



SwinGame: Battleships

Software Development Tutorial

In this tutorial you will be guided through the process of completing the development of a **Battleships** game.

To build Battleships we will use Visual Basic .NET and the SwinGame SDK. Visual Basic is a computer programming language that can be used to create a variety of programs, including games. The SwinGame SDK is a games development kit that was made by the **Professional Software Development** group at Swinburne University. You can find out more about it at <http://www.swingame.com>.

For this tutorial you will begin with a partly working game and add some missing features.

Finished Game

Below is a picture of how the game will look after we have finished. The game has background music, animations, artificial intelligence, high scores, and sound effects. You and the AI take it in turns to attack each others sea grids. The winner is the one who is able to locate and bomb all of their opponent's ships first.



Getting Started

Before we can start creating the game we need to make sure that we have got everything working correctly. Follow these steps to get a "Hello World" program working with SwinGame.

1. Open Visual Studio, and click File > New > Project.
2. Select the Visual Basic/Basic Language and Click the SwinGame VB.NET Project Template. Click OK.
3. Select File - Save. Enter "HelloWorld" as the file name, and choose the location it is to be saved and click OK.

In the Solution Explorer on the right hand side of the screen double click on the **GameLogic.vb** file to open it. You should be presented with the following code:

```
Module GameLogic
```

```
Public Sub Main()
```

```
    'Opens a new Graphics Window
    Core.OpenGraphicsWindow("Game", 800, 600)
```

```
    'Open Audio Device
    Audio.OpenAudio()
```

```
    'Load Resources
    LoadResources()
```

```
    'Game Loop
    Do
```

```
        'Clears the Screen to Black
        SwinGame.Graphics.ClearScreen()
```

```
        Graphics.FillRectangle(Color.Red, 20, 200, 200, 100)
        Graphics.FillRectangle(Color.Green, 220, 200, 200, 100)
        Graphics.FillRectangle(Color.Blue, 420, 200, 200, 100)
```

```
        Text.DrawText("Hello World", Color.Red, GameFont("Courier"), 20, 310)
        Text.DrawText("Hello World", Color.Green, GameFont("Courier"), 220, 310)
        Text.DrawText("Hello World", Color.Blue, GameFont("Courier"), 420, 310)
```

```
        Text.DrawFramerate(0, 0, GameFont("Courier"))
        Text.DrawText("Hello World", Color.White, GameFont("ArialLarge"), 50, 50)
```

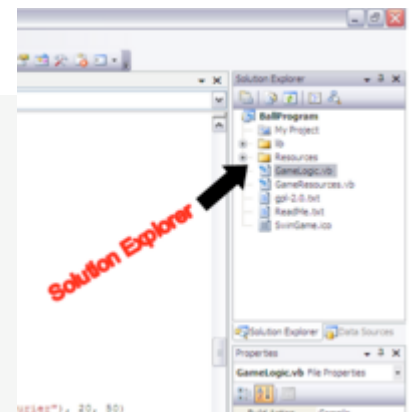
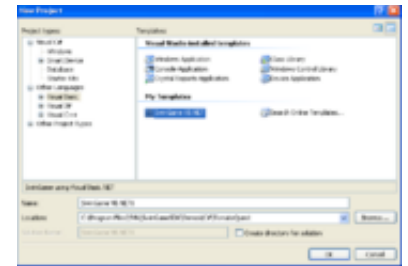
```
    'Refreshes the Screen and Processes Input Events
    Core.RefreshScreen()
    Core.ProcessEvents()
```


```
Loop Until SwinGame.Core.WindowCloseRequested() = True
```

```
    'Free Resources and Close Audio, to end the program.
    FreeResources()
    Audio.CloseAudio()
```

```
End Sub
```

```
End Module
```



The purpose of this code is to create a starting point for your project. If you press the "Start Debugging" button at the top of the screen (looks like a green arrow , F5 works too) you will see what it does. Have a look, then close the window.

You can basically read what it is doing

'comments are in green with a single quote at the start like this sentence

See if you can find the parts of the code that are doing the following things:

1. Loading all the resources you need for the game
2. Opening up a blank screen 800 pixels wide (x axis) by 600 tall (y axis).
3. Start a Loop, in which it:
 - a. Draws 3 filled rectangles setting their sizes and different colours width height and position
 - b. Putting the game's "Frame rate" up on the screen in the top right hand corner
 - c. Writing the words "Hello World" 3 times in different colours below the rectangles
 - d. Writing the words "Hello World" once at the top in large letters
4. Telling the loop to stop when someone closes the window

To help you familiarise yourself with the SwinGame library do the following:

1. Change the big "Hello World" text to your name
2. Move the Red Rectangle so it is in the bottom right hand corner
3. Make the Red Rectangle twice as big

You have completed your "Hello World" program, now we can start work on the Battleships game. Close Visual Studio and then continue the tutorial.

Guide to the Tutorial

Before you start reading this tutorial there are a few things that you can use to make these tasks easier.

What to Change

In the tutorial there will be snippets of code. In many cases these code blocks contain both code you need to change and code that already exists and doesn't need changing. The different sections of code are highlighted with different colours as shown below. The light grey colour indicates code you don't need to change, the darker grey is the code you need to change or add.

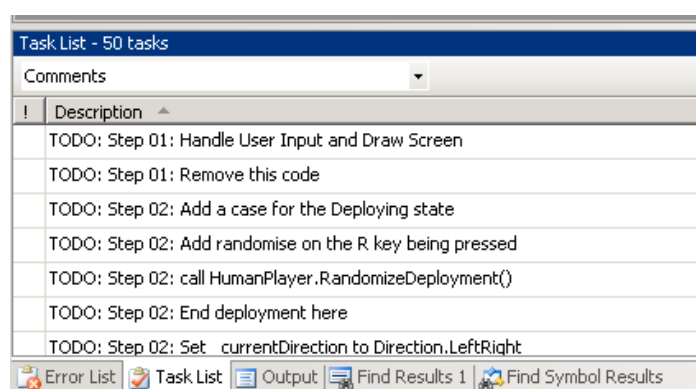
```
This is code that has been included as a guide, don't change this
This is the code you needed to add or change.
This is more code that is here as a guide, don't change it.
```

The code for the tutorial contains markers to help you locate the changes that you need to make. These are marked in the code using a *TODO:* comment as shown in the following code. Use these *TODO* comments to locate the place in the code where the changes should be made. You can replace these markers with the changed code. The following example shows the comments for Step 1.

```
'Game Loop
Do
    'TODO: Step 1: Handle User Input and Draw Screen
    Core.ProcessEvents() 'TODO: Step 1: Remove this code
Loop Until SwinGame.Core.WindowCloseRequested() = True Or _
    CurrentState = GameState.Quitting
```

Locating Markers

You can find all of these comments using Visual Studio's task list. To show the task list click the **View** menu and select **Other Windows** and then **Task List**. When the list appears it will be showing the **User Tasks**, which will be empty. Change this to view the **Comments** and you should see all of the tasks you need to complete the tutorial, as shown below. You can now double click the tasks to take you to that line in the code. Please be aware that while the comments are in step order they may not be in the same order as in the tutorial, so make sure you have the right comment before making the changes.



