GameMaker Cookbook

GameMaker is designed to allow its users to develop games without having to learn any of the complex programming languages such as C++ or Java. This book teaches you to harness GameMaker Studio's full potential and takes your game development to new heights. It begins by covering the basics and lays a solid foundation for advanced GameMaker concepts. Moving on, it covers topics such as controls, physics, and advanced movement, employing a strategic approach to the learning curve. The book concludes by providing insights into complex concepts such as the GUI, menus, save system, lighting, particles, and VFX. By the end of the book, you will be able to design captivating games using GameMaker Studio.

What this book will do for you...

- Set up player control schemes of various types, such as touch controls and a gamepad
- Create your own physics sandbox
- Get accustomed to advanced player movement
- Implement music and 3D sound in your games
- Utilize GameMaker’s GUI layers to create exciting games
- Generate adjustable game settings and save systems
- Add depth to your game with lighting and special effects

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions to solve problems efficiently
- Clear explanations of what you did
- Solutions that can be applied to solve real-world problems

Over 50 hands-on recipes to help you build exhilarating games using the robust GameMaker system
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 6 'It's All GUI! - Creating Graphical User Interface and Menus'
- A synopsis of the book’s content
- More information on GameMaker Cookbook
Brandon Gardiner is a game developer and designer living out his love of video games. Though he started his foray into games through 3D art and level design, he always kept a notebook of ideas for games and game mechanics he wished to see. When he discovered GameMaker: Studio, he found that he could bring these ideas to life.

He is a graduate of the first iteration of the game development program at Toronto's George Brown College. In college, he worked as an artist and designer on several game projects for outside companies, including tie-ins for children's television shows and educational titles. After graduating, he founded MechaBee Studios where, being the sole developer of mobile and PC games, he is a jack of all trades.

He also writes a blog at http://www.gamemakerhq.com, through which he hopes to build a resource for other independent developers. He lives with his wife in Toronto where he is a veteran of the annual Toronto Game Jam (tojam.ca), an active member of the International Game Developers Association.
Since 1999, GameMaker: Studio has enabled fledgling and experienced developers alike to help create video games quickly and easily. In the years since its initial release (as a program called Animo), GameMaker has grown and evolved into a viable commercial game engine that has helped developers release games for PC, Mac, mobile devices, and even the Microsoft Xbox and Sony PlayStation family of consoles.

During its progression, GameMaker has changed in many ways, adding new features and abilities to be taken advantage of by developers around the world. Though these features may be out of reach for new users, the goal of this book is to familiarize beginners and bring such elements within reach. This book is not just a how-to manual, it is an answer to the question "What can GameMaker do for me?"

By reading this book and completing the recipes therein, you will gain a greater understanding of GameMaker's capabilities as well as start them on the path to unlock their full potential.

**What this book covers**

*Chapter 1, Game Plan – Creating Basic Gameplay*, shows you how to create the basic elements of a game.

*Chapter 2, It's Under Control – Exploring Various Control Schemes*, helps you figure out how to implement user controls for a variety of devices.

*Chapter 3, Let's Move It – Advanced Movement and Layout*, teaches you more advanced ways to move players and non-player characters alike.

*Chapter 4, Let’s Get Physical – Using GameMaker's Physics System*, introduces you to the physics system and demonstrates how GameMaker handles gravity, friction, and so on. You will also learn how to implement this system to make more realistic games.

*Chapter 5, Now Hear This! – Music and Sound Effects*, helps you pick up the ins and outs of GameMaker's audio system.
Preface

Chapter 6, It's All GUI! – Creating Graphical User Interface and Menus, discusses the most important element of the Graphical User Interface (GUI).

Chapter 7, Saving the Day – Saving Game Data, discusses how GameMaker handles how to save data as well as its various uses.

Chapter 8, Light 'em up! – Enhancing Your Game with Lighting Techniques, helps you understand how GameMaker's surfaces can add lighting effects to your game.

Chapter 9, Particle Man, Particle Man – Adding Polish to Your Game with Visual Effects and Particles, shows you how GameMaker handles particles and how you can draw players in with simple effects.

