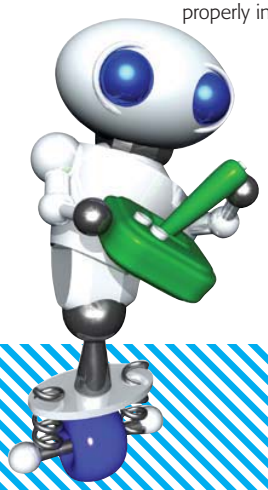
CHAPTER 5

Collision Theory and Logic

To use a computer system to create a game, you must first be able to speak the language of the computer. While it is not necessary to have a tremendous background in programming languages to be a designer, you will need to understand the basic principles of logic and collision theory. This chapter introduces the building of a logic statement and how to program objects to properly interact during a collision.



Objectives

After completing this chapter, you will be able to:

- **Use** game design software to create a playable video game.
- **Integrate** animated objects into a video game.
- **Create** sound and music effects in a video game.
- **Debug** a video game.
- **Describe** basic computer logic.
- **Build** applied mathematics logic statements.
- **List** features of object oriented programming.

Logic

The first concept of designing and programming a video game is an action-reaction relationship. To create a game environment that the player can control, the player's actions must cause something to change or react. This is the ***action-reaction relationship.*** Often, obstacles and challenges are placed within a game to force the player to take action.

CHEAT CODE: *COLLISION*
Collision is the most-used action command in game programming. Often substituted with *hit* or *touch*, a *collision* occurs when an object contacts something. This may be a player contacting an obstacle or other player. In may also be two obstacles contacting each other.

Programmers use logic statements to break down these action-reaction relationships. For example, if the action is ***colliding*** your go-cart into a banana peel, the reaction will be the go-cart spinning out, **Figure 5-1.** To begin this programming process, you will need to understand the five basic operators of a programming language: **IF**, **THEN**, **AND**, **OR**, and **ELSE**.

Basic Logic Statement

Two of the basic operators fit together to make a logic statement. A ***logic statement*** tests a condition and determines an action based on the result. An **IF**..**THEN** statement is the most basic example of a logic statement. The operator **IF** is used with the basic statement to test a condition. This is the *action* side of the logic statement. In the go-cart example, the action side of the logic statement is written as:

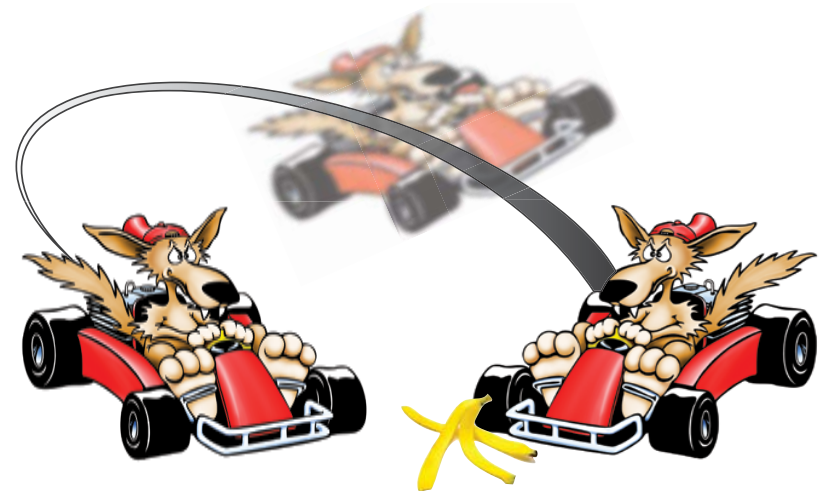


Figure 5-1. The go-cart spins out when it hits the banana peel. This can be written as an **IF**…**THEN** statement: **IF** the go-cart hits the banana peel, **THEN** it spins out.

IF the go-cart collides with the banana peel…

The ***reaction*** side of the logic statement describes what should happen when the condition is met. The reaction side of the logic statement for the go-cart is:

THEN the go-cart spins out.

Together, the action and reaction sides of the statement form a complete logic statement. The interaction between the go-cart and the banana peel is completely stated as:

IF the go-cart collides with the banana peel, **THEN** the go-cart spins out.

You see that **IF** an ***action*** occurs, **THEN** a ***reaction*** takes place. From this basic statement, more complex statements can be created.

Conditions and Events

The action-reaction relationship is everywhere and not just in games. You have several interactions every day. Looking at everyday interactions, they are defined in terms of ***cause and effect.*** Since cause and effect are exactly the same as action and reaction, the go-cart example can be rewritten in cause-effect language. Here, the ***cause*** is a banana peel in the road and the ***effect*** is slipping on the peel. Cause-effect relationships can be written in the same **IF**…**THEN** logic format. This looks exactly the same:

IF the go-cart collides with the banana peel, **THEN** the go-cart spins out.