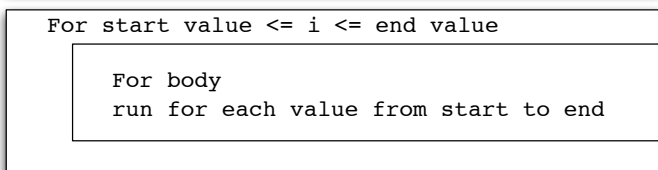
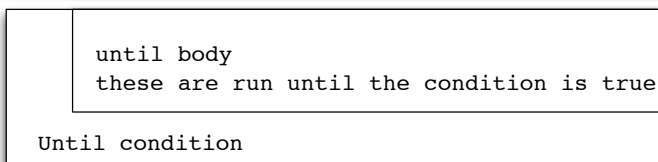
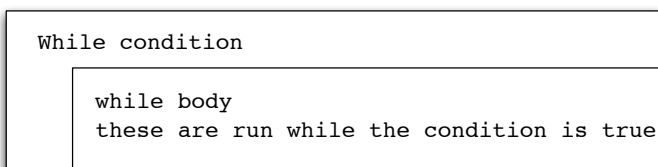
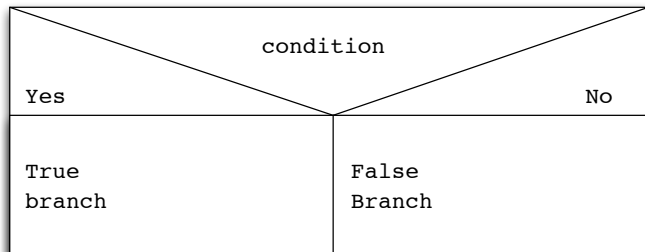
For many of the exercises in the tutorial you will be given either pseudo-code, flowcharts, or Nassi-Schneiderman diagrams to convert into code. If you get stuck with any of these there are solutions at the end of the tutorial that will show you the code you need to write for each of these. Try to work it out for yourself before checking the solution, but do use them to check that you are on the right path.

Diagrams in the Tutorial

This tutorial uses two kinds of diagrams to communicate the different algorithms.

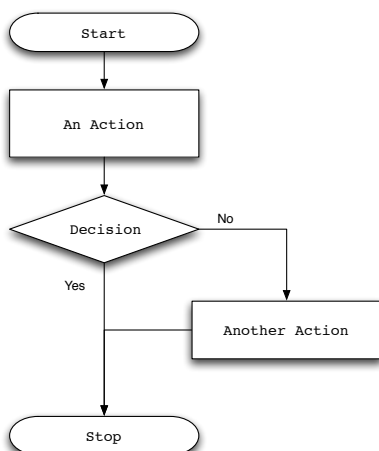
Nassi-Schneiderman Diagrams

The Nassi-Schneiderman diagrams can be used to visually represent the actions that occur in an algorithm. In this tutorial there will be four basic variations that we use. These map to **if else**, **while**, **until**, and **for** statements in Visual Basic. The four variations on these are shown below in order.



Flowcharts

Flowcharts show the sequence of actions that occur within an algorithm. The flowchart has four basic symbols, a rounded rectangle to mark the start and end of the algorithm, rectangles for actions, diamonds for decisions, and arrows to indicate the sequence.



Step 1: Getting the Game Started

Accompanying this tutorial document there will be a Battleships folder containing a Visual Studio Solution file named **battleships.sln**. Double click this file to open the solution in Visual Studio. Now lets make some changes to get the game started.

In this first step you will add the code needed to define the main steps of the game. Games are programs that usually involve a central controlling loop that repeatedly performs a number of actions. In this case the game will perform two actions: handling user input and drawing the screen. These actions are coded in the **GameLogic.vb** file. This file contains the main game loop that controls the basic actions that the game performs.

1. Run the program using the play button or by pressing F5. You should see the start up screen and then... nothing. This is the starting point for this game. Close the game and then continue.
2. Open the **GameLogic.vb** file.
3. Locate the **Main()** procedure.
4. Locate the 'Game Loop' comment and change the loop that follows it to match the following code. Remember you only need to change the highlighted code. In this case you need to do the following. Once you have done this the code should be exactly the same as the code that is shown below.
 - Delete the 'TODO: Step 1' comment
 - Delete the *Core.ProcessEvents()* call
 - Add in the two new lines, the ones highlighted below

```
'Game Loop
Do
    HandleUserInput()
    DrawScreen()
Loop Until SwinGame.Core.WindowCloseRequested() = True Or _
    CurrentState = GameState.Quitting
```

5. Run the game and you should see the main menu of the game. If not go back to the game loop and check that it is the same as above.
6. If you click the *Play* button on the menu you will see... a totally blank screen. If you still see the main menu return back to the game loop and make sure that it exactly the same as above.
7. To exit press the *Escape* key and then click the *Quit* button on the game menu.
8. Run the game again and try some of the other menu options.



Step 2: Getting the Ships Deployed

The Battleships game is divided into a number of controllers, each of which control one aspect of the game. The controllers will be responsible for handling user input, mapping the users actions to actions in the game, and for drawing the screen to show the user the current state of the game. The code in the **GameController.vb** file is responsible for controlling the overall game, taking requests from the *GameLogic.vb* code we looked at before and passing this on to the other controllers. The **DeploymentController.vb** code is one of the other controllers that the *GameController.vb* code uses, it is responsible for controlling the deployment actions for the game.

In this step you will need to change code in two locations: in the game controller and in the deployment controller. At the moment the *GameController.vb* is not telling the *DeploymentController* to draw the deployment when the game is in the deploying stage. In the last step you got the *GameLogic* to call *DrawScreen*, now you will get *DrawScreen* to call *DrawDeployment* when the game is deploying.

Once the drawing is working you will need to alter the code that handles user input so that the game responds to user interactions in the deployment step. This code is also in the *DeploymentController*. You will add in all of the actions for the buttons on the deployment screen.

