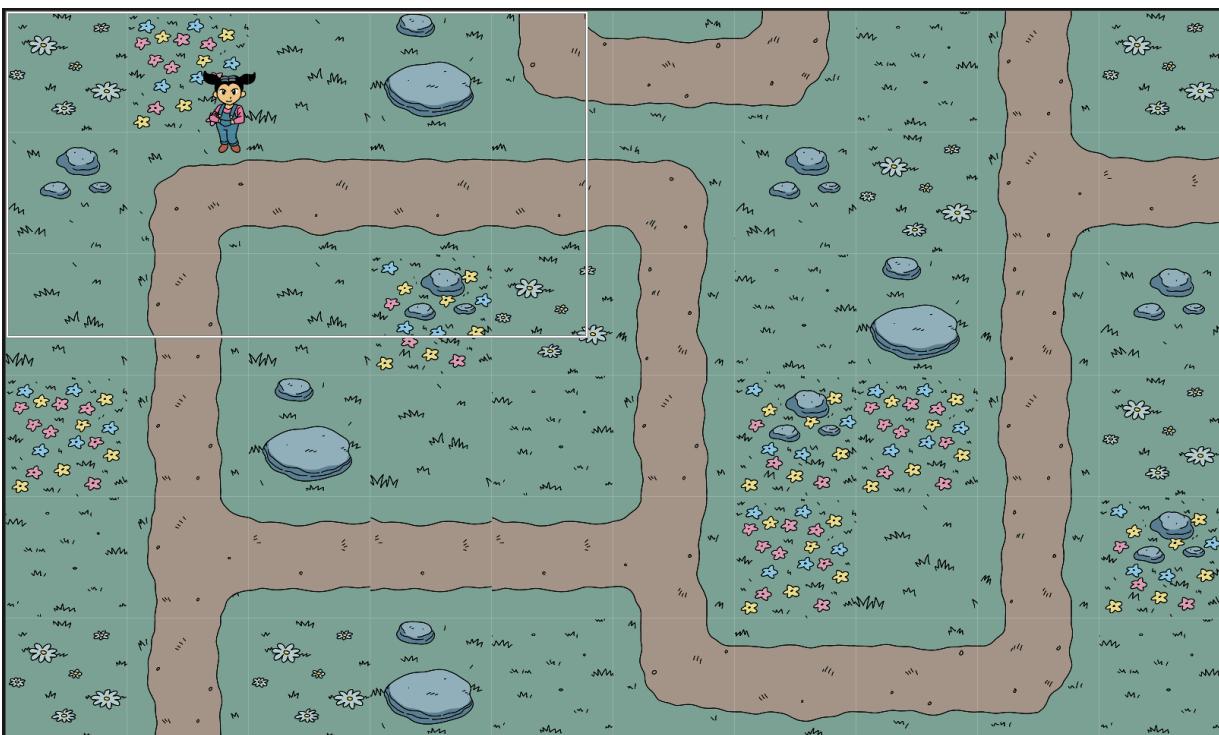


2) Expand “Viewport 0.” These are preset Viewports, but we only need just the one.

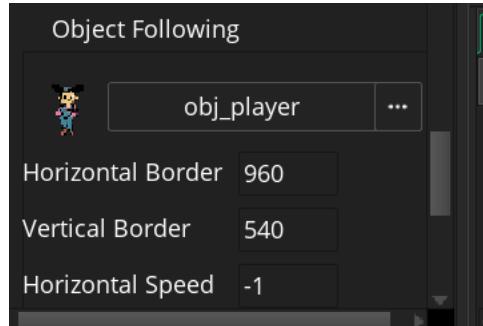
3) Check “Visible” to turn this Viewport on. As soon as you do, you’ll see a white border appear in your Room; this is what you’ll see when you play the game now (instead of the entire Room).

4) Change the width and height of both the Camera Properties and the Viewport Properties to 1920 (width) and 1080 (height). If you’re working on a laptop with a lower resolution, or on a desktop computer with a high-resolution monitor, you can change this.

Note that these numbers need to be the same for both Camera *and* Viewport. Otherwise, your game could look stretched or distorted.



Once you make Viewport 0 visible, you'll see it represented in the Room Editor as a white box.

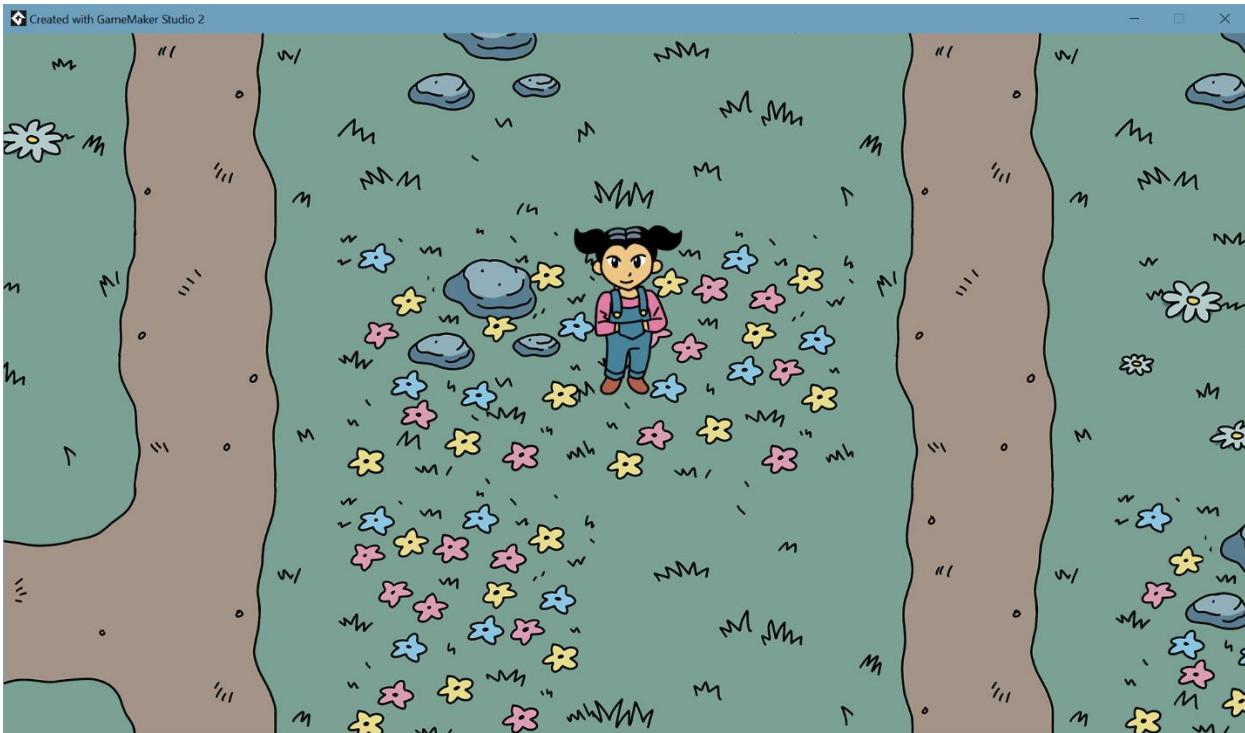


5) Under “Object Following,” click “None” and choose your player Object (`obj_player`). This way, when you move around in the Room, the Camera will follow this Object. If you leave this as “none,” the Camera will stay where it is and you’ll be able to wander out of sight,

6) Finally, we need to change “Horizontal Border” and “Vertical Border.” Think of these as forming a buffer around the edge of the screen when your game runs. When the Object that the Camera is following (`obj_player` in our case) reaches this buffer, the Camera will start moving to follow it.

By making the Horizontal Border 960 and the Vertical Border 540, the buffer is dead centre (since we made our Camera/Viewport Properties 1920x1080). This means that as soon as the player Object moves, the Camera will follow it.

All right! Now, let’s click the Run button to test our game out again. We should see the game window pop up at a 1920x1080 size and we should be able to move around our large, newly-tiled town with the Camera keeping up as we do.



We can move around our town now, and the camera will follow!

4.20 Adding Objects to our town

With our town partially laid out, let's create an Object using some of the other assets we imported.

Close the game window and return to GameMaker Studio 2. Go back to your Workspace and in the Asset Browser, find the group of town background Sprites we imported. Double-click on `spr_barrel01` to open it in the Sprite Editor.

You can actually place just plain old Sprites on a layer in a Room but doing so means our player Object won't be able to interact with them. So, we're going to make each of these environmental details into Objects.

First, let's take that origin point on our barrel Sprite and move it to the bottom center of the Sprite, so it lines up with the middle of the base of the barrel.

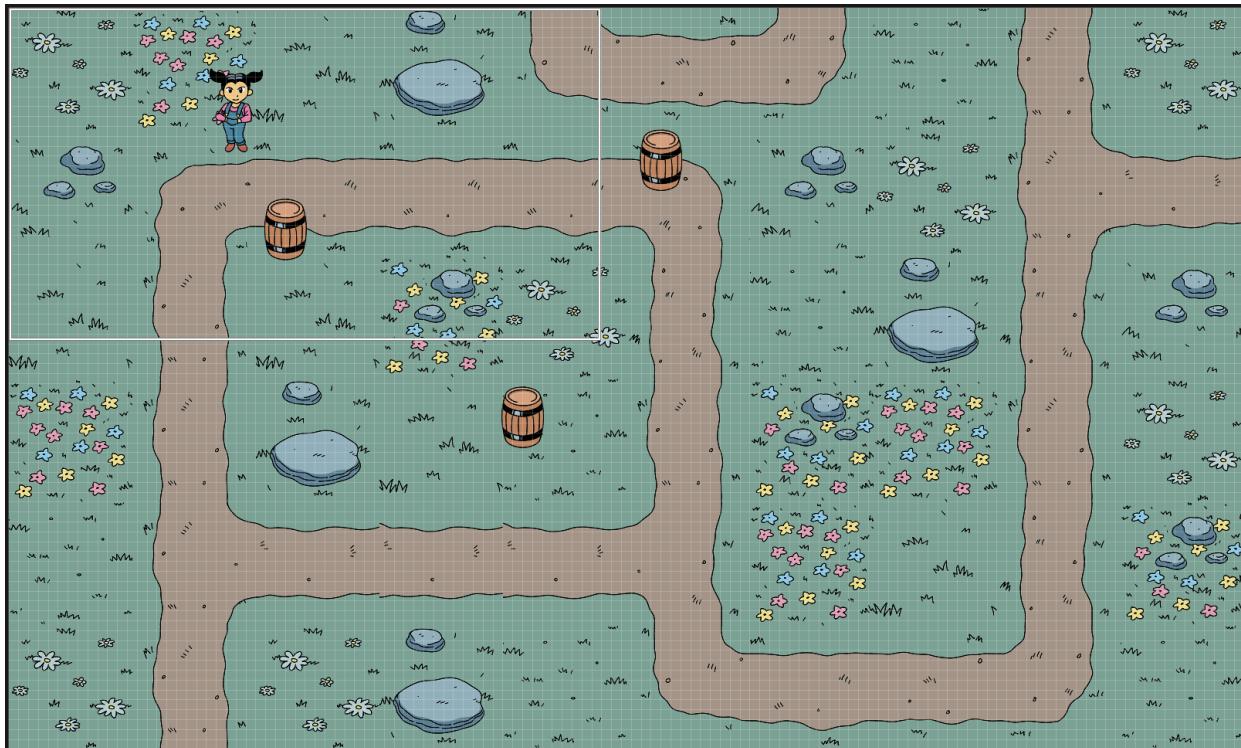
Now, in the Asset Browser, right-click and choose Create > Object. Name this new Object `obj_barrel01`. Click on the menu that says, "No Sprite," and choose `spr_barrel01`.

This is all good for now. Open our big room (`rm_gameMain`) again and click on the Instances layer to make sure it's selected.

In the Asset Browser, find our new `obj_barrel01` and drag it into the Room. (If you can't see your Asset Browser, look for the Assets tab; it might be paired with something else, or the Room Editor tab might still be in focus..)

	<p>Tip: Dragging an Object into a Room creates an <i>instance</i> of that Object, so you can place as many as you like. If you update an Object, all its instances are automatically updated too.</p>
---	--

Go ahead and drag one or two more barrels into the Room and position them wherever you like.



Multiple Instances of obj_barrel01 in our Room

When you're happy with your barrel placement, run the game again and walk around. Notice anything weird? You should have observed that:

- The barrels appear “flat” — we don’t seem to move in front or behind them correctly
- We can just walk through the barrels

We'll solve the first problem now and well return to the second problem during the next session.

4.21 Simple depth sorting

All Objects in a Room in GameMaker Studio 2 have a *depth* assigned to them. You'll also notice if you click on the various Layers in the Room Editor that those layers have depths too. You can read more about depth in the GameMaker Studio 2 manual, but here's an easy thing to remember: the *lower* the depth number, the *closer* the Object/layer/whatever is to your game's Camera.

So, an Object with a depth of 10 is closer (and thus, "in front of") an Object with a depth of 20. Depths can range from 16000 (all the way back) to -16000 (all the way front).

But our game can't magically determine which Objects are meant to be in front or behind other ones and when. We need to tell it that. So, we're going to apply a quick trick we call *depth sorting*.

Open `obj_player` and its Step Event. Create a new line after the other code blocks (at the bottom of the event) and add this new code:

```
// Depth sorting  
depth =-y;
```

This takes `obj_player`'s current y value in the Room (e.g., 500 pixels from the top) and reverses it to assign it a new depth, every step. So as the player moves *down* in the Room, its depth will change (-500, -501, -502, etc.). As it moves *up*, it will change in reverse (-500, -499, -498, etc.). This simple trick works well because our game has a top-down view.

However, this only works if we apply the same line of code to all *other* Objects in our Room for which we want to have correct depth sorting. In our case right now, those are our barrels.

4.22 Object parenting for easy editing

So we should add the same depth sorting code we just wrote for `obj_player` to our `obj_barrel01`, right? Well, not exactly.

You see, we also want to add other Objects based on those Sprite assets we imported — like fences, shop buildings, trees and so on. We haven't even made those Objects yet, but we know we're going to have to deal with them sooner or later.

So why not make life easy for us and use one of GameMaker Studio 2's best features: *Object parenting*. With this, we can make a "parent" Object and make our barrels, trees, fences, and whatnot its "children."

Whatever Events and code we create for our parent are then *inherited* by the children automatically (unless we say otherwise). This is a great way to organize Objects that are all meant to behave the same way and deal with them all at once.

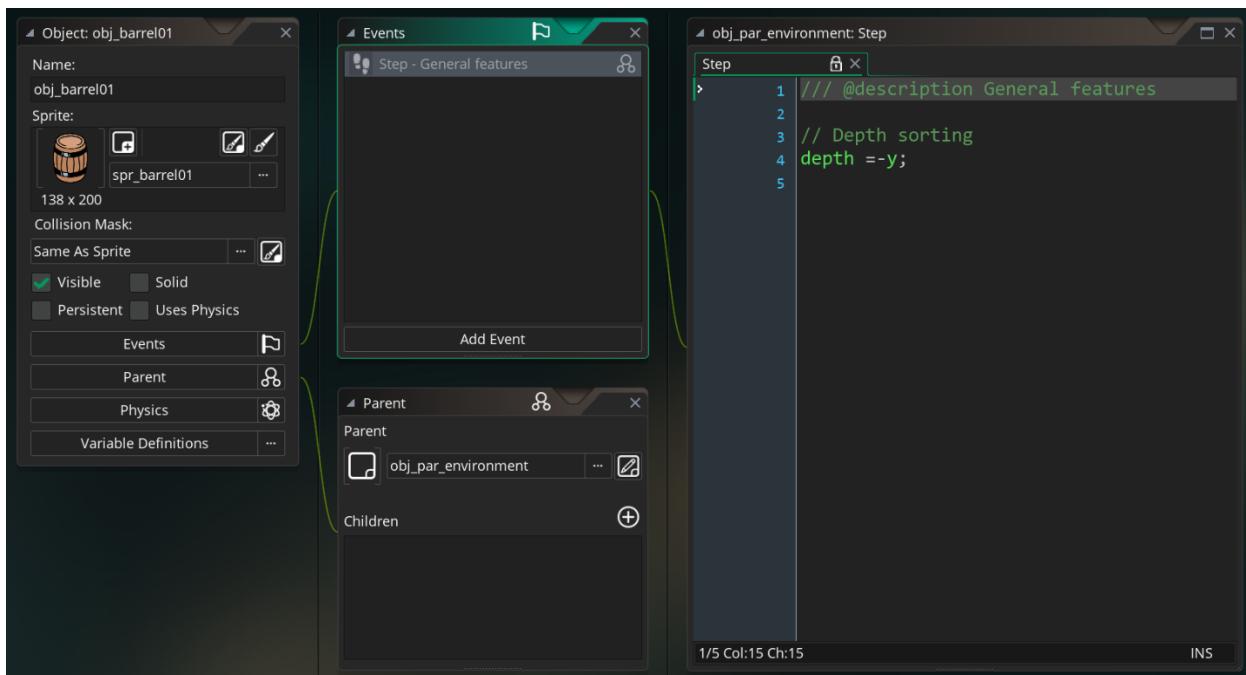
In the Asset Browser, right-click on the Objects group and choose Create > Object. Let's name this new Object obj_par_environment.

In the Object Editor, add a Step Event. Change the description to something that's helpful for you and add the same depth sorting code we wrote above:

```
// Depth sorting  
depth =-y;
```

Now, open obj_barrel01 again.

Notice in the Object Editor there's a button that says Parent. Click this and then No Object, and choose the obj_par_environment we just created.



Here, obj_barrel01 has obj_par_environment selected as its parent, and it has inherited the parent object's Step event

You'll notice that in the Events list for our barrel Object, a Step Event has appeared, but it's greyed out. This is the inherited Step Event from obj_par_environment. We can override this if we want, but for now just leave it be.

	<p>Tip: Notice how we didn't choose a Sprite for our parent Object? That's because we know we'll be doing so for all its children (like obj_barrel01).</p>
--	---

Now, run the game again and walk around those barrels we placed. *Voila!* The player should correctly appear in front of or behind the barrels, depending on its vertical position. Now *that* looks correct!



Now the player Object should appear correctly behind or in front of each barrel as it moves around

4.23 End of Session 1

That's it for this session. In the next one, we're going to deal with collisions (so we can't just walk through our barrels) and a whole lot more. Make sure to save your project!

5 Session 2

Welcome back! In this session, we're going to make Objects correctly collide with each other, create our townspeople, apply sound effects and music, control timing and Events with Alarms and more!

5.1 Creating solid Objects with collisions

If you recall, we can still walk right through our barrels, which is not the behaviour we want. In order to have one Object recognize another, we need to deal with *collisions*.

There are many ways to do *collision checks* in GMS2; we will explore a couple of them throughout this tutorial, but you can read about the other ways by looking up “Collisions” in the official manual. To start, however, we’re going to use a simple check: `collision_point`.

Open `obj_player` and open its Create Event.

Open `obj_player` and open its Step Event. Let’s edit first part of the `// If moving` code block, which currently looks like this:

```
| x += vx;  
| y += vy;
```

Replace these two lines with this new pair of if statements:

```
| if !collision_point(x+vx,y,obj_par_environment,true,true) {  
|   x += vx;  
| }  
| if !collision_point(x,y+vy,obj_par_environment,true,true) {  
|   y += vy;  
| }
```

`Collision_point` checks for a specific x and y spot for the Object in question. In this case, we’re looking ahead (using the same `vx` and `vy` variables we set up in Session 1) on each axis for `obj_par_environment`. Since `obj_barrel01` is a child Object of `obj_par_environment`, it too will be checked.

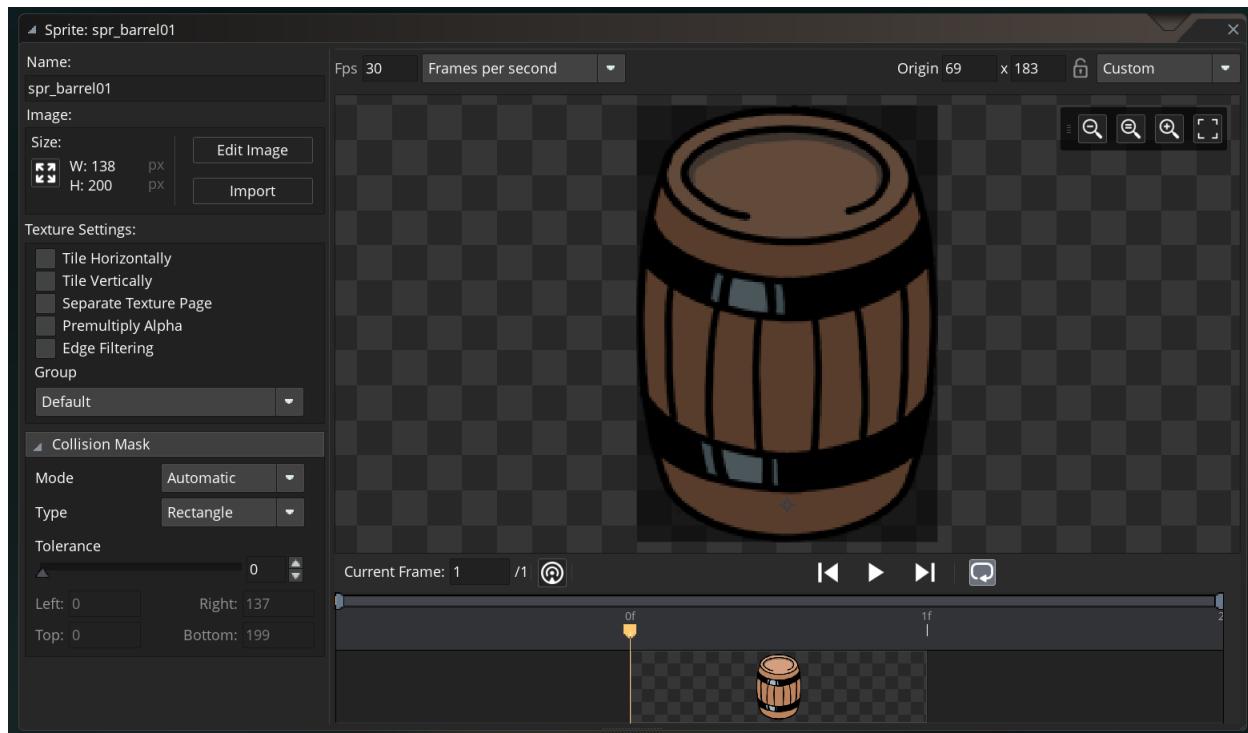


Tip: A “!” is a way to say “is not” in GML (and many other languages). By placing a “!” before the `collision_point` function here, we’re stating, “if this *is not the case*, then do something.”

5.2 Collision masks

When Objects check each other for collision they’re using what are called *collision masks*. These are invisible “hot spots” on each Sprite in the game.

Let’s look at a simple example. Open `spr_barrel01` from the Asset Browser. On the left of the Sprite Editor, you’ll see a section called **Collision Mask**. Click the arrow beside it to open this section.



The `spr_barrel01` Sprite with its default Collision Mask

You’ll notice that the Sprite now has a dark grey box around it. This is (as the name suggests) the *collision mask* for the current frame of the Sprite. When our player Object is checking for `obj_par_envrionment`, this is what determines if it’s found something or not. If the collision

mask of the current frame of the current Sprite for an environmental Object is that the precise point that `obj_player` is checking, then the collision is true.

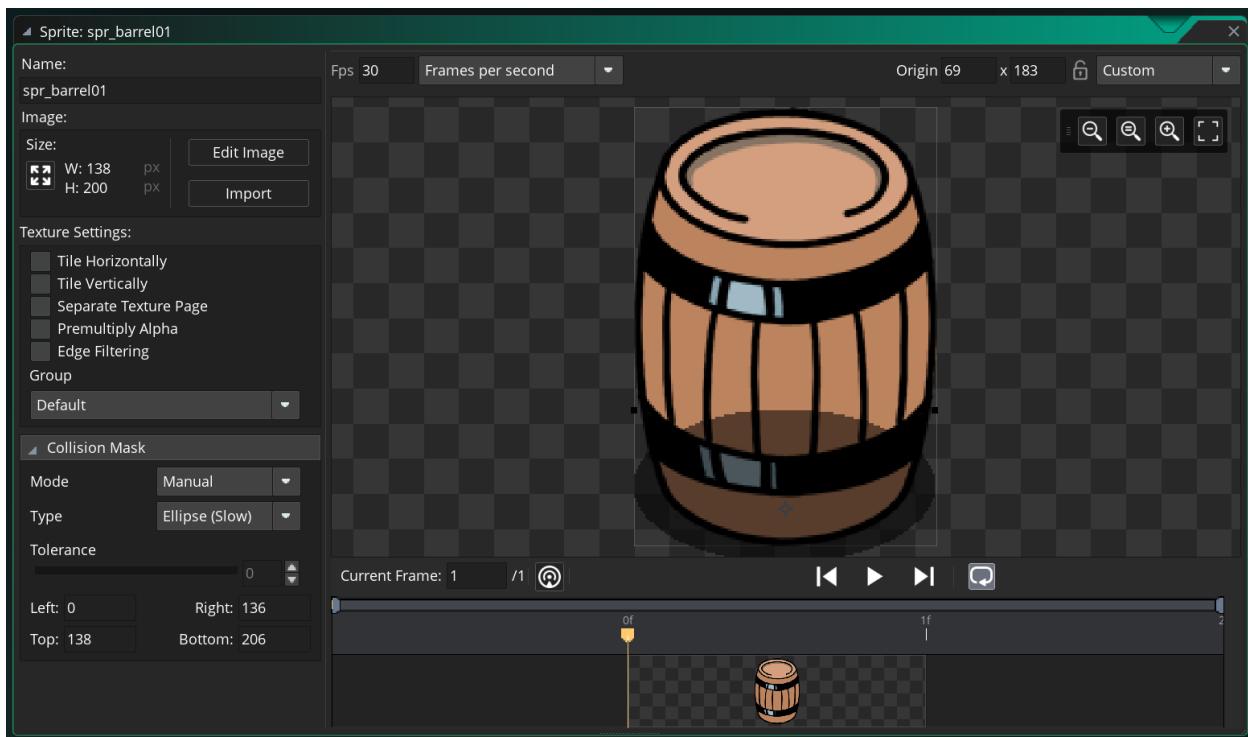
Every Sprite can have its own collision mask and every *frame* of a Sprite can have its own as well! But we're going to keep this simple. Right now, the Sprite's collision mask spans the whole image, but this isn't what we want.

Rather, we just want the collision mask for our barrel to be at its base. Since the player can walk in front and behind the barrel, it doesn't make sense visually for the player to collide with it if its feet, say, are touching the barrel's top.

So, in the Editor, click beside "Mode" and choose Manual. Beside "Type," choose Ellipse (Slow).

You'll notice that the dark grey Collision Mask has turned from a rectangle to an ellipse. If you click on this shape, you can adjust it. (You can also enter numbers for the four vertices of the Collision Mask at the bottom-left of the Sprite Editor.)

Adjust the mask so that you get a shape at the bottom of the barrel, like so:

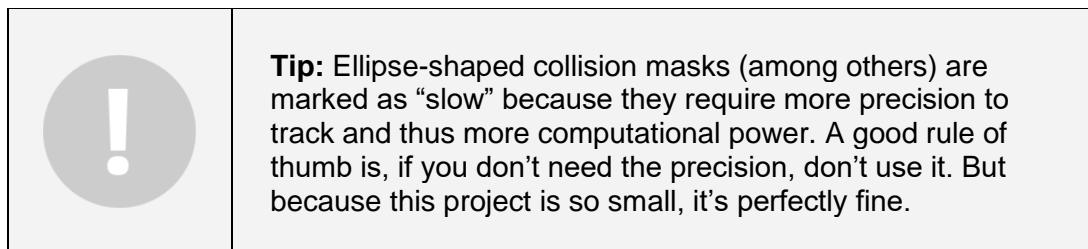


The Collision Mask is now correctly around the bottom of the barrel. It's okay if the mask extends beyond the barrel like this.

Once you've correctly edited the mask for the barrel, you can close the sprite.

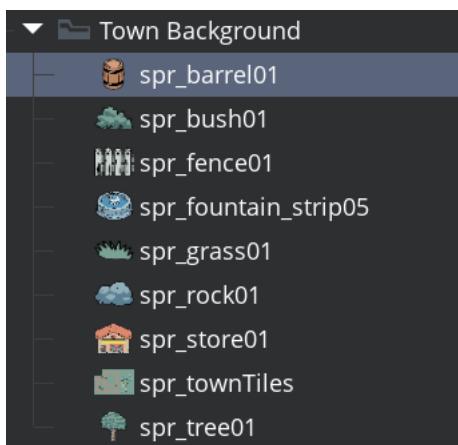
Run your game again to test the change. The player should correctly stop when moving up against a barrel, from any direction.

When you're ready, close the game window and return to GMS2.



5.3 Creating the other environmental details

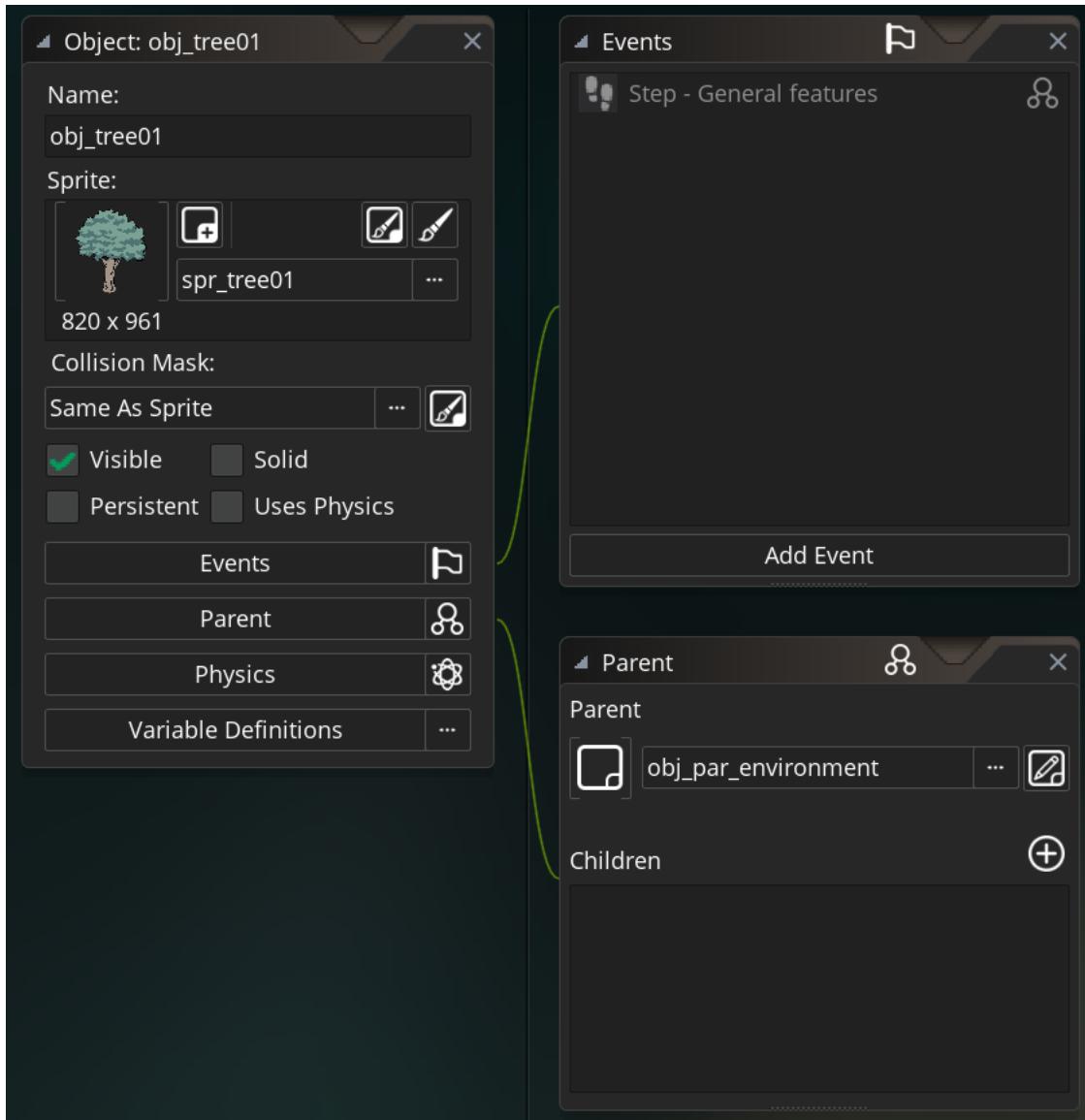
Now that we have a parent Object for our environmental details (`obj_par_environment`) and that our player Object can correctly track collisions with it, it's time to create the assortment of Objects we'll need to properly design our town.



We've already created our barrel Object, but now we're going to create Objects to utilize the rest of these environment Sprite we imported in Session 1.

Create a new Object and name it `obj_tree01`. In the Object Editor, click “Parent” and choose `obj_par_environment` as the parent Object.

Next, click beside the “Sprite” preview in the Object Editor, where it says “none,” and choose spr_tree01 to attach this Sprite to our new Object.

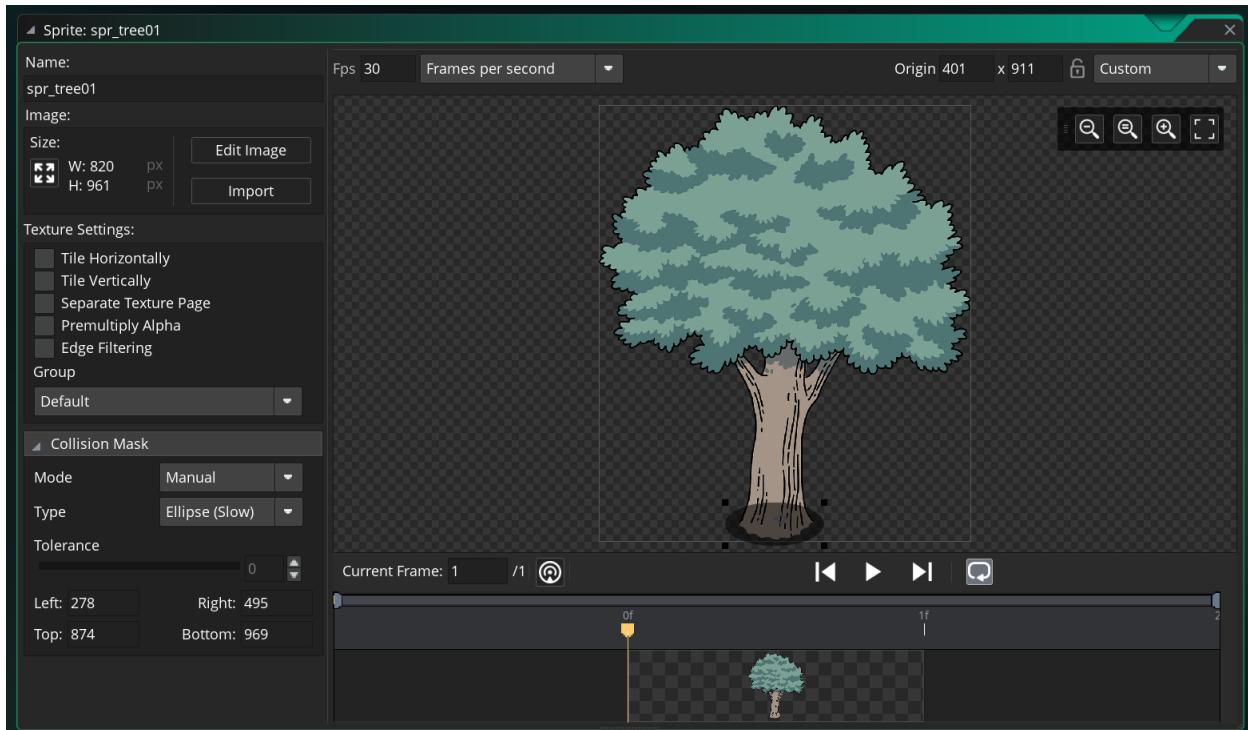


The correct setup for each of our new environment Objects: with obj_par_environment selected as its parent, and with the correct Sprite attached.

You'll also want to open the spr_tree01 Sprite and adjust its origin and collision mask, as we did with spr_barrel01.

In the Sprite Editor, make the origin at the base of the tree trunk. Open the “Collision Mask” section on the left of the editor and change the mask mode to Manual; for Type choose Ellipse (slow).

Just as you did with the spr_barrel01, adjust the elliptical collision mask for spr_tree01 to fit the bottom of the sprite.



Our spr_tree01 Sprite with correct, elliptical collision mask.

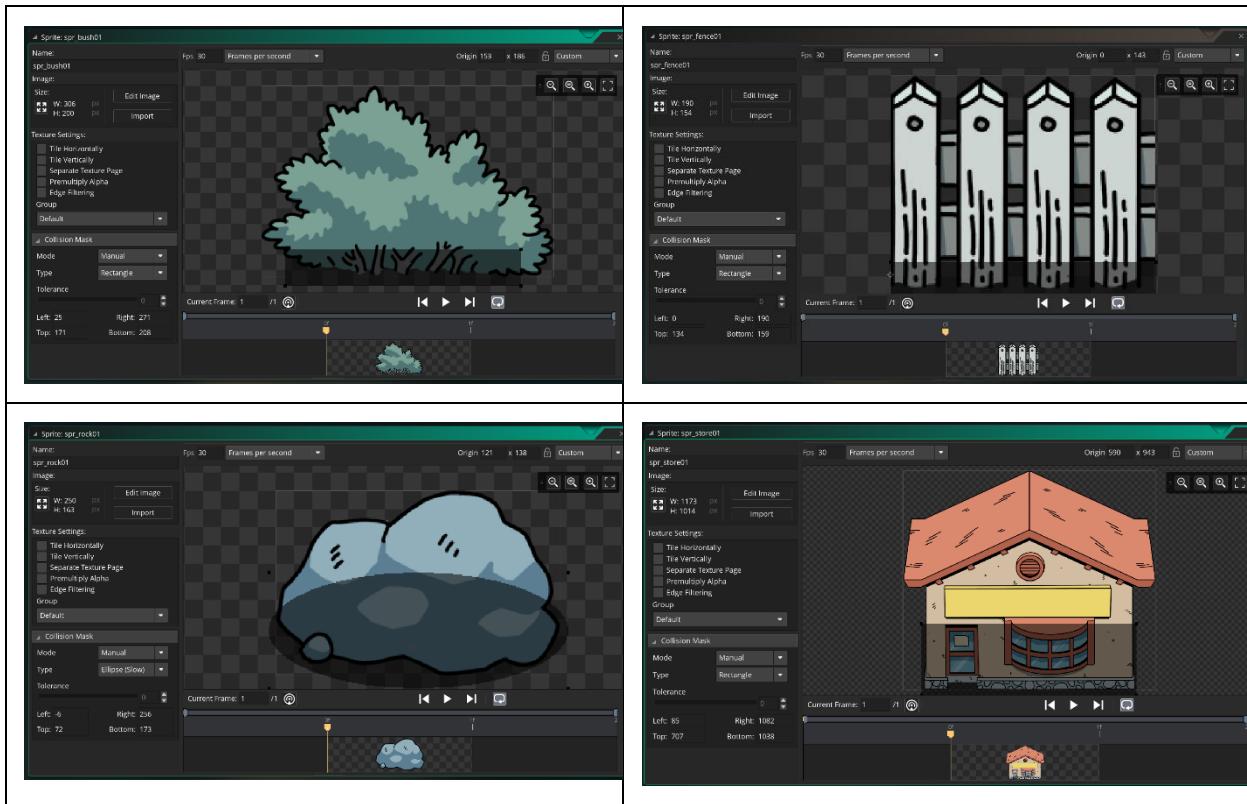
Repeat the above steps to make the following Objects:

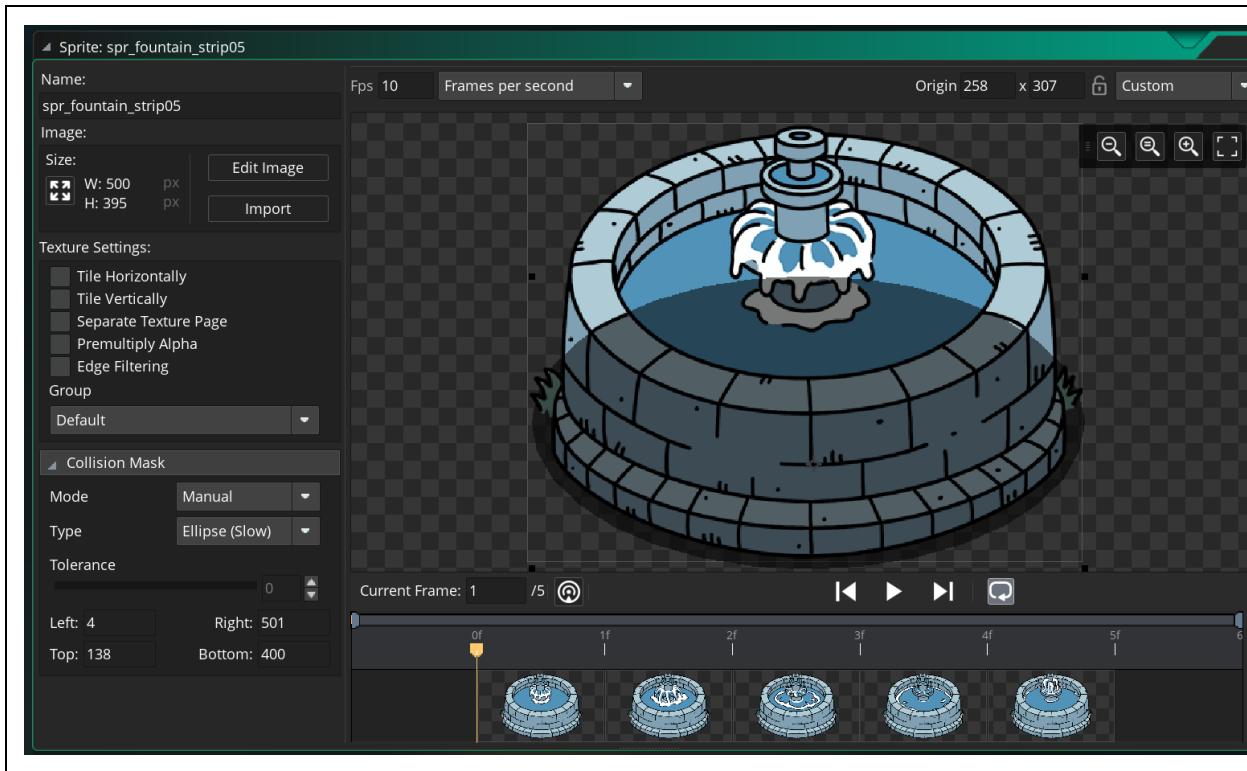
Object name	Sprite to attach	Collision Mask type to use
obj_bush01	spr_bush01	Rectangle
obj_fence01	spr_fence01	Rectangle
obj_rock01	spr_rock01	Ellipse (slow)
obj_store01	spr_store01	Rectangle
obj_fountain01 <i>(see below for further instruction)</i>	spr_fountain_strip05	Ellipse(slow)

- For each of these objects, make sure to:
- Select obj_par_environment as the parent Object
- Attach the correct Sprite
- Adjust the Sprite's Collision Mask

- Adjust the Sprite's Origin

Refer to these screenshots for details on Collision Mask and Origin recommendations:





5.3.1 A note about obj_fountain01

As you're creating the five town Objects we need, you'll notice that spr_fountain01 is a Sprite Strip, like the ones used for our player Object.

(If this Sprite Strip has not been correctly imported with multiple frames of animation, follow the steps in [Converting a Sprite Strip manually](#) to convert it.)

Make sure to also adjust the FPS setting for spr_fountain01 (we recommend 10).

When you're done, make sure to organize your new Objects into a group within the Asset Browser to keep things organized. (Right-click in the Asset Browser and choose Create Group, then rename it to something like Town Details.)

5.4 Filling out the town

We have an assortment of Objects now, so it's time to use them to design our town further and give it some personality.

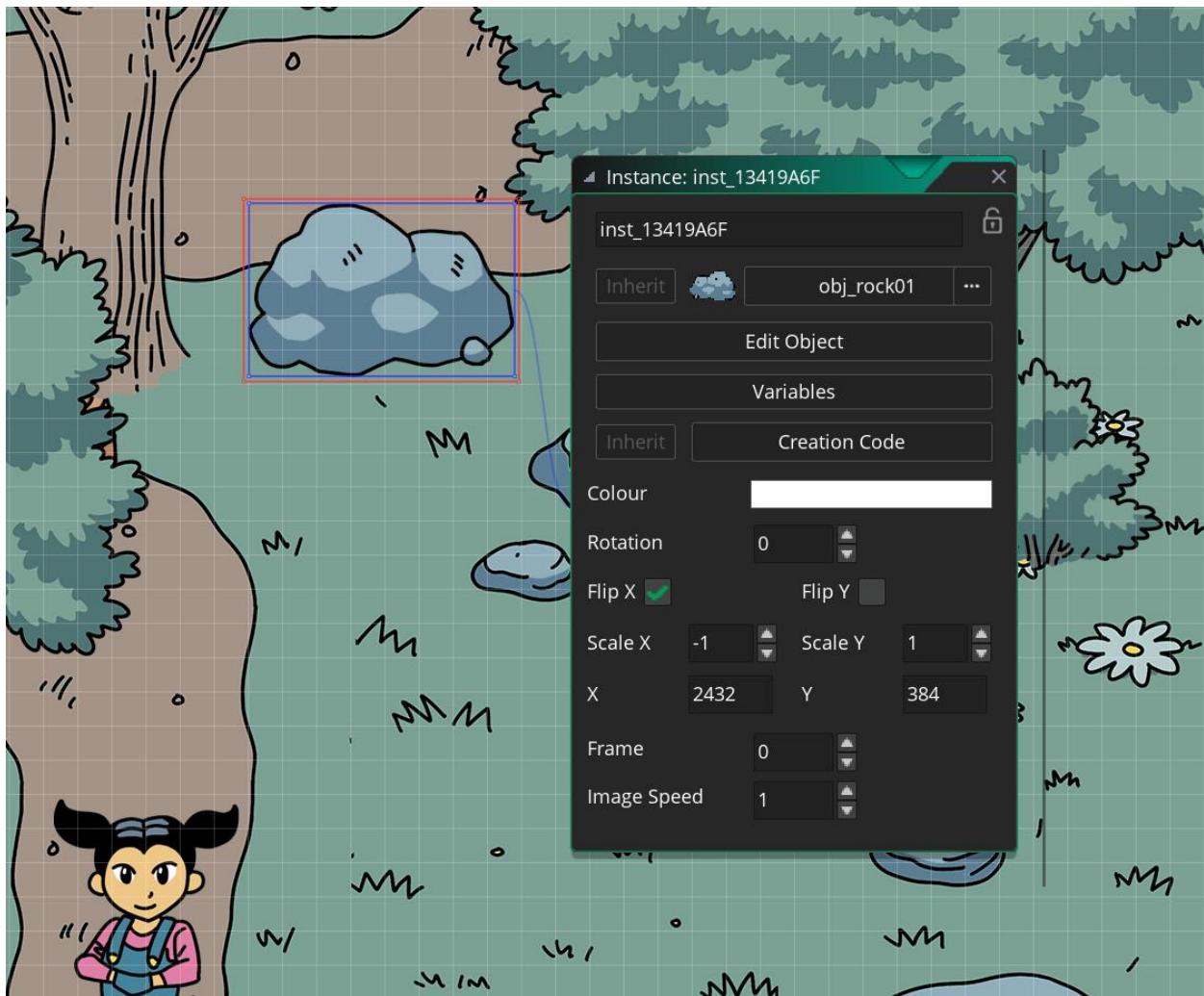
Open `rm_gameMain` and make sure the Instances layer is selected.

Drag in as many of these new Objects as you like to design your town. (We recommend you only put in three instances of `obj_store01` and one instance of `obj_fountain`, however.)

If you need to, feel free to re-arrange or delete the original barrels you placed into the Room (to delete an Instance from a Room, click on it and press Delete on your keyboard). You can also move `obj_player` around as well.

	<p>Tip: As you move Instances around your Room, you'll notice they're snapping to the Grid in the Room (unless you turned this off). If you want to move an Instance without it snapping, hold Alt (Windows) or Option (Mac) as you drag it around.</p>
---	--

5.4.1 Scaling and flipping placed Instances



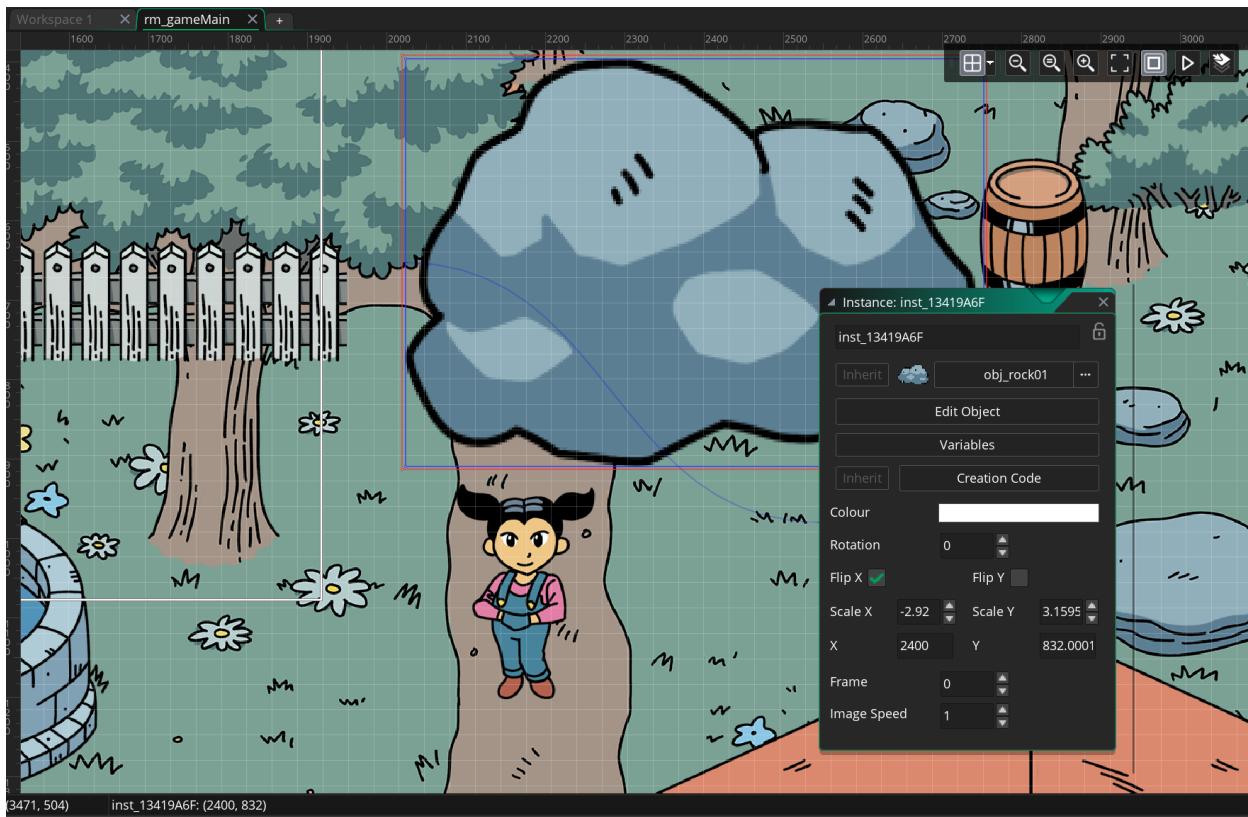
Using "Flip X" on an Instance of `obj_rock01`. Notice how "Scale X" is now -1.

If you want some variety, you can adjust an individual Instance of an Object within a Room. Double-click on any Instance (for example, `obj_rock01`) and you'll see Properties for that Instance.

There are many options here that we won't get into now, but notice the one labeled "Flip X." If you check this, that particular Instance will be flipped horizontally.

You'll also notice that if you check Flip X, the option below it, Scale X, changes from 1 to -1. If an Instance's Scale X or Scale Y setting is a negative number, it means it's been flipped.

You can enter numbers here to manually change the scale of an Instance, or you can click and drag on the sides or corners of an Instance within a Room to scale it. If you hold the Shift key while doing this, you can scale the Instance proportionately.



A rather enormous rock Object. You can see the exact scale of the Instance within its Properties (double-click on the Instance to bring this up)

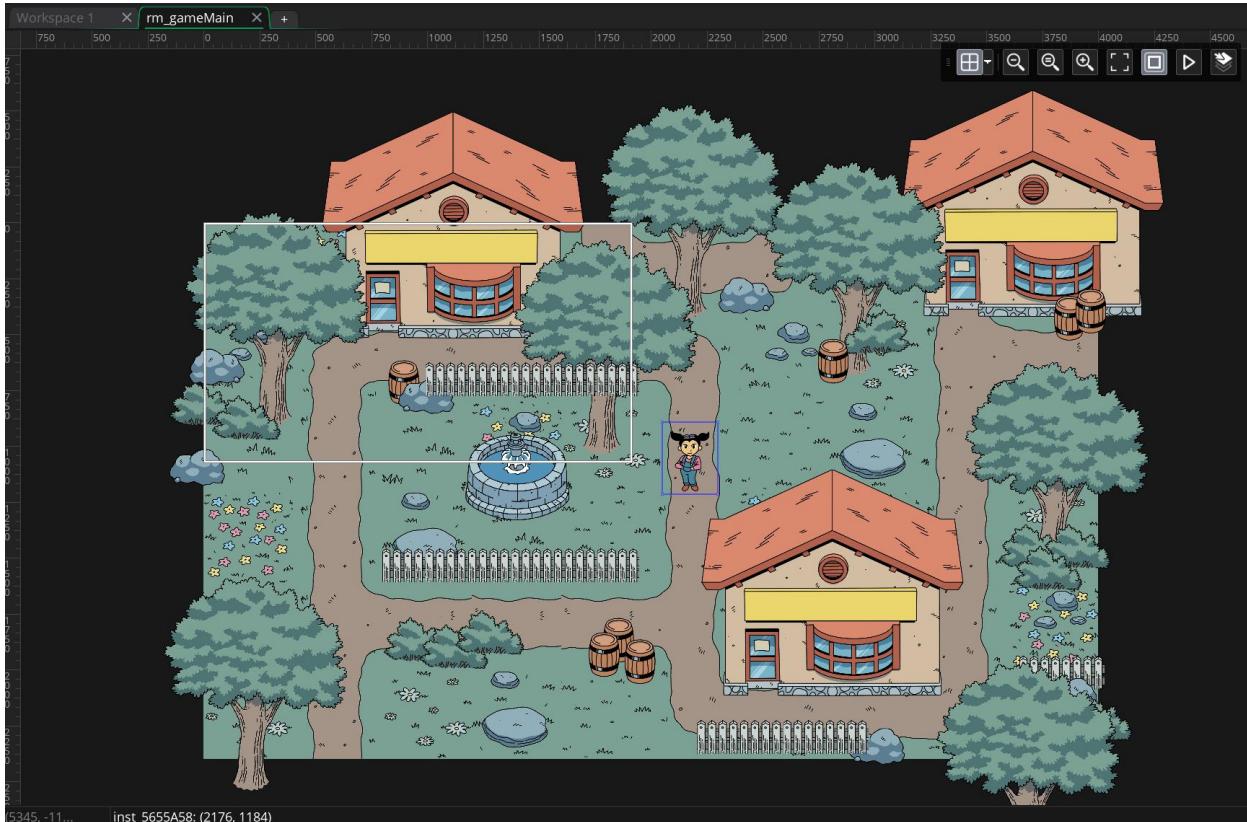
However, though flipping an Instance for variety is encouraged, scaling Instances in this way is not. (But hey, it's your town!) If you want to reset an Instance's scale, you can change its Scale X and Scale Y settings back to 1.

5.4.2 Adjusting Tiles

If you want to adjust the Tiles you placed, select the TilesMain layer and click on the Room Editor tab to see the Tile Set we used again. You can remove or change Tiles as needed.



Tip: As you move Instances around your Room, you'll notice they're snapping to the Grid in the Room (unless you turned this off). If you want to move an Instance without it snapping, hold Alt (Windows) or Option (Mac) as you drag it around.



Our designed town with three instances of `obj_store01` and one instance of `obj_fountain`. Note how some Instances don't appear to overlap correctly; this is okay.

You might notice as you're placing Instances in your Room and moving them around that your Objects overlap each other in odd ways.

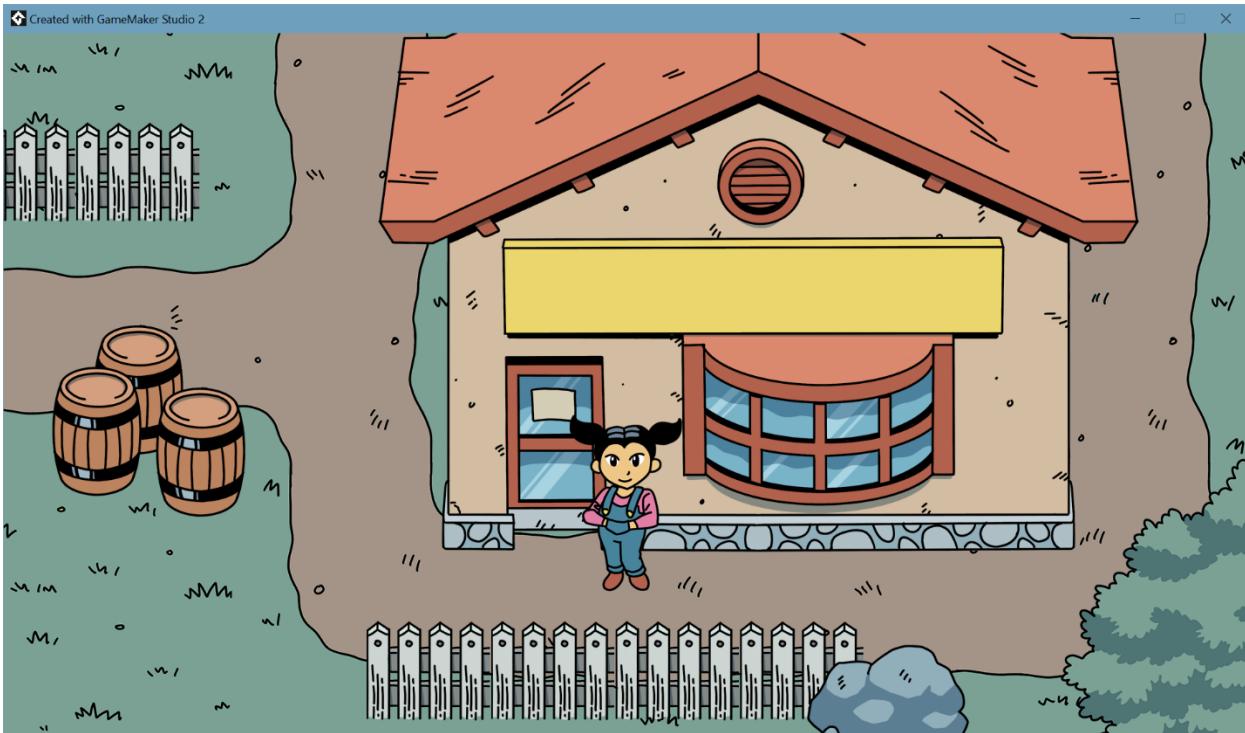
If you want to adjust this: make sure the Instances layer is selected in the Room Editor; in the Instance Layer Properties section, you'll see every instance you've placed in the Room. You can click and drag these up and down to change their sorting.

However, keep in mind: we already wrote that *depth sorting* code for `obj_par_environment` (back in [Simple depth sorting](#)). If your town Objects have `obj_par_environment` correctly selected as their parent Object, they will appear correctly when you run your game.

Instances Layer Properties - rm_gameMain		
<input checked="" type="checkbox"/>	 obj_player	inst_5655A58
<input checked="" type="checkbox"/>	 obj_barrel01	inst_5CC9358C
<input checked="" type="checkbox"/>	 obj_barrel01	inst_379132E3
<input checked="" type="checkbox"/>	 obj_barrel01	inst_4ECB4225
<input checked="" type="checkbox"/>	 obj_store01	inst_22B2530B
<input checked="" type="checkbox"/>	 obj_store01	inst_42ED4C2C
<input checked="" type="checkbox"/>	 obj_store01	inst_69698BED
<input checked="" type="checkbox"/>	 obj_fountain	inst_6F9D2029
<input checked="" type="checkbox"/>	 obj_bush01	inst_547E6424
<input checked="" type="checkbox"/>	 obj_bush01	inst_3326410F
<input checked="" type="checkbox"/>	 obj_bush01	inst_7890840E
<input checked="" type="checkbox"/>	 obj_tree01	inst_3E6AED0E
<input checked="" type="checkbox"/>	 obj_tree01	inst_39884AD4
<input checked="" type="checkbox"/>	 obj_tree01	inst_C6FA0B7
<input checked="" type="checkbox"/>	 obj_tree01	inst_752E865D

The Instance order for all those Objects. You can drag Instances up and down the list to sort them, but you don't really need to.

Once you've got your town looking the way you want it, run the game again walk around. Now this is more like it!



Walking around our newly designed town. Notice how objects are correctly sorted for depth.

5.5 Keeping the player in bounds

Something else you may have noticed while walking around your town is that our player Object can still trot right off the map, which we don't want.

There are many ways to deal with this, but let's do something simple and visual.

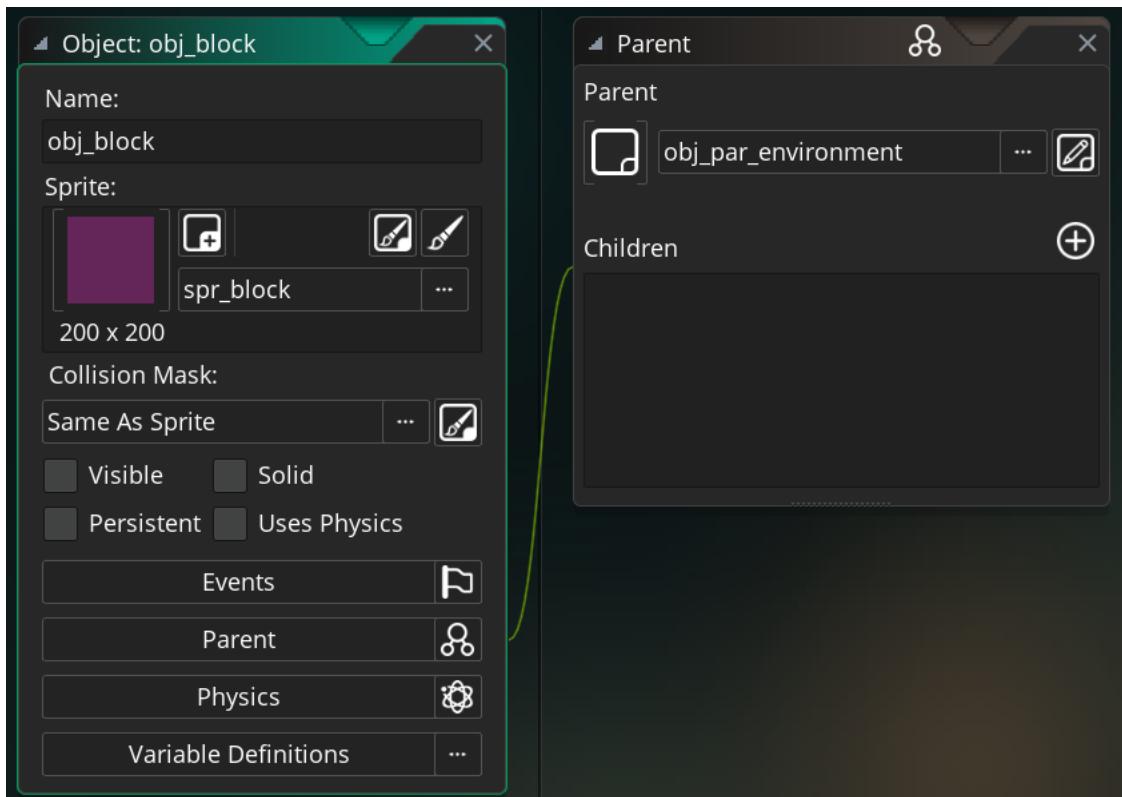
Using File Explorer (Windows) or Finder (Mac), navigate to the Assets folder provided with this course and open the Sprites > User Interface folder.

Drag the magenta spr_block image into the Sprites group in GameMaker Studio 2's Asset Browser.

In the Asset Browser, create a new Object and name it obj_block. In the Object Editor, attach the magenta spr_block Sprite we just imported.

We're going to use this Object as a quick-and-easy way to block off areas we don't want the player to access. To do this, do two things:

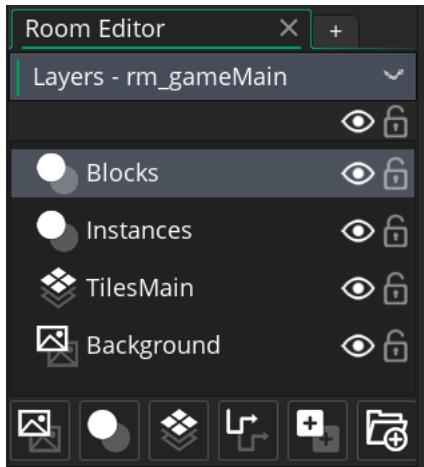
- In the Object Editor, click Parent and choose obj_par_environment
- Uncheck the Visible box



Setting up obj_block. Its “Visible” option is unchecked, and its parent is obj_par_environment

Open `rm_gameMain` again and look at the Room Editor panel, in the Layers section. We’re going to place instances of `obj_block` as invisible boundaries, but we want to do it on another layer. Why? Well, this just helps us keep organized if we need to tweak our Room later.

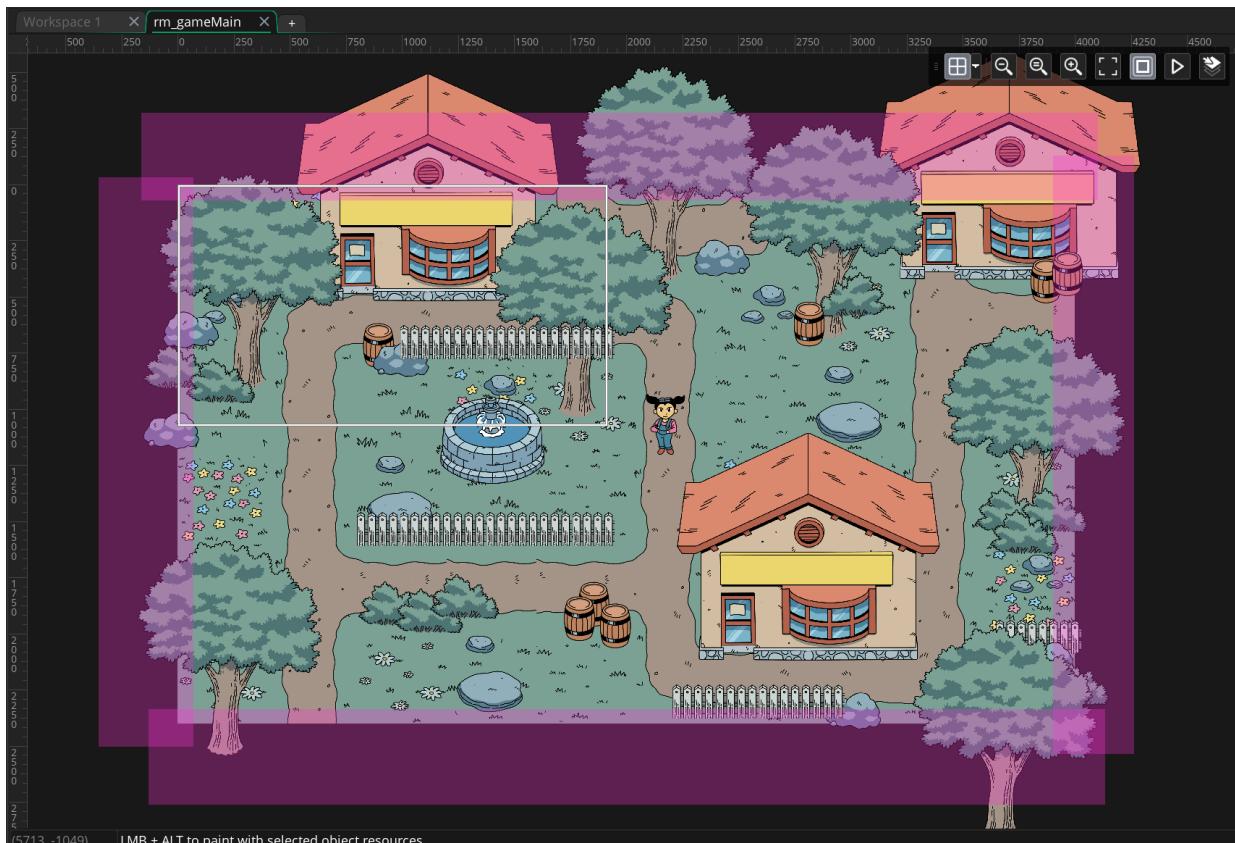
In the “Layers” panel, click the Create New Instance Layer (⊕) button and rename the layer it creates to **Blocks**. You can click and drag a layer to change its order (and thus its depth); make sure the **Blocks** layer is above the **Instances** layer.



Adding the Blocks layer and placing it on top

Next, make sure the Blocks layer is select. Then, drag in four instances of obj_block from the Asset Browser into the Room.

In the Room Editor, scale the four obj_block Instances and reposition them within the Room so that you create four “walls” blocking off the sides (see screenshot below).



Resizing the four obj_block Instances to create invisible “walls”

Run the game again and test. The player should now stay within the Room bounds. Since we made `obj_block` a child Object of `obj_par_environment`, it respects that collision code we wrote for our player Object.

Close the game window and return to Game Maker Studio 2 when you're ready. If you need to adjust any Instances or Tiles in `rm_gameMain`, feel free to do it now.

5.6 Creating the Baker character

With our town designed, we can now turn our attention toward the other characters who will populate it.

These characters are known as *NPCs* — non-playable characters — and our game will have three of them: the Baker, Teacher and Grocer. Let's start with the Baker.

In the Asset Browser, create a new Group and name it **NPCs**. Drag this new Group into the **Sprites** group.

Next, using File Explorer (Windows) or Finder (Mac), navigate to the Assets folder provided with this course and open the **Sprites > Characters and Items** folder. Drag the following Sprite into the new **NPCs** Group you made in the Asset Browser:

- `spr_baker_idle_down_strip04`

This is another Sprite Strip, and it should have imported so that it has four frames of animation. If it didn't, follow the steps in [Converting a Sprite Strip manually](#) to convert it. (You can use the dimensions in the below screenshot for your Frame Width and Frame Height.)



The Baker's idle Sprite Strip imported, and neatly organized in the Asset Browser.

5.7 Editing Sprite animations

Now, let's do something new: edit a Sprite inside GameMaker Studio 2 itself. If you click the Play button in the Sprite Editor, you'll see that our Baker is just bouncing up and down endlessly with a long blink in the middle, which isn't very interesting.

So, let's use the Sprite Editor to add frames, move them around and create a different kind of idle animation.

Place your cursor on the line between frames it changes to a double-arrow; you can click and drag to make frames longer or shorter. This doesn't change the content of the frames, just their duration.

You can also right-click on any frame (or shift-click multiple frames and then right-click) to copy them; then you can right-click again and choose Paste to paste the frames in the timeline.

With these techniques, alter the Baker's idle Sprite to be a little more interesting. Make sure the last frame is a bit longer to create a pause.

Just as we did with the Player Sprites last session, make sure to:

1. Set the origin point is down between the Baker's feet
2. Change the FPS setting to suit your edited animation (on our example, below, the FPS setting is 8)

When you're done, you can change the name of the Sprite to just `spr_baker_idle_down` to keep things simple (this won't affect the animation you just edited).



Our revised Baker idle Sprite. Note the “FPS” setting, and the placement of the Origin.

5.8 Thinking ahead with parent Objects

With that done, you can close all your Sprites and any other assets (remember, when you're in a Workspace, you can right-click on the background and choose Windows > Close all to clean it up). We need to create our Baker Object to continue, but hold on a moment — we need to make three characters, right?

Though these three characters are different, they're the same *kind* of Object: NPCs. And since we know all three are going to be the same kind of Object, we can think ahead here and do the same thing we did in Session 1: make a *parent Object* first.

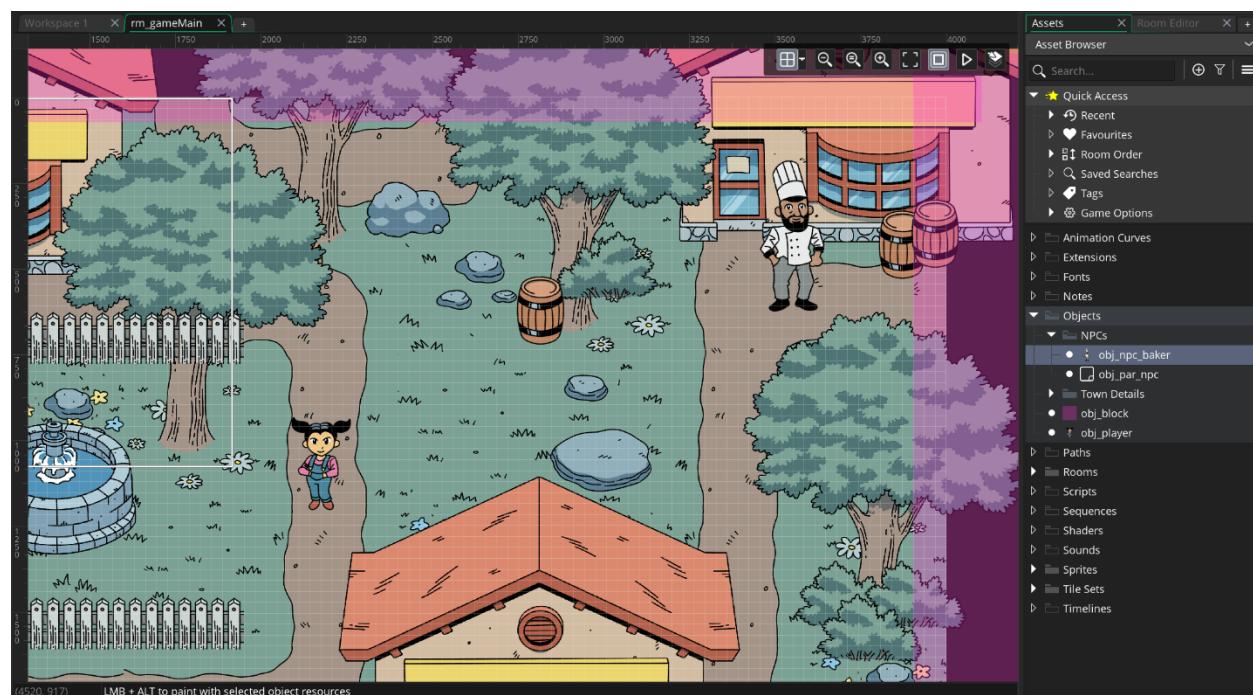
So, we're going to make the parent Object for all NPCs, then make our three characters from that. This will mean, just like it did with the environment Objects in our town, that we can do more with less and stay organized.

In GameMaker Studio 2's Asset Browser, right-click on the Objects group and create a new Object. Name it obj_par_npc. For now. We don't need to apply a Sprite or create any Events; we just need it to exist.

Now create another Object and name it obj_npc_baker. In the Object Editor, click "No Sprite" and choose our spr_baker_idle_down to apply it. Finally, click on Parent and choose our new obj_par_npc as a parent Object.

With that, our Baker NPC is now a child Object of obj_par_npc, which means we can make changes to the parent and the Baker will inherit those changes.

Open rm_gameMain and place an instance of the Baker into the Room, in front of one of the shops. Run the game and watch him do his thing.



Placing the Baker Object next to one of the shops in our Room. Notice how we've organized the NPC objects in the Asset Browser.

You'll notice a couple of things:

- His animation always loops at the exact same speed
- He doesn't respond correctly to depth (like our environment Objects do)

So next, we're going to use the parent-child relationship between Objects to make our NPCs more unique and livelier, and to fix that depth issue.

5.9 Variable Definitions

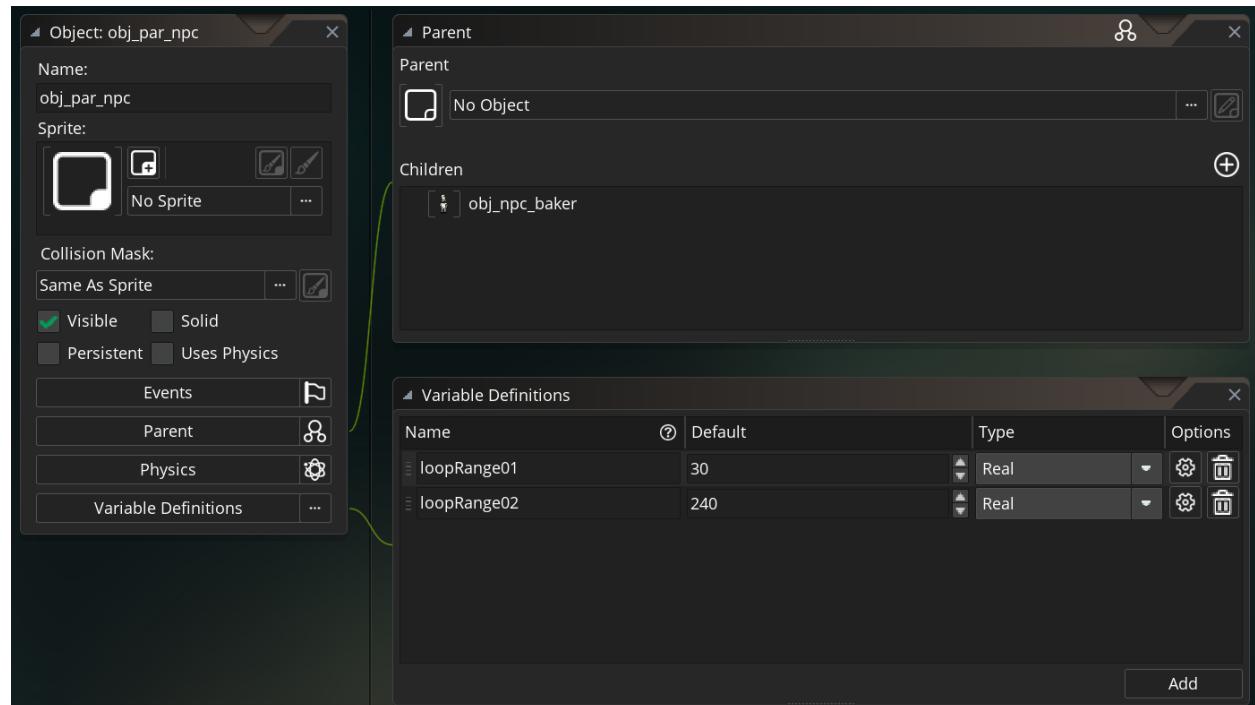
Open our obj_par_npc Object again and click the button labeled Variable Definitions.