The table in **Figure 5-2** shows some everyday cause and effect relationship you might encounter.

Operator	Cause	Operator	Effect
IF	"you turn in homework,"	THEN	you get a good grade.
IF	"you buy a ticket,"	THEN	you watch the movie.
IF	"you miss the bus,"	THEN	you are late to school.

Figure 5-2. This table shows the cause-and-effect relationship for some everyday occurrences.

Operator	Condition	Operator	Event
IF	action	THEN	reaction
IF	cause	THEN	effect
IF	condition	THEN	event

Figure 5-3. This table shows the relationship between the common terms for constructing a logic statement.

When programming a video game, the formal term for an action is a ***condition.*** The computer checks to see if a condition is met. When it finds a condition has been met, it executes the programmed events. An ***event*** is a change that occurs when a condition is met. In other words, it is a reaction to the condition.

To program the go-cart example using condition-and-event relationships, the logic statement is only slightly modified to describe the exact action the computer must take to carry out the command. That logic statement would read:

IF the go-cart object collides with the banana peel object, **THEN** the go-cart object will change from a linear animation to a spinning animation.

This is still the same idea, but more specific to help the computer identify the object and how it changes when the collision occurs. The table in **Figure 5-3** shows the relationship between the common terms used in constructing a logic statement.

To design a game, many events are required to get it to work properly. When programming a game, logic is used to do more than just describe what happens on screen. The same logic format is used to program the user interface, increase score, change levels, and perform every other interaction the player encounters. A simple user interface to control a player moving North, South, East, and West requires a logic statement for each controlling motion. Refer to **Figure 5-4.**

THINK ABOUT IT
ACTIVITY 5.1

In the workbook activities for Chapter 4, you created simulations of a soccer game. Think about these simulations. How could you write a logic statement to make the ball move on its own when kicked? How could you write a logic statement to describe how a goal is scored for each team?

Figure 5-4. This illustration shows logical statements for a player using the arrow keys to move a character.

Advanced Logic Statements

The next step in basic programming is to add multiple actions or multiple reactions to logic statements. This is done using the **AND** and **OR** operators. These operators work just as they would as conjunctions in any sentence.

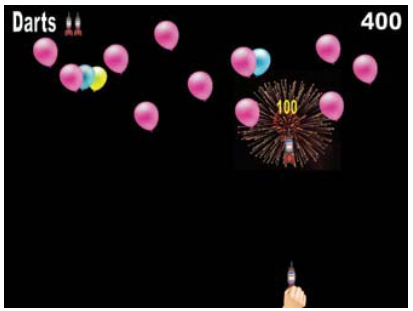
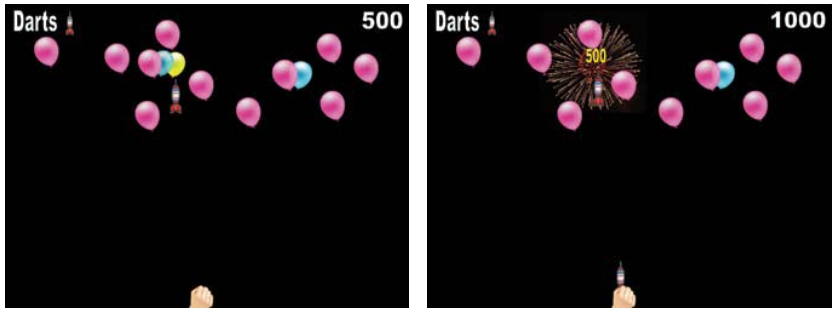


Figure 5-5. The action of the dart hitting the balloon creates the reaction of the balloon disappearing, an explosion appearing, and the score changing by 100 points. The balloon does not actually "pop." To create the illusion of popping, the balloon object is destroyed and replaced by a popping animation.

The **AND** operator will join two or more outcomes for a given condition or action. Refer to **Figure 5-5:**

IF the dart object collides with the balloon object,
THEN destroy the balloon object
AND create an explosion animation object
AND add 100 points to the player's score.

In this example, the **AND** operator allows three events to occur from a single collision action. The balloon is destroyed (1), an explosion appears (2), and the player scores 100 points (3). An **AND** operator can also be included in a condition statement. Refer to **Figure 5-6.**



A B

Figure 5-6. A—The dart is about to hit the yellow balloon. The yellow balloon is overlapped by a blue balloon, which is overlapped by a pink balloon. B—The dart pops the yellow balloon and any overlapping balloons with a single explosion animation; the player scores 500 points.