1. Open the **GameController.vb** file.
2. Locate the **DrawScreen()** procedure.
3. Add the following two lines of code. These add a new case for the *Deploying* state, getting the game to draw using the *DrawDeployment* procedure.

```
Select Case CurrentState
    Case GameState.Deploying
        DrawDeployment()
    Case GameState.ViewingMainMenu
        DrawMainMenu()
```

4. Open the **DeploymentController.vb** file.
5. Locate the **DrawDeployment()** procedure.
6. Read this code and work out what it is going to draw to the screen. On the following page roughly draw what you expect to see.
7. Run the game and click the **Play** button to see the deployment screen. Does it look the same as you expected?
8. Try clicking the text buttons at the top of the large field. They don't currently do anything, so the next task will be configuring these to perform some actions.
9. Locate the **HandleDeploymentInput()** procedure.
10. Locate the **if** and three **elseif** statements at the bottom of this procedure.
11. Alter them so that they perform the following actions (don't change the *IsMouseInRectangle(...)* code, this has been edited to fit better in this document).

```
If HumanPlayer.ReadyToDeploy And IsMouseInRectangle(PLAY...) Then
    EndDeployment()
ElseIf IsMouseInRectangle(UP_DOWN...) Then
    _currentDirection = Direction.UpDown
ElseIf IsMouseInRectangle(LEFT_RIGHT...) Then
    _currentDirection = Direction.LeftRight
ElseIf IsMouseInRectangle(RANDOM...) Then
    HumanPlayer.RandomizeDeployment()
End If
```

12. Run the program and check that these four buttons now work.

13. Return to the **HandleDeploymentInput()** procedure.
14. Locate the code at the start of this procedure that processes keyboard input. There are three if statements that allow the user to press *Escape* to view the game menu, and to press the arrow keys to change the way a ship will be placed.
15. Add the following code so that pressing the *R* key will randomise the deployment.

```
If Input.WasKeyTyped(Keys.VK_LEFT) Or ... Then
    _currentDirection = Direction.LeftRight
End If

If Input.WasKeyTyped(Keys.VK_R) Then
    HumanPlayer.RandomizeDeployment()
End If

If Input.MouseWasClicked(MouseButton.LeftButton) Then
```
16. Run the game and try these keyboard shortcuts.

Draw the Deployment Screen here as part of point 6 above:

Step 3: Discovering the Enemy

In the last step you saw how the main parts of the game are coordinated and you implemented some of the logic from the *DeploymentController*. In this step you will look at the *DiscoveryController* and implement its drawing and user input. All of these actions are in the **DiscoveryController.vb** file.

The drawing of the discovery involves drawing two *fields*, showing the two sea grids, one small field showing the players ships and a larger field showing the enemies sea but hiding their ships. There are procedures that draw these fields in the **UtilityFunctions.vb** file, so in the *DiscoveryController* you can call these procedures to get them to draw the required fields. The other thing drawn is the messages that are generated when the player makes a shot, this is drawn using the *DrawMessage()* procedure.

The *DiscoveryController* only needs to handle switching to the game menu using the Escape key, and attacking the enemies field. You need to add the code that takes the user's input and maps it to a row and column that they want to attack. With this done the game will be fully functional, but still missing some features.

1. Open the **DiscoveryController.vb** file.
2. Locate the **DrawDiscovery()** procedure.
3. Add the code to the *DrawDiscovery()* procedure so that it draws the large field, small field, and message. The three lines of code that you need to add are shown below.

```
Public Sub DrawDiscovery()
    Const SCORES_LEFT As Integer = 172
    Const SHOTS_TOP As Integer = 157
    Const HITS_TOP As Integer = 206
    Const SPLASH_TOP As Integer = 256

    DrawField(HumanPlayer.EnemyGrid, ComputerPlayer, False)
    DrawSmallField(HumanPlayer.PlayerGrid, HumanPlayer)
    DrawMessage()

    'TODO: Step 7: Add score drawing code to this location
End Sub
```

4. Run the game and click the **Play** button.
5. On the deployment screen click the **Play** button again to enter the discovery/battle part of the game. Once you have seen the battle fields drawn correctly close the game and return to Visual Studio.
6. Locate the **HandleDiscoveryInput()** procedure.
7. Add the missing three lines of code highlighted below.

```
Public Sub HandleDiscoveryInput()
    If Input.WasKeyTyped(Keys.VK_ESCAPE) Then
        AddNewState(GameState.ViewingGameMenu)
    End If

    If Input.MouseWasClicked(MouseButton.LeftButton) Then
        DoAttack()
    End If
End Sub
```

8. Locate the **DoAttack()** procedure.
9. This procedure needs to be modified so that it reads the location of the mouse and maps it to a position in the grid. It can then use this position to tell the game to attack that location. Use the following code to perform this attack.

```
Dim mouse As Point2D
mouse = Input.GetMousePosition()

'Calculate the row/col clicked
Dim row, col As Integer
row = Convert.ToInt32(Math.Floor((mouse.Y - FIELD_TOP) / (CELL_HEIGHT
+ CELL_GAP)))
col = Convert.ToInt32(Math.Floor((mouse.X - FIELD_LEFT) / (CELL_WIDTH
+ CELL_GAP)))

If row >= 0 And row < HumanPlayer.EnemyGrid.Height Then
    If col >= 0 And col < HumanPlayer.EnemyGrid.Width Then
        Attack(row, col)
    End If
End If
```

10. See if you can follow the actions that are performed when an attack occurs. Start by right clicking on the *Attack(...)* call and selecting **Go To Definition**. This will take you to the *Attack(...)* procedure in the *GameController.vb* file. This then calls *_theGame.Shoot(..)*, right click on the *Shoot(...)* call and select *Go To Definition* to see what that does. Follow this through the *BattleshipsGame Shoot(...)* to *Player's Shoot(...)* finally to *SeaGrid's HitTile(...)*.
11. Draw a flowchart of the code in *HitTile(...)* on the following page. What are the different results you can get from attacking a tile?
12. Run the game and you should be able to take on the AI player. Try to get all of the different types of results you identified from *HitTile(...)*.

Draw a flowchart of the *HitTile* procedure from *SeaGrid*.

Step 4: Setting the Difficulty

With the basic interactions of the game working the next step is to allow the user to switch from the default difficulty level. The *MenuController* is another one of the controllers used by the *GameController*. This controller is responsible for drawing the menus and handling the user input when these menus are shown. In this step you will implement the drawing of the settings menu and the user interactions that can occur when this menu is shown.

1. Open the **MenuController.vb** file.
2. Locate the **DrawSettings()** procedure.
3. Implement *DrawSettings()* using the following pseudo-code. Pseudo-code is a means of expressing an algorithm¹ using structured English. The idea of pseudo-code is to communicate the actions that need to be performed without having to worry about the exact syntax used to express them. See if you can convert this pseudo-code to Visual Basic code, but if you get stuck the solutions are at the end of the tutorial.

```
Call DrawButtons with MAIN_MENU
Call DrawButtons with SETUP_MENU, 1, 1, and Int(AISetting)
```

4. Locate the **PerformSetupMenuAction(...)** procedure.
5. Implement the *PerformSetupMenuAction(...)* procedure using the following pseudo-code.

```
Select Case Button
  When Button is SETUP_MENU_EASY_BUTTON
    Call SetDifficulty with AIOption.Easy
  When Button is SETUP_MENU_MEDIUM_BUTTON
    Call SetDifficulty with AIOption.Medium
  When Button is SETUP_MENU_HARD_BUTTON
    Call SetDifficulty with AIOption.Hard
End Select

EndCurrentState()
```

