CHAPTER 24

Completing the Game

Welcome to the final chapter in the DungeonTrap game build. Up to this point, we've built a number of classes and enumerations creating a hierarchy of class functionality. The end result is a number of game objects we demonstrated at the end of Chapter 23. At the start of this chapter, we're going to update the project with a new version of the `ScreenGame` class.

We'll do this by copying and pasting a set of new classes including a new `ScreenGame` class into your project. This version of the file has all of the class fields and a number of more basic methods already defined. It's also properly stubbed out to allow the project to compile. We'll consider this to be Phase 1 of this chapter and our starting point.

The class methods that are set up by default are listed in the following in groups. Please take a moment to review these methods once you're done updating your project with the new classes. Details on how to do so follow.

Listing 24-1. ScreenGame Class Default Methods 1

```
//Main Methods
public ScreenGame(GameStates State, GamePanel Owner) { ... }

public void MmgDraw(MmgPen p) { ... }
public boolean MmgUpdate(int updateTick, long currentTimeMs, long
msSinceLastFrame) { ... }

public void RemoveObj(MmgObj obj) { ... }
private void UpdateClearObjects() { ... }
public void UpdateRemoveObj(MdtBase obj, MdtPlayerType p) { ... }

//Support Methods
public GameType GetGameType() { ... }
public void SetGameType(GameType gt) { ... }
public int GetOppositeDir(int d) { ... }
```


V. G. Brusca, *Introduction to Video Game Engine Development*, https://doi.org/10.1007/978-1-4842-7039-4_24

```
public boolean GetPlayer1Broken() { ... }
public MmgVector2 GetPlayer1Pos() { ... }
public boolean GetPlayer2Broken() { ... }
public MmgVector2 GetPlayer2Pos() { ... }
public void LoadResources() { ... }
public void UnloadResources() { ... }
```

Take the time to read through and understand these class methods. We won't go over them during the `ScreenGame` class review phases, so familiarize yourself with their functionality now. The `ScreenGame` class is the heart of the game, providing a connection between game items, objects, and characters, both players and enemies. We'll continue using the strict class review method that we've used for more complex classes.

ScreenGame: Phase 1 Demo

The Phase 1 copy of the `ScreenGame` class is the starting point for this chapter. We'll add functionality to the class until we get to Phase 2; then we'll stop and demo the game. We'll pick things up once again and add game logic to the project, which will bring us to Phase 3 and a complete game.

Navigate to your game engine's project folder. Locate the following directory: "cfg/asset_src/dungeon_trap_chapter24_phase1." Copy and paste the classes from the folder into your game project. Make sure to adjust the values of the `package/namespace` of the newly pasted files to match your project's setup. You should see something similar to the following screenshots when you run the project.

The main difference between this version of the ScreenGame class and the Chapter 23 demo is that we now have access to the MdtEnemyWave, MdtUiHealthBar, and MdtUiPoints classes. These are simple classes I want you to review on your own along with the default methods provided in the Phase 1 ScreenGame class. These classes are located in the 'asset_src/dungeon_trap_chapter24_phase1' folder found in the 'cfg' directory of the game engine's project directory. This folder contains all the files necessary to complete Phase 1 which adds some helper classes and ScreenGame class methods to the project to move things forward faster. You should copy these files into your project and review the recommended classes and methods. This step is preparation for the chapter's activities and should be completed before proceeding to Phase 2. Once you give everything a good once-over, we can start the Phase 2 review.

The Phase 2 review will add functionality to control the game's display both on a large scale by changing the screen state and on a small scale by adjusting HUD/UI elements.

ScreenGame: Class Review, Phase 2

As mentioned in the preceding, we've copied in some new classes to get to a good starting point. This will accelerate the development of the game by giving us a head start. If you want access to just the non-ScreenGame classes, you can find them in the "cfg/asset_src/dungeon_trap_other_classes" folder in the game engine's main project folder.

Now that that's all done, we can start the class review. The ScreenGame class has static class members and enumerations to cover, so we'll start the review process there. While we won't cover every method like we usually do, we will review every single class field.

ScreenGame: Static Class Members

The ScreenGame class has a few static class members for us to review before we look at the pertinent class enumerations.

Listing 24-2. ScreenGame Static Class Members 1

```
01 public static int GetSpeedPerFrame(int speed) {
02     return (int)(speed/(DungeonTrap.FPS - 4));
03 }
```

The snippet of code for us to look at has one method, `GetSpeedPerFrame`. This method is used to calculate how many pixels per frame an object has to move to maintain the specified speed. The next set of static class members for us to look at are the static class fields.

Listing 24-3. ScreenGame Static Class Members 2

```
public static int BOARD_BOTTOM = GAME_BOTTOM - MmgHelper.ScaleValue(56);

public static int BOARD_LEFT = GAME_LEFT + MmgHelper.ScaleValue(20);

public static int BOARD_RIGHT = GAME_RIGHT - MmgHelper.ScaleValue(132);

public static int BOARD_TOP = GAME_TOP + MmgHelper.ScaleValue(106);

public static int GAME_BOTTOM = MmgScreenData.GetGameBottom();

public static int GAME_LEFT = MmgScreenData.GetGameLeft();

public static int GAME_RIGHT = MmgScreenData.GetGameRight();

public static int GAME_TOP = MmgScreenData.GetGameTop();

public static int BOARD_WIDTH = BOARD_RIGHT - BOARD_LEFT;

public static int BOARD_HEIGHT = BOARD_BOTTOM - BOARD_TOP;

public static int GAME_WIDTH = GAME_RIGHT - GAME_LEFT;

public static int GAME_HEIGHT = GAME_BOTTOM - GAME_TOP;

public static int SRC_PLAYER_1 = GameSettings.SRC_KEYBOARD;

public static int SRC_PLAYER_2 = 255;
public static int WAVE_COUNT = 12;
```

As you can see, the fields are used to map out the game board as well as the entire game area with static class fields. You can also access the dimensions, width and height, of the game's board and display space. The next two fields, `SRC_PLAYER_1` and `SRC_PLAYER_2`, are used to mark input as coming from one player or the other. This will show up in the user input processing methods. The last variable listed, `WAVE_COUNT`, controls how many levels, waves of enemies, the game supports. The specifications, in Chapter 15, call for 10 waves; but I felt that 12 was a better value.

ScreenGame: Class Fields

The next section in the class review process covers the class' fields. The ScreenGame class has numerous fields for us to look at. We'll review them all in detail because it gives us a good idea of what the class does and what UI features it controls. The first group of class fields for us to review are listed in the following group.

Listing 24-4. ScreenGame Class Fields 1

```
private MmgBmp bground;
public MmgBmp doorBotLeftLockIcon;
public MmgBmp doorBotRightLockIcon;
public MmgBmp doorLeftLockIcon;
public MmgBmp doorLockFull;
public MmgBmp doorLockIcon;
public MmgBmp doorOpenFull;
public MmgBmp doorRightLockIcon;
public MmgBmp doorTopLeftLocked;
public MmgBmp doorTopLeftOpened;
public MmgBmp doorTopRightLocked;
public MmgBmp doorTopRightOpened;
private MmgVector2 screenPos;
public MmgContainer gameObjects = null;
public MmgContainer gameItems = null;
public MmgContainer gameEnemies = null;
```

The first block of class fields for us to review, listed in the preceding, contains entries necessary to display the game board and some game features. The game uses a split screen with a static game board on the left and the game HUD on the right. The bground field holds the board's background image. The subsequent 11 fields are all used to display either a locked or open door.

In the DungeonTrap game as you progress through the levels, different doors open or close to let enemies in. The images listed here are used to drive that functionality. The screenPos field is an MmgVector2 object instance that points to the game screen's top-left corner. The next three fields listed are used to hold the different game objects that are generated and placed into the game.

The gameObjects field is used to hold objects that extend the MdtObj class. The gameItems and gameEnemies fields are used to hold MdtItem and MdtCharInter object instances, respectively. The containers give us a placeholder to add and remove game objects during game play without disturbing the order certain game objects are rendered to the screen.

Listing 24-5. ScreenGame Class Fields 2

```
private MmgSprite enemyBansheeFrames = null;
private MmgSprite enemyDemonFrames = null;
private MmgSprite enemyWarlockFrames = null;
private MdtCharInterPlayer player1;
private MdtCharInterPlayer player2;
private MmgSound sound1 = null;
private MdtObjTorch torch1;
private MdtObjTorch torch2;
private MdtObjTorch torch3;
private MdtObjTorch torch4;
public MmgBmp gameLogo = null;
public boolean playerSnapToFront = false;
public int frameMsPerFrameMoving = 100;
public int frameMsPerFrameNotMoving = 300;
```

In the second block of class fields, we have entries that power the enemy characters, the players, and more. The first three entries are MmgSprite instances that are used to power the frames of the enemy characters. The next two entries, player1 and player2, are the player character classes used by human players during the game. The sound1 field powers the game's only sound effect that plays when a player damages an enemy character.

The next four entries, torch1–torch4, are used to help create the level's atmosphere by flickering throughout the game. The gameLogo field is used as part of the game's HUD. It's displayed at the top of the HUD to add a little style to the game's data display. The playerSnapToFront field is a Boolean flag that indicates if the player characters should snap to the front-facing direction when no direction pad input is being received.

The next two fields are also used as part of the player and enemy characters' movements. The frameMsPerFrameMoving field is used to speed up the character's walking frames so that they appear to be walking faster as the character moves across

the board. Similarly, the frameMsPerFrameNotMoving field is used to slow down the character's walking frames when they are not moving.

Listing 24-6. ScreenGame Class Fields 3

```
private MmgBmp bgroundPopupSrc;
private Mmg9Slice bgroundPopup;
private MmgFont exit;
private MmgBmp exitBground;
private MmgBmp number1;
private MmgBmp number2;
private MmgBmp number3;
public Mmg9Slice numberBground = null;
private NumberState numberState = NumberState.NONE;
private State state = State.NONE;
private State statePrev = State.NONE;
```

This third block of class fields power features like the popup background, the exit message background, and the countdown numbers among others. The first entry in the block is the bgroundPopupSrc field. It's an MmgBmp instance that loads a popup window image that's used as the source for an Mmg9Slice object, specifically the bgroundPopup field. The exit and exitBground fields are used to display the exit game message at the top of the game board.

The next three entries, number1–number3, are MmgBmp object instances that power the countdown numbers shown at the start of the game and in between each level. The numberBground field is an Mmg9Slice object used to draw a background window around the countdown numbers. This UI feature is used during the in-between levels' countdown screen state.

The last three fields listed are used to track screen states. The numberState field is used to control which number is displayed during the countdown, while the state and prevState fields are used to track the screen's current and previous states, respectively. The fourth block of fields control various game HUD/UI features.

Listing 24-7. ScreenGame Class Fields 4

```
public MdtUiHealthBar player1HealthBar = null;
public MmgBmp player1ModBmp;
public MmgBmp player1WeaponBmp;
public MdtUiHealthBar player2HealthBar = null;
public MmgBmp player2ModBmp;
public MmgBmp player2WeaponBmp;
private int popupTotalWidth = MmgHelper.ScaleValue(300);

private int popupTotalHeight = MmgHelper.ScaleValue(120);
```

The entries in this block of class fields are used to drive the HUD/UI on the right-hand side of the game screen and some UI elements used at the bottom of the game's board. The `player1HealthBar` and `player2HealthBar` are the health meter displays at the bottom of the game's board. The `player1ModBmp` and `player2ModBmp` fields are used to hold an image representation of the player character's current modifier.

Similarly, the `player1WeaponBmp` and `player2WeaponBmp` fields are used to display which weapon the associated player character is using. The last two entries, `popupTotalWidth` and `popupTotalHeight`, are used to control the display dimensions of the exit game popup window.

Listing 24-8. ScreenGame Class Fields 5

```
private MmgFont txtCancel;
private MmgFont txtOk;
private MmgFont txtDirecP1;
private MmgFont txtDirecP2;
private MmgFont txtGameOver1;
private MmgFont txtGameOver2;
private MmgFont txtGameOver3;
private MmgFont txtGoal;
```

The fifth set of class fields listed in the preceding are used to power some of the UI elements of the game screen's different states. The `txtCancel` and `txtOk` fields are part of the game exit popup. The `txtDirectP1` and `txtDirectP2` fields are `MmgFont` instances that are used to show the player what keys control their character. Similarly, the next three fields, `txtGameOver1`–`txtGameOver3`, are used to display information about the end

of the game, while the txtGoal field is an MmgFont instance that explains what the goal of the DungeonTrap game is. The sixth block of fields drive more UI text and associated data.

