Chapter No. 6
"Finite State Machines – Starting the 2D Platformer"
In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.6 "Finite State Machines – Starting the 2D Platformer"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Matthew DeLucas has been a gameplay engineer with Schell Games in Pittsburgh, Pennsylvania for over five years. He has worked on a wide range of interactive projects for PC, Web, mobiles, and consoles. Matt has also released independent projects for PC and Xbox 360, such as Convextrix, a puzzle game, and Battle High, which is a fighting game series. Being a programmer and designer, Matthew has also participated in almost every official, 48-hour Global Game Jam, managing to help his team achieve success while experimenting with new ideas.

Matthew began his programming career in GameMaker: Studio and has become proficient with additional game engines, such as Gamebryo and Unity3D, and scripting languages such as C#, Python, Lua, and MaxScript for 3DS Max. Often, he chronicles his experiences with game production on his blog at www.mattrifiedgames.blogspot.com.

Matthew has had a desire to work in the game industry ever since he was young, and he enjoys all of the facets of game production—programming, design, and art. His favorite genres include platformer, puzzles, racing, and fighting games, all of which influence his designs.

For More Information:
I wish I could list everyone I am thankful to for helping me complete this book; however, I don't think you, the reader, whom I am also thankful to, would appreciate pages of acknowledgments. Instead, I'd like to simply thank the teachers who inspired me, the friends and co-workers who give me the confidence and drive to finish this book, and most importantly, my family, for supporting me and my choice to pursue a career in the gaming industry.

For More Information:
GameMaker Game Programming with GML

This book came about as an opportunity to create and share the knowledge of one game development enthusiast with others in the hope that they would be inspired to create their own great works. The projects in this book are not meant to be final products for readers to clone, but instead, starting points to learn basic and advanced techniques used to create games. Similarly, the code in this book shouldn't be merely copied-and-pasted but understood. Game creation is sometimes described as an exercise in problem solving. By understanding the code instead of regurgitating it, solutions will arise for dozens—if not hundreds—of possible problems, as opposed to just those introduced in this text.

GameMaker: Studio is just one of many game engines; likewise, GameMaker Language is just one of many programming languages out there. Learning these tools should not be the end of one's journey into the vast topic of game development, but instead another stone in a strong foundation, even if it is the first one.

What This Book Covers

Chapter 1, Getting Started – An Introduction to GML, introduces you to the basic formatting and syntax of GameMaker Language (GML). These topics will be expanded by creating a simple button.

Chapter 2, Random Organization – Creating a Puzzle Game, discusses sprite resources and randomization. A grid of puzzle pieces is created, which acts as the base for a puzzle game project.

Chapter 3, So How Do I Play? – Adding Player Interaction, teaches us how to add player interaction to the puzzle game using the mouse and keyboard.

Chapter 4, Juicy Feedback – Aural and Visual Effects, elaborates upon the implementation of sound effects and particle systems and the use of alarms, so the game can better inform the players about their progress.

Chapter 5, Solving the Puzzle – Finishing Touches to the Puzzle Game, helps us create a menu that allows the player to adjust various parameters of the puzzle game using the Draw events. A score and timer will also be implemented.

Chapter 6, Finite State Machines – Starting the 2D Platformer, starts a platformer game, focusing on the creation of the main character who is controlled by a finite state machine.

Chapter 7, It's in the Name – Platforms and Collisions, expands the platformer game started in the previous chapter by adding collision through static and moving platforms.

For More Information:
Chapter 8, Setting the Stage – Views, Backgrounds, and Tiles, helps us create a camera system using views, while the platformer game's environment will be fleshed out using background resources and tiles.

Chapter 9, Breaking Vlad – Pickups, Hazards, and Enemies, helps the character interact with pickups to increase score and health, and also hazards and enemies to create a challenge for the player.

Chapter 10, GOAL – Timelines and Feedback Review, uses timeline resources to create a way to trigger a series of events. Then, particle systems and audio will be reviewed, adding some finishing touches to the platformer.