6. Locate the **HandleSetupMenuInput()** procedure.

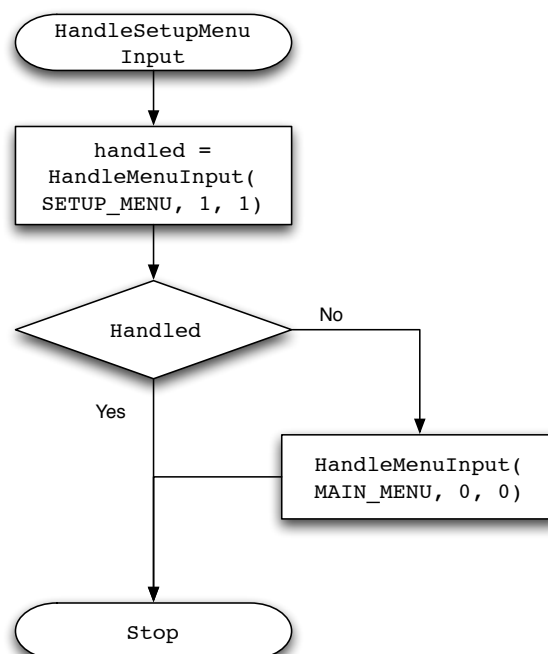
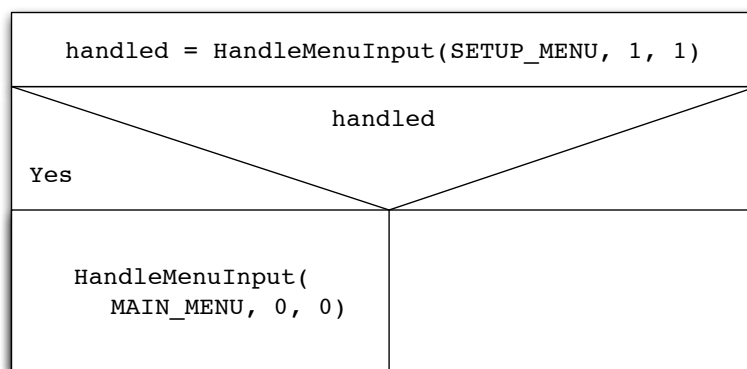
¹ An algorithm is a set of steps or actions. All computer programs are algorithms.

7. Below you will find three representations of an algorithm, one expressed as pseudo code, one as a Nassi-Schneiderman diagram, and one as a flowchart. Two of these have the same logic, and one contains a small mistake. Identify which two have the same logic and use that to implement the *HandleSetupMenuInput()* procedure. What would have occurred if you used the incorrect logic?

This procedure is responsible for getting the *MenuController* to check which buttons of the Setup Menu have been clicked, and if none were clicked to check if the main menu was clicked. The Call to *HandleMenuInput(...)* tells the *MenuController* which menu to check and its offsets. The main menu is not offset but the Setup Menu is offset one position vertically and one position horizontally.

```
Declare handled as a Boolean
Set handled to the result of calling HandleMenuInput with SETUP_MENU,
1, and 1
```

```
If Not handled Then
    HandleMenuInput(MAIN_MENU, 0, 0)
End If
```



8. Run the game and check that you can switch between the different difficulty settings.

Step 5: Adding Sounds

In this step you will add the sound effects for the game. These sounds will be played by the *GameController* and the *DeploymentController* in response to certain things that happen in the game or in response to certain actions the user performs. In order to use sounds we need to load them into the game. This is done in the **GameResources.vb** file. You will need to load the sounds in there so that you can use them in other locations in the game.

1. Copy the sound files from the resources that come with the tutorial into the Resources\sounds folder in the *Solution Explorer*. This can be done by dragging the files from the file explorer and dropping them on the Resources\sounds folder you can see in the *Solution Explorer*.
2. Open the **GameResources.vb** file.
3. Locate the **LoadSounds()** procedure. This is used while the game loads to load in the sounds.
4. Add the following code to load all of the sound effects.

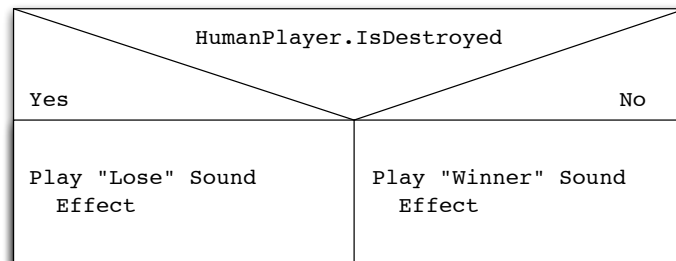
```
Private Sub LoadSounds()  
    NewSound("Error", "error.wav")  
    NewSound("Hit", "hit.wav")  
    NewSound("Sink", "sink.wav")  
    NewSound("Miss", "watershot.wav")  
    NewSound("Winner", "winner.wav")  
    NewSound("Lose", "lose.wav")  
    NewSound("Deploy", "deploy.wav")  
End Sub
```

5. First lets add sound to the deployment. Open the **DeploymentController.vb** file.
6. Locate the **DoDeploymentClick()** procedure.
7. Locate the following code and add in the call to play the deploy and error sounds.

```
Try  
    HumanPlayer.PlayerGrid.MoveShip(row, col, _selectedShip, _  
        _currentDirection)  
    Audio.PlaySoundEffect(GameSound("Deploy"))  
    Message = "Ship deployed"  
Catch ex As Exception  
    Audio.PlaySoundEffect(GameSound("Error"))  
    Message = ex.Message  
End Try
```

8. Compile and run the program.
9. Try to move a ship so that it is off the edge of the screen or overlapping another ship. You should hear the error sound in these cases. Also try moving a ship to another location to hear the deployment sound.
10. Close the program and return to Visual Studio.
11. Now lets add sounds to the attack results. This is done by the GameController. Open the **GameController.vb** file.
12. Locate the **PlayHitSequence(...)**.
13. Add in a call to play the "Hit" sound effect.

14. Perform the same steps with the **PlayMissSequence(...)**, but this time play the "Miss" sound effect
15. Locate the **PlaySinkSequence(...)**.
16. Add the code to play the "Sink" sound effect in the *PlaySinkSequence(...)* procedure.
17. Locate the **PlayGameOverSounds()** procedure.
18. Use the following Nassi-Schneiderman diagram for the logic of *PlayGameOverSounds()*

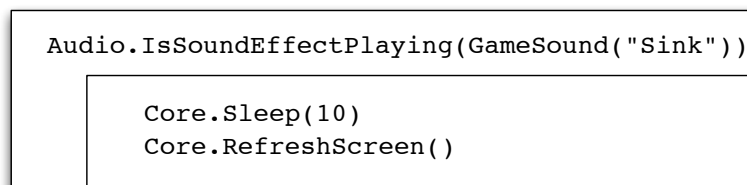


19. Locate the **AttackCompleted(...)** procedure.
20. At the bottom of the procedure locate the *Case ResultOfAttack.ShotAlready* code and add code to play the error sound effect. The code should appear as shown below.

```

Case ResultOfAttack.ShotAlready
  Audio.PlaySoundEffect(GameSound("Error"))
End Select
  
```

21. Locate the *Case ResultOfAttack.GameOver* code in **PlayGameOverSounds(...)** and add the code that implements the following Nassi-Schneiderman diagram. This code causes the game to delay until the "Sink" sound has finished.