Variable Definitions let us set up predefined variables of many kinds and allow us to change the value of those variables in child Objects, or even in *Instances* of those child Objects placed in a Room. With this feature, we can set up things we'll need (like animation speed, or strings of text) for easy access. It's a powerful feature that lets us keep track of things visually.

Let's add some new variables here by clicking the Add button. Enter the variable name as loopRange01 and change the Default to 30.

You'll notice there's a drop-down menu labeled Type; here you change what kind of value you want to enter for each variable (like Real numbers, Integers, Strings and so on). If you roll-over the "?" tooltip icon beside each type, you can learn more. For now, though, keep the Type set to Real (since we're entering numbers).

Add a second variable called loopRange02 and set its Default to 240.



Adding Variable Definitions in the Object Editor

5.10 Controlling animation with Alarms

We're going to use a combination of the Create Event, our Variable Definitions and a new kind of Event called *Alarms* to make our NPCs' idle animations a little more interesting.

So, in obj_par_npc, let's add a Step Event. In the Object Editor, click Add Event and Choose Step > Step.

In this Step Event, add the following code:

```
// Random loop timing
if (image_speed > 0) {
    if (image_index == image_number) {
        image_speed = 0;
        alarm[0] = irandom_range(loopRange01,loopRange02);
    }
}

// Depth sorting
depth -=y;
```

You already know what the // Depth sorting code block does; it's the same as what we wrote for our environment Objects. This will make sure our NPCs correctly appear behind and in front of other Objects in the town.

But above that, in the // Random loop timing code block, is the fun stuff. Let's break it down:

// Random loop timing if (image_speed > 0) {	Image_speed is a built-in variable that multiplies the "FPS" setting for a Sprite. So, an image_speed of 1 would give you 1x speed, 0.5 would give you half and so on.
if (image_index == image_number-1) {	Here, we're checking if the current frame in our Sprite (image_index) is the last one (image_number-1 being the total).
image_speed = 0;	And if so, let's set the image_speed to 0 (so it no longer animates)
alarm[0] = irandom_range(loopRange01,loopRange02); }	And then, let's set an Alarm with a random integer (more on this below)

Notice how in the code above, we're using `loopRange01` and `loopRange02` — these are the Variable Definitions we set up in our parent Object. In this code, we're setting an Alarm to a random value between these two numbers (which, by default, we set to 30 and 240, respectively).

5.11 Our first Alarm

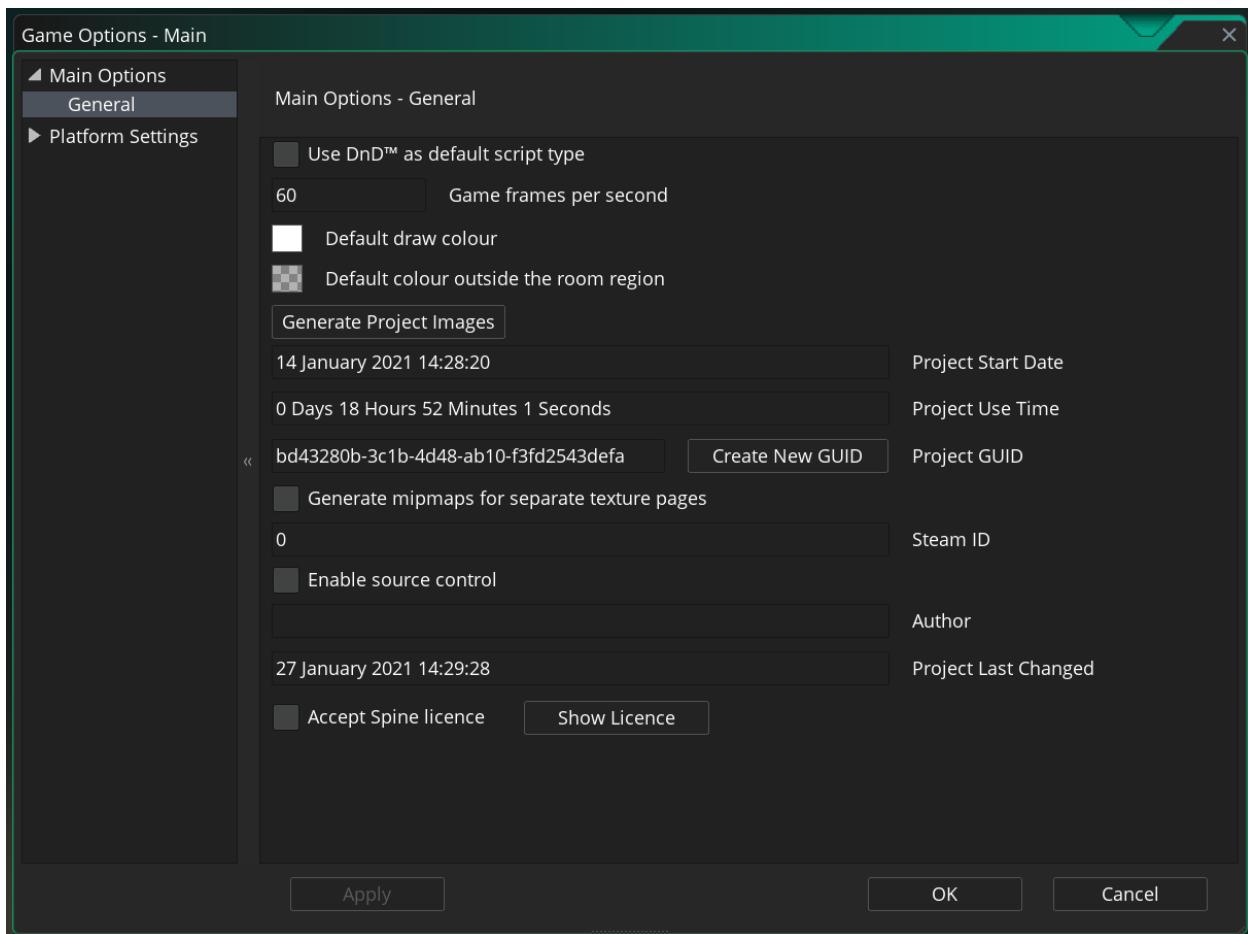
So, what does an Alarm do anyway? Well, as the name suggests, it's an Event that you can set (and reset) any time you like; every time an Alarm "goes off," it runs whatever code you've written there.

The number you set an Alarm to is a countdown in *steps* — so setting an Alarm to 60 means that after 60 steps, it will "go off."

5.11.1 A quick note on steps and game speed

Your game has several options associated with it for various platforms, as well as a general set. You can find this by going to the Asset Browser and expanding Quick Access.

Here, you'll see whatever options are available to you (depending on which platform modules to which you have access). Click on Main to bring up the Game Options – Main window:



The main Game Options window, with a Game frames per second setting of 60.

In these options is the Game frames per second, which by default is set to 60. This is the frame rate that your game attempts to maintain as it's being played — in this case, 60 *frames per second*, or FPS. So, a step in our game is one of those frames per second.

This *not* a real-time value per se; if your game gets bogged down with too many objects or poorly optimized code, it can slow down. Keep this in mind when setting Alarms.

5.11.2 Back to our Alarm

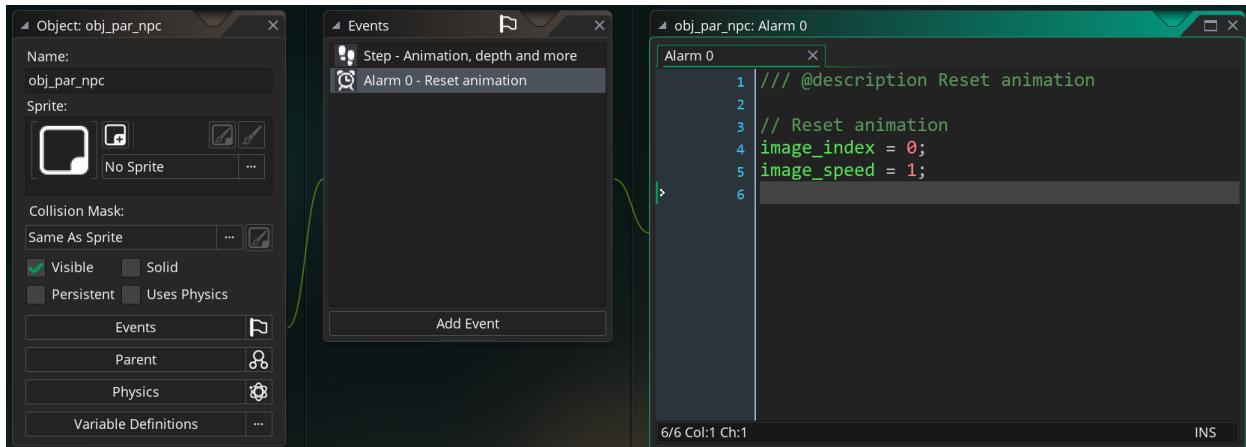
Right now, we're going to make a simple Alarm that randomizes our NPCs' idle animation to give them a bit more personality.

In obj_par_npc, add an Event and choose Alarms > Alarm 0. (GameMaker Studio 2 gives you twelve possible Alarms per Object — 0 through 11 — so we're just going to choose the first one.)

In Alarm 0, write the following code:

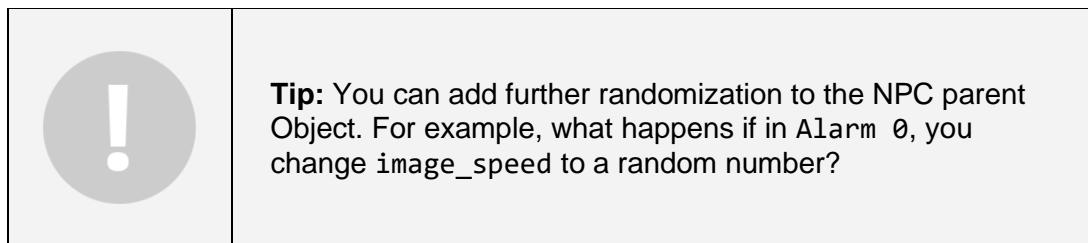
```
// Reset animation  
image_index = 0;  
image_speed = 1;
```

This will reset the animation back to the first frame (frame 0) and that `image_speed` multiplier back to 1, so the animation will play again.



Creating our first Alarm.

Run the game again and pay attention: now the Baker's idle animation will restart at different intervals, giving him a bit more life. And as the player Object moves around him, you'll see it correctly appears in front and behind our bouncing Baker.

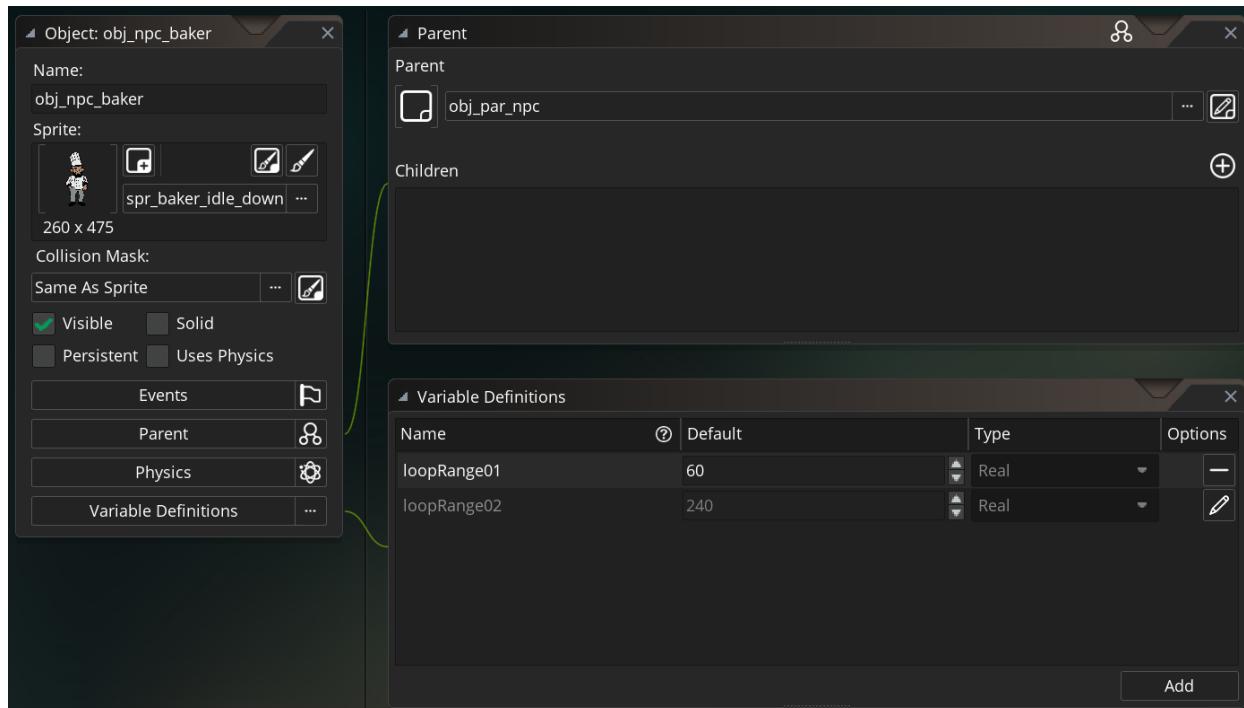


5.12 Using those Variable Definitions

We set up Variable Definitions in `obj_par_npc` for two variables, but how do we use them and what do they do?

Open obj_npc_baker again and click on “Variable Definitions.” You’ll see the Definitions inherited from the Object’s parent (in this case, obj_par_npc). If they are unchanged from the parent Object, they will appear greyed out.

Click the pencil icon beside loopRange01 to change its value; try changing it to 60. Do the same for loopRange02 and change the default value to 320.



Editing the inherited Variable Definitions in obj_npc_baker. Notice how a Definition that we’ve edited here appears differently than one we haven’t yet.

Run the game again and pay attention to the Baker; you’ll notice the pause at the end of his idle animation is still random, but the pause is longer.

Close the game window and return to GameMaker Studio 2. Feel free to return to obj_npc_baker and change these values further or click the minus button beside a Definition to restore it to the parent Object’s default.

	<p>Tip: Variable Definitions can be changed in an <i>Instance</i> of an Object as well once it’s been placed in a Room. You could place three Baker Objects in the same Room and each of them could have different values for loopRange01 and loopRange02. To do this, double-click on an instance in a Room to open its properties; then click Variables, and</p>
--	---

	then change the values here. The values in an Instance override the values in a child Object, which override the values of its parent Object.
--	---

5.13 Populate the town

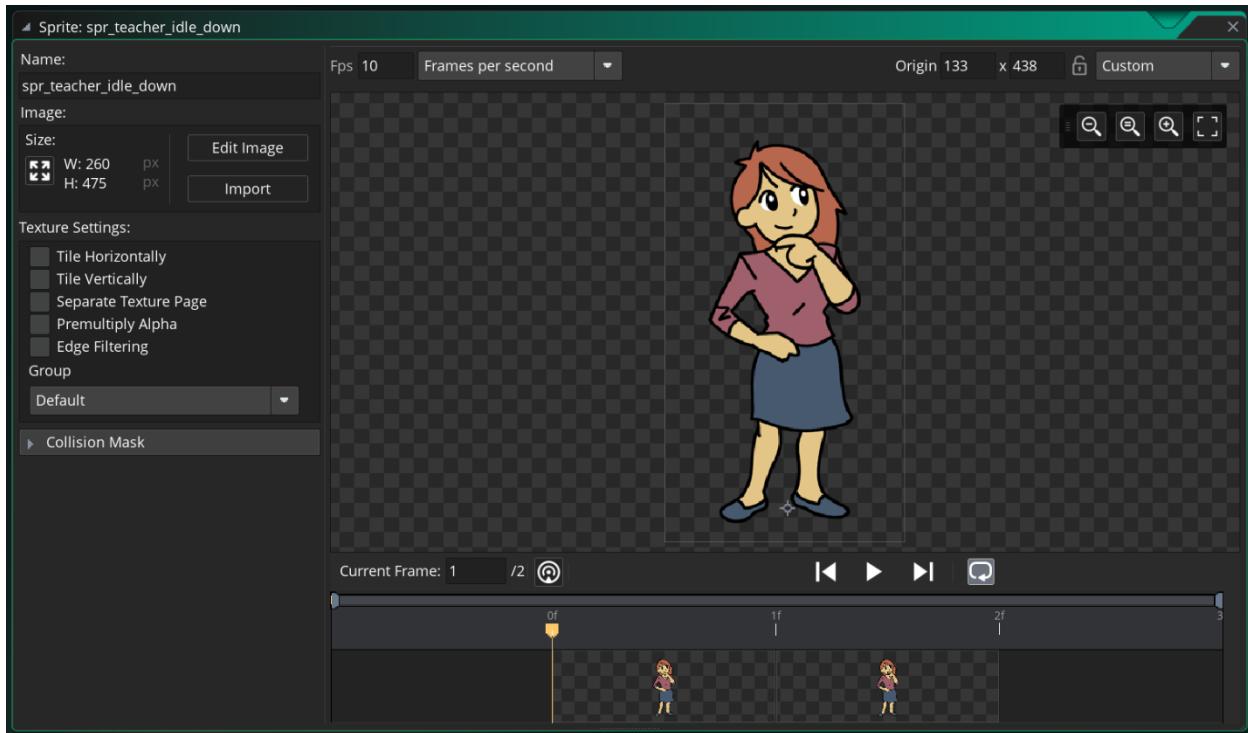
Now that we have a basic NPC parent set up, we can create our other two characters, the Teacher and the Grocer.

Using File Explorer (Windows) or Finder (Mac), navigate to the Assets folder provided with this course and open Sprites > Characters and Items. Drag the following assets into the Sprites > NPCs group in GameMaker Studio 2's Asset Browser.

- spr_teacher_idle_down_strip02
- spr_grocer_idle_down_strip02

Open each Sprite and:

- Set its origin as before (down between the character's feet). Remember, you can always adjust it later if it doesn't look quite right
- Change the FPS to 10
- Remove the _stripXX suffix from each asset name
- Ensure the Sprite Strip has been correctly converted into frames of animation. If it hasn't, follow the steps in [Converting a Sprite Strip to frames of animation](#) to do so



The two-frame Teacher idle Sprite, automatically converted to frames of animation.

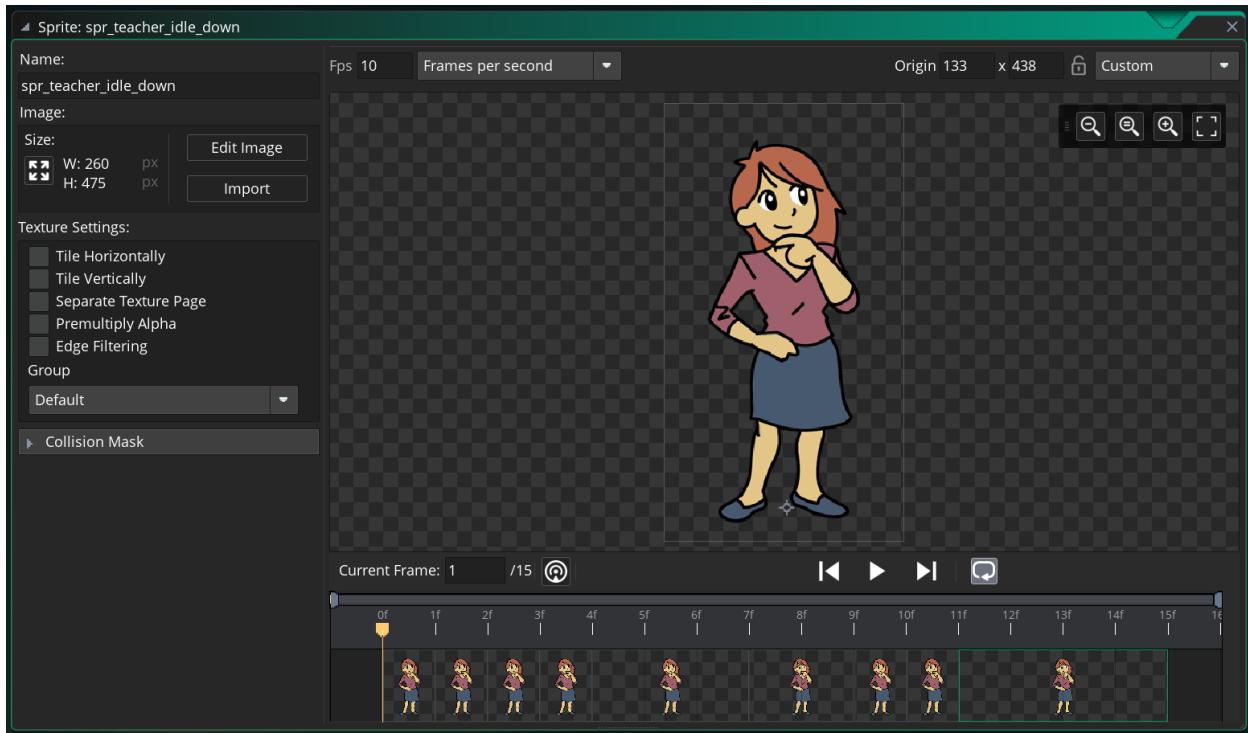
Right-click in the Asset Browser and create a new Object. Name it obj_npc_teacher and attach the spr_teacher_idle_down Sprite to it.

In the Object Editor, click Parent and choose obj_par_npc to make the Teacher a child Object.

For our Teacher Object, let's do two things:

1. Edit its idle Sprite the same way we did with the Baker
2. Change the values of its Variable Definitions

First, open spr_teacher_idle_down again and edit the animation as we did with the Baker. Right now, the Sprite is just two frames of animation. Play around here however you like to add a different cadence to the Teacher's animation, and make sure to stretch the final frame out so there's a pause.



The Teacher Sprite again, edited to have more frames of animation.

Next, open `obj_npc_teacher` and click **Variable Definitions**. Change the values for `loopRange01` and `loopRange02` to something other than the defaults.

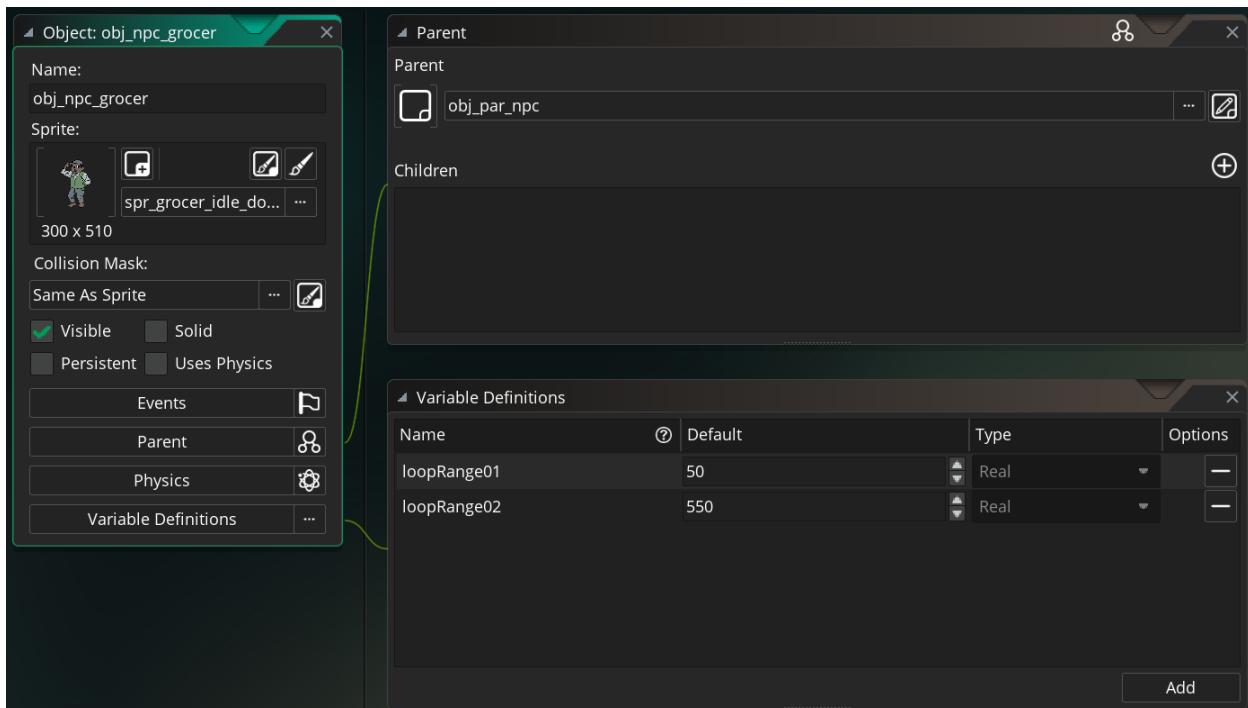
The screenshot shows the Construct 3 object editor for the object `obj_npc_teacher`. On the left, the properties panel shows the object's name, sprite (260 x 475 pixels), collision mask (Same As Sprite), and variable definitions section. A green arrow points from the 'Variable Definitions' section in the properties panel to the 'Variable Definitions' panel on the right. The 'Parent' panel shows the object is a child of `obj_par_npc`. The 'Variable Definitions' panel lists two variables: `loopRange01` (Default: 90, Type: Real) and `loopRange02` (Default: 400, Type: Real). An 'Add' button is at the bottom right of the panel.

Name	Default	Type	Options
loopRange01	90	Real	[minus]
loopRange02	400	Real	[minus]

Editing the Variable Definitions for obj_npc_teacher

Create another Object and name it obj_npc_grocer. Attach the spr_grocer_idle_down Sprite to it. Just like above, make the Grocer Object a child of obj_par_npc.

In the Object Editor, click Variable Definitions and change the defaults here to be different values (they don't have to be the same as the values you chose for the Teacher Object).



Editing the Grocer: we've chosen obj_par_npc as its parent, and have edited the inherited Variable Definitions

Next, open spr_grocer_idle_down again and repeat the steps we took above for the Teacher Sprite to enhance the Grocer's idle animation; again, make sure the last frame of animation has a pause.

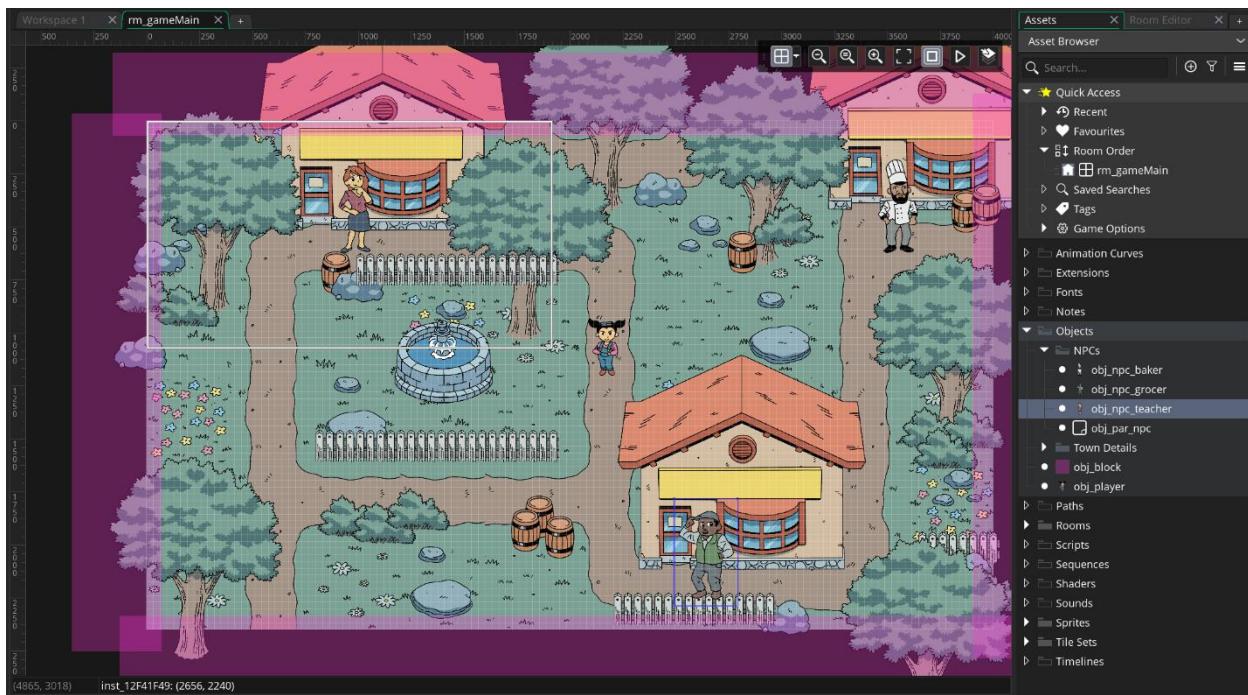


Editing spr_grocer_idle_down to have more frames of animation, and different timing

Once you have both the Teacher and Grocer ready, Go to `rm_gameMain` again.

Make sure you have the Instances layer selected and drag a single Instance each of `obj_npc_teacher` and `obj_npc_grocer` into the Room.

Position the two NPCs in front of the two unoccupied stores; now our town is populated!



Placing the Grocer and the Teacher into our town Room.

5.14 Detecting the NPCs

We're well on our way to having a nice, lively town. But we're going to need a few more pieces to make it all work.

To start, our three NPCs are just standing there right now, not doing much of anything; we're going to want to interact with them. But to do that, our player Object first needs to be able to even notice that they're there.

Thankfully, we already have some idea how to do this; we learned how to check for one Object from another in [Creating solid objects with Collisions](#).

Open `obj_player` and its Create Event. Add these lines to the code already there:

```
nearbyNPC = noone;
lookRange = 30;
```

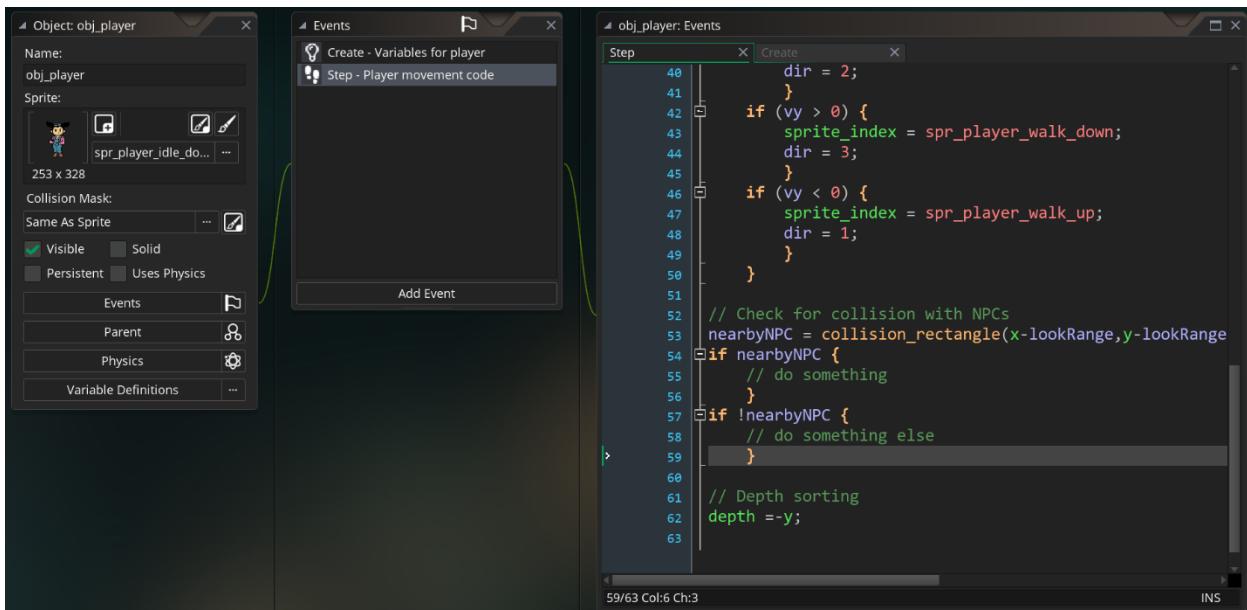
Now, open `obj_player`'s Step Event. Find or make a new line at the bottom, right above the `// Depth sorting` code block. Add the following new code block:

```
// Check for collision with NPCs
nearbyNPC = collision_rectangle(x-lookRange,y-
lookRange,x+lookRange,y+lookRange,obj_par_npc,false,true);
```

```

if nearbyNPC {
    // do something
}
if !nearbyNPC {
    // do something else
}

```



Add the new code block to obj_player. (The Event window here is narrow, so the code continues off the right side.)

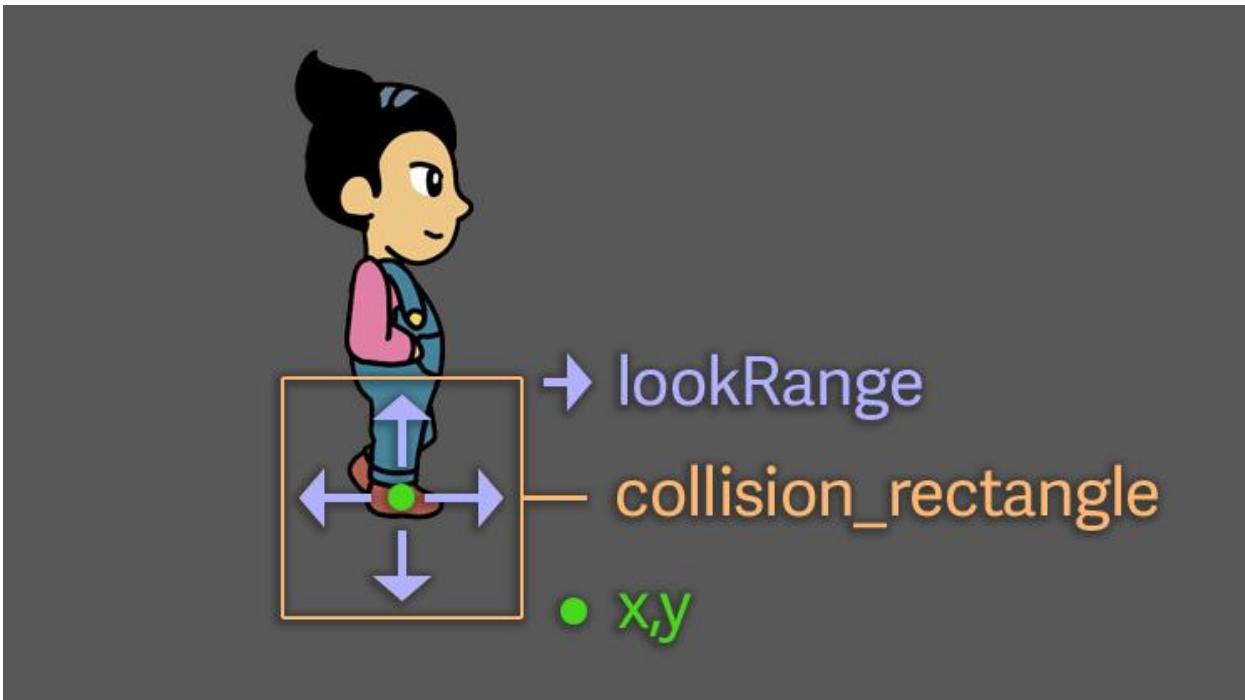
Let's go over what we're doing here:

In the Create Event, we declared the new variable `nearbyNPC` and set it to `noone`, which is a built-in value in GameMaker Studio 2 that means exactly what it says. We also created a new variable called `lookRange`.

In the Step Event (remember, every single frame of the game), we are using a *collision check* to look for an Object (`obj_par_npc`).

This collision check differs from the `collision_point()` check we used before in that it doesn't check a specific point, but rather a rectangular area.

The area we're checking is a rectangle that uses `lookRange` to determine its size. Make `lookRange` a larger or smaller number and the range that our player Object checks for NPCs grows or shrinks.



With `collision_rectangle()`, we're checking an area surrounding our player Object's `x` and `y` position. The area is defined by the value `lookRange`

(If you want to know more about the `collision_rectangle` function, refer to the GameMaker Studio 2 manual.)



Tip: If you see a line of code that says something like `someVariable = fancyFunction()`, it means we're using a function (`fancyFunction`) and storing its result immediately in a variable (`someVariable`). It's just a handy way to combine two steps in one.

Below the `nearbyNPC` collision check you'll see two if statements:

```
if nearbyNPC {  
    // do something  
}  
if !nearbyNPC {  
    // do something else  
}
```

Since we passed the result of our collision function immediately to the variable nearbyNPC, we can quickly check if it found something (`if nearbyNPC`) or if it did not (`if !nearbyNPC`).

But how can we confirm that this works? Well, that's where *debug messages* come in.

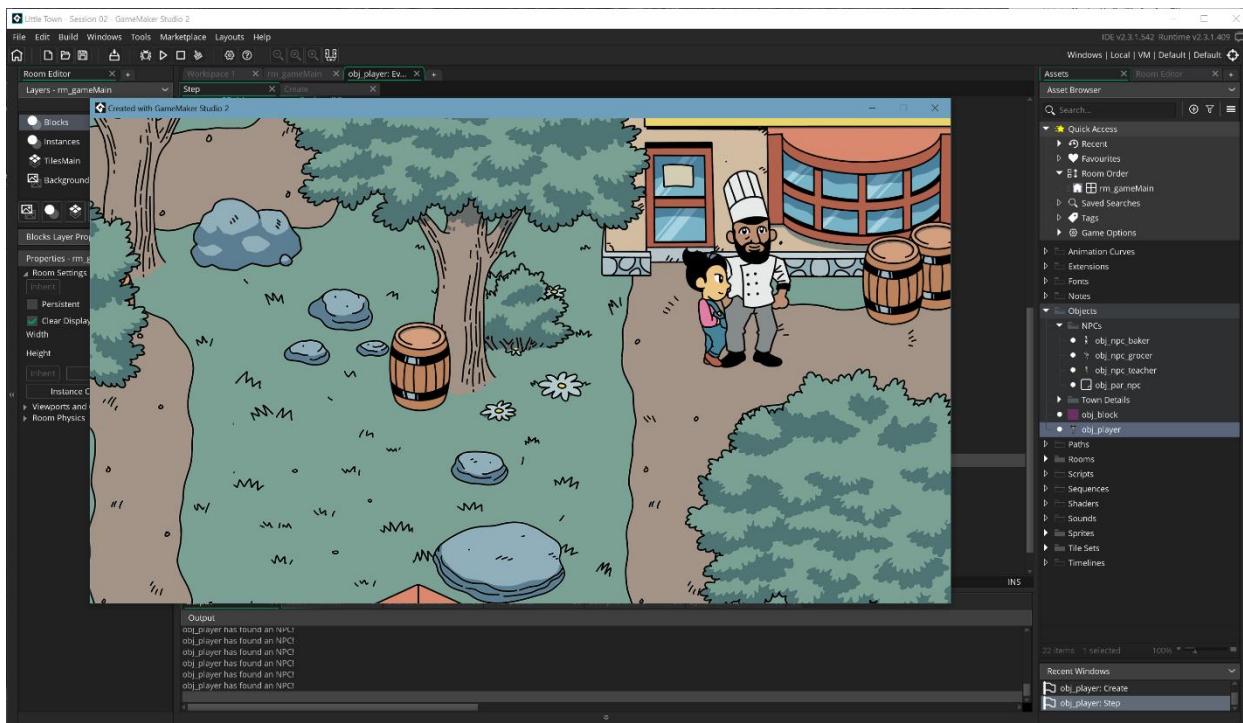
5.15 Using debug messages

Sometimes, the easiest way to test something is to have your game spit out a *debug message*, so you can tell if something works before putting in more time. That's what we're going to do here.

In `obj_player`'s Step Event, update the `// Check for collision with NPCs` code block we just wrote like so:

```
// Check for collision with NPCs
nearbyNPC = collision_rectangle(x-lookRange,y-
lookRange,x+lookRange,y+lookRange,obj_par_npc,false,true);
if nearbyNPC {
    // do something
    show_debug_message("obj_player has found an NPC!");
}
if !nearbyNPC {
    // do something else
    show_debug_message("obj_player hasn't found anything");
}
```

Now, run your game and have your player Object walk around. Pay attention to the Output window at the bottom of your GameMaker Studio 2 IDE. If your player isn't in contact with an NPC, it will keep displaying "obj_player hasn't found anything!" over and over again. Once you get near the Baker, Teacher or Grocer, the message will change.



The *Output* window (bottom) in GameMaker Studio 2 can display debug messages, which we can use to quickly check things.

Just like with the code we wrote for checking `obj_par_environment`, this `collision_rectangle()` code is checking for `obj_par_npc`. Since `obj_npc_baker`, `obj_npc_teacher` and `obj_npc_grocer` are all children of `obj_par_npc`, the check is still valid.

When you're done, close the game window and return to GameMaker Studio 2.

5.16 Updating our NPC Collision Masks

As you may have noticed while testing out the player Object's new collision code, we currently have to get too close to an NPC for our player Object to recognize it.

This is because of the collision masks each Sprite has (just as we discussed in [collision masks](#)). Remember, whenever any collision check is being done, the collision mask of a Sprite is the determining factor.

So, let's first edit the collision mask for our Baker's idle Sprite, to make sure it's how we want it.

Open `spr_baker_idle_down` in the Sprite Editor. On the left of the editor, click Collision Mask to reveal the mask for this Sprite.



The default collision mask for spr_baker_idle_down. It definitely isn't what we need.

You'll see that the collision mask here is a problem. First, it covers most of the Sprite, which doesn't make sense. Given our game's top-down perspective, we only want the player Object to detect the Baker when it steps close. With the collision mask as is, the player Object could "collide" with the Baker, even if it were standing behind and above it.

With that in mind, look beside "Mode" and click Automatic (or the arrow beside this). Choose Manual instead, and edit the collision mask to encompass only the area around the Baker's feet, like so:



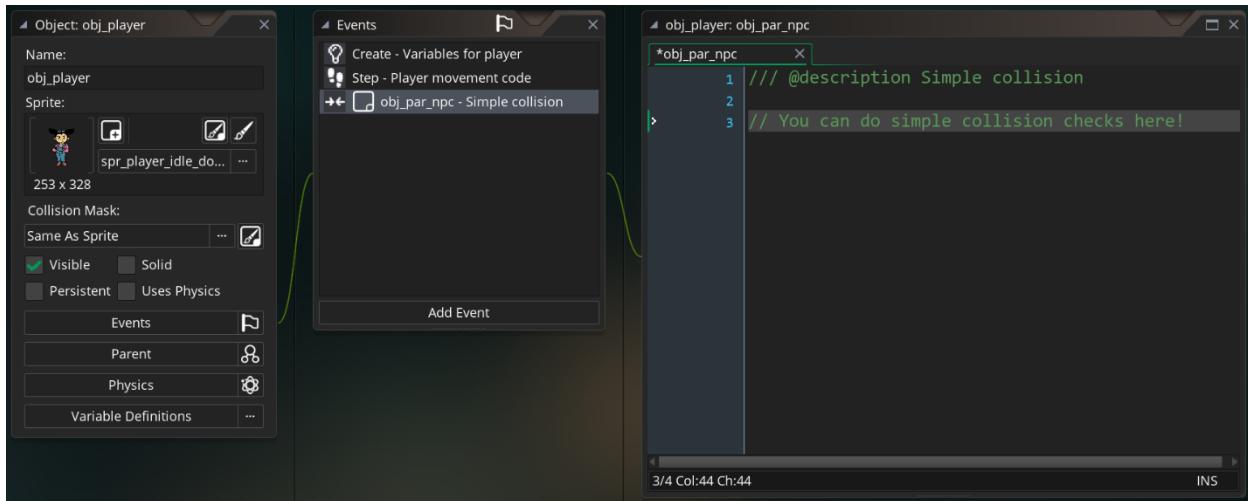
The edited collision mask for spr_baker_idle_down. Notice that the collision mask is generous, and that it even extends past the Sprite itself.

5.16.1 A note on collision Events

Why aren't we editing the Sprites for obj_player too? This is because the `collision_rectangle()` check we wrote in [Detecting the NPCs](#) is very specific (as demonstrated in the diagram in that section) and it isn't using obj_player's Sprites to detect collision.

If you want to make a simple collision check, you can also use a Collision Event. We will *not be using these for this tutorial*, but to do so:

- Open the Object from the Asset Browser
- In the Object Editor, click Add Event and choose Collision > Objects > [object]

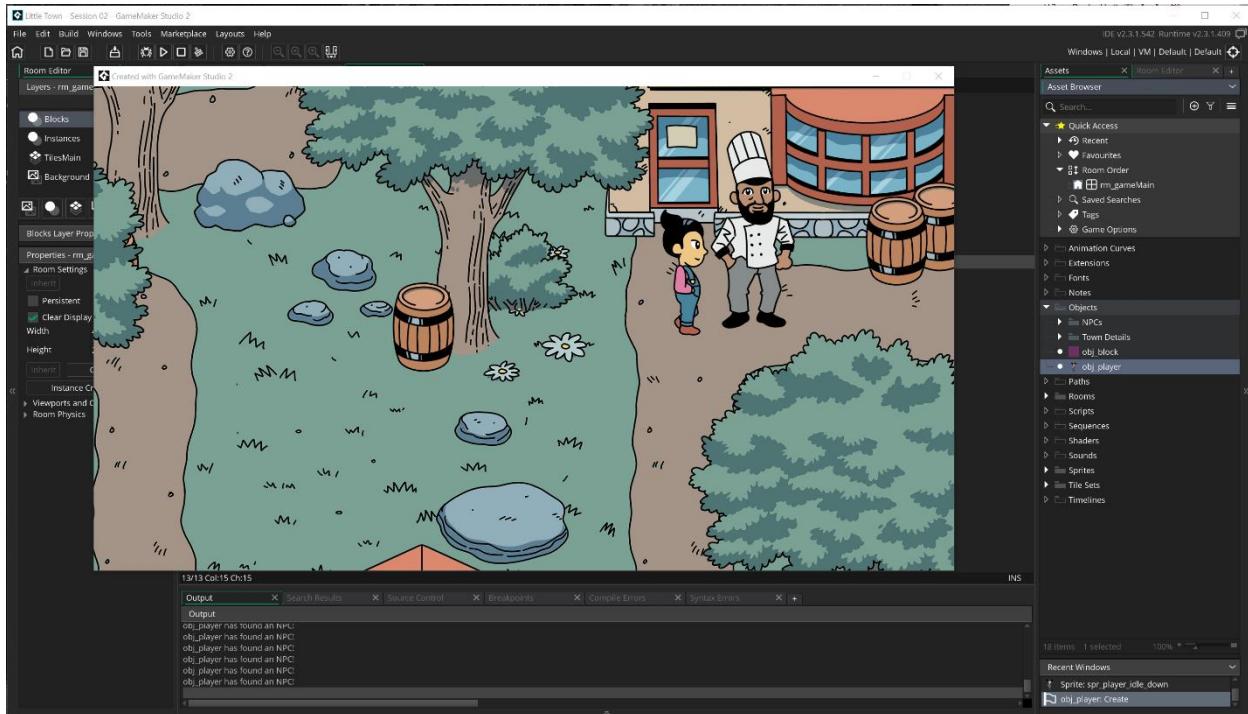


An example of a simple collision Event. We won't be using these for this particular tutorial.

Back to the matter at hand: when you've edited the collision mask for `spr_baker_idle_down`, run your game again and have the player Object walk over to the Baker. The points at which the player Object recognizes the Baker should make a lot more sense.

If you think the collision isn't quite right (i.e., you must get too close, or it works if you're too far), you can also edit the value of `lookRange` in `obj_player`'s Create Event.

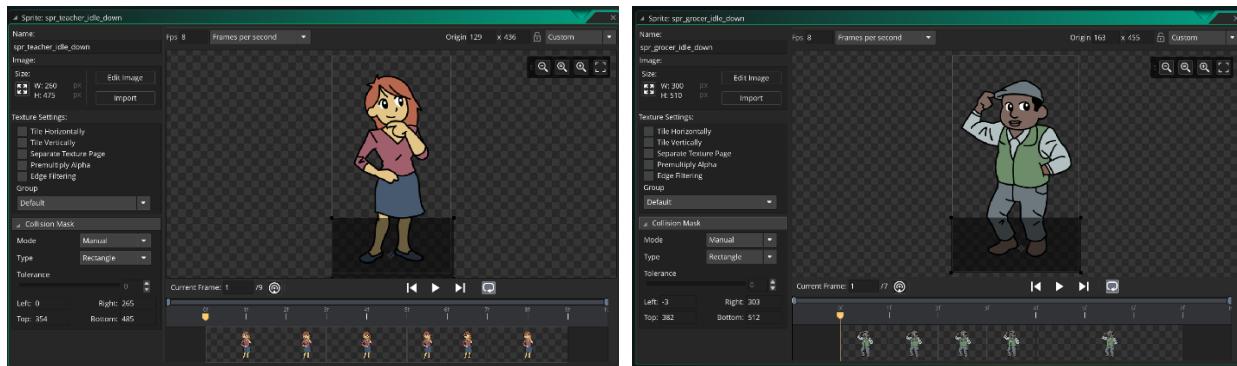
(Don't forget, look in the Output window at the bottom of the IDE for the debug messages we wrote.)



Now the range at which the player Object detects the Baker makes more sense.

Once you're happy with the Collision Mask for the Baker, repeat these steps for:

- spr_teacher_idle_down
- spr_grocer_idle_down

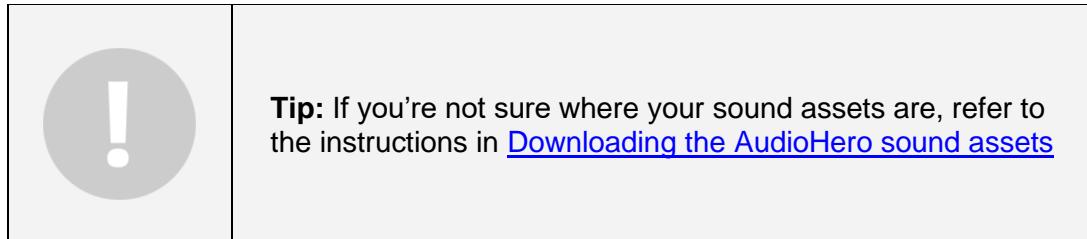


Editing the Collision Masks for the Teacher and Grocer idle Sprites.

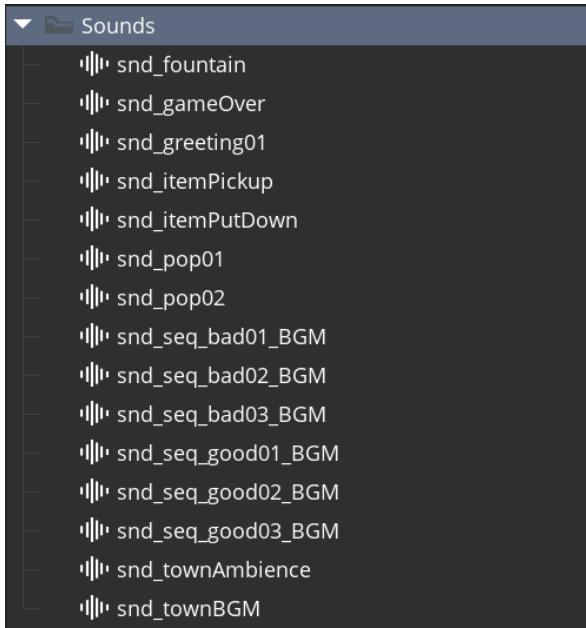
5.17 Sound and music

Now that our game is looking nice, it's time to make it *sound* nice as well. We're going to apply some sound effects and music and experiment with 3D audio.

Using File Explorer (Windows) or Finder (Mac), navigate to the Assets folder provided with this course and open the Sounds folder.

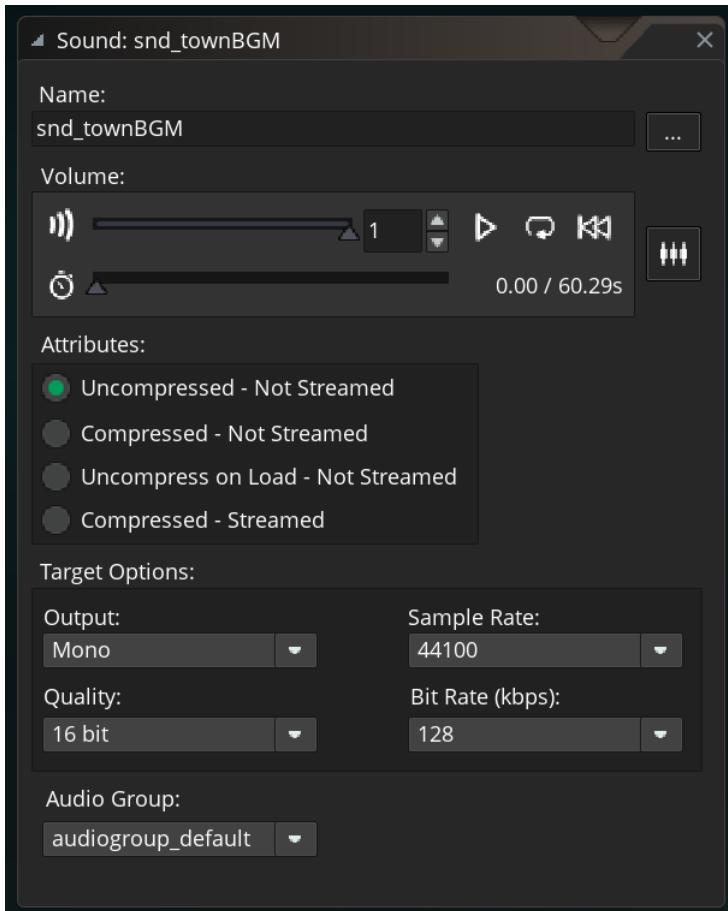


Drag these files into GameMaker Studio 2's Asset Browser, within the Sounds group (if you have one). If you do not have a Sounds group, create one and place these new Sound assets in there.



Our imported Sound assets, organized within a group in the Asset Browser.

Open the asset called `snd_townBGM` ("BGM" stands for "background music," in case you were wondering). In the Sound Editor, you'll see some simple options:



The default state of `snd_townBGM` while opened in the Sound Editor.

The Sound Editor lets us change a Sound asset's base volume, playback speed, deal with compression and other options. (For details on all the features in the Sound Editor, refer to the GameMaker Studio 2 manual.)

For now, however, we only need to change one thing. Below the target option "Output," click the drop-down menu (it will likely say Mono by default) and choose Stereo. We want to make sure our beautiful background music is heard correctly.

5.18 Using control Objects

Close the Sound Editor and right-click on the Objects group in the Asset Browser to create a new Object. Name it `obj_control1`. This Object is going to be an invisible "control Object" — think of it like the director of a play, making sure certain things go smoothly behind the scenes.

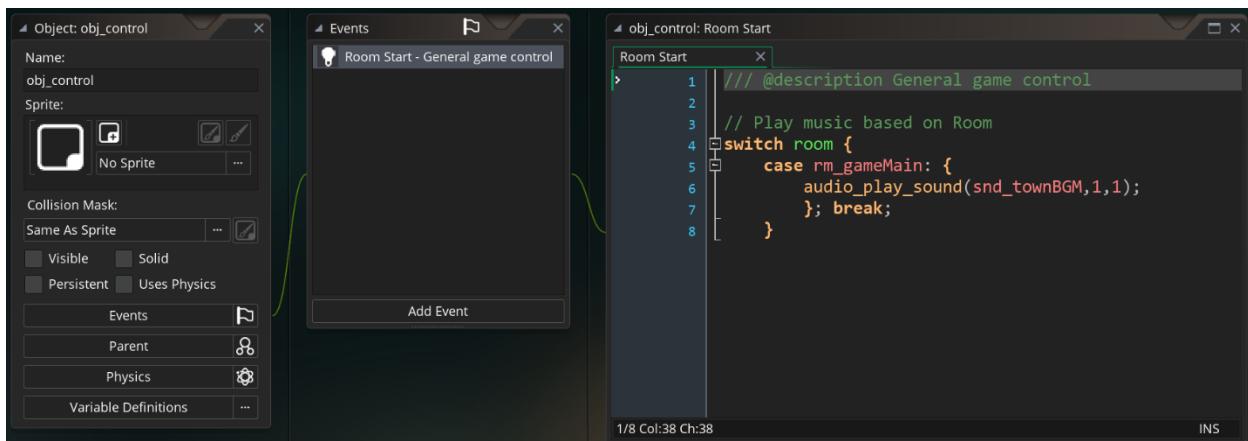
Open `obj_control1` and in the Object Editor and uncheck the box that says "visible."

Next, click Add Event. Choose Other > Room Start. This is a special Event that occurs (as you might have guessed) only when we enter a Room for the first time. It's different than the Create Event, which occurs when an Object first pops into being.

In this new Room Start Event, write the following code:

```
// Play music based on Room
switch room {
    case rm_gameMain: {
        audio_play_sound(snd_townBGM,1,1);
    }; break;
}
```

This is another switch function (which we first covered in [Changing our player Sprites](#)). In this instance, we're checking for which Room has just started and since we only have one Room, there's only one case.



Our new `obj_control` with its Room Start event.

You can read more about this basic audio function (`audio_play_sound()`) in the GameMaker Studio 2 manual, but here is a quick breakdown:

<code>audio_play_sound(</code>	<code> snd_townBGM,</code>	<code> 1,</code>	<code> 1</code>	<code>);</code>
	which Sound Asset to play	Priority (an arbitrary number; the higher the number, the higher the priority)	1 = will loop 0 = play once (you can also use true or false instead)	

Open `rm_gameMain` in the Room Editor and drag our new `obj_control` onto the Instances layer. It doesn't matter where you put it, since it's invisible, but I recommend you tuck it just outside the Room in the top-left, so you can easily find it later.

	<p>Tip: Objects without Sprites attached to them will appear as a grey "(?)" in the Room Editor. It's often helpful to make your own Sprites for these invisible Objects so you can keep track of them within GameMaker Studio 2. If you uncheck the "visible" box in the Object Editor, the player will never see these Objects while the game is running.</p>
--	--

With our new control Object in place, run the game again and feast your ears — music!

When you're done enjoying the tunes, close the game window and return to GameMaker Studio 2.

In the Sounds group within the Asset Browser, open `snd_townAmbience`. Just as we did with `snd_townBGM`, we want to make one change to this asset.

In the Sound Editor, click the menu under "Output" and change this to Stereo (just as we did with the background music).

Next, open `obj_control` again, and return to its Room Start Event.

Update the // Play music based on Room code block like so:

```
// Play music based on Room
switch room {
    case rm_gameMain: {
        audio_play_sound(snd_townBGM,1,1);
        audio_play_sound(snd_townAmbience,1,1);
    }; break;
}
```

This will play both the lovely background music and the pleasant country ambience at the same time.

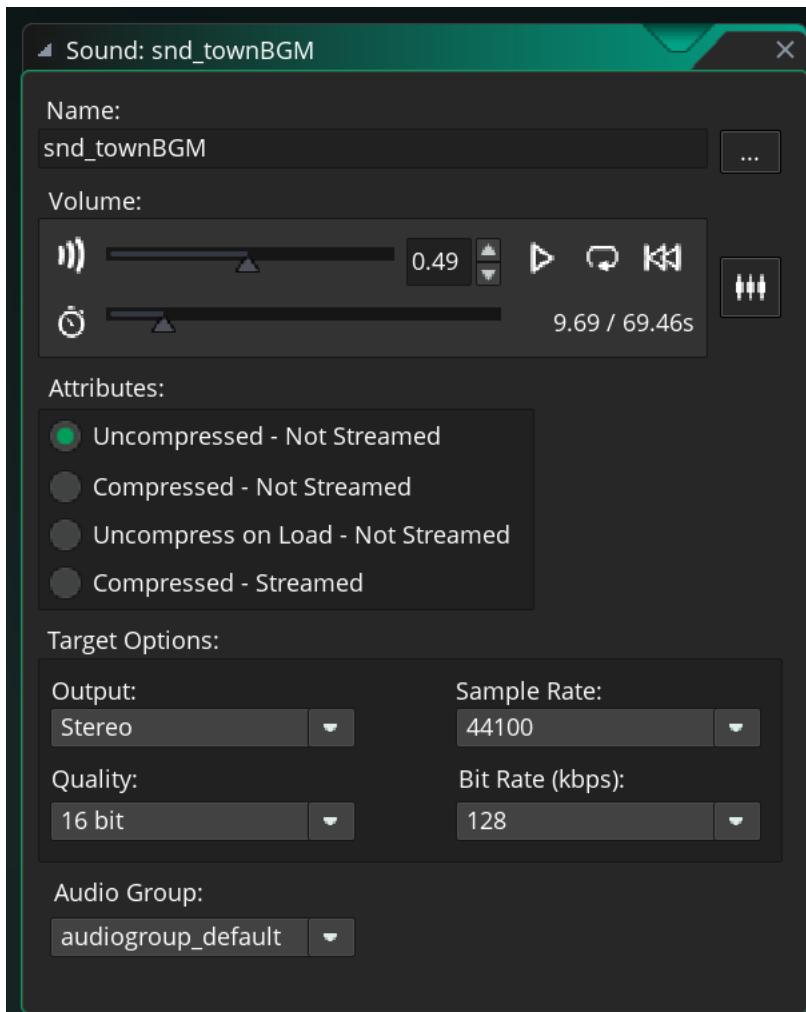
To test this run the game again. You'll notice that the music now seems a little loud in comparison to the ambient loop.

When you're ready, close the game window and return to GameMaker Studio 2.

Open `snd_townBGM` again from the Asset Browser. In the Sound Editor, pay attention to the play controls (under where it says “Volume”).

You can click the play button here to preview a Sound asset. You can also adjust its volume by move the first triangular slider under “Volume,” or by manually changing the number value beside that.

Lower the volume of `snd_townBGM` until it seems right to you.



Changing the volume of `snd_townBGM` within the Sound Editor.

When you’re done, run the game again and listen to the results. If the mix still isn’t quite right, adjust the volume of `snd_townBGM` until you’re satisfied.

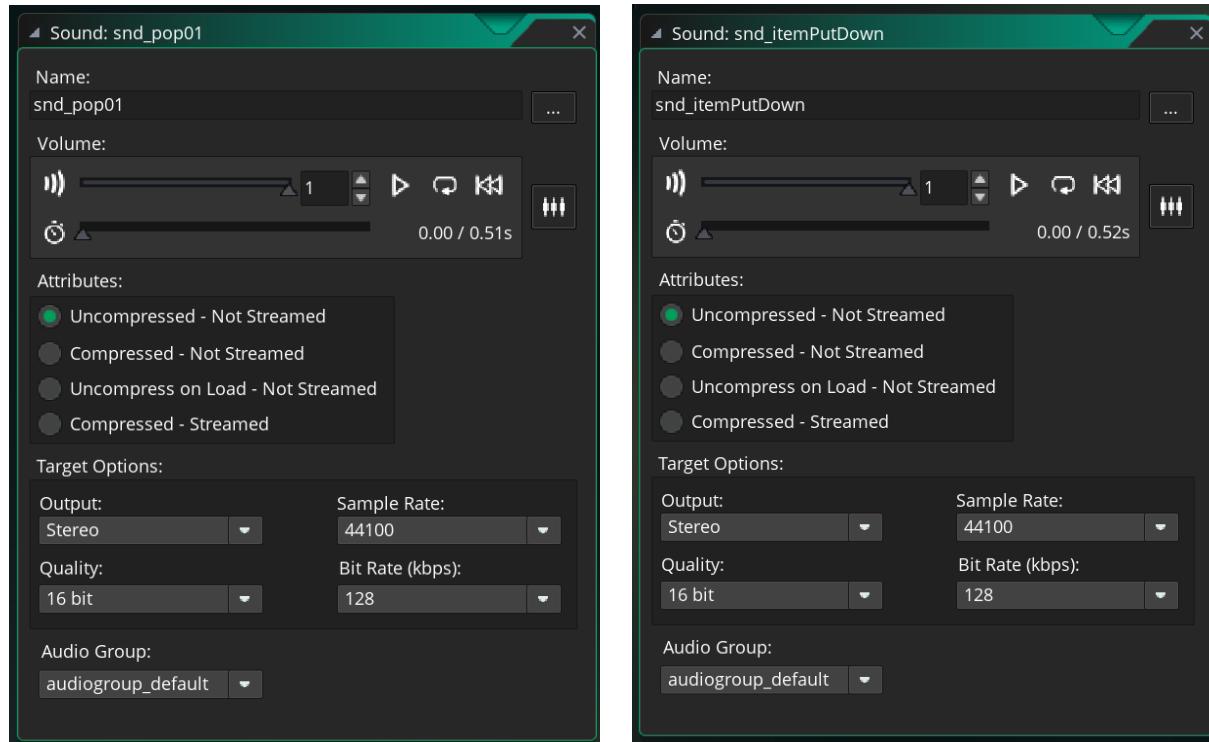
5.19 Say hello with sound

All right, let's add some more sound to our game. We're going to have our player Object say something whenever they run into another character.

We're also going to be utilizing the rest of the Sound assets we imported throughout the rest of the course. To save us some time later on, let's make a quick change to several Sound assets.

Open the following Sound assets from the Asset Browser and change their Output setting to Stereo:

- snd_gameOver
- snd_greeting01
- snd_itemPickup
- snd_itemPutDown
- snd_pop01
- snd_pop02
- snd_seq_bad01_BGM
- snd_seq_bad02_BGM
- snd_seq_bad03_BGM
- snd_seq_good01_BGM
- snd_seq_good02_BGM
- snd_seq_good03_BGM



Editing Sound assets to make them output in stereo

Open obj_player and its Step Event. Go to the // Check for collision with NPCs code block and update it like so:

```
// Check for collision with NPCs
nearbyNPC = collision_rectangle(x-lookRange,y-
lookRange,x+lookRange,y+lookRange,obj_par_npc,false,true);
if nearbyNPC {
    // Play greeting sound
    audio_play_sound(snd_greeting01,1,0);
    show_debug_message("obj_player has found an NPC!");
}
if !nearbyNPC {
    // do something else
    show_debug_message("obj_player hasn't found anything");
}
```

We're including the exact same audio function we used to play the background music, except the last value is 0, which means this one won't loop.

However, there's something you should see (or, rather, hear). Run the game again and move the player so they collide with one of the characters, then stand still.

Oh boy, that's not good! Even though the audio function is set to not loop, you can hear that our player is greeting the NPC over and repeatedly. This is because in the player Object's Step Event, we're using that nearbyNPC line to check for an NPC. And if that's true, the game will play our greeting sound.

However, we only want our player to say their greeting *once* each time it collides with another character, so how do we fix this? As it turns out, very simply.

5.20 Using variables to maintain control

Oftentimes when we need to be specific about when or how something happens, we can use a variable and change its value as a simple control technique.

Open obj_player's Create Event and add a new line:

```
| hasGreeted = false;
```

Now go back to obj_player's Step Event and let's edit that nearbyNPC code one more time:

```
// Check for collision with NPCs (v1)
nearbyNPC = collision_rectangle(x-lookRange,y-
lookRange,x+lookRange,y+lookRange,obj_par_npc,false,true);
if nearbyNPC {
    // Play greeting sound
    if (hasGreeted == false) {
        audio_play_sound(snd_greeting01,1,0);
        hasGreeted = true;
    }
    show_debug_message("obj_player has found an NPC!");
}
if !nearbyNPC {
    // Reset greeting
    if (hasGreeted == true) {
        hasGreeted = false;
    }
    show_debug_message("obj_player hasn't found anything");
}
```

What we've done is use that new variable, hasGreeted, to determine if our player has already said "hello" yet. When the player collides with an NPC, it checks that variable and if it's false, we play the greeting sound effect. Then we set hasGreeted to true, so the sound won't play again.

Finally, when the player isn't colliding with an NPC, it checks to see if hasGreeted has been set to 1 and resets it to false, so the next time we approach one of our characters, our player will say "hello" again (just once).

Run the game and see our much-more-polite player in action. That's better!

Let's do one last thing here. Sometimes you *really* want to control when a sound plays and not allow overlapping or duplicate Sounds. So, in obj_player's Step Event, let's make one last edit to that nearbyNPC code:

```
// Play greeting sound
if (hasGreeted == false) {
    if !(audio_is_playing(snd_greeting01)) {
        audio_play_sound(snd_greeting01,1,0);
        hasGreeted = true;
    }
}
```

All we've added here is one further check: to see if the greeting sound is (not) already playing and *then* play the sound.

Run the game again and try it out. Even if you quickly walk up to a character over and over, the player Object won't play its greeting sound until it's finished doing so. It may not appear to make much of a difference since snd_greeting01 is so short, but this change will still help our soundscape feel a bit more natural.

5.21 3D positional audio

Okay, now we're *really* going to get fancy. We have background music, we have a simple example of a standard sound effect, but what about 3D audio?

What we're going to do is take that big fountain that we have sitting somewhere in our town and have it play a looping sound effect. The difference here will be that it will be *positional*, so that as the player Object moves around town, we'll be able to hear where that fountain sound is, relative to the player's position.

In order to use 3D audio, we need two components: at least one *Emitter* and one *Listener*.

An Emitter does just as the name suggests; it *emits* (or plays) a Sound asset. A Listener can hear Sounds that an Emitter plays, and it's the relationship between these two that creates the positional effect.

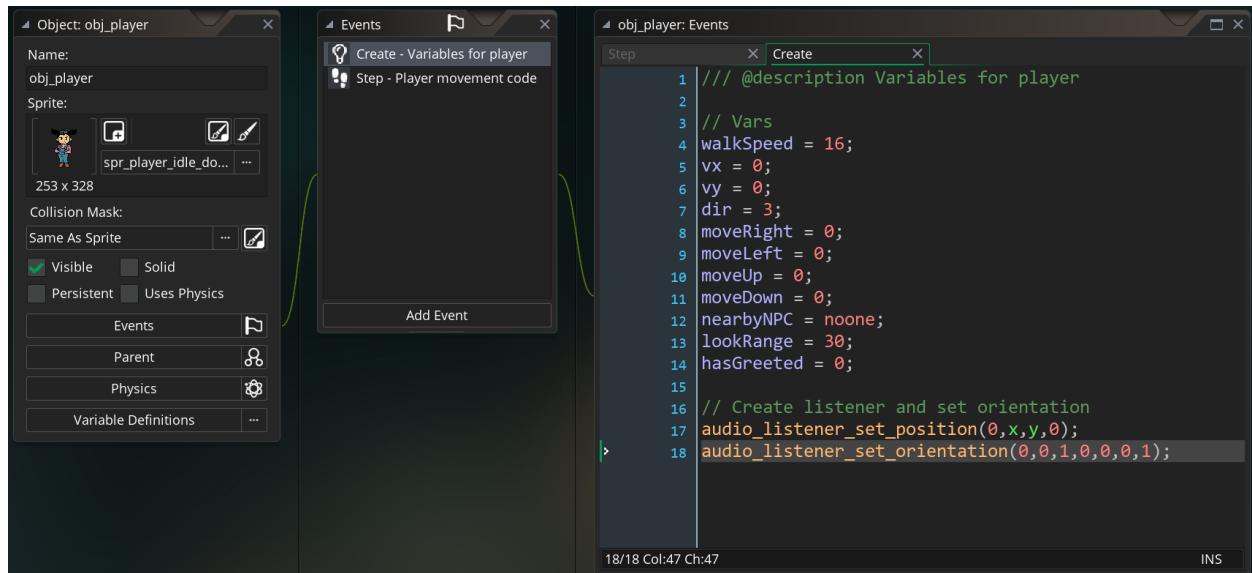
5.22 Moving the Audio Listener with the player

First, let's ensure we have a Listener set up. Usually, we set the Listener in a game as our player Object. So, Open obj_player and open its Create Event.

In the Create Event, add a few spaces below the variables we wrote previously and add these lines of code:

```
// Create listener and set orientation  
audio_listener_set_position(0,x,y,0);  
audio_listener_set_orientation(0,0,1,0,0,0,1);
```

You can read more technical details about that second line in the GameMaker Studio 2 manual, but what you need to know here is simple: the first line sets the game's Audio Listener position to the x and y values of the player and the second line sets it correctly within 3D space.



Adding the new Audio Listener setup code in obj_player's Create Event.

Next, open obj_player's Step Event and find the // If moving code block. Update that code block like so, to add these new lines:

```
// If moving  
if (vx != 0 || vy != 0) {  
    if !collision_point(x+vx,y,obj_par_environment,true,true) {  
        x += vx;  
    }  
    if !collision_point(x,y+vy,obj_par_environment,true,true) {  
        y += vy;
```

```

}

// Change walking Sprite based on direction
if (vx > 0) {
    sprite_index = spr_player_walk_right;
    dir = 0;
}
if (vx < 0) {
    sprite_index = spr_player_walk_left;
    dir = 2;
}
if (vy > 0) {
    sprite_index = spr_player_walk_down;
    dir = 3;
}
if (vy < 0) {
    sprite_index = spr_player_walk_up;
    dir = 1;
}

// Move audio listener with me
audio_listener_set_position(0,x,y,0);
}

```

This is the same code that's in the Create Event. We include it here so that as the player moves, it updates the Audio Listener position to its own. (There's no need to keep doing this if the player is standing still, which is why we're only putting it within the movement code block.)

Run the game and notice that so far, everything we've done with audio is the same. It's only when we add emitters that things get interesting.

When you're ready, close the game window and return to Game Maker Studio 2.

5.23 Adding an Audio Emitter

We want our fountain Object to *emit* a looping sound effect that the game's Listener will pick up in 3D space, so we need to create an *Emitter* and then play its audio through that.

Since the fountain is the child of a parent Object (`obj_par_environment`), however, it makes more sense to add this functionality to the parent Object and allow any of our environmental Objects to use it.

So, open `obj_par_environment` and let's do two things:

First, click on Variable Definitions and click the Add button.

Give this new Variable Definition use the name useSound. Make its Type Asset and enter noone for its default.

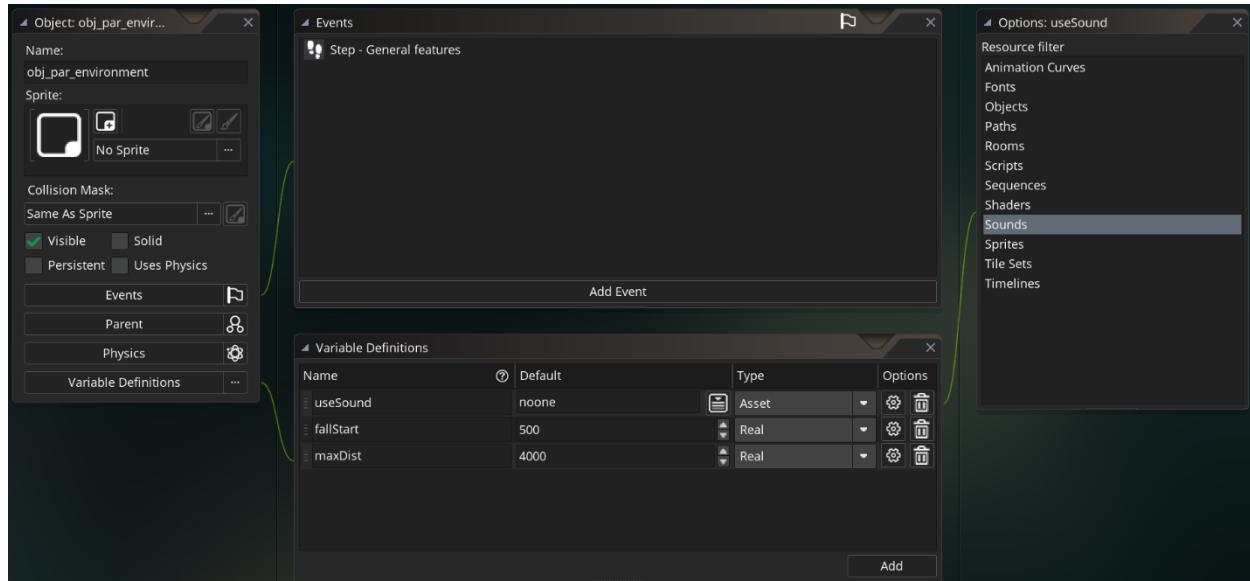
Choosing the variable type Asset lets us restrict choices here to specific kinds of assets (Sprites, Sounds, etc.). Click the gear icon (under Options and choose Sounds in the Options window that opens up.

To complete this step, click the drop-down menu under Default and choose None. You'll see this value, noone, in the Default column.

Next, add two more Variable Definitions with the following properties:

Name	Default	Type
fallStart	500	Real
maxDist	4000	Real

(We'll address what these do in a moment.)



Adding the three new Variable Definitions to obj_par_environment.