IF the dart object collides with a yellow balloon object
AND
IF the yellow balloon object overlaps any other balloon object,
THEN destroy yellow balloon object
AND destroy all balloon objects it overlaps
AND create an explosion animation
AND add 500 points to the player's score.

THINK ABOUT IT
ACTIVITY 5.2
Look at the dart in Figure 5-6B. When the balloon pops, the dart has not been programmed to stop or be destroyed. How do you think a logic statement should be written to describe what happens to the dart when it hits a balloon?

Just as with the **AND** operator, the **OR** operator works as a conjunction in programming logic. The **OR** operator allows multiple results to take place under a given condition or event. In the balloon pop game, a random balloon begins to deflate during gameplay. When it does, the existing balloon object is replaced with an animation of the deflating balloon and a small balloon underneath the animation. The small balloon is only visible when the deflating animation has finished. In this example, the deflating animation and the small balloon should be treated as if they were only one object. The **OR** operator is perfect for making this happen.

If the dart collides with either the deflating animation or the small balloon, the game should display the same events. The events need to "pop" both the animation and the small balloon underneath. This operation would look something like the example below. Refer to **Figure 5-7.**

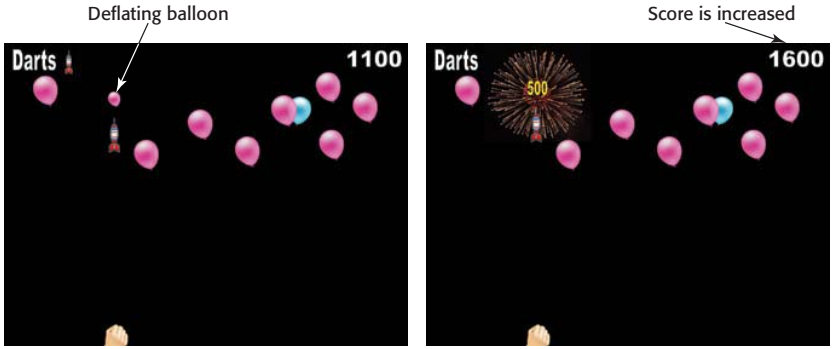


Figure 5-7. A—The dart is about to hit the small balloon. B—When the dart hits the small balloon, the balloon pops and the player scores 500 points.

IF the dart object collides with a small balloon object
OR
IF the dart object collides with a deflating balloon animation,
THEN destroy the small balloon object
AND destroy the deflating balloon animation
AND create explosion animation
AND add 500 points to the player's score.

The last of the basic programming operators is the **ELSE** operator. This operator may also be called the **OTHERWISE** operator. The **ELSE** operator describes what will happen if a certain action or reaction does ***not*** take place. You have likely seen this many times when trying to beat a level in a video game.

Think about a game that requires you to collect gold coins and a key. You cannot open the door to the next level without meeting both objectives. The doorway will usually display a message telling you what you are missing, **Figure 5-8.** In the example below, the condition tests if the player has at least 100 gold and one key.

IF gold > = 100
AND
IF key = 1,
THEN display the message "Well done. You may pass to level 2,"
ELSE display the message "You need 100 gold and the key to pass."

The **ELSE** operator works like a true/false test. If the condition is true, the **THEN** events are initiated. If the condition is false, the **ELSE** events are initiated. In the balloon pop example, the **ELSE** operator helps end the game when the player runs out of darts. Every time a dart is launched, a test needs to be performed to see if there are any more darts. In other words, the question is asked, is number of darts more than zero? If the condition is true, then a dart needs to be loaded into the hand (avatar). If the condition is false, the game ends.

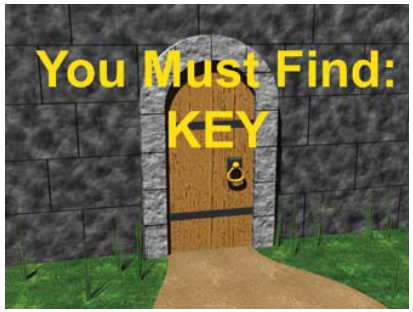


Figure 5-8. A game should tell you what is missing when trying to complete an objective. Here, the game is telling the user the key must be located to open the door and enter the next level.

Figure 5-9. When a dart is used, the computer looks at the number of darts to see if more than 0 darts are available. Here, the result of the test is **FALSE** since the last dart has been used. The computer then initiates the **ELSE** operation to display the game over screen.

The following logic statement can be used to program the end of the balloon pop game.

IF number of darts > 0,
THEN load one dart in the hand,
ELSE display the message "Game Over."