22. Run the game to ensure everything is working at this stage.

Step 6: Adding High Scores

You need to add the code to load the high scores from file, and to add in the player's high scores at the end of each game. The high scores are stored in a text file called *highscores.txt* this file has the following format:

The first line has the number of high scores on it

Each line after the first line has three characters for the player's name followed by their score

Open the **highscores.txt** file from the Resources folder. This is the starting high score file.

1. Open the **HighScoreController.vb** file.
2. Locate the **LoadScores()** procedure
3. Someone designed the following algorithm for this:

```
Call Clear() on the _Scores List

Set filename to Core.GetPathToResource("highscores.txt")
Set input to a New StreamReader(filename)

Set numScores = Convert.ToInt32(input.ReadLine())

For i = 0 to numScore
    Declare s as a Score
    Declare line as a String

    line = input.ReadLine()

    Set s.Name to line.Substring(0, NAME_WIDTH)
    Set s.Value to Convert.ToInt32(line.Substring(NAME_WIDTH))
    Add s to the _Scores List
Next

Call Close() on input
```

4. When this code was implemented the program crashed after reading in the last score. Locate the source of the problem and implement the corrected version in *LoadScores()*.

Hint: It appeared that the program tried to load one to many scores from the file. Refer to the solutions if you ca not locate the error.

5. Open the **EndingGameController.vb** file.
6. Add the following line of code above the *EndCurrentState()* call. This changes the actions at the end of the game, enabling the user to enter in their name if they got a high score.

```
ReadHighScore(HumanPlayer.Score)
```

7. Right click on *ReadHighScore(...)* and select **Go To Definition**. What do you think the purpose of the second if statement is? (If value > _Scores.Item(_Scores.Count - 1).Value Then)
8. Run the game and check that you can see the high scores... see if you can get a high score and that it is saved to the file.

Step 7: Drawing Backgrounds

The next step is to start using the images to draw more enticing backgrounds for the game. The *GameController* uses the **UtilityFunctions** code to draw the background for all stages of the game. To get this working you are going to need to load the images you want to use and then draw these as the background.

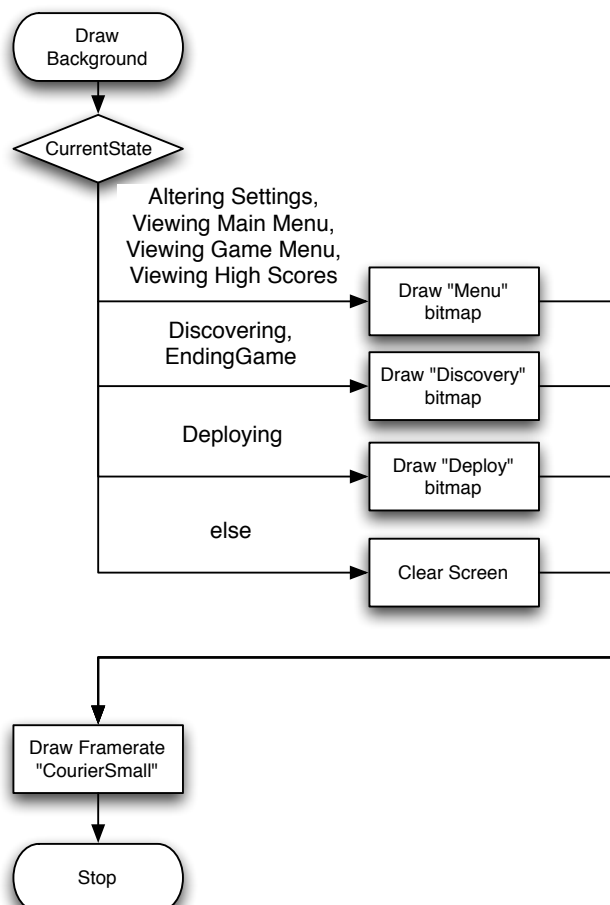
1. Copy the discover, deploy, and main_page images into the Resources/images folder in the *Solution Explorer*.
2. Open the **GameResources.vb** file.
3. Locate the **LoadImages()** procedure.
4. Locate the *'Backgrounds* comment and add the code shown below. This will load the images into the game and assign them names we can use to refer to these images in the code.

```
'Backgrounds
NewImage("Menu", "main_page.jpg")
NewImage("Discovery", "discover.jpg")
NewImage("Deploy", "deploy.jpg")
```

5. Open the **UtilityFunctions.vb** file.
6. Locate the **DrawBackground()** procedure.
7. Alter the procedure so that it matches the logic shown in the following flowchart.

Hint: use a **select case** statement to create these branches.

Hint: draw bitmaps using `Graphics.DrawBitmap(GameImage("Menu"), 0, 0)` for example.



8. Locate the **DrawCustomField(...)** procedure.
9. Remove the following line of code. This code draws a separate background behind the field. This was needed when the old background was just a black window, but now the image includes a pre-drawn background for this purpose so the following line is no longer needed.

```
Graphics.FillRectangle(Color.Blue, left, top, width, height)
```

10. Also remove the following line of code. This drew a border around each of the tiles in the fields. Once again, these are now part of the background picture.

```
Graphics.DrawRectangle(OUTLINE_COLOR, colLeft, rowTop, cellWidth, _  
    cellHeight)
```

11. Run the program and check that the backgrounds are drawn correctly.

Step 8: Drawing Buttons

In this step you will change from drawing line based buttons to using the button images supplied for the deployment phase. The buttons are pictured below. These are buttons for randomising the fleet, placing the ship vertically or horizontally, starting the game, and for marking the selected ship.



1. Add the button images to the Resources/images folder in the *Solution Explorer*, there are five images, the names are in the following code.
2. Open the **GameResources.vb** file.
3. Locate the **LoadImages()** procedure.
4. Locate the *'Deployment* comment and add the following code to load the images.

```
'Deployment
NewImage("LeftRightButton", "deploy_dir_button_horiz.png")
NewImage("UpDownButton", "deploy_dir_button_vert.png")
NewImage("SelectedShip", "deploy_button_hl.png")
NewImage("PlayButton", "deploy_play_button.png")
NewImage("RandomButton", "deploy_randomize_button.png")
```

5. Open the **DeploymentController.vb** file.
6. Locate the **DrawDeployment()** procedure.
7. Examine the flowchart on the following page, and verify that it matches the code in *DrawDeployment()*
8. The flowchart has some highlighted boxes, these mark the locations that need to be changed. The code that matched the dashed boxes needs to be removed, while the other code changed to use the following which now draws these images. When done run the program and check that it all works.