Chapter 10, Hello, World – Creating New Dimensions of Play Through Networking, teaches you how to create multiplayer games with GameMaker's networking capabilities.
In this chapter, we’ll cover the following topics:

- Setting up a basic HUD with code
- Making your HUD scalable
- Using the GUI layer in full screen mode with views
- Adding a title screen
- Creating splash pages
- Adding a game over screen

Introduction

Of all the game terms that are fun to say, GUI (often pronounced "gooey") is probably the most important. Graphical User Interface (GUI) refers to any on-screen visual cues that allow the user to control the software directly. This is usually associated with images and texts that provide information and points of interaction, such as menus and icons.
GUIs are most commonly associated with software applications, as opposed to video games. While video games are a form of software, many prefer this differentiation. In video games, a more common term for display and menu items would be **HUD** (also fun to say), which stands for **Heads-Up Display**. While these two terms mean largely the same thing, HUD is viewed by most as the preferred term. Having said this, GameMaker uses GUI to describe such systems, so we'll stick to this. With these formalities out of the way, let's take a look at how GameMaker handles menus, game screens, and the GUI in general.

### GUI basics

In *Chapter 1, Game Plan – Creating Basic Gameplay*, we made a basic GUI using GameMaker's drag and drop interface. This works just fine but, by coding it yourself, you can fine-tune this interface to your needs.

- Setting up a basic HUD with code
- Making your HUD scalable
- Using the GUI layer in full screen mode

### Game screens and menus

Having a game that starts as soon as you open it is fine for testing or for a proof of concept but it's not ideal for making an actual game product. We'll take a look at how we can implement various game screens:

- Adding a title screen
- Creating splash pages
- Adding a "Game Over" screen

### Setting up a basic HUD with code

Have you ever played a game and stopped to think, "Gee, I wish I knew what my score was!" No? That's probably because your score, health, and other important pieces of information are presented to you on the screen at any given moment. Now, this isn't the case for all games; of course, but it's true for classic and arcade games. *Pac-Man* and *Mario* will always tell you how many lives you have left with because it's important to have this information. Let's take a look at how we can display such information by making a simple game.
Getting ready

You'll need a few things before we can really begin. Firstly, you'll need to create a room called rm_game. You'll also need four objects: obj_display, obj_gameControl, obj_base, and obj_enemy. The last two will require sprites. You'll also need to set up a new font. This does not need to be a custom font designed by you. Simply, click on the Create a font button on the toolbar, select the font you wish to use to display health, lives, and scores, and customize the style and size.

Name the font fnt_HUD in order to reference it in your code. Once this is complete, you're ready to go.

How to do it

1. Open obj_display and add a Create event.
2. Drag a code block to the Actions box and enter the following code:
   ```
   global.playerHealth = 100;
   global.playerLives = 3;
   global.playerScore = 0;
   global.gameOver = false;
   draw_set_font(fnt_HUD);
   ```
3. Add a **DrawGUI** event (under **Draw**) and place a code block in the **Actions** box.

4. Enter this code:

   ```
   draw_text(32, 32, string("Health: ") + string(global.playerHealth));
   draw_text(32, 64, string("Lives: ") + string(global.playerLives));
   scoreString = string(global.playerScore);
   stringLength = 6 - string_length(scoreString);
   stringTemp = "";
   while(stringLength > 0)
   {
       stringTemp += "0";
       stringLength -= 1;
   }
   scoreString = stringTemp + scoreString;
   draw_text(room_width - 96, 32, scoreString);
   ```

5. In **obj_gameControl**, add a **Create** event.

6. Place the following code in a code block:

   ```
   global.enemy = 0;
   global.enemyMax = 10;
   alarm[0] = 30;
   instance_create(room_width/2, room_height/2, obj_base);
   ```

7. Add an event to **Alarm 0** using the following code:

   ```
   spawn_point = choose(0, 1, 2, 3)
   if (spawn_point = 0)
   {
       instance_create(-64, random(room_height), obj_enemy);
   }
   if (spawn_point = 1)
   {
       instance_create((room_width+64), random(room_height), obj_enemy);
   }
   if (spawn_point = 2)
   {
       instance_create(random(room_width), -64, obj_enemy);
   }
   if (spawn_point = 3)
   {
       instance_create(random(room_width), (room_height+64), obj_enemy);
   }
   alarm[0] = 45;
   ```
8. Add a Step event and drag a code block to the Actions box.