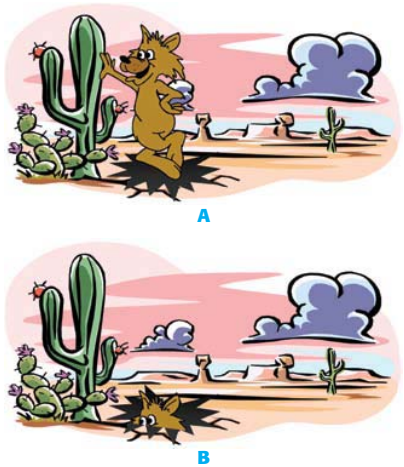
This statement describes what happens each time the player throws a dart. If there is still a dart available, then the player gets to throw another dart. Otherwise, the game is over, **Figure 5-9.**

Collision Theory

The most used condition in video game design is ***collision.*** You may guess the concept of ***collision theory*** deals with an object running into or hitting another object. It does. However, also included in collision theory is the idea that when objects collide the movements, animations, and events must provide an illusion of reality.

One of the most difficult concepts for beginning designers to grasp is that a picture of an item does not act the same as the real item. When programming a game, the fact that an object ***looks*** like a wall does not make it ***act*** like a wall. For it to act like a wall, the object must be programmed with the ***properties*** of a wall.

CHEAT CODE: *PROPERTIES*
Images used in video games are programmed to work properly by setting the object's properties. Visibility, interactivity, and movement are some of the properties assigned to an object to make it act like a real item.



A

B

Figure 5-10. A—The coyote can stand on the black spot because the spot has not been programmed to be a hole. B—The coyote now falls into the hole. The difference is collision programming to make the coyote interact with the hole object.

A good example of how an image is not a real object can be found in cartoons. The old cartoon trick is to paint a black spot on the ground, **Figure 5-10.** The black spot looks like a hole, but it is just paint and you should be able to simply walk right over top of the black spot. The key here is interactivity. ***Interactivity*** is how one object behaves when it encounters another object. In the cartoon, the interactivity is defined so the black spot actually functions like a hole. When a character walks onto the black spot (interacts with the hole), they fall into the hole.

The black spot is just an image unless you tell the computer to make it act like a hole. To create the properties of a hole, the hole must be programmed so the computer knows how to react when the player comes in contact with the spot. The programming interaction would look something like this:

IF the coyote collides with the black spot,
THEN the coyote falls.

The black spot is still not an actual hole, just an object that triggers a fall event by the coyote. This provides the ***illusion*** that a black spot is really a hole.

In the cartoon, the coyote falls through the painted hole, while the roadrunner is able to pick up the black spot and run away. Here, the hole reacts differently for two different characters. This would be contrary to collision theory. The hole should act like a hole for all the characters unless one has a magical ability or flight. Anything else would be a glitch.

Collision theory works throughout the game environment. Every object including the background must be programmed to look, feel, and act like it should. Imagine a scene from the ***Spiko the Hedgehog*** game. During gameplay, Spiko jumps onto a grassy platform, **Figure 5-11.** You expect Spiko to stay on the platform and walk over to the coin. Instead, Spiko falls through the platform and off of the screen. What is going on here? This common glitch happens when the designer forgets to apply collision theory to the entire scene.

The designer needs to program the platform to act like a solid object. That is to say, **IF** the character collides with the grassy platform, **THEN** Spiko stops falling. The gravity setting makes Spiko fall until he collides with an object programmed to act solid. When an object has no collision statement, it will not alter the character's movement.

A ***collision statement*** must exist for each object the player touches. If no collision statement exists, then the player cannot interact with it. Take the example of a player flying an airplane. There is no collision statement for the sky or the clouds. This allows the airplane to fly through these objects without any reaction. However, if the airplane collides with a bird, then the engine would sputter and plane would lose altitude. Therefore, birds are programmed to trigger interaction events when touched. In other words, the bird objects have a collision statement.

Remember, just because an object looks like a dart does not mean a balloon will pop if the dart touches it. In the balloon pop game, if an event is not associated with the condition:

IF the dart object collides with the balloon object

then nothing will happen when the dart hits the balloon. No events will occur at all, no balloon pop, no explosion, and no increase in score. The computer has no way of knowing the proper event unless you tell it exactly what to do and how to do it.

This can be a difficult topic to understand. It is easy to think that if an object ***looks*** like grass, then it should ***act*** like grass. That is true in real life. A real grassy surface stops you from falling to the center of the earth. But, this real-world logic does not apply in a video game. In a video game, the grass is just an ***image*** of a grass. The object will only ***act*** like real grass if the designer programs it to do so. Every interaction with the grass object needs to be programmed to react as though it came in contact with real grass.