Next, add an Event to obj_par_environment: click Add Event and choose the Create Event. In this new Create Event, write the following code:

```

// Emitter variables
myEmitter = 0;

// Create emitter
if (useSound != noone) {
    if !audio_is_playing(useSound) {
        myEmitter = audio_emitter_create();
        audio_emitter_position(myEmitter,x,y,0);
        audio_falloff_set_model(audio_falloff_exponent_distance);
        audio_emitter_falloff(myEmitter,fallStart,maxDist,1);
        audio_play_sound_on(myEmitter,useSound,1,1);
    }
}

```

Here, we're creating an Audio Emitter, storing its ID in the variable `myEmitter` and then setting a bunch of options about how sound is dealt with. The three key things you need to pay attention to are:

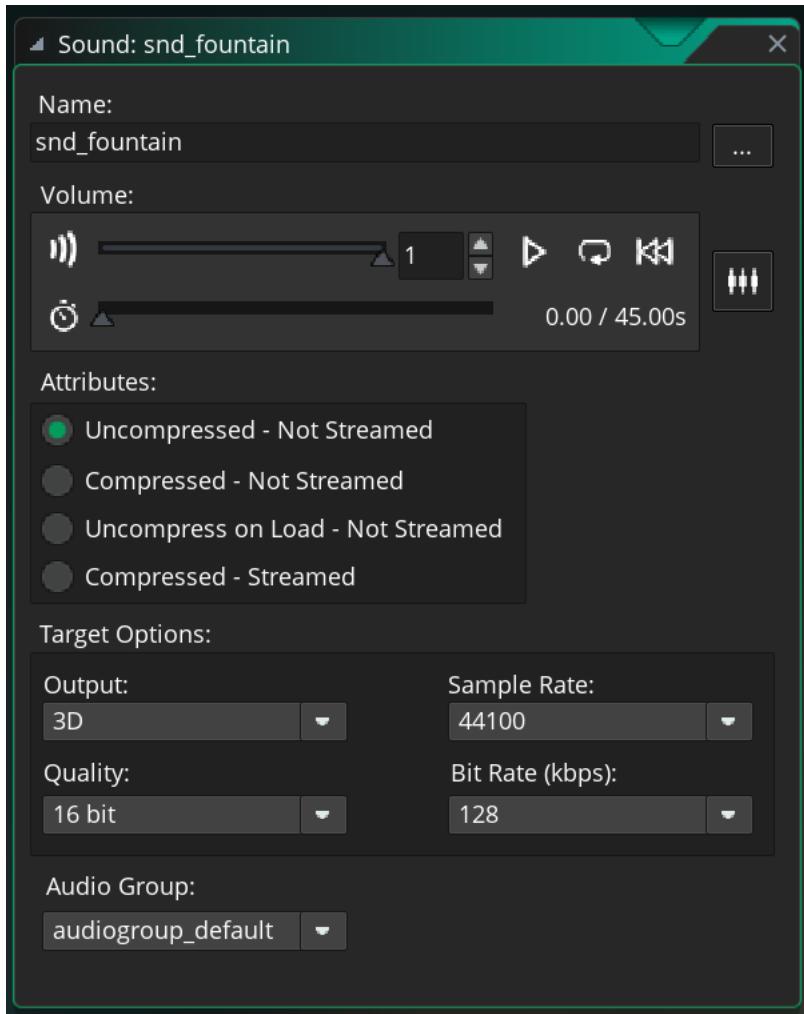
<code>fallStart</code>	This Variable Definition we set determines when the audio being played through the Emitter begins to "fall off"(get quieter)
<code>maxDistance</code>	This Variable Definition we set determines the maximum distance at which the Listener (our player Object) can hear the audio
<code>audio_play_sound_on(myEmitter,useSound,1,1);</code>	Notice that this is similar to the audio function we used before, but it is being played <i>on</i> an Emitter (the one we just created)

The next thing we need to do to make this work is to choose a sound for `obj_fountain` to play.

Open `obj_fountain` and click **Variable Definitions** in the Object Editor. Click the pencil icon to unlock the `useSound` variable that our fountain has inherited from its parent and then click the drop-down menu under Default. Choose `snd_fountain`.

The last thing we need to do is make sure our fountain Sound is compatible with the 3D positional audio setup we've created.

Open `snd_fountain` from the Asset Browser. In the Sound Editor, under "Target Options," make sure to set the "Output" to 3D.



Changing the Output setting of snd_fountain to 3D

Once all this is done, run the game again and take a stroll around town; you should hear that glorious fountain bubbling away in a realistic, 3D space.

(If you find the fountain is too quiet when farther away, you can open obj_fountain and edit the Variable Definitions for fallStart and maxDistance.)

	<p>Tip: You'll notice that we set up our Emitter in obj_par_environment's Create Event, but it checks for a variable (useSound) that we only set up in the Variable Definitions window. This is because anything declared in Variable Definitions runs before that Object's Create Event.</p>
--	--

5.24 End of Session 2

We now have a burgeoning town with three characters, 3D audio and a chatty main character. In the next session, we'll deal with textboxes and creating Objects on the fly. Don't forget to save your project!

6 Session 3

In this session, we're going to get our three characters (the Baker, Teacher and Grocer) all chatting with textboxes, and introduce some User Interface (*UI*) elements into our game.

6.1 Setting up our textbox

Using File Explorer (Windows) or Finder (Mac), navigate to the Assets folder provided with this course. Find the folder called User Interface and drag these Sprites into GameMaker Studio 2:

- spr_gui_prompt
- spr_gui_textbox

In the Asset Browser, create a new group in the Sprites group called User Interface.

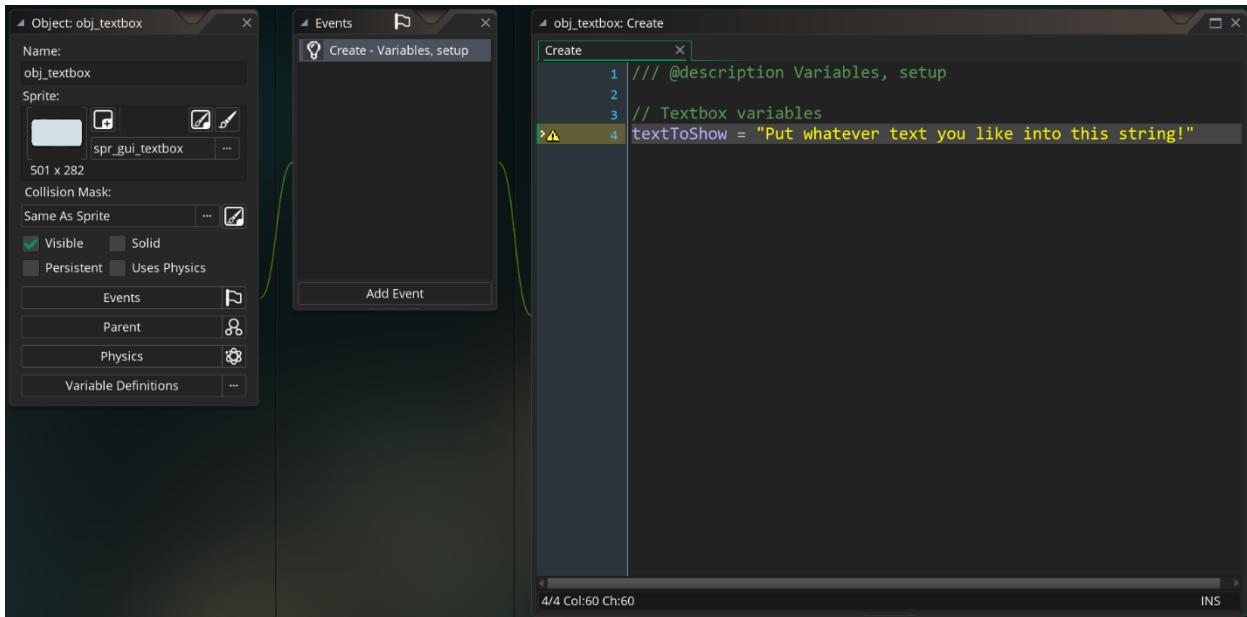
Drag these new assets into Sprites > User Interface.

Create a new Object and name it obj_textbox. In the Object Editor, attach the Sprite spr_gui_textbox that we just imported.

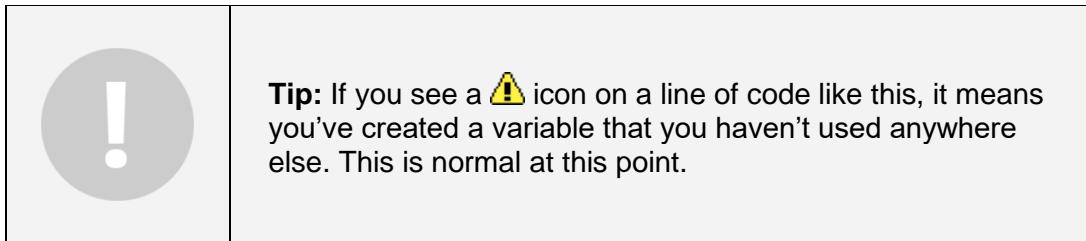
Then click Add Event and add a Create Event. In this new Event, let's enter a basic variable to get started:

```
// Textbox variables  
textToShow = "Put whatever text you like into this string!"
```

This is the text that we're going to have our textbox display. Write whatever you like here, but don't make it too long.



Our new obj_textbox with its Sprite attached and a Create event.



6.2 Drawing things manually

Now we're going to do something new: make our textbox display a few things at once using what's called a *Draw Event*.

In the Object Editor for obj_textbox, click Add Event and roll over Draw — you'll notice several options in here, but for now, just choose the one simply named Draw.

Click on this new Event in the Object Editor and write in the following:

```

// Draw textbox
draw_self();

// Draw Text
draw_text(x,y,textToShow);

```

So what's going on here?

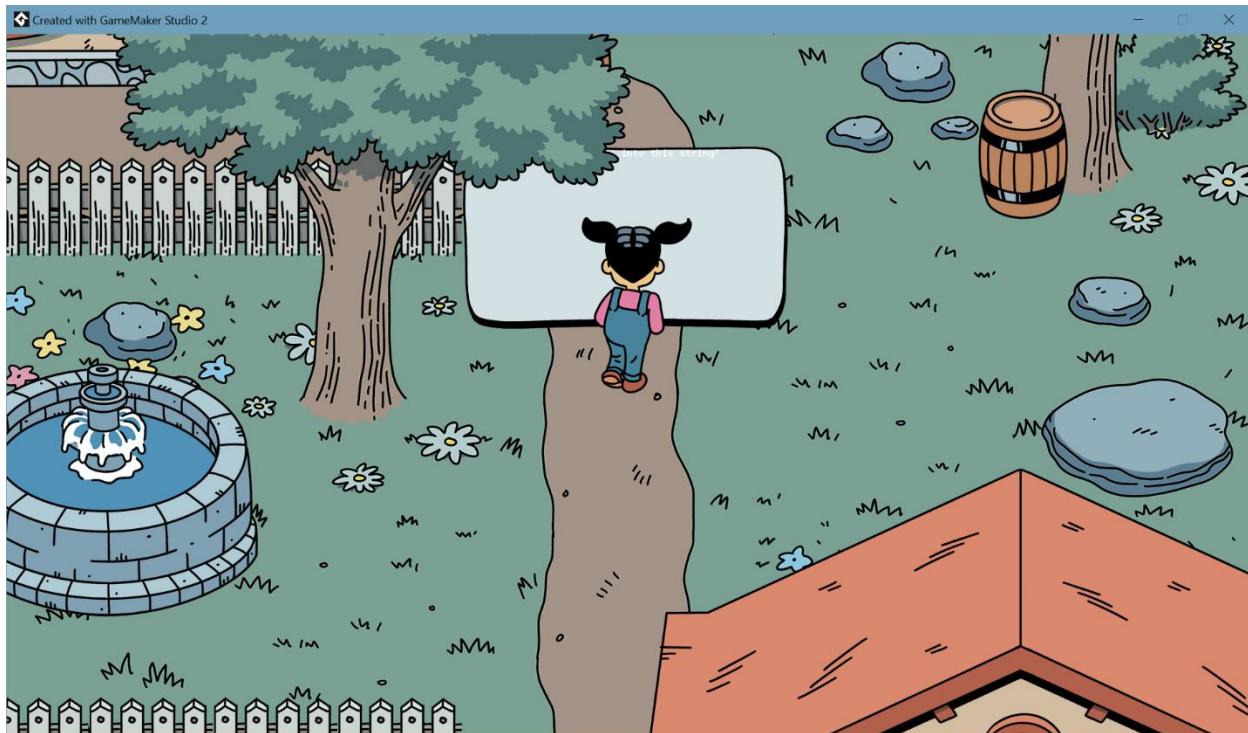
When you attach a Sprite to an Object, that Object *draws* it automatically, even without a Draw Event. Adding a Draw Event overrides that automatic behaviour, so we must now tell GameMaker Studio 2 that we still want our textbox Object to draw the Sprite we attached.

Here, `draw_self()` does exactly that. We can get much fancier with drawing Sprites, but for now, this will do.

Next, `draw_text` also does what it says: draws a *string* of text (which could be a letter, a number, a word, a sentence, etc.) and does so at a specific x and y coordinate. (Remember: just putting x and y means: "draw this at *my* x and y position.")

Okay, let's see what we've done so far. Open `rm_gameMain` and make sure to click on the Instances layer. Drag an instance of `obj_textbox` into the Room, wherever you like.

Run the game again and notice what's happening:



Our new `obj_textbox`, positioned within the game's town. This isn't quite what we want.

You'll notice that this isn't quite what we were hoping for. However, though the text may be very difficult to read (and it may look different depending on your system), our `textbox` Object is correctly drawing both the `textbox` Sprite and the text we wrote, so that's a plus.

When you're ready, close the game window and return to GameMaker Studio 2.

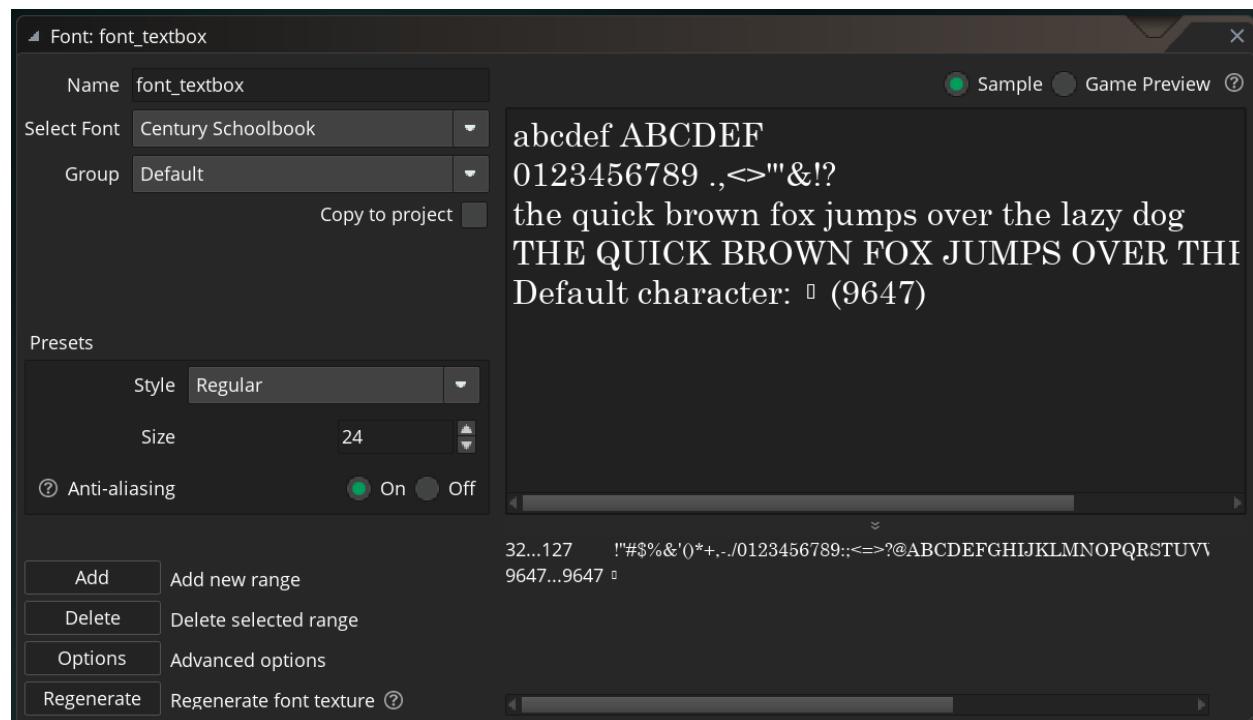
6.3 Working with fonts

In order to make our `textbox` behave the way it's supposed to, we need to be more specific about how we want our text to look and what happens to it.

In the Asset Browser, right-click on the Fonts group and choose Create > Font. GameMaker Studio 2's Font Editor will open with some very default-looking text.

Name our new font `font_textbox`.

Click beside `Select Font` and choose any available font on your system that you like.



Creating our first font and changing its typeface and size.

We don't need to worry about most of the options here, but make sure "Anti-aliasing" is turned On; this will make sure your font is smoothed correctly and looks good, no matter the size. Aim for a font size around 24 (but feel free to play around).



Tip: if you create a font from a typeface that might be removed from your system, it can cause issues later. It's a good idea to install special fonts you're planning on using or use font management software to make sure they're activated when you're working on your project.

When you're happy with your new font, open obj_textbox again and its Draw Event. Let's spruce up that text. Update the // Draw Text code block like so:

```
// Draw Text  
draw_set_font(font_textbox);  
draw_set_color(c_black);  
draw_set_halign(va_center);  
draw_set_valign(va_middle);  
draw_text(x,y,textToShow);
```

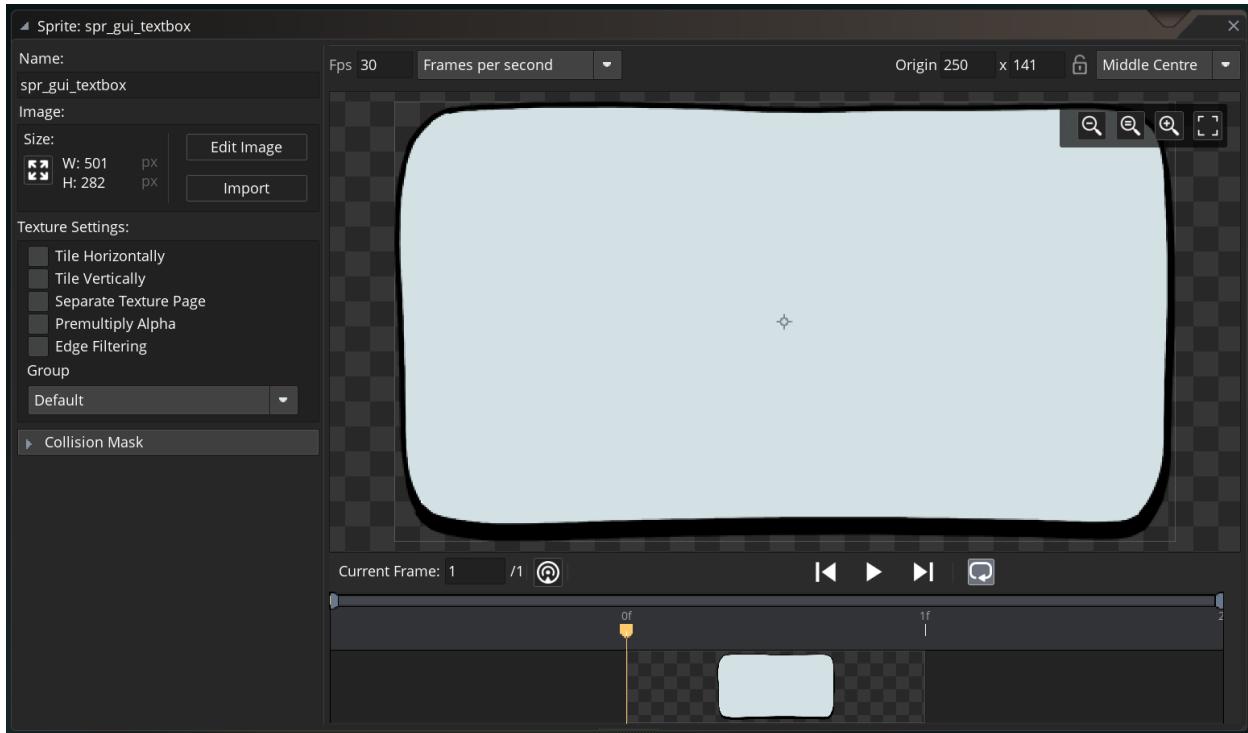
In this order, we are:

- Setting the font with which to draw the text
- Setting the colour (c_black is a built-in variable for black)
- Setting a horizontal alignment for the text
- Setting a vertical alignment
- Drawing the actual text



Tip: GameMaker Studio 2's Draw Events, like in all other Events, execute their code in linear order. As well, whatever is drawn here will be done "bottom up" — meaning that Sprites or text you declare first will appear below elements you draw second, third and so on.

One last thing: open spr_gui_textbox again from the Asset Browser and make sure the Origin for this Sprite is in the middle. An easy way to do this in the Sprite Editor is to click the drop-down menu at the top of the window next to Origin and choose Middle Centre.



Changing the Origin of spr_gui_textbox to Middle Centre in the Sprite Editor.

When you're done, open `rm_gameMain` again and make sure the `textbox` Object is placed so you can clearly see it.

Then run the game again and take a look at our updated `textbox`. Though it's not quite right yet, it should look better with black text that you can read over the `textbox` Sprite:



The updated obj_textbox, with centre-aligned, black text in the font of our choosing.

6.4 A more advanced textbox

Depending on what text you put in that `textToShow` variable, your text may have run outside of the textbox (as ours did). That's because the simple function we're using (`draw_text()`) doesn't have a lot of options and doesn't take into consideration details like a maximum width.

But GameMaker Studio 2 has several text-drawing functions; so, let's replace our simple `draw_text` code with something a little more flexible.

First, open `obj_textbox` and its Create event. Update the `// Textbox variables` code block like so:

```
// Textbox variables
textToShow = "Put whatever text you like into this string!"
textWidth = 450;
lineHeight = 28;
```

Next, open `obj_textbox`'s Draw event. Update the `// Draw text` code block again, like so:

```
// Draw Text
draw_set_font(font_textbox);
draw_set_halign(fa_center);
```

```
draw_set_valign(va_center);
draw_text_ext_color(x,y,textToShow,lineHeight,fontWeight,c_black,c_black,c_black,c_black,1);
```

You'll notice that we no longer have the line of code that sets the colour. This new `draw_text_ext_color()` function lets us do several things all on one line, including setting the colour of the text we're drawing. (We can set *four* colours here, so feel free to play around and see what happens!)



Tip: There are several built-in colours in GameMaker Studio 2 that you can access by typing `c_` and choosing one of the auto-suggested options (such as `c_white`, `c_blue`, etc.). In order to make your own colours, see the GameMaker Studio 2 manual entry on `make_colour_rgb()`.

You can read further details of what the `draw_text_ext_color()` function does in the GameMaker Studio 2 manual, but the three new arguments you see here (`lineHeight` and `textWidth`, which we set in the Create Event, and the `1` at the end) are for *leading* (space between lines), maximum width (to keep the text within our Sprite) and *alpha* (transparency), respectively.

Run the game again to see how the textbox looks now. If it's not quite right, you can adjust the variables we set in `obj_textbox`'s Create Event. You can also change the value of `textToShow` to test out a larger block of text.



Our textbox object with more text, and with leading (space between lines) and a maximum width applied.

6.5 Calling a textbox when we need one

Our textbox looks neat, but having it just sit there in the town isn't terribly useful. What's more, depending on where you placed it in the Room, it may be behind some Objects, which isn't what we want.

So, let's learn how we can create a textbox when we need it (for example, when we get close to an NPC) and how we can change the text it displays.

First, go back to `rm_gameMain` and delete the instance of the textbox that you dragged into the town; we won't need this here anymore.

Next, open `obj_player`. We've been coding all sorts of functionality into our player's Step Event so far, but let's try something new.

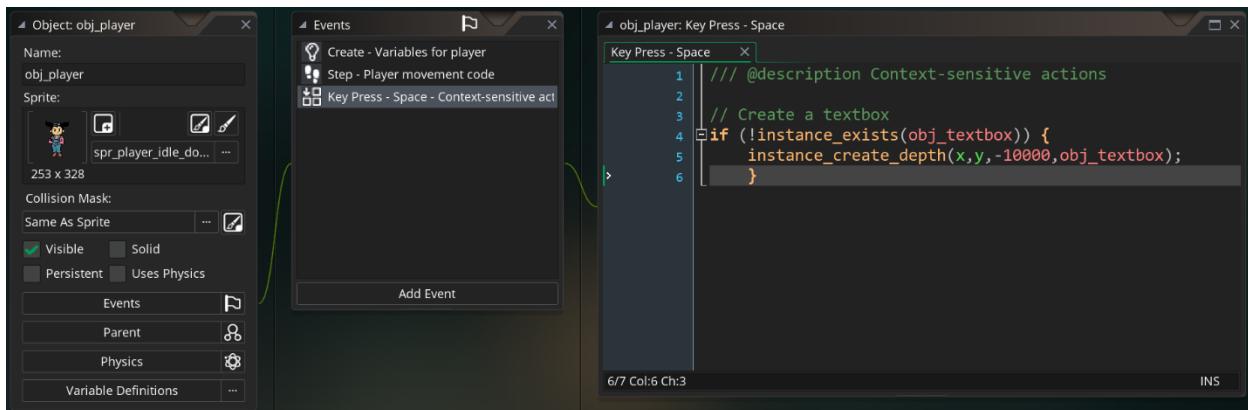
Click Add Event and choose Key Press > Space. This is an Event specifically for when the Space key is first pressed down.



Tip: GameMaker Studio 2 has Events for pressing a key (Key Pressed), holding a key down (Key Down) and releasing a key (Key Up). Just pay attention to which one you need to get the results you want.

In this new Key Press – Space Event, write the following code:

```
// Create a textbox
if (!instance_exists(obj_textbox)) {
    instance_create_depth(x,y,-10000,obj_textbox);
}
```



Adding a Key Press – Space Event to obj_player, with code to create a textbox

What this will do is:

- Make sure a textbox Object doesn't already exist
- Create a new one at the player's x and y position

Notice the function here is `instance_create_depth`, which means we are creating an Object instance at a particular *depth*. We can also create instances on specific *layers* in a Room (with `instance_create_layer`), so long as that layer exists.



Tip: Remember, depth in GameMaker Studio 2 spans from 16000 (all the way back) to -16000 (all the way front). We're just choosing a really low number here (-10000) to make sure the textbox appears above everything else in the Room.

With this done, run your game again and press the Space key to see what happens. *Voila!* textbox on command.



Pressing the Space bar should now create a textbox.

6.6 Talking to our NPCs

So, what if we want to create a textbox only when we're near the Baker, Teacher or Grocer? Well, Open obj_player and look in its Step Event. We already have a way to check for that, don't we? It's that variable we created called nearbyNPC, within the // Check for collision with NPCs code block.

So, open obj_player's Key Press - Space Event and update the // Create a textbox code block like so:

```
// Create a textbox if NPC is nearby
if (nearbyNPC) {
    if (!instance_exists(obj_textbox)) {
        instance_create_depth(x,y,-1000,obj_textbox);
    }
}
```

Now run the game again and test it out; you shouldn't be able to create a textbox with the Space key unless you're near one of the three characters.



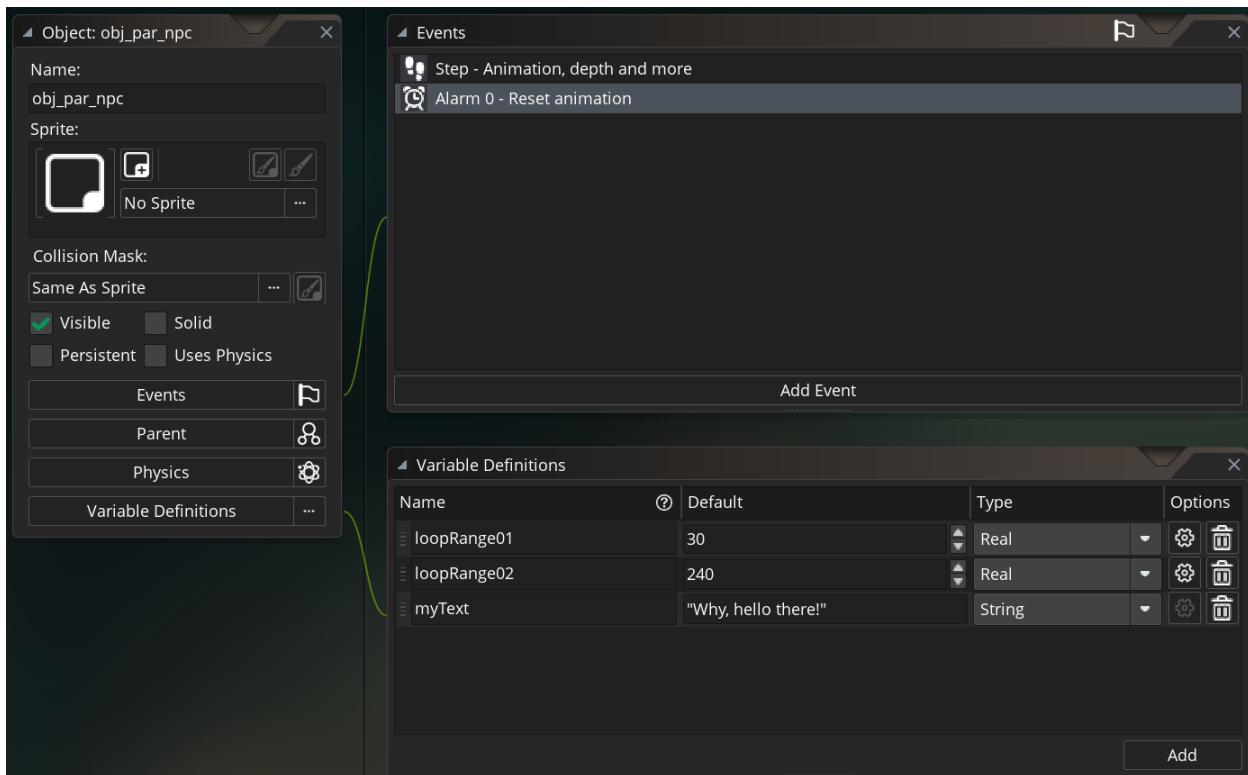
Creating a textbox can now only occur when the player Object has detected an NPC.

Close the game window and return to GameMaker Studio 2. We want to have each of our three characters say something different when the player goes up to them; to do this, we're going to use Variable Definitions again and introduce a new feature of GameMaker Studio 2.

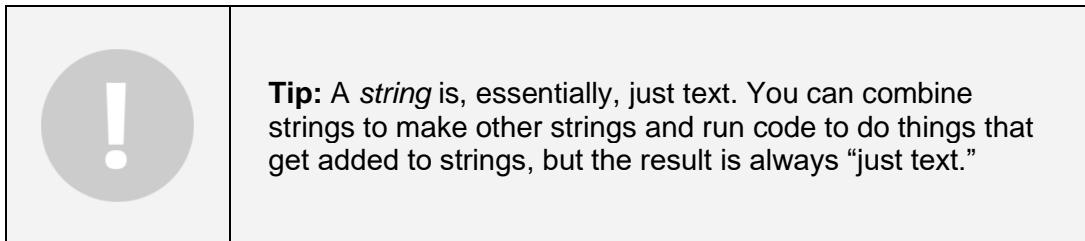
6.7 Giving our NPCs a voice of their own

First, open obj_par_npc and click on Variable Definitions again. Currently we only have the loopRange01 and loopRange02 variables here.

Click Add and add a new definition. Name it myText, set its Type to String and enter "Why, hello there!" under Default (with quotation marks).



Adding a new myText variable to obj_par_npc's Variable Definitions.



Open once again the obj_player Object and open its Key Press - Space Event. In here, we wrote the code to create an instance of obj_textbox. Now, however, we're going to do something a little extra.

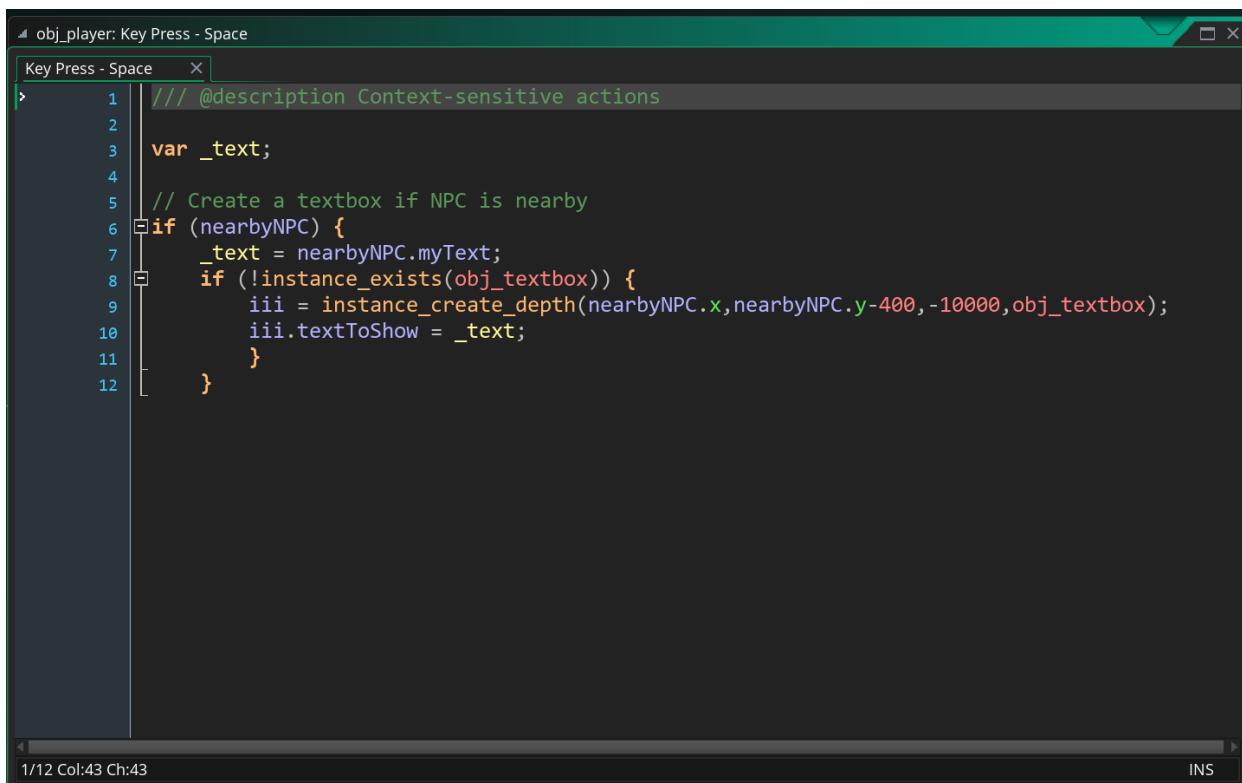
At the top of the Event, before the // Create a textbox if NPC is nearby code block, add this line:

```
| var _text;
```

(Here, we are creating a new variable, _text, but not assigning any value to it yet. We'll be doing so in different ways throughout this session, so we just want to have it ready now.)

Next, update the // Create a textbox if NPC is nearby code block like so:

```
// Create a textbox if NPC is nearby
if (nearbyNPC) {
    _text = nearbyNPC.myText;
    if (!instance_exists(obj_textbox)) {
        iii = instance_create_depth(nearbyNPC.x,nearbyNPC.y-400,-10000,obj_textbox);
        iii.textToShow = _text;
    }
}
```



The screenshot shows the Construct 3 script editor with a dark theme. A script named "Key Press - Space" is open. The code is as follows:

```
1 // @description Context-sensitive actions
2
3 var _text;
4
5 // Create a textbox if NPC is nearby
6 if (nearbyNPC) {
7     _text = nearbyNPC.myText;
8     if (!instance_exists(obj_textbox)) {
9         iii = instance_create_depth(nearbyNPC.x,nearbyNPC.y-400,-10000,obj_textbox);
10        iii.textToShow = _text;
11    }
12 }
```

The code is highlighted in green and blue. The status bar at the bottom left shows "1/12 Col:43 Ch:43". The status bar at the bottom right shows "INS".

Updating the Key Press – Space Event for obj_player to tell a textbox what text to display.



Tip: Remember, whenever you see someVariable = fancyFunction(), it just means that the *result* of whatever function is on the right is being stored in the variable on the left.

You can see that we have done something with that new `_text` variable:

```
| _text = nearbyNPC.myText;
```

You can read this line as: “take the value of `myText` in `nearbyNPC` (which is the character we’re standing next to) and store it in `_text`.

The next updated line of code we have says:

```
| iii = instance_create_depth(nearbyNPC.x,nearbyNPC.y-400,-10000,obj_textbox);
```

In this updated line of code, we’re storing the ID of the textbox instance we’re creating in the variable `iii`. Then in the new line below that, we’re doing something neat: passing a variable from one instance to another!

```
| iii.textToShow = _text;
```

We’re saying, “in that new textbox we just created (`iii`), set the value of the variable `textToShow` to be the same as whatever we stored right here in `_text`.”

	<p>Tip: Why are we taking the extra step to store <code>nearbyNPC.myText</code> in <code>_text</code>, only to pass <code>_text</code> on to a new textbox Object? That’s because as we progress through these sessions, we’re going to create textboxes for different reasons and will want them to have different text contents. This saves us a lot of work later.</p>
--	--

With this done, run your game again and walk up to one of the three NPCs. Press the Space key and that NPC should say “Why, hello there!” instead of what it said before.



Now our textbox correctly displays the text that was passed on to it.

Finally, let's make sure each of our three characters all say something different. Close your game window and return to GameMaker Studio 2.

For obj_npc_baker, obj_npc_teacher and obj_npc_baker, do the following:

1. Open the Object from the Asset Browser
2. Click on Variable Definitions in the Object Editor
3. On the line that says myText, click the Edit button
4. Change the text string within Default to something unique for that character

The image displays three separate configurations of NPC objects within the Construct 2 editor, each with its own object properties and a linked Variable Definitions panel.

Object: obj_npc_baker

- Name:** obj_npc_baker
- Sprite:** spr_baker_idle_down (260 x 475)
- Collision Mask:** Same As Sprite
- Visible:** Checked
- Events:** None
- Parent:** None
- Physics:** None
- Variable Definitions:** loopRange01 (Default: 60, Type: Real), loopRange02 (Default: 320, Type: Real), myText ("I, good morning! What a lovely day.", Type: String)

Object: obj_npc_teacher

- Name:** obj_npc_teacher
- Sprite:** spr_teacher_idle_d... (260 x 475)
- Collision Mask:** Same As Sprite
- Visible:** Checked
- Events:** None
- Parent:** None
- Physics:** None
- Variable Definitions:** loopRange01 (Default: 90, Type: Real), loopRange02 (Default: 400, Type: Real), myText ("You're late again. Shouldn't you be in school?", Type: String)

Object: obj_npc_grocer

- Name:** obj_npc_grocer
- Sprite:** spr_grocer_idle_d... (300 x 510)
- Collision Mask:** Same As Sprite
- Visible:** Checked
- Events:** None
- Parent:** None
- Physics:** None
- Variable Definitions:** loopRange01 (Default: 50, Type: Real), loopRange02 (Default: 550, Type: Real), myText ("Whoa, there! Slow down!", Type: String)

Updated myText values for the Baker, Teacher and Grocer.

6.8 Taking control away from the player

All three of our NPCs have their own dialogue now, which is great. However, there are a couple more things we need to do to make the textbox make sense:

- When a textbox is up, the player shouldn't be able to move (that'd just be rude)
- The textbox should disappear when the player presses the Space key again

Let's tackle the first part now.

An easy way to allow and disallow player control in situations is the same as what we did with controlling when our player says “hello”: create a variable and set it to *true* when you want the player to have control and set it to *false* when you don’t.

Open `obj_control` and in the Object Editor, click Add Event. Choose Other > Game Start.

This is a special Event that (as you might have guessed), runs when your game first starts. An Object that uses this Event must be placed in the first Room of your game and the Object must be placed with the Room Editor.

In the new Game Start Event, add this new code block to the end of the Event:

```
// Game variables  
global.playerControl = true;
```

This is called a *global* variable. It works the same as any other variable, except that every single Object in the game can look for it and change it.

Now open `obj_player` and its Step Event and edit the // Check keys for movement code block so it now looks like this:

```
// Check keys for movement  
if (global.playerControl == true) {  
    moveRight = keyboard_check(vk_right);  
    moveUp = keyboard_check(vk_up);  
    moveLeft = keyboard_check(vk_left);  
    moveDown = keyboard_check(vk_down);  
}  
if (global.playerControl == false) {  
    moveRight = 0;  
    moveUp = 0;  
    moveLeft = 0;  
    moveDown = 0;  
}
```

We're just implementing the new `global.playerControl` variable. If it's true, then the player can move as usual. If it's not, we take control away.

Next, go to `obj_player`'s Key Press - Space Event and edit the `// Create a textbox if NPC is nearby` code block like so:

```
// Create a textbox if NPC is nearby
if (nearbyNPC && global.playerControl == true) {
    _text = nearbyNPC.myText;
    if (!instance_exists(obj_textbox)) {
        iii = instance_create_depth(nearbyNPC.x, nearbyNPC.y-400, -10000, obj_textbox);
        iii.textToShow = _text;
    }
}
```

What we've done is change that first if statement to include a check for `global.playerControl`. Makes sense, right? If we shouldn't be able to move when player control has been taken away, we shouldn't be able to press the Space bar to do things either.

Finally, open `obj_textbox` again. In its Create Event, add this line to the `// Textbox variables` code block:

```
| global.playerControl = false;
```

Every time we create a `textbox` Object, we'll take control away from the player; that solves our first issue. But what do we do to get it back?

6.9 Closing our textbox

We're going to solve our second issue; to close the `textbox` when the player presses Space again and then allow the player to move once more.

Make sure you have `obj_textbox` open and click Add Event in the Object Editor. Add a Key Press - Space Event and write the following code in this Event:

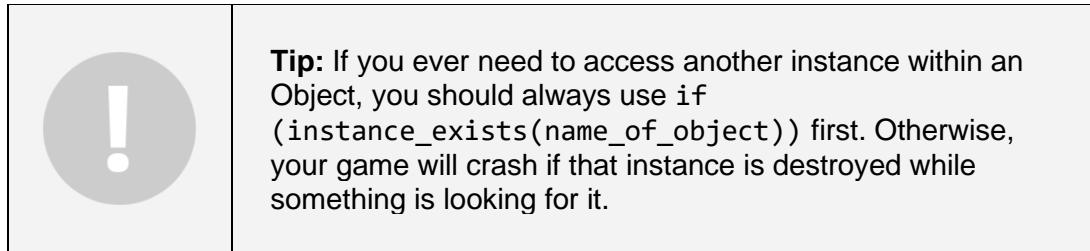
```
| // Queue my destruction
| alarm[0] = 2;
```

Next, add another Event: Alarm > Alarm 0. In this new Alarm Event, write the following:

```
| // Destroy me
```

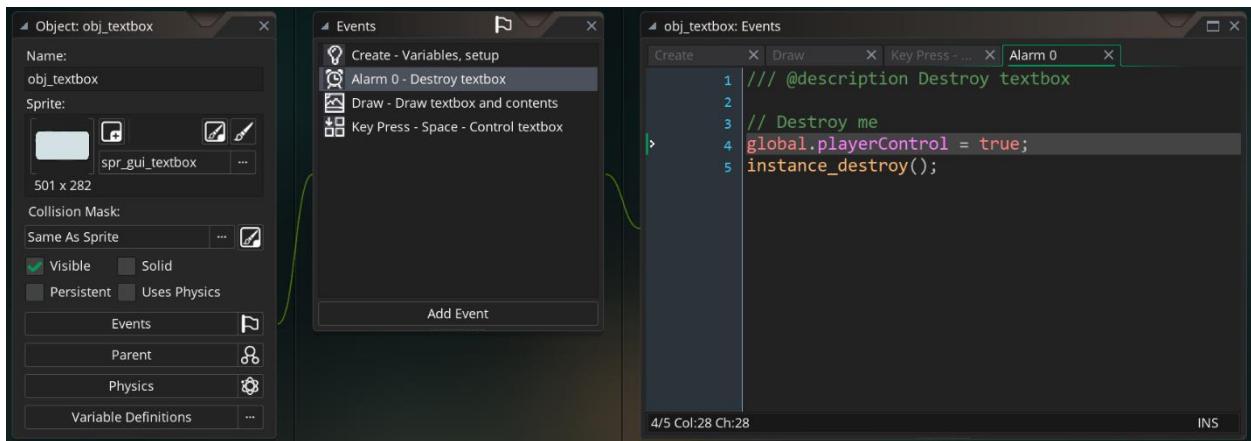
```
global.playerControl = true;  
instance_destroy();
```

`Instance_destroy()` is a powerful function that, by default, destroys the instance that calls it (you can also target other instances in a Room). And though code written in an Event *after* `instance_destroy()` is used can still technically run, it's a good idea not to. (You can read more in the GameMaker Studio 2 manual.)



So, with this setup, we're triggering Alarm 0 only 2 steps after we press the Space key. Why are we using an Alarm to do this? Shouldn't we just be able to close the textbox within the Key Press - Space Event itself?

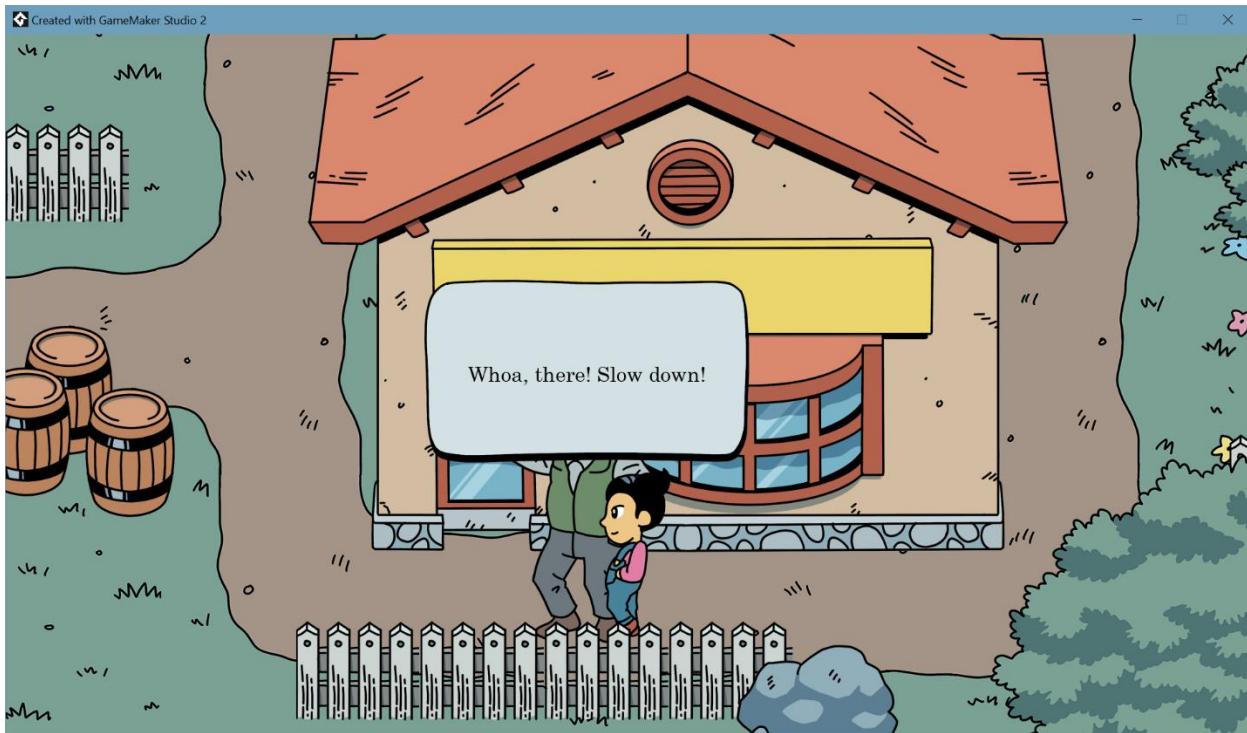
Well, you'd think so, but this is one of those quirks of programming. You see, pressing the Space Key when the player has control (in `obj_player`) and pressing it when the player doesn't have control (in `obj_textbox`) can happen at the same time! So just to be safe we're doing it this way.



Updating `obj_textbox` with a Key Press – Space Event and Alarm 0 Event.

In any case, once you've done all this, run the game again and talk to each of the three characters. You should notice that:

- All three NPCs say their own phrases
- The player cannot move when a textbox exists
- You can press Space to dismiss a textbox once it's up



Testing our updated textbox, which will now remove control from the player and can be closed by pressing the Space bar.

Once you're done checking everything, close the game window and return to GameMaker Studio. Make sure to save your project and let's move onward!

6.10 Adding effects to our textbox

Right now, our textbox just appears and disappears when we invoke it, which is a bit dull. How about we animate it a bit and have it fade in and out?

Open obj_textbox again and its Create Event. Add these new lines to the // Textbox variables code block:

```

fadeMe = 0;
fadeSpeed = 0.1;
image_alpha = 0;
```

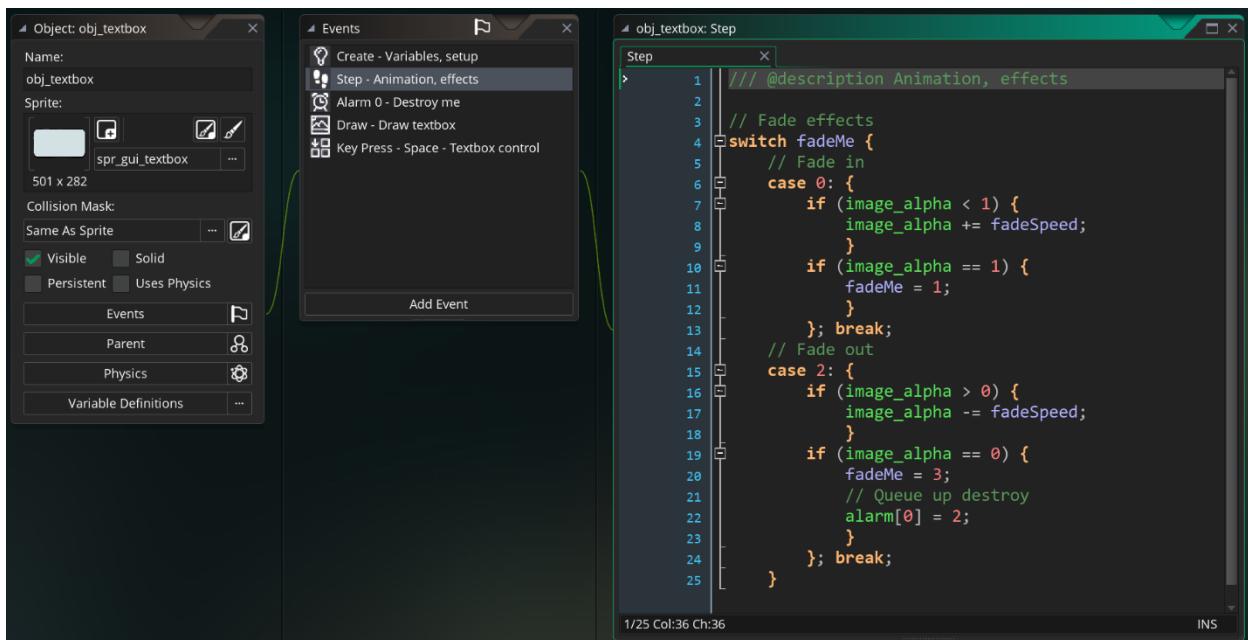
We're going to use these variables to control how the box fades in and out. Note that `image_alpha` is a built-in variable to set transparency (0 is completely transparent and 1 is completely opaque). By setting this here as 0, we start the textbox off as invisible, so we can fade it in.

Next, in the Object Editor, click Add Event and add a regular Step Event. In this Event, add this entire code block, which contains a switch statement:

```
// Fade effects
switch fadeMe {
    // Fade in
    case 0: {
        if (image_alpha < 1) {
            image_alpha += fadeSpeed;
        }
        if (image_alpha == 1) {
            fadeMe = 1;
        }
    }; break;
    // Fade out
    case 2: {
        if (image_alpha > 0) {
            image_alpha -= fadeSpeed;
        }
        if (image_alpha == 0) {
            fadeMe = 3;
            // Queue up destroy
            alarm[0] = 2;
        }
    }; break;
}
```

Let's break down a few things here: we've written switch statements before and this one is no different. So here we're saying, "if `fadeMe` is 0 and my `image_alpha` is less than 1, increase my `image_alpha` by a specific amount each step" (using `fadeSpeed`, which we set in the Create Event). Then, "once `image_alpha` reaches 1, set `fadeMe` to 1 to stop."

The next case, 2, does the opposite. It also sets that Alarm we created before — the one that destroys the textbox. So how do we set `fadeMe` to 2 so that this case will run?



Adding the fadeMe switch statement in obj_textbox's Step Event.

Open obj_textbox's Key Press - Space Event. Here, we previously set Alarm 0 if we pressed the Space bar. But we don't need this anymore since the Step Event takes care of this.

So, let's replace the entire // Queue my destruction code block with this:

```

// Begin fade out
if (fadeMe == 1) {
  fadeMe = 2;
}

/*
//// Old code
// Queue my destruction
alarm[0] = 2;
*/

```



Tip: Any code that's between /* and */ will be commented out. This is easier than using // comments on several lines. Here, we're commenting out our old code but keeping it for reference. (You'll do this a lot as you experiment and learn.)

Instead of immediately destroying the textbox Object, we're setting up the "fade out" effect that we just wrote in the Step Event. Perfect!

One last thing: open obj_textbox's Draw Event and make one small edit to the draw_text_ext_color() line in the // Draw text code block:

```
// Draw Text
draw_set_font(font_textbox);
draw_set_halign(fa_center);
draw_set_valign(fa_middle);
draw_text_ext_color(x,y,textToShow,lineHeight,textWidth,c_black,c_black,c_black,c_black,image_alpha);
```

All you're doing here is changing the last value, 1, to image_alpha. This is so that the text also respects the fade effect we just coded in.

Run the game again and walk up to any NPC and press the Space key. My goodness, it's beautiful!



The textbox object now fades in and out when it appears and disappears.



Tip: if you want to change the speed at which the textbox fades in and out, just change the value for fadeSpeed.

6.11 Adding a sound effect to the textbox

To complete our textbox, let's make sure it plays a sound effect when it pops open.

Open obj_textbox again and return to its Create Event.

Add the following after the // Textbox variables code block:

```
// Play UI sound  
audio_play_sound(snd_pop01,1,0);
```

This is the same sound function that we used in obj_control1 to play our background music and ambience. The only difference here is the last number (0), which is for the loop argument. By setting this to 0, the sound will only play once.

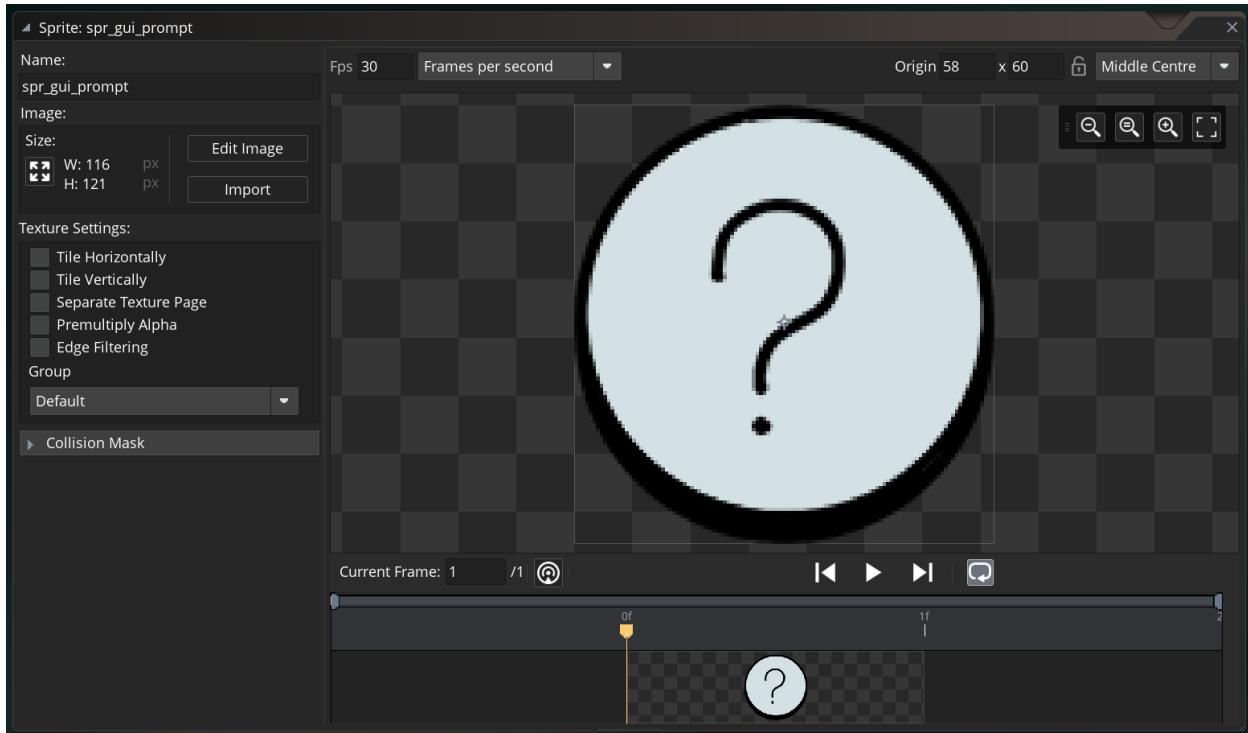
Run your game again to test out the textbox one more time; when it pops up, you should actually hear it pop!

6.12 Adding a visual prompt

Normally in a game, if a player can perform an action (like speak to an NPC), there is some kind of user interface (UI) element that lets them know beforehand.

So, in our game, we're going to have a little bubble pop up when we're within range of our NPCs that lets us know we can talk to them.

We already imported the Sprite for this new element, so let's open that now from the Asset Browser: it's called spr_gui_prompt. Make the origin point for this Sprite Middle Centre and then close the Sprite Editor.

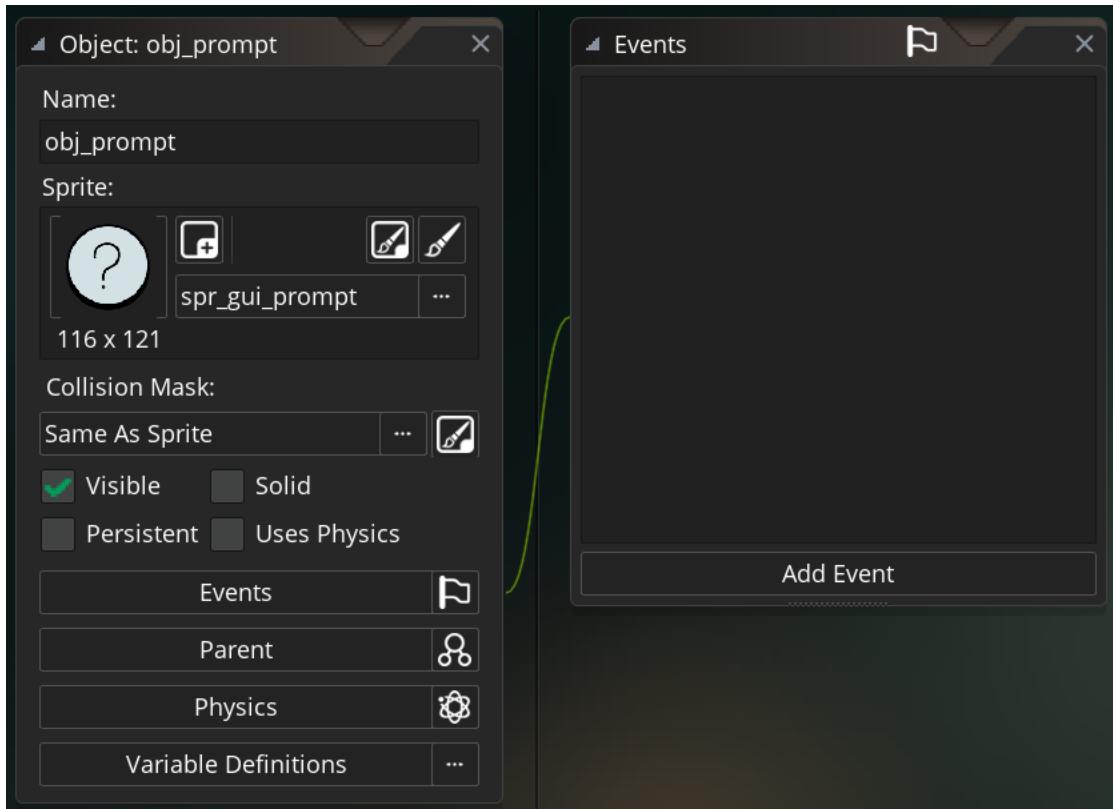


Setting the Origin for spr_gui_prompt

In the Asset Browser, create a new Object and call it obj_prompt.

In the Object Editor, attach spr_gui_prompt and close the Object. (We'll do more with this Object later, but for now, we just need it to exist and have a Sprite.)

	<p>Tip: Ongoing reminder that it's a good idea to organize any stray assets you have in your Asset Browser. We recommend creating a Group called UI within the Objects Group to keep Objects like obj_textbox and obj_prompt.</p>
---	--



The new `obj_prompt` doesn't need any Events just yet; it just needs to exist.

6.13 Creating our own functions within Script Assets

Sometimes in GameMaker Studio 2, you want to write code to do the exact same thing at different times, or for different reasons (like pop up a prompt to let the player know there's something interesting in front of them).

Instead of always writing a code block and then copying and pasting it where we need it, we can create our own functions and store them in *Script Assets*.

These Script Assets become part of the project itself, and the functions they contain behave just like the functions that are built into GameMaker Studio 2.

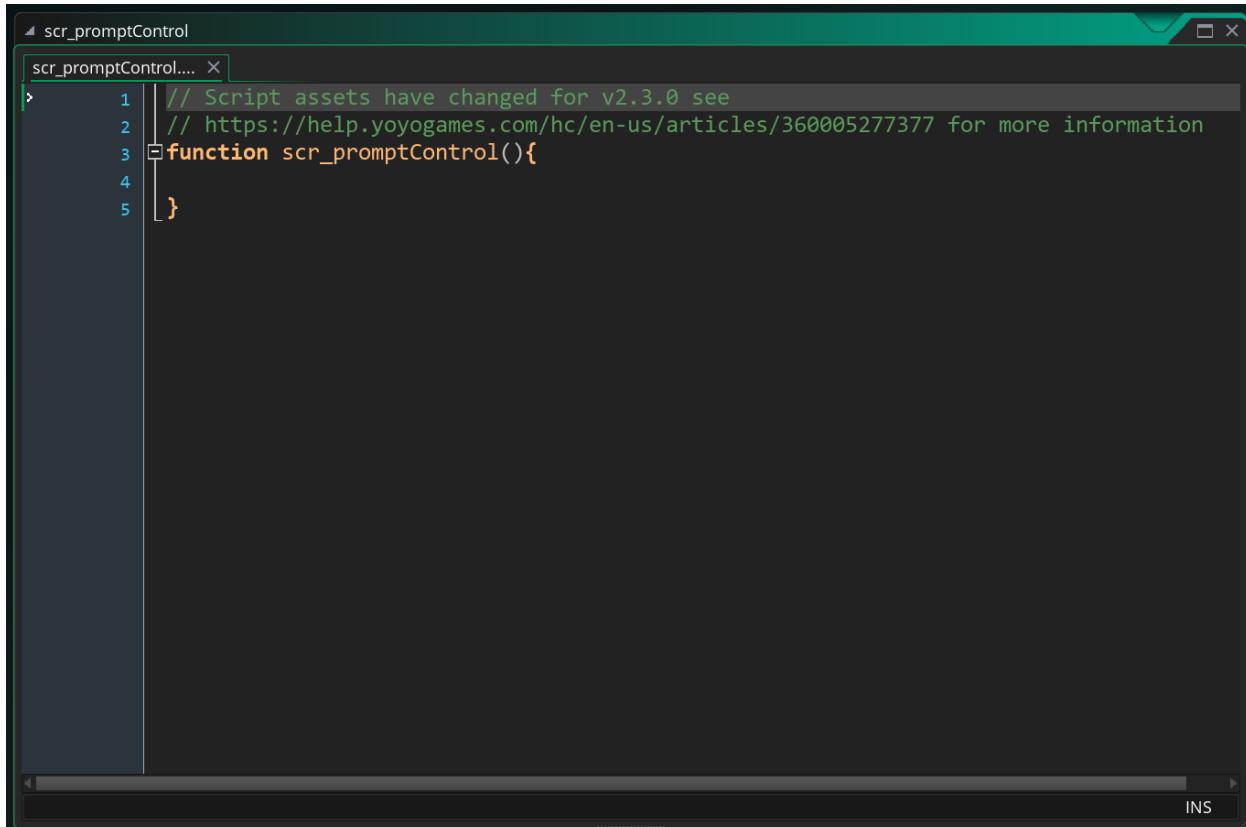
We're going to create a Script Asset for a couple of new functions — the first one will be for popping up that prompt Object we just made. As we continue through the course, you'll see that we'll re-use it in a few different places.

In the Asset Browser, right-click and choose Create > Script. Name this new Script `scr_promptControl` and drag it into the Scripts Group for safekeeping.

You'll see that GameMaker Studio 2 has opened the Script, which will include this by default:

```
function scr_promptControl(){  
}
```

This might seem a bit confusing at first, but GameMaker Studio 2 has created both a *Script Asset* and a *function* within that Script Asset with the same name.



The screenshot shows the GameMaker Studio 2 script editor window. The title bar says "scr_promptControl". The code area contains the following script:

```
// Script assets have changed for v2.3.0 see  
// https://help.yoyogames.com/hc/en-us/articles/360005277377 for more information  
function scr_promptControl(){  
}
```

By default, GameMaker Studio 2 creates a function inside a new Script asset with the same name.

However, we want our first function to have a different name, so update the code block like so:

```
function scr_showPrompt(){  
}
```

6.14 Writing a custom function

Are you ready to feel like a super-powerful developer? Because we're going to write our first function! (Spoiler: it's pretty easy.)

We've already had lots of experience with functions — specifically, *runtime functions*, which are the built-in functions GameMaker Studio 2 already has, such as `instance_create_depth()`. And just like those, we can create a function that's very simple or one that has lots of options.

For now, edit the `scr_showPrompt()` function we just made like so:

```
function scr_showPrompt(_object,_x,_y) {
    if (instance_exists(_object)) {
        if (!instance_exists(obj_textbox) && !instance_exists(obj_prompt)) {
            iii = instance_create_depth(_x,_y,-1000,obj_prompt);
            return iii;
        }
    }
}
```

This is mostly code we've written before: here, we're making sure that a textbox doesn't exist *and* that an instance of the prompt Object we want to pop up doesn't already exist.

If both of those things are true, we're creating an instance of the prompt at a certain spot and at a depth that will put it close to the Camera. (This is just like the code we wrote back in [Giving our NPCs a voice of their own](#).)

6.14.1 How functions work

However, it appears there's a new twist. What are `_object`, `_x` and `_y`? As we covered back in [Changing all our movement Events to code](#), these are called *arguments* — parameters that a function needs to work.

It's like ordering lunch: you might ask for a sandwich, but your server is going to ask you which sandwich, what kind of bread you want and would you like pickles on that?

If this example transaction was a function, it might look like this:

```
| scr_orderSandwich(_whatKind,_whichBread,_pickles);
```

If we don't tell the server all this information, they can't process our order and we don't get our sandwich.

For our `scr_showPrompt()` function, we're asking three questions; which Object is our player looking at (`_object`), at which x position (`_x`) and y position (`_y`) should we create a prompt Object?

6.15 Returning the value of a function

You can also see that we're using another `iii = something` line here. This is so we store the unique instance ID of the prompt Object we create for later use. But below that you'll see something new:

```
| return iii;
```

In a function, you can, say, do a bunch of math and *return* the result of all that math at the end. This allows whatever Object ran the function in the first place to take that result and do something with it.

Here we're returning the unique instance ID of the `obj_prompt` we create. If we don't do this, our player Object won't have any idea what it is (even though it's going to run the function).

	<p>Tip: a function ends with a <code>return</code> line. Any code after that line won't be executed.</p>
--	---

6.16 Using the `scr_showPrompt` function

Let's implement this new function so we can understand how it all works.

Open `obj_player` and its Create Event; we need to add one new variable, so add a new line to the // Variables code block:

```
| npcPrompt = noone;
```

Now open obj_player's Step Event.

Update the // Check for collision with NPCs code block like so:

```
// Check for collision with NPCs
nearbyNPC = collision_rectangle(x-lookRange,y-
lookRange,x+lookRange,y+lookRange,obj_par_npc,false,true);
if nearbyNPC {
    // Play greeting sound
    if (hasGreeted == false) {
        if !(audio_is_playing(snd_greeting01)) {
            audio_play_sound(snd_greeting01,1,0);
            hasGreeted = true;
        }
    }
    // Pop up prompt
    if (npcPrompt == noone || npcPrompt == undefined) {
        npcPrompt = scr_showPrompt(nearbyNPC,nearbyNPC.x,nearbyNPC.y-450);
    }
    show_debug_message("obj_player has found an NPC!");
}
if !nearbyNPC {
    // Reset greeting
    if (hasGreeted == true) {
        hasGreeted = false;
    }
    show_debug_message("obj_player hasn't found anything");
}
```

Hopefully, this makes more sense now: we want to create an instance of obj_prompt at a specific x and y position, but we want to do it based on an NPC we're standing next to.

	<p>Tip: we're checking here if npcPrompt is noone or if it's undefined just to be thorough. You can read more about undefined values in the manual.</p>
---	--

But there's one final wrinkle here; let's explain this line:

```
| npcPrompt = scr_showPrompt(nearbyNPC,nearbyNPC.x,nearbyNPC.y-450);
```

With this diagram:

The screenshot shows the Construct 3 IDE interface with three main windows:

- scr.promptControl**: A script window containing the following code:

```

1 // @description Manage our UI prompts
2
3 function scr_showPrompt(_object,_x,_y) {
4     if (instance_exists(_object)) {
5         if (!instance_exists(obj_textbox) && !instance_exists(obj_prompt)) {
6             iii = instance_create_depth(_x,_y,-10000,obj_prompt);
7             return iii;
8         }
9     }
10}

```
- Object: obj_player**: An object properties window for the player character. It shows the sprite is spr_player_idle_d... and has a collision mask of Same As Sprite.
- obj_player: Events**: An events script window for the player. It contains an event for "Step" which triggers a greeting sound and calls the scr_showPrompt function:

```

66 // Play greeting sound
67 if (hasGreeted == false) {
68     if (!audio_is_playing(snd_greeting01)) {
69         audio_play_sound(snd_greeting01,1,0);
70         hasGreeted = true;
71     }
72
73 // Pop up prompt
74 if (npcPrompt == none || npcPrompt == undefined) {
75     npcPrompt = scr_showPrompt(nearbyNPC,nearbyNPC.x,nearbyNPC.y-450);
76 }
77 show_debug_message("obj_player has found an NPC!");
78
79 if !nearbyNPC {
80     // Reset greeting
81     if (hasGreeted == true) {
82         hasGreeted = false;
83     }
84     show_debug_message("obj_player hasn't found anything");
85 }
86
87 // Depth sorting
88 depth = -y;
89

```

A callout arrow points from the line "return iii;" in the scr.promptControl script to the line "scr_showPrompt(nearbyNPC,nearbyNPC.x,nearbyNPC.y-450);" in the obj_player Events script. The text "This gets returned here" is enclosed in a box at the end of the arrow.

When a function *returns* a result, that result can be stored in a variable by using the same `someVariable = fancyFunction();` technique we mentioned before.

Run the game and walk over to one of the three characters. As you get within range, you should see our prompt pop up. Hurray!



Testing to confirm that obj_prompt appears above our NPCs when the player Object is near.

6.17 Adding effects to the prompt icon

Of course, you may notice our prompt has a few problems. We still need to:

- Add fade effects as we did with the textbox
- Add a sound effect when the prompt pops up
- Make it disappear if we walk away from the NPC
- Make it disappear if a textbox pops up

Let's tackle the first two problems. We're going to edit `obj_prompt` to have a similar fade system as `obj_textbox`, and give it a simple sound effect.

Open `obj_prompt` in the Object Editor. Click Add Event and add a Create Event. In this new Event, write the following to start:

```
// Prompt variables  
fadeMe = "fadeIn";  
fadeSpeed = 0.1;  
image_alpha = 0;
```

As you can see, we're using some simple variables here the same way we did in `obj_textbox`. But what's up with that `fadeMe` variable? We'll explain that momentarily.

Next, add the following after the // Prompt variables code block:

```
// Play UI sound  
audio_play_sound(snd_pop02,1,0);
```

Just as we did with the textbox, we've now added a simple sound effect to the prompt.

Now, in the Object Editor, click Add Event again and add a Step Event. In it, add the following code block:

```
// Fade effects  
switch fadeMe {  
    // Fade in  
    case "fadeIn": {  
        if (image_alpha < 1) {  
            image_alpha += fadeSpeed;  
        }  
        if (image_alpha >= 1) {  
            fadeMe = "fadeVisible";  
        }  
    }; break;  
    // Fade out  
    case "fadeOut": {  
        if (image_alpha > 0) {  
            image_alpha -= fadeSpeed;  
        }  
        if (image_alpha <= 0) {  
            fadeMe = "fadeDone";  
            alarm[0] = 2; // Queue up destroy  
        }  
    }; break;  
}
```

This is essentially the same code as what was in obj_textbox. However, there's one important difference: here, instead of using numbers as values for our fadeMe variable, we've used text (a *string*, remember).

The screenshot shows three panels of the GameMaker Studio 2 interface:

- Object Properties Panel:** Shows the object name as "obj_prompt", sprite as "spr_gui_prompt", and collision mask as "Same As Sprite".
- Events List:** Shows the "Step - Animate, fade" event selected.
- Code Editor:** Displays the GML code for the Step event. The code uses a switch statement based on the variable "fadeMe" to handle fading in and out.

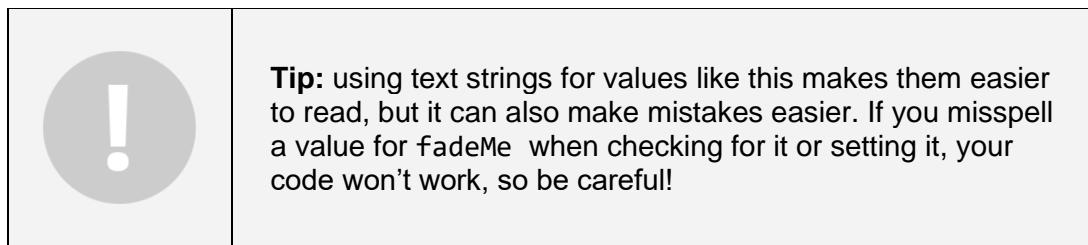
```

// @description Animate, fade
switch fadeMe {
    case "fadeIn": {
        if (image_alpha < 1) {
            image_alpha += fadeSpeed;
        }
        if (image_alpha >= 1) {
            fadeMe = "fadeVisible";
        }
    }; break;
    case "fadeOut": {
        if (image_alpha > 0) {
            image_alpha -= fadeSpeed;
        }
        if (image_alpha <= 0) {
            fadeMe = "fadeDone";
            alarm[0] = 2; // Queue up destroy
        }
    }; break;
}
  
```

Adding the fadeMe switch statement in obj_prompt's Step Event.

So instead of 0, 1 and 2, we've got easy-to-read labels like fadeIn, fadeVisible and fadeOut. Why did we do this and not use numbers as we did with obj_textbox?

There's no secret trick here: this is just to demonstrate the flexibility you have with variables and their values. As far as GameMaker Studio 2 is concerned, either solution is fine, so experiment with what makes sense for you.



To complete our fade effect, we need to add an Alarm, just like we did with obj_textbox; you can see in the code we just wrote into the Step Event that Alarm 0 is being called in the fadeOut case. Let's try something new to get this done quickly!

Open obj_textbox and look at the list of Events for the Object.

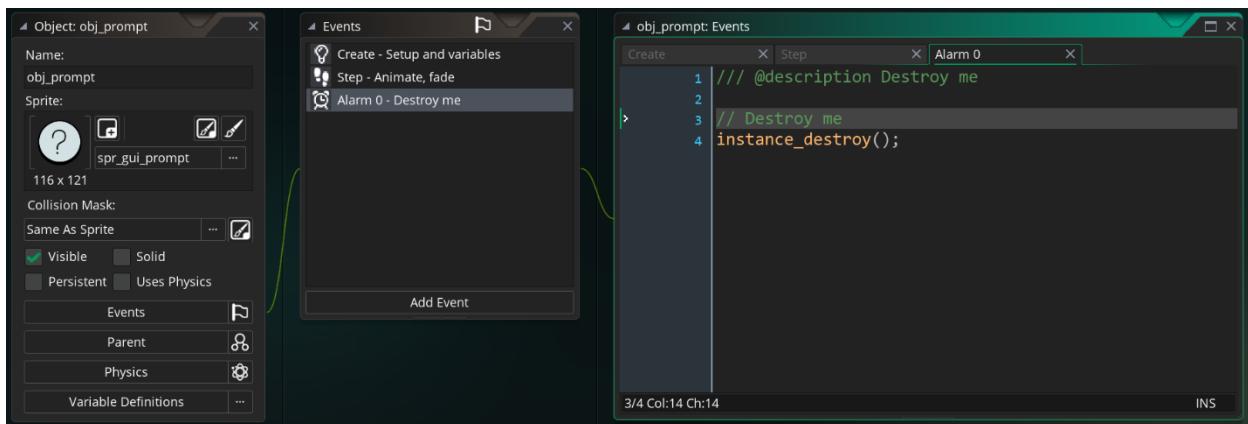
Right-click on Alarm 0 and choose Copy Event.

Open obj_prompt again and in the Object Editor, right-click on a blank space within the Events window. Choose Paste Event and voila!— you can easily copy and paste Events across Objects.

Open this new Alarm 0 Event and delete this line:

```
| global.playerControl = true;
```

We don't need this in obj_prompt since it's not supposed to remove control from the player.



After pasting the Alarm 0 Event to obj_prompt, we remove one line of code.

With that done, run your game again and walk over to one of the characters. You should see the prompt Object nicely fade in when you get close.



Testing our prompt again; it should now fade in nicely instead of appearing suddenly.

When you're done, close the game window and return to GameMaker Studio 2.

6.18 Making the prompt go away

Things are looking lovely now, but we still need to solve those second and third problems and make sure our prompt disappears when we walk away from a character, or when a textbox appears.

Let's tackle this with another function within that `scr_promptControl` Script Asset, so whenever we want to get rid of `obj_prompt`, we can.

Open `scr_promptControl` and add the following function code block beneath the `scr_showPrompt()` function we already wrote:

```
function scr_dismissPrompt(){  
}  
}
```

Let's edit this function so it now looks like this:

```

function scr_dismissPrompt(_whichPrompt,_toReset) {
    if (_whichPrompt != undefined) {
        if (instance_exists(_whichPrompt)) {
            // Tell prompt Object to fade out
            with (_whichPrompt) {
                fadeMe = "fadeOut";
            }
            // Reset appropriate prompt variable
            if (instance_exists(obj_player)) {
                with (obj_player) {
                    switch _toReset {
                        // Reset npcPrompt
                        case 0: npcPrompt = noone; break;
                    }
                }
            }
        }
    }
}

```



Tip: The “with” statement is a powerful feature of GameMaker Studio 2. It lets one instance temporarily duck into another to execute code. In this case, this function is going to execute code within obj_prompt and set its fadeMe variable to the desired value. It’s also going to reset that npcPrompt variable that we just created in obj_player.

This function has two arguments: `_whichPrompt` and `_toReset`. The second argument might seem redundant right now, but we’ll need this in the next session. You’ll see we’ve got a switch statement within the function, though it only has one case. We’ll expand on that later.