Listing 24-9. ScreenGame Class Fields 6

```
public MmgFont txtLevel = null;
public MmgFont txtLevelTime = null;
public MmgFont txtPlayer1 = null;
public MmgFont txtPlayer1Mod = null;
public MmgFont txtPlayer1ModTime = null;
public MmgFont txtPlayer1Score = null;
public MmgFont txtPlayer1Section = null;
public MmgFont txtPlayer1Weapon = null;
public MmgFont txtPlayer2 = null;
public MmgFont txtPlayer2Mod = null;
public MmgFont txtPlayer2ModTime = null;
public MmgFont txtPlayer2Score = null;
public MmgFont txtPlayer2Section = null;
public MmgFont txtPlayer2Weapon = null;
```

The next set of class fields for us to cover are used to power HUD/UI elements. The first two entries display the current level, enemy wave, and the remaining level time, respectively. The txtPlayer1 entry is used to display the name of the player on the bottom of the game board next to their health meter. The txtPlayer1Mod and txtPlayer1ModTime fields are used to display a text representation of the current player's active modifier and the remaining time the modifier is active.

The txtPlayer1Score field tracks the player1 character's current score. The txtPlayer1Section and txtPlayer1Weapon fields are used to display text in the game HUD/UI on the right-hand side of the screen. The same set of fields exist for player2 also listed in the preceding.

Listing 24-10. ScreenGame Class Fields 7

```
private int playersAliveCount = 0;
public MmgRect randoLeft = null;
public MmgRect randoRight = null;
private int scorePlayerOne = 0;
```

```
private int scorePlayerTwo = 0;
private boolean scoreTimeUp = false;
private long timeNumberDisplayMs = 1000;
private long timeTmpMs = 0L;
private MdtEnemyWave[] waves;
private int wavesCurrentIdx;
private MdtEnemyWave wavesCurrent;
private boolean randomWaves = false;
```

The seventh set of class fields for us to look at are a bit of a mixed bunch. The first entry indicates how many players are still alive in the game, `playersAliveCount`. The next two fields, `randoLeft` and `randoRight`, are used to determine where to place random items and objects at the start of a new level, enemy wave.

The next two fields, `scorePlayerOne` and `scorePlayerTwo`, are used to track the actual integer values of the players' scores. The `scoreTimeUp` field is a Boolean flag that tracks if the current level has run out of overtime. The `timeNumberDisplayMs` field along with the `timeTmpMs` field is used to indicate when a number should change during the countdown screen states.

Next up, the `waves` class field is an array of `MdtEnemyWave` object instances. These objects are used to define the properties of an enemy wave and are generated as part of the `ScreenGame` class' `LoadResources` method. The next two entries `wavesCurrent` and `wavesCurrentIdx` are used to track the current enemy wave object and the array index of the wave. The last field, `randomWaves`, is a Boolean flag that's used to indicate if the next level in the game should be randomly chosen. This field is used in the game's debugging to test different levels randomly.

Listing 24-11. ScreenGame Class Fields 8

```
private Color c;
private int padding = MmgHelper.ScaleValue(4);
private boolean lret;
private Random rand;
private MmgSprite sprite;
private MmgBmp spriteMatrixSrc;
private MmgSpriteMatrix spriteMatrix;
private GameType gameType = GameType.GAME_ONE_PLAYER;
```

The last block of class fields for us to review are a random assortment of internal, temporary, and supporting class fields. The `c` field is used internally by certain class methods to hold a color value during drawing. The `padding` and `lret` fields are used internally by certain class methods. The `rand` field is used to generate random values to support certain random decisions like item and object creation and positioning.

The next three fields, `sprite`, `spriteMatrixSrc`, and `spriteMatrix`, are used as temporary variables during the game screen's resource loading process. They are used to help prep resources for certain game objects. The last entry, `gameType`, is used to indicate which type of game should be run. Currently, DungeonTrap has one- or two-player local play supported. Next, we're going to outline the class methods that we'll need to cover to finish the game.

ScreenGame: Pertinent Method Outline

Now that we have the class field review completed, we can move on to tackle the class' methods. You won't have to worry about entering the class fields in by hand as they were already included in the prep classes for this section. We're going to be adding functionality to the class one method at a time. You can choose to type in the method code as we review it or on your own by looking at the completed chapter code as you type. To make it a little easier on you, at the end of the section, I'll provide instructions on how to copy and paste this section's work into your project.

Listing 24-12. ScreenGame Phase 2 Pertinent Method Outline 1

```
//HUD/UI Methods
private String FormatLevel(int idx) { ... }
private String FormatMod(long ms) { ... }
private String FormatTime(long ms) { ... }
private String FormatTime(long ms, int l) { ... }
private void SetScoreLeftText(int score) { ... }
private void SetScoreRightText(int score) { ... }
private void SetState(State in) { ... }
public void UpdateClearPlayerMod(MdtPlayerType p) { ... }

private void UpdatePlayerMod(MdtPlayerType p, MdtPlayerModType m) { ... }

public void UpdatePlayerWeapon(MdtPlayerType p, MdtWeaponType w) { ... }
```

```
private void UpdateUnlockDoor(MdtDoorType d) { ... }
private void UpdateLockAllDoors() { ... }
private void UpdateHideAllDoors() { ... }
public void UpdateAddPoints(MmgVector2 pos, MdtPointsType pts,
MdtPlayerType p) { ... }

//User Input Methods
public boolean ProcessAClick(int src) { ... }
public boolean ProcessBClick(int src) { ... }
public void ProcessDebugClick() { ... }
public boolean ProcessDpadPress(int dir) { ... }
public boolean ProcessDpadRelease(int dir) { ... }
public boolean ProcessKeyPress(char c, int code) { ... }

public boolean ProcessKeyRelease(char c, int code) { ... }
```

Notice that the methods are broken up into groups that indicate their general purpose with regard to the game. The class definitions for the `ScreenGame` class in Java and C# are as follows.

Listing 24-13. ScreenGame Class Definitions 1

```
//Java Version
public class ScreenGame extends Screen {

//C# Version
public class ScreenGame : Screen {
```

We still have a lot of material to cover, so prepare yourself. Notice that there aren't any class fields dedicated to potion items, weapons, or tables. Why do you think that is? It's because most, if not all, of the resources are handled inside the class itself as part of the class' construction. Thus, no fields are present in the `ScreenGame` class to support them. In the next section, we'll have a look at the class' HUD/UI methods.

ScreenGame: HUD/UI Method Details

Let's start off our review of the HUD/UI methods by looking at some number formatting methods.

Listing 24-14. ScreenGame HUD/UI Method Details 1

```
01 private String FormatLevel(int idx) {
02     String ret = (idx + 1) + "";
03     while(ret.length() < 2) {
04         ret = "0" + ret;
05     }
06     return ret;
07 }
```

```
01 private String FormatMod(long ms) {
02     String ret = ms + "";
03     while(ret.length() < 4) {
04         ret = "0" + ret;
05     }
06     return ret;
07 }
```

```
01 private String FormatTime(long ms) {
02     return FormatTime(ms, 3);
03 }
```

```
01 private String FormatTime(long ms, int l) {
02     int mul = 1;
03     if(ms < 0) {
04         mul = -1;
05     }
06     ms *= mul;
07     String ret = (ms / 1000) + "";
08     while(ret.length() < l) {
09         ret = "0" + ret;
10     }
11
12     if(mul == -1) {
13         ret = "-" + ret;
14     }
15     return ret;
16 }
```

```
01 private void SetScoreLeftText(int score) {
02     String tmp = score + "";
03     while(tmp.length() < 6) {
04         tmp = "0" + tmp;
05     }
06     txtPlayer1Score.SetText(tmp);
07 }
```

```
01 private void SetScoreRightText(int score) {
02     String tmp = score + "";
03     while(tmp.length() < 6) {
04         tmp = "0" + tmp;
05     }
06     txtPlayer2Score.SetText(tmp);
07 }
```

The first entry, `FormatLevel`, is used to format a level number so it is at least two digits long with left-padded zeros. This approach is applied to most of the HUD/UI number elements. The next method listed, `FormatMod`, is very similar to the first method we've reviewed except that it forces the number to use at least four digits, also left-padded with zeros.

The first `FormatTime` method listed is a simple pass-through to an overloaded version of the method that is called with default values for the missing arguments. The subsequent `FormatTime` method is the full version, and it's used to format a number to be left-padded with zeros until the number string length matches the provided argument. If the original numeric value was negative, then the resulting string is prefixed with a negative sign.

The `SetScoreLeftText` and `SetScoreRightText` methods format the incoming numeric value so that it's left-padded with zeros before updating the HUD/UI with the player's score. Notice that these methods don't set the player character's actual score. They set the UI element's display value. There is a separation between display values and game values and in the way the class handles game information, so keep that in mind as we review more code.

Listing 24-15. ScreenGame HUD/UI Method Details 2

```
01 public void UpdateClearPlayerMod(MdtPlayerType p) {
02     if(p == MdtPlayerType.PLAYER_1) {
03         RemoveObj(player1ModBmp);
04     } else {
05         RemoveObj(player2ModBmp);
06     }
07 }
```

```
01 private void UpdatePlayerMod(MdtPlayerType p, MdtPlayerModType m) {
02     if(p == MdtPlayerType.PLAYER_1) {
03         RemoveObj(player1ModBmp);
04
05         if(m == MdtPlayerModType.INVINCIBLE) {
06             MdtItemPotionYellow p3 = new MdtItemPotionYellow();
07             player1ModBmp = p3.GetSubj();
08         } else if(m == MdtPlayerModType.FULL_HEALTH) {
09             MdtItemPotionGreen p2 = new MdtItemPotionGreen();
10             player1ModBmp = p2.GetSubj();
11         } else if(m == MdtPlayerModType.DOUBLE_POINTS) {
12             MdtItemPotionRed p1 = new MdtItemPotionRed();
13             player1ModBmp = p1.GetSubj();
14         }
15
16         if(player1ModBmp != null) {
17             player1ModBmp.SetPosition(txtPlayer1Mod.GetPosition().
               Clone());
18             player1ModBmp.SetPosition(player1ModBmp.GetPosition().
               GetX() + txtPlayer1Mod.GetWidth() + MmgHelper.ScaleValue(5),
               player1ModBmp.GetPosition().GetY() - MmgHelper.
               ScaleValue(24));
19             player1ModBmp.SetIsVisible(false);
20             AddObj(player1ModBmp);
21         }
22     } else if(p == MdtPlayerType.PLAYER_2) {
23         RemoveObj(player2ModBmp);
```

```
24
25          if(m == MdtPlayerModType.INVINCIBLE) {
26              MdtItemPotionYellow p3 = new MdtItemPotionYellow();
27              player2ModBmp = p3.GetSubj();
28          } else if(m == MdtPlayerModType.FULL_HEALTH) {
29              MdtItemPotionGreen p2 = new MdtItemPotionGreen();
30              player2ModBmp = p2.GetSubj();
31          } else if(m == MdtPlayerModType.DOUBLE_POINTS) {
32              MdtItemPotionRed p1 = new MdtItemPotionRed();
33              player2ModBmp = p1.GetSubj();
34          }
35
36          if(player2ModBmp != null) {
37              player2ModBmp.SetPosition(txtPlayer2Mod.GetPosition().
                Clone());
38              player2ModBmp.SetPosition(player2ModBmp.GetPosition().
                GetX() + txtPlayer2Mod.GetWidth() + MmgHelper.ScaleValue(5),
                player2ModBmp.GetPosition().GetY() - MmgHelper.
                ScaleValue(24));
39              player2ModBmp.SetIsVisible(false);
40              AddObj(player2ModBmp);
41          }
42      }
43 }
```

```
01 public void UpdatePlayerWeapon(MdtPlayerType p, MdtWeaponType w) {
02      if(p == MdtPlayerType.PLAYER_1) {
03          RemoveObj(player1WeaponBmp);
04          if(w == MdtWeaponType.SPEAR) {
05              player1WeaponBmp = player1.weaponCurrent.subjRight.
                CloneTyped();
06              player1WeaponBmp.SetPosition(txtPlayer1Weapon.GetPosition().
                Clone());
07              player1WeaponBmp.SetPosition(player1WeaponBmp.GetPosition().
                GetX() + txtPlayer1Weapon.GetWidth() + MmgHelper.
                ScaleValue(5), player1WeaponBmp.GetPosition().GetY() -
                MmgHelper.ScaleValue(12));
16
```

```
08          } else if(w == MdtWeaponType.SWORD) {
09              player1WeaponBmp = player1.weaponCurrent.subjRight.
                CloneTyped();
10              player1WeaponBmp.SetPosition(txtPlayer1Weapon.GetPosition().
                Clone());
11              player1WeaponBmp.SetPosition(player1WeaponBmp.GetPosition().
                GetX() + txtPlayer1Weapon.GetWidth() + MmgHelper.
                ScaleValue(5), player1WeaponBmp.GetPosition().GetY() -
                MmgHelper.ScaleValue(12));
12          } else if(w == MdtWeaponType.AXE) {
13              player1WeaponBmp = player1.weaponCurrent.subjRight.
                CloneTyped();
14              player1WeaponBmp.SetPosition(txtPlayer1Weapon.GetPosition().
                Clone());
15              player1WeaponBmp.SetPosition(player1WeaponBmp.GetPosition().
                GetX() + txtPlayer1Weapon.GetWidth() + MmgHelper.
                ScaleValue(5), player1WeaponBmp.GetPosition().GetY() -
                MmgHelper.ScaleValue(12));
16          }
17
18          if(player1WeaponBmp != null) {
19              AddObj(player1WeaponBmp);
20          }
21      } else if(p == MdtPlayerType.PLAYER_2) {
22          RemoveObj(player2WeaponBmp);
23          if(w == MdtWeaponType.SPEAR) {
24              player2WeaponBmp = player2.weaponCurrent.subjRight.
                CloneTyped();
25              player2WeaponBmp.SetPosition(txtPlayer2Weapon.GetPosition().
                Clone());
26              player2WeaponBmp.SetPosition(player2WeaponBmp.GetPosition().
                GetX() + txtPlayer2Weapon.GetWidth() + MmgHelper.
                ScaleValue(5), player2WeaponBmp.GetPosition().GetY() -
                MmgHelper.ScaleValue(12));
27          } else if(w == MdtWeaponType.SWORD) {
28             player2WeaponBmp = player2.weaponCurrent.subjRight.CloneTyped();
```

```
            player2WeaponBmp.SetPosition(txtPlayer2Weapon.GetPosition().
            Clone());
            player2WeaponBmp.SetPosition(player2WeaponBmp.GetPosition().
            GetX() + txtPlayer2Weapon.GetWidth() + MmgHelper.
            ScaleValue(5), player2WeaponBmp.GetPosition().GetY() -
            MmgHelper.ScaleValue(12));
        } else if(w == MdtWeaponType.AXE) {
            player2WeaponBmp = player2.weaponCurrent.subjRight.
            CloneTyped();
            player2WeaponBmp.SetPosition(txtPlayer2Weapon.GetPosition().
            Clone());
            player2WeaponBmp.SetPosition(player2WeaponBmp.GetPosition().
            GetX() + txtPlayer2Weapon.GetWidth() + MmgHelper.
            ScaleValue(5), player2WeaponBmp.GetPosition().GetY() -
            MmgHelper.ScaleValue(12));
        }

        if(player2WeaponBmp != null) {
            AddObj(player2WeaponBmp);
        }
    }
}
```

```
private void UpdateUnlockDoor(MdtDoorType d) {
    if(d == MdtDoorType.TOP_LEFT) {
        doorTopLeftLocked.SetIsVisible(false);
        doorTopLeftOpened.SetIsVisible(true);
    } else if(d == MdtDoorType.TOP_RIGHT) {
        doorTopRightLocked.SetIsVisible(false);
        doorTopRightOpened.SetIsVisible(true);
    } else if(d == MdtDoorType.LEFT) {
        doorLeftLockIcon.SetIsVisible(false);
    } else if(d == MdtDoorType.RIGHT) {
        doorRightLockIcon.SetIsVisible(false);
    } else if(d == MdtDoorType.BOTTOM_LEFT) {
        doorBotLeftLockIcon.SetIsVisible(false);
```

```
18
```

```
    } else if(d == MdtDoorType.BOTTOM_RIGHT) {
        doorBotRightLockIcon.SetIsVisible(false);
    }
}
```

```
private void UpdateLockAllDoors() {
    doorTopLeftLocked.SetIsVisible(true);
    doorTopLeftOpened.SetIsVisible(false);

    doorTopRightLocked.SetIsVisible(true);
    doorTopRightOpened.SetIsVisible(false);

    doorLeftLockIcon.SetIsVisible(true);
    doorRightLockIcon.SetIsVisible(true);
    doorBotLeftLockIcon.SetIsVisible(true);
    doorBotRightLockIcon.SetIsVisible(true);
}
```

```
private void UpdateHideAllDoors() {
    doorTopLeftLocked.SetIsVisible(false);
    doorTopLeftOpened.SetIsVisible(false);

    doorTopRightLocked.SetIsVisible(false);
    doorTopRightOpened.SetIsVisible(false);

    doorLeftLockIcon.SetIsVisible(false);
    doorRightLockIcon.SetIsVisible(false);
    doorBotLeftLockIcon.SetIsVisible(false);
    doorBotRightLockIcon.SetIsVisible(false);
}
```

```
public void UpdateAddPoints(MmgVector2 pos, MdtPointsType pts,
MdtPlayerType p) {
    MmgBmp ptsBmp  = null;
    MdtUiPoints ptsUi = null;
    MmgVector2 posC = null;