CHEAT CODE: *COLLISION STATEMENT*
A collision statement is a logic statement that has the condition side of an event begin with two or more objects colliding.

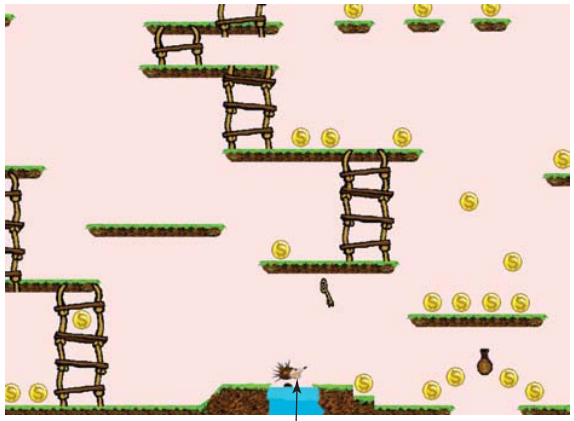


Figure 5-11. In this platform game, *Spiko the Hedgehog*, the main character must walk on the grassy platforms to reach the gold. However, notice the glitch in this game. Spiko can walk on the water hole, when he should sink. The collision with the water was incorrectly programmed to act as solid platform.

THINK ABOUT IT
ACTIVITY 5.3
Think about a racecar game and how the player's car must interact with the other cars on the track. What do you think will happen if you do not program a collision between the player and the other cars? Would that be fun?

Of course, you do not have to program objects to act the way they do in the real world. Some surreal and fantasy games program unusual properties for objects that appear as real-world objects. For example, you may program a road to act as a river. On the other hand, you may have a waterfall act like an elevator.

Programming with Collision Theory

Looking at the balloon pop game, a balloon pops when it collides with the dart. This appears to be one event triggered by one condition. However, it is actually a series of events activated when the dart object collides with the balloon object. When the computer recognizes this collision, it sets into action the events programmed by the logic statements. Shown in **Figure 5-12** is an event frame used in The Games Factory 2. This is an object-oriented, game development software. The event frame shows the programming of a collision condition and the resulting events.

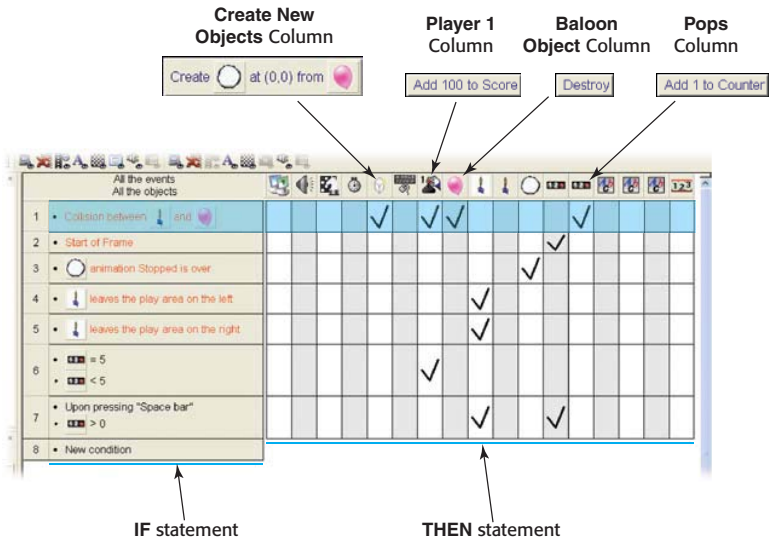


Figure 5-12. This is an event view from The Games Factory 2. Examine this to see how a logic statement is constructed in the software.

Look at line 1 in the event frame. The **IF** side of the logic statement is in the first column. The **THEN** side of the logic statement is shown in the remaining columns. The condition on line 1 states "collision between dart object and balloon object." When that condition is met, the computer processes the events in the **THEN** statement.

Notice the four check marks in line 1 in the event frame. The first check mark is in the **Create New Objects** column. This event creates a new explosion animation object at coordinates 0,0 ***relative*** to the balloon object. The next check mark is in the **Player 1** column. This event adds 100 points to Player 1 score. The next check mark is in the **Balloon Object** column (the name of this column matches the name assigned to the object). This event is set to destroy the balloon object. The last check mark is in the **Pops** column. This event increases by one the counter keeping track of the number of pops. To see this type of object-oriented programming as a logic statement, add the word **IF** before the condition and the word **THEN** before an event. See **Figure 5-13.**

CHEAT CODE: *RELATIVE*
The term relative is used when placing or moving an object in a game to describe from where the position is determined. When you place a duck three units to the left of a frog, you are placing it relative to the position of the frog. If something is placed at coordinates 0,0 relative to an original object, it is in the exact same spot as the original.