Now, let’s use our new function to solve our second and third problems. Open `obj_player` and its Step Event. Look under the `// Check for collision with NPCs` code block for the `!nearbyNPC` code block, and update it like so:

```

if !nearbyNPC {
    // Reset greeting
    if (hasGreeted == true) {
        hasGreeted = false;
    }
    // Get rid of prompt
    scr_dismissPrompt(npcPrompt,0);
}

```

```
    show_debug_message("obj_player hasn't found anything");
}
```

You can see that we're using the `npcPrompt` variable that we get from our `scr_showPrompt()` function; this is so we target the exact instance of the prompt Object that we make there.

Then, open `obj_textbox` and its Create Event. After the `// Textbox variables` code block, add this code:

```
// Dismiss any visible prompts  
scr_dismissPrompt(obj_prompt,0);
```

(Note that here we're not targeting any specific instance of `obj_prompt`, but any that are open, since `obj_textbox` doesn't know what our player has been up to.)

Run your game again and walk over to one of the characters and then step away. The prompt should correctly pop up when you're nearby and disappear when you're not.

Test it again by pressing the Space bar to bring up a textbox; it should disappear when the textbox pops up, and reappear when you press Space to close the textbox.

Now, thanks to our little function, we can get rid of that prompt at any point, with only one line of code!

6.19 Bonus: animating the prompt

Let's do one last thing before we finish this session. Just to add some pizazz to our prompt Object, let's make it animate. We'll make it bob up and down to really draw the player's attention.

Open `obj_prompt` and its Create Event. Add the following lines of code after the `// Play UI sound` code block:

```
// Set up bobbing effect  
shift = 1;  
alarm[1] = 10;
```

Next, in the Object Editor, click Add Event and choose Alarm > Alarm 1. In this new Event, add this code block:

```
// Change bob direction  
shift = -shift;
```

```
| alarm[1] = 10;
```

Finally, open `obj_prompt`'s Step Event and add this code at the top of the Event, before the `// Fade effects` code block:

```
| // Bob up and down  
| y += shift;
```

This is another way to use Alarms to do something fun. Every 10 steps, Alarm 1 will fire, reversing the value of `shift` (from 1 to -1, back to 1 and so on). The `obj_prompt` will constantly be adjusting its `y` value by whatever `shift` is, so it will move down, then up, then down and so on, creating a simple bobbing effect.

Run the game again and get close to an NPC. Now when the prompt appears, it bobs up and down. Feel free to play around with the `shift` and `Alarm 1` values to see what happens.



The prompt Object should now bob up and down when it appears.

6.20 End of Session 3

That's it for this session. In the next one, we're going to make items that we can take and more. Don't forget to save your project!

7 Session 4

In this session, we’re going to make Objects that we can take and bring to our three characters. We’ll make it so our player Object knows which item it’s carrying and allow us to drop an item we don’t want. We’re also going to add the ability for our player to run and have the weight of each Object affect its speed. In doing so, we’re also going to learn about new data structures and techniques for keeping our code manageable.

7.1 Making a basic item Object

We’re going to have six different items in our town for the player to take, with their own Sprites, names and values. So, let’s build on what did back in [Object parenting for easy editing](#) and create a parent Object.

In the Asset Browser, right-click and choose Create > Object. Name this new Object `obj_par_item`.

Open `obj_par_item` if it’s not opened already and click “Add Event,” and create a regular Step Event. As we have done a few times before, simply add a line of code to deal with depth sorting:

```
// Depth sorting  
depth =-y;
```

Let’s add some Variable Definitions to our item parent that we’ll use a little later in the session. In the Object Editor, click Variable Definitions and then Add.

Name this first variable `itemName`, change its Type to be a String and type “nothing” (with quotation marks) as its Default value.

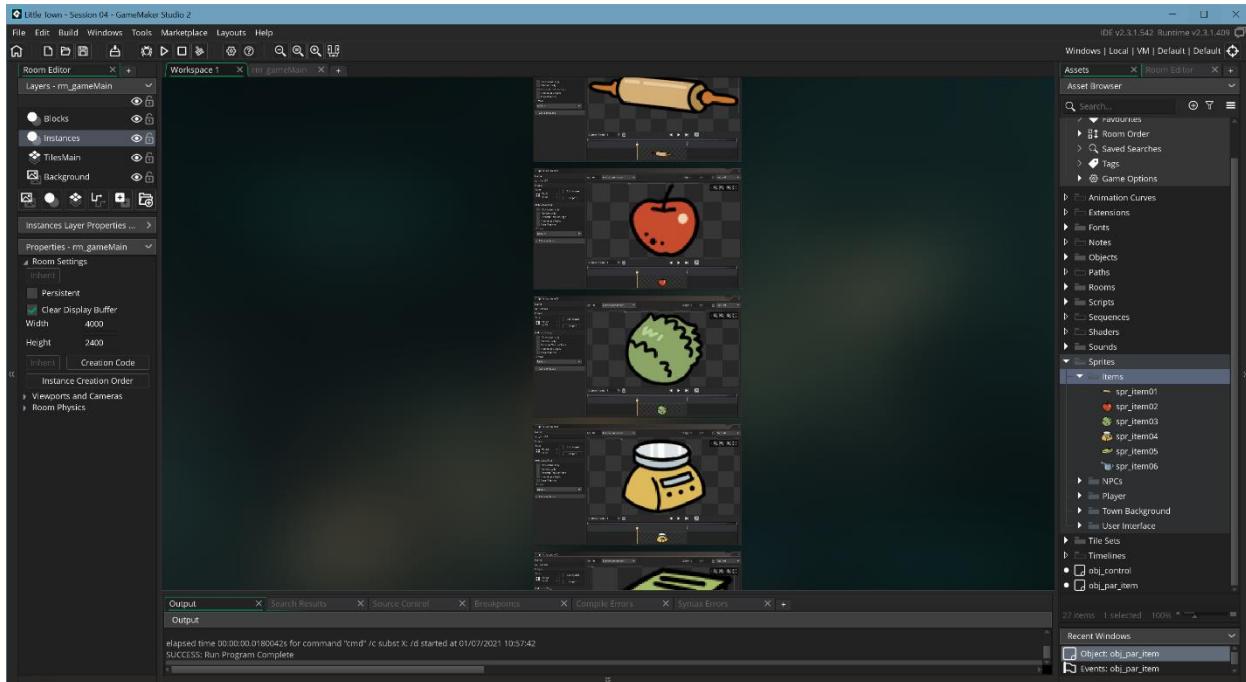
Add a second variable; name it `itemWeight`, make sure its Type is Real and change its Default value to 2.

7.2 Create our first item

Using File Explorer (Windows) or Finder (Mac), navigate to the Assets folder provided with this course and open the Sprites > Characters and Items folder. You’ll see six item Sprites named:

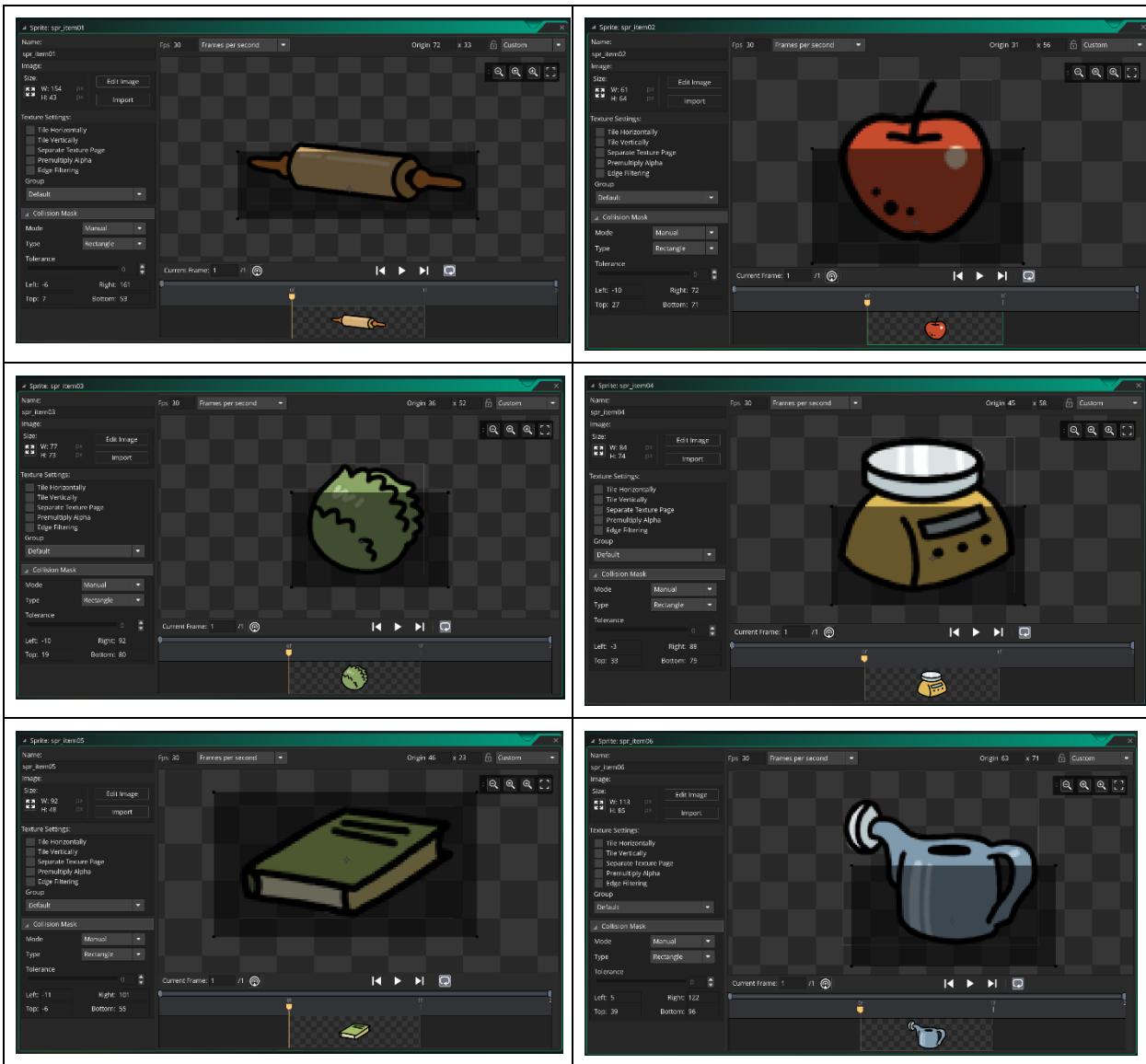
- spr_item01
- spr_item02
- spr_item03
- spr_item04
- spr_item05
- spr_item06

Drag all six of these Sprites into GameMaker Studio 2's Asset Browser. To keep things tidy, right-click in the Asset Browser and choose Create Group; make a new group for these Sprites (called Items) and drag this group within the pre-existing Sprites group.



Importing our item Sprites and organizing them within the Asset Browser.

As we've done before, open each of these six new Sprites and adjust their origin points and collision masks, like so:



It's okay to make the collision masks a little generous.

Once you're done, right-click in the Asset Browser again and create a new Object. Call it `obj_item01`.

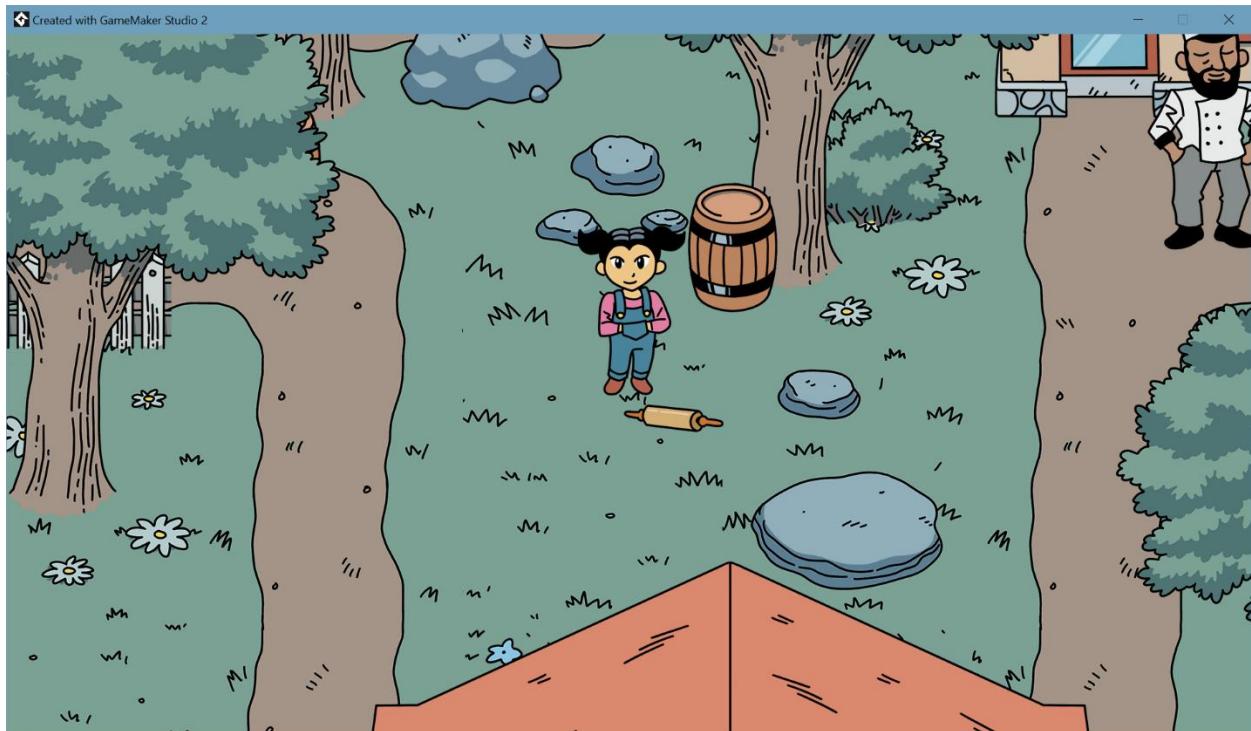
In the Object Editor for this new Object, click on Parent and click the drop-down menu that says "none." Choose the Object we just made, `obj_par_item`, so that our new item Object becomes its child.

Attach the `spr_item01` Sprite (the rolling pin) to this new Object.

Click on Variable Definitions and then the edit button beside the `itemName` variable. Replace the default value with “Rolling Pin”. (You can leave the `itemWeight` variable as is for now.)

Finally, open `rm_gameMain` and make sure the Instances layer is selected in the Room Editor. Drag an instance of `obj_item01` into the Room.

Run your game and have your player walk around and over the item to make sure the item behaves correctly in terms of depth. If not, you may have to open its Sprite again to adjust its origin point.



The new `obj_item01`, placed in the town.

7.3 Recognizing the items

What we want to do next is enable the player Object to recognize when it’s standing next to an item and pop up a little prompt. Thankfully, we’ve already done this with our NPCs (back in [Collisions](#)), so this should be a snap.

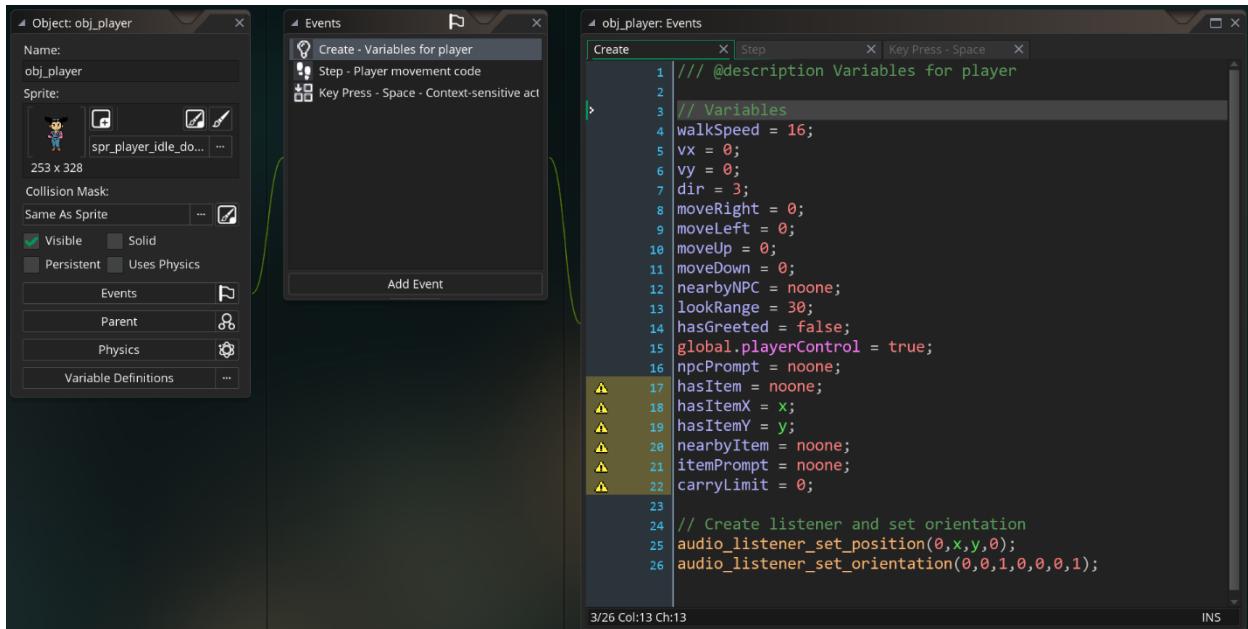
First, Open `obj_player` and its Create Event. Add the following lines to the // Variables code block to create some variables we’re going to need:

```
| hasItem = noone;
```

```

hasItemX = x;
hasItemY = y;
nearbyItem = noone;
itemPrompt = noone;
carryLimit = 0;

```



Our player Object has quite the collection of variables in its Create event.

Next, Open obj_player's Step Event. Find the entire // Check for collision with NPCs code block; we're going to write a near-duplicate of this block but change it to look for our new item Objects instead.

You can either copy and paste this block and then edit it to match, or you can write this out manually.

Either way, create a new code block below the // Check for collision with NPCs code block, like so:

```

// Check for collision with Items
nearbyItem = collision_rectangle(x-lookRange,y-
lookRange,x+lookRange,y+lookRange,obj_par_item,false,false);
if (nearbyItem) {
    // Pop up prompt
    if (itemPrompt == noone || itemPrompt == undefined) {
        show_debug_message("obj_player has found an item!");
        itemPrompt = scr_showPrompt(nearbyItem,nearbyItem.x,nearbyItem.y-300);
    }
}

```

```

        }
    if (!nearbyItem) {
        // Get rid of prompt
        scr_dismissPrompt(itemPrompt,1);
    }
}

```

```

Step X
61    }
62
63    // Check for collision with NPCs
64    nearbyNPC = collision_rectangle(x-lookRange,y-lookRange,x+lookRange,y+lookRange,obj_par_npc,false,true);
65    if (nearbyNPC {
66        // Play greeting sound
67        if (hasGreeted == false) {
68            if !(audio_is_playing(snd_greeting01)) {
69                audio_play_sound(snd_greeting01,1,0);
70                hasGreeted = true;
71            }
72        }
73        // Pop up prompt
74        if (npcPrompt == noone || npcPrompt == undefined) {
75            npcPrompt = scr_showPrompt(nearbyNPC,nearbyNPC.x,nearbyNPC.y-450);
76        }
77        show_debug_message("obj_player has found an NPC!");
78    }
79    if !nearbyNPC {
80        // Reset greeting
81        if (hasGreeted == true) {
82            hasGreeted = false;
83        }
84        // Get rid of prompt
85        scr_dismissPrompt(npcPrompt,0);
86        show_debug_message("obj_player hasn't found anything");
87    }
88
89    // Check for collision with Items
90    nearbyItem = collision_rectangle(x-lookRange,y-lookRange,x+lookRange,y+lookRange,obj_par_item,false,true);
91    if (nearbyItem {
92        // Pop up prompt
93        if (itemPrompt == noone || itemPrompt == undefined) {
94            show_debug_message("obj_player has found an item!");
95            itemPrompt = scr_showPrompt(nearbyItem,nearbyItem.x,nearbyItem.y-300);
96        }
97    }
98    if (!nearbyItem {
99        // Get rid of prompt
100       scr_dismissPrompt(itemPrompt,1);
101    }

```

You can duplicate the NPC collision code and edit it, or start from scratch to create this new item collision code

You'll notice that what we're doing here is:

- Looking for instances of `obj_par_item` (instead of `obj_par_npc`)
- Checking for our new variable, `itemPrompt` (instead of `npcPrompt`)
- Changing the arguments used in `scr_showPrompt()` and `scr_dismissPrompt()` to reflect our item variables, instead of the variables for NPCs

There's one more step to make our prompts work correctly with items: we need to reset the `nearbyItem` variable found in `obj_player` when we walk away from an item.

Notice that in our new code block, for `// Get rid of prompt`, we have:

```

if (!nearbyItem) {
    // Get rid of prompt
}

```

```
    scr_dismissPrompt(itemPrompt,1);
}
```

The second argument for `scr_dismissPrompt()` here is a `1`, not a `0`, as it was in the `// Check for collision with NPCs` code block. However, currently the `scr_dismissPrompt` function doesn't know what to do with that `1`, so let's fix that.

Open `scr_promptControl` from the Asset Browser. Update the `scr_dismissPrompt` function like so:

```
function scr_dismissPrompt(_whichPrompt,_toReset) {
    if (_whichPrompt != undefined) {
        if (instance_exists(_whichPrompt)) {
            // Tell prompt Object to fade out
            with (_whichPrompt) {
                fadeMe = "fadeOut";
            }
            // Reset appropriate prompt variable
            if (instance_exists(obj_player)) {
                with (obj_player) {
                    switch _toReset {
                        // Reset npcPrompt
                        case 0: npcPrompt = noone; break;
                        case 1: itemPrompt = noone; break;
                    }
                }
            }
        }
    }
}
```

We're adding a case (1) to the switch statement here and resetting `itemPrompt` in `obj_player`.

Run your game again and walk up to and away from the item we placed in the Room. Just like with the NPCs, you should see that familiar (?) prompt pop up, bob up and down, and fade away when you walk away from the item.



Items now should also pop up our prompt object.

When you're ready, close the game window and return to GameMaker Studio 2.

7.4 States and enums

Up until now, we've been using a simple system for changing which Sprites our player Object displays depending on whether it's moving or not and in which direction (back in [Changing our player Sprites](#).)

But as you can see, once you start adding more and more features to your Objects, the amount of code you must write increases. In order to keep our player Object manageable, we're going to change how we accomplish certain tasks (like change Sprites, among other things).

So, we're going to create *States* for our player and then make the player accomplish multiple things at once, depending on those States.

To do this, we're going to use something new: *enums*.

Open obj_control's new Game Start Event and add this code block:

```
| // Player states
```

```
enum playerState {  
    idle,  
    walking,  
    pickingUp,  
    carrying,  
    carryIdle,  
    puttingDown,  
}
```

We've created an *enum* called `playerState`. An *enum* is a collection of *constant* values — meaning that once it's declared, it cannot be changed while a game is running.

Enums are great for creating your own “built-in” values that you can reference from anywhere, because enums are inherently global (notice how the enum we've created here is not called `global.playerState`).

In the `playerState` enum we've added several entries, which we're going to use as our player states: `idle`, `walking`, `pickingUp` and so on.

	<p>Tip: What do these entries mean? Well, by themselves, absolutely nothing. They're just labels we've set up so we can reference them later. However, by default, these entries also have an integer value; <code>idle</code> is 0, <code>walking</code> is 1, <code>pickingUp</code> is 2 and so on. (Note that if you change the order of the entries, their integer value will also change.)</p>
--	---

With this new enum in place, let's return to `obj_player`.

7.5 Setting up our player states

Open `obj_player`'s Create Event and add this line to the list of variables in the `// Variables` code block:

```
| myState = playerState.idle;
```

Here, we're creating a new variable (`myState`) and setting it to the `idle` entry within the `playerState` enum. By doing so, we have set our player “state” to “idle.”

But just like with any other variable, that doesn't mean much if we don't use it.

Open obj_player's Step Event and look at the // If moving code block.

We used some if statements to check the player's movement and apply a Sprite based on that (see the screenshot below for a reminder):

```
Step X
21 // If Idle
22 if (vx == 0 && vy == 0) {
23     // Change idle Sprite based on last direction
24     switch dir {
25         case 0: sprite_index = spr_player_idle_right; break;
26         case 1: sprite_index = spr_player_idle_up; break;
27         case 2: sprite_index = spr_player_idle_left; break;
28         case 3: sprite_index = spr_player_idle_down; break;
29     }
30 }
31
32 // If moving
33 if (vx != 0 || vy != 0) {
34     if !collision_point(x+vx,y,obj_par_environment,true,true) {
35         x += vx;
36     }
37     if !collision_point(x,y+vy,obj_par_environment,true,true) {
38         y += vy;
39     }
40
41     // Change walking Sprite based on direction
42     if (vx > 0) {
43         sprite_index = spr_player_walk_right;
44         dir = 0;
45     }
46     if (vx < 0) {
47         sprite_index = spr_player_walk_left;
48         dir = 2;
49     }
50     if (vy > 0) {
51         sprite_index = spr_player_walk_down;
52         dir = 3;
53     }
54     if (vy < 0) {
55         sprite_index = spr_player_walk_up;
56         dir = 1;
57     }
58 }
```

We're currently using a couple of different ways to change our player Object's Sprite

Since we're about to give the player several new abilities (to pick up an item, to put it down and even to run), this could get messy very quickly.

Edit the // If moving code block to remove each line where we specify the sprite_index, and to add a line that will change the player's state whenever it is moving, like so:

```
// If moving
if (vx != 0 || vy != 0) {
    if !collision_point(x+vx,y,obj_par_environment,true,true) {
        x += vx;
    }
    if !collision_point(x,y+vy,obj_par_environment,true,true) {
        y += vy;
    }

// Change direction based on movement
if (vx > 0) {
    dir = 0;
}
if (vx < 0) {
    dir = 2;
}
if (vy > 0) {
    dir = 3;
}
if (vy < 0) {
    dir = 1;
}

// Set state
myState = playerState.walking;

// Move audio listener with me
audio_listener_set_position(0,x,y,0);
}
```

Next, we're going to simplify the // If idle code block to also remove specific Sprite declarations. Update the // If idle code block like so:

```
// If Idle
if (vx == 0 && vy == 0) {
    myState = playerState.idle;
}
```

Run the game again and you'll see we've taken a step backwards. Whether you move the player around or stand still, obj_player will only ever show the default idle Sprite. That's okay, because we're going to do this with another new technique, an array.



We're back to having a player Object that doesn't change Sprites correctly — but we're about to fix this.

7.6 Creating a simple array

Arrays are a great way to store data (like scores, hit points, asset names and more) and retrieve that data easily. (You can read more about them in the GameMaker Studio 2 manual.)

Think of them like invisible spreadsheets we can both write data to and read data from; they can have a single “cell” or be *multi-dimensional* and have as many “cells” (horizontally or vertically) as we need.

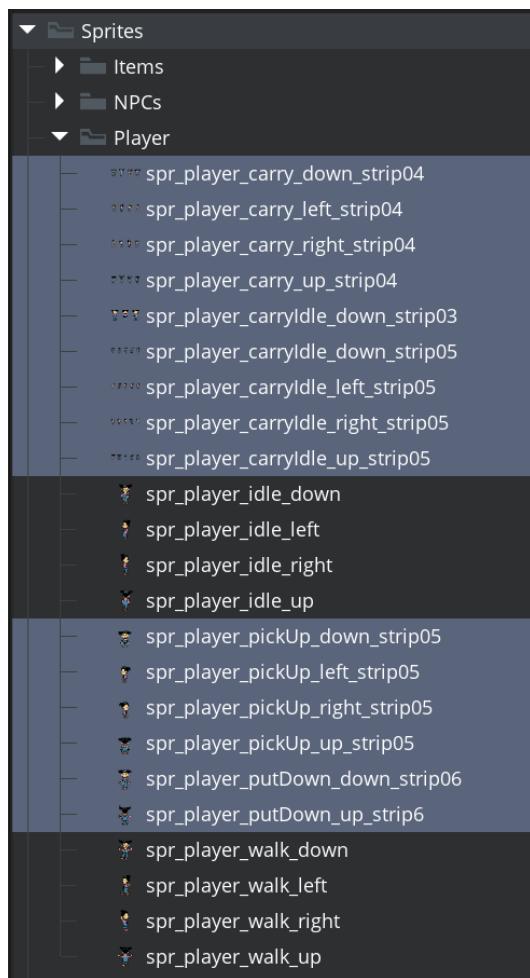
We're going to create an array within our player Object that holds information on Sprite assets and then use a simple expression to automatically display the correct player Sprite depending on whether the player is idle, moving, picking something up and more — and take into account the direction it's facing!

First, we're going to need to import a few more Sprite assets into GameMaker Studio 2. Using File Explorer (Windows) or Finder (Mac), navigate to the Assets folder provided with this course and find the Characters and Items folder.

Drag in the following Sprites into the Sprites > Player group in the Asset Browser:

- spr_player_carry_down_strip04

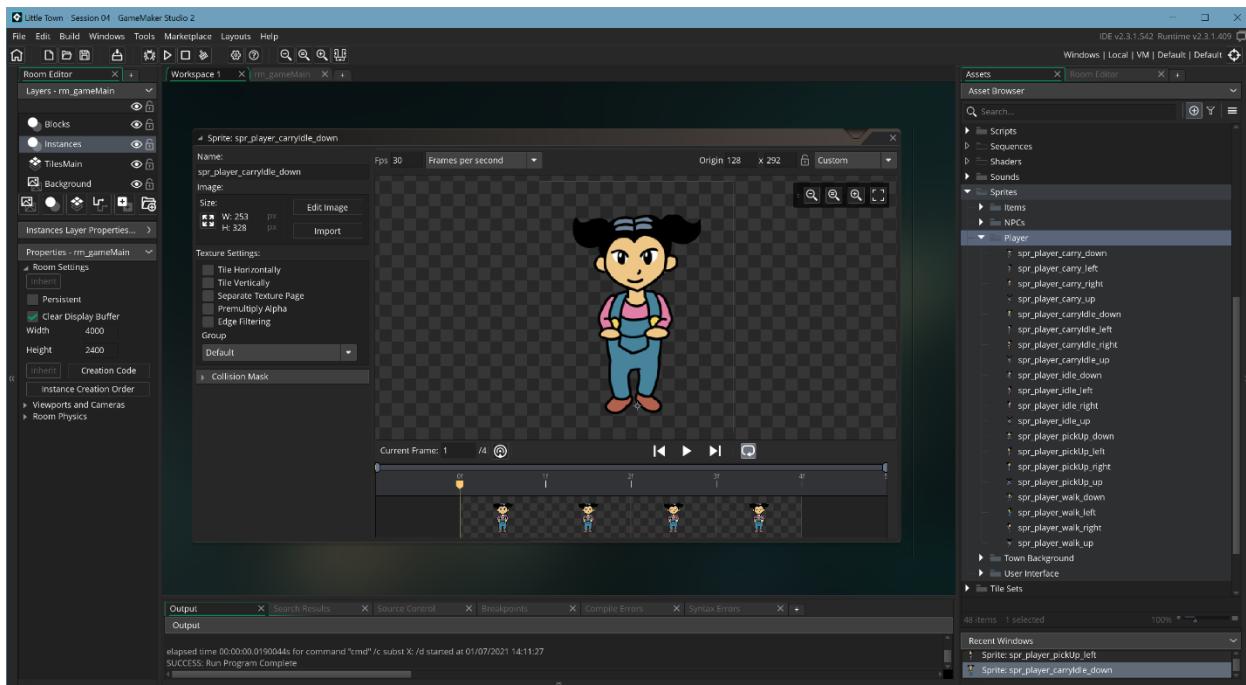
- spr_player_carry_left_strip04
- spr_player_carry_right_strip04
- spr_player_carry_up_strip04
- spr_player_carryIdle_down_strip05
- spr_player_carryIdle_left_strip05
- spr_player_carryIdle_right_strip05
- spr_player_carryIdle_up_strip05
- spr_player_pickUp_down_strip06
- spr_player_pickUp_left_strip06
- spr_player_pickUp_right_strip06
- spr_player_pickUp_up_strip06
- spr_player_putDown_down_strip06
- spr_player_putDown_left_strip06
- spr_player_putDown_right_strip06
- spr_player_putDown_up_strip06



Adding all the new player Sprites to the Asset Browser. We'll have to open each of them to adjust their Origin and FPS.

As you've done before in [Changing our player Sprites](#), open each of these new Sprites in the Sprite Editor and do the following:

- Make sure the Sprite Strip has been converted to frames of animation. If not, refer to the steps in [Converting a Sprite Strip manually](#)
- Set its Origin to between the player's feet (you can use existing Sprites such as spr_player_idle_down for reference)
- Set the FPS to 10
- Remove the _stripXX suffix from the Sprite name



Editing all the new player Sprites to be consistent.

When you're done with this, open obj_player again.

At the bottom of obj_player's Create Event add this new code block:

```
// Player Sprite array [myState][dir]
// Idle
playerSpr[playerState.idle][0] = spr_player_idle_right;
playerSpr[playerState.idle][1] = spr_player_idle_up;
playerSpr[playerState.idle][2] = spr_player_idle_left;
```

```

playerSpr[playerState.idle][3] = spr_player_idle_down;

// Walking
playerSpr[playerState.walking][0] = spr_player_walk_right;
playerSpr[playerState.walking][1] = spr_player_walk_up;
playerSpr[playerState.walking][2] = spr_player_walk_left;
playerSpr[playerState.walking][3] = spr_player_walk_down;

```

We've created an array called `playerSpr` to choose which Sprite our player should be using, and it's using two dimensions:

`[myState]` and `[dir]`

If you recall, we *just* changed that `// If Idle` code block in `obj_player`'s Step Event to say this:

```

// If Idle
if (vx == 0 && vy == 0) {
    myState = playerState.idle;
}

```

Furthermore, in `obj_player`'s Step Event we also set the value `dir` depending on our player movement (in the `// Change direction based on movement` code block), like so:

```

if (vx > 0) {
    dir = 0; // set direction as right-facing
}

```

So, suppose our player is just standing there, idle and it's facing right. According to the code we've written in `obj_player`'s Step Event, that would mean that:

- `playerState` is `idle`
- `dir` is 0

And if we look at the array we just wrote, we can see that we've built ourselves an invisible spreadsheet that says:

Array	myState	dir	
<code>playerSpr</code>	<code>[playerState.walking]</code>	<code>[0]</code>	<code>= sprPlayer_idle_right;</code>

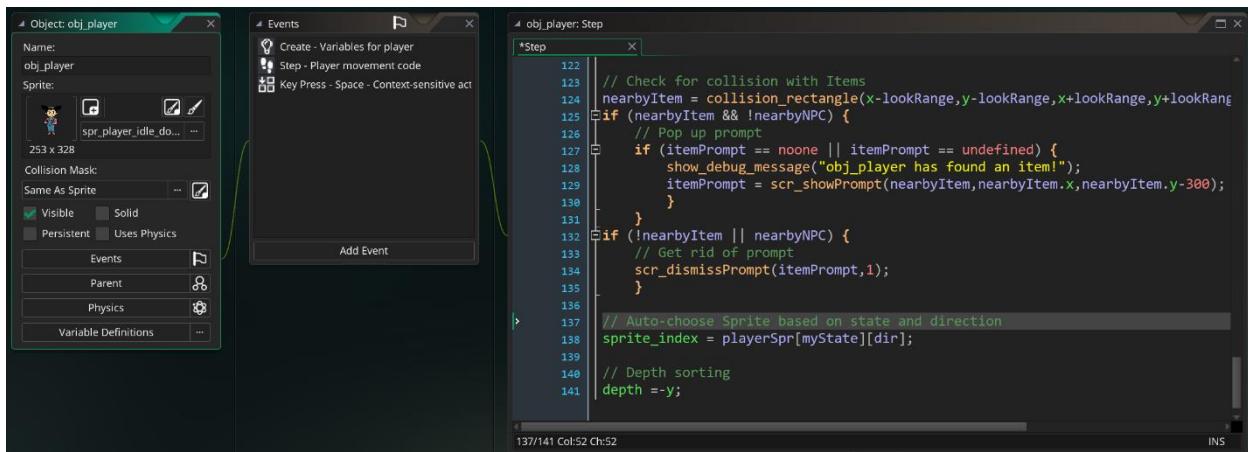
So with our enum, we've been able to write an array that we can read (more or less) visually, and specify which of those Sprite assets in the Sprites > Player Asset Browser group our game should use for the player Object.

7.7 Using an array to change Sprites

Let's put this array into action to really see how it works. Open `obj_player`'s Step Event and above the `// Depth sorting` code block, add this new code:

```
// Auto-choose Sprite based on state and direction  
sprite_index = playerSpr[myState][dir];
```

Here, we're changing the player's `sprite_index`, but instead of choosing a Sprite asset directly, we're referring to both the `myState` and `dir` variables to generate numbers and asking GameMaker Studio 2 to find whatever value is stored in the `playerSpr` array position based on those numbers. And since this is being done in the Step Event, the player Object is checking constantly.



Changing `obj_player`'s `sprite_index` by using an array.

Run your game again and move the player around the Room as before. You should see that your player Object correctly changes Sprites for both its idle and walking states, considering the direction it's facing, just like it did before!

With this system set up, we can do a whole lot more with our player Object without getting bogged down with too much code. As always, make sure to save your project!

7.8 Applying states to items

Now that we know how enums work and how to use them to create states, let's apply that to our items. We're going to want to our items to also have different states as well (*idle*, *taken*, *being put back*, etc.).

First, open `obj_control` and return to its Game Start Event. Underneath the `// Player states` code block we created before, add another enum, like so:

```
// Item states
enum itemState {
    idle,
    taken,
    used,
    puttingBack,
}
```

We're going to do the same thing with our items that we did with our player Object. So, Open `obj_par_item` and in the Object Editor, click Add Event and choose Create.

In this new Create Event, add the following variables:

```
// Set my state
myState = itemState.idle;
```

Open `obj_par_item`'s Step Event. So far, all we have is the `// Depth sorting` code block. Replace that with this code block instead:

```
// Depth, animation
switch myState {
    // If item is sitting on the ground
    case itemState.idle: {
        depth = -y;
    }; break;
}
```

So far, we aren't doing anything different at all, but we've built a switch function to take into account the new `itemState` enum we just created. We'll build on this more later.

7.9 Picking up an item

Right now, our player Object will notice when it's colliding with an item that's lying in the ground and we'll get a "?" prompt when this happens. So, let's put that to good use and allow our player Object to actually take an item!

Here's what we need to do to make this happen:

- Check to see if we're beside an item
- Check to make sure we're not already carrying an item

- If both conditions are okay, then pick up the item
- Change the player state so it can animate correctly
- Change the item's state so it can be carried by the player

It may seem like a lot, but thanks to the `playerState` enum, the `itemState` enum and the `playerSpr` array we started, it will be more straightforward than you think!

Open `obj_player` and its `Key Press - Space` Event. Previously, we used this Event to allow us to create a textbox when we were next to an NPC.

But we can use this Event for multiple actions; we just need to specify the conditions for each one. So, let's replace the *entire* contents of the `Key Press - Space` Event, like so:

```
var _text;

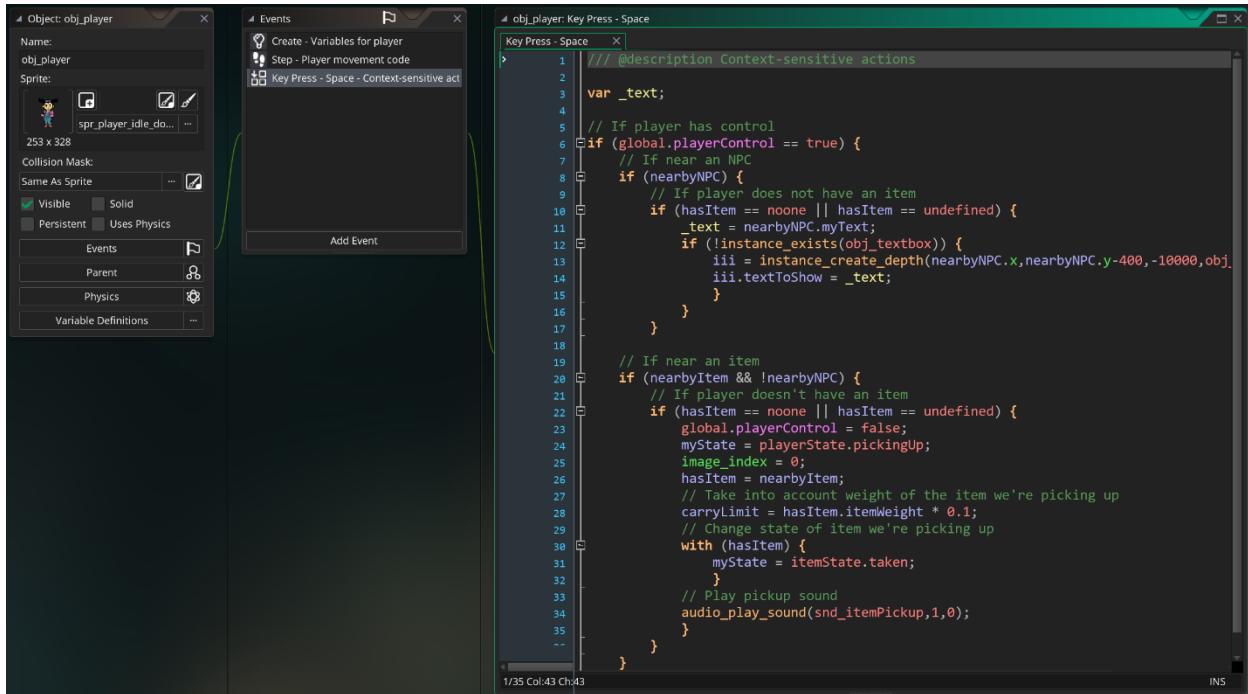
// If player has control
if (global.playerControl == true) {
    // If near an NPC
    if (nearbyNPC) {
        // If player does not have an item
        if (hasItem == noone || hasItem == undefined) {
            _text = nearbyNPC.myText;
            if (!instance_exists(obj_textbox)) {
                iii = instance_create_depth(nearbyNPC.x,nearbyNPC.y-400,-
10000,obj_textbox);
                iii.textToShow = _text;
            }
        }
    }

    // If near an item
    if (nearbyItem && !nearbyNPC) {
        // If player doesn't have an item
        if (hasItem == noone || hasItem == undefined) {
            global.playerControl = false;
            myState = playerState.pickingUp;
            image_index = 0;
            hasItem = nearbyItem;
            // Take into account weight of the item we're picking up
            carryLimit = hasItem.itemWeight * 0.1;
            // Change state of item we're picking up
            with (hasItem) {
                myState = itemState.taken;
            }
        }
        // Play pickup sound
        audio_play_sound(snd_itemPickup,1,0);
    }
}
```

```

        }
    }
}
```

Why are we replacing everything we've written so far? You'll notice that we haven't changed any functionality, just broken it up a bit. This is because as we progress through this tutorial, we're going to be adding more actions in this Event, and if we don't plan ahead a little, we could paint ourselves into a corner.



Replacing the contents of `obj_player`'s Key Press – Space Event with updated code blocks.

One example is that in the `// If near an NPC` code block, we've added a check that looks like this (**in bold**):

```

// If player does not have an item
if (hasItem == noone || hasItem == undefined) {
    _text = nearbyNPC.myText;
    if (!instance_exists(obj_textbox)) {
        iii = instance_create_depth(nearbyNPC.x, nearbyNPC.y-400, -10000, obj_textbox);
        iii.textToShow = _text;
    }
}
```

Later, we're going to want to check if the player *is* carrying an item, so we've added this here to make sure we also check if the player isn't carrying one.

As you can see, we are using that `nearbyItem` variable to set `hasItem`, which will be how we track what item the player is holding. And like we've done before, we're using a `with` statement to set that item's state to `taken`. That doesn't do anything now, but it will later.

You'll also notice that we're setting that `carryLimit` variable that we initialized in `obj_player`'s Create Event. We'll use this in a little bit to add some fun variation to how our player moves.

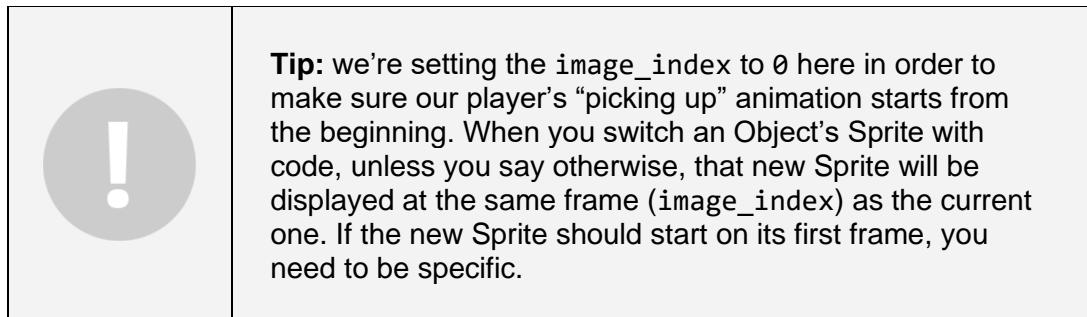
As well, we're using that simple `audio_play_sound()` function to play a "pickup" sound here. This code is exactly the same as we've used before for `obj_textbox` and `obj_prompt`.

Finally, note how the `// If near an item` code block now uses an if statement that says:

```
| if (nearbyItem && !nearbyNPC) {
```

Or, "if I'm near an item and *not* near an NPC."

This is to avoid a conflict between being near a character and being near an item. If both cases were true, then what? By updating this if statement, we're choosing to make being near a character more important.



Since we're using states for our player and we're using that expression in `obj_player`'s Step even to figure out which Sprite it should have, we need to be proactive and make sure all the necessary Sprites for picking up items, carrying them and putting them down are all in place.

Go to `obj_player`'s Create Event again and let's complete that array of Sprite assets.

Add the following new code to the bottom of the `// Player Sprite array [myState][dir]` code block:

```
| // Picking up
```

```

playerSpr[playerState.pickingUp][0] = spr_player_pickUp_right;
playerSpr[playerState.pickingUp][1] = spr_player_pickUp_up;
playerSpr[playerState.pickingUp][2] = spr_player_pickUp_left;
playerSpr[playerState.pickingUp][3] = spr_player_pickUp_down;

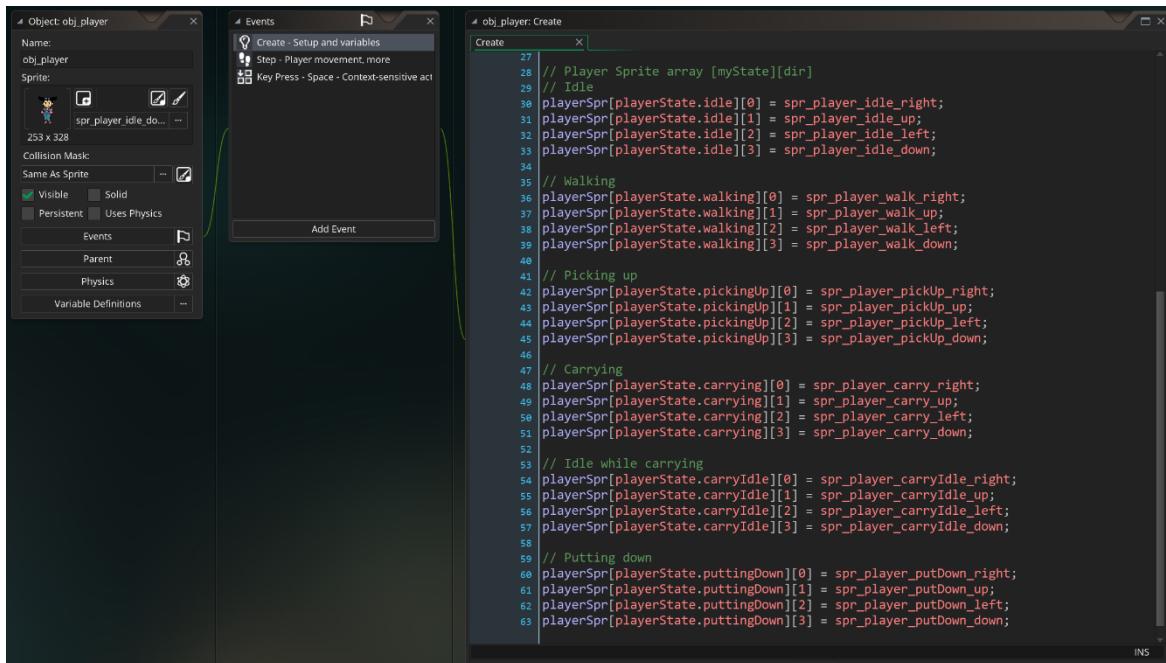
// Carrying
playerSpr[playerState.carrying][0] = spr_player_carry_right;
playerSpr[playerState.carrying][1] = spr_player_carry_up;
playerSpr[playerState.carrying][2] = spr_player_carry_left;
playerSpr[playerState.carrying][3] = spr_player_carry_down;

// Idle while carrying
playerSpr[playerState.carryIdle][0] = spr_player_carryIdle_right;
playerSpr[playerState.carryIdle][1] = spr_player_carryIdle_up;
playerSpr[playerState.carryIdle][2] = spr_player_carryIdle_left;
playerSpr[playerState.carryIdle][3] = spr_player_carryIdle_down;

// Putting down
playerSpr[playerState.puttingDown][0] = spr_player_putDown_right;
playerSpr[playerState.puttingDown][1] = spr_player_putDown_up;
playerSpr[playerState.puttingDown][2] = spr_player_putDown_left;
playerSpr[playerState.puttingDown][3] = spr_player_putDown_down;

```

Adding these lines will ensure that our `sprite_index` expression in `obj_player`'s Step Event will work as we continue to add functionality to our player.



Adding more Sprites to our player's array.

Now, open obj_player's Step Event again so we can do two things:

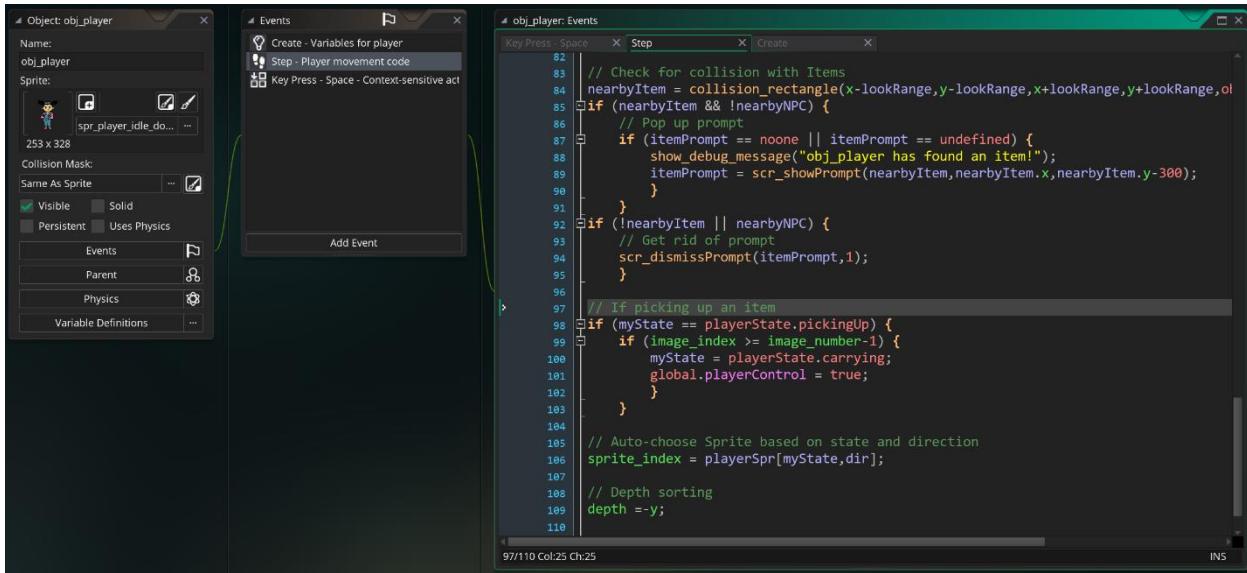
First, update the // Check for collision with Items code block, like so:

```
// Check for collision with Items
nearbyItem = collision_rectangle(x-lookRange,y-
lookRange,x+lookRange,y+lookRange,obj_par_item,false,true);
if (nearbyItem && !nearbyNPC) {
    // Pop up prompt
    if (itemPrompt == noone || itemPrompt == undefined) {
        show_debug_message("obj_player has found an item!");
        itemPrompt = scr_showPrompt(nearbyItem,nearbyItem.x,nearbyItem.y-300);
    }
}
if (!nearbyItem || nearbyNPC) {
    // Get rid of prompt
    scr_dismissPrompt(itemPrompt,1);
}
```

Just like we did in the Key Press – Space Event, we're making sure nearbyItem and nearbyNPC don't interfere with each other.

Second, add this new code block after the // Check for collision with Items code block:

```
// If picking up an item
if (myState == playerState.pickingUp) {
    if (image_index >= image_number-1) {
        myState = playerState.carrying;
        global.playerControl = true;
    }
}
```



Adding an if statement for when the player Object is picking up an item.

This new `// If picking up an item` code block is paying attention to our player Object's animation: since our player automatically changes which Sprite it displays as its state changes, it will be playing a "picking up" animation as soon as we press the Space key to pick up an item. Once this animation ends, we change the player state again to carrying and return control to the player.

However, there's one more thing we need to do here to make this work. While still in `obj_player`'s Step Event, find the `// If Idle` code block that currently looks like this:

```

// If Idle
if (vx == 0 && vy == 0) {
    myState = playerState.idle;
}

```

We need to change this since now, we're adding more functionality to our player. The player can be standing still, sure, but they can either be `idle` (without an item), or `carryIdle` (with an item). And, to complicate things, they can also be standing still while picking up or putting down an item, so we need to be very specific.

Update the `// If Idle` code block like so:

```

// If Idle
if (vx == 0 && vy == 0) {
    // If I'm not picking up or putting down an item
    if (myState != playerState.pickingUp && myState != playerState.puttingDown) {
        // If we don't have an item
    }
}

```

```
    if (hasItem == noone) {
        myState = playerState.idle;
    }
    // If we're carrying an item
    else {
        myState = playerState.carryIdle;
    }
}
```

Similarly, we need to be specific when the player is walking around as well. Because now, the player can walk while carrying something *or* while not carrying something, and those states are different, with their own Sprites.

So, in `obj_player`'s Step Event, find this line of code within the `// If moving` block:

```
// Set state
myState = playerState.walking;
```

We need to replace this line with this new code block:

```
// Set state
// If we don't have an item
if (hasItem == noone) {
    myState = playerState.walking;
}
// If we're carrying an item
else {
    myState = playerState.carrying;
}
```

All right — with all that new code out of the way, we're now ready to work on our item again!

7.10 Returning results from functions

Our player can now, technically, pick up an item and carry it around! But how does that actually work?

Well, as it turns out, all the player does is look after itself; it says, “I’m picking up an item now,” and then, “Okay, I’m carrying an item.” But it’s going to be the item’s job to leap into the player’s arms and follow it around.

In order to do this, the item (specifically, `obj_par_item`) needs to know where the player is at all times, so it can look like it's being carried by the player. However, we have a wrinkle here: our player can move all around the screen and its Sprite changes depending on its direction.

So, *where* the item needs to be (and whether it needs to be in front of, or behind, the player Object) depends on the player's direction too. So how do we track this?

Let's start by making a new Script Asset. Right-click in the Asset Browser and choose Create > Script. Name this new Script Asset `scr_itemControl`.

In the new Script Asset, you'll see the familiar default function code. Let's rename that function to `scr_itemPosition` and fill it in, like so:

```
function scr_itemPosition(){
    var _x, _y, _depth;
    if (instance_exists(obj_player)) {
        switch obj_player.dir {
            case 0: { // right
                _x = obj_player.x+65;
                _y = obj_player.y-112;
                _depth = obj_player.depth-2;
            }; break;
            case 1: { // up
                _x = obj_player.x;
                _y = obj_player.y-95;
                _depth = obj_player.depth+2;
            }; break;
            case 2: { // left
                _x = obj_player.x-65;
                _y = obj_player.y-112;
                _depth = obj_player.depth-2;
            }; break;
            case 3: { // down
                _x = obj_player.x;
                _y = obj_player.y-95;
                _depth = obj_player.depth-2;
            }; break;
        }
        return [_x,_y,_depth];
    }
}
```



Tip: We create variables that start with underscores (like `_x`) to indicate that they are temporary and local to a function. It's just a way to avoid mixing things up.

The screenshot shows the GameMaker Studio script editor with the file `scr_itemControl.gml` open. The code defines a function `scr_itemPosition()` that returns a tuple of `[_x, _y, _depth]`. The function first checks if the `obj_player` object exists. If it does, it uses a switch statement to determine the player's direction (right, up, left, down) and calculates the corresponding `_x`, `_y`, and `_depth` values. The `return` statement at the end of the function body specifies the return type as `[_x, _y, _depth]`.

```
// @description Keep an item with the player
function scr_itemPosition(){
    var _x, _y, _depth;
    if (instance_exists(obj_player)) {
        switch obj_player.dir {
            case 0: { // right
                _x = obj_player.x+65;
                _y = obj_player.y-112;
                _depth = obj_player.depth-2;
            }; break;
            case 1: { // up
                _x = obj_player.x;
                _y = obj_player.y-95;
                _depth = obj_player.depth+2;
            }; break;
            case 2: { // left
                _x = obj_player.x-65;
                _y = obj_player.y-112;
                _depth = obj_player.depth-2;
            }; break;
            case 3: { // down
                _x = obj_player.x;
                _y = obj_player.y-95;
                _depth = obj_player.depth-2;
            }; break;
        }
        return [_x, _y, _depth];
    }
}
```

Creating the new `scr_itemPosition` function within the `scr_itemControl` Script Asset.

This function will now check the player Object's position and set an `_x`, `_y` and `_depth` variable based on that. But the real fun here is at the end, where it says `return`.

To see how this works, open `obj_par_item` and its Step Event.

Currently in the `// Depth, animation` code block we have a switch function with only one case (`itemState.idle`). Let's expand on this function and make it look like so:

```

// Depth, animation
switch myState {
    // If item is sitting on the ground
    case itemState.idle: {
        depth = -y;
        }; break;
    // If item has been taken
    case itemState.taken: {
        // Keep track of player position
        var _result = scr_itemPosition();
        x = _result[0];
        y = _result[1];
        depth = _result[2];
        }; break;
}

```

Pay attention to what's happening in our new case (`itemState.taken`). As we have done before, we're storing the result of a function (in this case, the `scr_itemPosition()` function we just wrote) in a variable (`_result`).

But with `scr_itemPosition`, we have that *return* line and it has three values (`_x`, `_y` and `_depth`). So, what, exactly, is it returning and how do three values get stored in one variable (`_result`)?

Well, as it turns out, when you return multiple values like this, you get something with which you are already familiar: an array!

So, that `return [_x,_y,_depth]` line at the end of our function becomes:

```

_result[0] = _x;
_result[1] = _y;
_result[2] = _depth;

```

Review the rest of the code and you'll see we are now adjusting our item's x, y and depth values to what our new function gives us. Handy!

The screenshot shows three panels of the GameMaker Studio 2 interface:

- Object Properties Panel:** Shows 'obj_par_item' with a 'No Sprite' selected. Collision Mask is set to 'Same As Sprite'. Events, Parent, Physics, and Variable Definitions tabs are visible.
- Events Panel:** Shows a 'Create - Setup and variables' event with a single step named 'Step - Depth and more'.
- Script Editor:** Shows the 'Step' event script for 'obj_par_item'. The script uses a switch statement to handle item states (idle or taken) and retrieves depth information from a function.

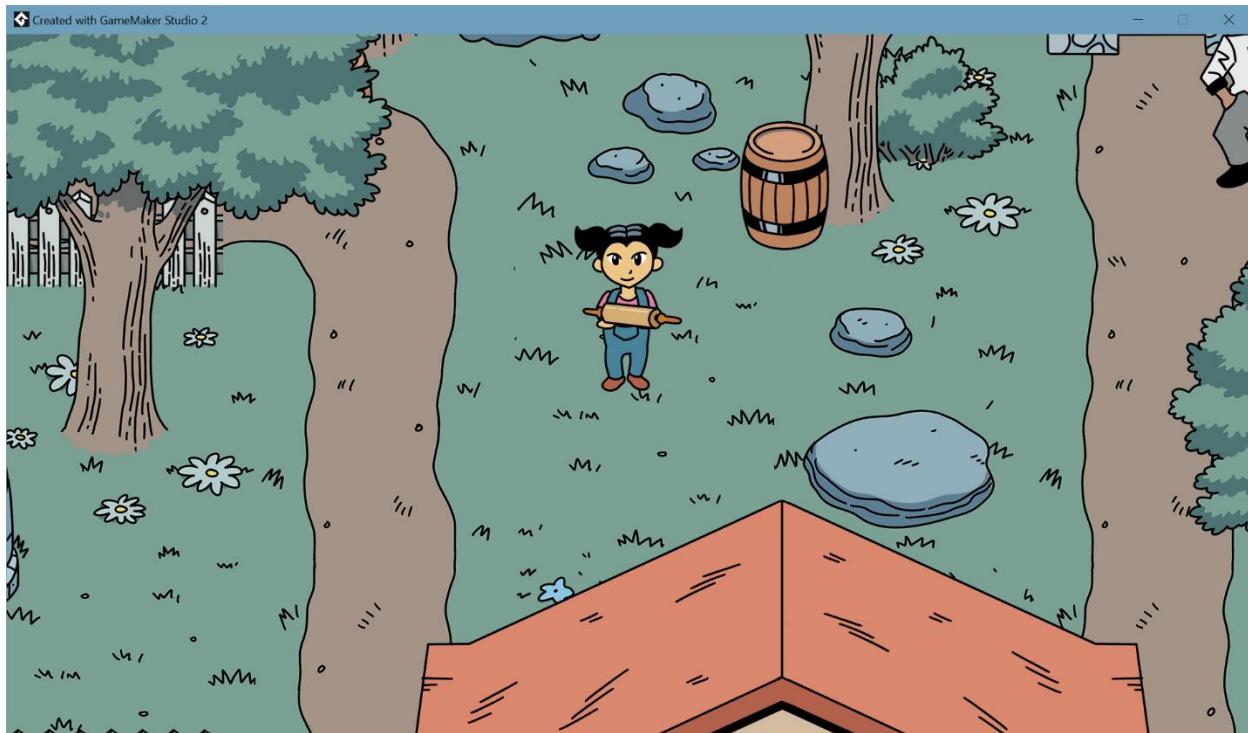
```

// @description Depth and more
// Depth, animation
switch myState {
    case itemState.idle: {
        depth = y;
    }; break;
    case itemState.taken: {
        // If item has been taken
        // Keep track of player position
        var _result = scr_itemPosition();
        x = _result[0];
        y = _result[1];
        depth = _result[2];
    }; break;
}

```

Getting three values from a single function in `obj_par_item`'s Step event.

Run your game again and walk over to the item you previously placed in the town. Press Space and you should see the player animate to pick it up, whereupon the item will magically leap into the player Object's arms. You can then continue to walk around, now with arms outstretched and the item will track with you!



Our player Object can now pick up and carry items around.

	<p>Tip: If the item doesn't correctly appear in front of or behind the player while it's been carrying, the game might be getting confused (we all have our days). To fix this, you can change <code>obj_par_item</code>'s Step Event to an End Step Event.</p> <p>Open <code>obj_par_item</code> and right-click the Step Event in the Events window. Choose Change Event and select Step > End Step.</p> <p>The End Step Event runs <i>after</i> the Step Event, but still happens every single frame of the game.</p>
---	--

7.11 Putting the item back down

Luckily, we've done most of the hard work already, so having the player put an item back down is straightforward.

First, Open `obj_par_item` if you don't have it opened already and open its Create Event. Add a new line after the `// Set my state` code block:

```
| putDownY = 0;
```

Next, Open `obj_player` again and its Key Press - Space Event. Enter this new code block after the `// If near an item item` block:

```
// If not near an NPC or another item
if (!nearbyItem && !nearbyNPC) {
    // Put down an item
    if (hasItem != noone) {
        myState = playerState.puttingDown;
        image_index = 0;
        global.playerControl = false;
        // Change state of item we were carrying
        with (hasItem) {
            putDownY = obj_player.y+5;
            myState = itemState.puttingBack;
        }
        // Play put-down sound
        audio_play_sound(snd_itemPutDown,1,0);
        // Reset item
```

```
    hasItem = noone;
}
}
```

What we're doing here is changing the player's state, changing the item's state and setting that new variable, `putDownY`, to be relative to the player's current position. Finally, we reset `hasItem` so our player knows it's no longer carrying anything.

And, as we did with picking up an item, we're playing a sound at this point (using the same `audio_play_sound()` function we've used several times now).

Next we need to deal with what happens once the player Object is putting down an item.

Open `obj_player`'s Step Event and add this new block after the `// If picking up an item` code block:

```
// If putting down an item
if (myState == playerState.puttingDown) {
    // Reset weight
    carryLimit = 0;
    // Reset my state once animation finishes
    if (image_index >= image_number-1) {
        myState = playerState.idle;
        global.playerControl = true;
    }
}
```

Here we're resetting that `carryLimit` variable (which we've set but not done much with yet) and we're using a simple check to see if we're on the last frame of our "putting down" animation. If so, we then reset the player state and return control.

```
        with (hasItem) {
            myState = itemState.taken;
        }
    }

    // If not near an NPC or another item
    if (!nearbyItem && !nearbyNPC) {
        // Put down an item
        if (hasItem != noone) {
            myState = playerState.puttingDown;
            image_index = 0;
            global.playerControl = false;
            // Change state of item we were carrying
            with (hasItem) {
                putDownY = obj_player.y+5;
                myState = itemState.puttingBack;
            }
            // Play put-down sound
            audio.play_sound(snd_itemPutDown,1,0);
            // Reset item
            hasItem = noone;
        }
    }
}
```

```
obj_player:Step
Step 112
113 // If picking up an item
114 if (myState == playerState.pickingUp) {
115     if (image_index == image_number-1) {
116         myState = playerState.carrying;
117         global.playerControl = true;
118     }
119 }
120
121 // If putting down an item
122 if (myState == playerState.puttingDown) {
123     // Reset weight
124     carryLimit = 0;
125     // Reset my state once animation finishes
126     if (image_index >= image_number-1) {
127         myState = playerState.idle;
128         global.playerControl = true;
129     }
130 }
131
132 // Auto-choose Sprite based on state and direction
133 sprite_index = playerSpr[myState,dir];
134
135 // Depth sorting
136 depth = -y;
137
```

In obj_player, new code blocks in the Key Press – Space and Step Events allow it to put down an item it is carrying.

Finally, we need to also reset our item. Open `obj_par_item` again and go to its Step Event. (If you changed this to an End Step Event, use that Event instead.)

Update the myState switch function in the // Depth, animation code block like so:

```
// Depth, animation
switch myState {
    // If item is sitting on the ground
    case itemState.idle: {
        depth = -y;
    }; break;
    // If item has been taken
    case itemState.taken: {
        // Keep track of player position
        var _result = scr_itemPosition();
        x = _result[0];
        y = _result[1];
        depth = _result[2];
    }; break;
    // If item is being put back
    case itemState.puttingBack: {
        y = putDownY;
        myState = itemState.idle;
    }; break;
}
```

You can see that we're using that `putDownY` variable, which we set in `obj_player`'s Key Press - Space Event, to return the item to the ground. Then, we simply reset the item's state.

Run the game again and give it a try: you should be able to pick up the item, walk around with it and then press Space again to drop it back on the ground!



Now in the game the player Object can pick up an item, carry it around, and put it down.



Tip: you'll notice that when we pick up an item and put it down, the item just plops into place and doesn't really follow the player animation. We address this in the Bonus Session at the end of this course.

7.12 Limiting when an item prompt can show

Thanks to our new player states, we can pick up and put down items. This does, however, introduce a minor conflict that we have to address.

Currently, our game can get confused and, at very specific times, can pop up `obj_prompt` while the player is putting down an item. To fix this, we're going to make a minor change to the `scr_showPrompt()` function.

Open `scr_promptControl` from the Asset Browser and update the `scr_showPrompt()` function like so:

```
function scr_showPrompt(_object,_x,_y) {  
    if (instance_exists(_object)) {  
        if (!instance_exists(obj_textbox) && !instance_exists(obj_prompt)) {  
            if (obj_player.myState != playerState.puttingDown) {  
                iii = instance_create_depth(_x,_y,-10000,obj_prompt);  
                return iii;  
            }  
        }  
    }  
}
```

You'll notice we're checking to make sure the player isn't putting down an item, and then creating the prompt Object instance as before.

If you run the game again and test, you'll see that the prompt no longer appears so quickly when the player puts down an item.

7.13 Affecting the player's speed

Now our player can pick up items and carry them around. But if you recall, we gave our `obj_par_item` a Variable Definition called `itemWeight`. Let's use that to add some fun variation to our gameplay.

Open `obj_player` and its Step Event.

Find the `// Calculate movement` code block, which currently looks like this:

```
// Calculate movement  
vx = ((moveRight - moveLeft) * walkSpeed);  
vy = ((moveDown - moveUp) * walkSpeed);
```

In order to have the weight of the items affect the player's movement, update the // Calculate movement code block like so:

```
// Calculate movement  
vx = ((moveRight - moveLeft) * (walkSpeed) * (1-carryLimit));  
vy = ((moveDown - moveUp) * (walkSpeed) * (1-carryLimit));
```

Here's an example of how a heavy item (say, with an `itemWeight` of 8) would affect the player's speed, if they were moving to the right.

First, in `obj_player`'s Key Press – Space Event, we set `carryLimit` based on the `itemWeight` of whatever item we're picking up:

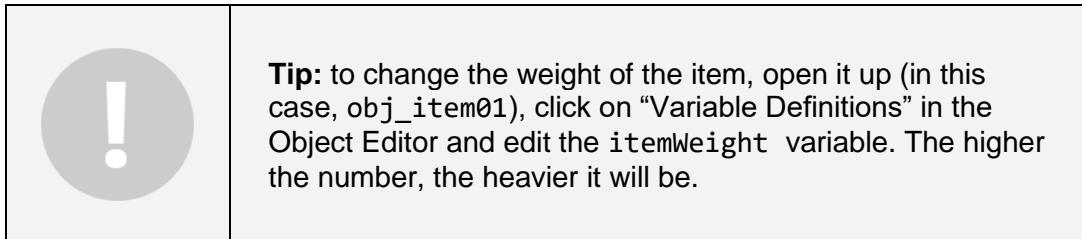
carryLimit =	hasItem.itemWeight * 0.1;
	8 x 0.1 = 0.8

Then in `obj_player`'s Step Event, we're using that `carryLimit` to affect the player's movement speed:

vx =	((moveRight - moveLeft)	* (walkSpeed)	* (1-carryLimit));
	1 - 0 = 1 (moving right)	x 16	x 0.2

In this instance, `vx` would be `3.2`, which is quite a bit slower than the usual `walkSpeed` (which we've set to `16`).

Run the game again and pick up the item in the Room. You'll notice now that the player moves a little bit more slowly than when they aren't carrying the Object.



7.14 Creating the other items

Before we continue, let's create the other five items for our game and place them in our town. Since we've done all the work already in `obj_par_item`, this will be very straightforward.

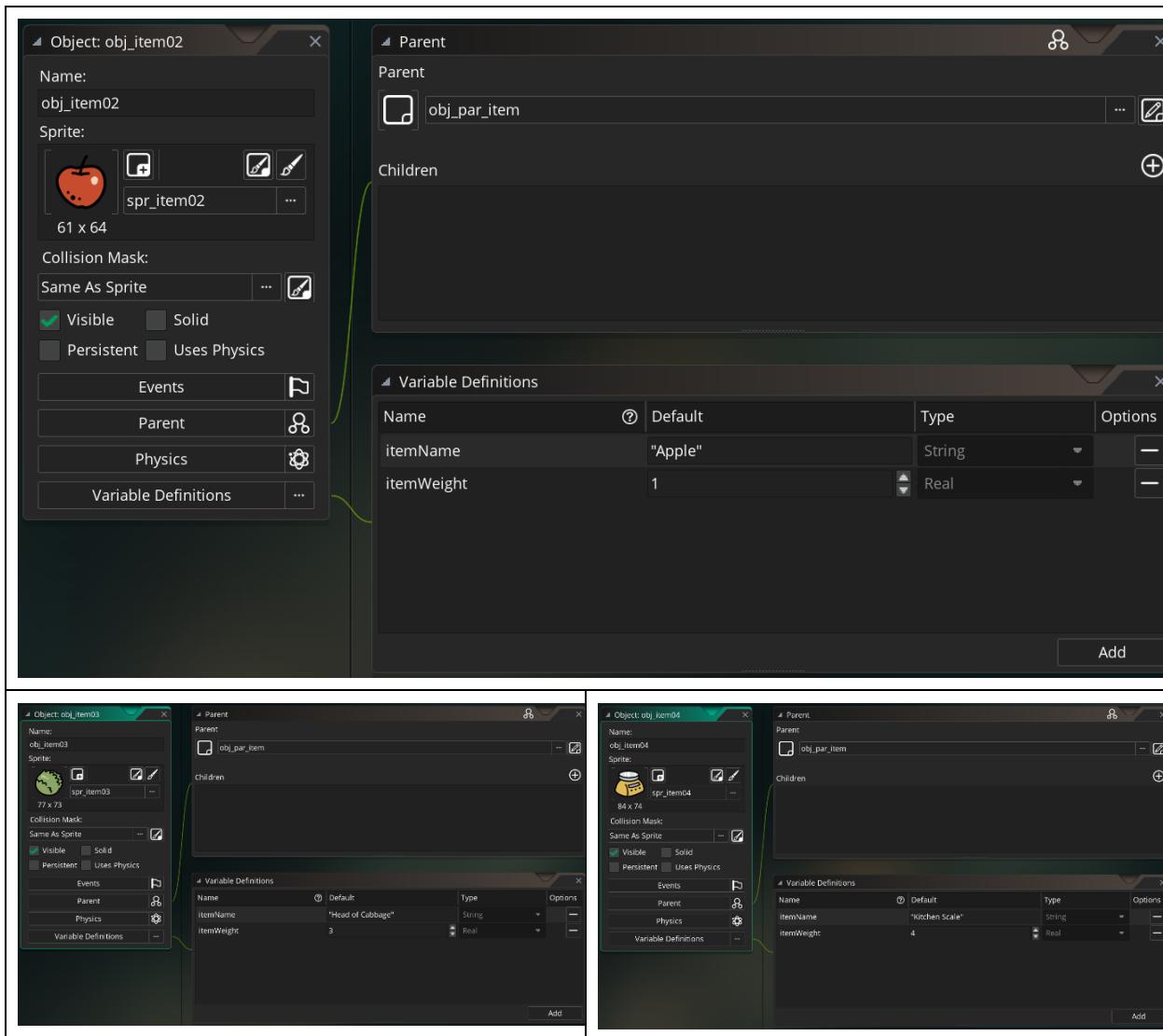
Now, we could manually create each of the five remaining item Objects from scratch, but why not save a little time?

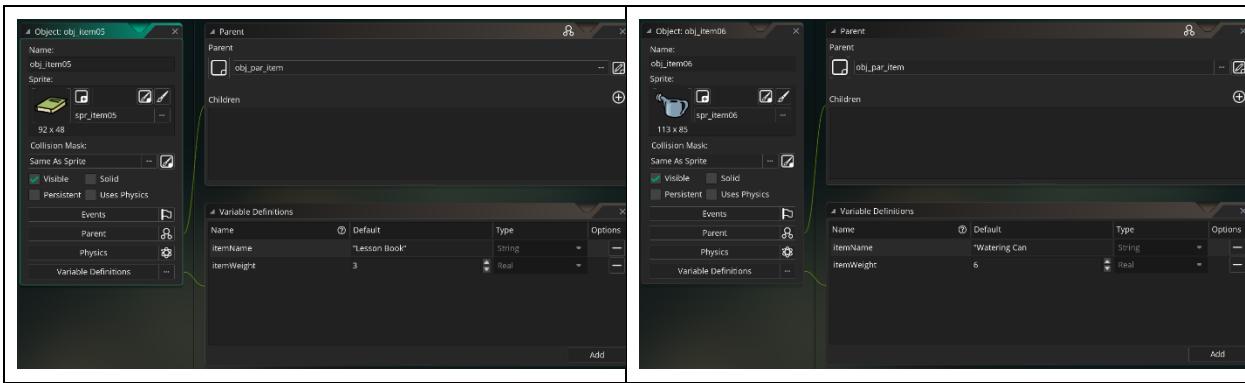
In the Asset Browser, find `obj_item01` and right-click on it. In the contextual menu that pops up, choose Duplicate (note that you can also just use CTRL-D on Windows, or CMD-D on Mac to do the same thing).

This will make a duplicate Object. Then double-click this to open the new Object in the Object Editor and do the following:

- rename it `obj_item02`
- click on the drop-down menu beside the attached Sprite and choose `spr_item02`
- click on Variable Definitions and edit both variables to give the new Object a different `itemName` and `itemWeight`

Repeat the above steps to create the third, fourth, fifth and sixth item Objects.





Once you've got all six Objects, open `rm_gameMain`. Make sure the Instances layer is selected in the Room Editor and drag in one instances of each item into the Room. Place the items wherever you like for now; for the final game, you'll want to hide the items a bit, so players have to look for them.



Placing all six items within our town.

Run your game again and walk around the town. Pick up each item and carrying it around and then drop it again. Notice how the different `itemWeight` values you set affect how slowly the player moves while carrying them. Feel free to adjust these if need be!

7.15 Adding the ability to run

Our player Object is getting quite active now, but let's take this one step further. Open obj_player and its Create Event and add the following lines to the // Variables code block:

```
runSpeed = 0;  
runMax = 12;  
running = false;
```

Now, open obj_player's Step Event and add the following between the // Check keys for movement and // Calculate movement code blocks:

```
// Run with Shift key  
running = keyboard_check(vk_shift);
```



Tip: using keyboard_check() in an Object's Step Event is the same thing as adding a Key Down Event for that key to the Object!

Since we are checking for the Shift key in Step and passing on the result (true or false) to the variable running, we now have a handy way to detect whether our character is running or not. But what do we do with that? Why, speed up the player, of course!

Below this new // Run with Shift key code block, let's add a new block to deal with the running speed of our player:

```
// Speed up if running  
if (running == true) {  
    // Ramp up  
    if (runSpeed < runMax) {  
        runSpeed += 2;  
    }  
}  
// Slow down if no longer running  
if (running == false) {  
    // Ramp down  
    if (runSpeed > 0) {  
        runSpeed -= 1;  
    }  
}
```

```
| }
```

Rather than change the speed immediately, we're ramping it up and down to provide a more realistic sense of motion.