For More Information:  
In the previous chapter, **Draw** events were used to create a menu as well as display score and timer information for the puzzle game. The production of new features for that game, however, will conclude in favor of starting a new game. In this chapter, and for the remainder of this text, a 2D platformer will be built!

The average 2D platformer deals with twitch-based gameplay that consists of a character usually running and jumping around in an environment. This environment is then filled with various hazards, such as spikes, pitfalls, and monsters. In this first chapter, the focus will be on setting the state of the character through the use of a finite state machine. By the end of this chapter, the following aspects of the platformer will be set up:

- A platformer character will be created
- This character will transition from the idle state to the walking or jumping state
- These state transitions will be made through keyboard input
- The character will also be moved through the use of built-in variables, such as friction and gravity

**Introducing finite state machines**

A **finite state machine (FSM)** is an abstract programming concept involving the management of an object, its different states, and the transition between those states. A **state** can be defined as a unique mode or condition that an object is in at a given time.

---

For More Information:
For example, a door can have various states: opened, closed, opening, and closing. What an FSM does is make sure the door is only in one state at any given time. A door can't be both open and closed at the same time, right? In addition, an FSM would manage the transitions of the object from one state to another, such as the door going from closed to opening and open, or open to closing and closed.

In the 2D platformer, the character will have a very simple finite state machine as illustrated in the following diagram:

![Finite State Machine Diagram](image)

To elaborate the previous screenshot, the player will start in the Idle State. Then, depending on player input or environment interaction, the player will transition into other states, such as the Walk State or the Jump State. From the walk state, they will either return to the idle state or transition to the jump state, and once in the jump state, they will only be able to return to the idle state. This is a very simple example of a finite state machine, and in many games, characters have dozens of different states. Usually, unique actions are triggered upon entry, update, or exit from a state. In this project, the transitions from one state to another and their updates will be focused on through the use of the Step event and designated with user-defined events.

**Gathering resources for the platformer**

Before the platformer can be started, resources must be created. These assets will focus primarily on the character, which in this case, will be a vampire named Lil' Vlad.

A new project should also be started when creating this new game. Again, a new project can be created by navigating to File | New Project or pressing Ctrl + N. This will bring up the Project dialog window. This project is simply titled Platformer.
Establishing Lil' Vlad's sprites

Relating to the finite state machine mentioned previously, Vlad's character will have three sprites associated with it, as shown in the following list:

- The spr_vlad_idle sprite for the idle state
- The spr_vlad_walk sprite for the walking state
- The spr_vlad_jump sprite for the jumping state

The standing sprite – spr_vlad_idle

This first sprite resource will be for Vlad's idle animation; it's a 30-frame, looping animation of him standing still. It has several attributes that are important in aiding with positioning and collision.

The first attribute is the origin that is centered horizontally and placed near the bottom of the sprite. The sprite is 50-pixels wide, so the X origin is placed at 25. This is done so that the character can properly be mirrored when facing different directions. The height of the sprite is 120 pixels. Because the character's position should be where his feet meet the ground, the Y origin will be a little higher; in this case, it should be set as 113. The following screenshot shows the coordinates of the origin as well as several frames of this sprite resource:

For More Information:
The bounding box is set manually. All bounding boxes used for Vlad will be 30 pixels wide. This setup is done so that the character can collide on either side of the bounding box upon moving through the environment and will be kept consistent for all of the sprites introduced in this chapter. To achieve this, the bounding box should have a value of 10 on the left and 40 on the right. The height for the bounding box will change between states. In the idle state, the bounding box is 108 pixels high, ending just below the tip of the character’s spiked hair. The Bottom value of the bounding box is the same as the origin, 113. The Top value is then set as 5. The following screenshot shows this setup:

![Mask Properties: spr_vlad_idle](image)

The walking sprite – spr_vlad_walk

The next sprite resource will be used for Vlad’s walk animation. As with the idle animation, it will loop too. Changing the animation of the character at any time serves as feedback to the player indicating that they have entered a new state and should be performing a new action, which is, in this case, horizontal movement. The setup of this sprite is similar to that of spr_vlad_idle. The origin is near the bottom at 130, but it is not quite centered horizontally. The horizontal origin is set to 48, which is around the character’s hips and will allow him to mirror.

