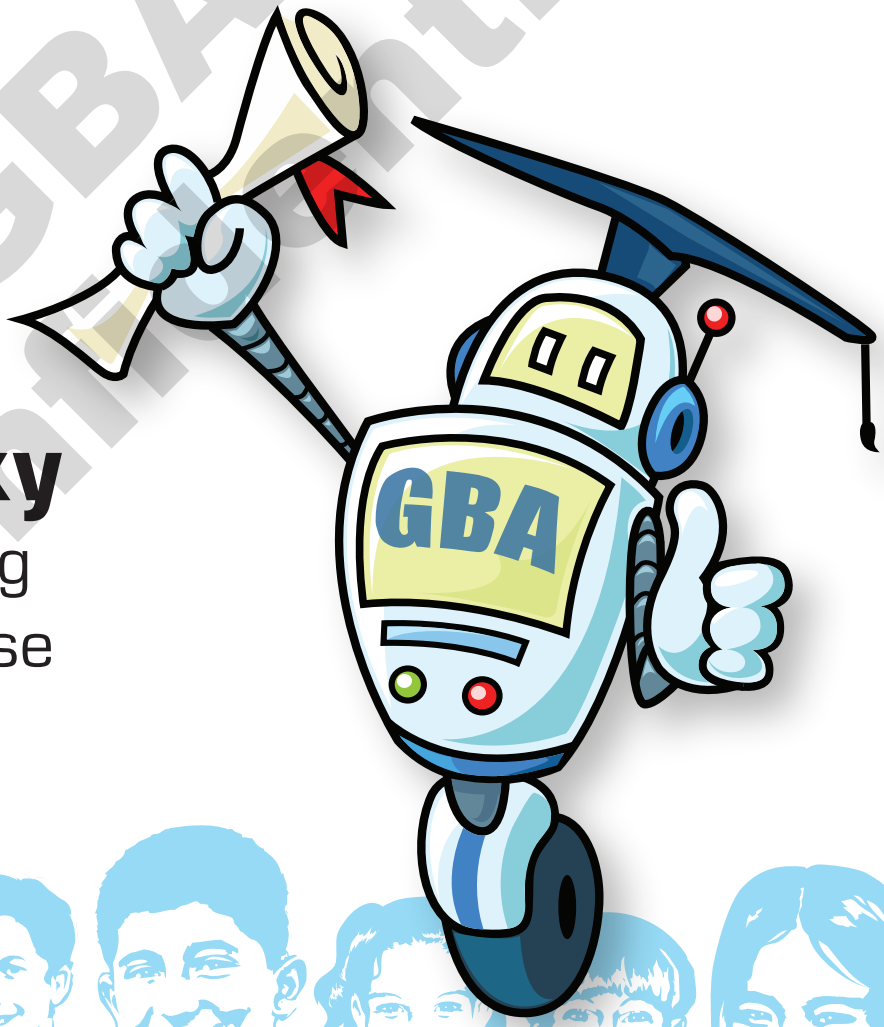
Game Builders Academy™

Learn ★ Grow ★ Have Fun ★ Succeed!

Video Game Design and Development

Level 2



Phil Lipsky
Sally Rosenberg
Michael Pugliese

TEAM GBA CORP. 35 Lace Lane, Westbury, New York 11590

Game Builders Academy™

Learn ★ Grow ★ Have Fun ★ Succeed!

Video Game Design and Development, Level 2
written by Phil Lipsky

Curriculum Development
Phil Lipsky

Box/Binder Design
Michael Pugliese

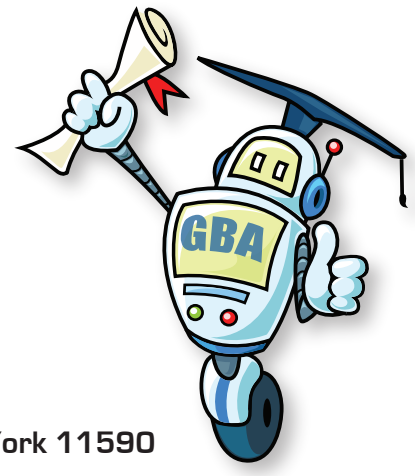
Production Editor
Sally Brecher Rosenberg

Game Art
Michael Pugliese

Production Assistants
Stacey Rosenberg,
Temma Clark-Braverman

Marketing and Sales
Walter Ebe
John Taylor

ISBN-13:
978-0-9815502-1-3



TEAM GBA CORP. 35 Lace Lane, Westbury, New York 11590

Welcome to this Exciting Teaching Tool

Enjoy this wonderful journey with your students - and let us know about your successes!

Course Overview

This course is designed to guide you and your students through the process of building a full, working video game. Along the way, there are many opportunities for students to apply, practice, and reinforce various academic subjects they've learned or are learning in school.