THINK ABOUT IT
ACTIVITY 5.4
Think about creating logic statements for the interactions programmed in Figure 5-12. Examine the conditions programmed on lines 6 and 7. These lines have AND/OR operators between the two conditions on each line. Which conjunction works best in each line?

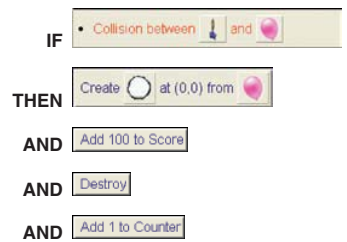


Figure 5-13. Logical operators can be added to help explain how a logic statement is constructed in The Games Factory 2.

Remember, collision theory is more than just setting collision events. To make objects appear solid, the programmer needs to add some realistic effects to the collision. Think about what happens when someone walks into a glass door. Do they just stop or do they bounce with their head whipping back and arms flailing? Adding a realistic animation after the collision will help with the illusion that an object is solid. An example of that programming might look like this:

IF the coyote collides with the brick wall,
THEN the coyote will move backward
AND the animation will change from walking to falling down.

Collision theory controls almost every interaction in video game action. The computer follows the programming of the collision statements that keep objects moving, stopping, exploding, or standing on a platform. Just because it is blue and has waves, does not make it water. The computer does not make these types of visual assumptions; only programmers

THINK ABOUT IT
ACTIVITY 5.5
Think about how collision theory works in a bowling game. What collisions need to take place? How would collision theory apply when programming the falling of the bowling pins?

do! The computer would be just as happy allowing a character to walk on water and sink in land than to do it the way we see in nature. The game world is yours to create. If you want people to walk upside down or walk on liquids, then program interactions in that way.

Writing a Logic Statement

Consider the following situations. Read the statement and then determine an appropriate logic statement for the situation. On a separate piece of paper, write the logic statements to describe the conditions and events.

1. The grasshopper jumps on a piece of food and the player earns 100 points.
   IF _____ collides with _____, **THEN** add _____ to score.
2. The grasshopper jumps on a lily pad and does not fall into the water.
   IF _____ collides with _____, **THEN** _____ stops.
3. The grasshopper runs into a mushroom and falls into the water.
   IF _____ collides with _____, **THEN** _____ movement falls **AND** _____ loses one life.
4. The grasshopper runs into a four-leaf clover and earns 50 points and an extra life.
   IF _____, **THEN** _____ **AND** _____.
5. The player achieves 10,000 points and receives a bonus extra life.
   IF _____ equals _____, **THEN** add _____ to the number of remaining _____.

The Games Factory 2

The Games Factory 2 (TGF2) is a game engine developed by Clickteam. It is a powerful game engine that uses object-oriented programming to make two-dimensional video games. The games can be large, multilevel, complex games as TGF2 is very powerful. Two extremely beneficial features of TGF2 are:

- The game engine runs very well on a standard PC.
- The game engine is very user friendly.

The next sections take a look at how the user interface of the software is organized, **Figure 5-14.** Understanding the interface is the first step in building some exciting games. Detailed explanations of each tool are given in the workbook lessons.

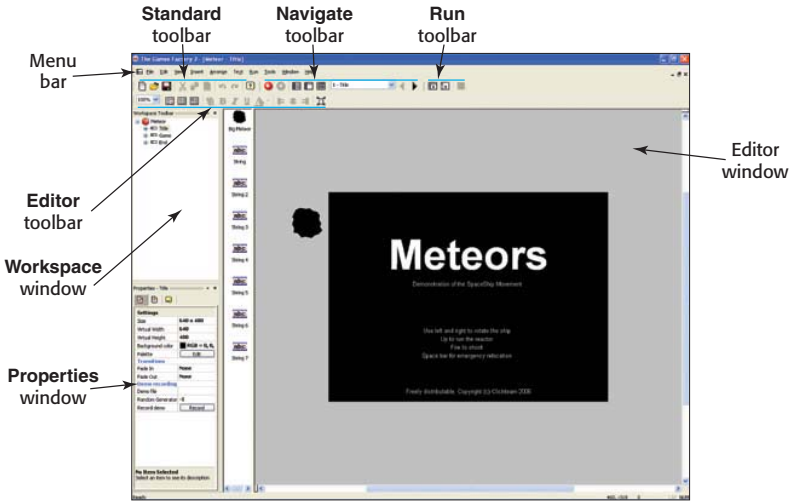


Figure 5-14. This is the basic interface of The Games Factory 2. Learning where the tools are located will help you become more efficient at designing games in the software.