---

For More Information:
The bounding box will be of the same width as `spr_vlad_idle`—30 pixels; the bounding box's left side will be at 33 and the right will be at 63. The bottom of the bounding box is at the origin, which is 130. Finally, this sprite is a bit taller than others, the top of it reaching 10, which makes the bounding box 120 pixels high. This is illustrated in the following screenshot:

![Bounding Box and Origin](image)

**The jumping sprite – `spr_vlad_jump`**

The final sprite resource for this chapter will be used for the animation of Vlad's jump. This will not only be used when the character jumps but also when the character falls off ledges. The first 11 frames account for his ascent and float, while the remaining frames account for his descent. As with the previous two sprites, the origin will be near the bottom and centered at a location where the character mirrors best. The bounding box will then be 30 pixels wide, with 15 pixels on each side of the origin and its bottom at the same point as the origin. The following list contains these different values on the **Sprite Properties** and **Mask Properties** pages:

- The X value in **Origin** is set to 41
- The Y value in **Origin** is set to 128
- The **Left** value in **Bounding Box** is set to 26
Finite State Machines – Starting the 2D Platformer

- The Right value in Bounding Box is set to 56
- The Top value in Bounding Box is set to 2
- The Bottom value in Bounding Box is set to 128

The following screenshot shows the use of different values on the Sprite Properties and Mask Properties pages:

For More Information:
Jumping with sound – snd_jump
Continuing the trend of giving feedback when possible, a sound resource will be created that will be played when Vlad jumps; this sound will be referred to as snd_jump. Jump sounds are usually quick and not grating, as they will be heard quite often.

Creating a new object resource – obj_vlad
The character, Vlad, will be represented with one object resource named obj_vlad. This object resource will utilize six events in total, as shown in the following list:

- The **Create** event: This event is triggered when the character is created
- The **Step** event: This event is triggered in every frame
- The **Animation End** event: This event is triggered when the current animation ends
- Three **User defined** events: These events will be triggered using `event_user` in the **Step** event

The scripts for these events will be defined later in this chapter. The initial sprite assigned should be spr_vlad_idle.

Utilizing the User defined events
The **User defined** events (Add Event | Other | User defined | User 0-15) are custom events that can be triggered through the built-in function `event_user`, whose first argument represents the corresponding index of the desired **User defined** event. Each object resource can have 16 unique **User defined** events.

When updating states, a `switch` statement could be used that tests all the possible states and then performs specific actions based on the current state, but a `switch` statement can become overpopulated very quickly, especially when more complex scripts are written for each state. An array that stores various references to script resources for each state could also be built, but because there will not be more than 16 states, the **User defined** events and `event_user` will be sufficient.

For More Information:
Placing Vlad in a room – rm_level1

A room resource, rm_level1, will be created. This room will be 1024 pixels by 768 pixels and contain one instance of obj_vlad, as illustrated in the following screenshot:

The rm_level1 room will also utilize the Creation Code page, which was discussed briefly in Chapter 1, Getting Started – An Introduction to GML. The Creation Code page is triggered upon the player entering a room. The following is the content in Creation Code of rm_level1:

```gml
// Defines the bottom of the room
global.room_bottom = 650;
```

The global variable in the previous code represents an artificial bottom of the room, and will be used when adding jump functionality to the character.
If the resources have been created properly, the game should appear similar to that shown in the following screenshot with Vlad standing idly in the room:

![Screenshot of Vlad standing idly in the room](image)

### Defining Vlad's state constants

Now that the resources have been created, the states that Vlad will transition to and from should be defined as constants for easier tracking. Again, constants are global variables that cannot be set, and are used to improve code readability for referencing information that is meant to be consistent. The User-Defined Constants dialog window can be opened by navigating to **Resources | Define Constants**... or pressing `Shift + Ctrl + N`. The list of name-value pairs will initially be empty. This dialog window has nine buttons (excluding the **OK** button) that are used to populate and edit the following list:

- **Insert**: This button inserts a new constant before the currently selected one
- **Add**: This button creates a new constant at the end of the list
- **Delete**: This button deletes the currently selected constant
- **Clear**: This button removes all defined constants
- **Up**: This button moves the currently selected constant up in the list
- **Down**: This button moves the currently selected constant down in the list
- **Sort**: This button sorts the constants alphabetically by name
- **Load**: This button loads a series of constants from a text file
- **Save**: This button saves the current set of constants as a text file

For More Information:

For this game, three states in the following table will be defined:

<table>
<thead>
<tr>
<th>Name of the state</th>
<th>Value</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>state_idle</td>
<td>0</td>
<td>This state is used to implement the idle state of Vlad</td>
</tr>
<tr>
<td>state_walk</td>
<td>1</td>
<td>This state is used to implement the walking state of Vlad</td>
</tr>
<tr>
<td>state_jump</td>
<td>2</td>
<td>This state is used to implement the jumping state of Vlad</td>
</tr>
</tbody>
</table>

These states are shown along with the interface for defining constants in the following screenshot:

The values of these constants are very important as they will correlate with the **User defined** events triggered in `obj_vlad`. If created properly, these three values will show up in different colors when editing scripts and in the auto-complete form as well.

For More Information:
Starting Vlad's events and scripts – walking

For the walk state of obj_vlad, the player will use the left and right keyboard arrows to move the character. To accomplish this, scripts will be written into the following events:

- The Create event
- The Step event
- The User Defined 0 event
- The User-Defined 1 event

The Create event

Similar to most Create events, the Create event of obj_vlad will handle defining important instanced variables. The following code assigns values to each of these variables that we need to go from the idle state to the walking state and vice versa:

```gml
/// Variable initialization for character

// The state of the character used within the update state.
state_id = state_idle;

// Has the player entered a new state.
entered_new_state = false;

// The speed at which the character walks by default.
def_walk_speed = 10;

// The friction used to reduced the character speed when returning to idle.
def_friction = 1;
```

Right now only four variables are defined. The state_id variable is the index of the state that Vlad is currently in, starting with his idle state. The entered_new_state value is a Boolean that signifies whether or not a new state has been entered. The next two variables are prefaced by def_ which stands for default. The def_walk_speed variable defines the walking speed of the character. Setting this to 10 means that Vlad will travel 10 pixels every frame. Finally, def_friction will be used to set Vlad's friction value when he is no longer running. The friction variable is a built-in variable that, when positive, affects speed by making it approach 0 in every frame. In this instance, setting friction to 1 will make Vlad's speed values—both horizontal and vertical—approach 0 by 1 unit every frame.
When friction is negative, the value of speed will move away from zero in every frame, which can create rather substantial changes in speed.

The Step event
To update Vlad in every frame, a Step event will be utilized. The following code will be written into the Step event:

```gml
/// State machine update.

// Gets the current state id.
var current_state = state_id;

// Updates the event.
event_user(state_id);

// If the event has changed, the entry of a new state is designated.
if (current_state != state_id)
{
    entered_new_state = true;
}
```

In the first portion of this code, the current value of state_id is assigned to a local variable. Then, one of the User defined events is triggered based on the state. Finally, state_id and current_state are compared; if they differ, which means that the state has changed while it was being updated, entered_new_state is set to true.

Standing still – the User Defined 0 event
The idle state is a rather important state; it is the starting state and can transition to most other states. In this case, the keyboard will be used to change the state so that the character can move left and right. Also, the use of entered_new_state will be demonstrated. The following is the code for the idle state:

```gml
/// IDLE STATE

// If the idle state has been entered.
if (entered_new_state)
{
    // Sets the sprite index, image index, and loop index.
    sprite_index = spr_vlad_idle;
```

For More Information:
image_index = 0;

// Sets the friction
friction = def_friction;

// Set to false so the entry functions are not called again.
entered_new_state = false;
}

// If left arrow is down...
if (keyboard_check(vk_left))
{
    // Changes the state id.
    state_id = state_walk;