In this course, your students will use and strengthen their math skills, logic skills, communication skills, concentration and critical thinking skills, problem solving and creative thinking – all in the context of learning how to design and program their own video game. Your students will be introduced to the technical and artistic concepts and techniques of designing and building a video game. Students will create some of their own art for their games. Students will also be introduced to the fundamentals of animation for use in their games.

Students will use the free version of Game Maker software to create their games. Game Maker is a software (program) that enables the user to create video games in either a full click-button (click-and-drag) environment, or fully by writing programming code, or using a combination of both. In this Level 2 course, most of the game development will be done using hand-written programming code.

How to Use this Manual

This manual is designed to guide you and your students through the process of building a full, working video game. Along the way there will be many opportunities for discussion about various academic subjects, creative and artistic ideas, and much more.

The project is divided into lessons, each consisting of a number of modules. The modules are designed to be, as the name implies, modular. Many of them can be done in different sequence or on their own for variations on the given project. But we highly recommend that you follow the sequence of lessons and modules in as laid out in this manual. The order of modules has been carefully designed to build sequentially to the finished project, and to build students' knowledge, skills, and confidence with each successive module.

Each module includes the detailed procedure(s), in a clear, numbered, step-by-step organization. Indented and offset within each module are questions to ask students (along with answers), additional explanations and clarifications on given steps, ideas for having students reinforce material or experiment with skills newly gained in the given module (or step), and more. We highly recommend that you incorporate as much of this material as possible.

Throughout the manual, there are also many sidebars on the right-hand side of the pages. These contain additional information, ideas for variations on the given module, ideas for discussion, academic connections, important information, and more. As with the indented material in the body of the modules, we recommend that you include as much of the sidebar material as possible for your class.

Show the sample game (OrbQuest) to your students during the first day of class so they can see the end goal, and so they will have a clear picture and overview of the project.

Along with the finished sample game, there is a file that accompanies each module and shows how the game should look up to, and after, completion of the module. These are called "postmods" for post module, or "after" module. It's usually helpful to show the postmod for a given module before teaching that module, but this is up to you. You may choose to have your students fully discover the result of their work on their own, without first seeing what the result will be.

If a student misses one or more classes, when they return, they can use the Module start file located on the Instructor Files as their "starter" file on the day they return. Although this is, of course, not as good as having the student's own game to work on that day, it at least allows the student to follow, complete, and learn the day's material, with a file that includes all the material that was covered on the day(s) missed.

The postmods also serve as reference for you and for your students as to exactly how the file is set up. So, as a teacher, you have each module on paper, with the steps - and you also have the specific postmod file that goes with that module to show exactly how the file looks with all the steps in the module completed.

Take the opportunities for academic discussions as far and as deep as you like, or not at all, as you see best for your class and the given day, students' responses, classroom situation, etc. **Enjoy this wonderful journey with your students - and let us know about your teaching successes and students' accomplishments!**

What's on the Included Disks

INSTRUCTOR FILES DISK

Link to Game Builders Academy Web site: http://www.gbalearning.com

Link to Game Maker, Audacity, Daz downloads:
http://www.gbalearning.com/gamedevresources.html

Student Starter File: This is the starter file for Module1. The emphasis in this course is on programming, so this file provides the basic sprites, objects, rooms, sounds, and backgrounds necessary to begin the game creation process.

Post Modules Files: There is a file that accompanies each Module and shows how the game should look up to, and after, the completion of each module. Also included is an executable file of how the game will look at the completion of each lesson.

You can also use these files if a student misses one or more classes. When the student returns, use any PostMod file as a "starter" file on the day they return.

Student Tutorials: PDF's of each Student Tutorial Module.

Code Documents: Word documents of all object code used in game creation. Simply copy and paste pieces of code, or an object's code in its entirety. Helps eliminates typos!

Enhanced Games: Two enhanced games are provided to show you some idea of how students may change and embellish their games with a variety of enhancements discussed in Module 24.

Design Document: This 2-page form sets the student mind working. The students create the design document for a game they would like to create.

STUDENT FILES DISK

Link to Game Builders Academy Web site: http://www.gbalearning.com

Link to Game Maker, Audacity, Daz downloads:
http://www.gbalearning.com/gamedevresources.html

Puggy Sprites: This file contains a number of different sprites that students may load into their games. They were created by Game Builders Academy Master Artist Michael Pugliese. Students may use them Royalty Free.

Backgrounds: For students who wish to concentrate on game design rather than game art, we have included backgrounds that may serve as a Game Intro, Game Win, and Game Lose screens. There are also a variety of other backgrounds the students may wish to import or edit.