```

```
    if(p == MdtPlayerType.PLAYER_1) {
        if(pts == MdtPointsType.PTS_100) {
            ptsBmp = MmgHelper.GetBasicCachedBmp("pts_red_100.png");
            scorePlayerOne += 100;
            if(player1.hasDoublePoints) {
                scorePlayerOne += 100;
            }
            SetScoreLeftText(scorePlayerOne);
        } else if(pts == MdtPointsType.PTS_250) {
            ptsBmp = MmgHelper.GetBasicCachedBmp("pts_red_250.png");
            scorePlayerOne += 250;
            if(player1.hasDoublePoints) {
                scorePlayerOne += 250;
            }
            SetScoreLeftText(scorePlayerOne);
        } else if(pts == MdtPointsType.PTS_500) {
            ptsBmp = MmgHelper.GetBasicCachedBmp("pts_red_500.png");
            scorePlayerOne += 500;
            if(player1.hasDoublePoints) {
                scorePlayerOne += 500;
            }
            SetScoreLeftText(scorePlayerOne);
        } else if(pts == MdtPointsType.PTS_1000) {
            ptsBmp = MmgHelper.GetBasicCachedBmp("pts_red_1000.png");
            scorePlayerOne += 1000;
            if(player1.hasDoublePoints) {
                scorePlayerOne += 1000;
            }
            SetScoreLeftText(scorePlayerOne);
        }

        ptsUi = new MdtUiPoints(ptsBmp, p, this, pos);
        AddObj(ptsUi);

        if(player1.hasDoublePoints) {
```

```
41              posC = pos.Clone();
42              posC.SetY(posC.GetY() + ptsBmp.GetHeight());
43              ptsUi = new MdtUiPoints(ptsBmp.CloneTyped(), p, this, posC);
44              AddObj(ptsUi);
45          }
46      } else {
47          if(pts == MdtPointsType.PTS_100) {
48              ptsBmp = MmgHelper.GetBasicCachedBmp("pts_blue_100.png");
49              scorePlayerTwo += 100;
50              if(player2.hasDoublePoints) {
51                  scorePlayerTwo += 100;
52              }
53              SetScoreRightText(scorePlayerTwo);
54          } else if(pts == MdtPointsType.PTS_250) {
55              ptsBmp = MmgHelper.GetBasicCachedBmp("pts_blue_250.png");
56              scorePlayerTwo += 250;
57              if(player2.hasDoublePoints) {
58                  scorePlayerTwo += 250;
59              }
60              SetScoreRightText(scorePlayerTwo);
61          } else if(pts == MdtPointsType.PTS_500) {
62              ptsBmp = MmgHelper.GetBasicCachedBmp("pts_blue_500.png");
63              scorePlayerTwo += 500;
64              if(player2.hasDoublePoints) {
65                  scorePlayerTwo += 500;
66              }
67              SetScoreRightText(scorePlayerTwo);
68          } else if(pts == MdtPointsType.PTS_1000) {
69              ptsBmp = MmgHelper.GetBasicCachedBmp("pts_blue_1000.png");
70              scorePlayerTwo += 1000;
71              if(player2.hasDoublePoints) {
72                  scorePlayerTwo += 1000;
73              }
74              SetScoreRightText(scorePlayerTwo);
75          }
76
```

```
77          ptsUi = new MdtUiPoints(ptsBmp, p, this, pos);
78          AddObj(ptsUi);
79
80          if(player2.hasDoublePoints) {
81              posC = pos.Clone();
82              posC.SetY(posC.GetY() + ptsBmp.GetHeight());
83              ptsUi = new MdtUiPoints(ptsBmp.CloneTyped(), p, this, posC);
84              AddObj(ptsUi);
85          }
86      }
87  }
```

The block of methods listed in the preceding are used to control different visual elements of the game board and game HUD/UI. The `UpdateClearPlayerMod` method simply removes the HUD player modifier image from the screen. The next method listed, `UpdatePlayerMod`, clears the player's modifier image. A new image is then created based on the specified player's active modifier, lines 5–14. The same functionality exists for player2 in the subsequent lines of code.

At the end of either player's section of code, the new player modifier image is positioned, marked as invisible, and added to the game screen. Note that this feature of the game is active but not visible. You'll notice that the game uses words to describe the player's modifier instead of an image. This is an example of an aspect of the game you can customize on your own. The next method we'll look at is the `UpdatePlayerWeapon` method.

This method is used to update the HUD/UI with a representation of the player's weapon. For player1, on lines 4–16 and based on the provided weapon type, the `player1WeapnBmp` field is updated with a copy of the weapon image. The same steps are performed for player2 on lines 23–35.

The `UpdateUnlockDoor` method is used to unlock a specific door on the game's board. The method handles turning on or off the board images that control displaying the lock or unlock image for a specific door. A similar method is the `UpdateLockAllDoors` method. This method is designed to set all the doors on the game board to a locked state. This is a good starting point for configuring the level.

Next up, we have the `UpdateHideAllDoors` method. This method makes sure all door open and lock images are hidden from view. Lastly, we have the `UpdateAddPoints` method. This method is used to add a new `MdtUiPoints` object to the screen. This object

extends the MdtBase class, but its only use is for displaying awarded points, so we won't review it in any detail.

At the specified position, the requested number of points is added to the player's total, updated in the game's HUD/UI, and a floating points image is placed on the screen. The points image has a specific color depending on which player scored the points and floats up slightly before disappearing. Not too bad. As you can see at this point in the class code, we have a lot of control over the game's HUD/UI elements but not much else.

The next method we'll review is the main UI control method for this screen. It determines which UI elements are visible for the given screen state. Because this method is so long, we'll only look at a few choice blocks of code here. Be sure to carefully add this to your copy of the ScreenGame class. Refer to the chapter's completed project code if you run into any issues, and as always, I'll provide instructions on how to copy and paste the work done in this section at the end of the section. Let's take a look at some code!

Listing 24-16. ScreenGame HUD/UI Method Details 3

```
01 case SHOW_GAME_OVER:
02     UpdateHideAllDoors();
03     numberBground.SetIsVisible(false);
04     player1.SetIsVisible(false);
05     player2.SetIsVisible(false);
06
07     torch1.SetIsVisible(false);
08     torch2.SetIsVisible(false);
09     torch3.SetIsVisible(false);
10     torch4.SetIsVisible(false);
11
12     txtLevel.SetIsVisible(false);
13     txtLevelTime.SetIsVisible(false);
14     player1HealthBar.SetIsVisible(false);
15     player2HealthBar.SetIsVisible(false);
16
17     exit.SetIsVisible(false);
18     exitBground.SetIsVisible(false);
19     gameLogo.SetIsVisible(false);
```

```
20
21      bground.SetIsVisible(false);
22      number1.SetIsVisible(false);
23      number2.SetIsVisible(false);
24      number3.SetIsVisible(false);
25      txtGoal.SetIsVisible(false);
26      txtDirecP1.SetIsVisible(false);
27      txtDirecP2.SetIsVisible(false);
28
29      txtPlayer1.SetIsVisible(false);
30      txtPlayer1Mod.SetIsVisible(false);
31      txtPlayer1ModTime.SetIsVisible(false);
32      txtPlayer1Score.SetIsVisible(false);
33      txtPlayer1Section.SetIsVisible(false);
34      txtPlayer1Weapon.SetIsVisible(false);
35
36      if(player1WeaponBmp != null) {
37          player1WeaponBmp.SetIsVisible(false);
38      }
39
40      if(player1ModBmp != null) {
41          player1ModBmp.SetIsVisible(false);
42      }
43
44      txtPlayer2.SetIsVisible(false);
45      txtPlayer2Mod.SetIsVisible(false);
46      txtPlayer2ModTime.SetIsVisible(false);
47      txtPlayer2Score.SetIsVisible(false);
48      txtPlayer2Section.SetIsVisible(false);
49      txtPlayer2Weapon.SetIsVisible(false);
50
51      if(player2WeaponBmp != null) {
52          player2WeaponBmp.SetIsVisible(false);
53      }
54
```

```
55      if(player2ModBmp != null) {
56          player2ModBmp.SetIsVisible(false);
57      }
58
59      bgroundPopup.SetIsVisible(false);
60      txtOk.SetIsVisible(false);
61      txtCancel.SetIsVisible(false);
62
63      UpdateClearObjects();
64
65      txtGameOver1.SetIsVisible(false);
66      txtGameOver2.SetIsVisible(false);
67      txtGameOver3.SetIsVisible(false);
68
69      if(scoreTimeUp) {
70          txtGameOver3.SetIsVisible(true);
71      } else {
72          if(gameType == GameType.GAME_TWO_PLAYER) {
73              if(scorePlayerOne >= scorePlayerTwo) {
74                  txtGameOver1.SetIsVisible(true);
75              } else {
76                  txtGameOver2.SetIsVisible(true);
77              }
78          } else {
79              txtGameOver1.SetIsVisible(true);
80          }
81      }
82
83      numberState = NumberState.NONE;
84      pause = false;
85      isDirty = true;
86      break;
```

The snippet of code listed in the preceding is from the `SetState` method and is set to handle the `SHOW_GAME_OVER` screen state. Notice how each aspect of the screen – UI, HUD, atmosphere, board, and so on – is adjusted so that the screen only displays the UI

elements needed for the given game state. This is how we control what the game screen is showing.