    // Mirrors the character so it is facing left.
    image_xscale = -1;
}

// If right arrow is down.
else if (keyboard_check(vk_right))
{
    // Changes the state id.
    state_id = state_walk;

    // Sets the x scale to 1 to make the character facing right.
    image_xscale = 1;
}

In the previous block of code, the first if statement checks entered_new_state. If entered_new_state returns true, several actions are performed. Firstly, the sprite_index value is set to the idle animation, and image_index is set to start at the frame 0. Then, the friction is set to def_friction. This is done to give the character a nice, smooth stop as opposed to a jarring one when returning from the walk state. Finally, entered_new_state is set to false so that it does not continually execute at every update.

After this, the keyboard is checked using keyboard_check. If the left arrow key is held down, state_id is set to the predefined constant state_walk, and the character is mirrored to look left by setting image_xscale to -1. If the left arrow key is not held down, but the right one is, the state is still set to state_walk, but image_xscale is set to 1.

For More Information:
Walk this way – the User Defined 1 event

If the game were to be tested now, nothing would happen if the right arrow key is pressed, and if the left arrow key is pressed, Vlad would simply mirror and face left. Behind the scenes, he would have entered the walk state, but because the User defined event has not been coded, he would get locked into the walk state and be unable to return to the idle state. By writing the following script in the User Defined 1 event, Vlad will be able to enter and exit the walk state:

```c
/// WALK STATE
// If the walk state has been entered.
if (entered_new_state)
{
    // Sets the sprite index, image index.
    sprite_index = spr_vlad_walk;
    image_index = 0;

    // Sets the friction and horizontal speed, reversing it if the
    // character is facing left.
    friction = 0;
    hspeed = def_walk_speed;
    if (image_xscale < 0)
    {
        hspeed *= -1;
    }

    // Set to false so the entry functions are not called again.
    entered_new_state = false;
}

// If the left arrow is released while facing left or the right arrow
// is released while facing right...
if ((image_xscale < 0 && !keyboard_check(vk_left)) || (image_xscale > 0 && !keyboard_check(vk_right)))
{
    // Return to the idle state.
    state_id = state_idle;
}
```

Similar to the User Defined 0 event, the first if statement checks whether or not Vlad has recently entered the walk state. If he has, the sprite_index object must be changed. Then, friction is set to 0 so that it doesn't affect hspeed, which is set to def_walk_speed. If the value of image_xscale is less than 0, meaning the character is facing to the left, the hspeed value is multiplied by -1. Finally, entered_new_state is set to false to prevent it from continually updating.
Then, a compound Boolean is checked to determine when Vlad should re-enter the idle state. If Vlad is facing to the left and the left key isn't being held down anymore, or he is facing to the right and the right arrow key isn't being held down, he should return to the idle state.

If these scripts are entered properly, Vlad should be able to run to left and right in the scene, slowing down slightly upon returning to the idle state. In the next section, the scripts of the Create and Step events will be amended and the User Defined 2 event and the Animation End event will be defined so that Vlad can jump!

Adding new variables for jumping
To make Vlad jump, more variables must be defined. These variables are associated with the default parameters for making Vlad jump, such as def_jump_speed and def_gravity. These variables will be scripted into the Create event as shown in the following code:

```gml
// The speed applied to the character when jumping by default.
def_jump_speed = -15;

// The gravity applied to the character by default.
def_gravity = 1;

// The number of times the character has jumped.
jump_count = 0;

// The number of times the character is allowed to jump
jump_limit = 2;

// Is the character falling; used to differentiate jumping from falling.
grounded = false;

// At which frame does the animation loop?
loop_index = 0;
```

The def_jump_speed variable represents the initial jump speed that Vlad's vspeed value will be set to when entering the jump state; meanwhile, def_gravity is the value that will set the built-in, instanced variable gravity. Using a value of -15 for def_jump_speed means that Vlad will start moving up at a rate of 15 pixels per frame. During every frame, the speed value will be increased by gravity, which is set to 1, eventually causing the character to fall back down.
The `jump_count` variable represents the number of times the character has jumped. The `jump_limit` variable represents the number of times Vlad is allowed to jump. Defining `jump_limit` is useful to allow skills such as double jumping, and could even be an unlocked ability; its value can start out as 1 and then, with the use of a special item, be increased to reach new sections in a level. This is jumping ahead a bit, so for now, `jump_limit` will be set to 2, which will allow Vlad to double jump.