Game Art Appendix: All the artwork shown in Appendix III: Art and Animation for Games. After class discussions, students may want the images to repeat the effects discussed.

Teacher's Manual Table of Contents

continued on following page

Teacher's Manual Table of Contents *Continued from previous page*

Appendixes

Object Code

Project 1 Overview: OrbQuest

The game that students will create is referred to as a 2-dimensional side-scrolling, platform game. Games such as Super Mario Bros and Sonic the Hedgehog are among the most famous side-scroller games.

Side-scrolling platform games typically involve a character that moves from left to right, jumping over gaps and onto various platforms, climbing ladders, and defeating adversaries. Side-scrolling platform games almost always feature a cartoonish hero. While these games can use 2D or 3D graphics (or both), the action occurs in a two-dimensional plane.

Students' games will have three or more levels (depending on time available and students' own game design choices). The game will be complete with a Game Start, Game Over, and Game Win screen, fully developed art, and gameplay functionality.

In different levels of the game, the player will be located inside or outside a castle and will be faced with challenging adversaries that will chase the player. When adversaries touch the player, or the adversary projectiles touch the player, the player's health will be reduced. When the player's health is fully depleted, the game ends. The player will be able to throw (food) projectiles at the adversaries to make them disappear. The player will have a number of objects to collect during his journey and transverse pitfalls, platforms, and ladders.

Students will be able, time permitting, to add additional programmed features, art, sound, and animation effects. At the end of the course, students will create an "executable" file of their game. This is a true PC gaming format file, which can then be played on any PC, with or without Game Maker being installed. Students can also post their games on the web or send them via email for friends and families to see and play.

Lesson 1 Overview

Modules 1-4

Module 1: Open and Save OrbQuest Starter File

OVERVIEW:

In this Module, students will: Open the OrbQuest Starter File and familiarize themselves with the contents of the Library.

INTRODUCTORY DISCUSSION:

Students already familiar with sprite, object, and room creation are given an OrbQuest starter file containing a rather robust Library containing all the resources necessary to create a 2-dimensional side-scrolling platform game.

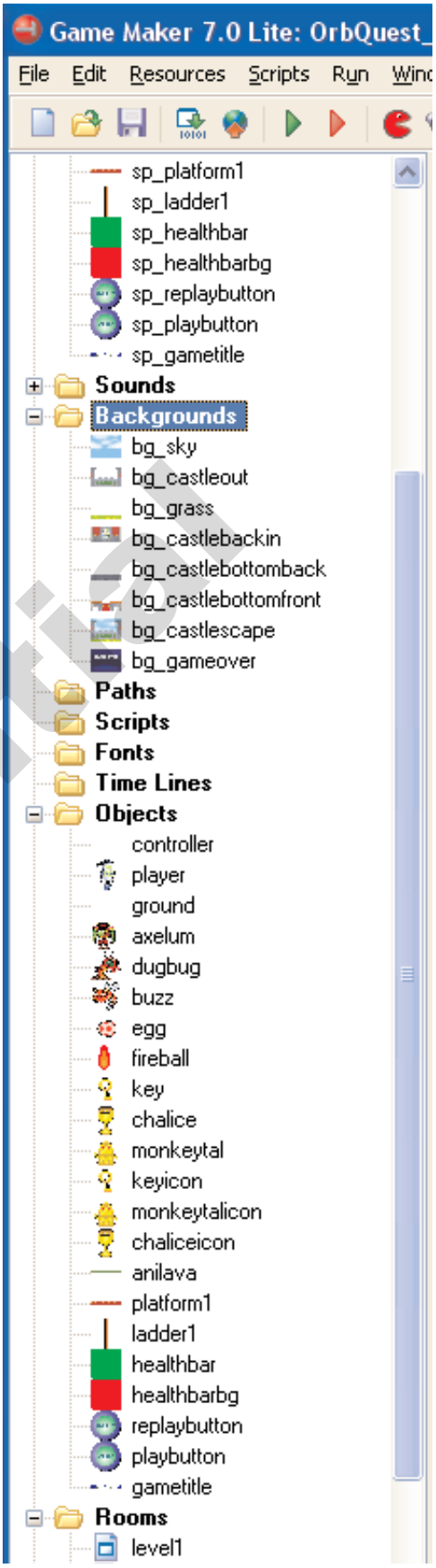
> **Note:** The starter file contains all the sprites, objects, rooms, backgrounds, and sounds that are needed in order to make a complete game. Also included are layered backgrounds set into various rooms and different sounds which will greatly enhance the students' games. As with the games created using Game Builders Academy's Video Game Design and Development: Level 1 Curriculum-In-A-Box, supplied Library resources can always be edited or new resources may be added in order to make the students' games individually personalized.