9. Enter the following code:
   ```
   if global.playerHealth <= 0
   {
       global.playerLives -= 1;
       global.playerHealth = 100;
   }
   if global.playerLives <= 0
   {
       global.gameOver = true;
   }
   if global.gameOver
   {
       game_restart();
   }
   ```

10. In `obj_enemy`, add a Create event to the following code:
    ```
    move_towards_point(obj_base.x, obj_base.y, 5);
    ```

11. Add a Step event and place a code block in the Actions box.

12. Enter the following code:
    ```
    if mouse_check_button_pressed(mb_left)
    {
        if instance_position(mouse_x, mouse_y, obj_enemy)
        {
            global.playerScore += 10;
            instance_destroy();
        }
    }
    ```

13. Add a Collision event to `obj_base` and use the following code:
    ```
    global.playerHealth -= 25;
    instance_destroy();
    ```

Now, simply place one instance each of `obj_display` and `obj_gameControl` anywhere in the room and you're set. There is no need to place an instance of `obj_base`, as this will be handled by the code in order to place it in the exact center of the room. If you wish to place the base elsewhere in the room or in multiple places, simply adjust the code or erase it and place the objects manually. When you test the game, simply click on the enemies before they reach your base and collect points. As enemies collide with your base, it will lose health. When it runs out of health, you'll lose a life.
How it works

This simple game demonstrates the basics of GameMaker's GUI layer and the **DrawGUI** event. **DrawGUI** renders to the application surface outside any draw events. Essentially, if you use **Draw** to render sprites or textures within the game, regardless of the depth, **DrawGUI** will render items on top of them. Not only that, it bases positions on the (0, 0) coordinates of the view, as opposed to the room. This makes it ideal for your HUD as it will follow the view instead staying put within the room. Even though **DrawGUI** events will supersede standard **Draw** events, you can still use layer items within **DrawGUI**, using the depth value as you would, normally.

In the **Create** event of `obj_display`, we used **draw_set_font** to change the font used when displaying any string we code. This prevents you from relying on GameMaker's default font and font settings and allows you to use a custom font, if you choose to create one. You'll notice that, when we set the font, it is used across any strings displayed on the screen. Do you want to use multiple fonts? You'll need to use multiple control objects and set a different font for each of them. If your game design calls for multiple fonts or styles, make sure to use descriptive naming conventions to avoid confusion.

In order to display health, life, and score values on the screen, we first need to convert them into strings. GameMaker must be told exactly what you wish to display, but it can display the value of any variable as a string alphanumerically. Since we want to display titles (that is, health, lives, and scores) as well as their values, we have to tell GameMaker to do so using the **+** sign. This allows us to display multiple titles or values without having to hardcode the positions of each after a lot of thought and confusion. You may also have noticed the use of the classic arcade scoring system with a finite number of potential digits. This was done on purpose for aesthetic reasons. By assigning the score value, the length (in digits) of that value and strings of the number 0, we were able to replace the zeros in these spaces with the actual score. It's a small price to pay for the layout you want. When your base runs out of lives, the game restarts, as we've seen in recipes from previous chapters. This isn't ideal from a game design perspective, but we'll get into remedies for this at a later point in this chapter.

These are some of the basics you can accomplish in terms of your user interface, but these simple functions coupled with custom graphics and your own creativity can make a great user experience.

Making your HUD scalable

Have you ever played a game that didn't quite fit your screen? It's frustrating, isn't it? In a perfect world, your game would work perfectly on any device. Since that's not the case, however, you'll need to make some adjustments if you want to create a game that will fit at least most devices. Let's see what it takes to scale your game to fit screens of different sizes and ratios.
Getting ready

For this recipe, we’ll use the game that we created previously. Open the project file and follow the given steps, but make sure Allow the player to resize the game window is activated and Keep aspect ratio is selected under Global Game Settings.

How to do it

1. Open obj_display.
2. Under the DrawGUI event, open the code block.
3. Enter the following code above the existing code:

```c
display_w = 1024;
display_h = 768;
ratio = display_get_width()/display_get_height();
if ratio > 1
```
adjust_w = display_h * ratio;
adjust_h = display_h;
display_set_gui_size(display_h * ratio, display_h);
}
else
{
    adjust_w = display_w;
    adjust_h = display_w/ratio;
    display_set_gui_size(display_w, display_w/ratio);
}

4. Replace the last line of code with this code:

    draw_text(adjust_w - 96, 32, scoreString);

Once this is done, test the game and try resizing the player window.