The `grounded` variable is used to represent whether or not Vlad's feet are "planted". It will also be used to determine whether or not Vlad is actually jumping or falling.

Finally, `loop_index` is a value that will be applied to `image_index`, making Vlad's animation loop correctly when performing the jump animation since it can look rather strange otherwise.

### Using up to jump – the Step event update

In this section, code will be added to the Step event so that Vlad can jump. Vlad can enter the jump state from both the idle and the walking states, so adding the following code to both events is redundant, and instead, is added to the Step event. This update, as shown in the following code, should be scripted after `event_user` is called, but before `current_state` and `state_id` are compared:

```plaintext
/// State machine update.

// Gets the current state id.
var current_state = state_id;

// Updates the event.
event_user(state_id);

// Jump functionality, which can be performed in either state except jump.
if (state_id != state_jump)
{
    // If above the bottom of the room, the character falls.
    if (y < global.room_bottom && place_empty(x, y + 1))
    {
        state_id = state_jump;
        grounded = false;
    }

    // If grounded, the jump count is in range, and the spacebar was pressed, the character will jump.
    if (grounded && jump_count < jump_limit && keyboard_check_pressed(vk_up))
```
An if statement is used first to make sure this check only occurs when Vlad is not in the jump state. Then `global.room_bottom`, which is defined within the Creation Code page of the room, is compared to the current position. If Vlad is above it, the jump state is entered; it is also designated that the character is not grounded. Meanwhile, if the character is grounded, the number of jumps performed is less than the specified limit, and the up arrow has been pressed, the state is changed to `state_jump`.

**Falling state – the User Defined 2 event**

The next script will execute several actions when Vlad enters the jump state. Jumping is a much more complex action than walking and standing, and small touches have been added to make jumping feel more responsive and give the player more control, similar to how friction is applied when returning to the idle state, as shown in the following code:

```gml
/// JUMP / FALL STATE

// If the state was newly entered.
if (entered_new_state)
{
  // Sets sprite index and looping index.
  sprite_index = spr_vlad_jump;
  loop_index = 11;

  // If grounded, the vertical speed is set to perform another jump and the jump counter is incremented.
  if (grounded)
  {
    vspeed = def_jump_speed;
    image_index = 0;
    audio_play_sound(snd_jump, 0, false);
  }
}
```
finite state machines – starting the 2d platformer

} else {
    image_index = 11;
}

// Jump counter is incremented, even for falling.
jump_count++;

// Sets the gravity and friction.
gravity = def_gravity;
friction = 0;

// Designates that the character is not grounded.
grounded = false;

// Set to false so the entry functions are not called again.
entered_new_state = false;
}

The initial portion of this script deals with entering the jump state. The image_index and loop_index variables are set to 11 since the descending loop cycle begins at frame 11. Now, the variable grounded is used. If the character is grounded, vspeed is set to def_jump_speed, the animation is set to start at image_index with the value 0, and the jump sound that was created earlier is played. If the character is not grounded, image_index is simply set to the same frame as loop_index. This is followed by incrementing the value of jump_count, setting gravity and friction to their designated default values, and changing grounded as well as entered_new_state to false. The code in the next section will handle changing the character's speed with left and right key input:

    // If the left or right arrow key is pressed, the character moves slightly in that direction...
    if (keyboard_check(vk_left)) {
        image_xscale = -1;
        hspeed = max(hspeed - def_friction, -def_walk_speed);
    } else if (keyboard_check(vk_right)) {
        image_xscale = 1;
        hspeed = min(hspeed + def_friction, def_walk_speed);
    } else
    // otherwise the character returns to a neutral, horizontal speed.
In the previous code, if the left arrow key is held down, Vlad will face to the left and approach the negative of the default speed at the rate of the predefined friction variable. If the right arrow key is held down, the opposite is done. Finally, if neither key is held down, the horizontal speed approaches zero. The hspeed value is compared with def_friction instead of 0 to prevent a teetering effect in case the addition or subtraction of the hspeed value results in the value going below or above 0 and not settling on it. These calculations must be performed because friction cannot be set; doing this would affect the vertical speed as well.