To draw the random button:

```
Graphics.DrawBitmap(GameImage("RandomButton"), RANDOM_BUTTON_LEFT, _
    TOP_BUTTONS_TOP)
```

To draw the Left/Right button as selected:

```
Graphics.DrawBitmap(GameImage("LeftRightButton"), DIR_BUTTONS_LEFT, _
    TOP_BUTTONS_TOP)
```

To draw the Up/Down button as selected:

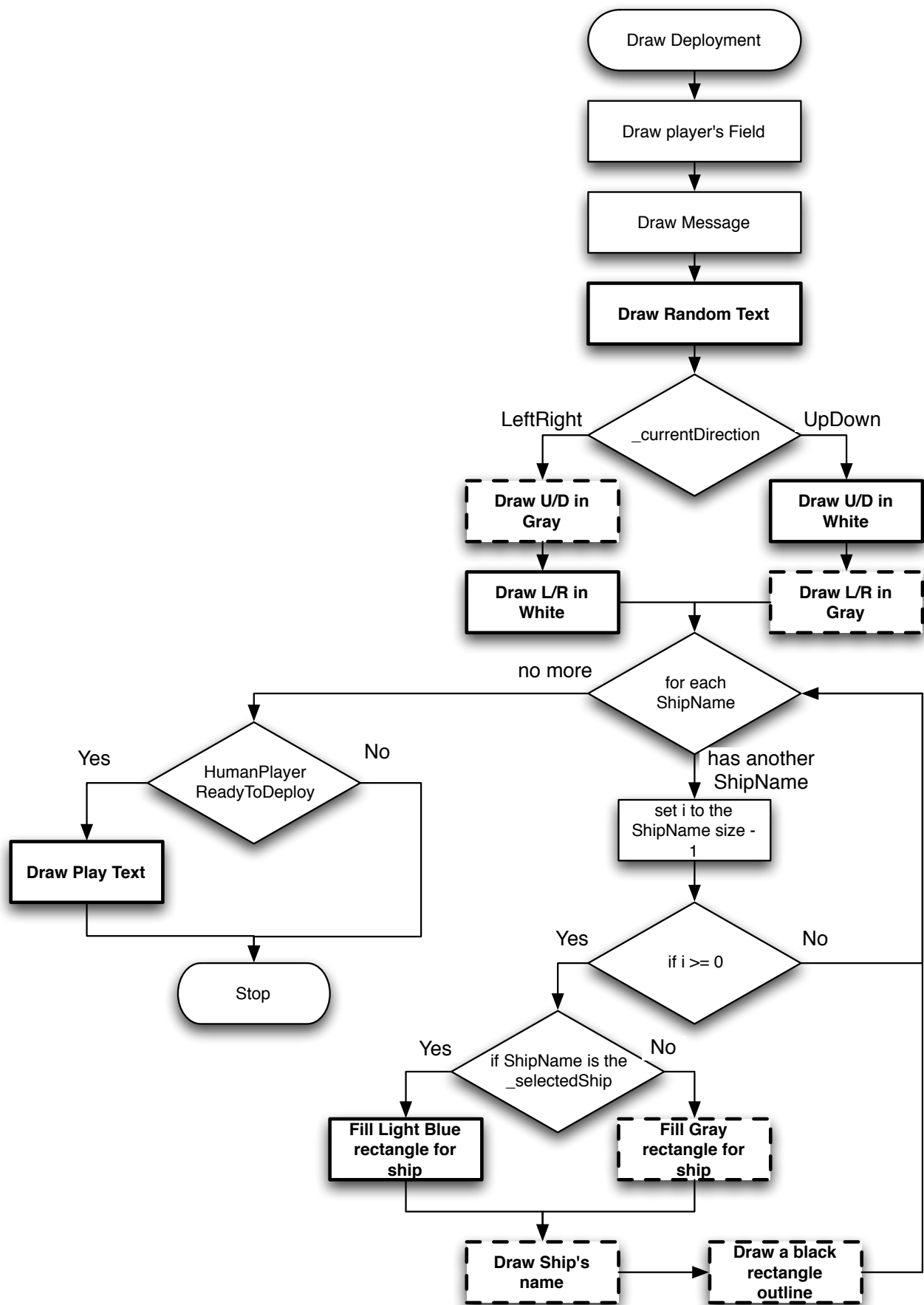
```
Graphics.DrawBitmap(GameImage("UpDownButton"), DIR_BUTTONS_LEFT, _
    TOP_BUTTONS_TOP)
```

To draw the Play button:

```
Graphics.DrawBitmap(GameImage("PlayButton"), PLAY_BUTTON_LEFT, _
    TOP_BUTTONS_TOP)
```

To draw the selected ship:

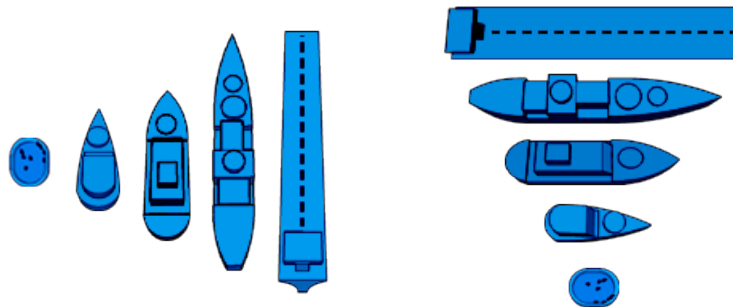
```
Graphics.DrawBitmap(GameImage("SelectedShip"), SHIPS_LEFT, _
    SHIPS_TOP + i * SHIPS_HEIGHT)
```



Flowchart for DrawDeployment

Step 9: Drawing Ships

In this step you will change the drawing of the playing field to use a number of ship images. There are ten different ship images, two for each ship, one up/down and one left/right. The images are shown below.



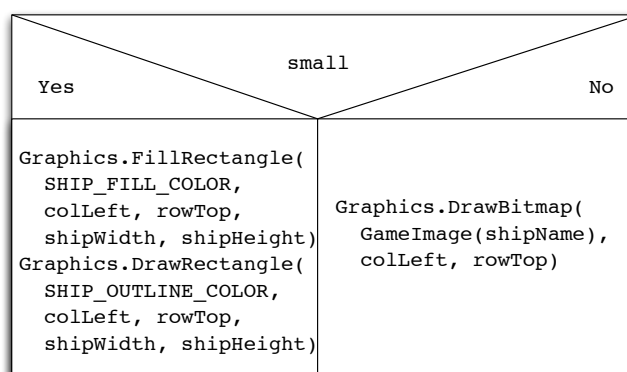
Notice that there are two variations of the filenames, and they differ only by the size indicator in each. This will help make it easier for us to load these into the game.

The drawing of the ships themselves is part of the **UtilityFunctions** code which contains a **DrawShips()** procedure. This one procedure is used to draw the ships on the large and small fields. We need to draw the ship images on the large field, but keep drawing the rectangles on the small field... so there is some logic here.

1. Add all ten images named "ship_deploy_*.png" to the Resources/images folder in the *Solution Explorer*.
2. Open the **GameResources.vb** file.
3. Locate the **LoadImages()** procedure
4. Insert the code to match the following Nassi-Schneiderman diagram after the 'Ships' comment. This will load all ten file images, giving them names like "ShipLR1" and "ShipUD5".

```
For 1 <= i <= 5
    NewImage("ShipLR" & i, _
        "ship_deploy_horiz_" & i & ".png")
    NewImage("ShipUD" & i,
        "ship_deploy_vert_" & i & ".png")
```

5. Open the **UtilityFunctions.vb** file.
6. Locate the **DrawShips(...)** procedure.
7. At the end of *DrawShips(...)* locate the code the two lines of code that fill and then draw a rectangle.
8. Alter this to match the logic in the following Nassi-Schneiderman diagram. Notice that it checks if this is the large or small field, and only draws the images on the large field.