Next, we must apply the speed to the player Object. Right there in obj_player's Step Event is the // Calculate movement block that we've written and updated before. Let's make one more change to this block, like so:

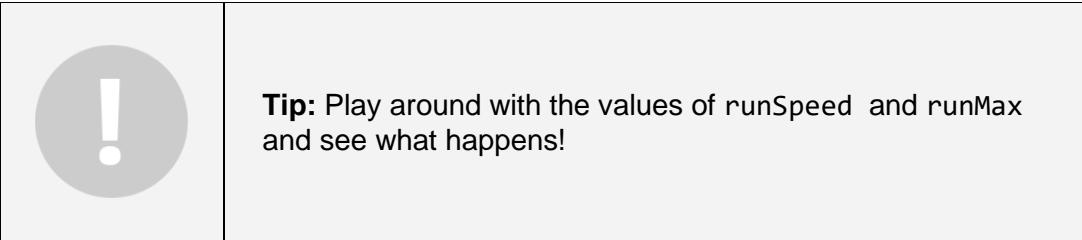
```
// Calculate movement
vx = ((moveRight - moveLeft) * (walkSpeed+runSpeed) * (1-carryLimit));
vy = ((moveDown - moveUp) * (walkSpeed+runSpeed) * (1-carryLimit));
```

```
16 // Run with Shift key
17 running = keyboard_check(vk_shift);
18
19 // Speed up if running
20 if (running == true) {
21     // Ramp up
22     if (runSpeed < runMax) {
23         runSpeed += 2;
24     }
25 }
26 // Slow down if no longer running
27 if (running == false) {
28     // Ramp down
29     if (runSpeed > 0) {
30         runSpeed -= 1;
31     }
32 }
33
34 // Calculate movement
35 vx = ((moveRight - moveLeft) * (walkSpeed+runSpeed) * (1-carryLimit));
36 vy = ((moveDown - moveUp) * (walkSpeed+runSpeed) * (1-carryLimit));
37
38
```

Adding the new running code, and updating the // Calculate movement code block, in obj_player's Step Event

Once again, run your game and move your player Object around. Try holding down the Shift key to watch the player take off and release it to see them slow down again. Note that you can also run while carrying an item, but each item's weight will continue to affect the player's speed.

When you're done having fun, close the game window and return to GameMaker Studio 2.

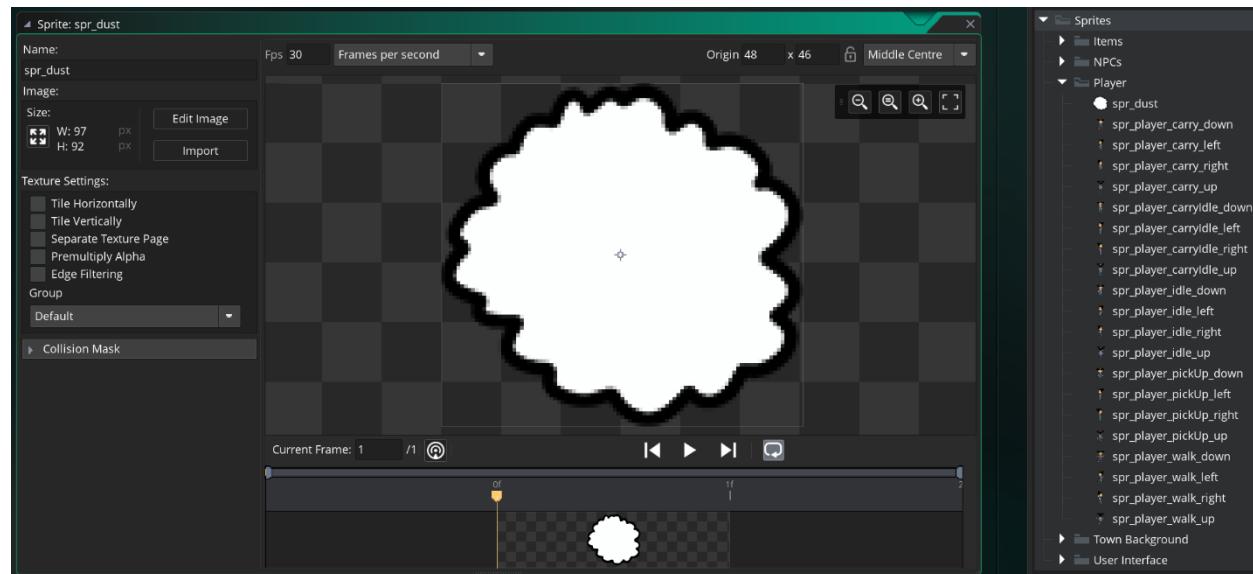


7.16 Adding a dust cloud effect

Let's add one final feature for this session to make our player's traversal through the town complete: a cartoon dust cloud effect that trails behind them when they run.

First, use Explorer in Windows or Finder on the Mac and navigate to the Asset folder that came with this course. In Characters and Items, you'll find the image file spr_dust. Drag that into GameMaker Studio 2's Asset Browser. (We recommend tucking it into the Sprites > Player Group.)

Open spr_dust and click on the Origin drop-down menu in the top-right of the Sprite Editor to set the origin to Middle Centre. (You don't need to worry about the collision mask for this asset.)



Adjust the Origin for our new spr_dust Sprite, and placing the asset into the Sprites > Player group.

Next, we need to create the dust Object; right-click in the Asset Browser and choose Create > Object. Name this new Object obj_dust.

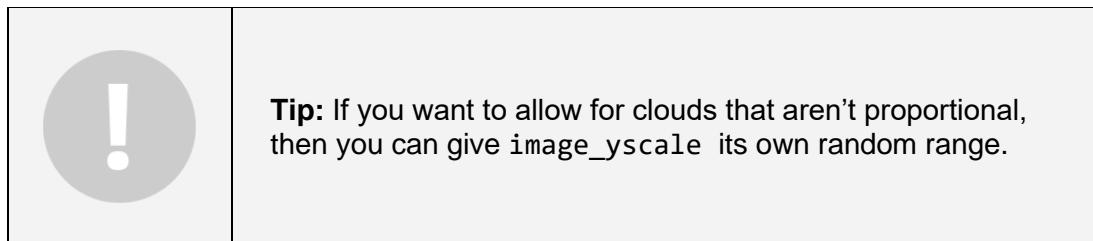
Open obj_dust in the Object Editor. Click on No Sprite and attach the new spr_dust asset we just imported.

Next, add a Create Event. In this Event, add the following code block:

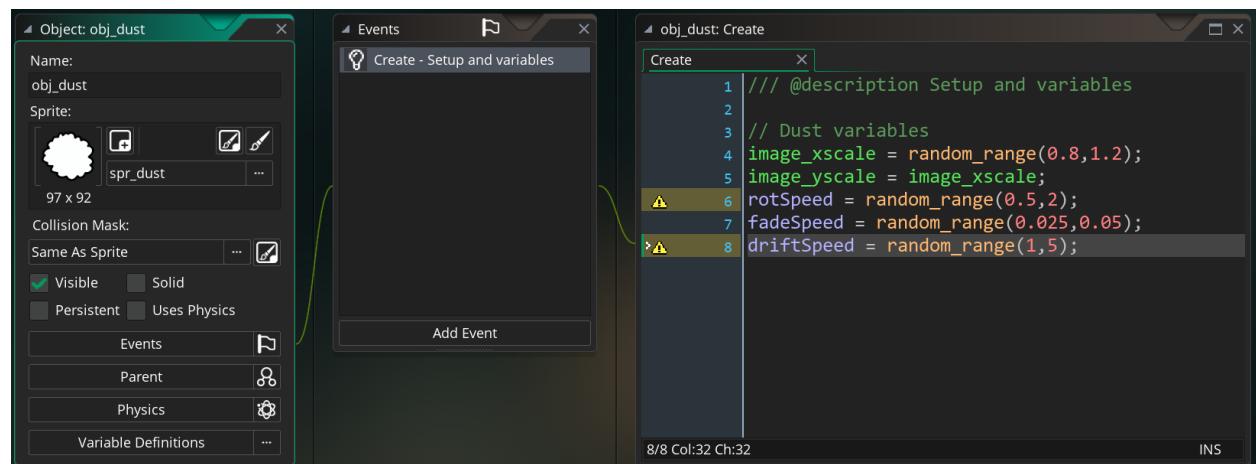
```
// Dust variables  
image_xscale = random_range(0.8,1.2);  
image_yscale = image_xscale;  
rotSpeed = random_range(0.5,2);  
fadeSpeed = random_range(0.025,0.05);  
driftSpeed = random_range(1,5);
```

Having every dust cloud appear at the exact same size and do the exact same thing isn't realistic. To really sell the effect, we want to give each cloud some variation in size, rotation speed and more.

To change the size, we're setting the Object's `image_xscale` first. This built-in variable is, as you might imagine, the scale of a Sprite on its x axis. To keep the width and height of the dust cloud proportional, we're setting `image_yscale` to be whatever `image_xscale` is.



The other variables here also have random ranges to allow for variety. You can change the numbers in these ranges later to see how that affects things, but for now, leave them like this.



Adding variables in the Create Event of our new dust Object.

Next, add a Step Event to our obj_dust in the Object Editor. Within this Step Event, write the following:

```
// Rotate cloud  
image_angle += rotSpeed;  
y -= driftSpeed;  
  
// Fade out  
if (image_alpha > 0) {  
    image_alpha -= fadeSpeed;  
}  
if (image_alpha <= 0) {  
    instance_destroy();  
}
```

The built-in variable `image_angle` lets us set the rotation of the Sprite, but since we're constantly adding a value (`rotSpeed`) to it, it will just keep rotating, creating a spinning effect.

Similarly, by constantly subtracting `driftSpeed` from `obj_dust`'s `y` value, it will continue to move upwards, creating a rising effect.

Finally, we're doing the same thing here with `image_alpha` that we've done before in sections like [Adding effects to our textbox](#). When our `image_alpha` reaches 0, we destroy the dust cloud. Since the `fadeSpeed` is randomized, each cloud will behave a little differently, giving us the variety we need.

	Tip: if you want the dust clouds to rotate clockwise instead of counterclockwise, change the "Rotate cloud" line to be <code>image_angle -= rotSpeed;</code>
--	---

7.17 Making the dust cloud effect work

With our dust cloud Object all set up, let's put it into action!

To do this, Open `obj_player` and its Create Event again. Add a new variable to the // Variables code block:

```
| startDust = 0;
```

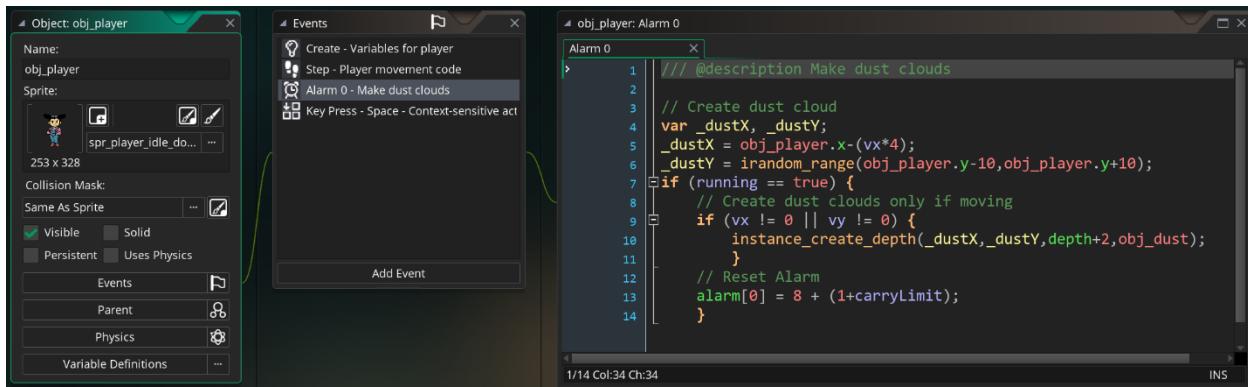
Next, in the Object Editor, click Add and choose Alarm > Alarm 0 to create a new Alarm Event. It will be in this Alarm that we spawn our dust clouds.

In Alarm 0, enter the following code block:

```
// Create dust cloud
var _dustX, _dustY;
_dustX = obj_player.x-(vx*4);
_dustY = irandom_range(obj_player.y-10,obj_player.y+10);
if (running == true) {
    // Create dust clouds only if moving
    if (vx != 0 || vy != 0) {
        instance_create_depth(_dustX,_dustY,depth+2,obj_dust);
    }
    // Reset Alarm
    alarm[0] = 8 + (1+carryLimit);
}
```

Here, we're creating temporary variables for the x and y position at which to create our dust cloud Object each time the Alarm goes off. We're making sure that the player is still running and that they're actually moving (by checking vx and vy). If all of this checks out, we create an instance of the dust cloud.

At the bottom, you'll see that we reset the Alarm from within the Alarm itself. We're setting it to 8 steps (you can change this to see what happens) and using that carryLimit variable to slow this down, just as carrying a heavy item will slow the player down.



Adding the code to create instances of the dust cloud as the player is running.

With this done, we only need to do two more things to make the dust cloud behave as expected:

1. start the process of creating the dust clouds when the player starts running; and
2. stop the process when the player is no longer running.

Let's take care of #1 first: open obj_player's Step Event and find the // Speed up if running and // Slow down if not running code blocks. Update these two blocks like so:

```
// Speed up if running
if (running == true) {
    // Ramp up
    if (runSpeed < runMax) {
        runSpeed += 2;
    }
    // Start creating dust
    if (startDust == 0) {
        alarm[0] = 2;
        startDust = 1;
    }
}
// Slow down if no longer running
if (running == false) {
    // Ramp down
    if (runSpeed > 0) {
        runSpeed -= 1;
    }
    startDust = 0;
}
```

You'll see that we're using that new variable, `startDust`, as a check to make sure we start triggering obj_player's Alarm 0 when the player starts running.

Run the game again and try both walking and running around. When you run, dust clouds should appear at the player's feet and gently spin and float away.



Running around our town with dust clouds kicking up at the player's feet.

7.18 End of Session 4

That's it for this session. In the next one, we're going to use another feature of GameMaker Studio 2, Sequences, to create fun animations that play when the player gives those items to our three characters.

Don't forget to save your project!

8 Session 5

In this session, we're going to dive deep into *Sequences* to create several animated cutscenes. Then we'll learn how to check for those items we created and display the correct Sequence, among many other things.

Load your project and let's get started!

8.1 Getting our Sequence assets ready

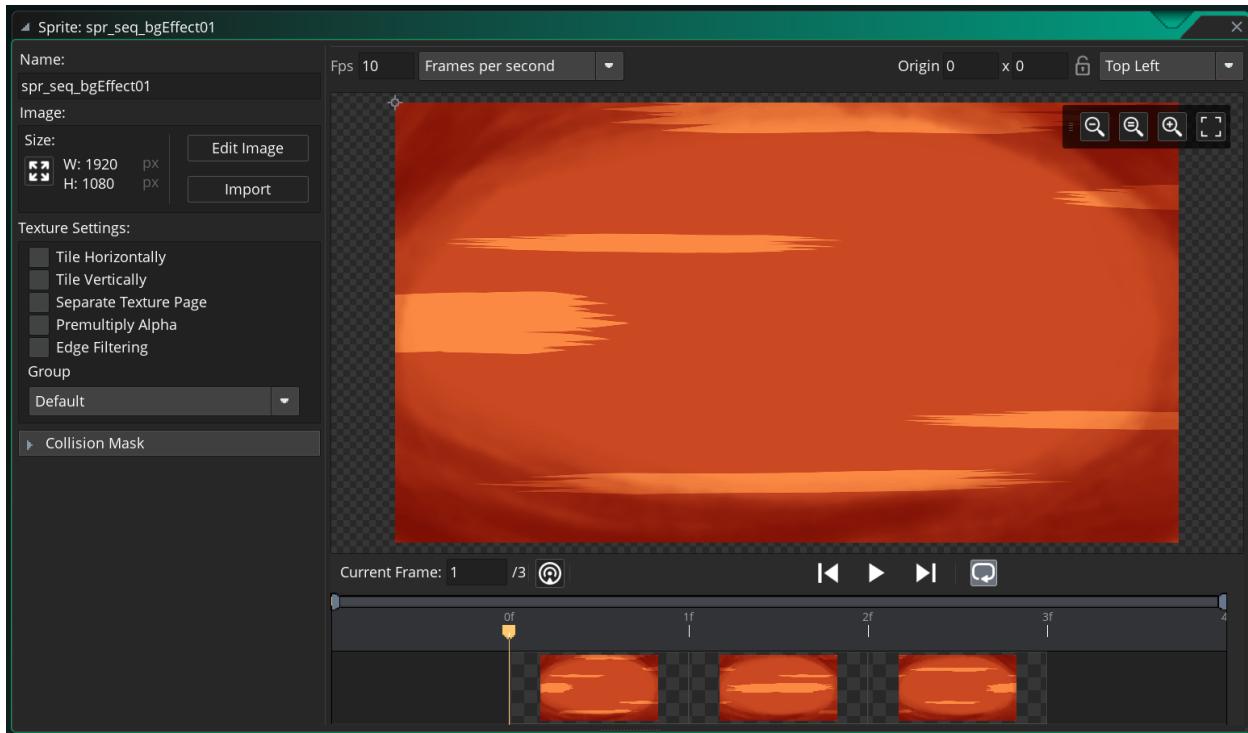
In the Asset Browser, right-click on the Sprites group and make a new group called Sequences.

Using File Explorer (Windows) or Finder (Mac), navigate to the Assets folder provided with this course and open the Sequences folder. Drag all the Sprites here into that new Sequences subgroup in the Asset Browser:

- spr_seq_bg_effect01
- spr_seq_bg_effect02
- spr_seq_bg_effect03
- spr_emote_exclamation_strip03
- spr_emote_flustered_strip05
- spr_emote_grumble_strip03
- spr_emote_shockLines_strip04
- spr_emote_sweatBubble_strip05
- spr_solidWhite

Open each of the three bg_effect Sprites in the Sprite Editor. For each one:

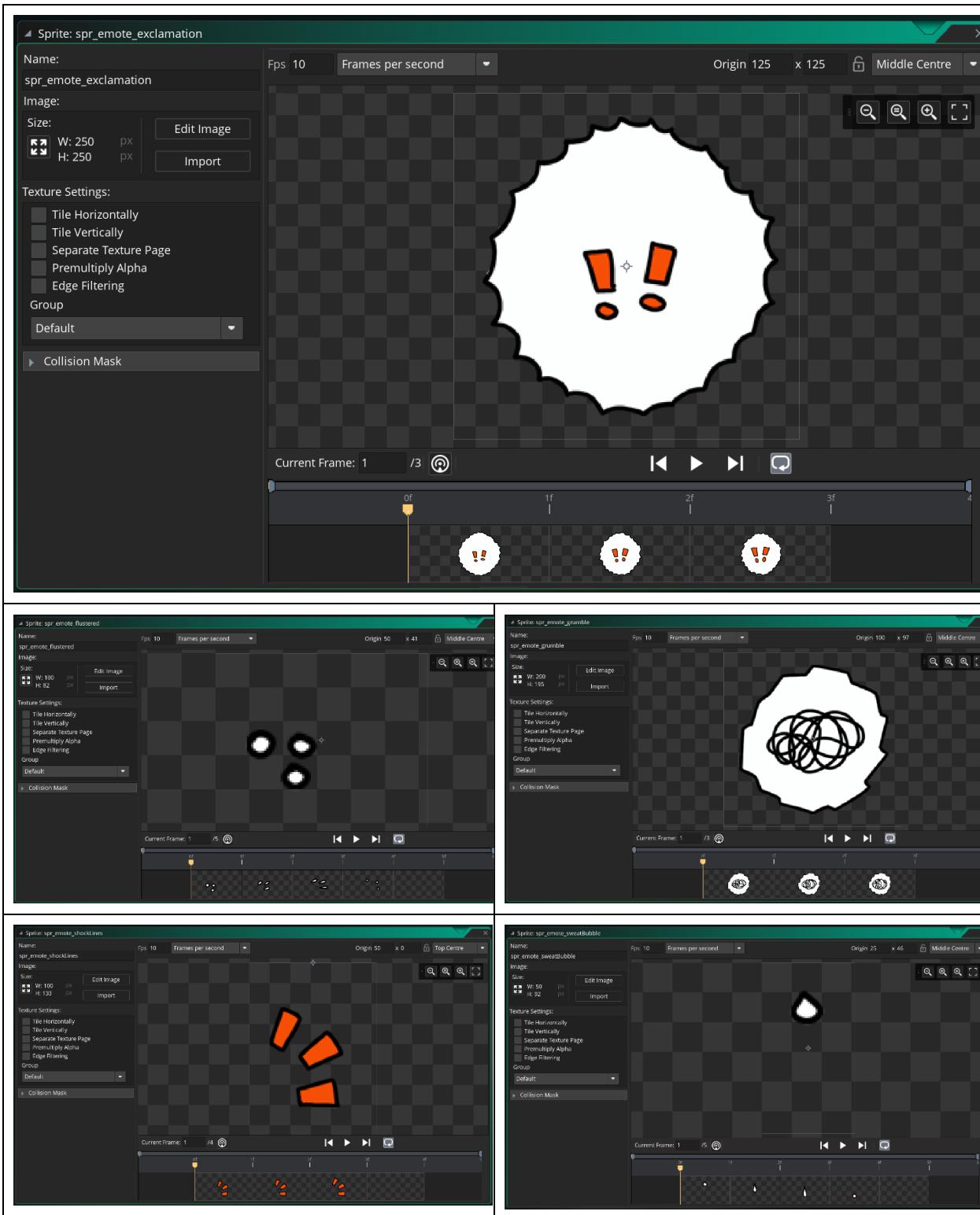
- Set the FPS value to 10 (or something else if you prefer)
- Set the Origin to be Middle Centre



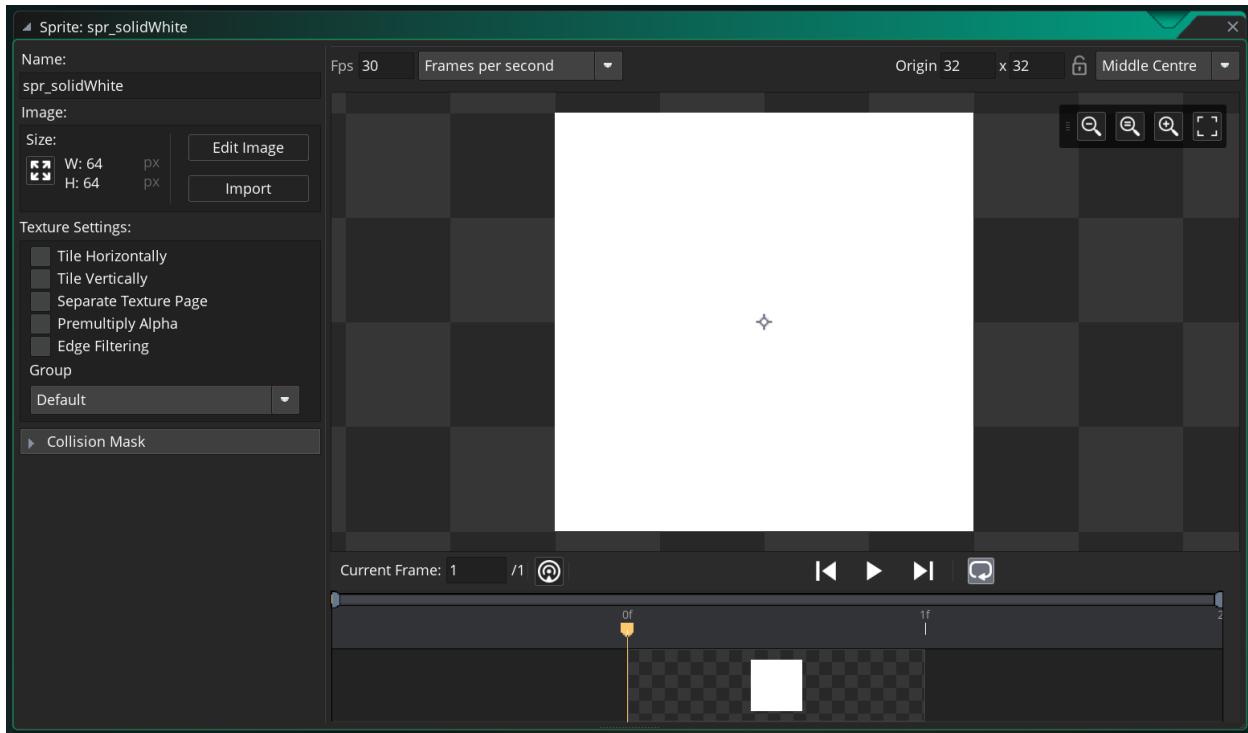
Setting the FPS of one of the sequence background images to 10.

Open each of the five spr_emote Sprites in the Sprite Editor and do the following:

- Make sure each Sprite Strip has been correctly converted to frames; if not, follow the steps we used before in [Converting a Sprite Strip manually](#) to do so
- Set the FPS value for each to 10
- Set the origin for each to Middle Centre
- Remove the _stripXX suffix from the asset



Next, open `spr_solidWhite` and in the Sprite Editor, set its Origin to Middle Centre.

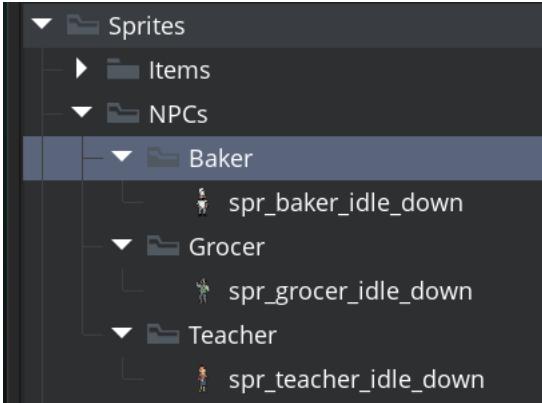


Editing the spr_solidWhite asset.

After you've imported and cleaned up all of the assets, right-click in the Sprites Group in the Asset Browser and choose Create Group once again. Name this new sub-group "Sequences" and drag these new assets in.

8.2 Preparing our new Baker assets

Next, we need to import several new Baker animations to use within our Sequence. This could get unwieldy if we don't plan now, so: If you have not done so already, organize your Sprite assets in the Asset Browser. Within the Sprites group, create an NPC group, and a new group for each of the three characters, like so:



Creating sub-groups for the Sprite assets for each of our three characters.

Using File Explorer (Windows) or Finder (Mac), navigate to the Assets folder provided with this course and open the Characters and Items folder. Drag all the Sprites list here into that new Sprites > NPCs > Baker subgroup within the Asset Browser:

- spr_baker_angry_strip04
- spr_baker_happy_strip04
- spr_baker_sad_strip04
- spr_baker_walk_down_strip04
- spr_baker_walk_left_strip04
- spr_baker_walk_right_strip04
- spr_baker_walk_up_strip04

As we just did with the emote Sprites, open each of these new Baker Sprites in the Sprite Editor and do the following:

- Make sure each Sprite Strip has been correctly converted to frames; if not, follow the steps we used before in [Converting a Sprite Strip manually](#) to do so
- Set the FPS value to 10
- Set the origin for each to between the Baker's feet
- Remove the _stripXX suffix from the asset



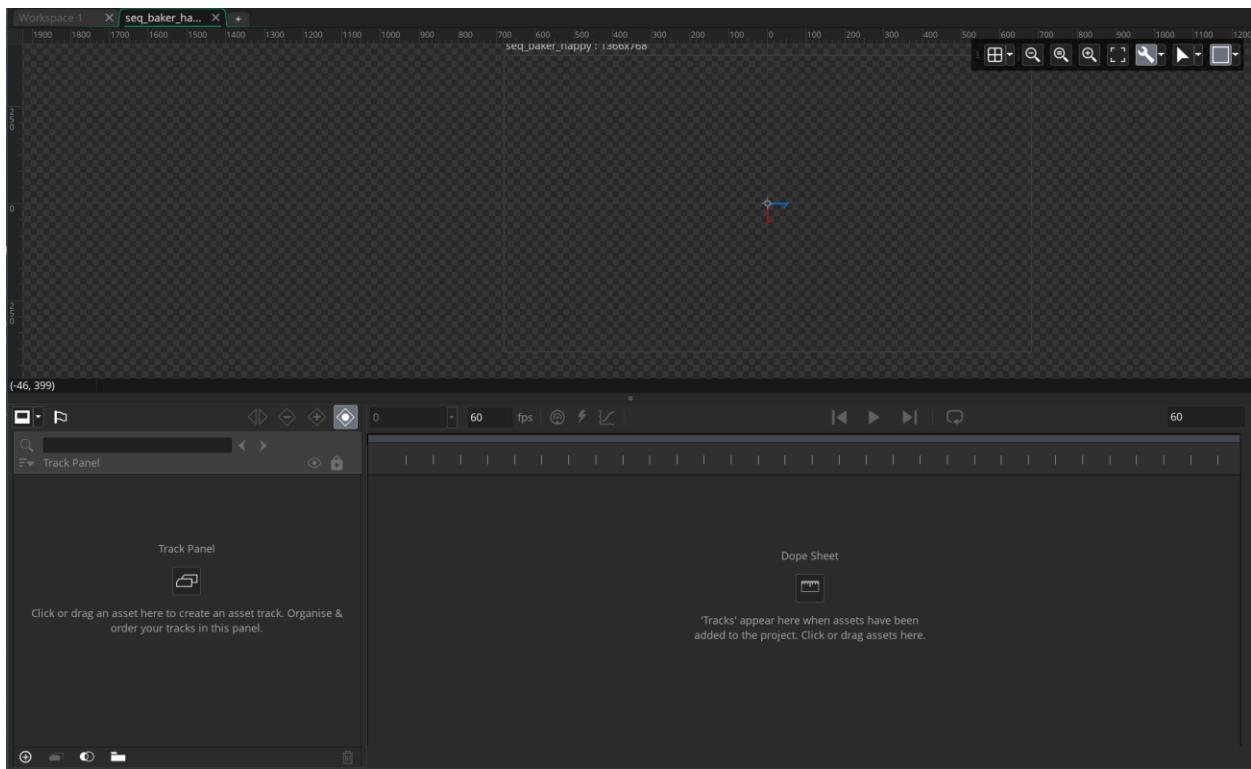
Editing the new Baker Sprites to have a correct FPS setting and Origin. Note how we've removed the _stripXX suffix from each Sprite and placed them in the Baker sub-group we previously created in the Asset Browser.

Once all these new assets have been dealt with, it's time to create our first Sequence.

8.3 Starting our first Sequence

Right-click on the Sequences group the Asset Browser and choose Create > Sequence; name this new asset seq_baker_happy.

You'll find yourself in the Sequences Editor. Details of all the main functions of the Sequence Editor can be found in the GameMaker Studio 2 manual, so we'll just go over what we need as we work.



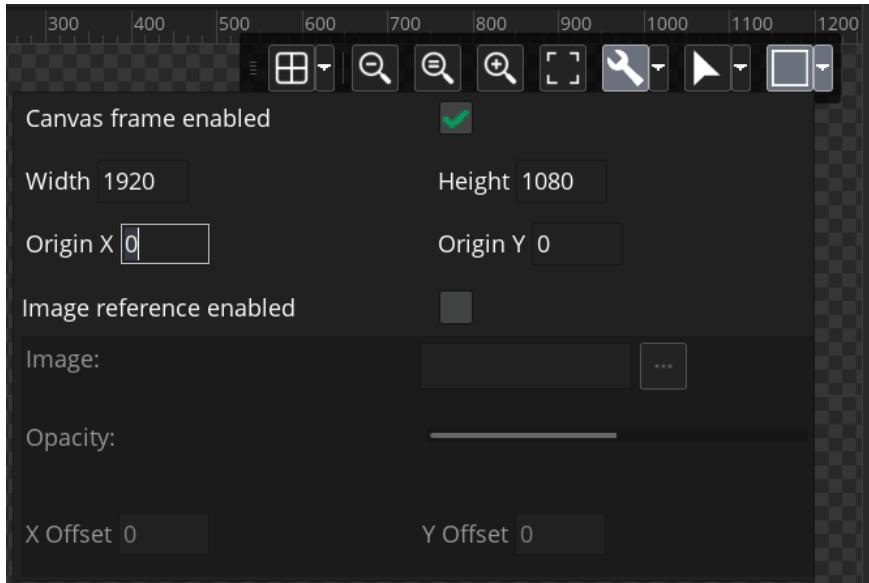
The GameMaker Studio 2 Sequence Editor.

Sequences and the Sequence Editor may look familiar if you've ever used a video editing program before. On the top you've got the Canvas view, on the bottom-left you have the Track Panel, and on the bottom-right you have the Dope Sheet (your timeline).

A Sequence is like a timeline animation, but we can do other things like call Scripts and send out messages to make them dynamic within our game. And like many other similar tools, Sequences use *keyframes* to let you edit various parameters of the assets you place in them.

8.4 Setting up the Sequence Canvas

First, look at the Canvas Toolbox (the horizontal toolbar at the top right of the Canvas view). Click the drop-down arrow on the right-most tool to access canvas settings. Change the width to 1920 and the height to 1080 (the same as our game's Camera if you recall). Make sure the Origin X and Y are both set to 0.



Editing the width and height of our Sequence canvas with the Canvas Toolbox.

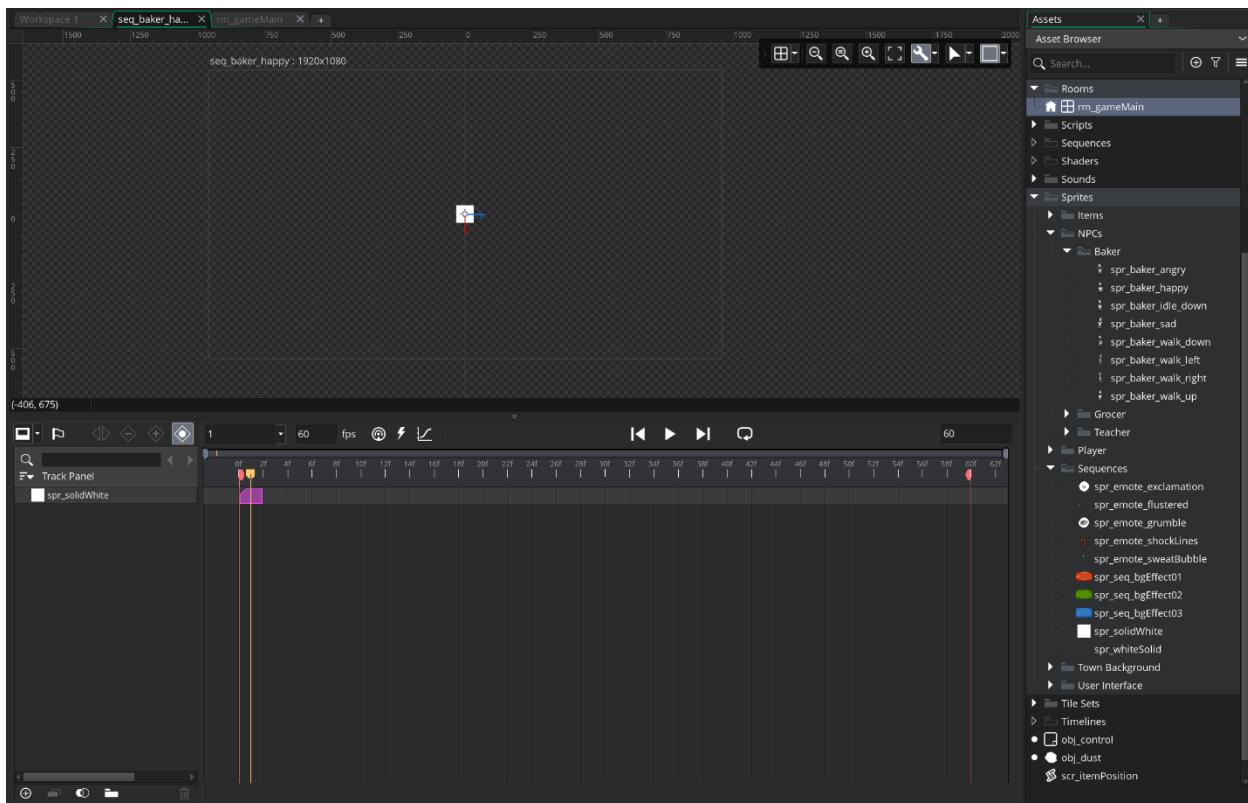
Next, let's start designing something! This is going to be the “happy” Sequence for our Baker NPC — it will be the one that plays when we give him the item he wants the most.

8.5 Adding our first track

From the Asset Browser, drag the `spr_solidWhite` Sprite onto your Sequence's Dope sheet.

A new *track* will be created with the name of the asset, and its duration (relative to the length of your Sequence) will be represented by a new *asset key* on the Dope Sheet timeline (the colourful block).

You should also see `spr_solidWhite` (a white square) on your Sequence canvas. If you don't, make sure to move the yellow playhead on the Dope Sheet to a point on the timeline where the solid sits.



Dragging in our first asset (spr_solidWhite) creates a new track in the Sequence Editor, and a new asset key in the Dope Sheet.

If your spr_solidWhite track looks like it's somewhere in the middle of the timeline, you can click and drag it all the way to the left to place it at the beginning of the Sequence. The left edge (with the diagonal corner) should align with 0f (which is the first frame).

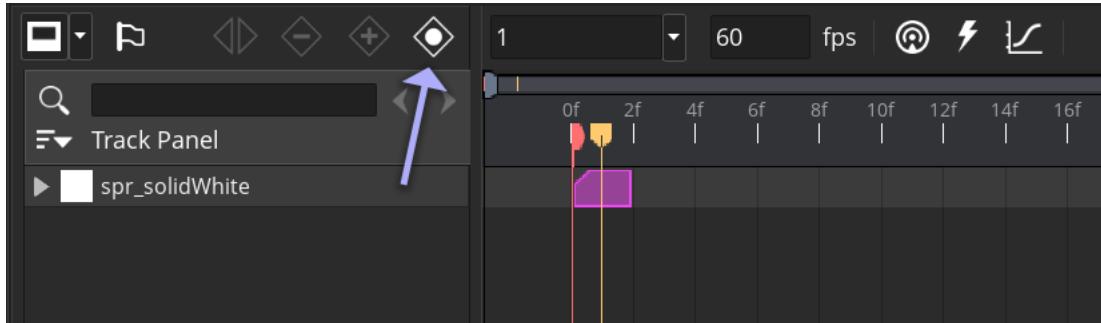
Now, we need to do three things with our white solid:

- Resize it so it covers the entire canvas
- Increase its duration so it lasts longer within the Sequence
- Create a “fade in” effect

8.6 Turning off automatic keyframes

Before we do anything further, though, here’s an important note: look above the Track Panel to see a series of diamond-shaped icons. (You can hover your mouse over them to see what they do.)

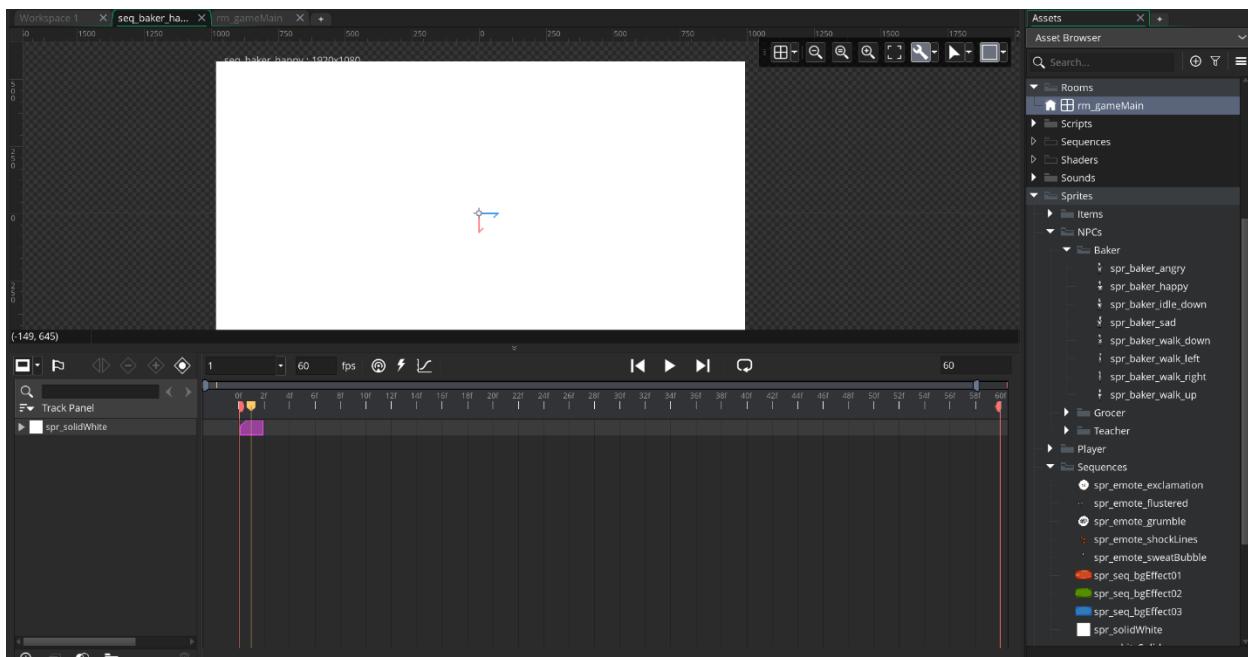
The right-most icon (⌚) lets us automatically create keyframes just by making changes to assets on the Canvas. This is fantastic in many situations, but for what we need to do right now, it's better if we turn this off. So, click it to make sure it's not highlighted, like so:



Disabling the option to automatically create keyframes by clicking the right-most icon above the Track Panel (as indicated by the arrow here).

8.7 Editing the first track

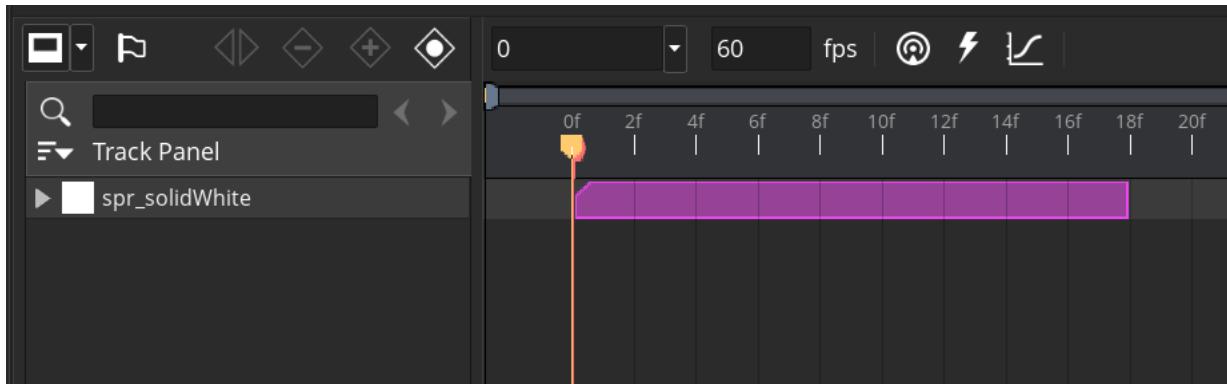
With that done, click on the white solid currently floating within your Canvas. You'll see familiar resize and rotation nodes appear when you do so. Using these, resize the white solid to cover the entire Canvas. (It's okay if it extends beyond the Canvas bounds.)



Stretching the white solid to cover the entire Canvas.

Next, we need to extend the asset key for the spr_solidWhite track, so the solid doesn't just disappear after a couple of frames.

Hovering your mouse over the right edge of the asset key will reveal a double-arrow; click and drag the asset key to stretch it out. (We set it to 18 frames long, but feel free to play around.)



Resizing the asset key so the white solid's duration is longer.

8.8 Adding parameter tracks and keyframes

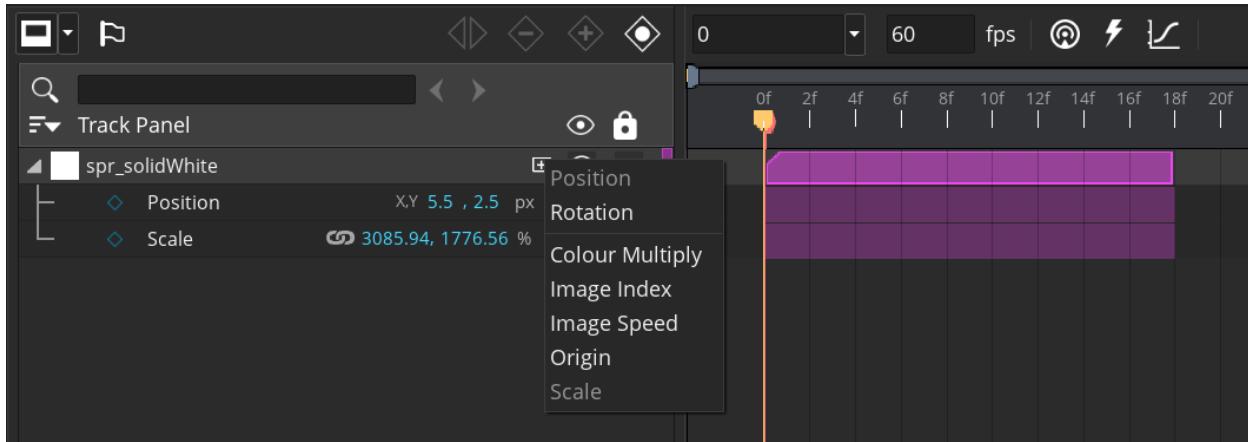
Now that our solid lasts more than a few frames and covers the whole Canvas, we need to create a fade effect. To do this, we're going to make our solid transition from transparent to opaque.

In the Track Panel, click the arrow to the left of the track name (spr_solidWhite). A series of rows will open up; these are called *parameter tracks*.

As the name suggests, for each parameter of an asset you edit (such as position, scale, opacity/colour and more), a new parameter track is created in the Track Panel.

You can see that the two existing parameter tracks, Position and Scale, have values reflecting the changes we made when we manually stretched out our white solid on the Canvas.

What we need to do is create a new parameter track so we can create our fade effect. To do this, click the Add parameter track button in the Track Panel, and choose Colour Multiply from the menu that appears.



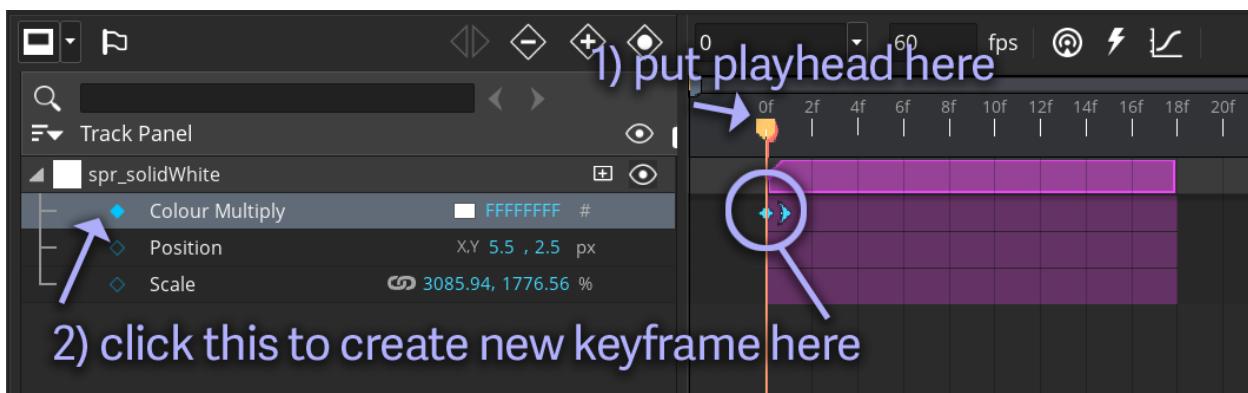
Clicking the “Add parameter track” button to add a Colour Multiply track.

This will create a new parameter track for Colour Multiply. Now, we don’t want to just make the white solid transparent all the time; we want it to fade from transparent to opaque — and for that, we need to use *keyframes*.

A keyframe is just that; a moment in time where something happens. Like similar editing tools, the Sequences Editor automatically transitions between parameters you set at keyframes — which is how we get fades, rotations, scaling and animation.

So, let’s create our first keyframe. Make sure to move the yellow playhead to the beginning of the Dope Sheet, since we want our solid to begin completely transparent.

Then, click the hollow blue diamond to the left of Colour Multiply to create a new keyframe, like so:

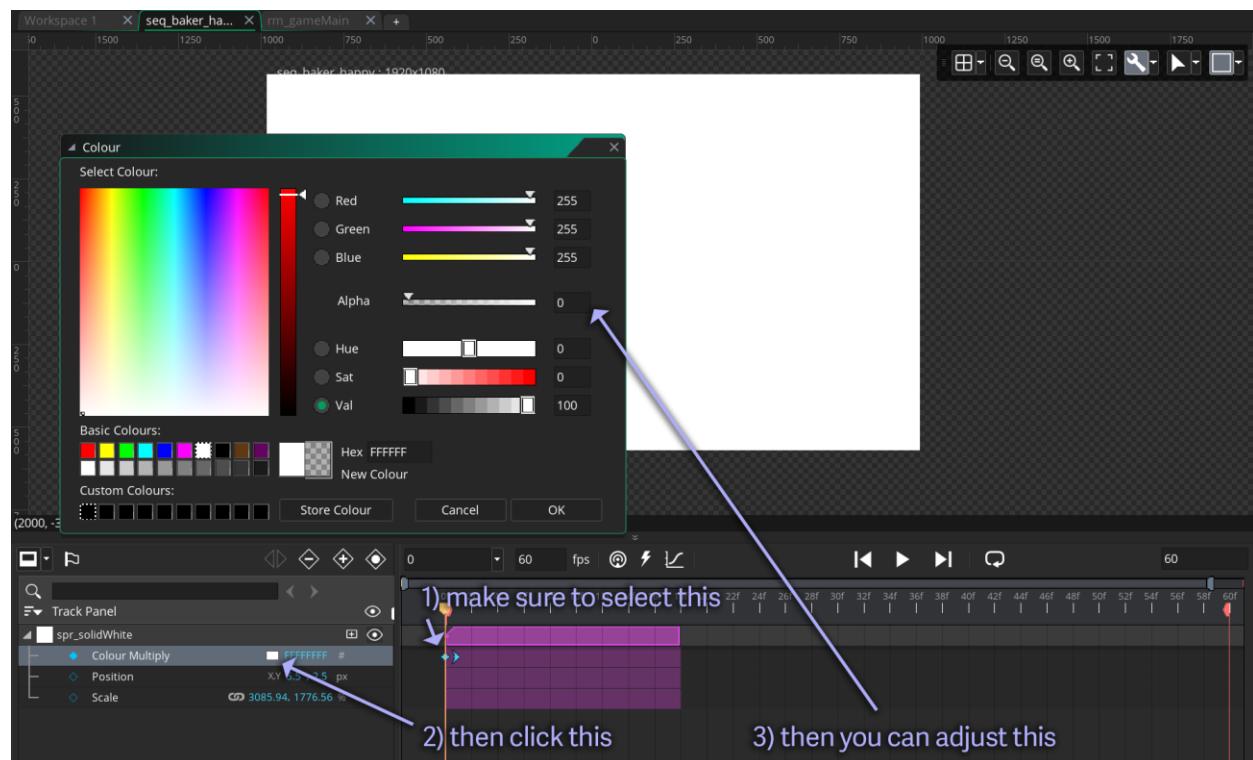


Creating a new keyframe at a specific point in the timeline.

Next, we need to change the value of Colour Multiply at that specific keyframe. Here is where you need to be careful: make sure to click on the new keyframe you created on the timeline, so it is highlighted blue.

Then, click on the small white swatch in the Colour Multiply parameter track to bring up a colour picker.

Set the Alpha channel all the way to 0 (either by changing its value or by dragging the triangular slider). When you're done, click OK.

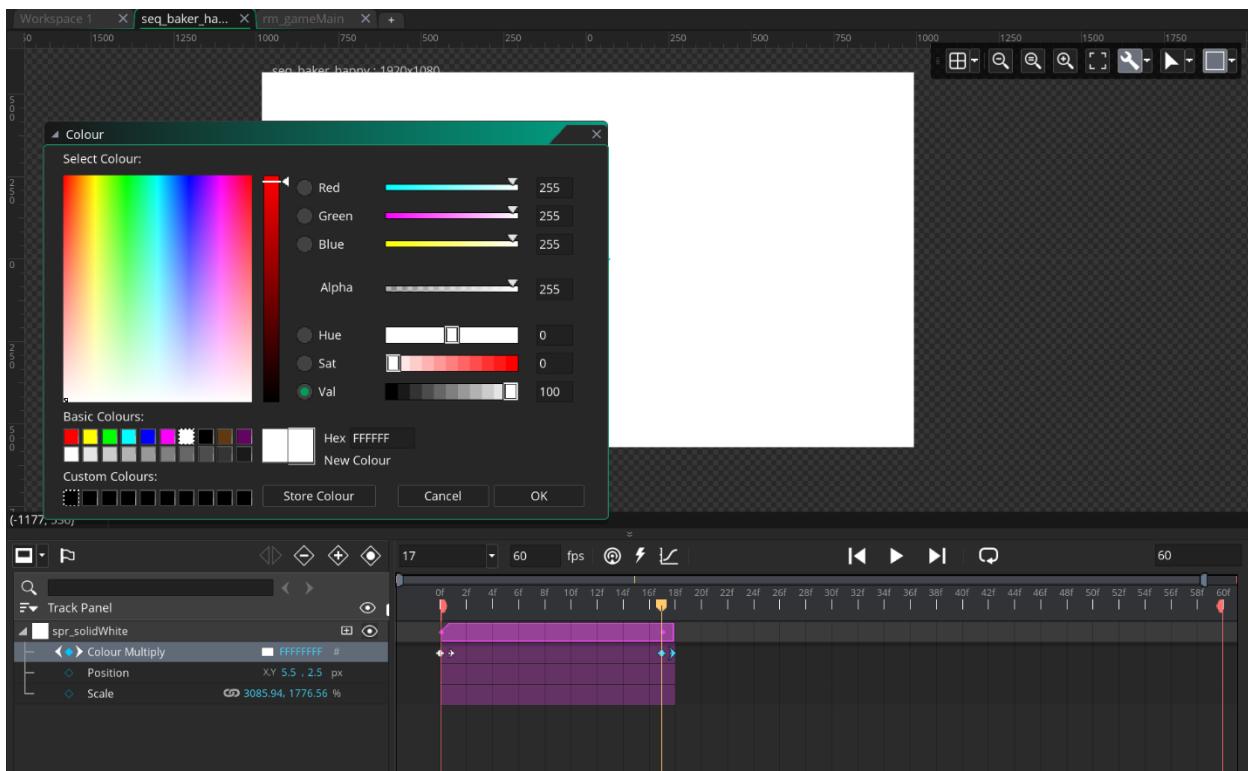


Changing the Alpha channel on the Colour Multiply parameter track for a specific keyframe.

Now your white solid has disappeared from the Canvas (which is what we wanted)! To finish the fade effect, we need to create and edit another keyframe.

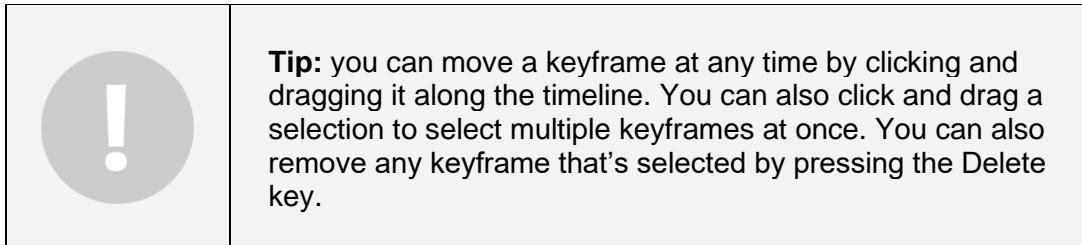
Move the yellow playhead to the end of the spr_solidWhite asset key on the Dope Sheet. Click the hollow blue diamond to the left of Colour Multiply to create a second keyframe.

Exactly as we just did, make sure to select the new keyframe, then click the small swatch in the Colour Multiply parameter track. This time, change the Alpha channel back to being opaque (255).



Changing the Alpha value for the second keyframe.

With this done, feel free to move the yellow playhead back and forth (also known as *scrubbing*) to test the transition. The solid should now fade in from transparent to opaque white.

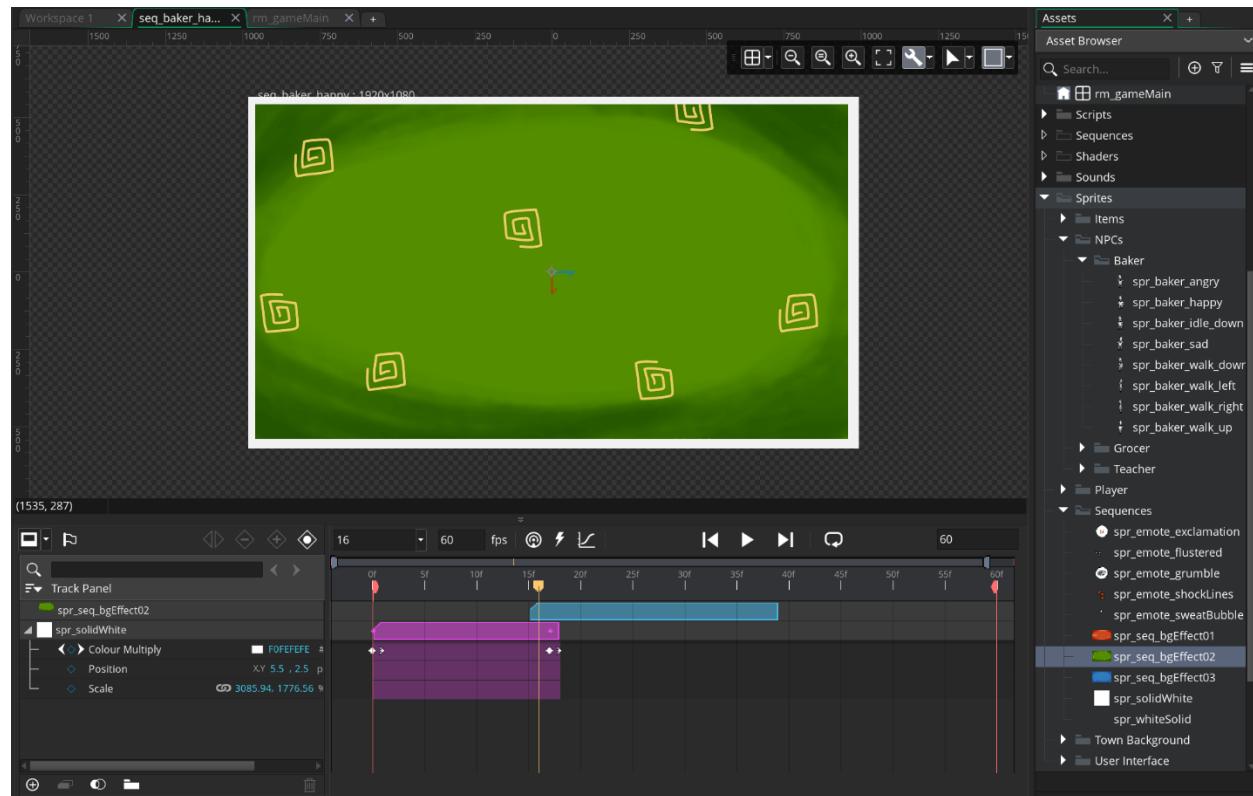


8.9 Adding a background to our Sequence

With this initial fade complete, we're going to add a background to the Sequence, and edit it as well.

From the Asset Browser, drag spr_seq_bgEffect02 onto the Track Panel in the Sequences Editor. (If this new track appears *below* the spr_solidWhite track, then drag it above in the Track Panel.)

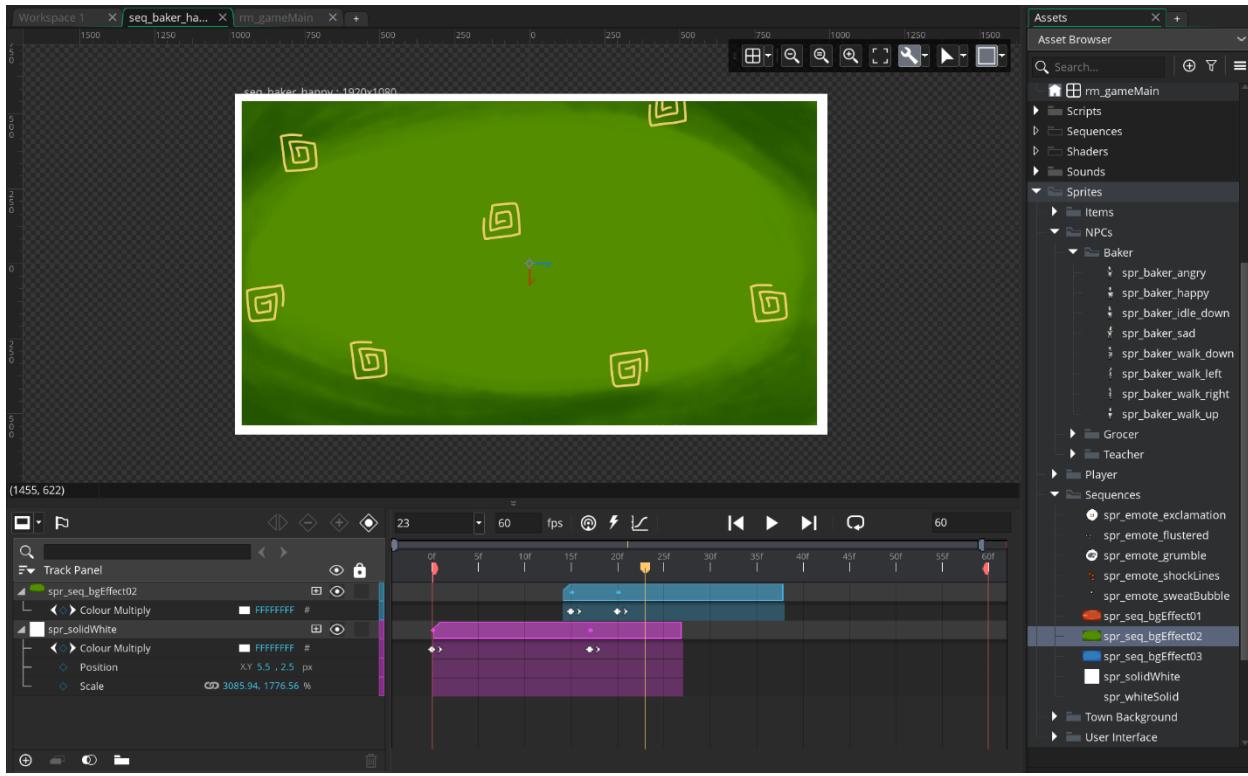
Click and drag the asset key for spr_seq_bgEffect02 so it begins a little bit before the white solid ends, like so:



Adding a background asset to the Sequence and positioning it within the timeline.

Next, we're going to add the same fade effect to this background asset that we did with the white solid.

Using the same steps we just took in [Adding parameter tracks and keyframes](#), add a fade effect to the background track, like so:



Adding the same fade effect to our new background track.

Note the position of the keyframes; this gives the `spr_seq_bgEffect02` track a faster fade in. You may need to extend the asset key for `spr_solidWhite`, or else as `spr_seq_bgEffect02` fades in, there'll be nothing under it, which will look odd.

When you're ready, press the Play button to preview the Sequence. You'll see that our solid fades in to white, and then the animated background fades in quickly on top of that.

!

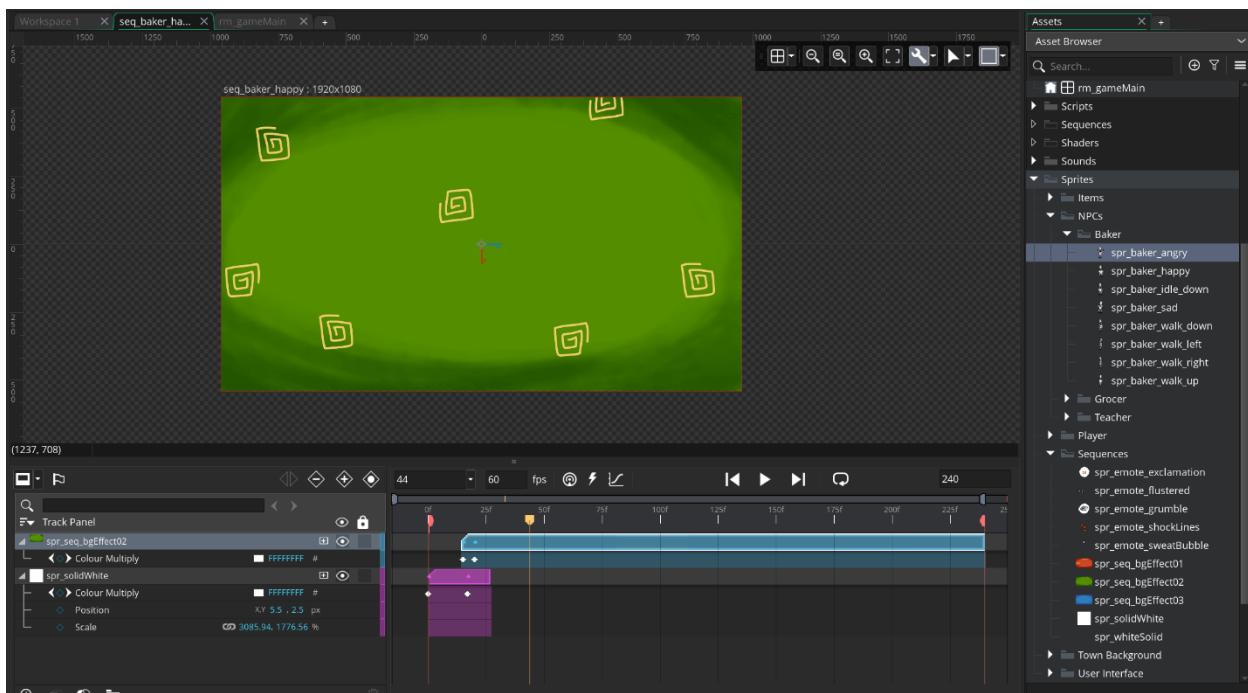
Tip: If you just move an asset key along the timeline, its keyframes *won't* automatically move with it. You'll have to select those as well and move everything as a unit, or Alt-click (on PC) or Option-click (on Mac) to select everything.

8.10 Extending our Sequence

You'll notice that our Sequence is very short. Let's make it longer so we can continue to add to it.

At the top right of the Dope Sheet, you'll see a number (60). This is the number of total frames in our Sequence. Change this number to 240 so we get a four-second sequence.

Next, click and drag the right side of the spr_seq_bgEffect02 asset key all the way to the end of the Sequence. Since this asset is an animated Sprite, it will continue to loop until the end of the timeline.



Extending the Sequence length and the duration of the background asset.

8.11 Bringing in our Baker

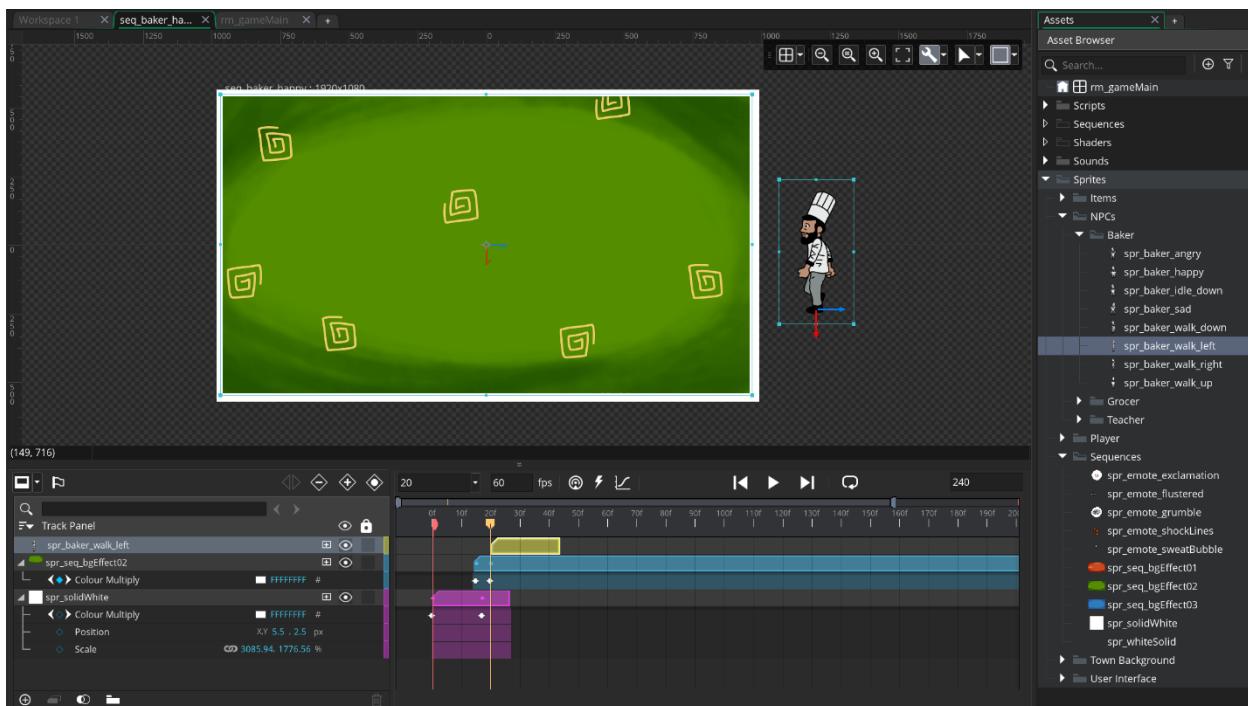
It's time to get our Baker in on the action. From the Asset Browser, drag the spr_baker_walk_left Sprite onto the Sequence Canvas. Position it outside of the Canvas and to the right.

This will create track for the Sprite; make sure this track is above the other two.



Tip: If you need more room to move while working in the Dope Sheet you can press CTRL-plus and CTRL-minus to zoom in and out (CMD on Mac).

The position of the yellow playhead when you dragged the Sprite into the Sequence editor determines where on the timeline this new track will start. Move the asset key for the Baker sprite so that it begins around where the background completes its fade.

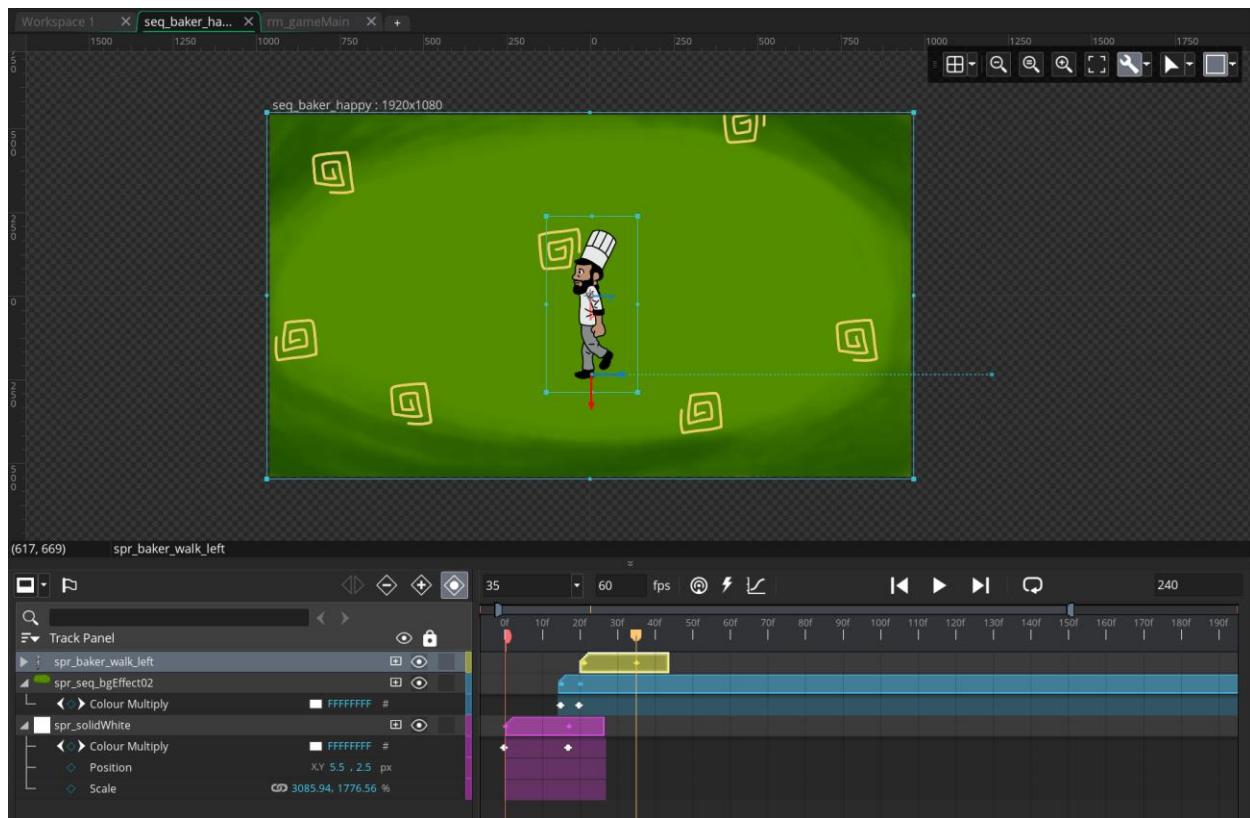
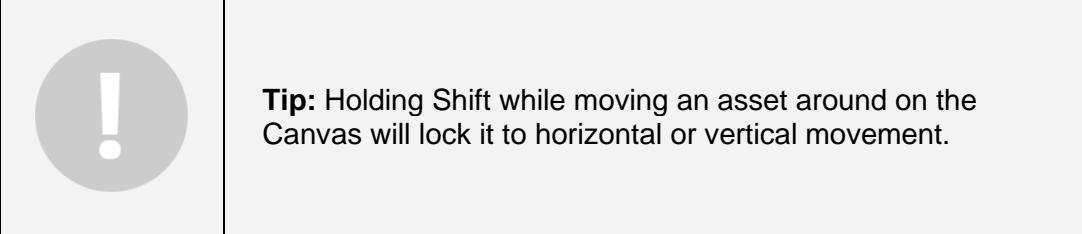


Positioning the Baker sprite outside of the Canvas, and its asset key just after the background fades in on the Dope Sheet.

Make sure the playhead is at the beginning of spr_baker_walk_left's asset key and that the asset key is selected.

Click the Record a new key button (◆) at the top of the Track Panel to create a new keyframe. Then, click the “Automatically record changes” button (◇) to track what we do next.

Move the playhead ahead about 15 frames or so on the Dope Sheet. Then, on the Canvas, click and drag the Baker Sprite to the left, so he appears in the centre of the scene.



By automatically recording our actions, we can move the playhead to a point on the timeline, move our Baker asset around on the Canvas, and the Sequences Editor will create the correct keyframe and parameter track.

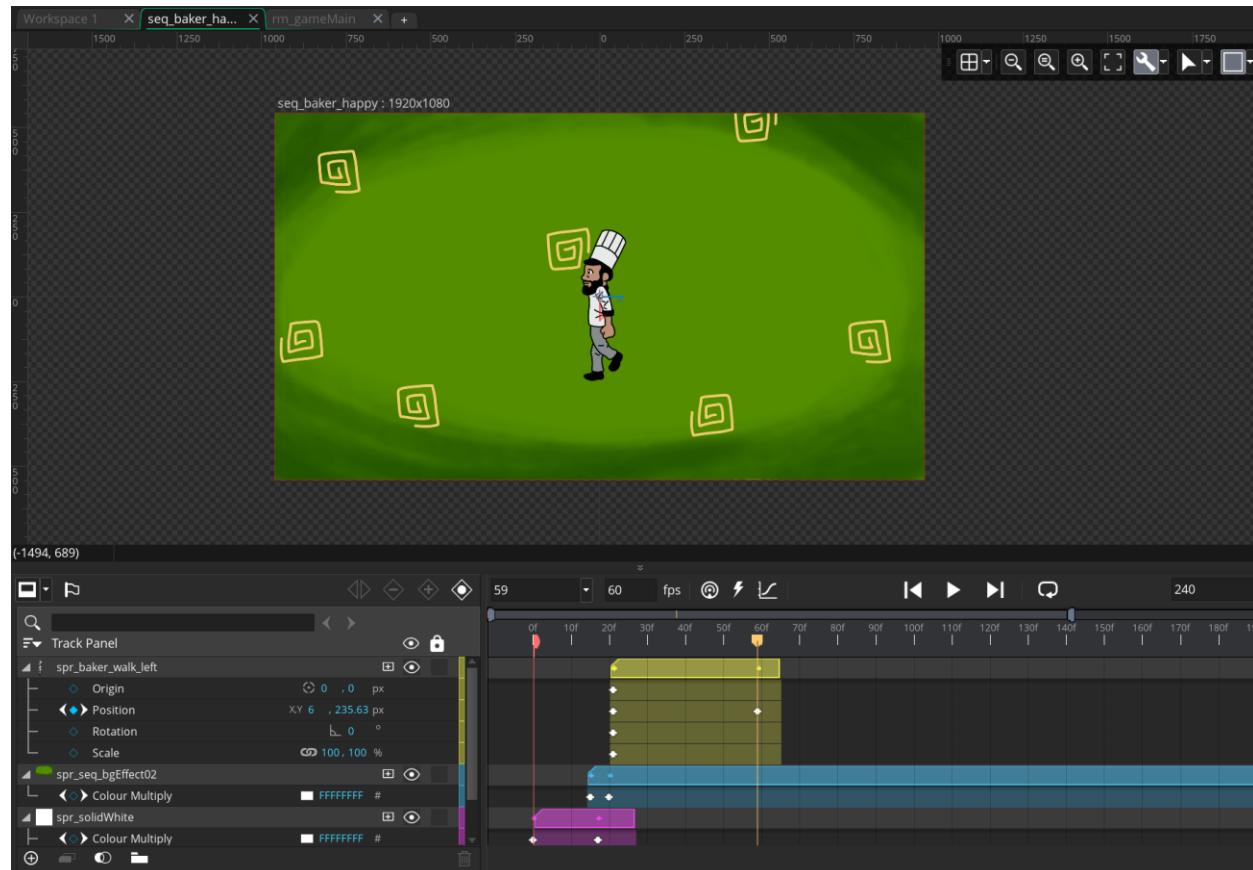
If you click the arrow to the left of the `spr_baker_walk_left` track in the Track Panel, you can see all the parameter tracks for this asset. You'll notice that there are two keyframes already for the Position parameter track because we moved the Baker sprite.