If these keyboard checks were omitted, the character's jumping speed would be determined by the previous state. If the character is in the idle state, he would jump vertically and if he is in the walking state, he would jump with the same speed and in the direction he is already facing. This is rather basic, but it's nice to be able to make the character change direction or slow down for more controlled jumps. At the same time, changing direction in the air shouldn't be as fast as it would be when the character is on the ground, so when buttons associated with vk_left or vk_right are pressed, he adjusts the trajectory of his jump slightly instead of immediately.

Another bit of polish that can be added to the jump is the ability to make the character not always jump at the same height. Also, an action to test for double (or any additional number of) jumps should be performed. Scripting the following will allow this to happen:

```c
// If the character is moving up, and the up arrow is released, the speed is divided in half...
if (vspeed < 0 && keyboard_check_released(vk_up))
{
    vspeed *= 0.5;
}
// If the jump count is less than the limit and the up arrow is pressed...
else if (jump_count < jump_limit && keyboard_check_pressed(vk_up))
{
    // Set to true to reenter the jump state.
```
entered_new_state = true;

// Sets grounded to true so the character jumps and doesn't fall.
grounded = true;
}

So first, vspeed is checked and if its value is less than 0, which means the character is ascending, and the up arrow is released, the value of vspeed is multiplied by 0.5. This will give the illusion of the jump ending early and smoothly instead of just immediately stopping in midair and reversing direction. However, if the up arrow is pressed while the value of jump_count is less than that of jump_limit, the entered_new_state and grounded variables are both set to true so that on next update, another jump is performed.

The following final section will define what happens when the character reaches global.room_bottom:

// If the y position plus the vertical speed is below the bottom of the room...
if (y + vspeed > global.room_bottom)
{
    // Character is moved to the bottom of the room.
y = global.room_bottom;

    // Speed and gravity are set to 0 to stop vertical movement.
    vspeed = 0;
gravity = 0;

    // Jump counter is reset.
jump_count = 0;

    // The state is set to idle.
    state_id = state_idle;
}

The final portion of code helps end the jump. If the sum of Vlad's y position and his vertical speed is greater than the value of the bottom of the room defined in the Creation Code script of rm_level1, he will enter the idle state. This prediction is made so that Vlad's position doesn't fall past this value for one frame and then snap back to the position he should settle at in the next. Before entering though, the vspeed and gravity values are both set to 0, and the y coordinate is set to global.room_bottom. The jump_count value is also set to 0 so that Vlad can jump again, now that he has returned to the ground. This is a very important step; if this were not done, Vlad would fall offscreen and into a virtual abyss forever or until the game is stopped.
The code for the jump state is rather complex and lengthy. For clarification, the following is the script in its entirety:

```gml
/// JUMP / FALL STATE

// If the state was newly entered.
if (entered_new_state)
{
    // Sets sprite index and looping index.
    sprite_index = spr_vlad_jump;
    loop_index = 11;

    // If grounded, the vertical speed is set to perform another jump
    // and the jump counter is incremented.
    if (grounded)
    {
        vspeed = def_jump_speed;
        image_index = 0;

        audio_play_sound(snd_jump, 0, false);
    }
    else
    {
        image_index = 11;
    }

    // Jump counter is incremented, even for falling.
    jump_count++;

    // Sets the gravity and friction.
    gravity = def_gravity;
    friction = 0;

    // Designates that the character is not grounded.
    grounded = false;

    // Set to false so the entry functions are not called again.
    entered_new_state = false;
}

// If the left or right arrow key is pressed, the character moves
slightly in that direction...
```