CASE STUDY:
GAME PROGRAMMER

A game programmer is the person who can talk the language of the computer. Using that language, the programmer asks the computer to perform tasks. This person is the "computer guru" who takes the design ideas and makes them happen in the game.

A programmer is the person who takes game design ideas and gives the computer the instructions needed to make the game function.

If you like math and are good in algebra, geometry, calculus, applied mathematics, and computer science, you would probably be a good game programmer. To be a good game programmer, you need to be skilled in math, logic, and problem solving.

As a game programmer, you will work with three types of programming languages: computer platform–specific languages, scripting languages, and object-oriented programming languages. Computer languages like C, C++, C#, Java, and assembly are popular for designing games. These languages can "speak" directly to the computer operating system. Scripting languages like Python, Ruby, and Perl are simplified languages. They are easier for a person to use than a computer language. When a script is finished, it is compiled into a computer language so the computer can read it. The last language type is an object-oriented language. These are very simple, user-friendly languages that build in

(Continued)

(Continued)

a script language and are compiled into a computer language. This type of programming language is used by The Games Factory 2.

Game programmers are very highly recruited by game design companies. These people have important skills that can be used to build games, proprietary software, and other tools outside of gaming. A programmer straight out of school would likely enter a company as a junior programmer. There they would learn how to function best on the programming team and learn how to make programs needed by the company. Later, they would be promoted to a game programmer and have duties to work with the design team and share ideas on what a game could be programmed to do.

Several programming languages are common in game programming.

Other game programming-related jobs include lead programmer, technical director, AI programmer, software engineer, network engineer, graphics engineer, and engine programmer. All of these higher-level jobs offer more responsibility and leadership in the projects being created.

Most game programming jobs require a college degree in computer science, game programming, or software engineering. In 2009, the average salary of a game programmer was between $65,000 and $85,000 a year, depending on experience and geographic location.

Menu Bar

The menu bar contains the pull-down menus. Clicking on the name of a pull-down menu displays the menu. You can then select the tool from the menu. The menu bar and pull-down menus function just as they do in other Windows programs, such as Microsoft Office.

Standard Toolbar

The **Standard** toolbar contains the most common tools found in the pull-down menus, but displayed as buttons for easy access. Basic tools such as **New**, **Open**, **Save**, **Cut**, **Copy**, **Paste**, **Undo**, **Redo**, and **Contents** (help) are located on the **Standard** toolbar. This places the tools just a mouse click away.

Navigate Toolbar

To move quickly from one area of the game programming to another, use the tools on the **Navigate** toolbar. Tools included on this toolbar are **Back**, **Forward**, **Storyboard Editor**, **Frame Editor**, **Event Editor**, **Previous Frame**, and **Next Frame**. Also included on this toolbar is the frame identification and selection drop-down list.

Run Toolbar

To see how your game is working, it is helpful to the tools on the **Run** toolbar. These tools allow you to test the game to see if everything is working as anticipated. The **Run Application** tool allows you to test play your game from the first frame. The **Run Frame** tool allows you to test play just the current page you are designing. Use the **Stop** button to cancel the **Run Application** or **Run Frame** tool and continue working on your game.

Editor Toolbar

There are two basic **Editor** toolbars. Which toolbar is displayed depends on which mode or view is currently displayed. One version of the **Editor** toolbar is displayed in frame view. The second version is displayed in event view.

In frame view, the editor toolbar contains the tools needed to view all the aspects of your frame creation and conditional programming. The **Zoom** tool allows you to see your work in greater detail by magnifying the view. This can help to properly align your background and character features. The **Zoom** tool is also used to reduce the view. Other options include applying a grid to the editor window and tools for controlling font and style, text color, and alignment of text. The last button is the **Center Frame** tool. This allows you to quickly have the background view centered on any selected object. This is helpful when using a large or scrolling background.

In event view, the **Editor** toolbar displays different tools. The **Zoom** tool is still available, but it appears slightly different. The other tools on the toolbar help the designer to view or exclude from view events and objects. This is very helpful when designing a large game and the designer needs to focus on a single programming element of the game or on a small set of features or objects.

Workspace Window

The **Workspace** window displays the programming tree for the game. The application is the top-level branch. Below that, each frame is displayed in order as separate branches. The branch for each frame can be expanded to display each object used in the frame as branches below the frame.

Think of the tree organization format as you would a real tree. There is a trunk that has branches, twigs (sub branches), and leaves (the final objects). In a program tree, you start with a large file, or trunk folders. From there, the trunk folder has smaller folders, or branches, that contain similar items grouped by categories. Inside each branch folder, there are often more folders that contain even more specific categories. Finally, there will be actual files or applications at the end of the tree, similar to the leaves at the end of a real tree branch.