STEPS:

1. Navigate to the Student Files disk and open OrbQuest_Starter.gmk.
2. Using the File menu, left-click on Save As.
3. In the Save the Game window, navigate to your designated folder and name your file your-name1, replacing "your-name" with your first and last name.
4. Open the folders in the Library and examine their contents.

> **Note on Modules:** In Video Game Design and Development: Level 2, students will be doing actual hand-coding of their games, as opposed to the click-and-build method in the earlier introductory course. This increases the difficulty of the course of study, but also increases the options available to the students, and the rewards that come from mastering difficult challenges.
>
> To help identify and navigate the coding in later Modules, programming code added to existing code will appear as black text, while previously scripted code will appear as grey text. This convention only appears in the written Modules and will not appear in the Post-Modules or Starter File code windows.

END



4. Examine OrbQuest_Starter Library contents.

Module 2: Program Creation of Ground and Player

OVERVIEW:

In this Module, students will: Program creation of an instance of the player object; Program creation of an instance of the ground object.

INTRODUCTORY DISCUSSION:

In Video Game Design and Development: Level 1, we added objects to the room (level) directly. Although this is quick and easy, it ultimately limits what can be done. Many new possibilities open up when we begin programming the actual creation of objects (instances). In this Module, we will create an instance of the player and an instance of the ground via programming.

Brief Review - Use of Pseudocode: Remind students that pseudocode is used by the developers of all the games they play at home. Before trying to program something to happen in a game (or other type of computer program), it's helpful to think through the programming steps using a spoken language (English, Spanish, Chinese, etc.). Once the programming steps have been defined in plain language, then those steps are translated into programming code. There is no "right" pseudocode. Its only purpose is to clearly think through and define the steps in the programming task at hand.

STEPS:

Prep: Place controller object in level1 room.

1. In the Library, double-click the controller object (this will bring up the controller Object Properties window).

   Have students: Try to pseudocode the ground creation functionality. Tell students there is no "right" pseudocode - pseudocode is just words that describe how to accomplish a programming task.

   Example of pseudocode for this particular task:
   When the first level starts, the ground is created.

   Ask students: Now that we've got the pseudocode, we take the first part, "When the first level starts" - do we have that event already in our programming? ***answer:*** no. We will add it. For now we will use the Room Start Event.

2. In the Object Properties window, left-click the Add Event button (this will bring up the Event Selector window).

3. In the Event Selector window, left-click the Other button, then select Room Start from the flyout menu.

4. On the right side of the Object Properties window, in the Actions panel, left-click the control tab, then right-click the Execute Code icon (this will bring up a blank Code window).

5. In the blank code window, type the following code:

```
//CONTROLLER

//room start

//creation of ground

instance_create(
```

   Tell students: Look in the bottom portion of the code window and you'll see the code reference area. As soon as you start to type code Game Maker shows you the available functions based on what you typed. The reference window also shows you whether or not the function takes arguments and if so what arguments it takes. For more information,

continued on following page

FUNCTIONS

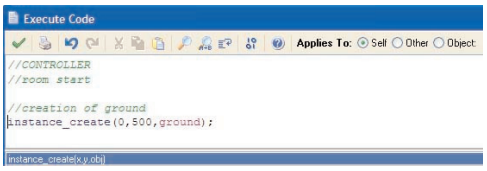
A function is a block of code that does something. Game Maker has many built-in functions. We can use built-in functions or we can write our own custom functions. In this Module will use one of the fundamental built-in functions in Game Maker -- the instance_create function.

Sometimes a function needs additional information to do its work, sometimes it does not. If the function needs additional information, the information goes inside of parentheses immediately following the function name. If the function does not need additional information, the parentheses are still written, but nothing is inside. The information inside of the parenthesis is referred to, in programming, as arguments. This will become very easy to understand as we work more with functions.

Examples:
do_something(argument or arguments);
do_something_else();
paintsomething(color,thingtopaint);
instance_create(x,y,object);

Note:
A semi-colon is written at the end of every function, after the close parentheses: instance_create(x,y,obj)**;**

For more information, refer to Appendix I: Programming Primer.



5,6. Type directly in the code window, note reference in bottom of window

Module 2: Continued from previous page

refer to sidebar on previous page, "Functions").

Ask students: Notice that the instance_create function has three things inside the parentheses. What are those things inside the parentheses? ***answers:*** information the function needs to do its work; arguments.

Ask students: What do you think those three specific things are. Think about the function name itself, instance_create, what does it do and what additional information might it need to do its work? ***answer:*** an x-position for the object, a y-position for the object, and the name of the object itself.

6. Type the remainder of the line of code as below:

```
instance_create(0,500,ground);
```

Explanation of code:
Create an instance of the ground at an x-position of 0 and a y-position of 500.