For More Information:
if (keyboard_check(vk_left))
{
    image_xscale = -1;
    hspeed = max(hspeed - def_friction, -def_walk_speed);
}
else if (keyboard_check(vk_right))
{
    image_xscale = 1;
    hspeed = min(hspeed + def_friction, def_walk_speed);
}
// otherwise the character returns to a neutral, horizontal speed.
else
{
    if (hspeed < def_friction)
    {
        hspeed = min(hspeed + def_friction, 0);
    }
    else if (hspeed > def_friction)
    {
        hspeed = max(hspeed - def_friction, 0);
    }
    else
    {
        hspeed = 0;
    }
}

// If the character is moving up, and the spacebar is released, the speed is divided in half...
if (vspeed < 0 && keyboard_check_released(vk_up))
{
    vspeed *= 0.5;
}
// If the jump count is less than the limit and the spacebar is pressed...
else if (jump_count < jump_limit && keyboard_check_pressed(vk_up))
{
    // Set to true to reenter the jump state.
    entered_new_state = true;

    // Sets grounded to true so the character jumps and doesn't fall.
}
For More Information:
grounded = true;
}

// If the y position plus the vertical speed is below the bottom of the room...
if (y + vspeed > global.room_bottom)
{
    // Character is moved to the bottom of the room.
    y = global.room_bottom;

    // Speed and gravity are set to 0 to stop vertical movement.
    vspeed = 0;
    gravity = 0;

    // Jump counter is reset.
    jump_count = 0;

    // The state is set to idle.
    state_id = state_idle;
}

If the game were to be played now, Vlad’s three states should almost be complete except for one minor detail. When Vlad is jumping, it is noticeable that about halfway through the jump, he re-enters the jump animation from the beginning, which creates a rather confusing effect. In the next section, this will be remedied with the Animation End event and minor updates to the idle and walk states events.

### Looping the jump – the Animation End event

Until this point, the variable `loop_index` has not been utilized. The Animation End event (`Add Event | Other | Animation End`), which is triggered whenever the final frame of an animation is reached, will use `loop_index` now to prevent Vlad’s jump animation from starting over midjump, as shown in the following code:

```gml
/// Loops the animation to the designated frame upon completion of the animation.
image_index = loop_index;
```

It can be said that this code is rather simple. It's handling a minor detail, but these details can sometimes help resolve or create confusion while playing that otherwise would be present. Before continuing, `loop_index` must be assigned when the idle and walk states are entered. In the User Defined 0 event, the following code can be added:

```gml
// If the idle state has been entered.
if (entered_new_state)
```
Finite State Machines – Starting the 2D Platformer

{  
    // Sets the sprite index, image index, and loop index.
    sprite_index = spr_vlad_idle;
    image_index = 0;
    loop_index = 0;
}

As shown in the following code, in the User Defined 1 event, loop_index should be assigned the value 0 similar to the previous code:

    // If the walk state has been entered.
    if (entered_new_state)
    {
        // Sets the sprite index, image index, and loop index.
        sprite_index = spr_vlad_walk;
        image_index = 0;
        loop_index = 0;
    }

Now that the previous code has been written, Vlad's animation should loop properly in all three states and Vlad should be able to run and jump around the room in those states, just as shown in the following screenshot:

[Idle, Walking, Jumping]
Summary

In this chapter, a new game was created starring a little vampiric character named Vlad. Code was written so that this character, using a finite state machine, could transition between three states: standing idle, walking, and jumping. Scripts were also written so the character could transition between these three states through the use of the User defined events, the Step event, and keyboard input. Some of GameMaker: Studio's built-in movement variables, such as hspeed, friction, and gravity, were also discussed. Getting the character basics down early is a very important step in building a successful platformer.

Now, character movement and interaction are nice, but the game is missing its most important asset: platforms! In the next chapter, collision will be discussed so that platforms can be created. This will give Vlad elements that he can actually walk and jump onto!
Where to buy this book

You can buy GameMaker Game Programming with GML from the Packt Publishing website:

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.