**How it works**

When resizing the test window, you will notice a couple of things: Firstly, you will notice that the game's aspect ratio remains the same even when you change the size and shape of the player window. By selecting **Keep aspect ratio** in **Global Game Settings**, you're telling GameMaker to do exactly that. Regardless of the device you're using or the aspect ratio of the screen on which you're displaying your game, GameMaker will keep the game's aspect ratio intact. Conversely, if you select **Full scale**, GameMaker will squash and stretch the image to fit the display.

Secondly, you will notice that, as you resize the window and game itself, the GUI remains in its place in conjunction with the play area. To achieve this, we must find the dimensions of the display being used and use this information to resize the play area to fit the screen while maintaining the game's aspect ratio. The aspect ratio is calculated by taking the width of the window or displaying and dividing it by its height. This value is then used to find the adjusted width (for landscape mode) or height (for portrait mode). To find the new width, we need to multiply the original height by the aspect ratio; in order to find the new height, we need to divide the original width by the aspect ratio. These new values are used to set the GUI dimensions using `display_set_gui_size()`, as mentioned earlier. In order to make GameMaker alter the proper dimension, we must check the value of the ratio that we calculated. If the ratio is greater than 1, then the display's width is greater than its height and the display is therefore in landscape mode.
We can then use the original height and find the adjusted width by multiplying the original height by the value of the ratio, thus ensuring that the GUI will retain the original aspect ratio but with new dimensions. If the ratio is one or less, the width is less than the height, which means that the display is in portrait mode. Here, we keep the width and find the adjusted height by dividing the original width by the calculated value of the ratio, again keeping the aspect ratio intact while finding the new height.

Once we've found the adjusted width and height, we can then use them to draw our GUI. In our code, we used the adjusted width to place the score counter on the right-hand side of the screen, relative to the width of the display. Whenever the dimensions of the display area are altered, their positions are recalculated and the score is moved proportionately.

### Using the GUI layer in full screen mode with views

When resizing the display, in order to keep the aspect ratio of the game screen (or application surface) intact, GameMaker will employ a letterbox to fill the rest of the screen. These black bars may be appropriate for your particular game, but in general they are not ideal; the black bars represent potentially wasted space. The letterbox is likely to be present when keeping the game's aspect ratio, unless the dimensions of the screen happen to match that of the room you created. For this reason, it is a good idea to employ views and allow GameMaker to change the size of these views depending on the display being used. Let's take a look at incorporating views for use in full screen mode in GameMaker.

### Getting ready

Once again, we'll use the same game project we've been working with in this chapter. However, in order to ensure that every line of code ends up where it belongs, it will be easier to follow along from start to finish. Before we begin, it's important to know that, when adapting the view to fit a display (such as various mobile devices and tablets), unless the screen is exactly the same size as the room being displayed, there is a likelihood that the viewable area will either expand or contract. In order to accommodate this, it's good practice to use views that are slightly smaller than the rooms themselves.
To accomplish this, you need to allow the use of views, shrink the view to a smaller size (preferably with the same aspect ratio), and center it within the room.

**How to do it**

1. In `rm_game`, under the **Settings** tab, click on the **Creation** code button.

2. Enter the following code:
   ```
   display_w = 1024;
   display_h = 768;
   display_wMax = display_get_width();
   display_hMax = display_get_height();
   ratio = display_get_width()/display_get_height();
   if (display_wMax < display_hMax) {
   ```
view_w = min(display_w, display_wMax);
view_h = view_w/ratio;
}
else
{
    view_h = min(display_h, display_hMax);
    view_w = view_h*ratio;
}
view_wview[0] = floor(view_w);
view_hview[0] = floor(view_h);
view_wport[0] = display_wMax;
view_hport[0] = display_hMax;
surface_resize(application_surface, view_wview[0], view_hview[0]);

3. In obj_display, add a Create event and drag a code block to the Actions box.

4. Enter the following code:
   global.playerHealth = 100;
global.playerLives = 3;
global.playerScore = 0;
global.gameOver = false;
draw_set_font(fnt_HUD);

5. Add a DrawGUI event and enter the following code in a code block:
   draw_text(32, 32, string("Health: ") + string(global.playerHealth));
   draw_text(32, 64, string("Lives: ") + string(global.playerLives));
   scoreString = string(global.playerScore);
   stringLength = 6 - string_length(scoreString);
   stringTemp = ";
   while(stringLength > 0)
   {
       stringTemp += "0";
       stringLength -= 1;
   }
   scoreString = stringTemp + scoreString;
   draw_text(display_get_gui_width() - 128, 32, scoreString);

6. In obj_gameControl, add a Create event to the following code and place a code block in the Actions box:
   global.enemy = 0;
global.enemyMax = 10;
alarm[0] = 30;
instance_create(room_width/2, room_height/2, obj_base);
7. Add an event to **Alarm[0]** and drag a code block to the **Actions** box.