Let's do one last thing before moving on; depending on how long it takes for the Baker Sprite to get to the centre of the Canvas, it may look like he's walking too slowly or too quickly.

So, to ensure that it looks correct, let's use a combination of two techniques.

First, make sure to click the “Automatically record changes” button at the top of the Track panel to turn this feature off. We’re going to need to do some manual work.

Then, extend the asset key for the Baker along the timeline if his movement seems too abrupt. Make sure to drag the second keyframe on the Position parameter track to increase the duration of that animation.

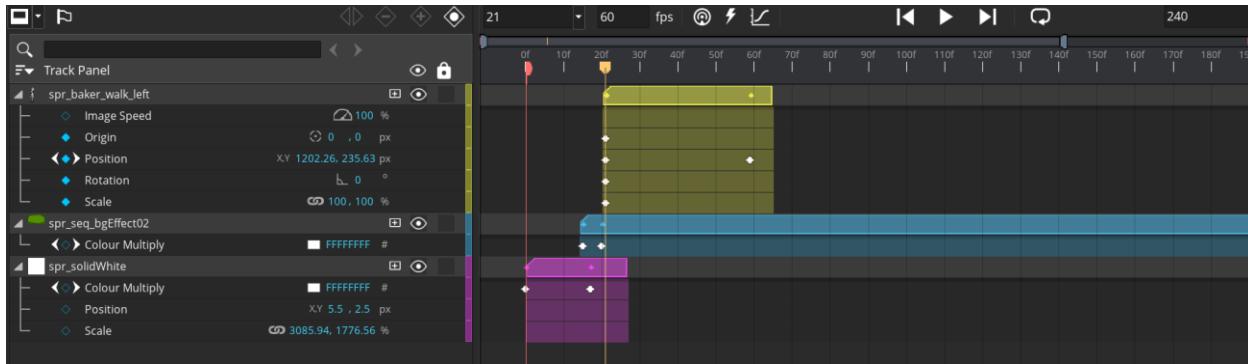


Extending the asset key for the Baker Sprite and moving its Position keyframe to create a longer animation.

Next, Click the Add parameter track button on the spr_baker_walk_left track in the Track Panel, and choose Image Speed.

Image Speed, if you recall, is a built-in variable to control the animation speed of Sprites. It’s a multiplier for the FPS setting in the Sprite Editor. By default, it’s set to 100% (or, $FPS \times 1$).

By changing this here, we can manually adjust the animation speed of any Sprite we place in a Sequence. So, let’s tweak our Baker’s animation speed a bit with this.

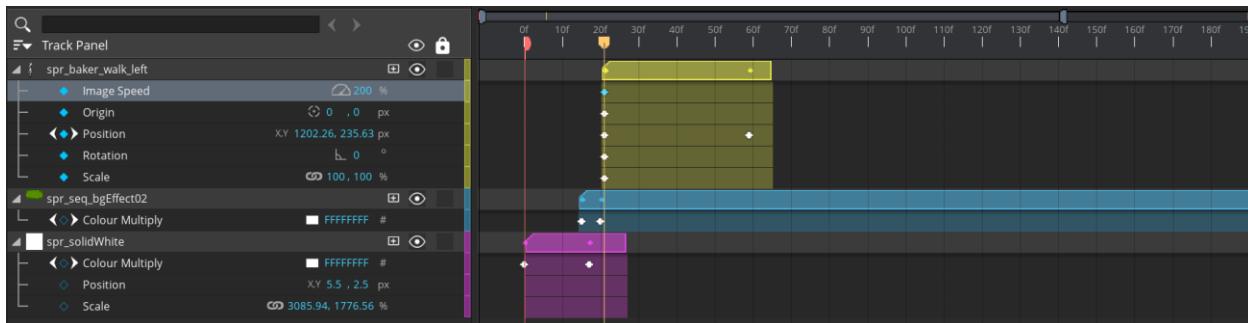


Adding the Image Speed parameter track to the Baker Sprite.

Move the playhead to the beginning of the spr_baker_walk_left asset key.

Click the hollow blue diamond to the left of the Image Speed parameter track to record a keyframe for this parameter.

Click the keyframe on the Dope Sheet to select it and then change the Image Speed value to a different number (we chose 200%). Since you only have the one keyframe here, this value will persist for the duration of the asset key.



Changing the value of the Image Speed parameter track.

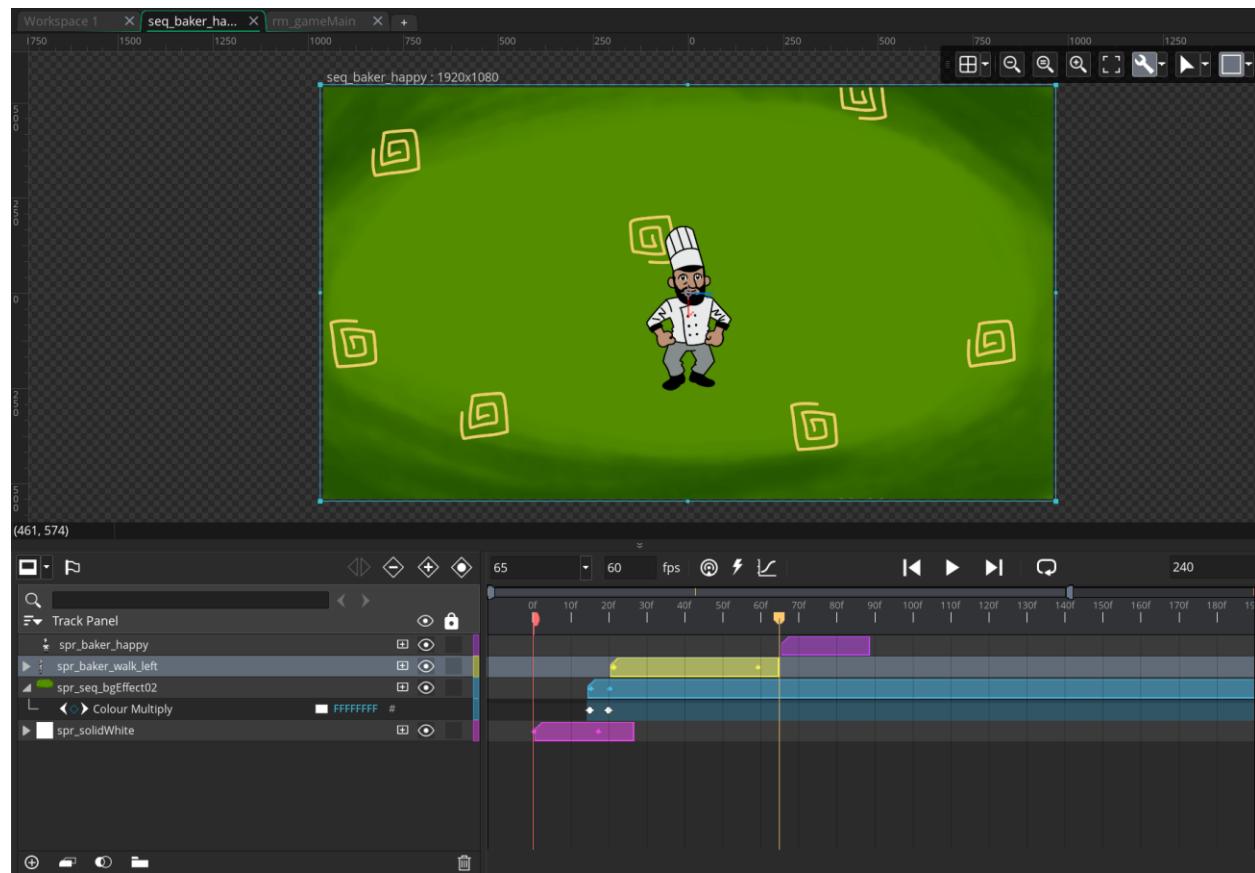
Play the Sequence and observe the change in animation timing and speed. If it doesn't look or feel quite right to you, go ahead and adjust the Image Speed value further, or adjust the animation within the Dope Sheet.

When you're happy with the results, make sure to save your project, and let's move on to the next step!

8.12 Making our Baker dance

Next, we're going to show off our Baker's dancing skills. Make sure the Automatically record changes button is off. Then, from the Asset Browser, drag the spr_baker_happy Sprite onto the Canvas.

Position the dancing Sprite to match where your walking Baker ends up in the middle of the scene, and make sure the asset key for the spr_baker_happy Track begins where the walking Baker track ends, like so:



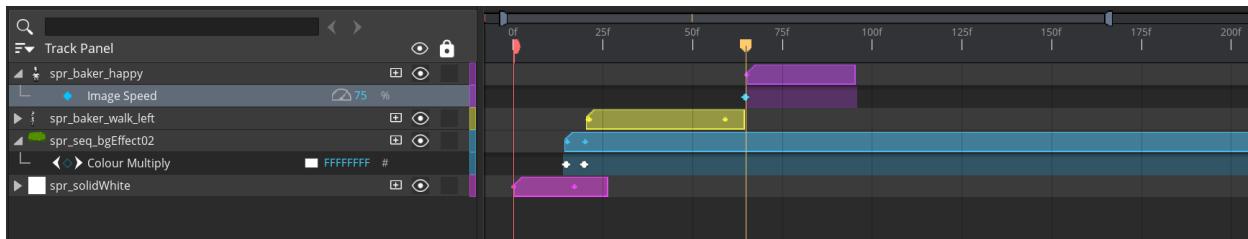
Position the new dancing Baker track.

If you play the Sequence (or scrub the playhead), you'll see that the Baker gets his groove on, back and forth. Right now, however, he's moving a little too quickly.

So, click on the spr_baker_happy track in the Track Panel, and add an Image Speed parameter track.

Position the playhead where the spr_baker_happy asset key begins and click the hollow blue diamond to the left of Image Speed to create a keyframe for this parameter.

Select the new keyframe, and change the value of Image Speed to 75%; this should give you a better result.

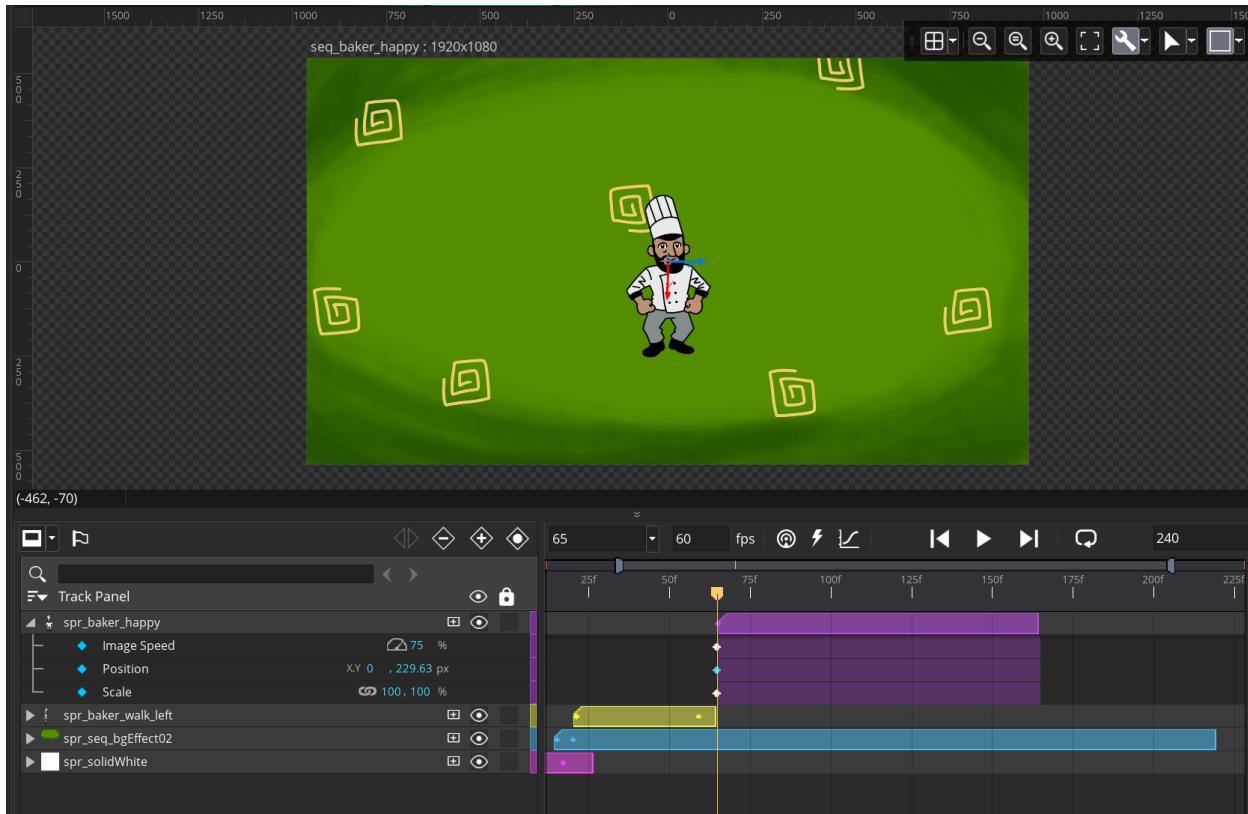


Adding the Image Speed parameter track for the dancing Baker Sprite and changing its value with a keyframe.

Next, we want to add a fun effect wherein the Baker doubles in size each time he loops through his dancing animation, to create some fun drama.

Add two more parameter tracks to the spr_baker_happy track: Scale and Position.

Move the playhead to the frame on the timeline where the Baker asset key first appears. Record a keyframe for both the Position and Scale tracks, like so:



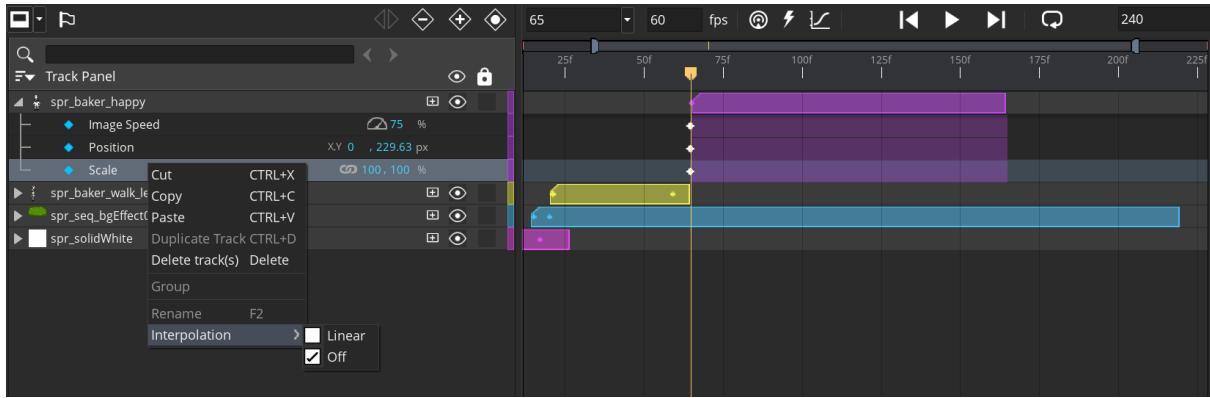
Recording keyframes for the new Position and Scale parameter tracks.

Then, move the playhead about 30 frames down the timeline. If your `spr_baker_happy` asset key isn't long enough, just stretch it out.

We want to make the Baker suddenly snap to 300% of its size. However, by default, adding keyframes for a parameter track (like `Scale`) and changing their values will automatically cause *interpolation* between the keyframes.

In other words, if we just add a second keyframe and change the `Scale` of the Baker to 300%, he will gradually *grow* from 100% to the larger size, which isn't what we want. So, to make our lives easier, we're going to turn interpolation off.

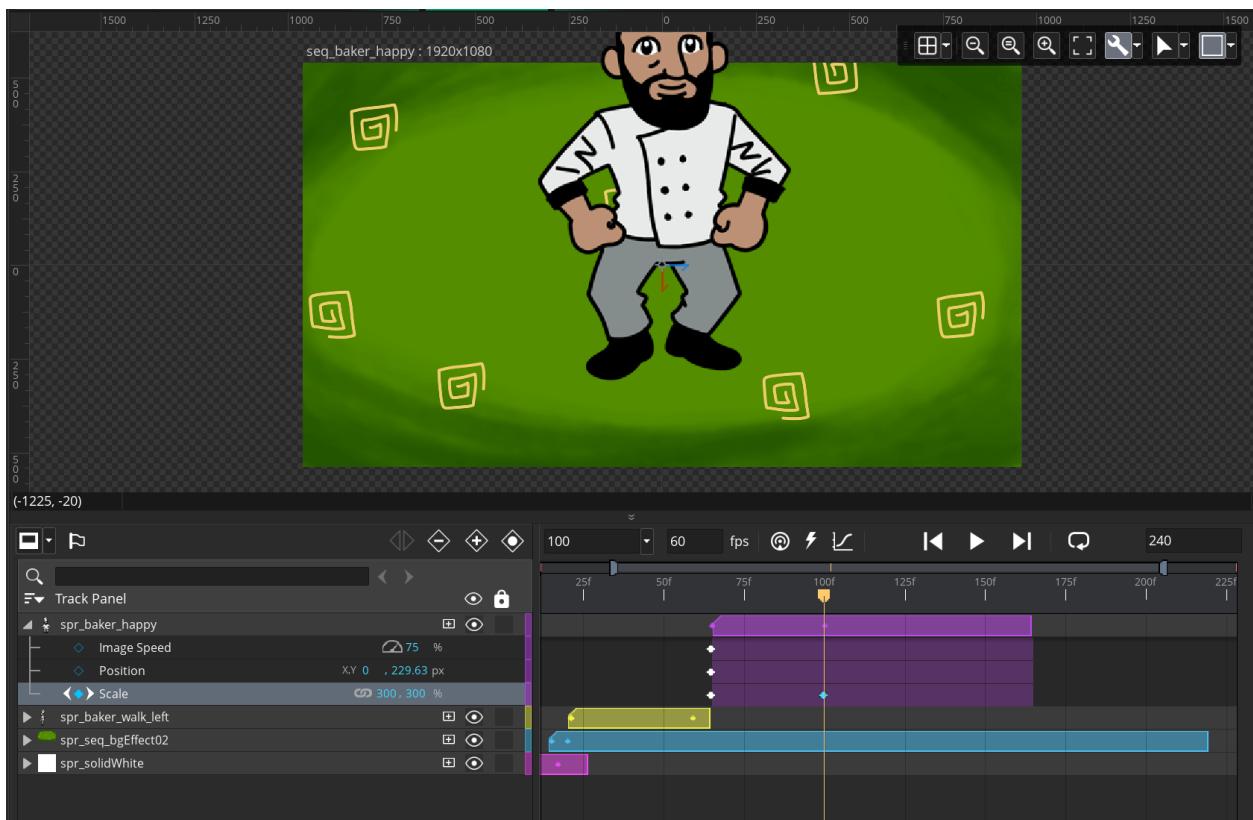
Right-click the `Scale` parameter track for the `spr_baker_happy` track (in the Track Panel) and choose `Interpolation > Off`, like so:



Turning off interpolation for the Scale parameter track.

Then, record another keyframe for the Scale parameter track.

Select this new keyframe and change the Scale values in the Track Panel to 300%, 300%. Now we've got a nice, big Baker. If you scrub the playhead back and forth, you'll see that the Baker doesn't change size gradually, but snaps to his new size wherever we placed the new keyframe.



Creating a sudden Scale change with the Baker dancing track

You'll notice that your Baker is probably standing too far up on the Canvas. Let's correct the Baker's position to keep him in frame.

As we just did for the Scale parameter track, right-click on the Position parameter track in the Track Panel and choose Interpolation > Off.

With the playhead on the same frame as the new Scale keyframe, click the Automatically record changes button again to turn this on.

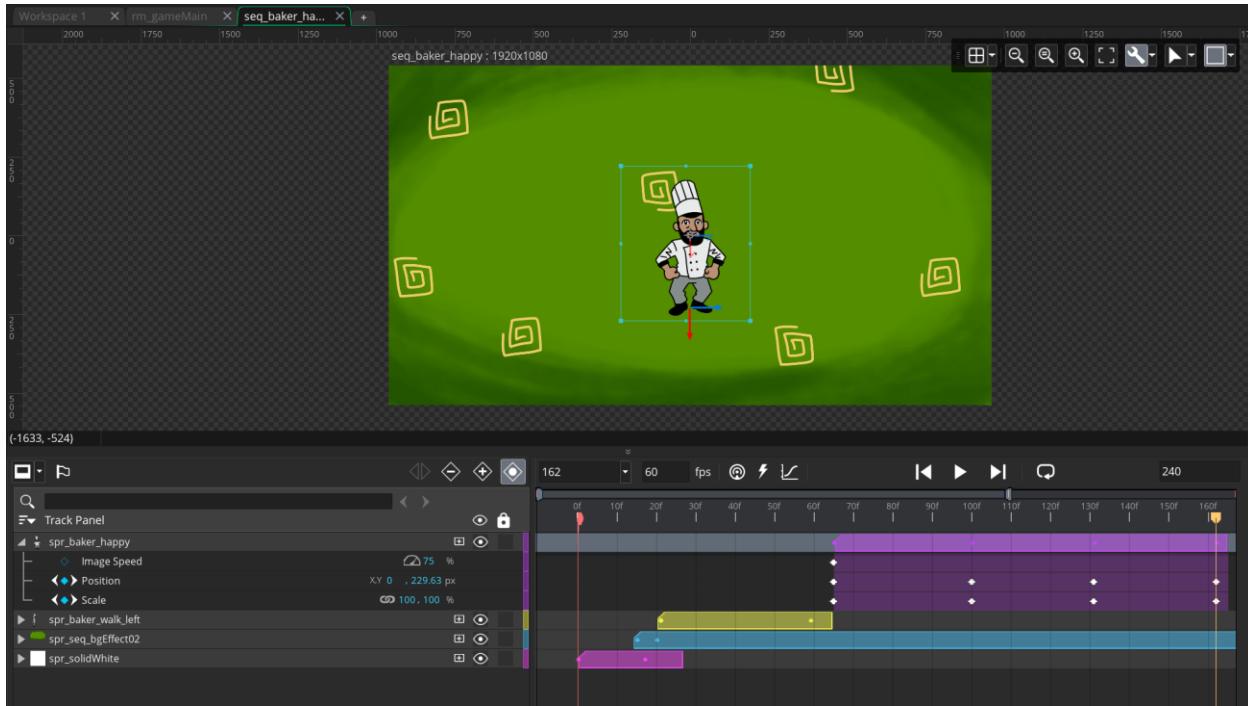
Then, click and drag the Baker asset on the Canvas down so he is more centered on the Canvas. You can hold the Shift key to restrict movement and make this easier, or use the *Transform Gizmo* (the red and blue arrows) to move the asset in specific ways.

A new keyframe for the Position track will be recorded automatically. You can click the "Automatically record changes" button again to turn this off once more.



Using the "Automatically record changes" feature to reposition the now-large Baker asset.

Repeat the above steps to "jump" the Baker's size two more times (from 300%, which you have, to 450% and then back to 100%). The following screenshot shows the keyframe setup we used for reference:



Creating an effect where the Baker “jumps” in size, then returns to normal.

8.13 Completing the Baker Sequence

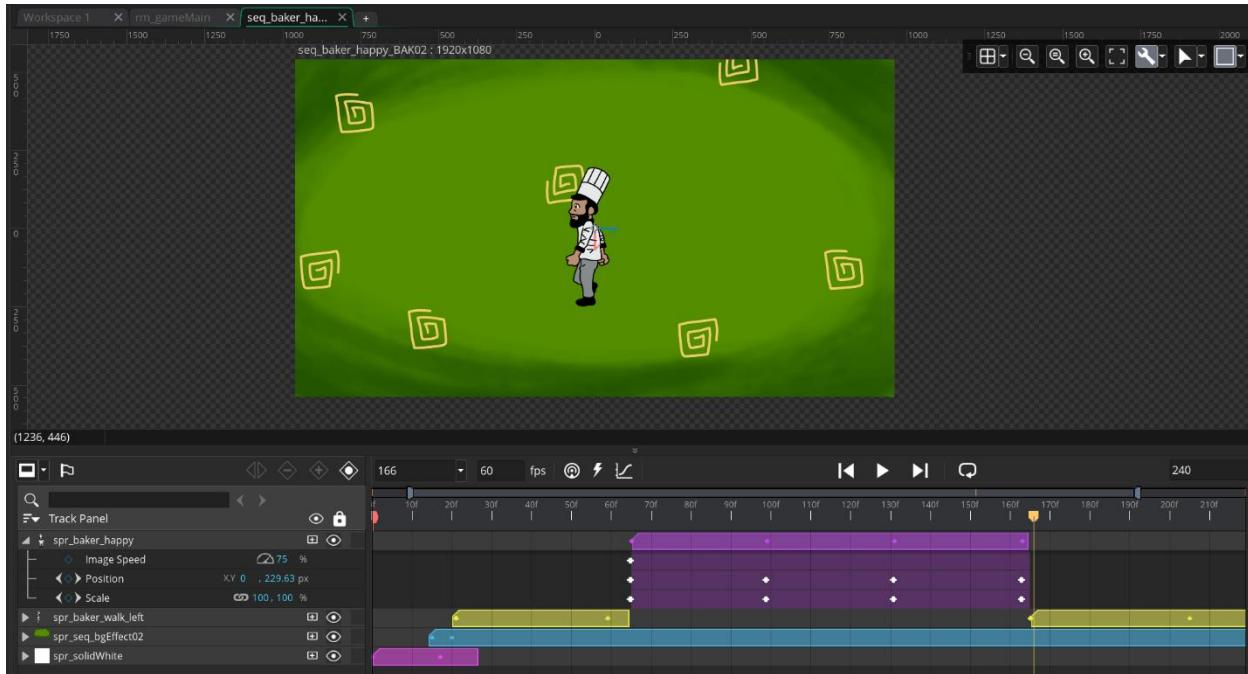
To end this animation, we’re going to do two more things: have the Baker walk off the Canvas, and then create another fade to help us transition back to our town gameplay.

First, make sure the asset key for the `spr_baker_happy` track ends a frame or two after the final pair of Position and Scale keyframes.

Next, click on the `spr_baker_walk_left` key asset in the Dope Sheet, and copy it (CTRL-C in Windows, CMD-C on Mac). We’re going to have our Baker continue to walk after his dancing routine ends.

Move the yellow playhead to the frame where the `spr_baker_happy` ends (you should not see the dancing Baker on the Canvas at this frame).

Paste (CTRL-V in Windows, CMD-V on Mac), and you should see another `spr_baker_walk_left` asset key from this point in the Dope Sheet, like so:

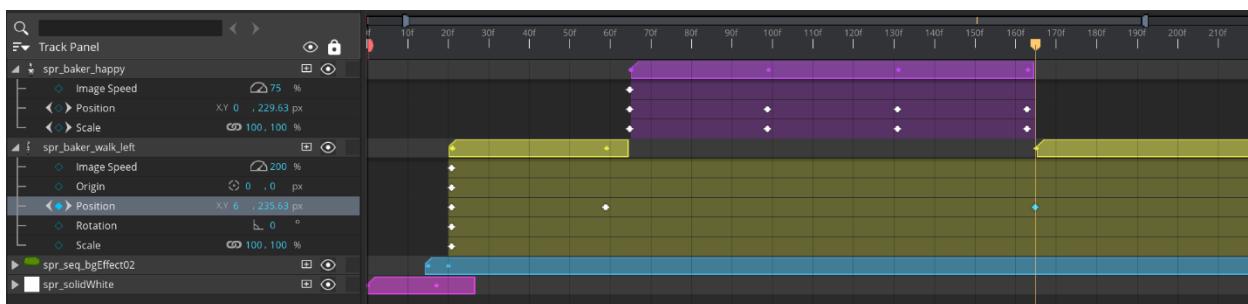


Pasting a second asset key on the same `spr_baker_walk_left` track.

By doing this, we can edit multiple asset keys for the same track — it allows us to create Sequences that don't get cluttered with too many duplicate tracks.

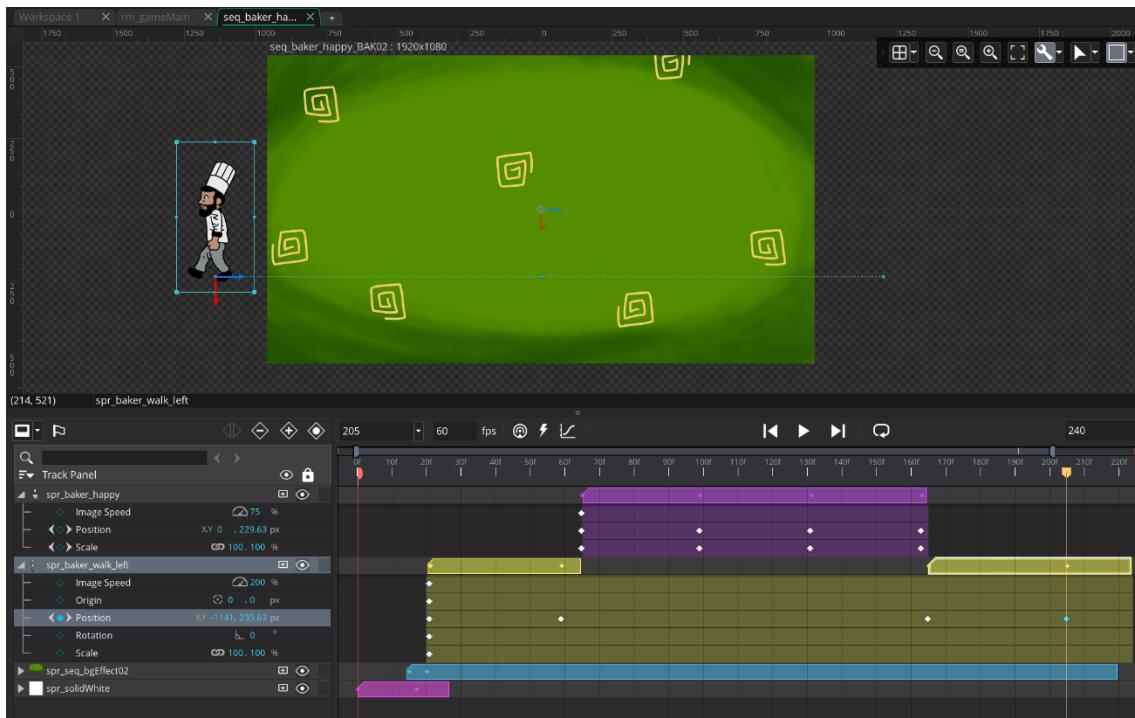
You'll notice that our walking Baker is right where we left him before he started dancing. If you click the arrow to the left of the `spr_baker_walk_left` track, you'll see the keyframes for the various parameter tracks are still in place.

With the playhead at the very first frame of the second `spr_baker_walk_left` asset key, record a new keyframe for the Position parameter.



Recording a Position keyframe for the second walking Baker asset key.

Then, either by using the Automatically record changes button, or by creating a manual keyframe and editing its value, change the Position parameter of the spr_baker_walk_left track and have the Baker walk off the left of the Canvas, like so:



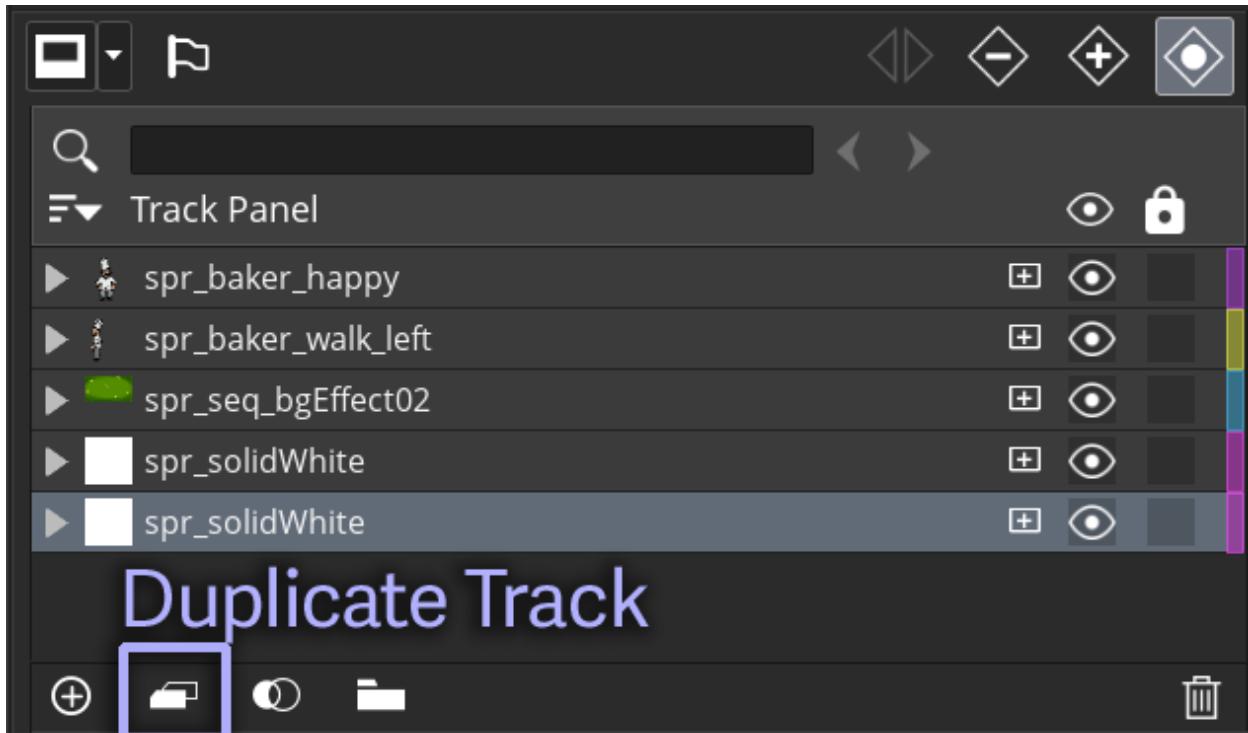
Editing our Baker so he can make his exit.

Now that our Baker makes his graceful exit, we need that final fade effect to complete the Sequence.

8.14 Copying and pasting elements for quick editing

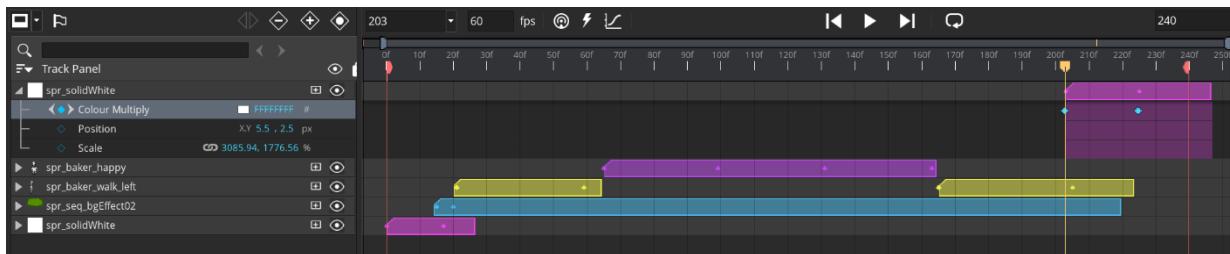
As you get more familiar with the Sequences tools, you'll find you can copy, paste and edit tracks, keyframes and more to make your work go more quickly.

Let's try this now by first duplicating an entire track. Click on the original spr_solidWhite track in the Track Panel. You can either press CTRL-D in Windows (CMD-D on Mac) or click the Duplicate Track (복사) button at the bottom of the Track Panel to make a copy.



Duplicating the spr_solidWhite track in the Track Panel.

Drag this duplicated track to the top of the Track Panel. Move its asset key along the timeline so that it begins just before the Baker walks off the Canvas. (Remember, you need to select the asset key *and* its keyframes to move everything together.



Moving the duplicated spr_solidWhite asset key and its keyframes.

Right now, our duplicated spr_solidWhite does the same thing the original track did; its Alpha channel transitions from 0 to 255. This is what we want, but we need to add a reverse fading-out effect as well.

In the Dope Sheet, extend the length of the spr_solidWhite asset key to give yourself more room to work.

Then, select the right-most keyframe on the Colour Multiply parameter track. Press CTRL-C (CMD-C on Mac) to copy this keyframe.

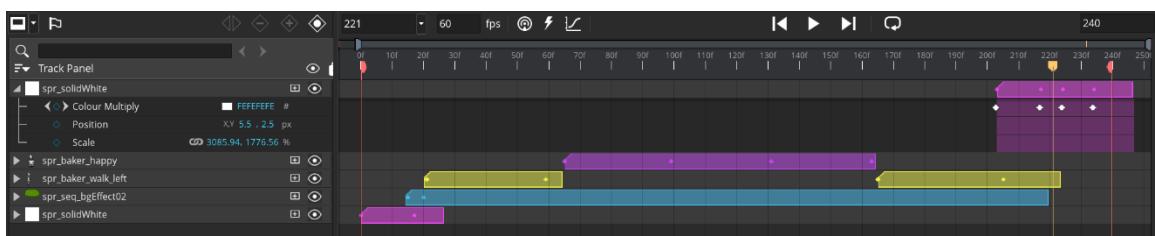
Move the playhead a few frames to the right and Paste the keyframe you copied. This will allow for a brief pause before we fade the white solid out again.

Next, copy the first keyframe in the Colour Multiply parameter track for this second spr_solidWhite.

Move the playhead again a little farther down the timeline and paste the keyframe at the new position on the Colour Multiply parameter track. Since this copied keyframe has the Alpha channel set to 0 (transparent), it will create a fade-out effect.

One last thing: we want this white solid to fade in to cover the entire Sequence. Then we want it to fade back out to reveal the gameplay (town) again. This means that any other tracks in the Sequence can't be visible when the white solid fades out.

Make sure the key asset for the spr_seq_bgEffect02 track ends before the white solid fades out, like so:



Copying keyframes to make editing parameter tracks easier. Note how the spr_seq_bgEffect02 key asset terminates at the point on the timeline so it is not present when the second white solid fades out again.

With this done, feel free to preview your Sequence again with the Play button. Adjust any key assets or keyframes to make your transitions shorter or longer, and to your liking.

	<p>Tip: If you set the playback mode of a Sequence to loop (by pressing the button in the Sequence Tools area above the Dope Sheet), it will automatically restart at the end of the Sequence. We don't want our cutscenes to loop, so ensure you have not done this.</p>
--	---

And with that, the animation for our first Sequence is complete!

8.15 Adding music to our Sequence

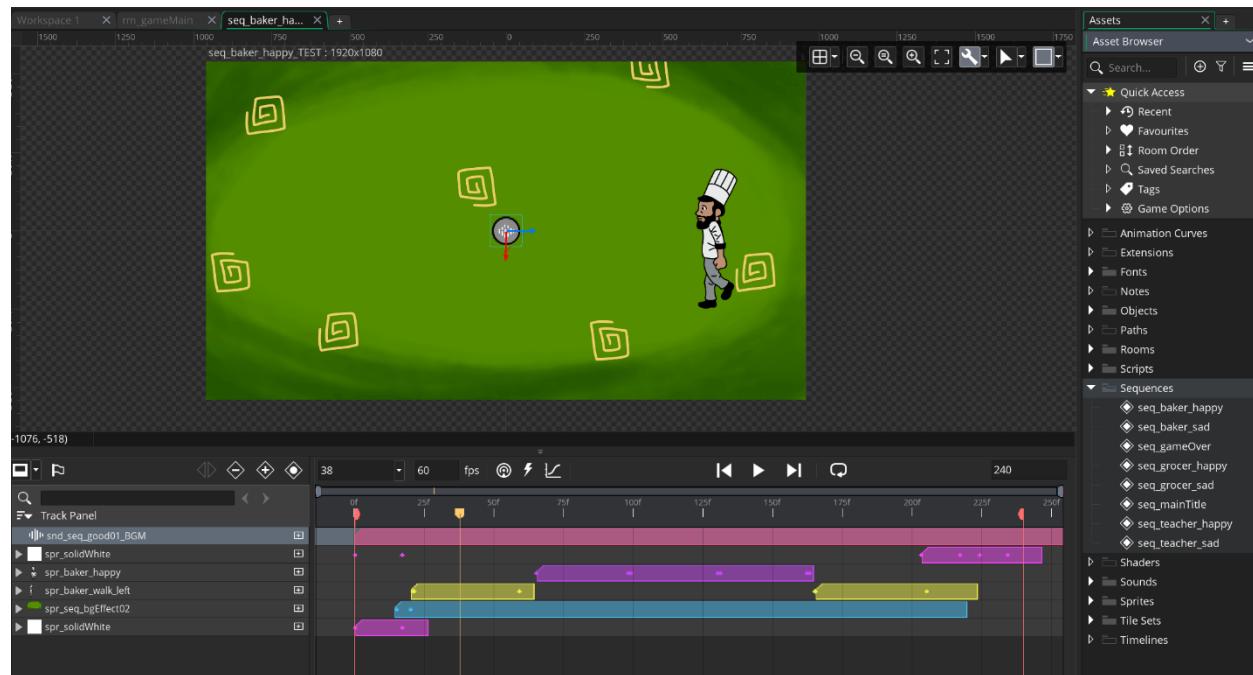
With the animation done, it's time to add some sound.

When we imported our Sound assets (back in [Sound and music](#)), we included six pieces of music specifically for our Sequences. These are the files that begin with `snd_seq` (such as `snd_seq_bad01`, `snd_seq_good01` and so on).

Choose one of the `snd_seq_good` files to use for our Baker Sequence. (You can double-click on a Sound in the Asset Browser to open it in the Sound Editor and preview it.)

Drag the music asset you've chosen onto the Track Panel in the Sequence Editor.

You can position the asset key for this new track however you like; in our example, we placed it right at the beginning of the Sequence.



Adding `snd_seq_good01_BGM` as a music track in the Baker Sequence.

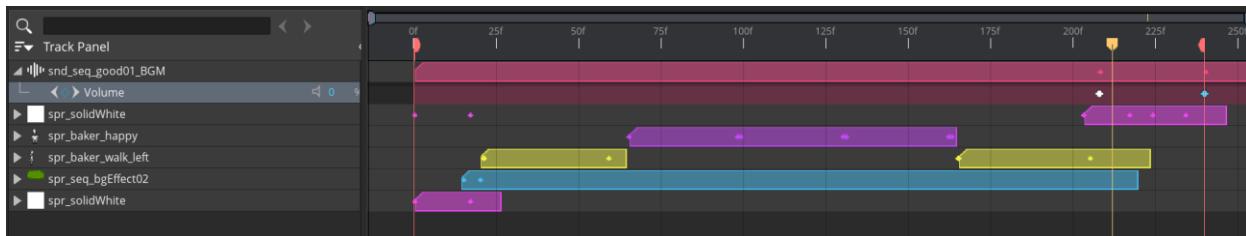
Sound asset tracks have parameter tracks unique to them (Pitch and Volume).

Click the Add parameter track button and add a Volume parameter track to the music.

Next, create a keyframe on this new Volume parameter track near or at the position where the Baker walks off the left of the scene.

Create a second keyframe on the Volume parameter track at the end of the Sequence, where second spr_solidWhite track has faded out.

Highlight this keyframe and change the value in the Volume parameter track to 0. If you scrub the yellow playhead across the timeline, you'll see that the volume of the snd_seq_good01_BGM track now fades from 100 to 0.



Fading out the volume of the music track by using keyframes on the Volume parameter track.

If you chose to start your music track at a different point, or you want to fade the music in at the beginning of the Sequence (as well as fade it out at the end), you can do so now.

Simply repeat the steps above to add two more keyframes to the music's parameter track and change the Volume from 0 to 100.

You can preview your Sequence by pressing the Play button. When you're satisfied with how your Sequence looks and sounds, it's time to move on.

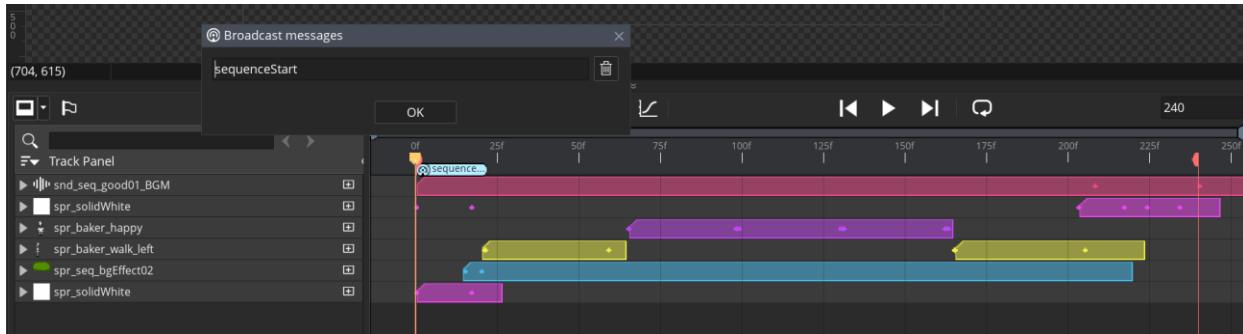
8.16 Broadcast Messages

To complete our Baker's "happy" Sequence, we need to add something brand new: a pair of *Broadcast Messages*.

Broadcast Messages are a way for Sequences within GameMaker Studio 2 to talk to the rest of your game by doing exactly what the name suggests: sending out a message.

Let's add our first Broadcast Message. Position the yellow playhead on the very first frame of the Sequence.

Then, click the Add Broadcast Messages button (@) at the top of the Dope Sheet. Enter this message as sequenceStart.

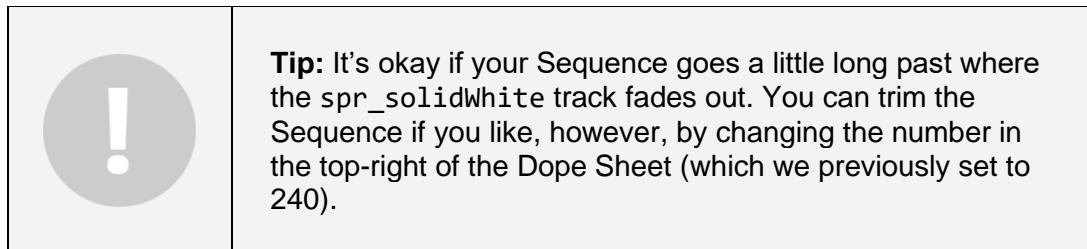


Adding the sequenceStart Broadcast Message to our Sequence.

Like with everything else in the Dope Sheet, this Broadcast Message will be sent out when the Sequence plays and reaches its position in the timeline. If you need to adjust the position of the Broadcast Message, just click and drag it. If you need to edit it, double-click it.

Next, move the playhead to the end of your Sequence, a frame or two after the top spr_solidWhite track fades out, but *before* the final frame (in our case, frame 240). If you need to adjust the spr_solidWhite asset key to make some room, do so.

Add another Broadcast Message here and enter it as sequenceEnd.



By themselves, these Broadcast Messages don't "do" anything — but as we'll see very soon, they'll become a powerful tool in letting us control and monitor our fun little animations.

8.17 Creating our “sad Baker” Sequence

Once you've got the Baker's "happy" Sequence done, it's time to make a second one. We'll use for when the player gives him an item about which he's not too thrilled.

In the Asset Browser, right-click on seq_baker_happy and choose Duplicate. Rename the duplicated Sequence seq_baker_sad.

Open this new Sequence in the Sequence Editor. We need to replace the animation we created here with an all-new one using different assets. But since we already set some parameters (like

the FPS, length and adding the two Broadcast Messages), working from this duplicate will save us some time.

Keep the spr_solidWhite tracks you used to create the fade in and fade out transitions.

Click on each of the other tracks for the various Baker Sprites and emotes you used to design your “happy” Sequence and press Delete (or right-click and choose Delete) to remove them.

Design a new Sequence to show the Baker being upset, using the following from the Asset Browser:

- spr_baker_sad or spr_baker_angry (or both!)
- any emote Sprites you think are appropriate (spr_emote_XX)
- one of the other spr_seq_bgEffect Sprites
- one of the snd_seq_bad Sound assets

If you change the length of this new Sequence, remember to move the sequenceEnd Broadcast Message to match. You can click the Broadcast Message at the top of the dope sheet and drag it left and right.

Don’t forget to change the music here too; position whichever snd_seq_bad Sound asset you chose however you like and press the Play button to preview your Sequence.

Remember to fade this new “bad” music out at the end of the Sequence (as we just did in [Adding music to our Sequence](#)).

To remove the original “good” music, click on the track for the Sound asset and press the Delete key (or right-click and choose Delete).



Creating a second Baker Sequence using a different background, Baker Sprite and other visual effects.

Once you're happy with this second Sequence, it's time to move on to the other two characters.

8.18 Create the Sequences for the Teacher

Now the Baker has his two dramatic Sequences complete, we need to do the same thing for the Teacher.

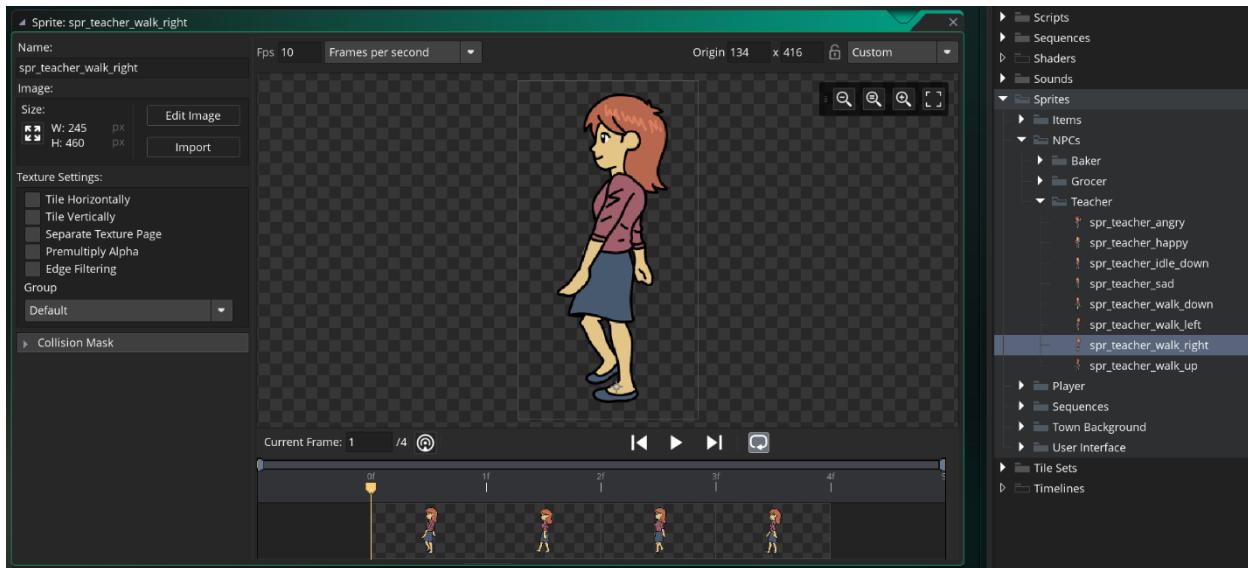
To help you with this, there are Teacher-specific assets. Using File Explorer (Windows) or Finder (Mac), navigate to the Assets folder provided with this course. Open the Sprites > Characters and Items folder and drag these Sprites into the Asset Browser in GameMaker Studio 2:

- spr_teacher_angry_strip04
- spr_teacher_happy_strip03
- spr_teacher_sad_strip04
- spr_teacher_walk_down_strip04

- spr_teacher_walk_left_strip04
- spr_teacher_walk_right_strip04
- spr_teacher_walk_up_strip04

As we just did with the Baker Sprites, open each of these in the Sprite Editor and do the following:

- Make sure each Sprite Strip has been correctly converted to frames; if not, follow the steps we used before in [Converting a Sprite Strip manually](#) to do so
- Set the FPS value to 10
- Set the origin for each to between the Teacher's feet (you can use spr_teacher_idle_down, which we previously set up, as reference)
- Remove the _stripXX suffix from the asset
- Organize the assets in the Sprites > NPCs > Teacher group



Editing the new Teacher Sprites and organizing them in the Asset Browser.

Once you have the Teacher assets organized, it's time to return to the Sequence Editor. Using everything you learned in this Session, create two more Sequences for the Teacher:

- seq_teacher_happy
- seq_teacher_sad

Again, the content, animation, duration, and timing and music choice are completely up to you; try to be creative and explore the possibilities of the Sequence Editor!

Important: make sure your two Teacher Sequences have the sequenceStart and sequenceEnd Broadcast Messages!



Tip: Remember that you can duplicate the Baker Sequences and edit the duplicates instead of starting from scratch.



Creating Sequences for our Teacher.

8.19 Create the Sequences for the Grocer

With both the Baker and Teacher complete, we can add the two Sequences for the Grocer as well.

As before, there are assets for you to use here. Using File Explorer (Windows) or Finder (Mac), navigate to the Assets folder provided with this course. Open the Sprites > Characters and Items folder and drag these Sprites into the Asset Browser in GameMaker Studio 2:

- spr_grocer_angry_strip04
- spr_grocer_happy_strip04
- spr_grocer_sad_strip04
- spr_grocer_walk_down_strip04
- spr_grocer_walk_left_strip04
- spr_grocer_walk_right_strip04
- spr_grocer_walk_up_strip04

And once again, open each of these Sprites in the Sprite Editor and do the following:

- Make sure each Sprite Strip has been correctly converted to frames; if not, follow the steps we used before in [Converting a Sprite Strip manually](#) to do so
- Set the FPS value to 10
- Set the origin for each to between the Grocer's feet (you can use spr_grocer_idle_down, which we previously set up, as reference)
- Remove the _stripXX suffix from the asset
- Organize the assets in the Sprites > NPCs > Grocer group



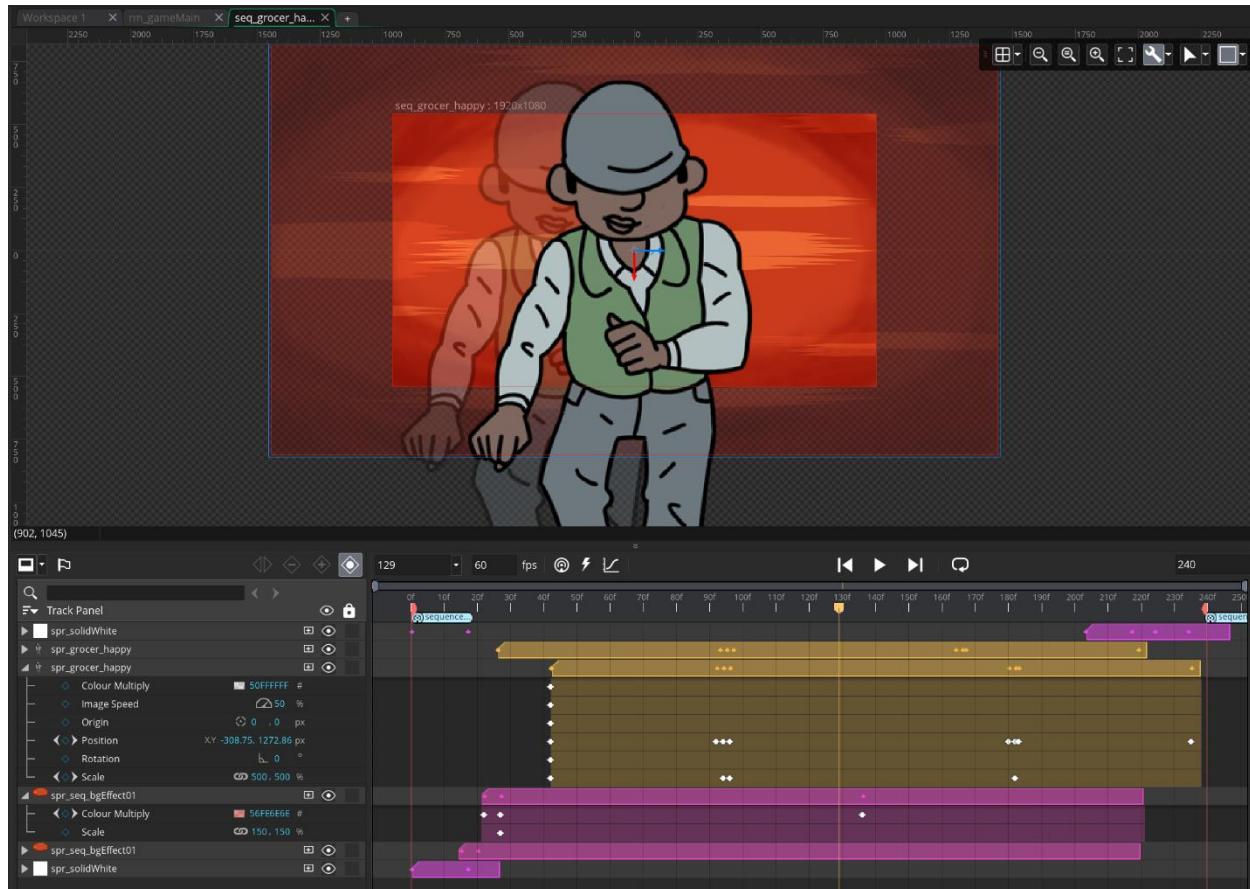
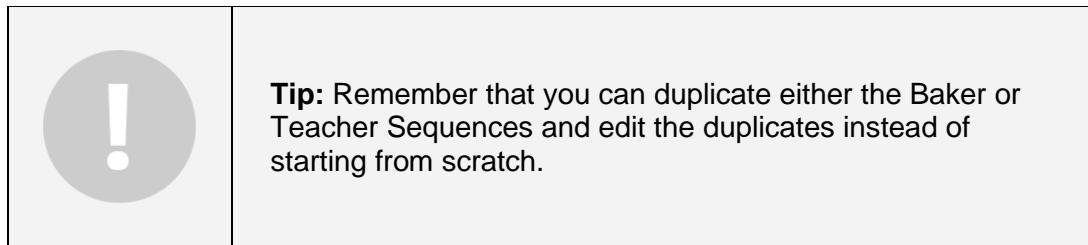
Editing the new Grocer Sprites and organizing them in the Asset Browser.

Once you have the Grocer assets organized, it's time to return to the Sequence Editor. Using everything you learned in this Session, create two more Sequences for the Grocer:

- seq_grocer_happy
- seq_grocer_sad

Again, the content, animation, duration, music choice and timing are completely up to you; try to be creative and explore the possibilities of the Sequence Editor!

Important: make sure your two Grocer Sequences have the sequenceStart and sequenceEnd Broadcast Messages!



Creating Sequences for the Grocer.

8.20 Preparing to control a Sequence

Right now, the two Sequences we just created don't appear in our game, so we need to implement them.

For this next stage, we're going to do the following:

- Allow the player to start playing a Sequence (first with a simple keystroke)
- Monitor for when a Sequence is playing and when it is finished playing
- Remove control from the player when a Sequence is playing (just like we did when a textbox is up)
- Restore control to the player once a Sequence is finished playing

This seems like an awful lot, but we'll start at the top and do this step-by-step.

8.21 Expanding the control object with Sequences

Our Sequences are, in effect, linear animations. In order to track what's going on with them and deal with some of the functionality listed above, we're going to use an Object to pay attention to things and take action.

Luckily, we already have one: `obj_control`. So, double-click `obj_control` in the Asset Browser to open it. Then open its Game Start Event.

In the Game Start Event, add the following code after the `// Item states` code block:

```
// Sequence states
enum seqState {
    notPlaying,
    waiting,
    playing,
    finished,
}
// Sequence variables
sequenceState = seqState.notPlaying;
curSeqLayer = noone;
curSeq = noone;
```

You should recognize that first batch; it's another `enum`, this time for tracking the state of our Sequences. Since we'll only ever play one of these Sequence animations at a time, we can get by with this simple set.

Below that you can see we're setting sequenceState to seqState.notPlaying and initializing two more variables that we'll use soon enough.

Next, open obj_control's Room Start Event, which we used before to play music in the town. Add the following code block after the // Play music based on Room code block:

```
// Mark Sequences layer
if (layer_exists("Cutscenes")) {
    curSeqLayer = "Cutscenes";
}
else {
    curSeqLayer = "Instances";
}
```

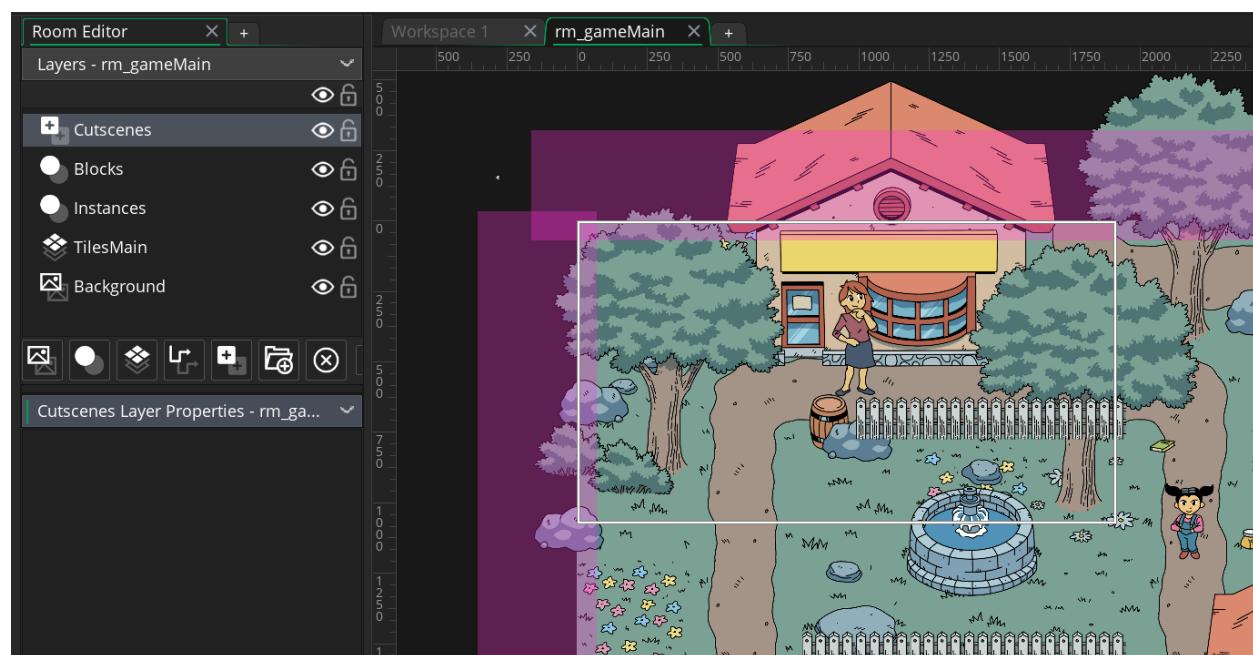
As you'll soon see, we'll need to pay attention to which Room layer our Sequences exist on and where they should be created (just like we do when we create instances).

There's one wrinkle, though. We need to make sure this layer (Cutscenes) exists within our town Room. So, open rm_gameMain in the Asset Browser and look at the Room Editor.

At the bottom of the Room Editor panel, you'll see icons for creating different kinds of layers.

Click the Create new Asset layer button (✚) to create a new *Asset Layer*.

Once the new layer appears in the Layers panel, right-click it and choose Rename to name it Cutscenes. Drag the new layer to the top of the stack.



Creating a new Asset Layer (“Cutscenes”) in our town room.

We won't place anything on this layer, but we will target it when displaying our Sequences.

8.22 Testing our Sequence

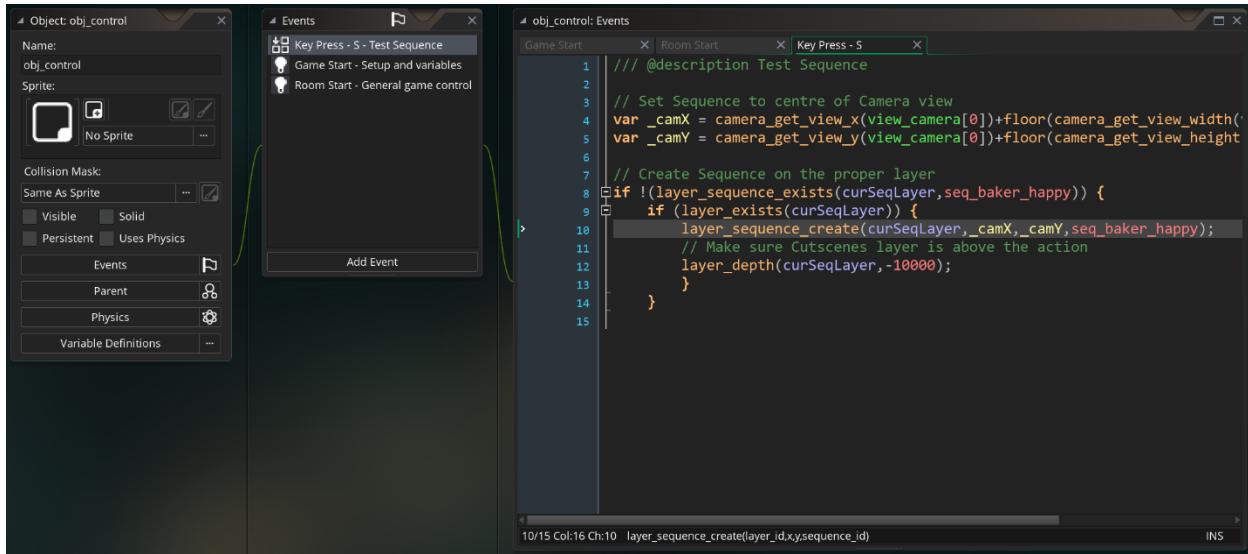
Before we go on, let's make sure everything we've set up so far is working and preview our Sequence within the game itself.

Open `obj_control` again and in the Object Editor, click Add Event. Choose Key Press > Letters > S.

Open this new Event and enter the following code block:

```
// Set Sequence to centre of Camera view
var _camX =
camera_get_view_x(view_camera[0])+floor(camera_get_view_width(view_camera[0])*0.5)
;
var _camY =
camera_get_view_y(view_camera[0])+floor(camera_get_view_height(view_camera[0])*0.5
);

// Create Sequence on the proper layer
if !(layer_sequence_exists(curSeqLayer,seq_baker_happy)) {
    if (layer_exists(curSeqLayer)) {
        layer_sequence_create(curSeqLayer,_camX,_camY,seq_baker_happy);
        // Make sure Cutscenes layer is above the action
        layer_depth(curSeqLayer,-10000);
    }
}
```

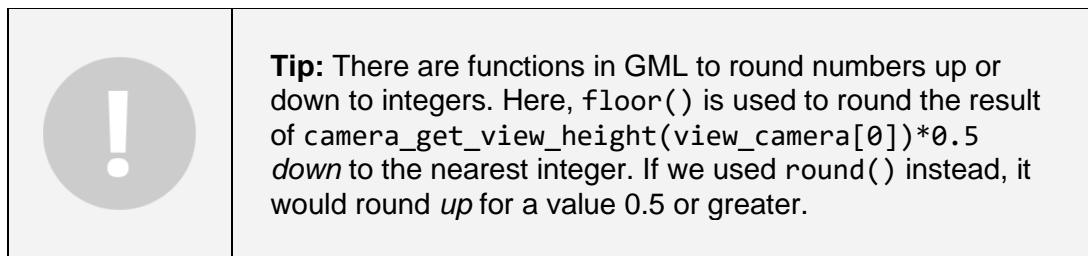


Adding a Key Press – S Event to obj_control.

Note the `_camX` and `_camY` variables we've set up here. When we created our `seq_baker_happy` Sequence (and all the others), we made sure the Origin X and Origin Y were both 0. But this actually corresponds to the middle-centre of the Sequence canvas (and not the top-left, as you might expect).

The `_camX` calculation finds the x position of our game's Camera and adds half of the Camera's width, which would be the horizontally middle point of our Camera view, as we see it when playing the game.

The `_camY` calculation does the same thing, but for the height. But calculating both, we get the middle-centre point of our game's Camera view and make sure our Sequence aligns with it when it is created.



Creating a Sequence here is just like creating an Instance; we're targeting a layer, x and y value and choosing which Sequence to pop in.

The final function here (`layer_depth()`) is a bit of housekeeping. We just need to make sure our Cutscenes asset layer is always lower (closer to the Camera) than the rest of the action, so our Sequences appear above everything else. (We did the same thing for our textbox object.)

But remember that our `obj_player`, NPCs and town items all use that depth-sorting code (`depth = -y`), which means we can't be 100% sure which depth is "closest" all the time.

So, we're manually setting a really low value here just to play it safe. (We stored Cutscenes in this `curSeqLayer` in `obj_control`'s Room Start Event.)

Okay, with this test Event in place, save your project. Run the game and press the "S" key on your keyboard once it's running. You should see your beautiful Baker Sequence appear and play overtop everything!



Testing our Baker Sequence in our game by pressing the "S" key.

8.23 Using Broadcast Messages to listen to Sequences

Now that we know our Sequence works and we've had some practice creating an instance of a Sequence, it's time to start setting up our game's actual functionality.

When we made both the `seq_baker_happy` and `seq_baker_sad` Sequences, we added two Broadcast Messages (`sequenceStart` and `sequenceEnd`). Now it's time to put these to use.

Whenever, say, the seq_baker_happy Sequence plays in our game, it sends out those Broadcast Messages as the animation reaches the appropriate points on its timeline. So, at the start, it silently yells out “sequenceStart” and as it reaches the end, it yells out “sequenceEnd.”

What we need to do is tell obj_control to listen for those messages and do something when it hears them.

Open obj_control again in the Object Editor. Click Add Event and choose Other > Broadcast Message.

In this new Broadcast Messages Event, input the following code block:

```
// Listen for Broadcast Messages
switch (event_data[? "message"]) {
    case "sequenceStart": {
        // Set our state
        sequenceState = seqState.playing;
        // Find out which Sequence just broadcast this message and mark it
        if (layer_get_element_type(event_data[? "element_id"]) ==
layerElementType_sequence) {
            curSeq = event_data[? "element_id"];
            show_debug_message("obj_control has heard that Sequence "+string(curSeq)+" is playing");
        }
    }; break;
    case "sequenceEnd": {
        // Set our state
        sequenceState = seqState.finished;
        show_debug_message("obj_control has heard that Sequence "+string(curSeq)+" has ended");
    }; break;
}
```

This may look complex, but it's mostly code we've used before. We're using a special event_data accessor to look out for Broadcast Messages (which GameMaker Studio 2 understands as “message”) and we're doing so in a switch function.

The screenshot shows the GameMaker Studio 2 code editor with a script titled "Broadcast Message". The code is as follows:

```
< obj_control: Broadcast Message >
Broadcast Message X
>
1 // @description Listen for messages from Sequences
2
3 // Listen for Broadcast Messages
4 switch (event_data[? "message"]) {
5     case "sequenceStart": {
6         // Set our state
7         sequenceState = seqState.playing;
8         // Find out which Sequence just broadcast this message and mark it
9         if (layer_get_element_type(event_data[? "element_id"]) == layer_elementtype_sequence) {
10            curSeq = event_data[? "element_id"];
11            show_debug_message("obj_control has heard that Sequence "+string(curSeq)+" is playing");
12        }
13    }; break;
14    case "sequenceEnd": {
15        // Set our state
16        sequenceState = seqState.finished;
17        show_debug_message("obj_control has heard that Sequence "+string(curSeq)+" has ended");
18    }; break;
19 }
```

Creating a new Broadcast Messages Event in obj_control.

There are two cases in this switch function; one for each of the Broadcast Messages we included in our Baker Sequences.

For sequenceStart, we're doing two things:

1. Changing the sequenceState (using those enums we just created in obj_control's Game Start Event)
2. Marking which Sequence is playing and storing that in the variable curSeq, so we can use it later

For sequenceEnd, we're doing one thing:

1. Changing the sequenceState again

We've included two debug messages here so we can test that this is working. Run your game again and press the "S" key to initiate the Sequence as before.

Pay attention to the Output window in GameMaker Studio 2. (If you don't see it, you can go to Windows > Output in the toolbar to open it again). You should see the debug messages we included here, even though nothing new visually is happening within our game. (The Sequence will be indicated by an ID number and not a name.)

The left screenshot shows the Output window with several lines of text: "obj_player hasn't found anything", "obj_control has heard that Sequence 135 is playing", "obj_player hasn't found anything", "obj_player hasn't found anything", and "obj_player hasn't found anything". The right screenshot shows similar messages, but includes an additional line: "obj_control has heard that Sequence 135 has ended".

Reviewing the Output window and finding the two debug messages that confirms our new obj_control Events are working.

When you're satisfied, return to GameMaker Studio 2 and let's continue.

8.24 Using states to control Sequences

Now that obj_control is correctly hearing what our Sequence is saying, we can enact some control.

Open obj_control if it's not opened already and click Add Event. Choose Step > Step.

Open this new Step Event and input the following code block:

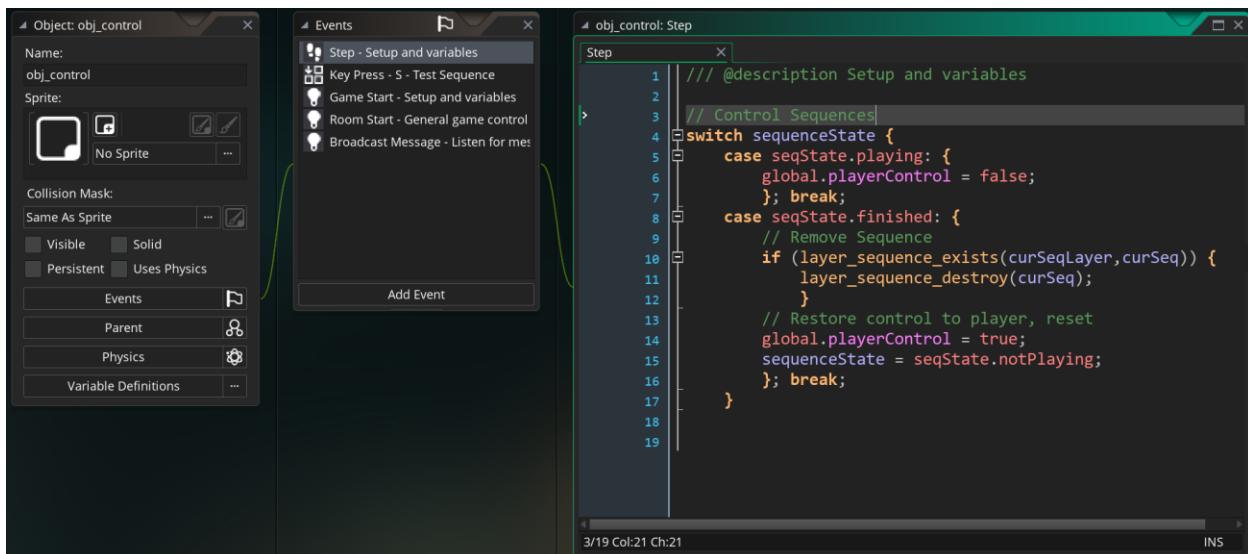
```
// Control Sequences
switch sequenceState {
    case seqState.playing: {
        global.playerControl = false;
    }; break;
    case seqState.finished: {
        // Remove Sequence
        if (layer_sequence_exists(curSeqLayer, curSeq)) {
            layer_sequence_destroy(curSeq);
        }
        // Restore control to player, reset
        global.playerControl = true;
        sequenceState = seqState.notPlaying;
        curSeq = noone;
    }; break;
}
```

This is yet another switch function, with cases for the two states we were setting in the Broadcast Messages Event.

For `seqState.playing`, we're simply removing control from the player, so they can't move around or do anything while the Baker cutscene plays.

For `seqState.finished`, we're doing four things:

1. Destroying the instance of the Sequence we created. This is why we stored the ID of the Sequence we created in `curSeq` in `obj_control`'s Broadcast Messages Event. By default, a Sequence that's created won't be destroyed automatically, so we have to do it to prevent memory or performance issues down the line.
2. Returning control to the player
3. Resetting the Sequence state to `seqState.notPlaying`
4. Resetting `curSeq` to noone



Adding a Step Event to obj_control.

Save your project and run the game again.

Press “S” on your keyboard to test the Sequence and then feel free to mash the arrow keys on your keyboard to move the player Object around. You shouldn't be able to move or do anything else until after the Sequence plays and it is destroyed.

And with that, we've accomplished everything we set out to achieve at the beginning of [Preparing to control a Sequence](#).

8.25 Controlling town audio when Sequences are playing

With `obj_control` correctly tracking when Sequences are being played, we can address another issue you may noticed. Currently, even if a Sequence plays, we can still hear the other audio in our game (the town music, its ambience, and even the fountain).

If a Sequence is playing, we only want to hear the audio from that Sequence, but unless we specify what to do in these situations, our game won't automatically know that.

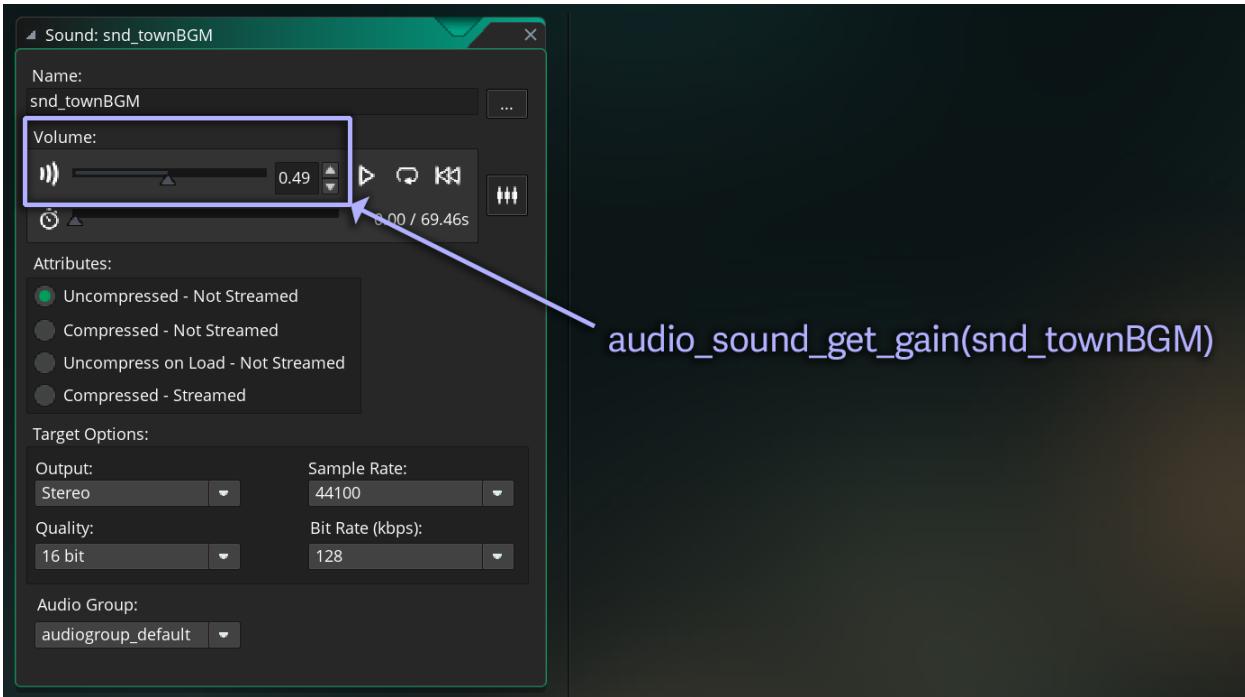
So, open `obj_control` and its Game Start Event.