7. Test game and debug as necessary (ground object is set to not Visible in Object Properties window).

Have students: Try to pseudocode the player creation functionality.

Example of pseudocode for this particular task:
When the first level starts, the player is created.

Ask students: Now that we've got the pseudocode, we take the first part, "When the first level starts" - do we have that event already in our programming? ***answer:*** yes (the Room Start Event added in step 3).

8. In the Object Properties window, left-click the existing Room Start event.
9. On the right side of the Object Properties window, in the Actions list, double-click the existing execute a piece of code icon (this will bring up the code window).
10. In the existing code window, add the following code:

```
//CONTROLLER
//room start
//creation of ground
instance_create(0,500,ground);
//creation of player
instance_create(room_width/2,ground.y,player);
```

Explanation of code:
Create an instance of the player at an x-position in the horizontal center of the room and a y-position at the base of the ground.

Explain to students: Since we set the position of the ground in step 6, we can now use the ground itself to position other items in relation to the ground.

11. Test game and debug as necessary (player should appear in game window).

Explain to students: The Room Start event runs at the beginning of **each and every** room. We will use an "if" statement to associate these actions only with the first level of the game (see sidebar at right, "If Statement").

continued on following page

Graph Origin in Programming

In mathematics, the origin (0,0) of a grid (or graph) is in the center. In programming, the origin is in the upper left corner of a grid (or graph, or game window area).

room_width/2

In step 10 of this Module, we use room_width/2 to specify an x-position for the player. As arguments in a function, we can use numbers, and we can also use formulas. The formula room_width/2 yields a number that is the horizontal center of the room. Students will use this technique frequently throughout this course.

If Statement

An "if" statement is the first conditional statement students will learn and use. Conditional statements enable a program to take actions based on certain conditions. The "if" statement is the most fundamental conditional and it works the same as human decision-making. For example: "If it is raining, I will take an umbrella". Or "If I am hungry, I will eat." Students will use the "if" statement extensively in this course. For more information, refer to Appendix I: Programming Primer.

Module 2: Continued from previous page

12. In the existing code window, add the following code:

```
//level1 room
if(room == level1)
{
    //creation of ground
    instance_create(0,500,ground);

    //creation of player
    instance_create(room_width/2, ground.y, player);
}
```

Explanation of code:
If the current room is level1, execute the code within the curly braces.

13. Test game and debug as necessary (everything should work exactly as it did after step 10, the room check we just added will serve us later in the project).

END 

Persistent Object, or Re-Create Object in Each Room?

There are two primary ways to make an object appear in more than one room (game level). You can set the object to be "Persistent" by clicking the Persistent checkbox in the given object's properties window, or you can use a room check (if statement, as in step 12 of this Module) to create the object each time a room is created. Each method has its own uses. We will specify which method to use within the steps of the various Modules.

Module 3: Program Player Left and Right Movement using Parallax Scrolling

OVERVIEW:

In this Module, students will: Program functionality to make it appear that the player is moving either right or left when the corresponding arrow key is pressed on the keyboard.

INTRODUCTORY DISCUSSION:

In a side scrolling game, the player itself can move left or right, or the background(s) can move as the player stays in one position. If the background moves to the left the optical illusion is that the player is moving to the right and vice versa. For level 1 of this game, we will have the player always remain in the middle of the room as the background moves left or right creating the optical illusion of the player moving. This technique offers advantages that we will see as we move forward. We will start by creating simple movement, then starting step 8, we will introduce the parallax effect.

STEPS:

1. In the Library, double-click the controller object (this will bring up the controller Object Properties window).

   Have students: Try to pseudocode the player (background) movement functionality. (Note: We will use the arrow keys for left and right movement.)

   Example of pseudocode for this particular task:
   When the right arrow key is pressed, the background(s) moves to the left.

   Ask students: Now that we've got the pseudocode, we take the first part, "When the right arrow key is pressed" - do we have that event already in our programming? ***answer:*** no. (So we have to add it.)

2. In the Object Properties window, left-click the Add Event button (this will bring up the Event Selector window).

3. In the Event Selector window, left-click the Keyboard button (this will bring up the Keyboard events menu).

4. In the Keyboard events menu, choose <Right> (this is the event that occurs when the right arrow key is pressed on the keyboard).

5. On the right side of the Object Properties window, in the Actions panel, left-click the control tab, then right-click the Execute Code icon (this will bring up a blank code window).

   Tell students: To make it appear that the player is moving, we move the background(s) in the opposite direction.

   Ask students: To make it appear that the player is moving right when the right arrow key is pressed, what do we have to do? ***answer:*** move the background(s) to the left.