We have support for controlling the game screen on a grand scale, using the SetState method to prep the screen for a specific state. And we have granular control over the HUD/UI features using class methods. That's just about one-third of the work we have to do to complete the game. If we add in the user input methods and the game logic methods, then we'll have a complete game. Next, let's take a look at the user input methods.

ScreenGame: User Input Methods

Let's take a moment to develop the landscape of where we are in the game development process. We have all our game screens in place, splash, loading, menu, and in-game. We have a set of game objects to work with, specifically items, objects, enemies, players, and weapons to name a few. And, last but not least, we have a game board, environment, HUD, and UI. By adding user input to the game, we can control the screen state and the players bringing us one big step closer to completing the game. Let's check out some code.

Listing 24-17. ScreenGame User Input Method Details 1

```
01 public boolean ProcessAClick(int src) {
02     if(pause || !isVisible) {
03         return false;
04     }
05
06     if(state == State.SHOW_GAME) {
07         if(gameType == GameType.GAME_ONE_PLAYER || gameType == GameType.
           GAME_TWO_PLAYER) {
08             if(src == ScreenGame.SRC_PLAYER_1) {
09                 if(!player1.isAttacking) {
10                     if(player1.weaponCurrent.attackType ==
                       MdtWeaponAttackType.THROWING && player1.
                       weaponCurrent.weaponType == MdtWeaponType.AXE) {
11                         player1.weaponCurrent = player1.weaponCurrent.
                           Clone();
```

```
12                      player1.weaponCurrent.SetPosition(GetX() +
                        GetWidth()/2, GetY() + GetHeight()/2);
13                      player1.weaponCurrent.current = null;
14                      player1.weaponCurrent.throwingPath =
                        MdtWeaponPathType.NONE;
15                      player1.weaponCurrent.screen = this;
16                      AddObj(player1.weaponCurrent);
17                  }
18
19                  player1.weaponCurrent.animTimeMsCurrent = 0;
20                  player1.weaponCurrent.animPrctComplete = 0.0d;
21                  player1.isAttacking = true;
22                  player1.weaponCurrent.active = true;
23              }
24          } else if(src == ScreenGame.SRC_PLAYER_2) {
25              if(!player2.isAttacking) {
26                  if(player2.weaponCurrent.attackType ==
                    MdtWeaponAttackType.THROWING && player2.
                    weaponCurrent.weaponType == MdtWeaponType.AXE) {
27                      player2.weaponCurrent = player2.weaponCurrent.
                        Clone();
28                      player2.weaponCurrent.SetPosition(GetX() +
                        GetWidth()/2, GetY() + GetHeight()/2);
29                      player2.weaponCurrent.current = null;
30                      player2.weaponCurrent.throwingPath =
                        MdtWeaponPathType.NONE;
31                      player2.weaponCurrent.screen = this;
32                      AddObj(player2.weaponCurrent);
33                  }
34
35                  player2.weaponCurrent.animTimeMsCurrent = 0;
36                  player2.weaponCurrent.animPrctComplete = 0.0d;
37                  player2.isAttacking = true;
38                  player2.weaponCurrent.active = true;
39              }
```

```
            }
        }
        return true;
    } else if(state == State.SHOW_GAME_EXIT) {
        owner.SwitchGameState(GameStates.MAIN_MENU);
        return true;
    } else if(state == State.SHOW_GAME_OVER) {
        owner.SwitchGameState(GameStates.MAIN_MENU);
        return true;
    }
    return false;
}
```

```
public boolean ProcessBClick(int src) {
    if(pause || !isVisible) {
        return false;
    }

    if(state == State.SHOW_GAME_OVER) {
        owner.SwitchGameState(GameStates.MAIN_MENU);
        return true;
    } else {
        if(state != State.SHOW_GAME_EXIT) {
            SetState(State.SHOW_GAME_EXIT);
            return true;
        } else {
            SetState(statePrev);
            return true;
        }
    }
}
```

```
public void ProcessDebugClick() {
    randomWaves = !randomWaves;
    MmgHelper.wr("RandomWaves: " + randomWaves);
}
```

The first method listed in the preceding set is the input handler for the A click event. Recall now that the method can be mapped to different keys, and the `src` argument tells us from which player the input originated. Depending on the screen's state, `SHOW_GAME`, line 6, and the type of game, line 7, the input is processed. You can see that the input source is checked on lines 9–24 separating the input processing for player1 and player2.

The code on lines 9–24 controls player1's weapon attack. Similarly, the code on lines 25–39 controls player2's attack. The only other screen states where this input handler is active are the `SHOW_GAME_EXIT` and `SHOW_GAME_OVER` states. In each case, the input from either player redirects the game back to the main menu.

Next let's take a look at the `ProcessBClick` method. The B input event is enabled for both players on the game over and other screen states. Follow the logic of the method and see that it makes sense to you. The last method listed in the preceding is the `ProcessDebugClick` method. Currently, all this method does is toggle the random level flag. With this flag set to true, the game will jump to a random level at the end of the current level. This is helpful during level debugging.

Listing 24-18. ScreenGame User Input Method Details 2

```
01 public boolean ProcessKeyPress(char c, int code) {
02     if(pause || !isVisible) {
03         return false;
04     }
05
06     if(state == State.SHOW_GAME) {
07         if(gameType == GameType.GAME_TWO_PLAYER) {
08             boolean found = false;
09
10             if(c == 'x' || c == 'X') {
11                 //down
12                 if(player2.GetDir() != MmgDir.DIR_FRONT) {
13                     player2.SetDir(MmgDir.DIR_FRONT);
14                 }
15                 found = true;
16             } else if(c == 's' || c == 'S') {
17                 //up
18                 if(player2.GetDir() != MmgDir.DIR_BACK) {
```

```
                    player2.SetDir(MmgDir.DIR_BACK);
                }
                found = true;
            } else if(c == 'z' || c == 'Z') {
                //left
                if(player2.GetDir() != MmgDir.DIR_LEFT) {
                    player2.SetDir(MmgDir.DIR_LEFT);
                }
                found = true;
            } else if(c == 'c' || c == 'C') {
                //right
                if(player2.GetDir() != MmgDir.DIR_RIGHT) {
                    player2.SetDir(MmgDir.DIR_RIGHT);
                }
                found = true;
            }

            if(found) {
                player2.isMoving = true;
                player2.subj.SetMsPerFrame(frameMsPerFrameMoving);
            }
        }
    }

    return false;
}
```

```
public boolean ProcessKeyRelease(char c, int code) {
    if(pause || !isVisible) {
        return false;
    }

    if(state == State.SHOW_GAME_EXIT) {
        if(gameType == GameType.GAME_TWO_PLAYER) {
            if(c == 'v' || c == 'V') {
                ProcessBClick(SRC_PLAYER_2);

30
```

```
            } else if(c == 'f' || c == 'F') {
                ProcessAClick(SRC_PLAYER_2);
            }
        }

        if(c == '/' || c == '?') {
            ProcessBClick(SRC_PLAYER_1);
        } else if(c == '.' || c == '>') {
            ProcessAClick(SRC_PLAYER_1);
        }
    } else if(state == State.SHOW_GAME) {
        if(gameType == GameType.GAME_TWO_PLAYER) {
            boolean found = true;
            if(c == 'x' || c == 'X') {
                //down
                found = true;
            } else if(c == 's' || c == 'S') {
                //up
                found = true;
            } else if(c == 'z' || c == 'Z') {
                //left
                found = true;
            } else if(c == 'c' || c == 'C') {
                //right
                found = true;
            } else if(c == 'f' || c == 'F') {
                ProcessAClick(SRC_PLAYER_2);
            } else if(c == 'v' || c == 'V') {
                ProcessBClick(SRC_PLAYER_2);
            }

            if(found) {
                if(playerSnapToFront == true) {
                    player2.SetDir(MmgDir.DIR_FRONT);
                }
```

```
45
46                  player2.isMoving = false;
47                  player2.isPushStart = false;
48                  player2.isPushing = false;
49                  player2.subj.SetMsPerFrame(frameMsPerFrameNotMoving);
50              }
51          }
52
53          if(c == '.' || c == '>') {
54              ProcessAClick(SRC_PLAYER_1);
55          } else if(c == '/' || c == '?') {
56              ProcessBClick(SRC_PLAYER_1);
57          }
58      }
59      return false;
60  }
```

```
01 public boolean ProcessDpadPress(int dir) {
02      if(pause || !isVisible) {
03          return false;
04      }
05
06      if(state == State.SHOW_GAME) {
07          if(gameType == GameType.GAME_ONE_PLAYER || gameType == GameType.
            GAME_TWO_PLAYER) {
08              boolean found = false;
09
10              if(dir == GameSettings.DOWN_KEYBOARD) {
11                  if(player1.GetDir() != MmgDir.DIR_FRONT) {
12                      player1.SetDir(MmgDir.DIR_FRONT);
13                  }
14                  found = true;
15
16              } else if(dir == GameSettings.UP_KEYBOARD) {
17                  if(player1.GetDir() != MmgDir.DIR_BACK) {
```

```
                    player1.SetDir(MmgDir.DIR_BACK);
                }
                found = true;

            } else if(dir == GameSettings.LEFT_KEYBOARD) {
                if(player1.GetDir() != MmgDir.DIR_LEFT) {
                    player1.SetDir(MmgDir.DIR_LEFT);
                }
                found = true;

            } else if(dir == GameSettings.RIGHT_KEYBOARD) {
                if(player1.GetDir() != MmgDir.DIR_RIGHT) {
                    player1.SetDir(MmgDir.DIR_RIGHT);
                }
                found = true;
            }

            if(found) {
                player1.isMoving = true;
                player1.subj.SetMsPerFrame(frameMsPerFrameMoving);
            }
            return true;
        }
    }
    return false;
}
```

```
public boolean ProcessDpadRelease(int dir) {
    if(pause || !isVisible) {
        return false;
    }

    if(state == State.SHOW_GAME) {
        if(gameType == GameType.GAME_ONE_PLAYER || gameType == GameType.
        GAME_TWO_PLAYER) {
            if(playerSnapToFront == true) {
```

```
09                  player1.SetDir(MmgDir.DIR_FRONT);
10              }
11
12              player1.isMoving = false;
13              player1.isPushStart = false;
14              player1.isPushing = false;
15              player1.subj.SetMsPerFrame(frameMsPerFrameNotMoving);
16              return true;
17          }
18      }
19      return false;
20  }
```

The next block of input methods for us to review are listed in the preceding. Let's look at the `ProcessKeyPress` method first. Note that the input is only active on the `SHOW_GAME` screen state, line 6. This method is primarily used to map keyboard input for player2 to direction pad input for player2's character. On lines 10–34, the input is converted to direction pad input, and the character's direction is adjusted accordingly.

The next method for us to look at is the `ProcessKeyRelease` method. This method handles the player2 direction pad release events and mapping more keys to the A and B input events for players 1 and 2. On lines 6–19, the input for the game exit screen state is handled for player2. Subsequently, the input for player1 is handled on lines 15–19.

This opens up more A and B inputs for player1 so they can share the keyboard during two-player games. The code on lines 20–51 runs the in-game controls for player2's release events. Player1's additional inputs are mapped on lines 53–57. Player1's direction pad input is processed in the `ProcessDpadPress` method. Notice that the input is only processed if the screen is in the `SHOW_GAME` state. The code on lines 10–38 is similar to the direction pad code we just reviewed for player2. Note that the character's direction is updated in the same way as player2's.

The last input method for us to review is the `ProcessDpadRelease` method. It performs the same direction pad release checks we saw in the `ProcessKeyRelease` method except for player1's character. Notice that on lines 12–15, the code deactivates the player since the character is no longer being moved. That wraps up the user input methods. Next up, we'll stop and take a look at the project as it stands at the end of Phase 2.

ScreenGame: Phase 2 Demo

In order to run the Phase 2 demonstration, you'll need to make sure you have all the methods properly added to your copy of the `ScreenGame` class. You should also have the following lines of code at the end of your `LoadResources` method.

Listing 24-19. ScreenGame Phase 2 Demo 1

```
01 SetState(State.SHOW_GAME);
02 ready = true;
03 pause = false;
```

The operative line of code here is line 1 where the screen's state is set to `SHOW_GAME`. Recall from our review of the user input methods that this should enable some user controls to function. A shortcut to typing up this code by hand can be found in the game engine's main project folder in the "cfg/asset_src/dungeon_trap_chapter24_phase2" folder.

Copy the `ScreenGame` class contained within the folder to your game project. Be sure to adjust the `package/namespace` values of any newly pasted files to match your project setup. Once you have everything configured correctly, run the project. If you're following along in C#, use the command line to launch your game after compiling.

this figure will be printed in b/w

Oh, no! That's the same boring demo from Phase 1 of this chapter! But wait. It just looks the same. Press some of the direction pad keys on your keyboard, and you'll notice that the player1 character is moving across the screen under the game objects and items in the demo. Press the "A" or "." key on your keyboard, and you'll see the character attack!