Step 10: Adding Theme Music

Now lets add some background music that will be played throughout the games execution. This will help set the scene for the player and provide a more engaging experience.

1. Copy the **horrrordrone.mp3** file into the projects Resources\sounds folder in the *Solution Explorer*. This file contains the music that will be played.
2. Find the **LoadMusic()** procedure in the **GameResources.vb** file.
3. In the LoadMusic() procedure load the music using the NewMusic function, give it the name "Background". The code for this is shown below.

```
Private Sub LoadMusic()  
    NewMusic("Background", "horrrordrone.mp3")  
End Sub
```

4. Find the **Main()** procedure in the **GameLogic.vb** file.
5. Now we will use the **Audio** capabilities of SwinGame to **PlayMusic(...)** after the call to *LoadResources()* and to **StopMusic()** before the call to *FreeResources()*. As I am sure you have already guessed, this will start the music playing when the game starts and end the music at the end of the game just before the resources such as the music itself are closed.
6. See the code for this below.
7. Run the game and check that the music is working as expected.

```
Public Sub Main()  
    'Opens a new Graphics Window  
    Core.OpenGraphicsWindow("Battle Ships", 800, 600)  
  
    'Open Audio Device  
    Audio.OpenAudio()  
  
    'Load Resources  
    LoadResources()  
  
    Audio.PlayMusic(GameMusic("Background"))  
  
    'Game Loop  
    Do  
        HandleUserInput()  
        DrawScreen()  
    Loop Until SwinGame.Core.WindowCloseRequested() = True Or _  
        CurrentState = GameState.Quitting  
  
    Audio.StopMusic()  
  
    'Free Resources and Close Audio, to end the program.  
    FreeResources()  
    Audio.CloseAudio()  
End Sub
```

Step 11: Adding Animations

The game is complete, but lets make it look a little more impressive by using an animation for the hits and the misses. This will involve two images, *explosion* and *splash*, each of which contains a number of smaller pictures that make up the animation. Using this image we can create a SwinGame Sprite that uses the individual smaller images to make a small animation by playing the images one at a time.

1. Copy the explosion and splash images into the Resources\images folder in the *Solution Explorer*.
2. Open **GameResources.vb** file.
3. Locate the **LoadImages()** procedure.
4. Locate the '*Explosions*' comment and alter it to match the following pseudo-code.

```
'Explosions
Call NewImage with parameters "Explosion" and "explosion.png"
Call NewImage with parameters "Splash" and "splash.png"
```

5. Open the **GameController.vb** file
6. Locate the **PlayHitSequence(...)** procedure.
7. Alter *PlayHitSequence(...)* so that its logic matches that shown in the following pseudo-code.

```
If showAnimation Then
    Call AddExplosion with row and column
End If
```

```
Use Audio to play the "Hit" sound effect
```

```
Call DrawAnimationSequence
```

8. Locate the **PlayMissSequence(...)** procedure.
9. Alter *PlayMissSequence(...)* so that its logic matches that shown in the following pseudo-code.

```
If showAnimation Then
    Call AddSplash with row and column
End If
```

```
Use Audio to play the "Miss" sound effect
```

```
Call DrawAnimationSequence
```

10. Locate the **AttackComplete(...)** procedure and remove the call to *DrawScreen()*. This is no longer needed as it draws the result of the attack before the animation plays, and we now want the animation to play first.

11. Open the **UtilityFunctions.vb** file.
12. Locate the **AddAnimation(...)** procedure.
13. Alter *AddAnimation(...)* so that its logic matches that shown in the following pseudo-code. This code creates the new SwinGame sprite and configures it as an animation that is 40 pixels wide and high, this is the size of the smaller images inside the explosion and splash files. The resulting sprite is then added to the *_animations* collection. The remaining code sets the actions that occur when the animation ends, and to set the location of the sprite.

```
Declare s as a Sprite
```

```
Set s to the result of using Graphics to CreateSprite with  
  GameImage(image), FRAMES_PER_CELL, ANIMATION_CELLS, 40, and 40 as  
  parameters.
```

```
Set s's EndingAction to SpriteEndingAction.Stop  
Set s's X to FIELD_LEFT + col * (CELL_WIDTH + CELL_GAP)  
Set s's Y to FIELD_TOP + row * (CELL_HEIGHT + CELL_GAP)
```

```
Add s to the _animations list
```

14. Locate the **DrawAnimations()** procedure. This procedure needs to be updated so that the individual animations are drawn to the screen.
15. Alter *DrawAnimations()* so that it matches the logic in the following pseudo-code.

```
For each sprite in _Animations  
  Use Graphics to draw the sprite  
Next
```

16. Run the game and check that you get the splash and explosion animations.

Beyond Step 11: Enhancements

The following are suggestions that you may want to make to improve the game.

1. In the **DiscoveryController.vb** file add a *cheat* so that if the player holds down the Shift and the C key that the enemy's ships are drawn. This will make it easier for testing...

Hint: Have a look at the parameters for the *DrawField()*, and make use of *Input.IsKeyPressed(...)*

2. In the resources folder there are two extra images *you_win.png* and *you_lose.png*. Replace the current text that is drawn when the player wins or loses the game with these images.

Hint: You will need to change *GameResources.vb* as well as the *EndingGameController.vb*.

3. Add extra keyboard input to allow the user to interact with the menus using the keyboard. For example the user should be able to press the *P* key to select the *Play* menu from the main menu.

Hint: Look in the *MenuController.vb* for the code that currently handles the mouse input.

4. Use an image editor to create small versions of the ship images by resizing the current ship images. Now alter the code so that the small ships are loaded and drawn with the small grid.
5. Now try making your own game :)

Solutions

The solutions show the code for any flowcharts, Nassi-Schneiderman diagrams, and pseudo-code from the steps in the tutorial.