6. In the blank code window, type the following code to move all backgrounds at the same speed:

```
//CONTROLLER
//keyboard right
//moving the backgrounds
background_x[0] -= 3;
background_x[1] -= 3;
background_x[2] -= 3;
```

   Explanation of code: Move all backgrounds to the left 3 units.

7. Test game and debug as necessary (press

continued on following page

TOP-SCROLLERS SAME PRINCIPLE AS SIDE-SCROLLERS

In this game project, we create side-scrolling motion. The same exact principles and techniques can be used to create a "top-scroller." A top-scroller is a game in which the motion is vertical as opposed to horizontal. Examples of top-scrollers are overhead view two-dimensional driving games, and many older "space" games. The original arcade version of a driving game called Spy Hunter is a prime example of a top-scroller.

The only technical difference is that instead of the primary movement being programmed along the x-axis, primary movement is programmed along the y-axis. And, of course, backgrounds and other graphics need to be designed to tile and scroll vertically.

PARALLAX SCROLLING

Parallax scrolling is the technique (or principle) of moving different backgrounds at different speeds in relation to the viewer. The further away from the viewer a background is (or should appear) the slower it moves. Or, conversely, the closer a background (or object) is to the viewer, the faster it moves.

This is the optical illusion that humans percieve and that adds to the sense of depth in human vision. To simulate this real-world optical illusion on a two-dimensional computer screen (in a game), we program the backgrounds to move at different speeds. Backgrounds (and objects) closest to the viewer move fastest.

Module 3: Continued from previous page

Right arrow key, backgrounds should move to the left, creating the illusion that the player is moving to the right).

Remind students: We're about to try something new — what do we expect to happen? Expect it not to work. If it works ... that's gravy.

Have students: Notice that although the player stays in the middle of the screen, the optical illusion is starting to take effect - it appears as if the player is moving, even though it's the backgrounds that are moving. We will enhance this illusion by adding a parallax effect (see sidebar page 16, "Parallax Scrolling") to enhance the illusion of depth (three-dimensionality).

8. Bring up the code window from step 6, and change the code as below:

```
background_x[0] -= .75;
background_x[1] -= 2;
background_x[2] -= 3;
```

9. Test game and debug as necessary (press Right arrow key, backgrounds should move at different speeds creating illusion of depth along with player movement to the right).

   Remind students: We're about to try something new — what do we expect to happen? Expect it not to work. If it works ... that's gravy.

10. In the conroller Object Properties window, right-click on Keyboard Event for <Right> Key, and left-click on Duplicate Event.

11. In the Event Selector window, choose Keyboard <Left>.

12. In the Actions panel, double-click on Execute a piece of code action and make the following changes:

```
//CONTROLLER
//keyboard left
//moving the backgrounds
background_x[0] += .75;
background_x[1] += 2;
background_x[2] += 3;
```

13. Test game and debug as necessary (press Left arrow key, backgrounds should move at different speeds creating illusion of depth along with player movement to the left).

    Have students: Notice that player does not face left when moving left. One very easy way to fix that is to flip the image horizontally when we change directions. To do this, we set the image's xscale (see next step, and sidebar at right, "Flipping Horizontal and Vertical with xscale and yscale").

14. Bring up the code from step 6, and add the following to the existing code:

```
//player sprite direction
with (player) image_xscale = 1;
```

15. Bring up the code from step 12, and add the following to the existing code:

```
//player sprite direction
with (player) image_xscale = -1;
```

16. Test game and debug as necessary (press Left and Right arrow keys, player should appear to move realistically, facing in the direction of movement).

END

Player speed is currently hard-coded as 3, Using a Variable Offers More Flexibility

In this game project, the player's horizontal movement speed is always the same - a constant speed of 3 units. In programming, when a number is used as a parameter or property instead of a variable being used, it's referred to as "hard-coding." Hard coding is easier, but it has limitations.

Since the player's speed is hard-coded, it cannot be changed while the game is running. This means that the player cannot get faster or slower due to conditions (moving through mud, getting tired, running fast, etc.).

For the player's speed to be able to change during gameplay, the player's speed (currently 3) would have to be programmed (created and referenced) as a variable. For more information, refer to Appendix I: Programming Primer.

Flipping Horizontal or Vertical by Reversing X-Scale or Y-Scale

In any programming language that handles graphics, you can easily flip an image along it horizontal or vertical axis to give a "mirror" image. This is the technique we use in steps 14 and 15 of this Module.

To flip an image along either axis, set the scale along the given axis to 100% or negative 100%.

Each programming language will have its own specific code for the x-scaling and y-scaling. In Game Maker, 1 is 100% and negative 1 is negative 100%. Those are the numbers you see in steps 14 and 15.

Module 6: Program Creation and Movement of Player Projectiles

OVERVIEW:

In this Module, students will: Program functionality to create a player projectile, and for the player to throw projectiles in a specific direction (using angles).