The **Workspace** window allows the designer to quickly access each part of the game. When an item is selected in the **Workspace** window, its properties are displayed in the **Properties** window (discussed in the next section).

Properties Window

The **Properties** window displays the physical features and properties of any selected object or frame. This window is where the designer sets the size, color, and movement properties. **Figure 5-15** shows the **Properties** window for three different types of objects. **Figure 5-15A** is for a background object, **Figure 5-15B** is for an active object, and **Figure 5-15C** is for a frame. Notice that different tabs are displayed, depending on what is selected. There are many different properties, but not every object has every property.

Library Window

Objects preloaded into TGF2 are stored in the ***library.*** The **Library** window, when displayed, allows you to quickly drag-and-drop objects from the library into the editor window. The items in the library are presented in a standard tree format. In the **Library** window shown in **Figure 5-16,** the branch folder (Games) narrows to a smaller branch folder (Miscellaneous). Inside of the Miscellaneous folder are the leaves (object files).

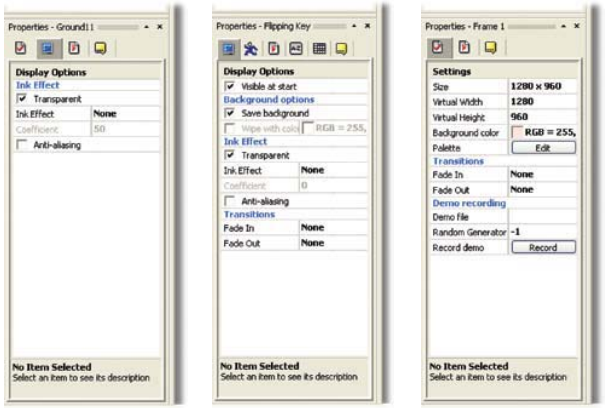


Figure 5-15. The **Properties** window in The Games Factory 2 shows different information depending on what object is selected. A—Background object. B—Active object. C—Frame.

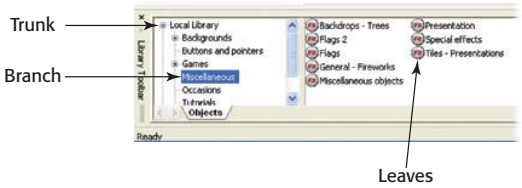


Figure 5-16. The library in The Games Factory 2 is arranged in a tree organization format. The tree format is commonly used in many different computer applications.

Chapter 5 Review Questions

Answer the following questions on a separate sheet of paper or complete the digital test provided by your instructor.

1. Briefly describe the ***action-reaction relationship*** in a video game.
2. What is the function of a logic statement?
3. The most basic example of a logic statement is the _____..._____ statement.
4. If a person slips on a banana peel and falls, the banana peel is the _____ and the fall is the _____.
5. In a video game, the formal term for an action is _____.
6. What is an ***event?***
7. Which logical operators are used to have multiple conditions or events?
8. Which logical operator is used to initiate an event when a condition is ***not*** met?
9. What are the two components of ***collision theory?***
10. Define ***interactivity.***
11. What is a ***collision statement?***
12. If object B is placed at 0,0 relative to object A, where is object B?
13. How can collision theory help provide the illusion of realism?
14. The Games Factory 2 is a powerful _____-oriented game engine used to create 2D game.
15. Describe how a tree organization format, such as the object library from The Games Factory 2, is arranged.

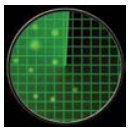
Cross-Curricular STEM Activities

1. Look at the historical events below. Research the cause-and-effect relationship that lead to the end result. Write a logic statement and a half-page summary of each action.

A. Beginning of World War I.

D. Emancipation Proclamation.

B. Sinking of the Titanic.



E. Development of RADAR.

C. Cuban Missile Crisis.

2. Real-world attributes like gravity need to be programmed into a game. Research gravity and create a PowerPoint presentation of five to ten slides to describe how different objects are affected by gravity on Earth. Discuss objects that are large, small, light, dense, in liquid form, in solid form, etc.

3. Consider the simple game of musical chairs. Write the rules and a game script using logic statements for this game. Test the game script with a few friends to make sure you have included all possible interactions.

4. Form into groups of two or three. Research, debate, and form a group opinion on each of the Think About It activities in this chapter. Prepare a PowerPoint presentation of ten slides (five to seven minutes) to present to the class explaining the group's opinions for each Think About It activity. Include text, pictures, video, animations, and slide transitions as appropriate to help explain your positions.