Take a moment to think about how simple and direct the input logic was. Most of the work is done by the classes we've already defined. In fact, the `MdtObjPush` and `MdtCharInter` classes know to keep the game object or player on the board automatically as part of their setup and connection to the `ScreenGame` class.

I wanted to take a moment to talk about the way the game objects are layered. The player1 character moves underneath the demo objects because they were added to the screen after the player. This presents a problem. How do we keep game objects in the correct order when we're adding new objects to the game screen all the time? That's a good question.

To handle this issue, we use `MmgContainer` objects to create placeholders for the game objects. The `gameItems`, `gameObjects`, and `gameEnemies` fields are placeholders for adding new objects to the game that hold the order between the different game objects on the board constant. After all, the containers aren't changing; just their contents are. Let's move on to the next phase in the `ScreenGame` class' development and the last set of methods we need to review to complete the game.

ScreenGame: Class Review, Phase 3

In Phase 3 of the `ScreenGame` class' review, we'll complete the remaining methods in the game as listed in the following.

Listing 24-20. ScreenGame Phase 3 Pertinent Method Outline 1

```
public MdtBase CanMove(MmgRect r, MdtBase i0) { ... }
public MdtBase CanMove(MmgRect r) { ... }
public void DrawScreen() { ... }
public MdtItem UpdateGenerateMdtItem(int x, int y, MdtItemType[] items) {
... }

public MdtBase UpdateGenerateMdtItem(int x, int y) { ... }

private void UpdateGenerateObjects(int cnt, MdtItemType[] items) { ... }

private void UpdateStartEnemyWave(int waveIdx) { ... }
private void UpdateResetPlayers() { ... }
private void UpdateCurrentEnemyWave(long msSinceLastFrame) { ... }
```

```
public void UpdateProcessWeaponCollision(MdtBase o1, MdtWeapon o2, MmgRect
weapon) { ... }

public void UpdateProcessWeaponCollision(MdtBase o1, MdtCharInter o2,
MmgRect weapon) { ... }

public void UpdateProcessCollision(MdtBase o1, MdtBase o2) { ... }
```

The next set of methods for us to review are the game logic methods. These methods provide the connections between game objects that drive much of the game's functionality. This is the last piece we need to complete the game.

ScreenGame: Game Logic Methods

The game logic methods handle the creation, collision, and destruction of game objects necessary to facilitate a game – in this case, the DungeonTrap game. The first set of methods we'll work on include the `CanMove` and `DrawScreen` methods. These new methods will add collision detection capability, `CanMove`, and dynamic drawing capability, `DrawScreen`, to the class.

These features stand alone, with regard to the remaining game logic, so we can implement them first safely. Note that the `DrawScreen` method checks for the game over condition and the `CanMove` method sets the basis for all collisions in the game.

Listing 24-21. ScreenGame Game Logic Methods 1

```
001 public MdtBase CanMove(MmgRect r, MdtBase i0) {
002     int len = GetObjects().GetCount();
003     MmgObj o = null;
004     MdtBase b = null;
005     MmgContainer c = null;
006
007     for(int i = 0; i < len; i++) {
008         o = GetObjects().GetChildAt(i);
009         if(o instanceof MmgContainer) {
010             c = (MmgContainer)o;
011             int len2 = c.GetCount();
012             for(int j = 0; j < len2; j++) {
013                 o = c.GetChildAt(j);
```

```
014                 if(o instanceof MdtBase && ((MdtBase)o).isVisible) {
015                     b = ((MdtBase)o);
016
017                     if(b != null && i0 != null && b.equals(i0) == false
                        && MmgHelper.RectCollision(r, b.GetRect())) {
018                         return b;
019                     } else if(b != null && i0 == null && MmgHelper.
                        RectCollision(r, b.GetRect())) {
020                         return b;
021                     }
022                 }
023             }
024         } else if(o instanceof MdtBase && ((MdtBase)o).isVisible) {
025             b = ((MdtBase)o);
026             if(b != null && i0 != null && b.equals(i0) == false &&
                MmgHelper.RectCollision(r, b.GetRect())) {
027                 return b;
028             } else if(b != null && i0 == null && MmgHelper.
                RectCollision(r, b.GetRect())) {
029                 return b;
030             }
031         }
032     }
033     return null;
034 }
```

```
001 public MdtBase CanMove(MmgRect r) {
002     return CanMove(r, null);
003 }
```

```
001 public void DrawScreen() {
002     pause = true;
003     switch(state) {
004         case NONE:
005             break;
006         case SHOW_GAME_EXIT:
```

```
            break;
        case SHOW_COUNT_DOWN_IN_GAME:
        case SHOW_COUNT_DOWN:
            switch(numberState) {
                case NONE:
                    timeNumberMs = System.currentTimeMillis();
                    numberState = NumberState.NUMBER_3;
                    number1.SetIsVisible(false);
                    number2.SetIsVisible(false);
                    number3.SetIsVisible(true);
                    if(state == State.SHOW_COUNT_DOWN) {
                        txtGoal.SetIsVisible(true);
                        txtDirecP1.SetIsVisible(true);
                        if(gameType == GameType.GAME_TWO_PLAYER) {
                            txtDirecP2.SetIsVisible(true);
                        } else {
                            txtDirecP2.SetIsVisible(false);
                        }
                    } else {
                        txtGoal.SetIsVisible(false);
                        txtDirecP1.SetIsVisible(false);
                        txtDirecP2.SetIsVisible(false);
                    }
                    break;
                case NUMBER_1:
                    timeTmpMs = System.currentTimeMillis();
                    if(timeTmpMs - timeNumberMs >= timeNumberDisplayMs) {
                        timeNumberMs = timeTmpMs;
                        numberState = NumberState.NONE;
                        number1.SetIsVisible(false);
                        number2.SetIsVisible(false);
                        number3.SetIsVisible(false);
                        txtDirecP1.SetIsVisible(false);
                        txtDirecP2.SetIsVisible(false);
```

```
041                         SetState(State.SHOW_GAME);
042                     }
043                     break;
044                 case NUMBER_2:
045                     timeTmpMs = System.currentTimeMillis();
046                    if(timeTmpMs - timeNumberMs >= timeNumberDisplayMs) {
047                         timeNumberMs = timeTmpMs;
048                         numberState = NumberState.NUMBER_1;
049                         number1.SetIsVisible(true);
050                         number2.SetIsVisible(false);
051                         number3.SetIsVisible(false);
052                         txtDirecP1.SetIsVisible(true);
053                         if(state == State.SHOW_COUNT_DOWN) {
054                             txtGoal.SetIsVisible(true);
055                             if(gameType == GameType.GAME_TWO_PLAYER) {
056                                 txtDirecP2.SetIsVisible(true);
057                             } else {
058                                 txtDirecP2.SetIsVisible(false);
059                             }
060                         } else {
061                             txtGoal.SetIsVisible(false);
062                             txtDirecP1.SetIsVisible(false);
063                             txtDirecP2.SetIsVisible(false);
064                         }
065                     }
066                     break;
067                 case NUMBER_3:
068                     timeTmpMs = System.currentTimeMillis();
069                    if(timeTmpMs - timeNumberMs >= timeNumberDisplayMs) {
070                         timeNumberMs = timeTmpMs;
071                         numberState = NumberState.NUMBER_2;
072                         number1.SetIsVisible(false);
073                         number2.SetIsVisible(true);
074                         number3.SetIsVisible(false);
```

```
075                     if(state == State.SHOW_COUNT_DOWN) {
076                         txtGoal.SetIsVisible(true);
077                         if(gameType == GameType.GAME_TWO_PLAYER) {
078                             txtDirecP2.SetIsVisible(true);
079                         } else {
080                             txtDirecP2.SetIsVisible(false);
081                         }
082                     } else {
083                         txtGoal.SetIsVisible(false);
084                         txtDirecP1.SetIsVisible(false);
085                         txtDirecP2.SetIsVisible(false);
086                     }
087                 }
088                 break;
089             }
090             break;
091         case SHOW_GAME:
092             if(wavesCurrent != null) {
093                 if(wavesCurrent.actEnemyCount >= wavesCurrent.
                    enemyCount && gameEnemies.GetCount() == 0) {
094                     wavesCurrentIdx++;
095                     if(wavesCurrentIdx < waves.length) {
096                         if(randomWaves) {
097                             wavesCurrentIdx = MmgHelper.
                                GetRandomIntRange(0, waves.length);
098                         }
099                         SetState(State.SHOW_COUNT_DOWN_IN_GAME);
100                     } else {
101                         SetState(State.SHOW_GAME_OVER);
102                     }
103                 } else if((wavesCurrent.timeTotalMs - wavesCurrent.
                    timeCurrentMs) <= 0) {
104                     int tmpt = (int)(wavesCurrent.timeTotalMs -
                        wavesCurrent.timeCurrentMs) / 1000;
105                     if(tmpt < 0) {
```

```
106                         tmpt *= -1;
107                         if(tmpt >= 15 && gameEnemies.GetCount() > 0) {
108                             scoreTimeUp = true;
109                             SetState(State.SHOW_GAME_OVER);
110                         } else if(tmpt >= 0 && gameEnemies.GetCount()
                            == 0) {
111                             tmpt *= 100;
112                             if(gameType == GameType.GAME_TWO_PLAYER) {
113                                 tmpt = tmpt/2;
114                                 scorePlayerOne -= tmpt;
115                                 if(scorePlayerOne < 0) {
116                                     scorePlayerOne = 0;
117                                 }
118                                 SetScoreLeftText(scorePlayerOne);
119
120                                 scorePlayerTwo -= tmpt;
121                                 if(scorePlayerTwo < 0) {
122                                     scorePlayerTwo = 0;
123                                 }
124                                 SetScoreRightText(scorePlayerTwo);
125                             } else {
126                                 scorePlayerOne -= tmpt;
127                                 if(scorePlayerOne < 0) {
128                                     scorePlayerOne = 0;
129                                 }
130                                 SetScoreLeftText(scorePlayerOne);
131                             }
132
133                             if(gameEnemies.GetCount() == 0) {
134                                 wavesCurrentIdx++;
135                                 if(randomWaves) {
136                                     wavesCurrentIdx = MmgHelper.
                                    GetRandomIntRange(0, waves.length);
137                                 }
```

```
138                             SetState(State.SHOW_COUNT_DOWN_IN_GAME);
139                         }
140                     }
141                 }
142             }
143             txtLevelTime.SetText("Time: " +
                FormatTime(wavesCurrent.timeTotalMs - wavesCurrent.
                timeCurrentMs));
144         }
145
146         if(player1 != null && player1.mod != null) {
147             if(player1.mod == MdtPlayerModType.INVINCIBLE) {
148                 txtPlayer1Mod.SetText("M: Invinc");
149                 txtPlayer1ModTime.SetText("MT: " +
                    FormatMod((player1.modTimingInvTotal - player1.
                    modTimingInv)));
150             } else if(player1.mod == MdtPlayerModType.DOUBLE_
                POINTS) {
151                 txtPlayer1Mod.SetText("M: DblPts");
152                 txtPlayer1ModTime.SetText("MT: " +
                    FormatMod((player1.modTimingDpTotal - player1.
                    modTimingDp)));
153             } else if(player1.mod == MdtPlayerModType.FULL_HEALTH) {
154                 txtPlayer1Mod.SetText("M: FullHlth");
155                 txtPlayer1ModTime.SetText("MT: " +
                    FormatMod((player1.modTimingFullHealthTotal -
                    player1.modTimingFullHealth)));
156             } else {
157                 txtPlayer1Mod.SetText("M: -");
158                 txtPlayer1ModTime.SetText("MT: 0000");
159             }
160         } else {
161             txtPlayer1Mod.SetText("M: -");
162             txtPlayer1ModTime.SetText("MT: 0000");
163         }
164
```

```
165             if(player2 != null && player2.mod != null) {
166                 if(player2.mod == MdtPlayerModType.INVINCIBLE) {
167                     txtPlayer2Mod.SetText("M: Invinc");
168                     txtPlayer2ModTime.SetText("MT: " +
                        FormatMod((player2.modTimingInvTotal - player2.
                        modTimingInv)));
169                 } else if(player2.mod == MdtPlayerModType.DOUBLE_
                    POINTS) {
170                     txtPlayer2Mod.SetText("M: DblPts");
171                     txtPlayer2ModTime.SetText("MT: " +
                        FormatMod((player2.modTimingDpTotal - player2.
                        modTimingDp)));
172                 } else if(player2.mod == MdtPlayerModType.FULL_HEALTH) {
173                     txtPlayer2Mod.SetText("M: FullHlth");
174                     txtPlayer2ModTime.SetText("MT: " +
                        FormatMod((player2.modTimingFullHealthTotal -
                        player2.modTimingFullHealth)));
175                 } else {
176                     txtPlayer2Mod.SetText("M: -");
177                     txtPlayer2ModTime.SetText("MT: 0000");
178                 }
179             } else {
180                 txtPlayer2Mod.SetText("M: -");
181                 txtPlayer2ModTime.SetText("MT: 0000");
182             }
183             break;
184     }
185     pause = false;
186 }
```

Let's focus our attention on the first method listed, the CanMove method. This method is used as the basis for all collisions in the game. The first method argument is the rectangle we're looking to find collisions for. The second argument is an optional ignore object. The ignore object will be escaped from the collision search. This feature is used to prevent an object finding a collision with itself. The code on lines 10–12 is for processing the contents of container child objects.