Introductory Discussion:

Tell students: We will now program functionality for the player to throw projectiles at the adversaries.

Ask students: Who decides which key on the keyboard the player will press to throw the projectile? ***answer:*** we do, because we are the programmers.

Ask students: In PC games, what key is most commonly used to throw projectiles?" ***answer:*** spacebar (students can program a different key if they prefer).

STEPS:

1. In the Library, double-click the controller object (this will bring up the Object Properties window for the controller).

   Have students: Try to pseudocode the creation of the player projectile.

   Example of pseudocode for this particular task:
   When the Spacebar is pressed, a player projectile is created.

   Ask students: Now that we've got the pseudocode, we take the first part, "When the Spacebar is pressed" - do we have that event already in our programming? ***answer:*** no (so we have to add it).

2. In the Object Properties window, left-click the Add Event button (this will bring up the Event Selector).

   Ask Students: What Event do you think we choose in this case? ***answer***: Key Press (Keyboard can also be used - see sidebar, page 18, "Keyboard or Key Press").

3 In the Event Selector, left-click the Key Press button (this will bring up the Key Press events menu).

4. In the Key Press events menu, choose <Spacebar> (this is the event that occurs when the Spacebar is pressed). The <Spacebar> event should now appear in the controller object Events list.

5. On the right side of the Object Properties window, in the Actions panel, left-click the control tab, then right-click the Execute Code icon (this will bring up a blank code window).

6. In the blank code window, type the following code:

```
//CONTROLLER
//keypress space
//creation of egg
instance_create(player.x,player.y-15,egg);
```

   Explanation of code:
   Create an instance of the projectile (egg) object, set the x-position at player.x (horizontal center of player), set the y- position at 15 units above the bottom of the player (player.y-15).

continued on following page

There's a Lot Here

This particular Module comprises a number of fundamental academic and social subject areas. At it's fullest, this Module includes: a primary application of math via use of angles to determine direction; an introduction to basic physics via creating projectiles with motion; a fundamental discussion about ethics when dealing with adversaries (in a game and in the real world).

Opportunity for Ethics Discussion

In building this game, there is one main rule: We can stop an adversary but we cannot harm an adversary. How can we do this? This will yield many interesting answers, will serve as a fun and engaging discussion with the students, and will also get students on track in terms of how to apply this rule (can stop adversary but cannot harm adversary) in their game development.

After this discussion, tell students that for these first projectiles, we will throw food at the attacking adversaries. The logic is that animals, both real and fictitious, i.e. dragons, monsters, etc. like food, so we throw food at them (thus feeding them), and that "stops the adversaries without harming the adversaries."

Module 6: Continued from previous page

7. Test Game and debug as needed (press Spacebar, projectile should be created at the player's position).

   Have students: Notice that the projectile is created, but does not move. **Ask students:** Why doesn't the projectile move? ***answer:*** we haven't programmed it yet. (We'll program the motion event in the next steps.)

   Have students: Try to pseudocode the projectile movement functionality. Remind students there is no "right" pseudocode - pseudocode is just words that describe how to accomplish a programming task.

   Example of pseudocode for this particular task:
   When the projectile is first created, set it in motion.

   Ask students: Now that we've got the pseudocode, we take the first part, "When the projectile is first created" - do we have that event already in our programming? ***answer:*** yes, it's on the code for the Key Press <Spacebar> event (steps 4-6). So we find that event and add code to set the projectile in motion.

   Tell students: There are numerous ways to program motion. For now we will use a simple, built-in function.

8. In the Object Properties window, left-click the existing Key Press <Spacebar> event.

9. On the right side of the Object Properties window, in the Actions list, double-click the existing Execute Code icon (this will bring up the code window).

10. In the existing code window, add the following code on a new line, below the code from step 6:

```
//making egg move
with (egg) motion_set(0,15);
```

    Explanation of code:
    Set the projectile motion with a direction of 0 (right), and a speed of 15 (see sidebar at right, "Angles and Direction").

11. Test Game and debug as needed (press Spacebar, projectile should be created and immediately set in motion).

    Have students: Notice that the projectiles always go to the right, even if the player is facing left. We will fix this in the next step, using an "if" statement.