8. Enter this code:

   spawn_point = choose(0, 1, 2, 3)
   if (spawn_point = 0)
   {
       instance_create(-64, random(room_height), obj_enemy);
   }
   if (spawn_point = 1)
   {
       instance_create((room_width+64), random(room_height), obj_enemy);
   }
   if (spawn_point = 2)
   {
       instance_create(random(room_width), -64, obj_enemy);
   }
   if (spawn_point = 3)
   {
       instance_create(random(room_width), (room_height+64), obj_enemy);
   }
   alarm[0] = 45;

9. Add a **Step** event to the following code:

   if global.playerHealth <= 0
   {
       global.playerLives -= 1;
       global.playerHealth = 100;
   }
   if global.playerLives <= 0 {
       global.gameOver = true;
   }
   if global.gameOver
   {
       game_restart();
   }
10. In `obj_enemy`, add a **Create** event and place a code block in the **Actions** box.

11. Enter the following code:

   ```
   move_towards_point(obj_base.x, obj_base.y, 5);
   ```

12. Add a **Step** event.

13. Drag a code block to the **Actions** box and enter the following code:

   ```
   if mouse_check_button_pressed(mb_left)
   {
     if instance_position(mouse_x, mouse_y, obj_enemy)
     {
       global.playerScore += 10;
       instance_destroy();
     }
   }
   ```

14. Add a **Collision** event to `obj_base`.

15. Place a code block in the **Actions** box and enter the following code:

   ```
   global.playerHealth -= 25;
   instance_destroy();
   ```

You can now test the game and play it by clicking on the enemies before they reach the base, but make sure to do this either on a device or by entering full screen mode (Alt + Enter) when you are testing on a PC.

**How it works**

Much of the code in this recipe works as it did in previous recipes; the game has enemies approaching your base and you must click on them to destroy them before they reach it. The score and player information are still displayed on the GUI layer, though in a slightly different way, thanks to some changes made to the display code, which is now housed within the room creation code.

As mentioned earlier, this code is used to allow a full screen gameplay for a variety of devices with varying aspect ratios. We must first establish the base dimensions of the game. In this case, we have a play area that is 1024 px wide by 768 px high. This information is used in conjunction with the maximum width and height of the play area, which are the dimensions of the display itself. We gather this information using the `display_get_width/height` functions. The width of the display is then divided by its height in order to get its aspect ratio, which is used in several places to alter the dimensions of the view and play area. As mentioned earlier, GameMaker must first determine whether the device is in landscape or portrait mode; it will determine which dimension to alter based on this outcome. We use the `min()` function to determine the lowest of a given set of values in order to scale the view.
In the case of a portrait view, the view width becomes the minimum value between the base width (the one we established) and the maximum width (the actual width of the display). This value is then divided by the aspect ratio value to determine the view's height. Conversely, in the case of a landscape display, we can find the view height by calculating the minimum value between the base height and maximum height, and then using this value to find the new view width by multiplying the view height by the aspect ratio. Once the view width and height are found, we can convert them into actual view dimensions using the floor (the given number rounded down to the nearest integer) of the returned view width and height. The view port is the area of the display where the view will be projected or drawn. Once the view is fully established, the application surface (the area to which all visuals in the game are to be drawn) is resized to match.

**There's more**

By creating a room larger than the view, you can be certain that, when the application surface is resized, no area will be drawn beyond what you created in the room. For example, my laptop screen has a wider aspect ratio than 1024 x 768. In fact, it's closer to 1280 x 768. So, when I run this code and enter full screen mode, GameMaker resizes the application surface and draws an area beyond the size of the room I initially created, leaving a blank area that was technically part of the game. This is also the reason why, when releasing a mobile game, testing on multiple devices is very important. If your game doesn't display properly on a popular device, you probably need to alter the room size as well as the views.