Notice that we don't recurse and search through multiple levels of containers. That's because we want this method to run as quickly as possible, so we only support one level of searching containers. Also note that only objects that are visible are considered for collisions for obvious reasons. The second version of the `CanMove` method is used to find the target rectangle's collision, if any, with no escape object. These methods are used by the `MdtObjPush`, `MdtCharInter`, and `MdtWeapon` classes to determine collisions and responses. The next method we'll cover is the `DrawScreen` method. This method is long, but the code is direct and redundant in many places.

The `DrawScreen` method is part of the class' update routine along with the `MmgUpdate` method. The `MmgUpdate` method handles timing-related updates, while the `DrawScreen` method handles drawing-related updates. The first main responsibility of the method is to handle flipping the numbers during the screen's countdown states, lines 8 and 9. The different number states are handled on lines 10–89. Note that each number state ticks down to number one, which then sets the screen to the `SHOW_GAME` state on line 41.

The second main responsibility of the method is to monitor the game and update the player's HUD/UI data while checking game exit conditions. The `SHOW_GAME` state is handled from lines 91 to 184. The code on lines 93–102 increments the level, enemy wave, to the next wave detecting a game over condition if the last level is reached.

The overtime segment of the game, when the level time goes below zero, is handled on lines 103–144. The last two blocks of code on lines 146–163 and 165–182 handle updating the HUD with current game data for both player1 and player2. The next block of methods for us to review are those that add new items and objects to the game.

Listing 24-22. ScreenGame Game Logic Methods 2

```
01 public MdtItem UpdateGenerateMdtItem(int x, int y, MdtItemType[] items) {
02     int idx = MmgHelper.GetRandomInt(items.length);
03     MdtItemType t = items[idx];
04     MdtItem itm = null;
05
06     if(t == MdtItemType.BOMB) {
07         MdtItemBomb bomb = new MdtItemBomb();
08         itm = bomb;
09     } else if(t == MdtItemType.COIN_BAG) {
10         MdtItemCoinBag coins = new MdtItemCoinBag();
```

```
11          itm = coins;
12      } else if(t == MdtItemType.POTION_GREEN) {
13          MdtItemPotionGreen potion1 = new MdtItemPotionGreen();
14          itm = potion1;
15      } else if(t == MdtItemType.POTION_RED) {
16          MdtItemPotionRed potion2 = new MdtItemPotionRed();
17          itm = potion2;
18      } else if(t == MdtItemType.POTION_YELLOW) {
19          MdtItemPotionYellow potion3 = new MdtItemPotionYellow();
20          itm = potion3;
21      } else {
22          MdtItemPotionGreen potion1 = new MdtItemPotionGreen();
23          itm = potion1;
24      }
25      return itm;
26  }
```

```
01 public MdtBase UpdateGenerateMdtItem(int x, int y) {
02      if(wavesCurrent != null) {
03            MdtItem itm =  UpdateGenerateMdtItem(x, y, wavesCurrent.
              activeItems);
04            if(itm != null) {
05             itm.SetScreen(this);
06             itm.SetPosition(x, y);
07             gameItems.Add(itm);
08          }
09          return itm;
10      }
11      return null;
12   }
```

```
01 public MdtObjPush UpdateGenerateMdtObj(int x, int y) {
02      int idx = MmgHelper.GetRandomInt(3);
03      MdtObjPush objPush = null;
04
```

```
    if(idx == 0) {
        objPush = new MdtObjPushBarrel(this);
    } else if(idx == 1) {
        objPush = new MdtObjPushTableSmall(this);
    } else if(idx == 2) {
        objPush = new MdtObjPushTableLarge(this);
    } else {
        objPush = new MdtObjPushBarrel(this);
    }
    return objPush;
}
```

```
private void UpdateGenerateObjects(int cnt, MdtItemType[] items) {
    int tmp = 0;
    int tmpl = 0;
    int x = 0;
    int y = 0;
    int w = 0;
    int h = 0;

    MdtBase obj = null;
    MdtItemType t;
    MmgRect r = null;
    MdtBase coll = null;

    for(int i = 0; i < cnt; i++) {
        tmp = MmgHelper.GetRandomInt(11) % 2;
        tmpl = MmgHelper.GetRandomInt(11) % 2;

        if(tmpl == 0) {
            x = MmgHelper.GetRandomIntRange(randoLeft.GetLeft(),
            randoLeft.GetRight());
            y = MmgHelper.GetRandomIntRange(randoLeft.GetTop(),
            randoLeft.GetBottom());
        } else {
```

```
            x = MmgHelper.GetRandomIntRange(randoRight.GetLeft(),
            randoRight.GetRight());
            y = MmgHelper.GetRandomIntRange(randoRight.GetTop(),
            randoRight.GetBottom());
        }

        if(tmp == 0) {
            MdtItem itm = UpdateGenerateMdtItem(x, y, items);

            if(itm != null) {
                itm.SetScreen(this);
                itm.SetPosition(x, y);
                w = itm.GetWidth();
                h = itm.GetHeight();
                pos = itm.GetPosition();
                obj = itm;
                gameItems.Add(itm);
            }
        } else {
            MdtObjPush objPush = UpdateGenerateMdtObj(x, y);

            if(objPush != null) {
                objPush.SetScreen(this);
                objPush.SetPosition(x, y);
                w = objPush.GetWidth();
                h = objPush.GetHeight();
                pos = objPush.GetPosition();
                obj = objPush;
                gameObjects.Add(objPush);
            }
        }

        if(obj != null) {
            x -= MmgHelper.ScaleValue(32);
            y -= MmgHelper.ScaleValue(32);
            w += MmgHelper.ScaleValue(32);
```

```
56              h += MmgHelper.ScaleValue(32);
57              r = new MmgRect(x, y, y + h, x + w);
58              coll = CanMove(r);
59
60              while(coll != null) {
61                  if(tmpl == 0) {
62                      x = MmgHelper.GetRandomIntRange(randoLeft.GetLeft(),
                        randoLeft.GetRight());
63                      y = MmgHelper.GetRandomIntRange(randoLeft.GetTop(),
                        randoLeft.GetBottom());
64                  } else {
65                      x = MmgHelper.GetRandomIntRange(randoRight.
                        GetLeft(), randoRight.GetRight());
66                      y = MmgHelper.GetRandomIntRange(randoRight.GetTop(),
                        randoRight.GetBottom());
67                  }
68
69                  if(rand.nextInt(11) % 2 == 0) {
70                      x -= MmgHelper.ScaleValue(32);
71                  } else {
72                      x += MmgHelper.ScaleValue(32);
73                  }
74
75                  if(rand.nextInt(11) % 2 == 0) {
76                      y -= MmgHelper.ScaleValue(32);
77                  } else {
78                      y += MmgHelper.ScaleValue(32);
79                  }
80                  r = new MmgRect(x, y, y + h, x + w);
81                  coll = CanMove(r);
82              }
83
84              obj.SetPosition(x, y);
85          }
86      }
87 }
```

```
01 private void UpdateStartEnemyWave(int waveIdx) {
02     wavesCurrentIdx = waveIdx;
03     wavesCurrent = waves[wavesCurrentIdx];
04     wavesCurrent.timeStartMs = System.currentTimeMillis();
05     wavesCurrent.timeCurrentMs = 0;
06     wavesCurrent.timeIntervalMs = 0;
07     wavesCurrent.actEnemyCount = 0;
08     txtLevel.SetText("Level: " + FormatLevel(wavesCurrentIdx));
09
10     UpdateLockAllDoors();
11     if(wavesCurrent.activeDoors != null) {
12         int i = 0;
13         int len = wavesCurrent.activeDoors.length;
14         for(; i < len; i++) {
15             UpdateUnlockDoor(wavesCurrent.activeDoors[i]);
16         }
17     }
18
19     UpdateClearObjects();
20     UpdateGenerateObjects(wavesCurrent.actObjCount, wavesCurrent.
       activeItems);
21 }
```

```
01 private void UpdateResetPlayers() {
02     if(gameType == GameType.GAME_ONE_PLAYER || gameType == GameType.
       GAME_TWO_PLAYER) {
03         MmgObj obj = new MmgObj(player1.GetWidth(), player1.
           GetHeight());
04         MmgHelper.CenterHorAndVert(obj);
05         obj.SetX(obj.GetX() - (GAME_WIDTH - BOARD_WIDTH)/2 + obj.
           GetWidth());
06         obj.SetY(obj.GetY() - player1.GetHeight() - MmgHelper.
           ScaleValue(10));
07
08         player1.isMoving = false;
09         player1.isAttacking = false;
```

```
10          player1.isPushStart = false;
11          player1.isPushing = false;
12          player1.SetPosition(obj.GetPosition().Clone());
13          player1HealthBar.RestoreAllHealth();
14          player1.SetHealthToMax();
15          playersAliveCount = 1;
16      }
17
18      if(gameType == GameType.GAME_TWO_PLAYER) {
19          MmgObj obj = new MmgObj(player2.GetWidth(), player2.
            GetHeight());
20          MmgHelper.CenterHorAndVert(obj);
21          obj.SetX(obj.GetX() - (GAME_WIDTH - BOARD_WIDTH)/2 + obj.
            GetWidth());
22          obj.SetY(obj.GetY() + player2.GetHeight() + MmgHelper.
            ScaleValue(20));
23
24          player2.isMoving = false;
25          player2.isAttacking = false;
26          player2.isPushStart = false;
27          player2.isPushing = false;
28          player2.SetPosition(obj.GetPosition().Clone());
29          player2HealthBar.RestoreAllHealth();
30          player2.SetHealthToMax();
31          playersAliveCount = 2;
32      }
33  }
```

The first method we'll look at is the `UpdateGenerateMdtItem` method. This method is used to generate a random item governed by an array of item types. The randomly chosen item is generated on lines 6–24. The resulting `MdtItem` instance is returned. The subsequent version of the method is used to automatically apply the current level's item list. The next method for us to cover is the `UpdateGenerateMdtObj` method, which is also used to generate a random game object.

This method follows a similar structure, but it's used to generate one of three `MdtObjPush` objects to add to the game. The randomly chosen game object is generated on lines 5–13. Both the previously reviewed methods are used by the next method up for review, the `UpdateGenerateObjects` method. This method is used to prepare the level with a few random game objects.

For the specified number of objects, a randomly generated position, using the `randoLeft` and `randoRight` class fields, is created on lines 18–24. Based on another random number, either an item or an object is created. The code that generates and registers the item is on lines 27–37. If a game object, as opposed to a game item, is generated, the code on lines 39–49 executes. To ensure that we don't pile up items, the code on lines 52–85 moves the generated object or item around until no collisions are detected.

The `UpdateStartEnemyWave` method is used to prep the next level, enemy wave, for use. On lines 2–7, the current wave is reset. All timing values and counts are reset and ready for use. The level text HUD/UI is updated on line 8, and all doors are locked on line 10 with select doors opened on lines 11–17. The board is cleared on line 19, and new game objects are added to the game on line 20. This method is called in conjunction with the `UpdateResetPlayers` method listed in the preceding. Notice how each player is reset for a new game.

At this point in the game, we can start a level and move around bumping into things, but no collision logic exists! Let's define how new enemies are added to the level and how collisions will work.