12. Bring up the code from step 10, and modify the code as below:

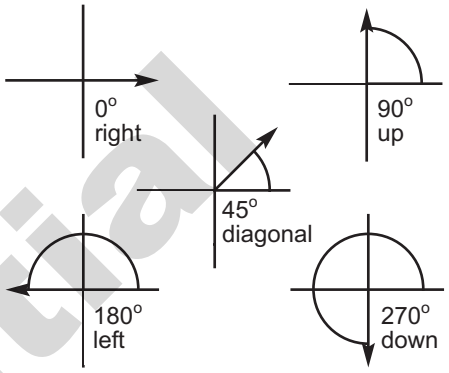
```
//check player direction is right
if (player.image_xscale == 1)
{
  //creation of egg
   instance_create(player.x,player.y-15,egg);
  //making egg move
   with (egg) motion_set(0,15);
}

//check player direction is left
if (player.image_xscale == -1)
{
  //creation of egg
```

continued on following page

Angles and Direction

In Game Maker and other programming environments, direction is based on the angles within a full 360 degree circle. Right is indicated by 0 degrees (360 will work in many programming environments, but in Game Maker, only 0 degrees can be used to specify right as a direction). Left is indicated by 180 degrees; up is indicated by 90 degrees; Down is indicated by 270 degrees. All incremental angles between 0 and 359 are usable - this allows very fine control of direction of motion in game programming.



Other Ways to Indicate Direction

In computer programming, there are two primary ways to specify direction: Degrees (based on angles) and Radians. Radians are more commonly used in higher-end programming and other mathematical and scientific application. But for purposes of class discussion and gameplay design, discuss with students the numerous other ways to indicate direction, such as: Compass directions (N,S,E,W); clock directions (6 o'clock to indicate behind, 12 to indicate in front, 9 to indicate right, etc.); longitude and latitude; most basic - left, right, up, down. There are other ways to indicate direction, but these examples should provide good jump-off points for class discussion.

Module 6: Continued from previous page


```
instance_create(player.x,player.y-15,egg);
  //making egg move
  with(egg) motion_set(180,15);
}
```

Explanation of code:
If the player is facing right (image_xscale == 1), create a projectile and set the motion with a direction of 0 (right), and a speed of 15; If the player is facing left (image_xscale == -1), create a projectile and set the motion with a direction of 180 (left), and a speed of 15.

13. Test Game and debug as needed (press Spacebar, projectile should be created and immediately set in motion, change direction, press Spacebar again to throw projectiles in other direction).

    Have students: Notice that projectiles that have already been thrown (and are traveling across the screen) actually change direction in mid-air if the player changes direction. This is because the "with (egg)" construction, controls **all** instances of the egg. One solution is to give each instance a new name as it's created. The code in the next steps affect each instance of the egg individually, as opposed to affecting all instances in the same manner (see sidebar at right, "The Difference Between Objects and Instances of Objects").

14. Bring up the code from step 12, and modify the code as below:

```
//check player direction is right
if (player.image_xscale == 1)
{
   //creation of egg
   newegg = instance_create(player.x,player.y-15,egg);
  //making egg move
   with (newegg) motion_set(0,15);
}

//check player direction is left
if (player.image_xscale == -1)
{
  //creation of egg
   newegg = instance_create(player.x,player.y-15,egg);
  //making egg move
   with (newegg) motion_set(180,15);
}
```

Explanation of code:
Same explanation as code in step 12, with one significant difference: each time a new instance is created, it is named "newegg" - then in the *with* construction, the newegg is specified instead of the "egg" as in the code in step 12.

15. Test Game and debug as needed (move the player in both directions, throwing projectiles as you move, also try jumping and pressing the Spacebar to throw projectiles in mid-air).

16. **Optional (and fun):** Have students try to program projectiles to be thrown in all four straight directions: up, down, left, right (have students use W,A,S,D keys on keyboard). The respective directions (in degrees) for up, down, left, and right are 90, 270, 180, 0.

END

The Difference Between Objects and Instances of Objects

An object is an item in the library, in the Objects folder. Objects never actually appear in a game. What appears in games are instances of objects (notice the name of the function: instance_create). An analogy is this: Think about a photocopy machine. The Object is the original document, and the instances are the photocopies of the original document.

This comes into play in steps 12 and 14 of this Module. In step 12, we use the object name (egg) to program the movement. This is why when one instance of the egg moves in a certain direction, **all** instances move in the same direction.

We solve this problem in step 14. To enable each instance of the egg object to have its own individual behavior, we name each new instance as it's created (newegg), and then we **use the new instance name, instead of the object name**. Refer to step 14 to see the revised code.

One Equal Sign Sets a Value, Two Equal Signs Checks a Value

In the if statement in step 14 (and other if statements), we use two equal signs (==) to check the value of the player.image_xscale. Then in the first statement within the curly braces, we use one single equals sign (=) to set the new instance name to "newegg."

This is the basic difference:
One equal sign **sets** a value.
Two equal signs **checks** a value.

Even experienced programmers, in the haste of writing code, often use the wrong number of equal signs in a given line of code. With experience, this is an error that students will learn to find and fix quickly. But in early stages of learning to program, this type of error, in particular, is common.