**Adding a title screen**

Of all the games you've ever played in your life, how many of them have jumped immediately to gameplay? Your answer is likely less than a few, if any. Any game that I can remember playing, personally, has had at least a title screen, if not several splash screens indicating the studios and publishers involved in the game's creation. Now, I understand why. Jumping right into gameplay without context can throw you into the mix before you're ready, causing confusion. I mean that you likely know what game you're playing at the time, but a title screen can at least set the mood or give you a clue as to what's about to happen when you start the game. In addition to this, title screens often house, or at least lead to, the main menu in which you can customize your experience, but we'll dive into this in the next chapter. For now, let's take a look at how to create a title screen.
Getting ready

Again, we will continue to use the game project in this chapter. This recipe will be simple, but you'll need a few things before we begin. We're going to make a title screen and main menu. You'll need a background image and sprites for the game's title (which you can assign to obj_title) and menu selections such as settings, back, play, and quit. You'll also need objects (with this naming scheme: obj_btn_buttonName) for each of these items, as well as obj_titleControl and obj_settings. You'll also need two new rooms called rm_title and rm_settings, and the room creation code from the previous recipe should be the same across every room. Since GameMaker starts in the first room listed, it's important that rm_title is first listed in the asset explorer.

How to do it

1. Open obj_titleControl and add a Create event.
2. Place a code block in the Actions box and add the following code:
   
   ```gml
   instance_create(room_width/2, 280, obj_title);
   instance_create(room_width/2, 450, obj_btn_play);
   instance_create(room_width/2, 495, obj_btn_settings);
   instance_create(room_width/2, 540, obj_btn_quit);
   ```
3. In `obj_settings`, add a `Create` event and drag a code block to the `Actions` box.
4. Enter the following code:
   ```
   instance_create(room_width/2, 400, obj_btn_back);
   ```
5. Open `obj_btn_settings` and add a `Left Mouse Button Pressed` event.
6. In a code block, enter the following code:
   ```
   room_goto(rm_settings);
   ```
7. Open `obj_btn_back` and add a `Left Mouse Button Pressed` event.
8. Place a code block in the `Actions` box and add the following code:
   ```
   room_goto(rm_title);
   ```
9. In `obj_btn_quit`, add a `Left Mouse Button Pressed` event and place a code block in the `Actions` box.
10. Enter the following code:
    ```
    game_end();
    ```
11. In `obj_btn_play`, add a `Left Mouse Button Pressed` event.
12. Place a code block in the `Actions` box and add the following code:
    ```
    room_goto(rm_game);
    ```

The following steps should be completed for each of the button objects:

1. Add a `Mouse Enter` event and drag a code block to the `Actions` box.
2. Enter the following code:
   ```
   image_xscale += 1;
   image_yscale += 1;
   ```
3. Add a `Mouse Leave` event with a code block.
4. Enter the following code:
   ```
   image_xscale -= 1;
   image_yscale -= 1;
   ```

Once all the steps are completed, test the program and play around with the buttons. The buttons should allow you to navigate to and from the `Settings` menu, play the game, or quit the program altogether.
How it works

GameMaker makes creating branching screens and a menu system quite simple. Thanks to the use of rooms and code embedded within objects, moving from one screen to another is a snap. As you can see, navigating from the title screen to the settings menu, or even right to the gameplay, is as simple as calling the `room_goto()` function in conjunction with the room to which you wish to jump. This function can be called in many ways, the most common way being a mouse click or button press. One thing you must keep in mind is that, when you move from one room to another, the room will begin fresh from the start, as if loaded for the first time. This happens even if you've been to that room previously. When you go from the title screen to the settings screen, the title screen you were on isn't saved; rather, it's destroyed in order to save memory. Then, when you go back to the title screen by clicking on the Back button, the title screen you see now is brand new and runs from the first line of code. This can be an issue if you wish to have gameplay that features moving from one room to another, but this problem can be circumvented by saving and recalling the game data, which we'll discuss in the next chapter.