Listing 24-23. ScreenGame Game Logic Methods 3

```
001 private void UpdateCurrentEnemyWave(long msSinceLastFrame) {
002     if(wavesCurrent != null) {
003         wavesCurrent.timeIntervalMs += msSinceLastFrame;
004         wavesCurrent.timeCurrentMs += msSinceLastFrame;
005
006         if(wavesCurrent.timeIntervalMs >= wavesCurrent.
            intervalBetweenEnemiesMs && wavesCurrent.actEnemyCount <
            wavesCurrent.enemyCount) {
007             wavesCurrent.timeIntervalMs = 0;
008             int len = wavesCurrent.actAtOneTime;
009             int dLen = wavesCurrent.activeDoors.length;
```

```
010             int eLen = 8;
011             int dIdx = 0;
012             int eIdx = 0;
013             MdtBase coll = null;
014             MdtCharInter emn = null;
015             MdtDoorType door;
016             int adjW = enemyDemonFrames.GetWidth() + MmgHelper.
                ScaleValue(12);
017             int adjH = enemyDemonFrames.GetHeight() + MmgHelper.
                ScaleValue(12);
018             boolean found = false;
019
020             for(int i = 0; i < len; i++) {
021                 dIdx = MmgHelper.GetRandomIntRange(0, dLen);
022                 found = false;
023                 door = wavesCurrent.activeDoors[dIdx];
024
025                 eIdx = MmgHelper.GetRandomIntRange(0, eLen);
026                 if(eIdx == 7) {
027                     emn = new MdtCharInterBanshee(enemyBansheeFrames.
                        CloneTyped(), 0, 3, 12, 15, 4, 7, 8, 11, this);
028                 } else if(eIdx % 2 == 0 || eIdx == 3) {
029                     emn = new MdtCharInterDemon(enemyDemonFrames.
                        CloneTyped(), 0, 3, 12, 15, 4, 7, 8, 11, this);
030                 } else if(eIdx % 2 == 1 || eIdx == 5) {
031                     emn = new MdtCharInterWarlock(enemyWarlockFrames.
                        CloneTyped(), 0, 3, 12, 15, 4, 7, 8, 11, this);
032                 } else {
033                     emn = new MdtCharInterBanshee(enemyBansheeFrames.
                        CloneTyped(), 0, 3, 12, 15, 4, 7, 8, 11, this);
034                 }
035
036                 if(door == MdtDoorType.TOP_LEFT) {
037                     emn.SetDir(MmgDir.DIR_FRONT);
```

```
038                     emn.SetPosition(doorTopLeftOpened.GetPosition().
                        Clone());
039                     emn.SetY(emn.GetY() + adjH);
040
041                     coll = CanMove(emn.GetRect(), emn);
042                     if(coll == null) {
043                         found = true;
044                     } else {
045                         emn.SetX(emn.GetX() + adjW);
046                         coll = CanMove(emn.GetRect(), emn);
047                         if(coll == null) {
048                             found = true;
049                         } else {
050                             emn.SetX(emn.GetX() - adjW - adjW);
051                             coll = CanMove(emn.GetRect(), emn);
052                             if(coll == null) {
053                                 found = true;
054                             } else {
055                                 emn.SetX(emn.GetX() + adjW);
056                                 emn.SetY(emn.GetY() + adjH);
057                                 coll = CanMove(emn.GetRect(), emn);
058                                 if(coll == null) {
059                                     found = true;
060                                 }
061                             }
062                         }
063                     }
064                 } else if(door == MdtDoorType.TOP_RIGHT) {
065                     emn.SetDir(MmgDir.DIR_FRONT);
066                     emn.SetPosition(doorTopRightOpened.GetPosition().
                        Clone());
067                     emn.SetY(emn.GetY() + adjH);
068
069                     coll = CanMove(emn.GetRect(), emn);
070                     if(coll == null) {
```

```
071                         found = true;
072                     } else {
073                         emn.SetX(emn.GetX() + adjW);
074                         coll = CanMove(emn.GetRect(), emn);
075                         if(coll == null) {
076                             found = true;
077                         } else {
078                             emn.SetX(emn.GetX() - adjW - adjW);
079                             coll = CanMove(emn.GetRect(), emn);
080                             if(coll == null) {
081                                 found = true;
082                             } else {
083                                 emn.SetX(emn.GetX() + adjW);
084                                 emn.SetY(emn.GetY() + adjH);
085                                 coll = CanMove(emn.GetRect(), emn);
086                                 if(coll == null) {
087                                     found = true;
088                                 }
089                             }
090                         }
091                     }
092                 } else if(door == MdtDoorType.LEFT) {
093                     emn.SetDir(MmgDir.DIR_RIGHT);
094                     emn.SetPosition(doorLeftLockIcon.GetPosition().
                        Clone());
095                     emn.SetX(emn.GetX() + adjW);
096
097                     coll = CanMove(emn.GetRect(), emn);
098                     if(coll == null) {
099                         found = true;
100                     } else {
101                         emn.SetY(emn.GetY() + adjH);
102                         coll = CanMove(emn.GetRect(), emn);
103                         if(coll == null) {
104                             found = true;
```

```
105                         } else {
106                             emn.SetY(emn.GetY() - adjH - adjH);
107                             coll = CanMove(emn.GetRect(), emn);
108                             if(coll == null) {
109                                 found = true;
110                             } else {
111                                 emn.SetX(emn.GetX() - adjW);
112                                 emn.SetY(emn.GetY() - adjH);
113                                 coll = CanMove(emn.GetRect(), emn);
114                                 if(coll == null) {
115                                     found = true;
116                                 }
117                             }
118                         }
119                     }
120                 } else if(door == MdtDoorType.BOTTOM_LEFT) {
121                     emn.SetDir(MmgDir.DIR_BACK);
122                     emn.SetPosition(doorBotLeftLockIcon.GetPosition().
                        Clone());
123                     emn.SetX(emn.GetY() - adjH);
124
125                     coll = CanMove(emn.GetRect(), emn);
126                     if(coll == null) {
127                         found = true;
128                     } else {
129                         emn.SetX(emn.GetX() + adjW);
130                         coll = CanMove(emn.GetRect(), emn);
131                         if(coll == null) {
132                             found = true;
133                         } else {
134                             emn.SetX(emn.GetX() - adjW - adjW);
135                             coll = CanMove(emn.GetRect(), emn);
136                             if(coll == null) {
137                                 found = true;
138                             } else {
```

```
                                emn.SetX(emn.GetX() - adjW);
                                emn.SetY(emn.GetY() - adjH);
                                coll = CanMove(emn.GetRect(), emn);
                                if(coll == null) {
                                    found = true;
                                }
                            }
                        }
                    }
                } else if(door == MdtDoorType.BOTTOM_RIGHT) {
                    emn.SetDir(MmgDir.DIR_BACK);
                    emn.SetPosition(doorBotRightLockIcon.GetPosition().
                    Clone());
                    emn.SetX(emn.GetY() - adjH);

                    coll = CanMove(emn.GetRect(), emn);
                    if(coll == null) {
                        found = true;
                    } else {
                        emn.SetX(emn.GetX() + adjW);
                        coll = CanMove(emn.GetRect(), emn);
                        if(coll == null) {
                            found = true;
                        } else {
                            emn.SetX(emn.GetX() - adjW - adjW);
                            coll = CanMove(emn.GetRect(), emn);
                            if(coll == null) {
                                found = true;
                            } else {
                                emn.SetX(emn.GetX() - adjW);
                                emn.SetY(emn.GetY() - adjH);
                                coll = CanMove(emn.GetRect(), emn);
                                if(coll == null) {
                                    found = true;
                                }
                            }
```

```
174                         }
175                     }
176                 } else if(door == MdtDoorType.RIGHT) {
177                     emn.SetDir(MmgDir.DIR_LEFT);
178                     emn.SetPosition(doorRightLockIcon.GetPosition().
                    Clone());
179                     emn.SetX(emn.GetX() - adjW);
180
181                     coll = CanMove(emn.GetRect(), emn);
182                     if(coll == null) {
183                         found = true;
184                     } else {
185                         emn.SetY(emn.GetY() + adjH);
186                         coll = CanMove(emn.GetRect(), emn);
187                         if(coll == null) {
188                             found = true;
189                         } else {
190                             emn.SetY(emn.GetY() - adjH - adjH);
191                             coll = CanMove(emn.GetRect(), emn);
192                             if(coll == null) {
193                                 found = true;
194                             } else {
195                                 emn.SetX(emn.GetX() - adjW);
196                                 emn.SetY(emn.GetY() - adjH);
197                                 coll = CanMove(emn.GetRect(), emn);
198                                 if(coll == null) {
199                                     found = true;
200                                 }
201                             }
202                         }
203                     }
204                 }
205
```

```
206                 if(found) {
207                     wavesCurrent.actEnemyCount++;
208                     gameEnemies.Add(emn);
209                     emn.SetMotor(MdtEnemyMotorType.MOVE_Y_THEN_X);
210                 }
211             }
212         }
213     }
214 }
```

```
01 public void UpdateProcessWeaponCollision(MdtBase o1, MdtWeapon o2,
   MmgRect weapon) {
02     MdtPlayerType tp = MdtPlayerType.NONE;
03     if(o2.GetHolder().GetMdtSubType() == MdtObjSubType.PLAYER_1) {
04         tp = MdtPlayerType.PLAYER_1;
05     } else if(o2.GetHolder().GetMdtSubType() == MdtObjSubType.PLAYER_2) {
06         tp = MdtPlayerType.PLAYER_2;
07     } else if(o2.GetHolder().GetMdtSubType() == MdtObjSubType.ENEMY_
       BANSHEE || o2.GetHolder().GetMdtSubType() == MdtObjSubType.ENEMY_
       DEMON || o2.GetHolder().GetMdtSubType() == MdtObjSubType.ENEMY_
       WARLOCK) {
08         tp = MdtPlayerType.ENEMY;
09     }
10
11     if(o1 instanceof MdtCharInter && ((MdtCharInter)o1).GetPlayerType()
       == MdtPlayerType.ENEMY && o2.GetHolder().GetMdtType() != MdtObjType.
       ENEMY) {
12         MdtCharInter mci = (MdtCharInter)o1;
13         if(!mci.isBouncing && tp != MdtPlayerType.NONE) {
14             UpdateAddPoints(weapon.GetPosition(), MdtPointsType.PTS_100,
               tp);
15             mci.Bounce(weapon.GetPosition(), weapon.GetWidth()/2,
               weapon.GetHeight()/2, o2.GetHolder().GetDir(), tp);
16
17             if(sound1 != null) {
18                 sound1.Play();
```

```
19              }
20          }
21      } else if(o1 instanceof MdtCharInter && ((MdtCharInter)o1).
        GetPlayerType() != MdtPlayerType.ENEMY && o2.GetHolder().
        GetMdtType() == MdtObjType.ENEMY) {
22          MdtCharInter mci = (MdtCharInter)o1;
23          if(!mci.isBouncing && tp != MdtPlayerType.NONE) {
24              UpdateAddPoints(weapon.GetPosition(), MdtPointsType.PTS_100,
                tp);
25              mci.Bounce(weapon.GetPosition(), weapon.GetWidth()/2,
                weapon.GetHeight()/2, o2.GetHolder().GetDir(), tp);
26
27              if(sound1 != null) {
28                  sound1.Play();
29              }
30          }
31      }
32  }
```

```
01 public void UpdateProcessWeaponCollision(MdtBase o1, MdtCharInter o2,
   MmgRect weapon) {
02      if(o1 instanceof MdtCharInter && ((MdtCharInter)o1).GetPlayerType() ==
        MdtPlayerType.ENEMY && o2.GetPlayerType() != MdtPlayerType.ENEMY) {
03          MdtCharInter mci = (MdtCharInter)o1;
04          if(!mci.isBouncing) {
05              UpdateAddPoints(weapon.GetPosition(), MdtPointsType.PTS_100,
                o2.GetPlayerType());
06              mci.Bounce(weapon.GetPosition(), weapon.GetWidth()/2,
                weapon.GetHeight()/2, o2.GetDir(), o2.GetPlayerType());
07
08              if(sound1 != null) {
09                  sound1.Play();
10              }
11          }
```

```
12      } else if(o1 instanceof MdtCharInter && ((MdtCharInter)o1).
        GetPlayerType() != MdtPlayerType.ENEMY && o2.GetPlayerType() ==
        MdtPlayerType.ENEMY) {
13          MdtCharInter mci = (MdtCharInter)o1;
14          if(!mci.isBouncing) {
15              UpdateAddPoints(weapon.GetPosition(), MdtPointsType.PTS_100,
                o2.GetPlayerType());
16              mci.Bounce(weapon.GetPosition(), weapon.GetWidth()/2,
                weapon.GetHeight()/2, o2.GetDir(), o2.GetPlayerType());
17
18              if(sound1 != null) {
19                  sound1.Play();
20              }
21          }
22      }
23  }
```

The `UpdateCurrentEnemyWave` method is a long one, and at first glance it looks complex; but once you break down what's happening, it's not so bad. If a new group of enemies is active and we have more to create on the current level, we prepare to add more enemies, line 6. For the known number of enemies that we can add at one time, line 20, we generate a new random enemy.

The randomly chosen enemy is created on lines 25–34. For each different door type, there is a block of code that tries to find an open space to place an enemy character. One such code block, on lines 36–63, finds an open space for the top-left door. The rest of the method performs the same operation but for each of the different doors on the game board. The last few lines of code, lines 206–210, register the enemy and turn on its AI motor. This makes the enemy automatically track a player character across the game board.

The next method we'll look at, also listed in the preceding, is the first `UpdateProcessWeaponCollision` method. The method takes an `MdtBase`, an `MdtWeapon`, and an `MmgRect` as arguments. The player type is determined from the holder of the weapon, lines 2–9. If the weapon holder is not an enemy and the target object is an enemy, the enemy is bounced and takes damage, points are displayed, and a sound effect is played, lines 12–20. The same thing happens if an enemy weapon damages a player, lines 21–31. That feature isn't fully implemented yet, but you'll have plenty of time to add that and more.

The subsequent version of the method is very similar, but it doesn't determine the player type from the weapon holder. Instead, the player type is determined from the `MdtCharInter` argument. Look over this method and make sure you understand it before adding it to your project. I'll show you a shortcut on how to copy and paste the Phase 3 code into your game project at the end of this section. The last method we have to review is the `UpdateProcessCollision` method. This method will activate game object interactions with the player character and complete the game's logic.