Add two new lines to the `// Game variables` code block, like so:

```
// Game variables
global.playerControl = true;
global.gameOver = false;
global.gameStart = false;
townBGMvolume = audio_sound_get_gain(snd_townBGM);
townAmbienceVolume = audio_sound_get_gain(snd_townAmbience);
```

These two new lines are going to help us keep track of the initial volumes (known in GameMaker Studio 2 as *gain*) for both the music and ambient sound loop.

Since you might have adjusted the volume (*gain*) of these Sound assets in the Sound Editor, we first want to capture what that adjustment is. We use `audio_sound_get_gain()` to do this.



Using `audio_sound_get_gain()` to capture the volume of a Sound asset, as defined in the Sound Editor.

Next, open `obj_control`'s Step Event.

In the `sequenceState` switch statement under the `// Control Sequences` code block, update the `seqState.playing` case like so:

```
// Control Sequences
switch sequenceState {
    case seqState.playing: {
        // Fade out town music
        if (audio_is_playing(snd_townBGM)) {
            audio_sound_gain(snd_townBGM,0,60);
        }
        // Fade out town ambience
        if (audio_is_playing(snd_townAmbience)) {
            audio_sound_gain(snd_townAmbience,0,60);
        }
        global.playerControl = false;
    }; break;
```

Here we're using another function, `audio_sound_gain`, to set the volume of our two Sound assets to 0.

The third argument (60) in these functions is the number of steps we want to use to make this change. If the number is 0, the volume change will happen instantly; if it's any higher, GameMakerStudio 2 will gradually change the Sound asset's volume, creating a fade effect. You can change this number to make the sound fade out more quickly or more slowly.

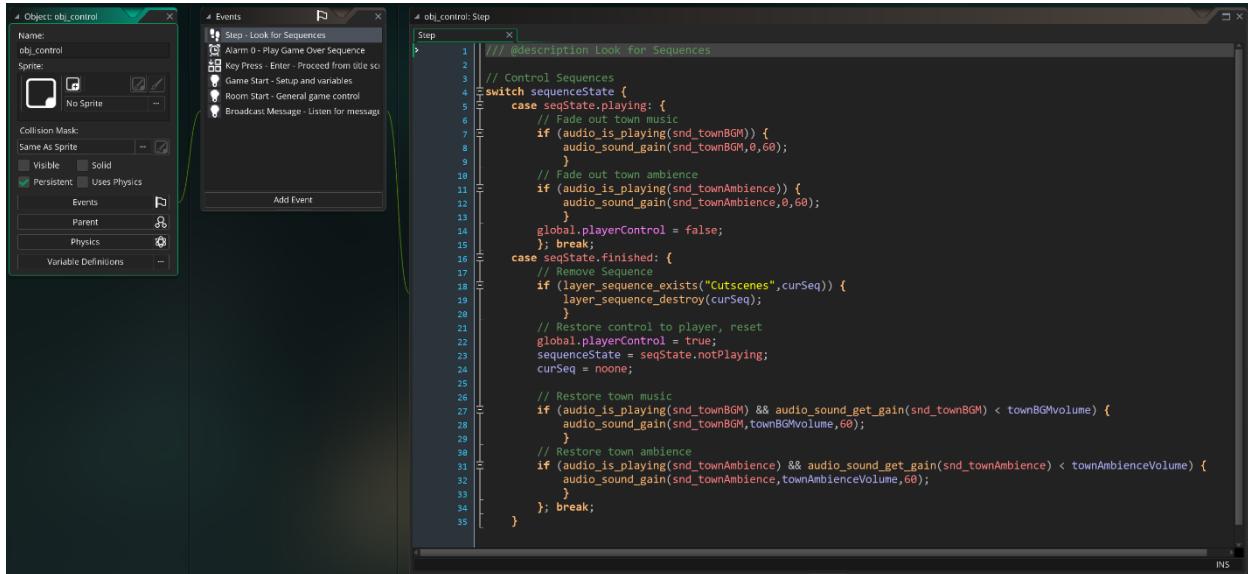
Next, update the seqState.finished case with new code, like so:

```
// Control Sequences
switch sequenceState {
    case seqState.playing: {
        // Fade out town music
        if (audio_is_playing(snd_townBGM)) {
            audio_sound_gain(snd_townBGM,0,60);
        }
        // Fade out town ambience
        if (audio_is_playing(snd_townAmbience)) {
            audio_sound_gain(snd_townAmbience,0,60);
        }
        global.playerControl = false;
    }; break;
    case seqState.finished: {
        // Remove Sequence
        if (layer_sequence_exists(curSeqLayer,curSeq)) {
            layer_sequence_destroy(curSeq);
        }
        // Restore control to player, reset
        global.playerControl = true;
        sequenceState = seqState.notPlaying;
        curSeq = noone;

        // Restore town music
        if (audio_is_playing(snd_townBGM) && audio_sound_get_gain(snd_townBGM) < townBGMvolume) {
            audio_sound_gain(snd_townBGM,townBGMvolume,60);
        }
        // Restore town ambience
        if (audio_is_playing(snd_townAmbience) &&
audio_sound_get_gain(snd_townAmbience) < townAmbienceVolume) {
            audio_sound_gain(snd_townAmbience,townAmbienceVolume,60);
        }
    }; break;
}
```

Here, we're using those new variables we declared in obj_control's Game Start Event to restore the volume (gain) of both the music and ambience Sound assets once a Sequence has been marked as finished.

We use `audio_sound_gain` again to fade the volume back up over 60 steps (or one second).



Changing audio volume in the updated Step Event for `obj_control`.

Run your game again and test it but giving an item to one of the three NPCs. You'll hear your Sequence audio come through crystal-clear, and the town music and ambience will naturally fade out. When you return to gameplay, the town audio will fade right back in.

When you're ready, close the game window and return to GameMaker Studio 2.

8.26 Adjusting 3D positional audio when a Sequence is playing

You'll have noticed when testing your game that the 3D positional audio from the gurgling town fountain could still be heard while a Sequence was playing.

Using a similar technique as what we just did, let's also make sure any environmental sounds are correctly being quieted when a Sequence is playing.

Open `obj_par_environment` and its `Create` Event.

Update the `// Create` emitter code block like so:

```
| // Emitter variables
```

```
| myEmitter = 0;  
| volInit = 1;
```

We're adding an initial variable that we'll use in the next step.

Also in the Create Event, update the // Create emitter code block like so:

```
// Create emitter  
if (useSound != noone) {  
    if !audio_is_playing(useSound) {  
        myEmitter = audio_emitter_create();  
        audio_emitter_position(myEmitter,x,y,0);  
        audio_falloff_set_model(audio_falloff_exponent_distance);  
        audio_emitter_falloff(myEmitter,fallStart,maxDistance,1);  
        volInit = audio_sound_get_gain(useSound);  
        audio_play_sound_on(myEmitter,useSound,1,1);  
    }  
}
```

Here, we're setting that new variable, volInit, again, but by getting the volume (gain) of whatever sound an environmental object has been set to play.

Next, open obj_par_envrionment's Step event.

Below the // Depth sorting code block, add a new code block:

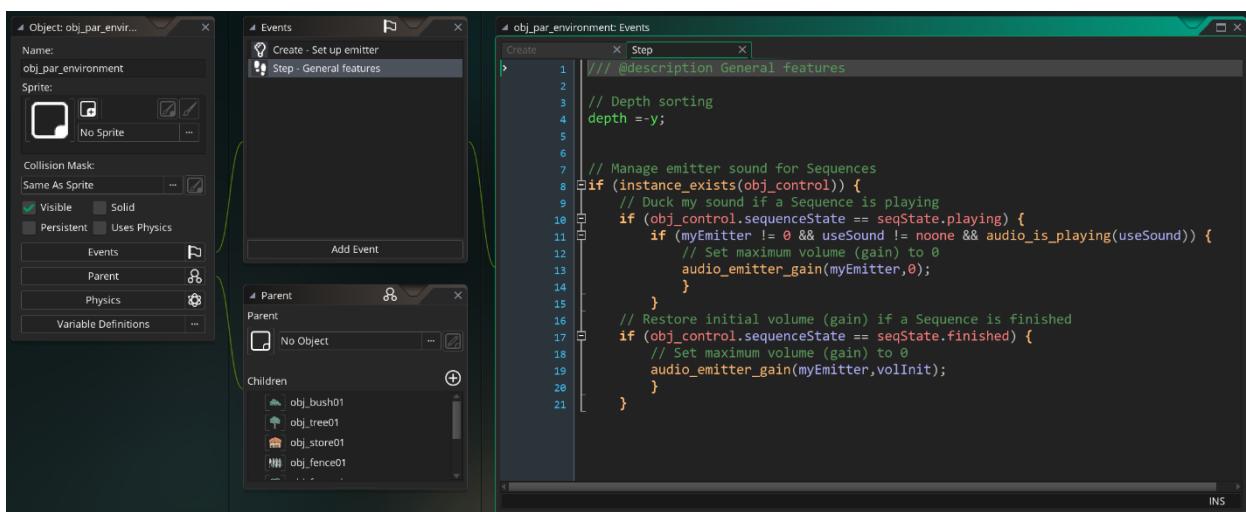
```
// Manage emitter sound for Sequences  
if (myEmitter != -1 && useSound != noone && audio_is_playing(useSound)) {  
    if (instance_exists(obj_control)) {  
        // Duck my sound if a Sequence is playing  
        if (obj_control.sequenceState == seqState.playing) {  
            // Set maximum volume (gain) to 0  
            audio_emitter_gain(myEmitter,0);  
        }  
        // Restore initial volume (gain) if a Sequence isn't playing  
        else {  
            if (audio_emitter_get_gain(myEmitter) < volInit) {  
                // Reset maximum volume (gain) back to volInit  
                audio_emitter_gain(myEmitter,volInit);  
            }  
        }  
    }  
}
```

If you read the code, you'll see that we're using `obj_control`'s states to check if a Sequence is playing or is finished, and then adjusting the volume (gain) of our environmental sound accordingly.

The wrinkle here, though, is that since 3D sound is always played via an *Emitter* (as we covered in [Adding an Audio Emitter](#)), we use `audio_emitter_gain()`. This doesn't allow us to transition the volume of a sound over time, as `audio_sound_gain()` does — that's because an emitter already does that!

Instead, `audio_emitter_gain()` sets the *maximum* volume a 3D sound can be; if it's 0, then you'll never hear it, even if the listener (our player Object) is right next to the Object that has that Emitter.

We're doing this in `obj_par_environment`'s Step Event, and not in `obj_control` for a simple reason: you could have multiple environmental Objects with their own 3D audio, so it makes sense for each one to look after itself.



Changing the maximum volume of a 3D environmental sound depending on whether a Sequence is playing.

Run the game again and test our new functionality by press the "S" key. This time, the town music, ambience *and* the fountain should fade out, and you should only hear the audio from the Sequence that plays. When the Sequence is finished, all three sounds should come back in again.

When you're satisfied, close the game window and return to GameMaker Studio 2.

8.27 End of Session notes

That's going to do it for this session. We've learned how to design Sequences, how to implement them into our game, and how to deal with controlling audio. As always, make sure to save your project!

9 Session 6

In our final session, we're going to put those Sequences to use for real. We'll learn to check which item our player is carrying when they approach each character and play the correct Sequence.

Then, we'll track if the player has correctly given all three NPCs the items they want, and if so, mark that they've completed the game.

Let's get started!

9.1 Adding our core gameplay loop

Now that we can create a Sequence, monitor it with Broadcast Messages and use some enum-enabled states to control a few things, we're ready to get a little more ambitious.

What we want to do is the following:

- Talk to an NPC while holding an item
- Have the NPC check which item we're carrying
- Pop up a textbox (which we already do) and change the text depending on the item
- Play the correct Sequence depending on the item
- If the item we're giving the NPC is the one they're looking for, remove the item from the player
- If we gave the NPC the correct item, mark the NPC as "done" to track progress

It may look like a lot, but if we proceed step-by-step, you'll see that we've already built most of the tools we need!

9.2 Changing what the NPCs say

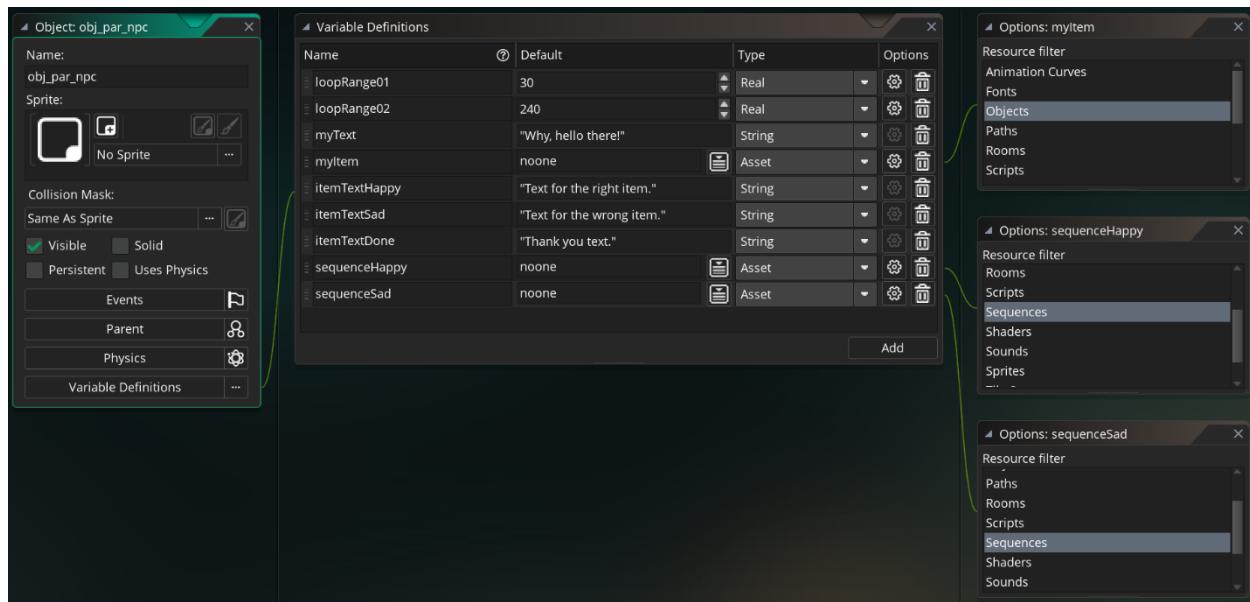
First, let's make sure our NPCs say different things to the player, depending on whether the player is carrying an item and if it's the right one.

To do this, first open `obj_par_npc` and click **Variable Definitions** in the Object Editor.

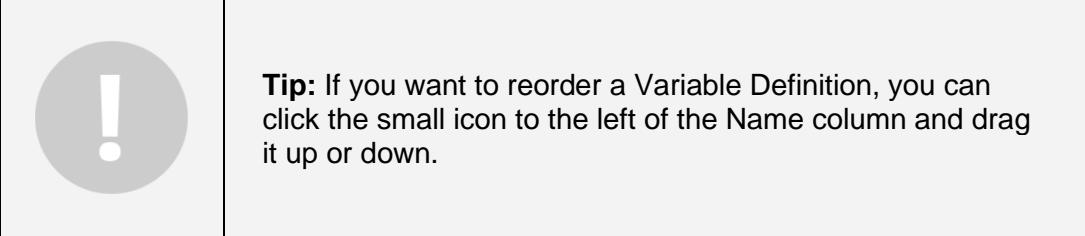
We need to add several new Variable Definitions. For each one, click the Add button and then edit each column to match what you see here:

Name	Default	Type	What is it for?
myItem	noone	Asset (Object)	Correct item the NPC is looking for
itemTextHappy	“Text for the right item.”	String	Text to display when player gives NPC correct item
itemTextSad	“Text for the wrong item.”	String	Text to display when player gives NPC any other item
itemTextDone	“Thank you text.”	String	Text to display after NPC has been marked as “done”
sequenceHappy	noone	Asset (Sequence)	Sequence to play when player gives NPC correct item
sequenceSad	noone	Asset (Sequence)	Sequence to play when player gives NPC any other item

Remember that if you make a Variable Definition an Asset, GameMaker Studio 2 will allow you to choose any Asset by default. If you want to restrict this choice to a particular kind of Asset (Object, Sequence, etc.), you need to click the gear icon for that Variable Definition and specify the Asset type.



Adding new Variable Definitions to obj_par_npc.



Tip: If you want to reorder a Variable Definition, you can click the small icon to the left of the Name column and drag it up or down.

Now that we have these presets, you can customize the content for each of the three NPCs.

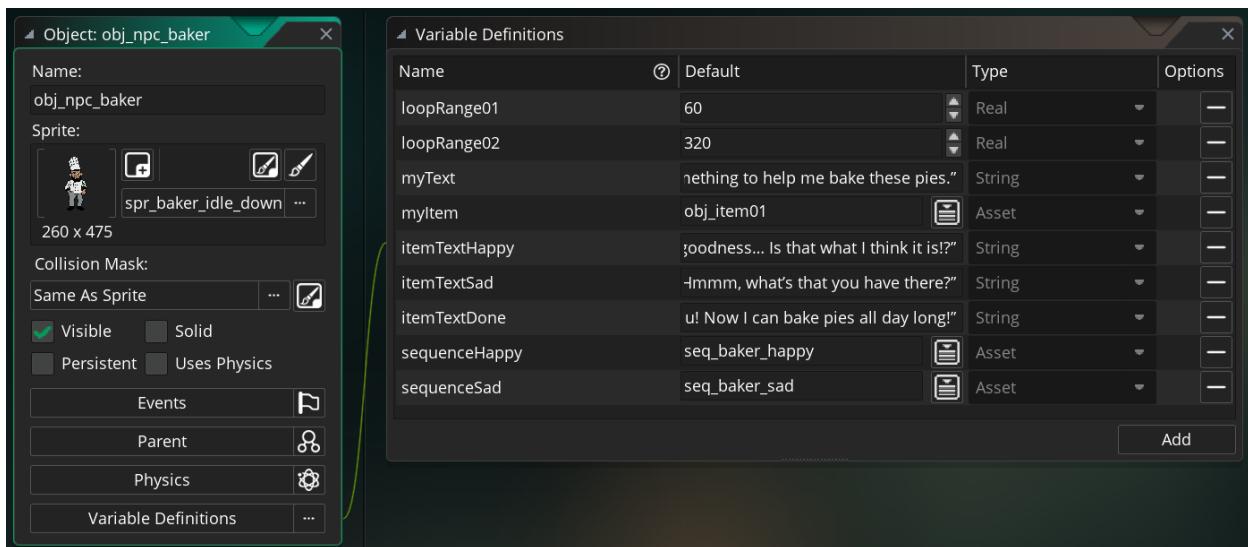
9.2.1 Updating the Baker

First, Open `obj_npc_baker`. In the Object Editor, click “Variable Definitions.”

Click the Edit icon beside each of the new Variable Definitions we just added and change the default values. You can also revise what the Baker says when the player first talks to him (`myText`), to provide a hint as to the item he wants.

Here are some suggestions for the Baker:

Name	Default
<code>myText</code>	“I sure wish I had something to help me bake these pies.”
<code>myItem</code>	<code>obj_item01</code> (the rolling pin)
<code>itemTextHappy</code>	“Oh my goodness... Is that what I think it is!?”
<code>itemTextSad</code>	“Hmmm, what’s that you have there?”
<code>itemTextDone</code>	“Thank you! Now I can bake pies all day long!”
<code>sequenceHappy</code>	<code>seq_baker_happy</code>
<code>sequenceSad</code>	<code>seq_baker_sad</code>



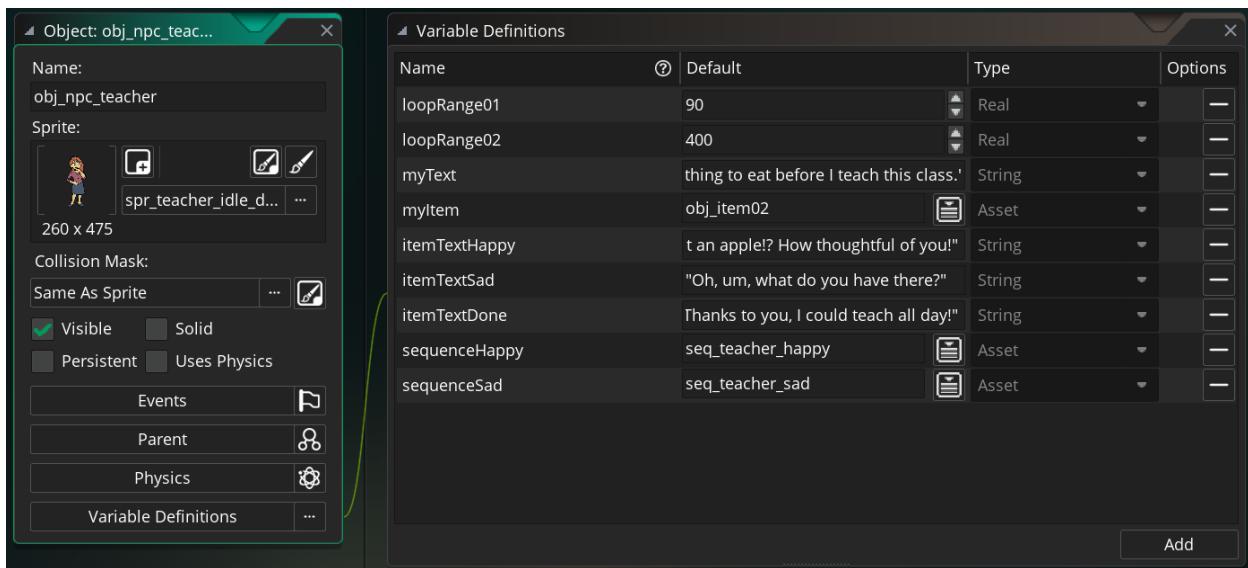
Adding custom values for new NPC Variable Definitions for the Baker.

9.2.2 Updating the Teacher

Once you're happy with the Baker, open `obj_npc_teacher` and do the same thing: edit each of the new Variable Definitions.

Make sure to choose a different object for `myItem`. Here are some recommendations:

Name	Default
<code>myText</code>	"I wish I had something to eat before I teach this class."
<code>myItem</code>	<code>obj_item02</code> (the apple)
<code>itemTextHappy</code>	"Is that an apple!? How thoughtful of you!"
<code>itemTextSad</code>	"Oh, um, what do you have there?"
<code>itemTextDone</code>	"Thanks to you, I could teach all day!"
<code>sequenceHappy</code>	<code>seq_teacher_happy</code>
<code>sequenceSad</code>	<code>seq_teacher_sad</code>



Editing the Teacher with updated Variable Definitions.

9.2.3 Updating the Grocer

Once the Teacher is complete, let's move on to the Grocer. Open obj_npc_grocer and edit its Variable Definitions.

Once again, choose a different object for `myItem`. Here are our recommendations:

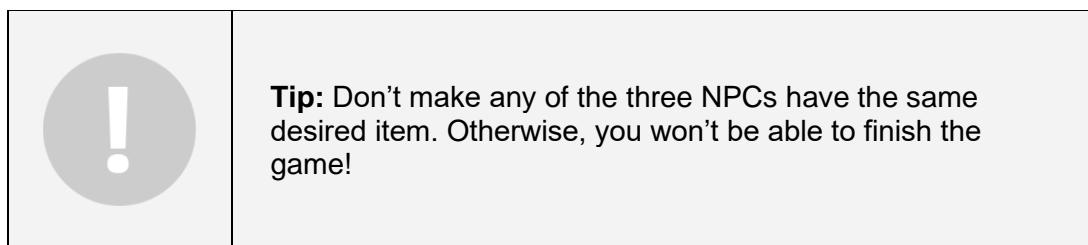
Name	Default
myText	"How am I going to keep these veggies fresh?"
myItem	obj_item06 (the watering can)
itemTextHappy	"Oh, a watering can! Of course, that's perfect!"
itemTextSad	"What's that you have there?"
itemTextDone	"With that watering can, now my veggies will be perfect!"
sequenceHappy	seq_grocer_happy
sequenceSad	seq_grocer_sad

The screenshot shows two panels side-by-side. On the left is the 'Object: obj_npc_grocer' properties panel. It includes fields for Name (obj_npc_grocer), Sprite (spr_grocer_idle_d...), Collision Mask (Same As Sprite), and various physics and visibility settings. On the right is the 'Variable Definitions' panel, which lists nine variables with their names, default values, types, and options. The variables are:

Name	Default	Type	Options
loopRange01	50	Real	
loopRange02	550	Real	
myText	I going to keep these veggies fresh?"	String	
myItem	obj_item06	Asset	
itemTextHappy	"Oh, a watering can! That's perfect!"	String	
itemTextSad	"What's that you have there?"	String	
itemTextDone	:an, now my veggies will be perfect!"	String	
sequenceHappy	seq_grocer_happy	Asset	
sequenceSad	seq_grocer_sad	Asset	

Editing the Grocer's Variable Definitions.

When all three NPC Objects have been updated, we are ready to move on.

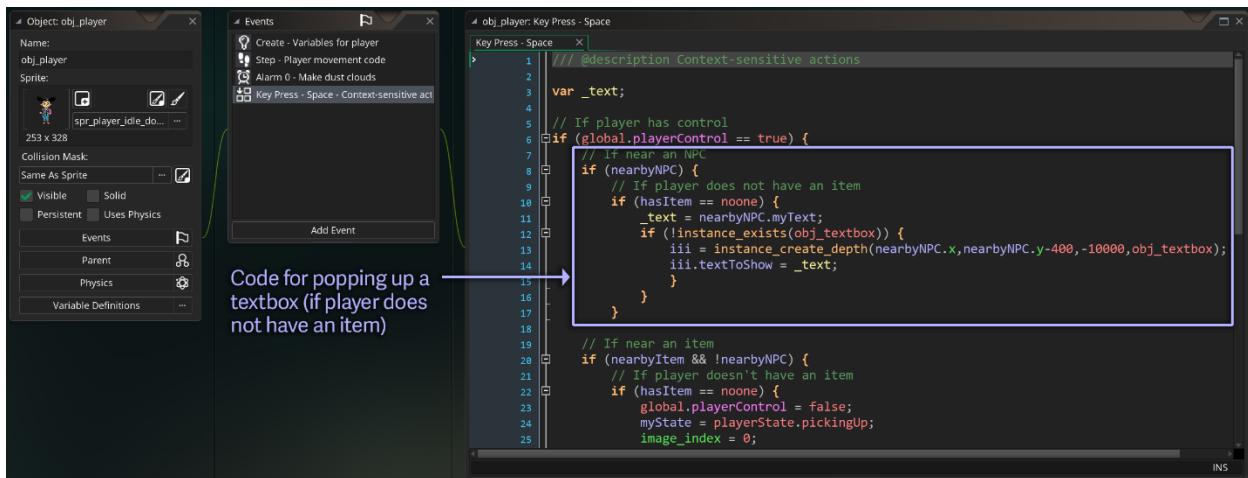


9.3 Checking for which item the Player has

If we just play our game again now, nothing will have changed. That's because we need to add the ability to check which item the player is carrying (if it's carrying anything at all) and then display the correct text.

Luckily, we thought ahead way back in [Picking up an item](#) and added an if statement in obj_player's Key Press - Space Event. This if statement made sure that a textbox would appear if we approached an NPC and pressed the Space bar, but only if we weren't carrying an item.

That code was within the // If player has control code block, like so:



Reviewing the textbox code in obj_player's Key Press – Space Event.

This gives us the ability to add *another* if statement — one for when the player *is* carrying an item.

So, update the // If near an NPC code block (which is within the // If player has control code block) like so:

```

// If near an NPC
if (nearbyNPC) {
    // If player does not have an item
    if (hasItem == noone || hasItem == undefined) {
        _text = nearbyNPC.myText;
        if (!instance_exists(obj_textbox)) {
            iii = instance_create_depth(nearbyNPC.x, nearbyNPC.y-400, -10000, obj_textbox);
            iii.textToShow = _text;
        }
    }
    // If player has item (and it still exists)
    if (hasItem != noone && instance_exists(hasItem)) {
        // If player has correct item
        if (hasItem.object_index == nearbyNPC.myItem) {
            _text = nearbyNPC.itemTextHappy;
        }
        // Or if player has incorrect item
        else {
            _text = nearbyNPC.itemTextSad;
        }
        // Create textbox
        if (!instance_exists(obj_textbox)) {

```

```

        iii = instance_create_depth(nearbyNPC.x,nearbyNPC.y-400,-
10000,obj_textbox);
        iii.textToShow = _text;
    }
}
}

```

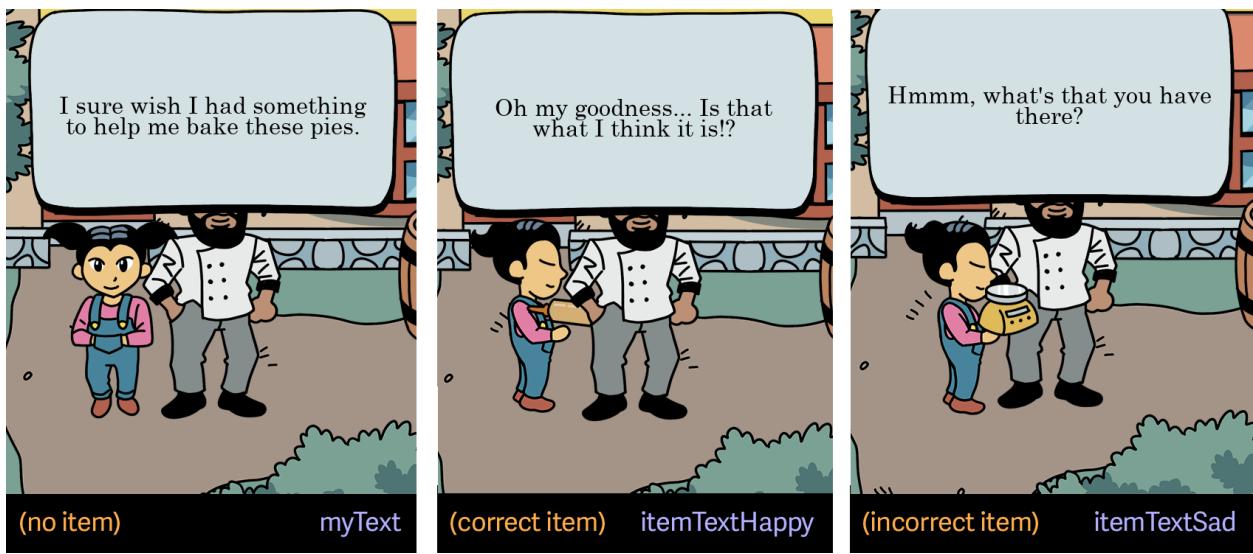
If you review the code, you can see that we are:

- Checking to see if the player has an item (and if that object still exists)
- Then we're checking if that item is the correct one (by checking the value of `myItem` within the NPC we're standing next to (`nearbyNPC`)
- If this is the case, we're setting the variable `_text` to be what we wrote for the `itemTextHappy` Variable Definition
- If it's *not*, we're setting `_text` instead to be what we wrote for the `itemTextSad` Variable Definition
- In either case, we create a textbox Instance, and populate it with whatever the value of `_text` is

	<p>Tip: You'll notice we're using a new built-in variable: <code>object_index</code>. That's because the value stored in <code>hasItem</code> is actually the ID of that Object's particular <i>instance</i>. By checking <code>hasItem</code>'s <code>object_index</code>, we're not looking at the specific instance ID, but the general ID of the <i>Object</i> (in this case of the Baker, that would be <code>obj_item01</code>).</p>
--	---

Save your project and run your game. You should be able to do three things:

1. Talk to any of the three NPCs without carrying an Object; they should say whatever you put into the `myText` Variable Definition
2. Talk to an NPC while carrying the object you defined for them in the `myItem` Variable Definition; they should say whatever you set as their `itemTextHappy` Variable Definition
3. Talk to an NPC while carrying any Object other than their `myItem` Object; they should say whatever you set as their `itemTextSad` Variable Definition



Pressing the Space bar while next to the Baker should now give us different text if we aren't carrying an item, if we are carrying the correct item and if we're carrying an incorrect item.

9.4 Playing a Sequence via the Textbox Object

Our NPCs now can tell whether we're carrying an item and which one. Now it's time to make sure the game plays the correct Sequence, depending on the item.

Currently we can play our Baker's "happy" Sequence (`seq_baker_happy`) by pressing the "S" key. However, what we want to happen is this:

1. The player brings an item over to an NPC
 2. Depending on whether we have the correct item or not, the NPC will greet us with their `itemTextHappy` or `itemTextSad` dialogue, respectively
 3. After the player presses the Space bar to close the textbox, the appropriate Sequence will play (again, depending on the item)
 4. After the Sequence finishes, we return to gameplay and can control the player Object again

So, since we want to play our Sequences *after* we pop up a textbox, we're going to use the textbox itself to do the heavy lifting.

First, open obj_textbox. In its Create Event, add this line to the list of variables in the // Textbox variables code block:

```
| sequenceToShow = noone;
```

Next, open obj_textbox's Alarm 0 Event and find the // Destroy me code block that we wrote before:

```
// Destroy me
global.playerControl = 1;
instance_destroy();
```

Delete the entire code block and replace it with this new one:

```
// Return control to player if no Sequence to load
if (sequenceToShow == noone) {
    global.playerControl = true;
}

// Create Sequence if appropriate
if (sequenceToShow != noone) {

    // Set Sequence to centre of Camera view
    var _camX =
camera_get_view_x(view_camera[0])+floor(camera_get_view_width(view_camera[0])*0.5)
;
    var _camY =
camera_get_view_y(view_camera[0])+floor(camera_get_view_height(view_camera[0])*0.5
);

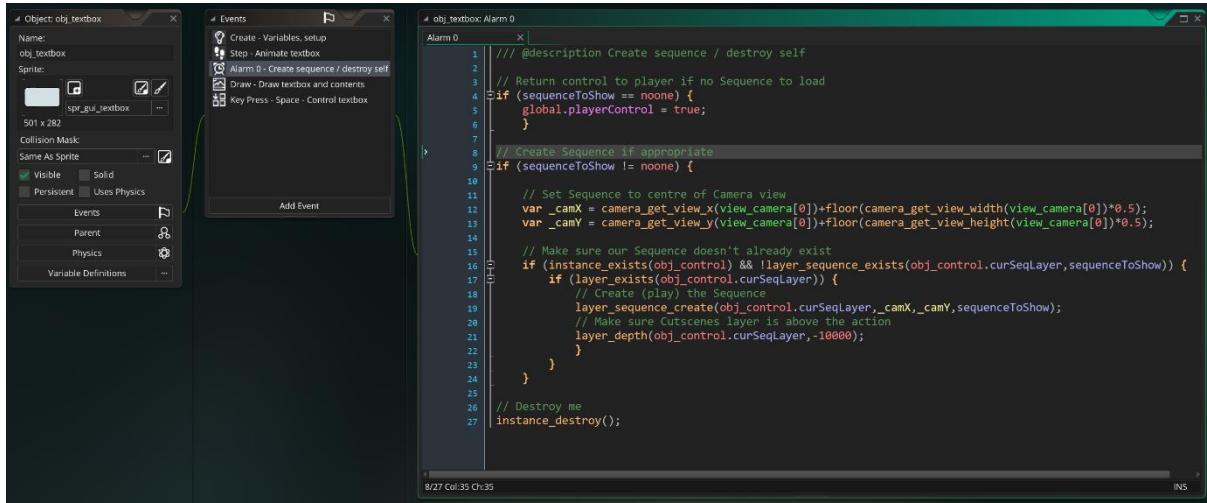
    // Make sure our Sequence doesn't already exist
    if (instance_exists(obj_control) &&
!layer_sequence_exists(obj_control.curSeqLayer,sequenceToShow)) {
        if (layer_exists(obj_control.curSeqLayer)) {
            // Create (play) the Sequence
            layer_sequence_create(obj_control.curSeqLayer,_camX,_camY,sequenceToShow);
            // Make sure Cutscenes layer is above the action
            layer_depth(obj_control.curSeqLayer,-10000);
        }
    }
}

// Destroy me
instance_destroy();
```

You'll notice we're performing a check to see if that new variable we just added (sequenceToShow) is anything other than its default value. If it's not, we continue as usual.

If `sequenceToShow` has a different value, though, we're doing something with which we should already be familiar: it's the exact same code we wrote back in [Testing our Sequence](#). Only now, instead of pressing a key to launch a Sequence, we're doing it through the `textbox` Object.

Finally, at the end of this code block is the same `instance_destroy()` function as before, so we can get rid of the `textbox` when we no longer need it.



Updating obj_textbox's Alarm 0 to now create the appropriate Sequence before it destroys itself.

9.5 Telling the Textbox which Sequence to play

Now that `obj_textbox` can start playing a Sequence, we need to trigger this actual behaviour and tell the textbox *which* Sequence to play. In order to do this, we need to head back to our player Object.

Open `obj_player` once again and open its Key Press - Space Event.

Edit the line that says:

```
| var _text;
```

And change it to:

```
| var _text, _seq;
```

(All we're doing here is initializing another variable (`_seq`) without any value on the same line as before.)

Next, we need to update the // If near an NPC code block again to allow us to tell a textbox which Sequence to play when it's done. Edit this code block like so:

```
// If near an NPC
if (nearbyNPC) {
    // If player does not have an item
    if (hasItem == noone || hasItem == undefined) {
        _text = nearbyNPC.myText;
        if (!instance_exists(obj_textbox)) {
            iii = instance_create_depth(nearbyNPC.x,nearbyNPC.y-400,-
10000,obj_textbox);
            iii.textToShow = _text;
        }
    }
    // If player has item (and it still exists)
    if (hasItem != noone && instance_exists(hasItem)) {
        // If player has correct item
        if (hasItem.object_index == nearbyNPC.myItem) {
            _text = nearbyNPC.itemTextHappy;
            _seq = nearbyNPC.sequenceHappy;
        }
        // Or if player has incorrect item
        else {
            _text = nearbyNPC.itemTextSad;
            _seq = nearbyNPC.sequenceSad;
        }
        // Create textbox
        if (!instance_exists(obj_textbox)) {
            iii = instance_create_depth(nearbyNPC.x,nearbyNPC.y-400,-
10000,obj_textbox);
            iii.textToShow = _text;
            iii.sequenceToShow = _seq;
        }
    }
}
```

As you can see, we're simply adding the new temporary variable `_seq` and assigning it a value that depends on the item the player is carrying. Then, if the player is carrying an item, we pass the value of `_seq` on to `obj_textbox` via that new `sequenceToShow` variable we just made.

The easiest way to understand this is to see it in action. So once again, save your project and run your game. You should be able to do the following three things:

1. Talk to any of the three NPCs without carrying an Object; they should say whatever you put into the `myText` Variable Definition, as before

2. Talk to the same NPC while carrying the object you defined for them in the `myItem` Variable Definition; they should say whatever you set as their `itemTextHappy` Variable Definition, and then their “happy” Sequence should play
3. Talk to the same NPC again while carrying any Object other than their `myItem` Object; they should say whatever you set as their `itemTextSad` Variable Definition, and then their “sad” Sequence should play

In all three cases, the player should be able to continue moving (or carrying an item) after interacting with the NPC.

9.6 Aligning Sequence emitters

You may have noticed something odd while testing our Sequences just now. Depending on which character you give an item to, you may notice that the music for their Sequence sounded quiet, or far away, or off in a particular direction.

What's going on here?

Well, as it turns out, a Sequence is also an *Emitter* — just like our gurgling fountain is. So depending on where the player is in the room when the Sequence plays, you may hear the Sequence audio differently.

When we used the “S” key to test our Sequence (back in [Testing our Sequence](#)), the Sequence was always being lined up with the player Object, which meant its Emitter was right on top of the player’s Listener — so everything sounded correct.

But now with our NPCs that’s not the case, so we need to make one small addition to `obj_player` to correct this.

Open `obj_player` and go to its Step Event.

In the // If moving code block, find the // Move audio listener with me line, which should look like this:

```
// Move audio listener with me  
audio_listener_set_position(0,x,y,0);
```

We need to update this to check for Sequences, and if one is playing, to adjust the player’s Listener so it lines up with the Sequence.

Replace the // Move audio listener with me code block with this:

```

// Set my listener if Sequence is playing
if (instance_exists(obj_control) && obj_control.sequenceState == seqState.playing)
{
    var _camX =
camera_get_view_x(view_camera[0])+floor(camera_get_view_width(view_camera[0])*0.5)
;
    var _camY =
camera_get_view_y(view_camera[0])+floor(camera_get_view_height(view_camera[0])*0.5
);

    audio_listener_set_position(0,_camX,_camY,0);
}
else {
    // Otherwise, move audio listener with me
    audio_listener_set_position(0,x,y,0);
}
}

```

In this updated code, we're checking for `obj_control`'s `sequenceState` variable, to see if it's been set to the enum constant `seqState.playing` (meaning that a Sequence is currently playing).

If this is true, we use the same camera-checking code we've used before in `obj_textbox`'s `Alarm 0` Event and using it to temporarily set the position of the player Object's Listener.

If a Sequence *isn't* playing, we keep that Listener aligned with the player, as we did before.

Run the game again and give an item to one of the NPCs. You should now hear that the Sequence audio is correctly “centered” and doesn’t sound like it’s playing from off-screen somewhere.

9.7 Removing “correct” Objects once given

Next, we need to make our item-giving gameplay loop a bit more realistic. When you bring the correct item to an NPC, that item should disappear and the player should no longer be carrying it, since you’ve returned the item to the NPC.

(This won’t happen with other items, since the NPC doesn’t want them and you might need to give them to somebody else.)

How do we know what the correct item is for each character? Well, currently, each NPC has a Variable Definition set in the Object Editor. For example, we set the Baker’s `myItem` variable to `obj_item01` back in [Changing what the NPCs say](#).

But now we also need to create another variable to help us track the status of an NPC, so we can change it when they've been given that correct item. And since we have a parent Object for all NPCs (`obj_par_npc`), this can be done easily.

9.8 Setting up NPC states

Just like we did for our player and our items, we're going to create an enum for our NPCs.

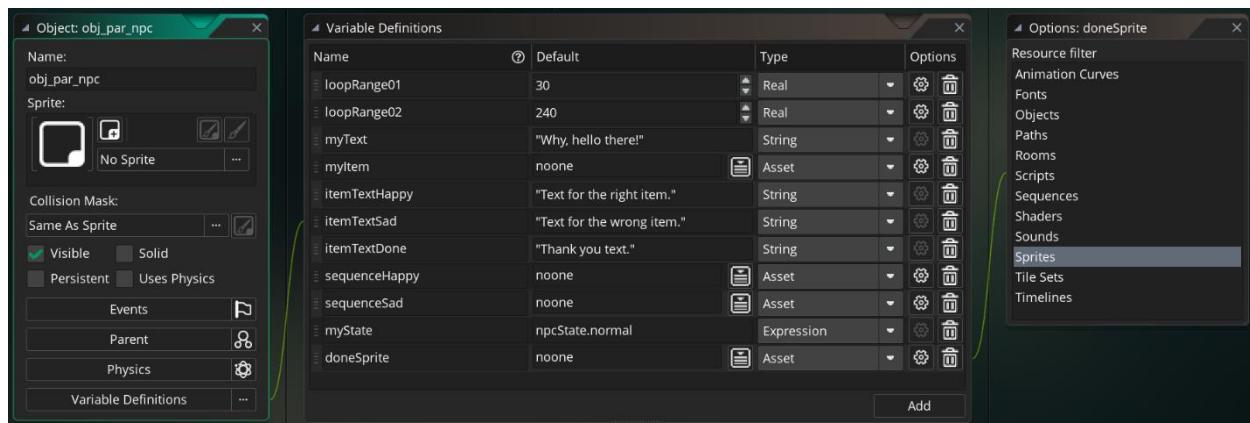
Open `obj_control` and its Game Start Event. Below the `// Sequence variables` code block, add this new code block:

```
// NPC states
enum npcState {
    normal,
    done,
}
```

Next, open `obj_par_npc` from the Asset Browser. In the Object Editor, click on “Variable Definitions” to bring up that ever-growing list of variables for our characters.

At the bottom of the Variable Definitions window, click Add and create two new variables with the following properties:

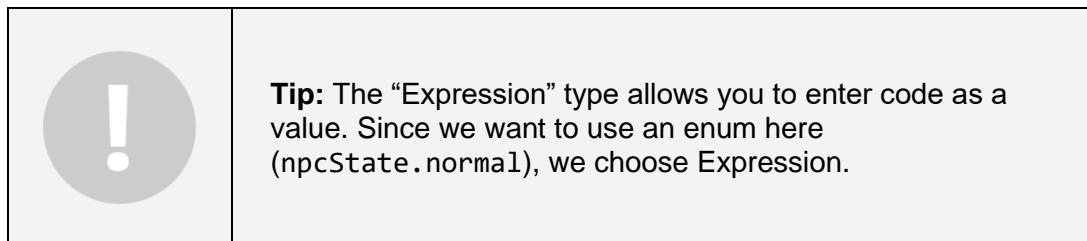
Name	Default	Type
myState	<code>npcState.normal</code>	Expression
doneSprite	<code>noone</code>	Asset (Sprite)



Adding the myState and doneSprite Variable Definitions to obj_par_npc.

We'll use this variable (myState) to mark when the NPC you give an item to has been given the correct one. Once an NPC has the item they wanted, you no longer can attempt to give them items, since they are "done."

And, as you'll see, we'll use the new doneSprite variable to visually let the player know that the NPC has been given their desired item.



9.9 Marking an NPC as "done"

To keep things simple, we're going to use an Alarm in the player Object to change the nearby NPCs state to "done" and remove the correct item from the player's possession.

Open `obj_player` from the Asset Browser. In the Object Editor, click Add Event and choose Alarms > Alarm 1.

In this new Alarm Event, add the following code block:

```
// Remove Object
```

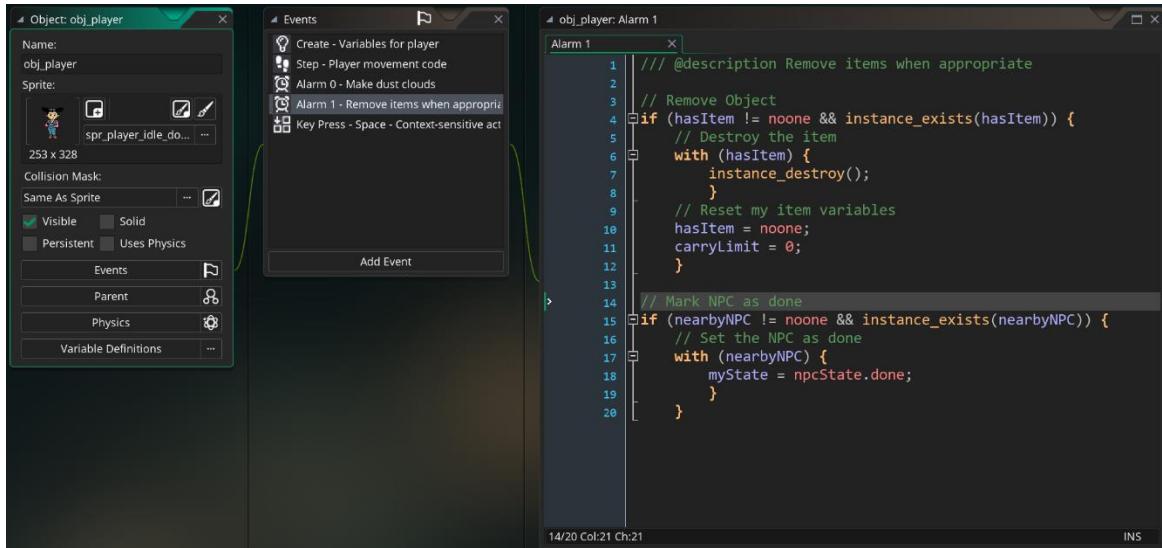
```

if (hasItem != noone && instance_exists(hasItem)) {
    // Destroy the item
    with (hasItem) {
        instance_destroy();
    }
    // Reset my item variables
    hasItem = noone;
    carryLimit = 0;
}

// Mark NPC as done
if (nearbyNPC != noone && instance_exists(nearbyNPC)) {
    // Set the NPC as done
    with (nearbyNPC) {
        myState = npcState.done;
    }
}

```

You'll see that there isn't anything new here, code wise. In the // Remove Object code block, we're telling the Object that the player is holding (hasItem) to destroy itself. In the // Mark NPC as done code block, we're using the new npcState enum to set that new variable (myState) in the nearbyNPC.



Adding the new Alarm 1 to obj_player to deal with items and marking NPCs as "done."

Now that we have this Alarm, we need to trigger it at the right time. Open obj_player's Key Press - Space Event.

Find the // If player has correct item code block (which is nested within the // If player has control code block), and update it like so:

```
// If player has correct item
if (hasItem.object_index == nearbyNPC.myItem) {
    _text = nearbyNPC.itemTextHappy;
    _seq = nearbyNPC.sequenceHappy;
    // Check if we should remove item, mark NPC
    alarm[1] = 10;
}
```

Since we already coded in an if statement here to check if our player is carrying the correct Object, we can use this spot to trigger Alarm 1.

9.10 Switching our NPCs state

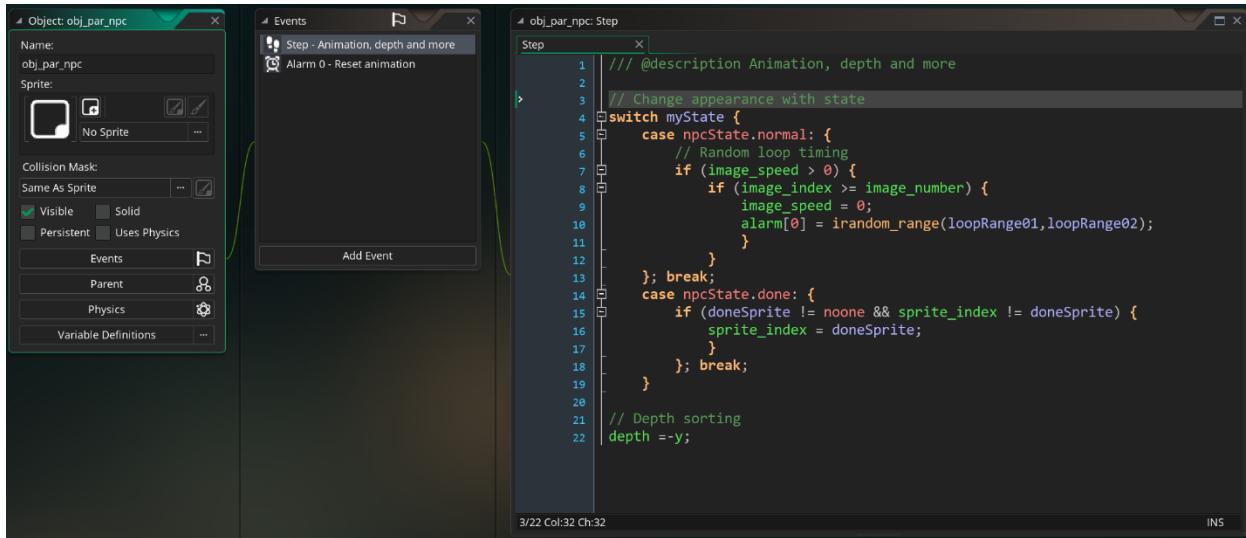
Now we can use that new npcState enum to control our NPCs a bit more. To do this, we need to add some code that will change the appearance of our NPCs, depending on their state.

So, open obj_par_npc and its Step Event.

Replace the entire // Random loop timing code block like so:

```
// Change appearance with state
switch myState {
    case npcState.normal: {
        // Random loop timing
        if (image_speed > 0) {
            if (image_index >= image_number-1) {
                image_speed = 0;
                alarm[0] = irandom_range(loopRange01,loopRange02);
            }
        }
    }; break;
    case npcState.done: {
        if (doneSprite != noone && sprite_index != doneSprite) {
            sprite_index = doneSprite;
        }
    }; break;
}
```

We've added a switch statement here; the `npcState.normal` case behaves exactly as before. In the `npcState.done` case, however, we're simply changing the NPC's sprite to the value of `doneSprite`.



Editing obj_par_npc's Step Event to account for our new states.

9.11 Setting the NPCs “done” Sprites

Now that our NPCs can change their own Sprite when they're marked as “done,” we need to specify what that Sprite will be for each of them.

First, open `obj_npc_baker` and its Variable Definitions.

Click the Edit button for the `doneSprite` Variable Definition and choose the `spr_baker_happy` Sprite.

The screenshot shows two panels side-by-side. On the left is the 'Object: obj_npc_baker' properties panel. It includes fields for Name (obj_npc_baker), Sprite (spr_baker_idle_down), Collision Mask (Same As Sprite), and various physics and visibility settings. On the right is the 'Variable Definitions' panel, which lists several variables with their types and values. The 'doneSprite' variable is highlighted, showing its current value as 'spr_baker_happy'. A green bracket on the right side of the image groups the 'Variable Definitions' panel and the 'doneSprite' row in the list.

Name	Default	Type	Options
loopRange01	60	Real	
loopRange02	320	Real	
myText	"I sure wish I had something to help	String	
myItem	obj_item01	Asset	
itemTextHappy	"Oh my goodness... Is that what I thi	String	
itemTextSad	"Hmmm, what's that you have there?	String	
itemTextDone	"Thank you! Now I can bake pies all	String	
sequenceHappy	seq_baker_happy	Asset	
sequenceSad	seq_baker_sad	Asset	
myState	npcState.normal	Expression	
doneSprite	spr_baker_happy	Asset	

Choosing the correct “done” Sprite for the Baker.

Next, open `obj_npc_teacher` and do the same thing: click on **Variable Definitions**, edit the `doneSprite` row, and choose `spr_teacher_happy`.

The screenshot shows the same two panels for `obj_npc_teacher`. The 'Variable Definitions' panel is shown again, with the 'doneSprite' row selected and its value set to 'spr_teacher_happy'. A green bracket on the right side of the image groups the 'Variable Definitions' panel and the 'doneSprite' row in the list.

Name	Default	Type	Options
loopRange01	90	Real	
loopRange02	400	Real	
myText	"I wish I had something to eat before	String	
myItem	obj_item02	Asset	
itemTextHappy	"Is that an apple!? How thoughtful of	String	
itemTextSad	"Oh, um, what do you have there?"	String	
itemTextDone	"Thanks to you, I could teach all day!	String	
sequenceHappy	seq_teacher_happy	Asset	
sequenceSad	seq_teacher_sad	Asset	
myState	npcState.normal	Expression	
doneSprite	spr_teacher_happy	Asset	

Choosing a “done” Sprite for the Teacher.

Finally, open `obj_npc_grocer` and repeat the process. Choose `spr_grocer_happy` as its `doneSprite` value.

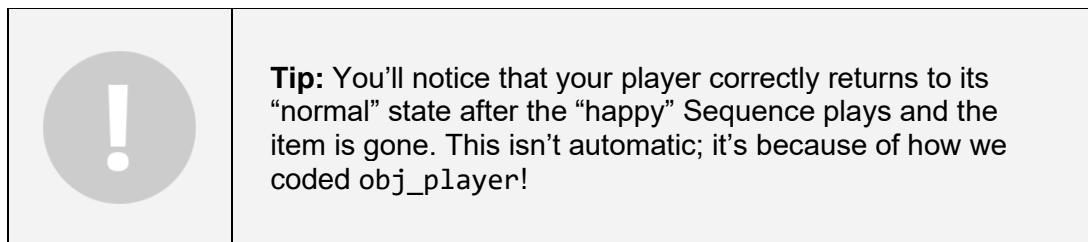
The screenshot shows two panels side-by-side. On the left is the 'Object: obj_npc_grocer' panel, which includes fields for Name (obj_npc_grocer), Sprite (spr_grocer_idle_d...), Collision Mask (Same As Sprite), and various visibility and physics settings. On the right is the 'Variable Definitions' panel, listing variables like loopRange01, myText, and itemTextHappy, each with a default value and type. A green line points from the 'doneSprite' variable in the Variable Definitions panel to the 'Variable Definitions' section in the Object Properties panel.

Name	Default	Type	Options
loopRange01	50	Real	
loopRange02	550	Real	
myText	"How am I going to keep these veggies?"	String	
myItem	obj_item06	Asset	
itemTextHappy	"Oh, a watering can! Of course, that's..."	String	
itemTextSad	"What's that you have there?"	String	
itemTextDone	"With that watering can, now my veg..."	String	
sequenceHappy	seq_grocer_happy	Asset	
sequenceSad	seq_grocer_sad	Asset	
myState	npcState.normal	Expression	
doneSprite	spr_grocer_happy	Asset	

Choosing the “done” Sprite for the Grocer.

With all this done, save your project and run your game again to test it. Try this:

1. Pick up an item that isn't the correct item for an NPC
2. Bring the item over to the NPC and press Space to talk to them
3. After their “sad” Sequence plays, note that the player still has the item in hand
4. Walk away from the NPC and put down the item
5. Grab the correct item and bring it to the NPC
6. Press Space to talk to the NPC
7. Note that the item disappears from the player's possession
8. See that the NPC now does their happy dance



9.12 Changing interaction when an NPC is “done”

There is one more step to make our main gameplay loop complete. When the player speaks to an NPC after the NPC already has their desired item, the NPC should say something different — that `itemTextDone` string that we added as a Variable Definition to `obj_par_npc`.

Thankfully, we’ve already set up everything we need to do this: we’ve created text for this situation for each of our NPCs, and we now mark an NPC as “done” when the player Object gives them the correct item.

All that’s left is to check if an NPC is “done” before popping up the usual textbox.

To do this, open `obj_player` once again and its Key Press – Space Event.

We need to make an additional change to the big `// If near an NPC` code block, so update it like so:

```
// If near an NPC
if (nearbyNPC) {
    // If NPC is still available
    if (nearbyNPC.myState == npcState.normal) {
        // If player does not have an item
        if (hasItem == noone) {
            _text = nearbyNPC.myText;
            if (!instance_exists(obj_textbox)) {
                iii = instance_create_depth(nearbyNPC.x,nearbyNPC.y-400,-
10000,obj_textbox);
                iii.textToShow = _text;
            }
        }
        // If player has item (and it still exists)
        if (hasItem != noone && instance_exists(hasItem)) {
            // If player has correct item
            if (hasItem.object_index == nearbyNPC.myItem) {
                _text = nearbyNPC.itemTextHappy;
                _seq = nearbyNPC.sequenceHappy;
                // Check if we should remove item, mark NPC
                alarm[1] = 10;
            }
            // Or if player has incorrect item
            else {
                _text = nearbyNPC.itemTextSad;
                _seq = nearbyNPC.sequenceSad;
            }
        }
        // Create textbox
```

```

        if (!instance_exists(obj_textbox)) {
            iii = instance_create_depth(nearbyNPC.x,nearbyNPC.y-400,-
10000,obj_textbox);
            iii.textToShow = _text;
            iii.sequenceToShow = _seq;
        }
    }
}

// If NPC is "done"
if (nearbyNPC.myState == npcState.done) {
    _text = nearbyNPC.itemTextDone;
    if (!instance_exists(obj_textbox)) {
        iii = instance_create_depth(nearbyNPC.x,nearbyNPC.y-400,-
10000,obj_textbox);
        iii.textToShow = _text;
    }
}
}

```

What we've done here is add yet another if statement to check if the nearbyNPC has had their myState variable set to npcState.done or not. If not, we do everything we did before. If so, however, the NPC will now only say the text stored in their itemTextDone Variable Definition.



Tip: See the  button on the left edge of your code? Clicking this lets you “fold” regions of code so you can focus. Once a region is folded, this button turns into . Click it again to “unfold” the region and see it again.

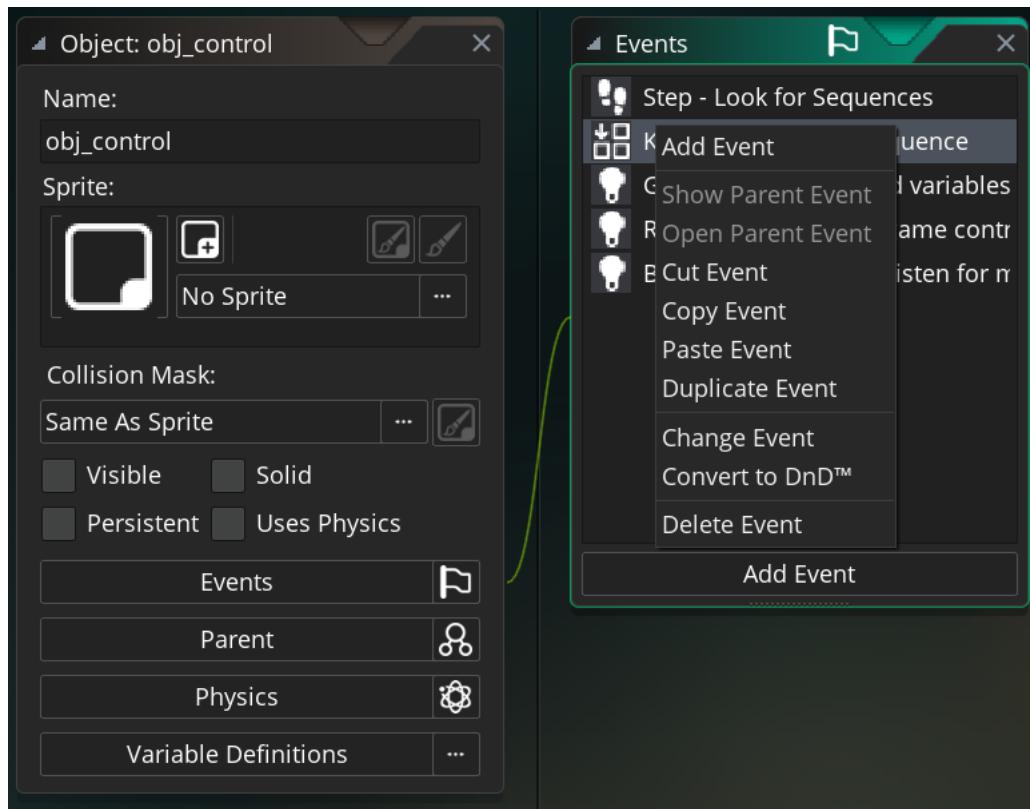
9.13 Removing the “S” key test Event

One last bit of clean-up before we continue: back in [Testing our Sequence](#) we added a Key Press – S Event to obj_control so we could test our Sequence.

We don't need this anymore, so we're going to remove it.

Open obj_control and review the Events in the Object Editor.

Right-click on Key Press – S Event and choose Delete Event. Now the player won't be able to accidentally trigger our test.



Deleting the Key Press – S Event from obj_control.

9.14 Designing a “Game Over” Sequence

Our core gameplay loop is now intact. But what we really need is a finale to our game. Since we can mark our NPCs as being “done,” we have a way to track the player’s progress.

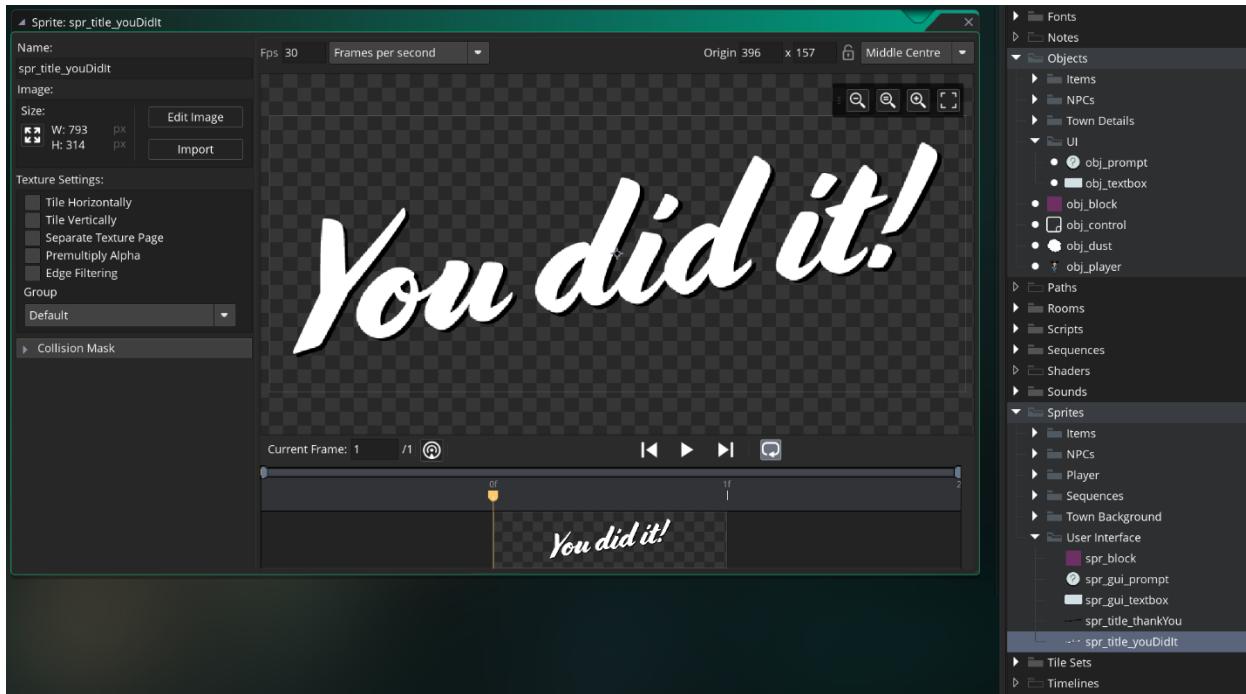
What we want to do now is check if all three NPCs are “done,” and then play a special “game over” Sequence before restarting the game.

First, let’s create the Sequence we’ll show at the end.

Using File Explorer (Windows) or Finder (Mac), navigate to the Assets folder provided with this course. Go to Sprites > User Interface and drag the following files into the Asset Browser.

- spr_title_youDidIt
- spr_title_thankYou

In GameMaker Studio 2, open both new Sprites and set their Origin to Middle Centre. Organize the Assets in the Asset Browser if you have been doing so.

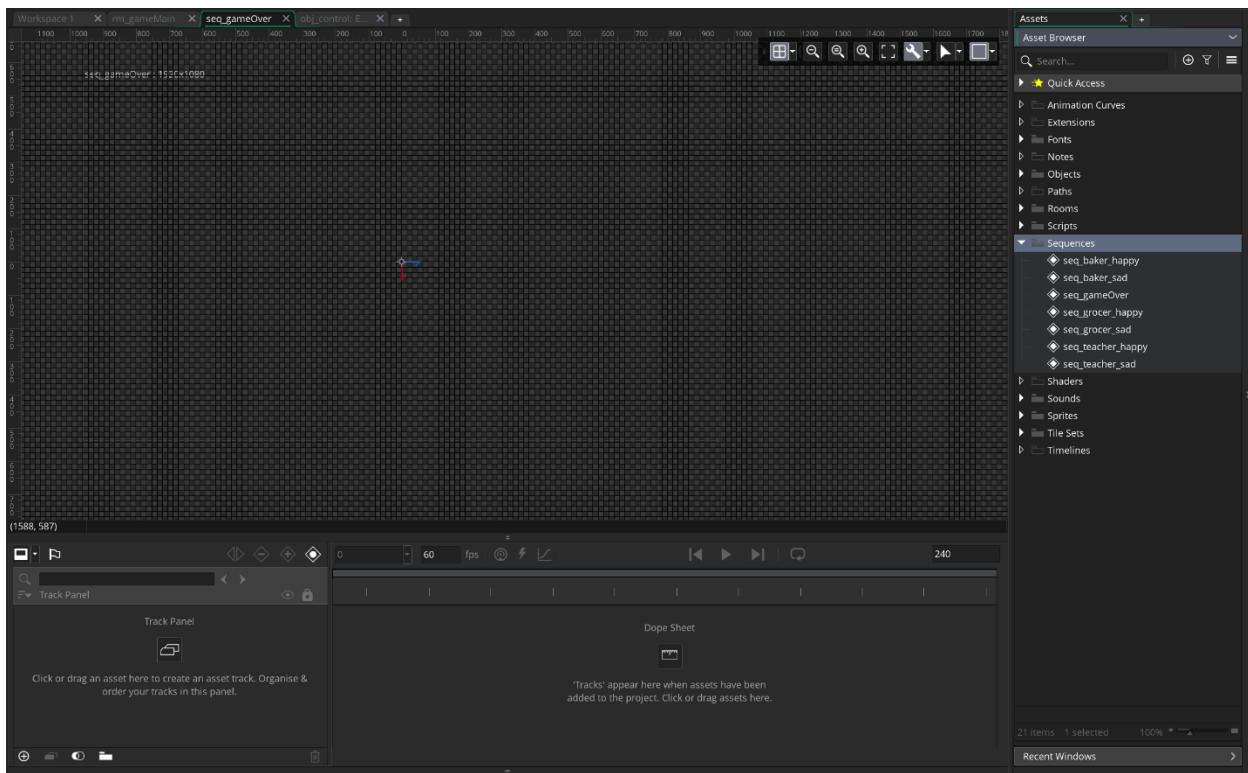


Setting the Origin of one of the new title Sprites. Note how we've organized them within the Asset Browser.

Next, right-click on the Sequences group in the Asset Browser and choose Create > Sequence. Name this new Sequence seq_gameOver.

Open seq_gameOver; in the Sequence Editor, click on the Canvas frame settings button in the Canvas toolbar (at the top right).

Change the Canvas width to 1920 and its height to 1080.



Changing the Canvas size of the new seq_gameOver Sequence.

Now that the Canvas is ready, design a Sequence using the following elements:

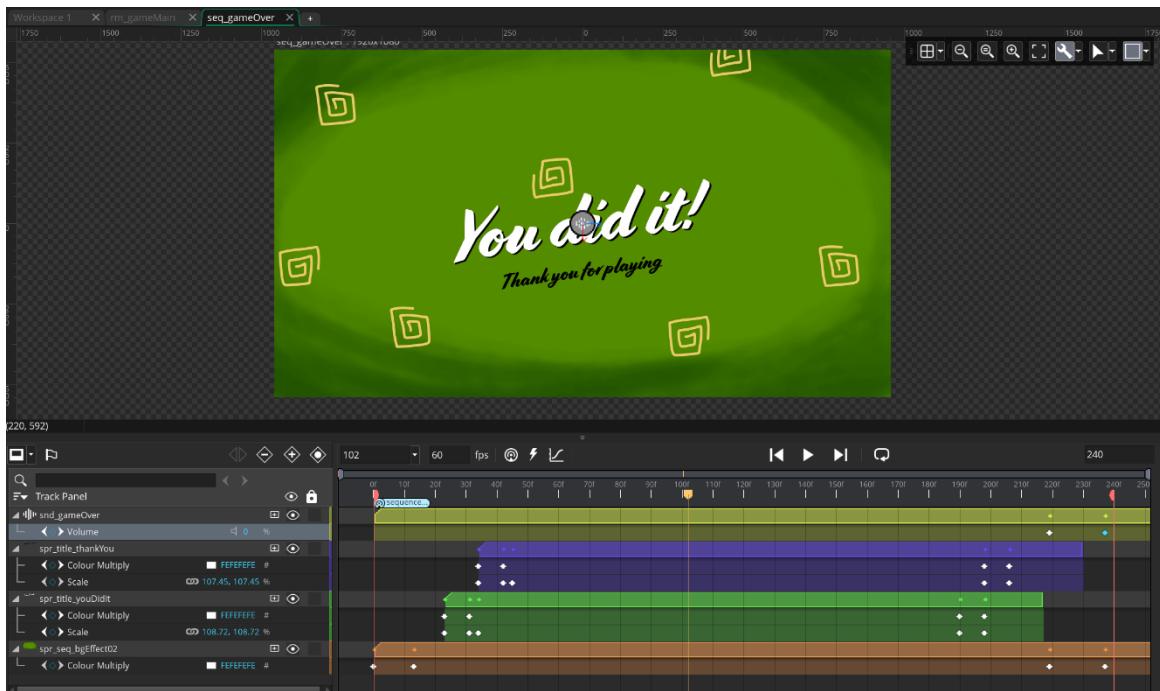
- the spr_title_youDidIt Sprite
- the spr_title_thankYou Sprite
- A background of some kind
- Any other visual elements you want to include

Make sure your Sequence has the following features:

- It has a background/colour that fills the Canvas
- It lasts around 4 seconds (240 frames)
- Uses the snd_gameOver Sound asset
- At the end of the Sequence, it transitions to black (it could be through a fade or any other effect)
- There is a Broadcast Message at the beginning of the Sequence called sequenceStart

If you need a refresher on using parameter tracks or Broadcast Messages, refer back to [Starting our first Sequence](#).

Here is our example:



In our example Game Over sequence, with title assets, fades and a music track

When you're happy with the design of your Sequence, let's move on to making it work!

9.15 Setting up the “Game Over” Sequence

To begin the finale for our game, we first need to track whether the player has accomplished its goal (giving the correct items to all three NPCs).

Since we already have a way to denote this (we set each of our NPCs' state to `npcState.done`), we can use some simple if statements to check!

Open `obj_control` and its Game Start Event. Add the following line to the // Game variables code block:

```
// Game variables
global.playerControl = true;
townBGMvolume = audio_sound_get_gain(snd_townBGM);
townAmbienceVolume = audio_sound_get_gain(snd_townAmbience);
global.gameOver = false;
```

We'll use this new global variable to control when we end the game.

Next, open obj_control's Step Event. Edit the // Control Sequences code block like so:

```
// Control Sequences
switch sequenceState {
    case seqState.playing: {
        // Fade out town music
        if (audio_is_playing(snd_townBGM)) {
            audio_sound_gain(snd_townBGM,0,60);
        }
        // Fade out town ambience
        if (audio_is_playing(snd_townAmbience)) {
            audio_sound_gain(snd_townAmbience,0,60);
        }
        global.playerControl = false;
    }; break;
    case seqState.finished: {
        // Remove Sequence
        if (layer_sequence_exists(curSeqLayer,curSeq)) {
            layer_sequence_destroy(curSeq);
        }
        // Restore control to player, reset
        global.playerControl = true;
        sequenceState = seqState.notPlaying;
        curSeq = noone;

        // Restore town music
        if (audio_is_playing(snd_townBGM) && audio_sound_get_gain(snd_townBGM) < townBGMvolume) {
            audio_sound_gain(snd_townBGM,townBGMvolume,60);
        }
        // Restore town ambience
        if (audio_is_playing(snd_townAmbience) && audio_sound_get_gain(snd_townAmbience) < townAmbienceVolume) {
            audio_sound_gain(snd_townAmbience,townAmbienceVolume,60);
        }

        // Check if NPCs are "done"
        if (global.gameOver == false) {
            if (instance_exists(obj_npc_baker) && instance_exists(obj_npc_teacher) && instance_exists(obj_npc_grocer)) {
                if (obj_npc_baker.myState == npcState.done && obj_npc_teacher.myState == npcState.done && obj_npc_grocer.myState == npcState.done) {
                    // Queue up "game over" sequence
                    global.playerControl = false;
                    alarm[0] = 60;
                    // Mark game as won
                    global.gameOver = true;
                }
            }
        }
    }
}
```

```

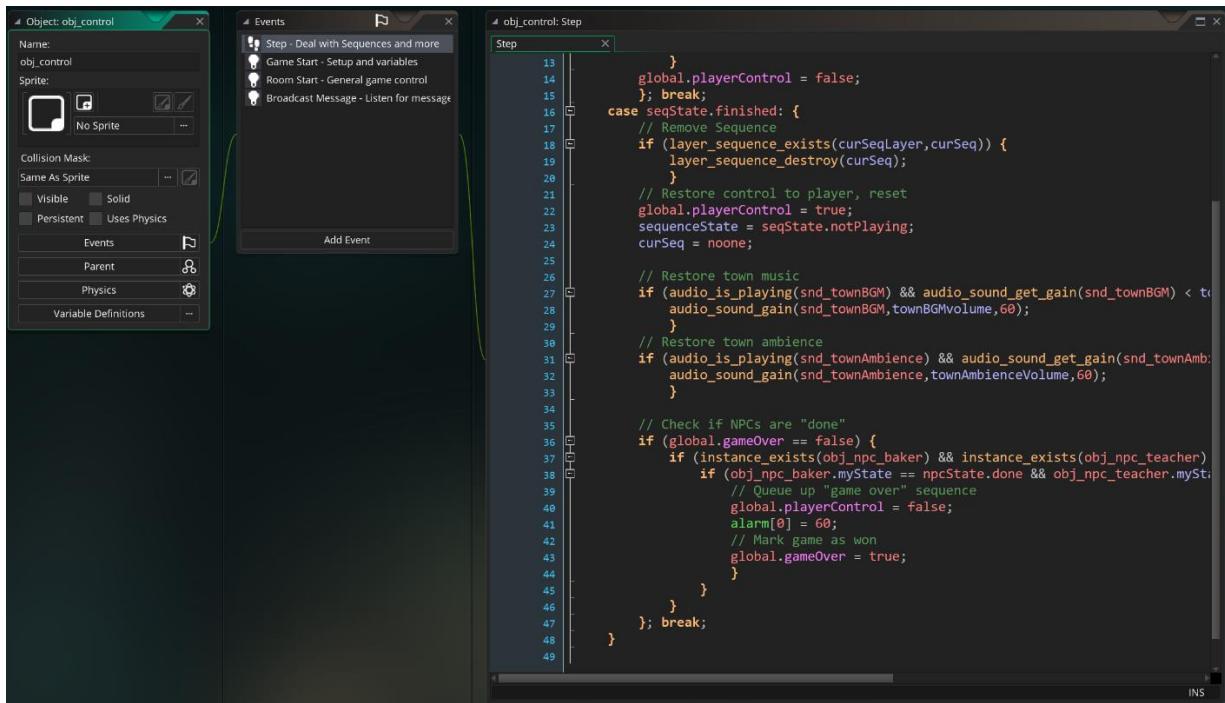
        }
    }
}; break;
}

```

This new // Check if NPCs are “done” code block will queue up the end of our game. This may look a little unwieldy, but there are just two if statements here:

- One to make sure that our three NPC Objects are there (just to be safe)
- Another to ask, in one line, if each of them has been set to “done”

If both if statements are satisfied, then we remove control from the player, set an Alarm, and set `global.gameOver` to true.