Creating splash pages

When you start a video game, especially a modern game, you'll often be greeted by one or several images or animations telling you who made the game (or even who made certain engines that were used to make the game). These splash pages are important to the developers because they want you to remember them when choosing your next game; if you liked the game they made, why not choose another by the same developer/publisher? Think of it as the information you see on the screen at the beginning of a movie; this recognition is important for everyone involved in making the movie happen. Now, some splash screens are simple images, such as the studio's logo or even simple text (though images are certainly more effective, in my opinion). Others involve an animated version of the studio's logo or even audio. Have you ever played an EA Sports game? When the EA logo shows up, you get an intense voice telling you "EA Sports. It's in the game!" It's the same across all EA Sports games because they want this ingrained in your head so that you'll remember it the next time you're looking to make a purchase. Let's add some splash screens to the game so that you can put your name on it.
**Getting ready**

We'll continue with the game in this chapter, so make sure it's loaded in GameMaker. You'll need two splash screens: one with an image (perhaps your logo or just your name) and one with an animation. The image could be a sprite attached to an object, but for this recipe we'll simply make the object the background. For the animation, place it in a sprite and attach this sprite to an object called `obj_animLogo`. As for the splash screens themselves, they should each be rooms called `rm_splash_01` and `rm_splash_02`. Order them in the asset explorer **before** `rm_title` (from the previous recipe) and make sure `rm_splash_01` has the logo as its background. You'll also need control objects called `obj_splashControl_01` and `obj_splashControl_02`.

**How to do it**

1. Open `obj_animLogo`.
2. Add a **Create** event and drag a code block to the **Actions** box.
3. Enter the following code:
   ```plaintext```
   image_speed = 0.8;
   ```plaintext```
4. Click on **Add Event**, then **Other**, and select **Animation End**.
5. In a code block, add the following code:
   ```plaintext```
   room_goto(rm_title);
   ```plaintext```
6. Close `obj_animLogo` and open `obj_splashControl_01`.
7. Add a **Create** event and drag a code block to the **Actions** box.
8. Enter the following code:
   ```plaintext```
   alarm[0] = 180;
   ```plaintext```
9. Add an event to **Alarm 0**.
10. Place a code block in the **Actions** box using the following code:
    ```plaintext```
        room_goto(rm_splash_02);
    ```plaintext```
11. Add a **Step** event and place a code block in the **Actions** box.
12. Enter the following code:
    ```plaintext```
        if keyboard_check_pressed(vk_space)
        {
            room_goto(rm_splash_02);
        }
    ```plaintext```
13. In obj_splashControl_02, add a Create event to the following code:

   `instance_create(room_width/2, room_height/2, obj_animLogo);
   alarm[0] = 180;`

14. Add an event to Alarm 0 and place a code block in the Actions box.

15. Enter the following code:

   `room_goto(rm_title);`

16. Add a Step event to the following code:

   ```
   if keyboard_check_pressed(vk_space)
   {
     room_goto(rm_title);
   }
   ```

Once these steps are complete, make sure each splash page has its respective control object and you're set.

### How it works

For this recipe, we used two splash pages but this number is arbitrary. You can add as many as you like with whatever you want on each one. Do you remember arcade machines? Many of them had splash pages with legal information or even the infamous "Winners don't use drugs." page. The point of this recipe was to demonstrate a few ways to transition from one room to another. Firstly, we set a timer via an Alarm event. When the timer hits the predetermined value, GameMaker follows the instructions which, in this case, are to move on to the next room. Secondly, we added a Step event whose sole instruction is to check whether the spacebar has been pressed or not. In the event that it has been pressed, GameMaker's instructions are to move on the next room. Essentially, we just allow the player to skip through the splash page (instead of waiting) in order to get to the game. Thirdly, and this applies only to animated logos, we used the Animation End event in conjunction with the animated logo object. Again, the player can skip through this using the spacebar, but instead of waiting for a timer, GameMaker simply waits for the animation to end. Once the last frame is drawn, GameMaker follows the instructions to move on to the next room which, in this case, is the game's title page. Now, something to keep in mind here is that, while you can make the animation as short or as long as you want, it will only be visible as long as it is playing. In order to ensure that the player sees your logo (assuming he/she doesn't skip it), you can either make the animation long enough for the player to take in the information, or you can combine the Animation End event with an Alarm event. Instead of calling the `room_goto()` function, you can have GameMaker set an alarm, at the end of which GameMaker will finally call `room_goto()`. This way is much easier as it does not involve a longer animation, which is more work for the artist and takes up more memory.
There's more

The methods outlined in this recipe don't apply only to splash and title pages; you can use these methods in the game as well. Imagine a puzzle game where, once the final event of a puzzle is solved, GameMaker moves on to another room. What if, once a certain item is collected or a specific score is reached, the player is whisked off to a bonus round? Moving from one room to another allows you to set up more elaborate gameplay elements that, if properly utilized, can add a lot of depth to a game. Now, in order to properly incorporate these methods into an actual gameplay and allow the player to move from one room to another and back again, you'll need to know how to save and recall a game's data and/or a player's stats, but this will be covered in the next chapter. In the meantime, think of this when designing your next game or gameplay mechanic.