Listing 24-24. ScreenGame Game Logic Methods 4

```
01 if(o1 instanceof MdtObjPushBarrel || o1 instanceof MdtObjPushTableSmall
   || o1 instanceof MdtObjPushTableLarge) {
02     hasObjO1 = true;
03 } else if(o2 instanceof MdtObjPushBarrel || o2 instanceof
   MdtObjPushTableSmall || o2 instanceof MdtObjPushTableLarge) {
04     hasObjO2 = true;
05 }
06
07 if(o1.GetMdtSubType() == MdtObjSubType.PLAYER_1) {
08     if(hasObjO1 || hasObjO2) {
09         found = true;
10     }
11 } else if(o2.GetMdtSubType() == MdtObjSubType.PLAYER_1) {
12     if(hasObjO1 || hasObjO2) {
13         found = true;
14     }
15 }
16
17 if(found) {
18     if(!player1.isPushStart && !player1.isPushing) {
19         player1.isPushStart = true;
20         player1.pushingCurrentMs = 0;
21         player1.pushingStartMs = System.currentTimeMillis();
22     } else if(player1.isPushStart) {
23
```

```
24      } else if(player1.isPushing) {
25          player1.isPushing = false;
26
27          if(hasObjO1) {
28              if(o1 instanceof MdtObjPushBarrel) {
29                  t = (MdtObjPushBarrel)o1;
30                  if(t.isBeingPushed == false) {
31                      UpdateAddPoints(o1.GetPosition().Clone(),
                        MdtPointsType.PTS_100, MdtPlayerType.PLAYER_1);
32                  }
33                  t.isBeingPushed = true;
34                  t.pushDir = player1.dir;
35                  t.pushedBy = MdtPlayerType.PLAYER_1;
36              } else if(o1 instanceof MdtObjPushTableSmall) {
37                  st = (MdtObjPushTableSmall)o1;
38                  if(st.isBeingPushed == false) {
39                      UpdateAddPoints(o1.GetPosition().Clone(),
                        MdtPointsType.PTS_100, MdtPlayerType.PLAYER_1);
40                  }
41                  st.isBeingPushed = true;
42                  st.pushDir = player1.dir;
43                  st.pushedBy = MdtPlayerType.PLAYER_1;
44              } else if(o1 instanceof MdtObjPushTableLarge) {
45                  lt = (MdtObjPushTableLarge)o1;
46                  if(lt.isBeingPushed == false) {
47                      UpdateAddPoints(o1.GetPosition().Clone(),
                        MdtPointsType.PTS_100, MdtPlayerType.PLAYER_1);
48                  }
49                  lt.isBeingPushed = true;
50                  lt.pushDir = player1.dir;
51                  lt.pushedBy = MdtPlayerType.PLAYER_1;
52              }
53          } else {
54              if(o2 instanceof MdtObjPushBarrel) {
55                  t = (MdtObjPushBarrel)o2;
```

```
56                  if(t.isBeingPushed == false) {
57                      UpdateAddPoints(o2.GetPosition().Clone(),
                        MdtPointsType.PTS_100, MdtPlayerType.PLAYER_1);
58                  }
59                  t.isBeingPushed = true;
60                  t.pushDir = player1.dir;
61                  t.pushedBy = MdtPlayerType.PLAYER_1;
62              } else if(o2 instanceof MdtObjPushTableSmall) {
63                  st = (MdtObjPushTableSmall)o2;
64                  if(st.isBeingPushed == false) {
65                      UpdateAddPoints(o2.GetPosition().Clone(),
                        MdtPointsType.PTS_100, MdtPlayerType.PLAYER_1);
66                  }
67                  st.isBeingPushed = true;
68                  st.pushDir = player1.dir;
69                  st.pushedBy = MdtPlayerType.PLAYER_1;
70              } else if(o2 instanceof MdtObjPushTableLarge) {
71                  lt = (MdtObjPushTableLarge)o2;
72                  if(lt.isBeingPushed == false) {
73                      UpdateAddPoints(o2.GetPosition().Clone(),
                        MdtPointsType.PTS_100, MdtPlayerType.PLAYER_1);
74                  }
75                  lt.isBeingPushed = true;
76                  lt.pushDir = player1.dir;
77                  lt.pushedBy = MdtPlayerType.PLAYER_1;
78              }
79          }
80      }
81 } else {
82     player1.isPushStart = false;
83     player1.isPushing = false;
84     player1.pushingCurrentMs = 0;
85     player1.pushingStartMs = 0;
86 }
```

```
87
88 if(found) {
89     return;
90 }
```

Because the `UpdateProcessCollision` method is so large, we'll only be able to review pertinent snippets of code from the method. On lines 1–5, a determination is made if a collision with an `MdtObjPush` object occurs. The player involved with the collision is determined on lines 7–15. If a collision is found for player1, then we determine if the player is pushing the object.

The code on lines 18–21 triggers the push start state. This is to prevent objects being immediately pushed by the player's character. You have to push up against the object for a few milliseconds, push start, in order to trigger a push. On line 24 when the push is triggered, then each type of object is handled separately. The object is sent flying, and points are awarded to the player that pushed the object.

On lines 28–52, the same collision is processed except that the player and the object they're pushing are reversed now. This pattern of code is repeated for player2 and for the three pushable objects supported by the game: the barrel, the small table, and the large table. It takes a lot of code to power these comparisons, which is why we escape the method after any work has been done, lines 88–90. Subsequent collisions can be picked up on the next game frame.

The next type of collisions handled by this method are the player and item collisions. Let's take a look at the code powering this feature.

Listing 24-25. ScreenGame Game Logic Methods 5

```
01 boolean hasItemPotionRedO1 = false;
02 boolean hasItemPotionRedO2 = false;
03 if(o1 instanceof MdtItemPotionRed) {
04     hasItemPotionRedO1 = true;
05     RemoveObj(o1);
06 } else if(o2 instanceof MdtItemPotionRed) {
07     hasItemPotionRedO2 = true;
08     RemoveObj(o2);
09 }
10
```

```
11 if(o1.GetMdtSubType() == MdtObjSubType.PLAYER_1) {
12     if(hasItemPotionRedO1 || hasItemPotionRedO2) {
13         player1.SetMod(MdtPlayerModType.DOUBLE_POINTS);
14         player1.modTimingDp = 0;
15         player1.SetHasDoublePoints(true);
16        UpdatePlayerMod(player1.playerType, MdtPlayerModType.DOUBLE_POINTS);
17         UpdateAddPoints(o2.GetPosition().Clone(), ((MdtItemPotionRed)
           o2).points, MdtPlayerType.PLAYER_1);
18         found = true;
19     }
20 } else if(o2.GetMdtSubType() == MdtObjSubType.PLAYER_1) {
21     if(hasItemPotionRedO1 || hasItemPotionRedO2) {
22         player1.SetMod(MdtPlayerModType.DOUBLE_POINTS);
23         player1.modTimingDp = 0;
24         player1.SetHasDoublePoints(true);
25        UpdatePlayerMod(player1.playerType, MdtPlayerModType.DOUBLE_POINTS);
26         UpdateAddPoints(o1.GetPosition().Clone(), ((MdtItemPotionRed)
           o1).points, MdtPlayerType.PLAYER_1);
27         found = true;
28     }
29 }
```

The next type of collision we'll look at, listed in the preceding, is an item modifier collision. In this case, we're again looking at the player1 collision detection. On lines 3–9, we determine if either collision object is a red potion. Notice that the object is removed upon identification, lines 5 and 8. In the same fashion as the previous collision detection code, the collision is processed if the other object is player1. The modifier, in this case double points, is registered on lines 13–18 or on lines 22–27 depending on the collision. Notice that the player's character is altered to indicate the modifier.

The same code exists for player2 and for all three potions, green, red, and yellow. Each modifier collision type causes a different effect in the game expressed in code. The last type of collision that we're going to review is the enemy collision.

Listing 24-26. ScreenGame Game Logic Methods 6

```
01 boolean hasEnemyO1 = false;
02 boolean hasEnemyO2 = false;
03 if(o1 instanceof MdtCharInter && ((MdtCharInter)o1).playerType ==
   MdtPlayerType.ENEMY) {
04     hasEnemyO1 = true;
05 } else if(o2 instanceof MdtCharInter && ((MdtCharInter)o2).playerType ==
   MdtPlayerType.ENEMY) {
06     hasEnemyO2 = true;
07 }
08
09 MdtCharInter ecO1;
10 MdtCharInter ecO2;
11
12 if(o1.GetMdtSubType() == MdtObjSubType.PLAYER_1) {
13     if(hasEnemyO1 || hasEnemyO2) {
14         if(player1.isBouncing == false) {
15             ecO2 = (MdtCharInter)o2;
16             if(!player1.hasFullHealth && !player1.hasInvincibility) {
17                 player1HealthBar.TakeDamage(1);
18                 player1.TakeDamage(1, ecO2.playerType);
19             }
20
21             if(!player1.isBroken) {
22                 player1.Bounce(ecO2.GetPosition(), ecO2.GetWidth()/2,
                    ecO2.GetHeight()/2, GetOppositeDir(player1.dir), ecO2.
                    playerType);
23             }
24         }
25         found = true;
26     }
27 } else if(o2.GetMdtSubType() == MdtObjSubType.PLAYER_1) {
28     if(hasEnemyO1 || hasEnemyO2) {
29         if(player1.isBouncing == false) {
30             ecO1 = (MdtCharInter)o1;
```

```
31              if(!player1.hasFullHealth && !player1.hasInvincibility) {
32                  player1HealthBar.TakeDamage(1);
33                  player1.TakeDamage(1, ec01.playerType);
34              }
35
36              if(!player1.isBroken) {
37                  player1.Bounce(ec01.GetPosition(), ec01.GetWidth()/2,
                    ec01.GetHeight()/2, GetOppositeDir(player1.dir), ec01.
                    playerType);
38              }
39          }
40          found = true;
41      }
42  }
```

The enemy collision type is detected on lines 3–7 of the code snippet listed in the preceding. Following the same pattern as the previously reviewed collision types, the same process is performed for the player and enemy collision: the first block of code on lines 14–25 and 29–40. In either case, the player takes damage, and that damage is registered in the HUD/UI on lines 16–19 and lines 31–34. If the damaged player is not already bouncing, then the player is bounced.

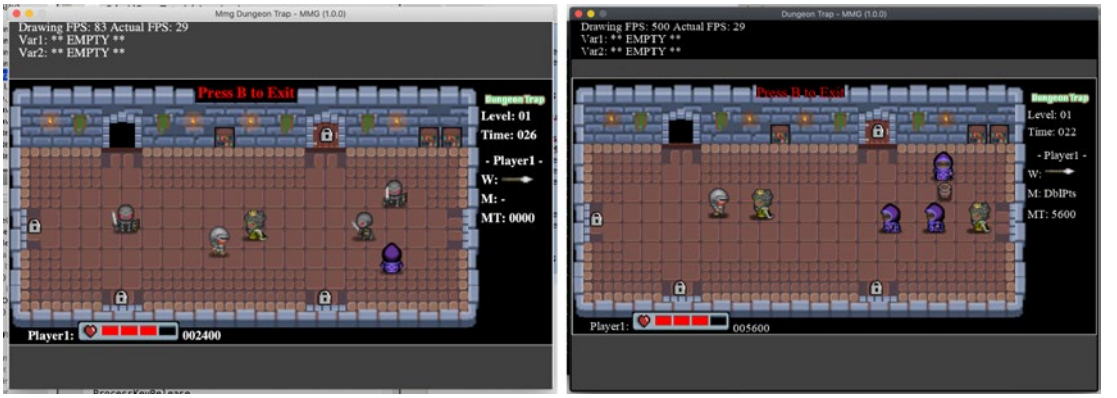
That brings us to the conclusion of the game logic method review. We've now added all the remaining functionality needed to the `ScreenGame` class to complete the game. If you've been following along typing up code by hand, then you'll need to make the adjustment to the last few lines of the `LoadResources` method as noted in the following.

Listing 24-27. ScreenGame Phase 3 Demo 1

```
01 SetState(State.SHOW_COUNT_DOWN);
02 ready = true;
03 pause = false;
```

This change forces the game to run through the `SHOW_COUNT_DOWN` state, which in turns prepares the level and starts the game. You can grab a completed copy of the work we've just done in this section by opening up the game engine project's main folder. Locate the "cfg/asset_src/dungeon_trap_chapter24_phase3" folder. Copy the contents of the folder and paste them into your game project. Make sure to adjust the `package/namespace` values of any newly pasted files to match your project's setup.

If you're following along in C#, be sure to compile your project. Address any issues you have in your game by looking at the completed chapter code in the game engine's project folder in the Chapter25_Phase3_CompleteGame directory. Fire up the game and check it out!



this figure will be printed in b/w

Take a moment to play the game. Your game. You can send barrels and tables flying into enemies. Slash your way through hordes of enemies trying to flood the dungeon chamber. Survive for as long as you can and gain as many points as you can while doing it! You're not done just yet. Check out Chapter 19, if you haven't already, for information and ideas on what you can create next.