Checking in obj_control’s Step Event if we can end the game.

In this Step Event, we’ve set Alarm 0, but we haven’t added this yet. Let’s do that now.

Still in obj_control, click Add Event and choose Alarms > Alarm 0.

In this new Alarm Event, we want to create an instance of the “game over” Sequence. Now, we could just copy and paste the code that we used in obj_textbox back in [Playing a Sequence via the Textbox Object](#).

However, as we mentioned back in [Creating our own functions within Script Assets](#), if we're going to re-use the same bits of code over and over, it's better to turn that code into a function. So, before we continue, let's take care of that.

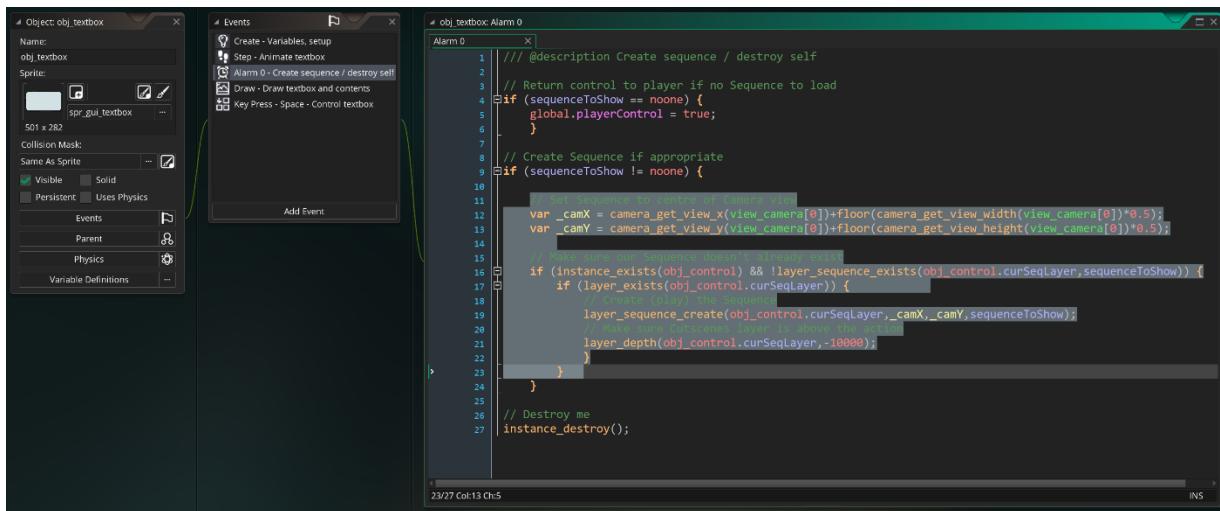
9.16 Creating a function to play Sequences

Open obj_textbox and its Alarm 0 Event.

Copy or cut this code that's within the // Create Sequence if appropriate code block:

```
// Set Sequence to centre of Camera view
var _camX =
camera_get_view_x(view_camera[0])+floor(camera_get_view_width(view_camera[0])*0.5)
;
var _camY =
camera_get_view_y(view_camera[0])+floor(camera_get_view_height(view_camera[0])*0.5)

// Make sure our Sequence doesn't already exist
if (instance_exists(obj_control) &&
!layer_sequence_exists(obj_control.curSeqLayer,sequenceToShow)) {
    if (layer_exists(obj_control.curSeqLayer)) {
        // Create (play) the Sequence
        layer_sequence_create(obj_control.curSeqLayer,_camX,_camY,sequenceToShow);
        // Make sure Cutscenes layer is above the action
        layer_depth(obj_control.curSeqLayer,-10000);
    }
}
```



Selecting the Sequence creation code to cut.

Next, right-click on the Scripts group in the Asset Browser and choose Create > Script. Name this new Script Asset scr_sequenceControl.

Rename the automatically-created function within scr_sequenceControl like so:

```
function scr_playSequence(){  
}
```

Paste the code from obj_textbox within this new scr_playSequence() function, like so:

```
function scr_playSequence(){  
    // Set Sequence to centre of Camera view  
    var _camX =  
camera_get_view_x(view_camera[0])+floor(camera_get_view_width(view_camera[0])*0.5)  
;  
    var _camY =  
camera_get_view_y(view_camera[0])+floor(camera_get_view_height(view_camera[0])*0.5)  
;  
  
    // Make sure our Sequence doesn't already exist  
    if (instance_exists(obj_control) &&  
!layer_sequence_exists(obj_control.curSeqLayer,sequenceToShow)) {  
        if (layer_exists(obj_control.curSeqLayer)) {  
            // Create (play) the Sequence  
            layer_sequence_create(obj_control.curSeqLayer,_camX,_camY,sequenceToShow);  
            // Make sure Cutscenes layer is above the action  
            layer_depth(obj_control.curSeqLayer,-10000);  
        }  
    }  
}
```

Then, edit the function to include an argument called _seqToPlay. Then, replace any instance of sequenceToShow with _seqToPlay, like so:

```
function scr_playSequence(_seqToPlay){  
    // Set Sequence to centre of Camera view  
    var _camX =  
camera_get_view_x(view_camera[0])+floor(camera_get_view_width(view_camera[0])*0.5)  
;
```

```

    var _camY =
camera_get_view_y(view_camera[0])+floor(camera_get_view_height(view_camera[0])*0.5
);

// Make sure our Sequence doesn't already exist
if (instance_exists(obj_control) &&
!layer_sequence_exists(obj_control.curSeqLayer,_seqToPlay)) {
    if (layer_exists(obj_control.curSeqLayer)) {
        // Create (play) the Sequence
        layer_sequence_create(obj_control.curSeqLayer,_camX,_camY,_seqToPlay);
        // Make sure Cutscenes layer is above the action
        layer_depth(obj_control.curSeqLayer,-10000);
    }
}
}

```

Review the screenshot, below, if you need clarification.

```

scr_sequenceControl X
> 1 // @description Play a sequence
2
3 function scr_playSequence(_seqToPlay){
4     // Set Sequence to centre of Camera view
5     var _camX = camera_get_view_x(view_camera[0])+floor(camera_get_view_width(view_camera[0])*0.5);
6     var _camY = camera_get_view_y(view_camera[0])+floor(camera_get_view_height(view_camera[0])*0.5);
7
8     // Make sure our Sequence doesn't already exist
9     if (instance_exists(obj_control) && !layer_sequence_exists(obj_control.curSeqLayer,_seqToPlay)) {
10         if (layer_exists(obj_control.curSeqLayer)) {
11             // Create (play) the Sequence
12             layer_sequence_create(obj_control.curSeqLayer,_camX,_camY,_seqToPlay);
13             // Make sure Cutscenes layer is above the action
14             layer_depth(obj_control.curSeqLayer,-10000);
15         }
16     }
17 }
18

```

1/18 Col:33 Ch:33 INS

The final scr_playSequence function within the scr_sequenceControl Script Asset.

With this done, you can close the scr_sequenceControl Script Asset.

Open obj_textbox again and its Alarm 0 Event once more.

Replace the entire // Create Sequence if appropriate code block with this simpler version:

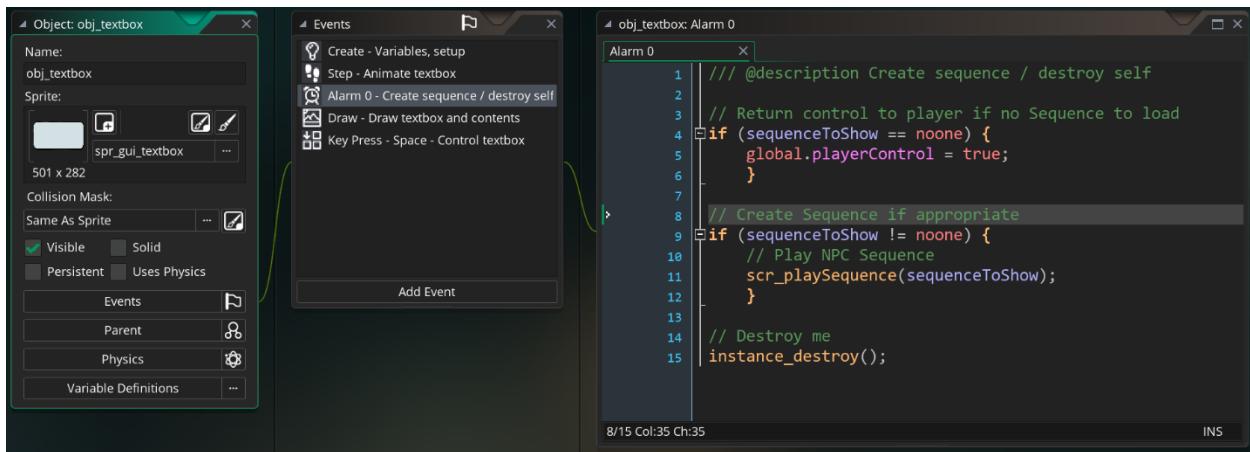
```
// Create Sequence if appropriate
```

```

if (sequenceToShow != noone) {
    // Play NPC Sequence
    scr_playSequence(sequenceToShow);
}

```

As far as your game is concerned, nothing has changed. We've just taken that large code block from this Event and turned it into a reusable function.



Our updated obj_textbox Alarm 0. It's simpler because it uses the new scr_playSequence function, but it does the same thing as before.

9.17 Playing the actual “Game Over” Sequence

Now that we've turned the ability to play a Sequence into a simple, reusable function, we can add this functionality to obj_control.

Open obj_control again. We set up obj_control's ability to know when it's time to trigger the “game over” Sequence, but now we actually have to do it.

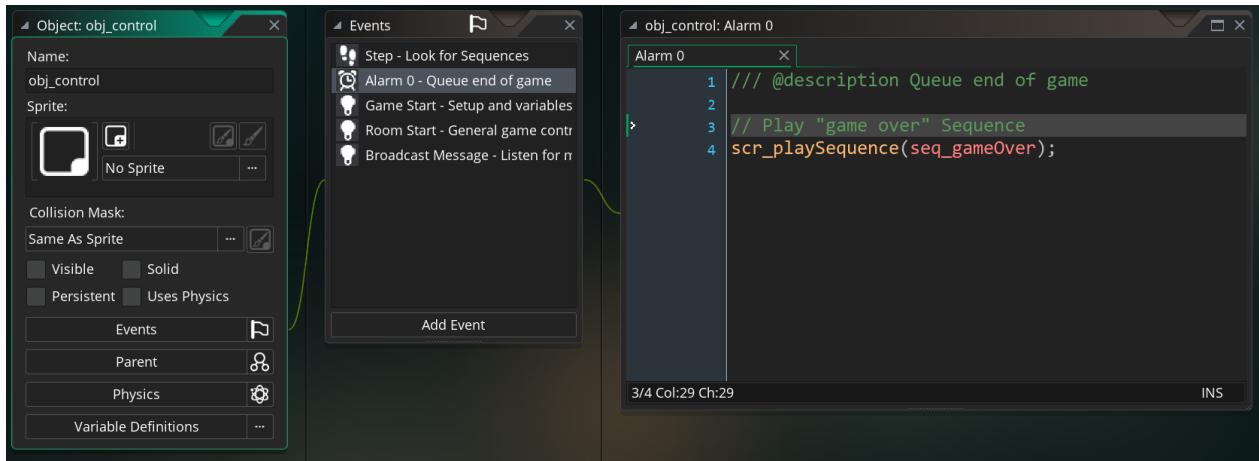
Open the Alarm 0 Event that we made in [Setting up the "Game Over" Sequence](#). It will be blank; add the following code block:

```

// Play "game over" Sequence
scr_playSequence(seq_gameOver);

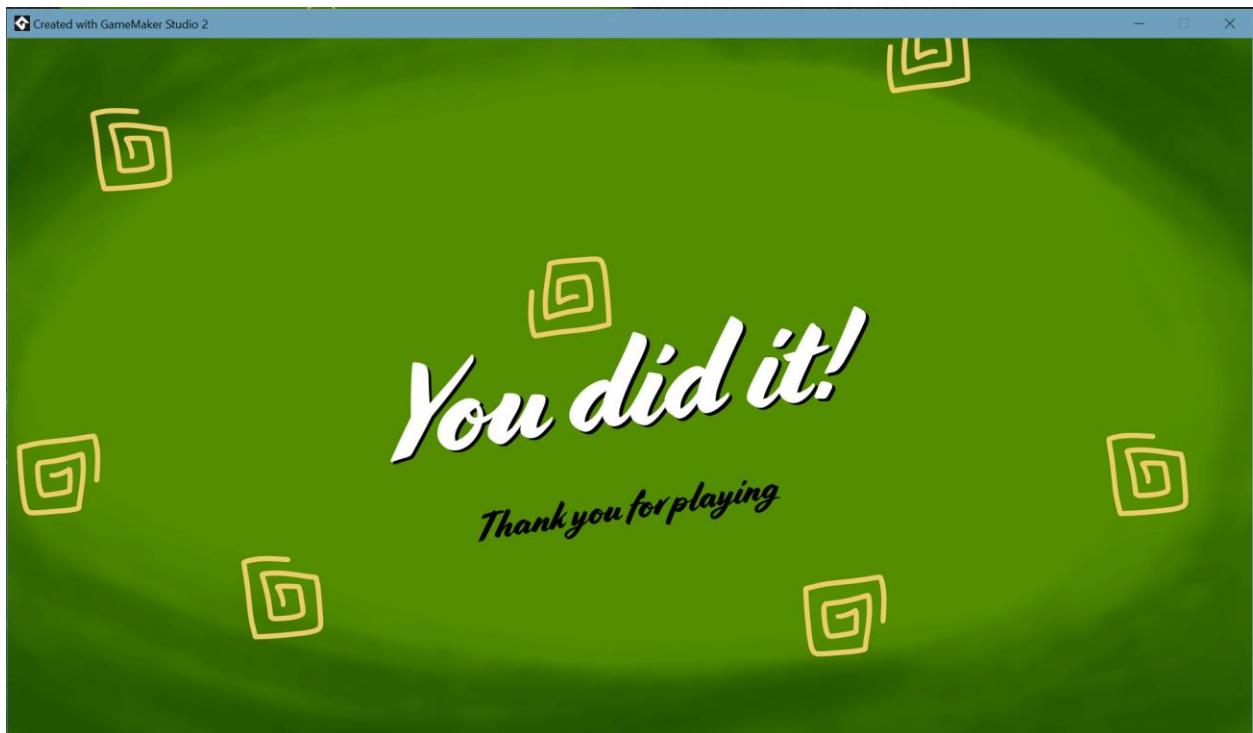
```

Here, we're using the new function and telling it to play the seq_gameOver Sequence we just designed.



Using the new scr_playSequence function to play the “game over” Sequence from obj_control.

With this done, run your game and test it. Bring each of the three NPCs their desired items; after you deliver the third correct item, your “game over” Sequence should automatically play.



Our “game over” Sequence playing in our game.

You’re going to notice one thing, though: once your Sequence ends, nothing else happens. What we need to do is include the final piece of the puzzle: a way for your game to automatically restart.

9.18 Using Moments to call function in Sequences

Up until now, we've used Broadcast Messages from our Sequences to tell other Objects in the game to do certain things. But we can take this a step further, and actually run code from within a Sequence itself!

So, we're going to create a new function that will end our game for us and call it from within our seq_gameOver Sequence.

First, right-click on the Scripts group in the Asset Browser and choose Create > Script. Call this new Script Asset scr_gameControl.

Open scr_gameControl and edit the automatically-created function to be scr_gameOver(), like so:

```
function scr_gameOver(){
    game_end();
}
```

As you might have guessed, this simple, built-in function immediately quits the game and closes its window.

Open seq_gameOver again and move the yellow playhead to a spot just after your animation end, but before the final frame of the Sequence. (If you need to adjust some assets keys to give yourself room, go ahead.)

At the top of the Dope Sheet, click the Add Moment button () ; a small “Moment” window will appear.

In the field in this window, type scr_gameOver — the full name of your function will likely auto-fill as you type.

When you've typed (or selected) this, click OK. You'll see that a small yellow tag has been added to the timeline:



Adding a Moment to our “game over” Sequence. The scr_gameOver() function will be called at this Moment.

This is a *Moment*; a point on the timeline in which we can call a function from within a Sequence.

Now when our “game over” Sequence plays, it will call the `scr_gameOver()` function we just wrote, which will quit our game immediately.

Run your game again and test; give the correct item to each of the three NPCs. After you give the third NPC their correct item and their “happy” Sequence plays, the `seq_gameOver` Sequence will play automatically, and which will now trigger the game to end.

9.19 Creating a title screen

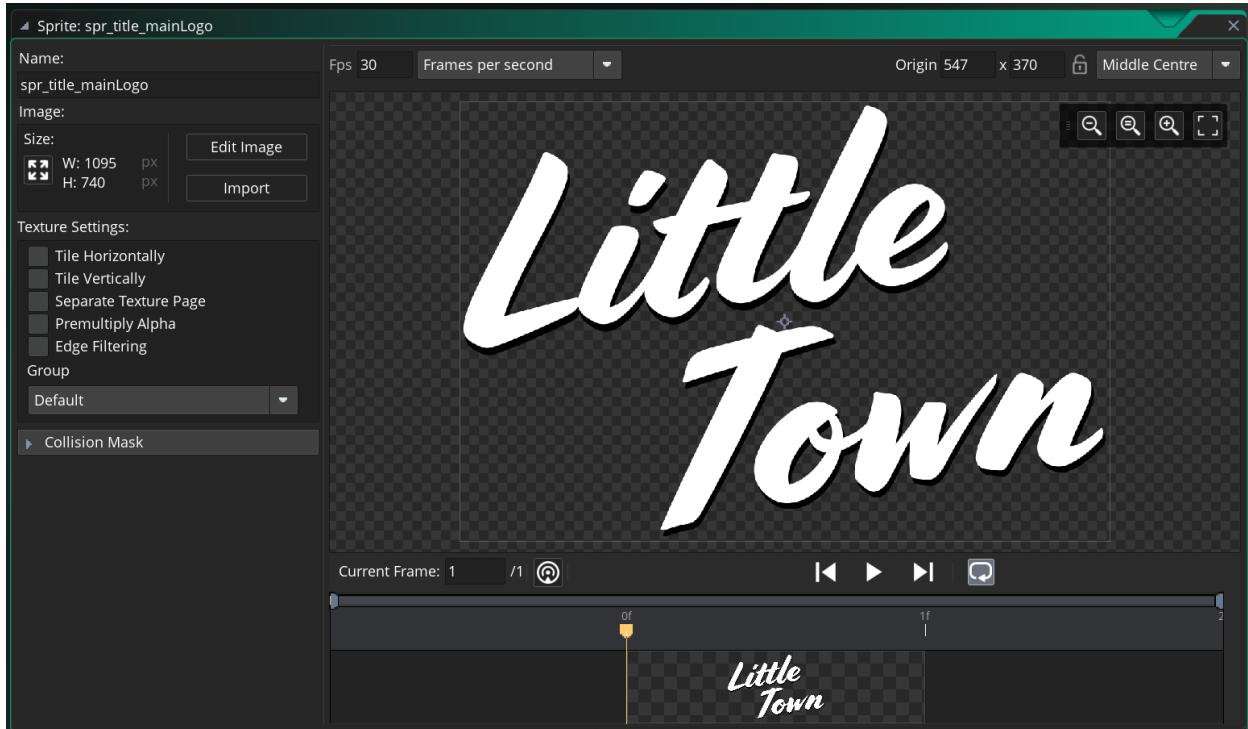
With the main gameplay loop complete, we’ll need a suitable introduction to our game — and that means a title screen!

Using File Explorer (Windows) or Finder (Mac), navigate to the Assets folder provided with this course. Go to Sprites > User Interface and drag the files there into the Asset Browser:

- `spr_bg_mainTitle`
- `spr_title_mainLogo`
- `spr_title_pressEnter`

(As always, we recommend keeping them organized in the Asset Browser — for example, in a Sprites > User Interface group.)

Open both spr_title_mainLogo and spr_title_pressEnter Sprites. Set the Origin for both to Middle Centre.



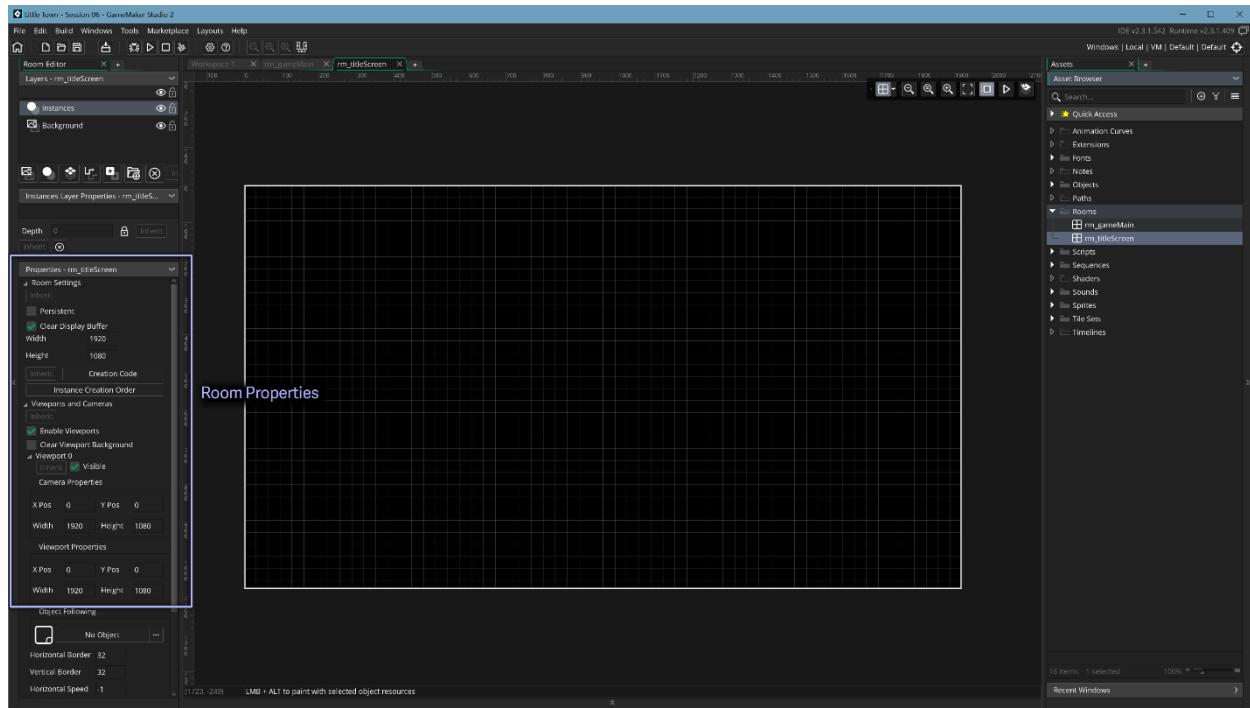
Editing spr_title_mainLogo to change its Origin.

Next, we need to create a new Room to use as a title screen.

In the Asset Browser, right-click on the Rooms group and choose Create > Room. Name this new Room rm_mainTitle.

Open rm_mainTitle if it's not open already; using the Room Editor panel, change the following properties:

- Set the Width to 1920 and the height to 1080
- Enable Viewports
- Make Viewport 0 visible
- Set the Camera width to 1920 and its height to 1080
- Set the Viewport width to 1920 and its height to 1080



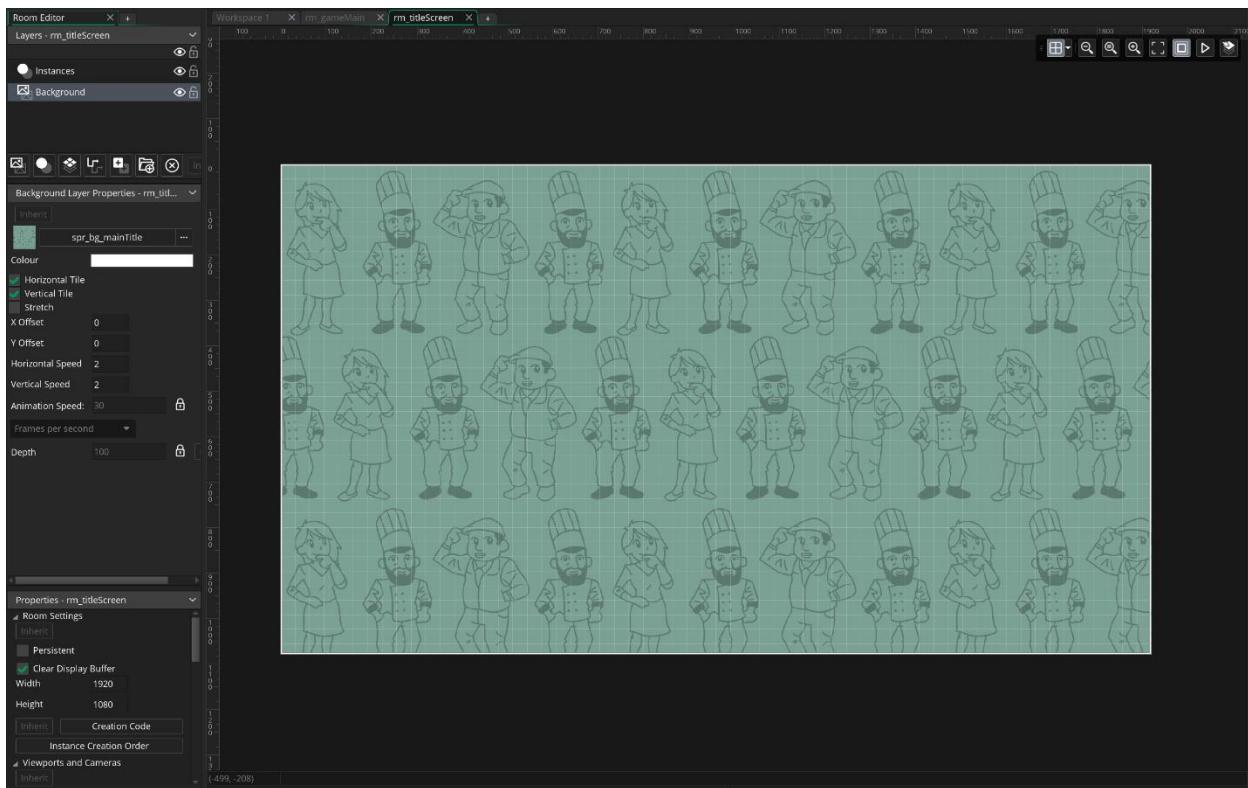
Editing our new title screen Room. The properties we need to change are all visible on the left.

Next, in the Background Layer Properties section, is a drop-down menu that currently says “No Sprite.”

Click this and choose the spr_bg_mainTitle Sprite we just imported. You’ll notice that it’s a square image, just sitting in the top-left corner of the room.

What we want to do is tile the image. So, in the Background Layer Properties section, do the following:

- Check Horizontal Tile
- Check Vertical Tile
- Change the Horizontal Speed to 2
- Change the Vertical Speed to 2



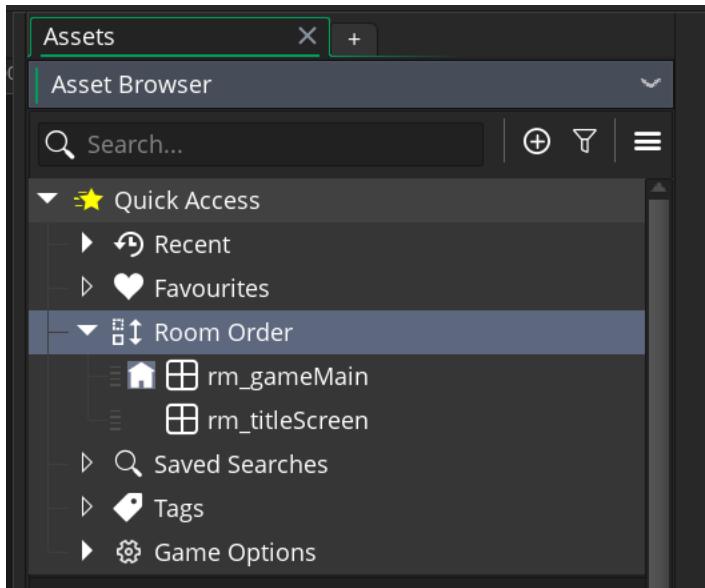
Tiling a background in the Room Editor.

9.20 Setting the Room Order

We should test our new Room to see what we've done so far in action. However, if we run the game again now, we won't see it.

That's because of our game's current *Room Order*. You might ask: if our game has multiple Rooms, how does GameMaker Studio 2 know which one to load *first*?

Look in the Asset Browser; you'll see a special section called Quick Access. Open it, and you'll see another sub-section called Room Order.

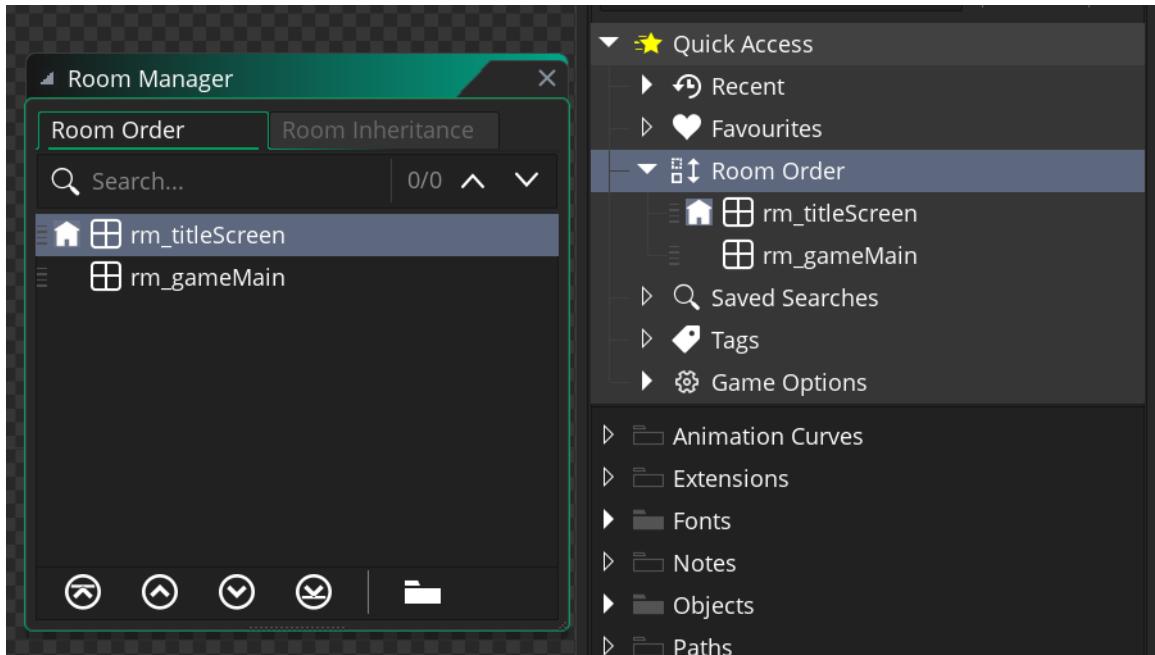


The Room Order lets you know which Room in your game will be loaded first.

In this Room Order list, you'll see every Room in your game. (We currently only have two.)

You'll notice that `rm_gameMain` has a little home icon next to it; this indicates that `rm_gameMain` will be the first Room shown to the player when our game is loaded.

To change this, drag `rm_titleScreen` above `rm_gameMain`. If `rm_titleScreen` has the home icon beside it, you know it will be the “first” Room in the game.



Setting rm_titleScreen to be the first Room the player sees. Note that if you click the home icon beside a Room, you can access the Room Manager.

With this done, run the game again to test it. You should see our new title screen Room, and the tiled background image scrolling diagonally.



The title screen has a gently scrolling patterned background.

	Tip: To change the direction and speed of the moving background, change the Horizontal Speed and Vertical Speed settings in the Room Editor for the title screen Room.
--	---

9.21 Making a title logo

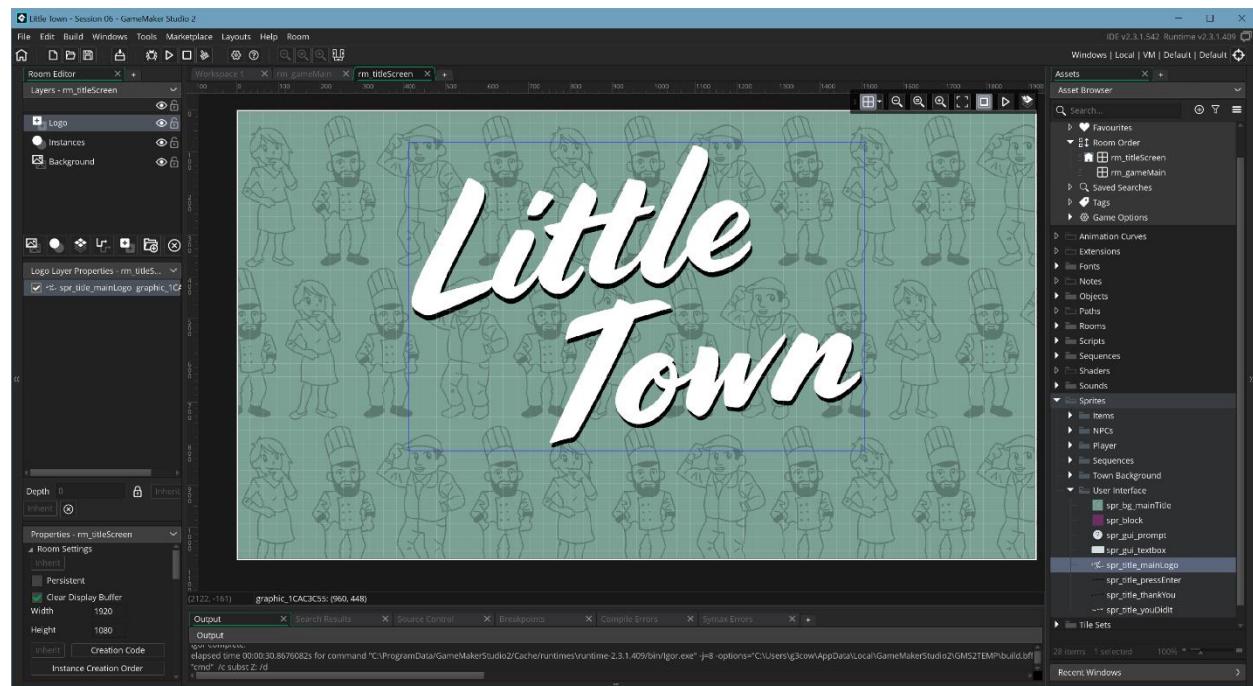
Now that we have our title screen Room set up, we need to populate it with two things: a logo and a prompt for the player to begin playing the game.

Let's start with the logo.

Open `rm_titleScreen` again. In the Layers section of the Room Editor, create a new Asset Layer.

Rename this layer Logo.

With this new layer selected, drag the `spr_title_mainLogo` Sprite from the Asset Browser into the Room. Position it however you like.



Adding the logo Sprite to the title screen Room.

9.22 Creating a prompt Sequence

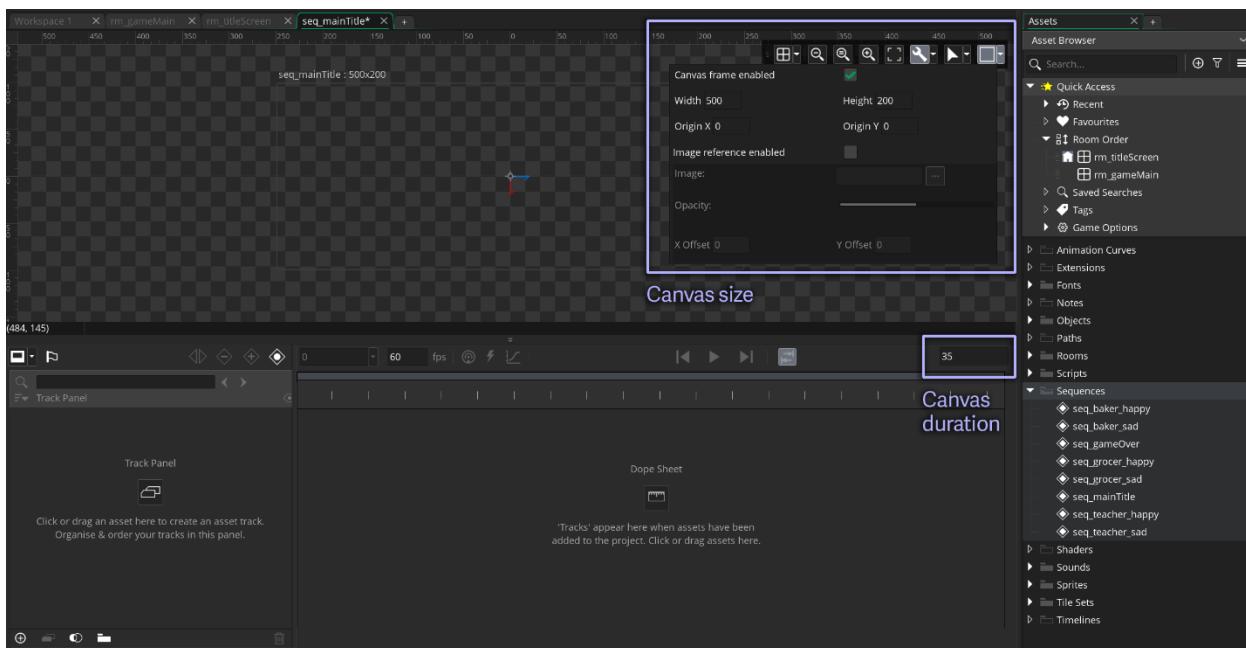
Next, we are going to create an animated prompt to encourage players to press a key in order to start the game. And thankfully, have the perfect tool for doing this: Sequences!

Right-click on the Sequences group in the Asset Browser and choose Create > Sequence. Rename this new Sequence seq_mainTitle.

This Sequence doesn't need to have a large Canvas like our previous ones, so let's do some setup.

In the Sequence Editor, make the following changes:

- Change the Canvas width to 500 and the height to 200
- Make the Sequence duration 35 frames

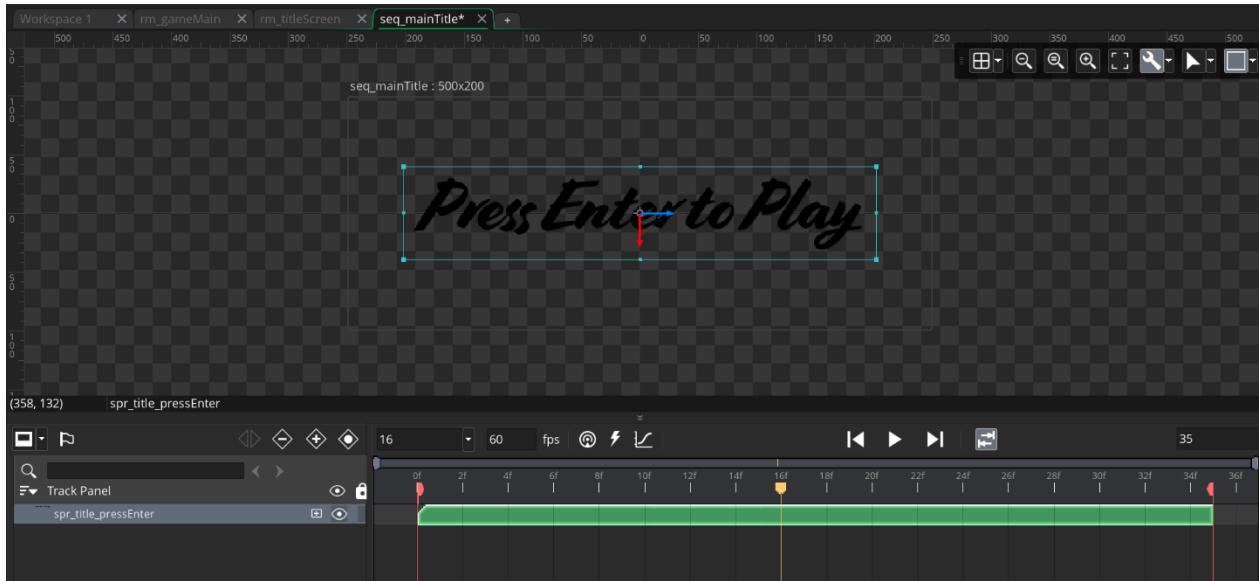


Editing the size and duration of seq_mainTitle.

(Technically speaking, we don't really need to resize the canvas, but we're doing it just to keep things tidy.)

Next, drag the spr_title_pressEnter Sprite from the Asset Browser onto the Canvas (position it in the centre).

In the Dope Sheet, make sure the asset key for the spr_title_pressEnter track spans the entire Sequence.

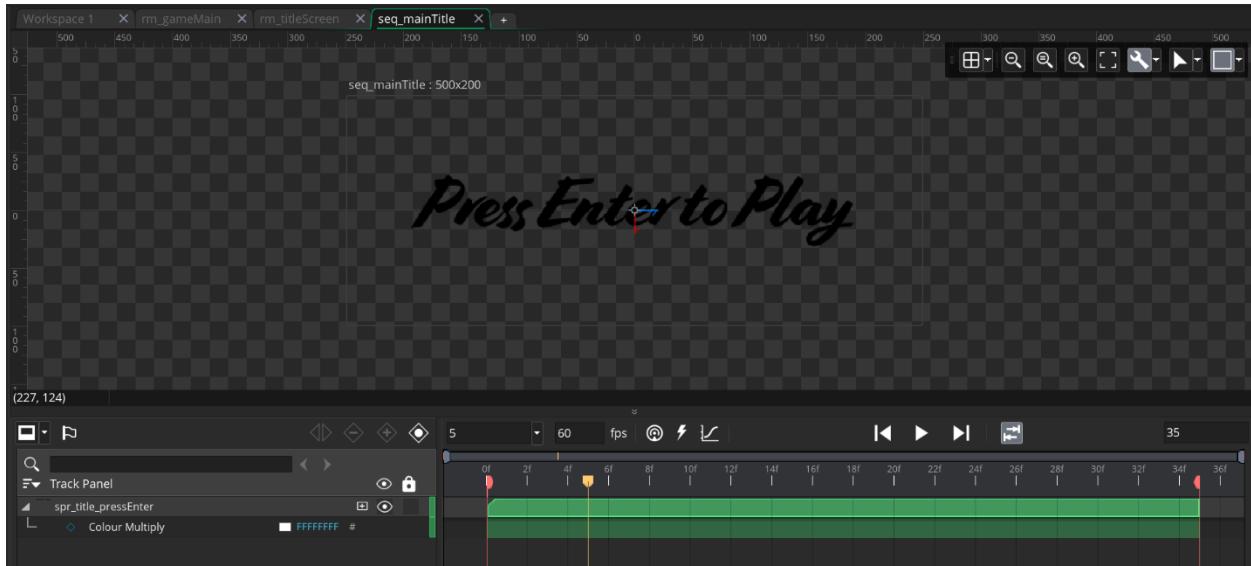


Setting up the spr_title_pressEnter asset on the Canvas and in the Dope Sheet.

Next, we want to create a simple fade effect.

Move the yellow playhead a little ways into the Sequence (say, around frame 5 or so).

Select the asset key and click the “Add parameter track” button in the Track Panel. Choose to add a Colour Multiply parameter track.



Adding a Colour Multiply parameter track to our lone asset.

Next, record a keyframe at this position.

Select the new keyframe and click the colour swatch in the Colour Multiply parameter track (if you don't see this detail, click the arrow to the left of spr_title_pressEnter in the Track Panel to reveal it).

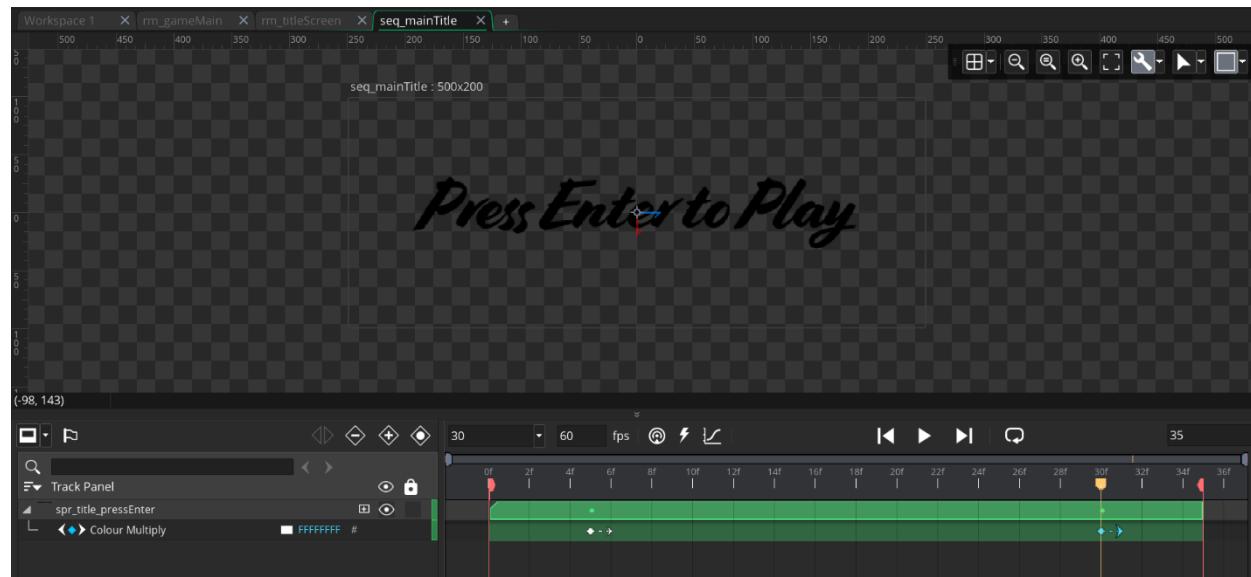
In the Colour Picker that pops up, set the Alpha channel to 0. Click "OK" when you're done. We've made the "Press Enter to Play" asset transparent.

Next, move the playhead to near the end of the timeline, but not to the last frame. We suggest around frame 30.

Repeat the process at this new position:

- Record a keyframe here
- Select the keyframe and click the colour swatch in the Colour Multiply parameter track
- In the Colour Picker, set the Alpha back to 255

If you scrub through the Sequence, or press the Play button to play it, you should see the text asset now fade from transparent to opaque.



Creating the fade effect in our Sequence.

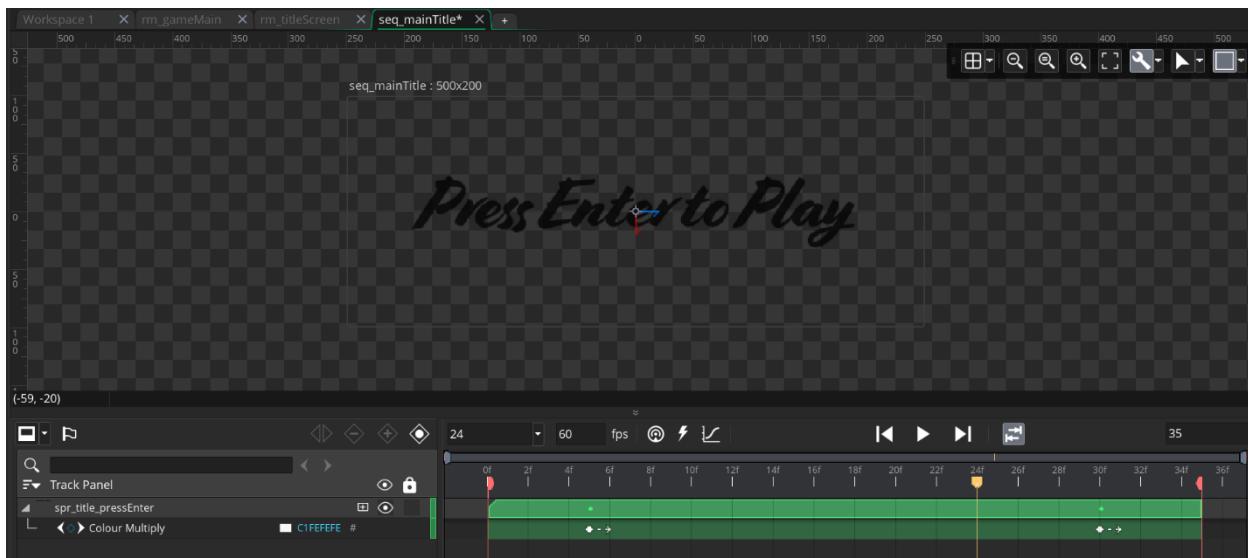
Now that we have a simple linear fade effect in place, we can do something new with it.

What we actually want is for this asset to fade in and out repeatedly, for as long as the player remains on the title screen of our game.

To do this, click the Change playback mode button (⌚) multiple times to cycle through the three playback options: play once, loop, and “ping-pong.”

Choose the “ping-pong” option (➡️➡️).

Click the Play button to preview your Sequence again, and you’ll see that it now plays forward until it reaches the end of the timeline, at which point it plays backward and then repeats.



The finished seq_mainTitle Sequence, with “ping-pong” playback option.

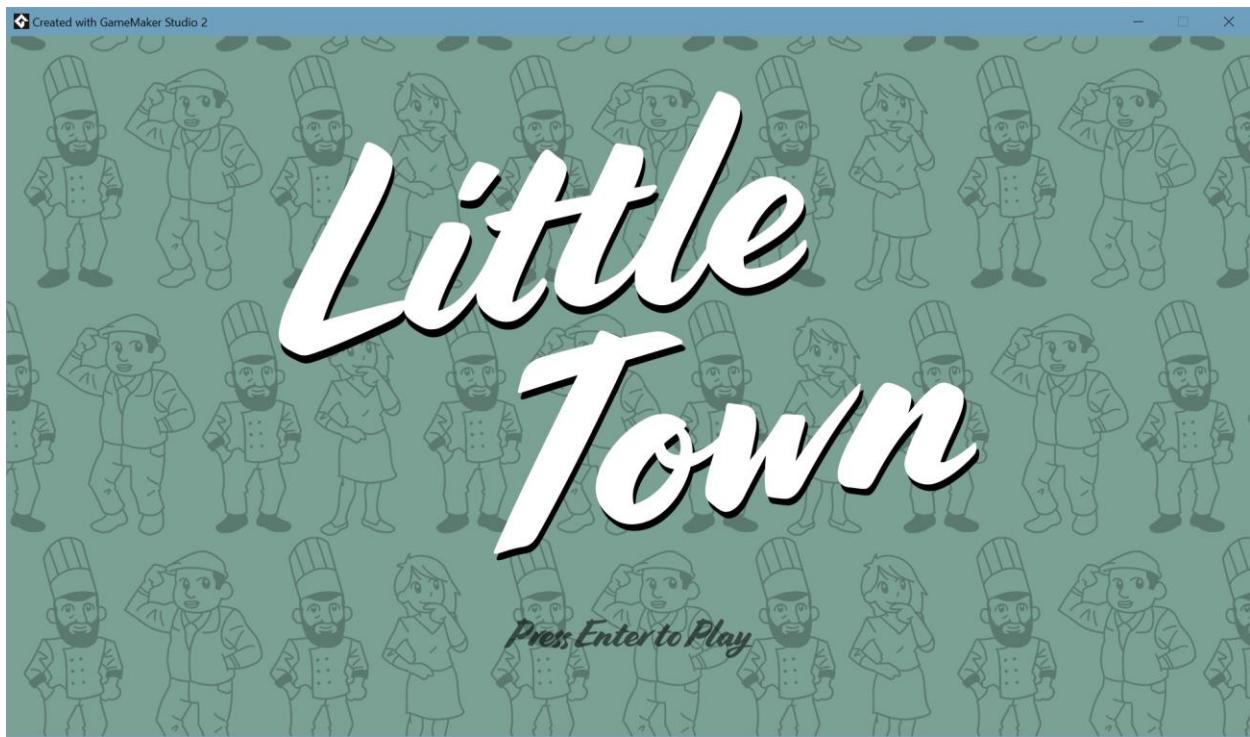
With this Sequence complete, return to rm_titleScreen.

Select the Logo Asset Layer that we created previously. Drag seq_mainTitle from the Asset Browser into the Room, on the Logo layer.

Position the prompt wherever you like in the Room.

	<p>Tip: Since the Sequence begins with an asset that is transparent, you won't actually see the “Press Enter to Play” text when you drag it into the Room. If you're finding it difficult to find or select the Sequence within the Room itself, you can click seq_mainTitle in the “Layer Properties” section to highlight it.</p>
--	--

Once you're happy with the layout of the title screen, run your game to test it. You should see the "Press Enter to Play" prompt fading in and out, while the tiled background scrolls indefinitely.



The title screen with animated background and prompt.

9.23 Adding control to the title screen

Our new title screen is looking lovely, but we still need to add functionality to it.

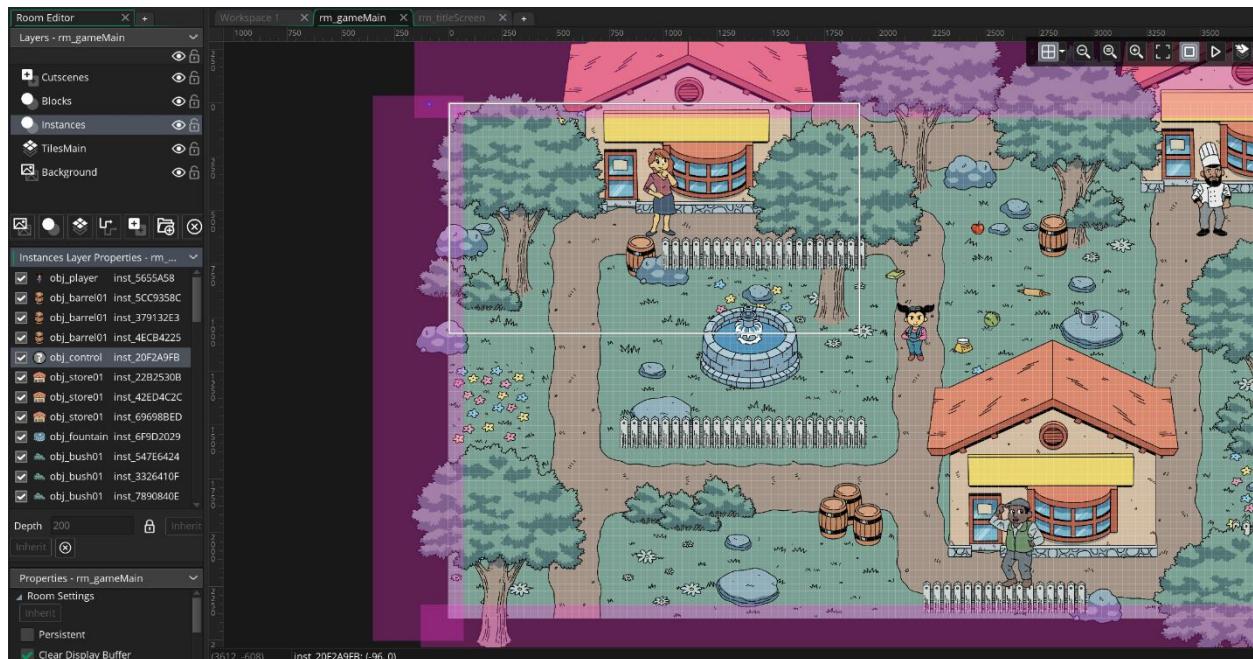
Rather than make new Objects, we're going to use one we already have: `obj_control`.

Currently, however, `obj_control` exists only in `rm_gameMain` (we placed it there back in [Using control Objects](#)). We need to change this.

Open `rm_gameMain` and select the Instances layer.

In the Instances Layer Properties section, you'll find a long list of every single instance that's been placed on that layer. If you scroll through the list, you'll find the instance of `obj_control` that we placed before.

Select this instance in the list, and press Delete to remove it from the Room.



Deleting obj_control from rm_gameMain.

Open `rm_titleScreen` again and select the Instances layer.

Drag `obj_control` from the Asset Browser into the room and place it wherever you like.



Placing obj_control in rm_mainTitle. (Here, we're placing it “outside” the Room, but it doesn't matter where it is.)

Next, open obj_control from the Asset Browser.

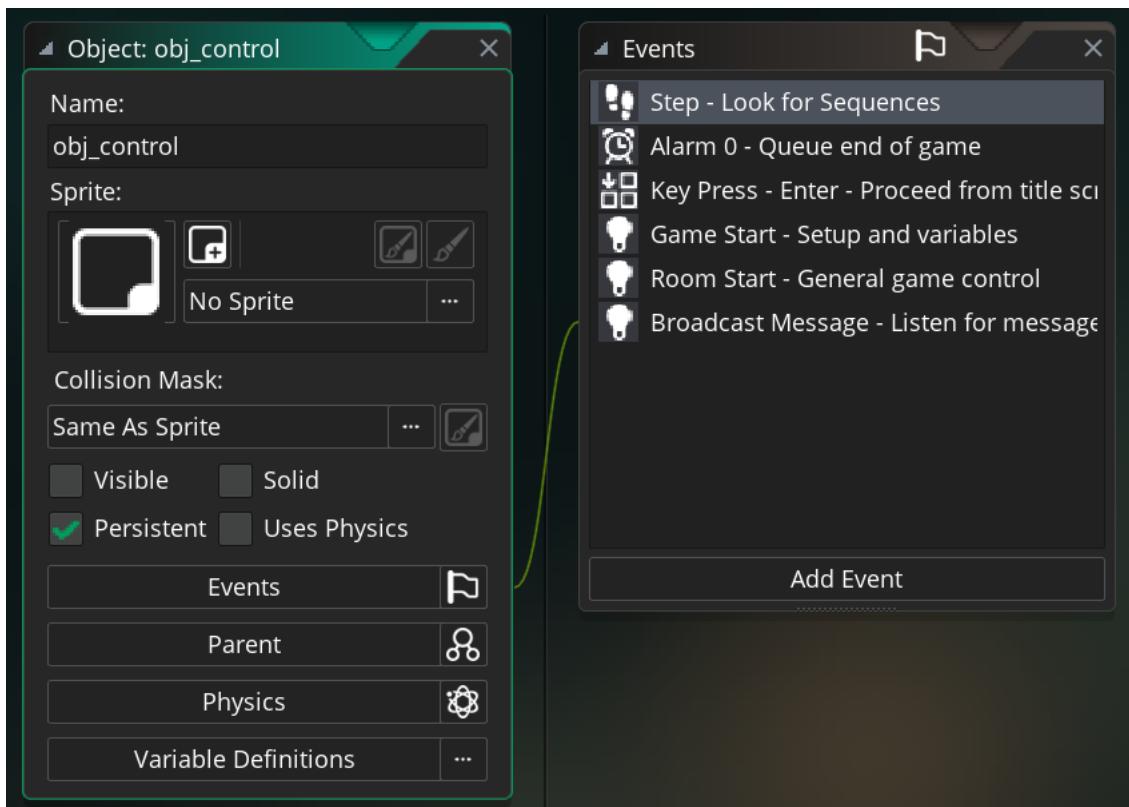
In the Object Editor, there are options such as **Visible**, **Solid**, etc. One of these options is **Persistent**.

Normally, an Object only exists within the Room it was created (or placed). If you change Rooms, any Object in the previous Room ceases to exist, as far as your game is concerned.

However, a **persistent** Object remains in the game once it is created (or once the game loads a Room that contains it). A persistent Object travels between Rooms automatically.

Since we've moved obj_control from rm_gameMain to rm_mainTitle, we need to make it **persistent** so that it will carry over to the main gameplay and continue to do all the things we've written for it to do.

So, check **Persistent** in the Object Editor.



Making obj_control a persistent Object.

Next, we're going to have obj_control check to see if the player presses the "Enter" key while on the title screen — if they do, they'll proceed to the main gameplay.

Open obj_control's Game Start Event, and update the // Game variables code block like so:

```
// Game variables
global.playerControl = true;
townBGMvolume = audio_sound_get_gain(snd_townBGM);
townAmbienceVolume = audio_sound_get_gain(snd_townAmbience);
global.gameOver = false;
global.gameStart = false;
```

Next, click on Add Event in the Object Editor and choose Key Pressed > Enter.

In this new Key Press – Enter Event, add the following code block:

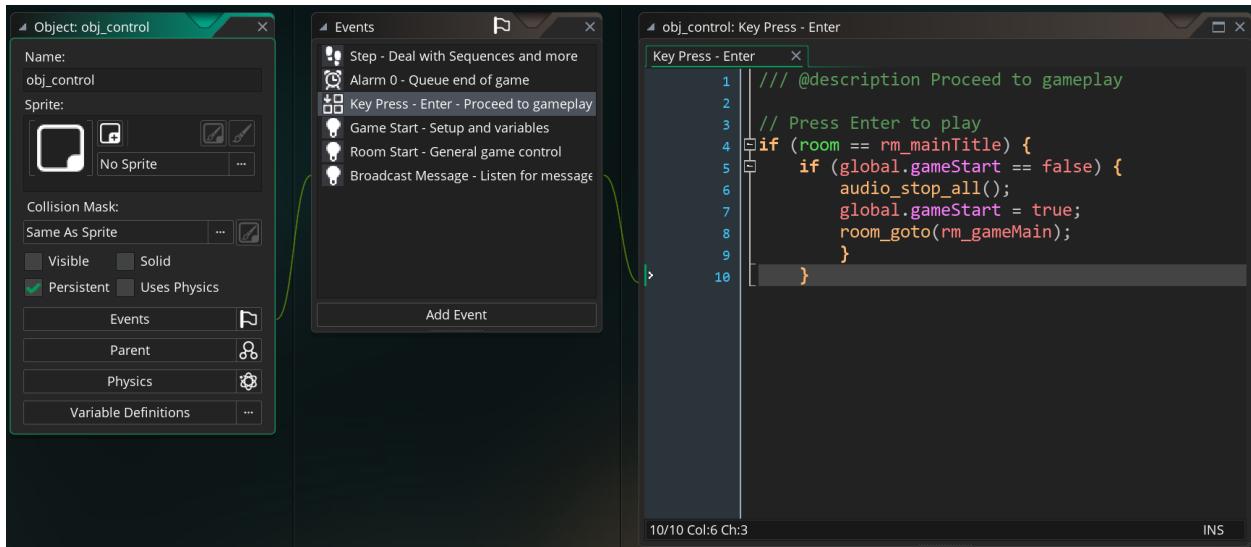
```
// Press Enter to play
if (room == rm_mainTitle) {
    if (global.gameStart == false) {
        audio_stop_all();
```

```

        global.gameStart = true;
        room_goto(rm_gameMain);
    }
}

```

In this Event, we're checking that we're on the title screen, and if so, we stop all audio that might be playing, and then we go directly to the main gameplay Room.



Adding the ability to press the Enter key to proceed to gameplay.

One last thing: let's add some music to this title screen!

We've already played music before, depending on the room we're in, so we're going to expand on that functionality.

Open obj_control's Room Start Event.

Update the switch statement in the // Play music based on Room code block like so:

```

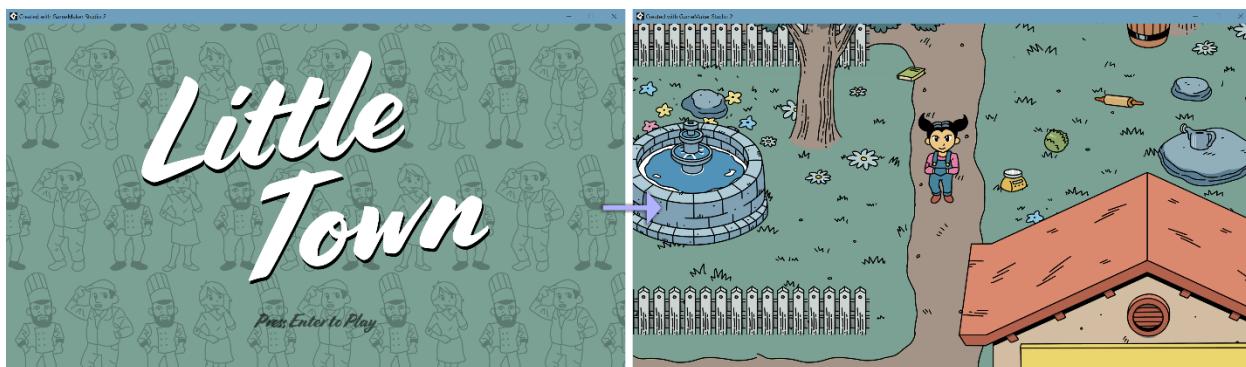
// Play music based on Room
switch room {
    case rm_mainTitle: {
        audio_play_sound(snd_seq_good01_BGM,1,1);
    }; break;
    case rm_gameMain: {
        audio_play_sound(snd_townBGM,1,1);
        audio_play_sound(snd_townAmbience,1,1);
    }; break;
}

```

This will play one of the Sequence music tracks as a title theme while you're on the title screen. (Feel free to choose a different Sound asset if you prefer.)

Run your game again to test it again. On the title screen, you should hear your music looping, and the “Press Enter to Play” asset will be gently blinking in and out.

Press the Enter key — you should transition immediately to the gameplay room, where you can begin the game as before.



Progressing to gameplay from the title screen.

And with this addition, you have in fact completed your Little Town! As always, make sure to save your project.

9.24 End of session notes

With the completion of these six sessions, you have learned an awful lot about GameMaker Studio 2. From setting up Rooms, to creating player Objects, to controlling animation, and even to designing Sequences.

But with every design tool, there is always much more to learn, and new ways to create. In the Bonus Session included with this course, you'll find some additional tricks and suggestions to keep improving your little town.

10 Bonus Session

In this session you'll find a few optional tips to improve your town, as well as some challenges that will help you learn more.

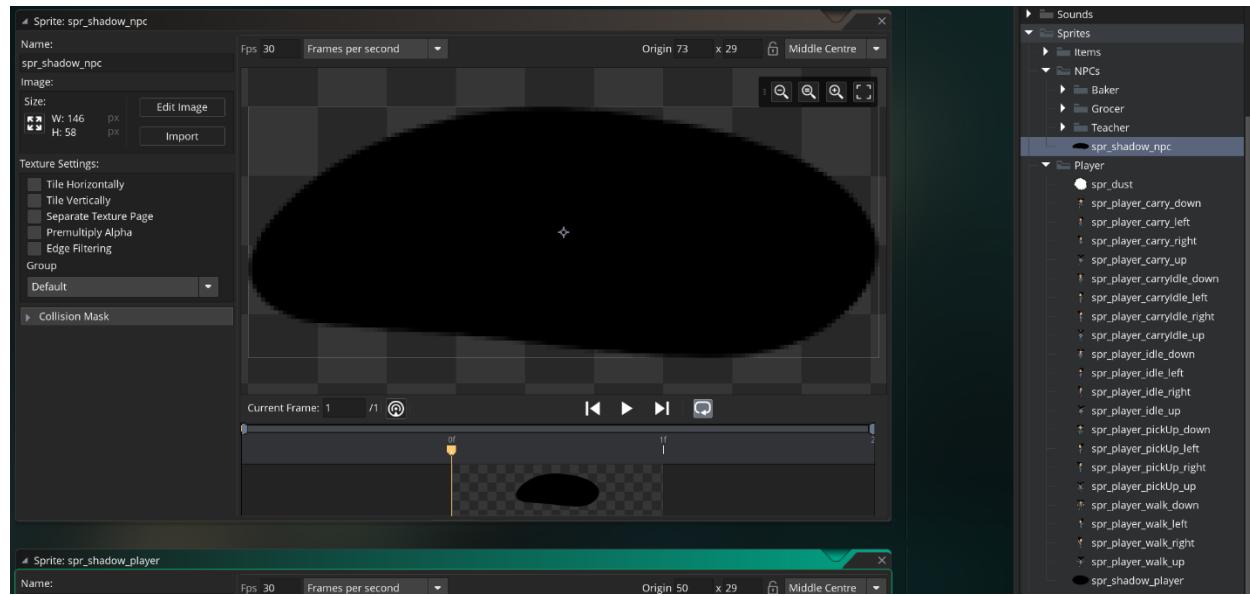
10.1 Bonus 1: Add shadows to the player

Using File Explorer (Windows) or Finder (Mac), navigate to the Assets folder provided with this course. Go to Sprites > Characters and Items and drag these two files into the Asset Browser:

- spr_shadow_npc
- spr_shadow_player

You can organize these assets if you want to within Groups.

Open both Sprites and set their Origins to Middle Centre.



Setting the Origin for both shadow Sprites and organizing each within the Asset Browser.

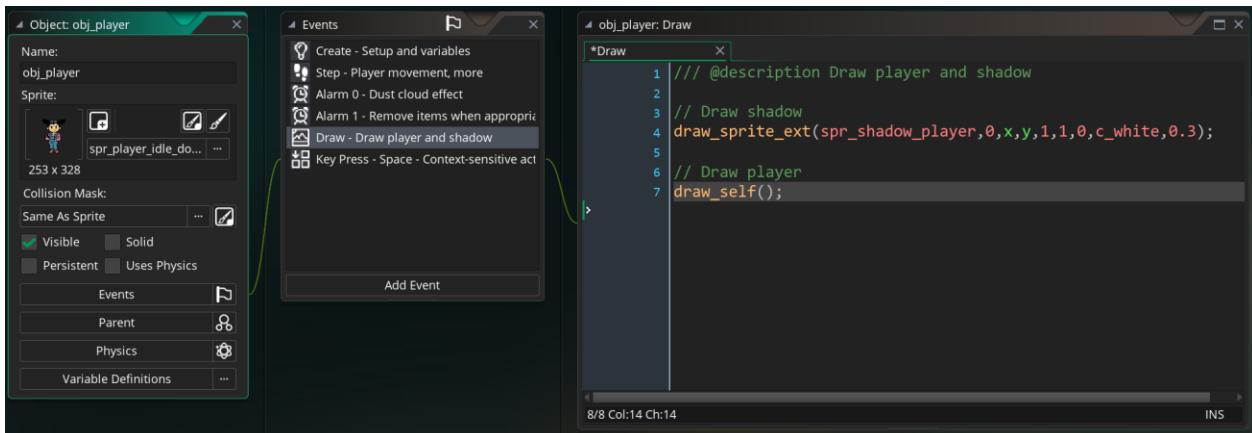
Next, open obj_player. In the Object Editor, click Add Event and choose Draw > Draw.

Back in [Drawing things manually](#), we used a Draw Event to make our textbox Object draw both the textbox sprite and the text that fits within it.

We're going to do the same thing here for our player; it will draw both the player Sprites (as it's already doing now) and the new shadow Sprite.

Open the new Draw Event and enter the following code block:

```
// Draw shadow  
draw_sprite_ext(spr_shadow_player,0,x,y,1,1,0,c_white,0.3);  
draw_set_alpha(1);  
  
// Draw player  
draw_self();
```



Drawing a shadow underneath our player.

Note in the `draw_sprite_ext()` function, we're setting the position of the shadow Sprite to be the same as the player (so it lines up with the player's feet). We're also setting the last argument in this function to `0.3` to make the sprite translucent.

And as we did before with the textbox Object, we're using `draw_self()` so we don't need to worry about any complex drawing code. The player Object will continue to correctly draw its various animations.

	<p>Tip: Remember, the order in which you draw elements in a Draw Event determines the order in which they appear in the game (from farthest away from the camera to nearest). We draw the shadow first, and then the player sprite, so that the shadow always appears under the player.</p>
--	--

When you're ready, run the game and check out that fancy new shadow!



Running around town with a translucent shadow beneath the player.

10.2 Challenge 1: add a shadow to the NPCs

Using what you just learned, can you use that new `spr_shadow_npc` Sprite to add a shadow to the NPCs? Remember, you can edit `obj_par_npc` to make a change that affects all three characters automatically.



The Baker with a translucent shadow just like the player's.

10.3 Bonus: add pick-up and put-down animations

Currently the items in the game leap up into the player's arms and plop down unceremoniously when you pick them up and put them down. But with a small bit of extra code, we can animate those items in time with the player's own animations.

Open `obj_par_item` and its `Create Event`.

Add these two lines to the `// Set my state` code block (the list of variables we made):

```
| putDownSp = 17;  
| pickUpSp = 17;
```

Next, open the `Step Event` (or `End Step Event`, if you changed it to that) and edit the `itemState.taken` and `itemState.puttingBack` cases within the `// Depth, animation` code block, like so:

```

// Depth, animation
switch myState {
    // If item is sitting on the ground
    case itemState.idle: {
        depth = -y;
        }; break;
    // If item has been taken
    case itemState.taken: {
        // Keep item lined up with player
        var _result = scr_itemPosition();
        x = _result[0];
        depth = _result[2];
        if (instance_exists(obj_player)) {
            // Animate item being picked up
            if (obj_player.myState == playerState.pickingUp) {
                // Wait until third frame of animation
                if (obj_player.image_index >= 2) {
                    if (y > _result[1]) {
                        y -= pickUpSp;
                    }
                }
            }
            // Position item while being carried
            else {
                y = _result[1];
            }
        }
    };
    break;
    case itemState.puttingBack: {
        // Animate item being put down
        if (instance_exists(obj_player) && obj_player.myState ==
playerState.puttingDown) {
            if (y < putDownY) {
                y += putDownSp;
            }
            // Reset item state after being put down
            if (y >= putDownY) {
                myState = itemState.idle;
            }
        }
    };
    break;
}

```

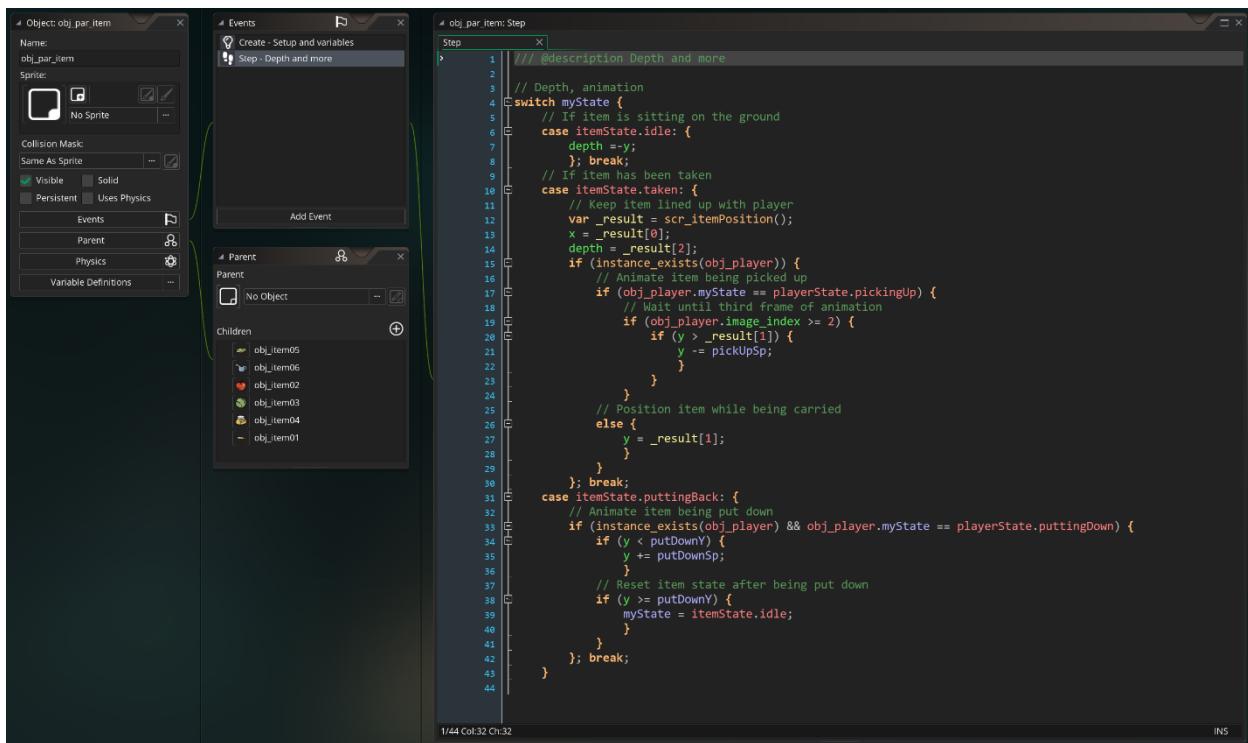
Those new variables we declared in `obj_par_item`'s Create Event are just for speed (and you can modify them if things don't look quite right to you).

In the switch case under // If item has been taken, we're simply checking if the player Object is in its “picking up” state. If so, we adjust the y position of the item until it reaches the y value returned by our `scr_itemPosition()` function. (Refer back to [Returning results from functions.](#))

If the player has finished its “picking up” state, we keep the item’s y position in check as we did before.

Below that, in the next case, we’ve replaced the previously simple code with a new // Animate item being put down code block.

In this code block, we’re also checking the player Object’s state, and if it’s “putting down,” we adjust the y position of the item down to the ground. (We set `putDownY` in `obj_player`’s Key Press – Space Event, back in [Putting the item back down.](#))

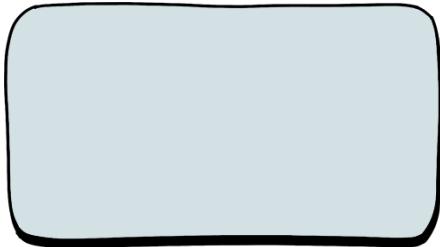


Updating `obj_par_item` to animate items being picked up and put down

With this update made, run your game again. Try picking up and putting down items and watch how they smoothly animate!

10.4 Challenge 2: redesign the prompt and textbox Sprites

The Sprites for the prompt and textbox are pretty simple; can you design new ones and implement them?



Can you imagine new Sprites for the prompt and textbox?

If you need a reminder, refer to [Setting up our textbox](#). And don't forget, these Sprites could be animated!

10.5 Challenge 3: design and add more items to the game

Currently our *Little Town* has six items in it. You'll notice that the items can seem relevant to more than one character (for example: who would appreciate an apple more? The Teacher or the Grocer?).

This makes writing hint text (as we did in [Changing what the NPCs say](#)) fun and gives us a way to get our player to think.

What other items could you add to the game to challenge the player?



What other items could you add to the game?

10.6 Challenge 4: make the game more challenging

You can change what each of the three NPCs says for their myText Variable Definition to give clearer or more difficult hints as to which item they want. You can also hide the six items in more devious locations, or design your town to make them harder to spot. How else can you challenge your players?

10.7 Export your game

We didn't go over exporting your game to give to others due to some differences depending on platforms. However, to get details on this:

1. Go to Help > Open Manual
2. In the GameMaker Studio 2 manual, search for "Creating A Final Executable Package"

10.8 End of Session

As always, save your project, and whatever you do — keep making awesome GameMaker Studio 2 games!