Adding a game over screen

How do you know when you've lost the game you're playing? Well, if you're playing the game we've been creating in this chapter, you will know this because it resets itself. Not very appealing from a player's standpoint, right? Most games will show you a game over screen that will, at the very least, tell you that your gameplay session has finished. Let's take a look at creating a game over screen and what information we can convey to the player.

Getting ready

We're going to finish our game with a game over screen, so make sure that the project file is open. You'll need a few things before we begin, so let's take a look at the list. First, you'll need a new room called rm_gameOver. You'll also need a controller object called obj_gameOverCtrl as well as objects called obj_btn_exit and obj_btn_restart. We'll be reusing obj_btn_quit, but this will be available since we're using the same game project.

How to do it

1. In rm_gameOver, open the creation code and enter the following code:
   ```
   display_w = 1024;
   display_h = 768;
   display_wMax = display_get_width();
   display_hMax = display_get_height();
   ratio = display_get_width()/display_get_height();
   if (display_wMax < display_hMax) {
   ```
view_w = min(display_w, display_wMax);
view_h = view_w/ratio;
}
else
{
    view_h = min(display_h, display_hMax);
    view_w = view_h*ratio;
}
view_wview[0] = floor(view_w);
view_hview[0] = floor(view_h);
view_wport[0] = display_wMax;
view_hport[0] = display_hMax;
surface_resize(application_surface, view_wview[0], view_hview[0]);
instance_create(view_w/2, (view_h/2)-64, obj_btn_restart);
instance_create(view_w/2, view_h/2, obj_btn_exit);

2. instance_create(view_w/2, (view_h/2)+64, obj_btn_quit); In obj_gameOverCtrl1, add a DrawGUI event.

3. Drag a code block to the Actions box and enter the following code:
   draw_text((display_get_gui_width()/2)-64, (display_get_gui_height()/2)-200, string("Score: ") + string(global.playerScore));

4. In obj_btn_restart, add a Left Mouse Button Pressed event.

5. Place a code block in the Actions box using the following code:
   room_goto(rm_game);

6. In obj_btn_exit, add a Left Mouse Button Pressed event and drag a code block to the Actions box.

7. Enter the following code:
   room_goto(rm_title);

The following steps should be completed for each of the new button objects:

1. Add a Mouse Enter event and drag a code block to the Actions box.

2. Enter the following code:
   image_xscale += 1;
   image_yscale += 1;

3. Add a Mouse Leave event.

4. Place a code block in the Actions box and enter the following code:
   image_xscale -= 1;
   image_yscale -= 1;
5. Finally, in the **Step** event of `obj_gameControl`, locate this code:

   ```
   if global.gameOver
   {
     game_restart();
   }
   ```

6. Replace the preceding code with this code:

   ```
   if global.gameOver
   {
     room_goto(rm_gameOver);
   }
   ```

Once these steps are completed, make sure that an instance of `obj_gameOverCtrl` is placed in `rm_gameOver` and you're all set. When you play the game, if you allow yourself to lose, you'll see your score for the game as well as a few options. Play for a few times to try each of the buttons.

**How it works**

The final recipe in this chapter combines elements of all of the previous recipes involved in making our game. The code we entered in the creation code for `rm_gameOver` is largely the same as in the other rooms, but here, we've added a few extras. By piggybacking on this code with the `instance_create` code for the button objects, we can utilize the variables created here. Using the `view_w` and `view_h` variables, we can ensure that the objects are created in relation to what will be viewed on the screen and therefore always in the same viewing space.

The real purpose of the game over room is to tie the game together and make it feel like an actual game. The Draw GUI event called in the control object displays the score based on the global `playerScore` variable from the game room. This shows the player how well he/she did and can encourage him/her to try again and improve the score. The buttons displayed can be used to quit the game by calling `game_end`, jumping to the title, or even playing again by reloading the game room with `room_goto(rm_game)`. All of the game elements from this chapter collaborate to create a cohesive game experience. It may not be an AAA blockbuster but it's a game that works. The next few chapters will be dedicated to taking a simple game and adding polish to it.
Where to buy this book
You can buy GameMaker Cookbook from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet
book retailers.
Click here for ordering and shipping details.