Step 4: Setting the Difficulty

The following is the code for **DrawSettings()** in **MenuController.vb**.

```
Public Sub DrawSettings()
    DrawButtons(MAIN_MENU)
    DrawButtons(SETUP_MENU, 1, 1, Int(AISetting))
End Sub
```

The following is the code for **PerformSetupMenuAction(...)** in **MenuController.vb**.

```
Private Sub PerformSetupMenuAction(ByVal button As Integer)
    Select Case Button
        Case SETUP_MENU_EASY_BUTTON
            SetDifficulty(AIOption.Easy)
        Case SETUP_MENU_MEDIUM_BUTTON
            SetDifficulty(AIOption.Medium)
        Case SETUP_MENU_HARD_BUTTON
            SetDifficulty(AIOption.Hard)
    End Select

    'Always end state - handles exit button as well
    EndCurrentState()
End Sub
```

The following code is for the **HandleSetupMenuInput()** procedure from **MenuController.vb**.

```
Public Sub HandleSetupMenuInput()
    Dim handled As Boolean
    handled = HandleMenuInput(SETUP_MENU, 1, 1)

    If Not handled Then
        HandleMenuInput(MAIN_MENU, 0, 0)
    End If
End Sub
```

Step 5: Adding Sounds

The code to **LoadSounds()** code from **GameResources.vb**.

```
Private Sub LoadSounds()
    NewSound("Error", "error.wav")
    NewSound("Hit", "hit.wav")
    NewSound("Sink", "sink.wav")
    NewSound("Siren", "siren.wav")
    NewSound("Miss", "watershot.wav")
    NewSound("Winner", "winner.wav")
    NewSound("Lose", "lose.wav")
    NewSound("Deploy", "deploy.wav")
End Sub
```

The code to play the sound effects in **PlayHitSequence(...)**, **PlayMissSequence(...)**, and **PlaySinkSequence(...)** in **GameController.vb**.

```
Private Sub PlayHitSequence(ByVal row As Integer, _
    ByVal column As Integer, ByVal showAnimation As Boolean)
    'TODO: Step 11: Add explosion animation if showAnimation is true

    Audio.PlaySoundEffect(GameSound("Hit"))

    'TODO: Step 11: call draw animation sequence.
End Sub
```

```
Private Sub PlayMissSequence(ByVal row As Integer, _
    ByVal column As Integer, ByVal showAnimation As Boolean)
    'TODO: Step 11: Add splash animation if showAnimation is true

    Audio.PlaySoundEffect(GameSound("Miss"))

    'TODO: Step 11: call draw animation sequence.
End Sub
```

```
Private Sub PlaySinkSequence(ByVal row As Integer, _
    ByVal column As Integer, ByVal showAnimation As Boolean)
    PlayHitSequence(row, column, showAnimation)
    Audio.PlaySoundEffect(GameSound("Sink"))
End Sub
```

The following code shows the **PlayGameOverSounds()** of **GameController.vb**.

```
Private Sub PlayGameOverSounds()
    If HumanPlayer.IsDestroyed Then
        Audio.PlaySoundEffect(GameSound("Lose"))
    Else
        Audio.PlaySoundEffect(GameSound("Winner"))
    End If
End Sub
```

The following is the code for **AttackCompleted(...)** in **GameController.vb** that delays until "Sink" has ended.

```
Case ResultOfAttack.GameOver
    PlaySinkSequence(result.Row, result.Column, isHuman)

    While Audio.IsSoundEffectPlaying(GameSound("Sink"))
        Core.Sleep(10)
        Core.RefreshScreen()
    End While

    PlayGameOverSounds()
```

Step 6: Adding High Scores

The code for **LoadScores()** in **GameResources.vb** is:

```
Private Sub LoadScores()
    Dim filename As String
    Dim input As StreamReader
    Dim numScores As Integer

    _Scores.Clear()

    filename = Core.GetPathToResource("highscores.txt")
    input = New StreamReader(filename)

    numScores = Convert.ToInt32(input.ReadLine())

    For i As Integer = 1 To numScores
        Dim s As Score
        Dim line As String

        line = input.ReadLine()

        s.Name = line.Substring(0, NAME_WIDTH)
        s.Value = Convert.ToInt32(line.Substring(NAME_WIDTH))
        _Scores.Add(s)
    Next
    input.Close()
End Sub
```

In **EndingGameController.vb** the following changes need to be made:

```
Public Sub HandleEndOfGameInput()
    If Input.MouseWasClicked(MouseButton.LeftButton) _
        OrElse Input.WasKeyTyped(Keys.VK_RETURN) _
        OrElse Input.WasKeyTyped(Keys.VK_ESCAPE) Then

        ReadHighScore(HumanPlayer.Score)

        EndCurrentState()
    End If
End Sub
```

Step 7: Drawing Backgrounds

The following code is from the **LoadImages()** procedure in **GameResources.vb**.

```
Private Sub LoadImages()  
    'Backgrounds  
    NewImage("Menu", "main_page.jpg")  
    NewImage("Discovery", "discover.jpg")  
    NewImage("Deploy", "deploy.jpg")  
  
    ...  
End Sub
```

The following code is from **UtilityFunctions.vb**.

```
Public Sub DrawBackground()  
  
    Select Case CurrentState  
        Case GameState.ViewingMainMenu, GameState.ViewingGameMenu, _  
            GameState.AlteringSettings, GameState.ViewingHighScores  
            Graphics.DrawBitmap(GameImage("Menu"), 0, 0)  
        Case GameState.Discovering, GameState.EndingGame  
            Graphics.DrawBitmap(GameImage("Discovery"), 0, 0)  
        Case GameState.Deploying  
            Graphics.DrawBitmap(GameImage("Deploy"), 0, 0)  
        Case Else  
            Graphics.ClearScreen()  
    End Select  
  
    Text.DrawFramerate(675, 585, GameFont("CourierSmall"))  
End Sub
```

The following changes are from the **DrawDiscovery()** procedure in the **DiscoveryController.vb** file.

```
Public Sub DrawDiscovery()  
    ...  
    DrawMessage()  
  
    Text.DrawText(HumanPlayer.Shots.ToString(), Color.White, _  
        GameFont("Menu"), SCORES_LEFT, SHOTS_TOP)  
    Text.DrawText(HumanPlayer.Hits.ToString(), Color.White, _  
        GameFont("Menu"), SCORES_LEFT, HITS_TOP)  
    Text.DrawText(HumanPlayer.Missed.ToString(), Color.White, _  
        GameFont("Menu"), SCORES_LEFT, SPLASH_TOP)  
End Sub
```

Step 8: Drawing Buttons

The following changes are from **LoadImages()** in **GameResources.vb**.

```
'Deployment
NewImage("LeftRightButton", "deploy_dir_button_horiz.png")
NewImage("UpDownButton", "deploy_dir_button_vert.png")
NewImage("SelectedShip", "deploy_button_h1.png")
NewImage("PlayButton", "deploy_play_button.png")
NewImage("RandomButton", "deploy_randomize_button.png")
```

The following changes are from **DeploymentController.vb**.

```
Public Sub DrawDeployment()
    DrawField(HumanPlayer.PlayerGrid, HumanPlayer, True)
    DrawMessage()
    Graphics.DrawBitmap(GameImage("RandomButton"), RANDOM_BUTTON_LEFT, _
        TOP_BUTTONS_TOP)

    'Draw the Left/Right and Up/Down buttons
    If _currentDirection = Direction.LeftRight Then
        Graphics.DrawBitmap(GameImage("LeftRightButton"), _
            DIR_BUTTONS_LEFT, TOP_BUTTONS_TOP)
    Else
        Graphics.DrawBitmap(GameImage("UpDownButton"), DIR_BUTTONS_LEFT, _
            TOP_BUTTONS_TOP)
    End If

    'DrawShips to select
    For Each sn As ShipName In [Enum].GetValues(GetType(ShipName))
        Dim i As Integer
        i = Int(sn) - 1 'Get the ship's number and subtract 1

        If i >= 0 Then
            If sn = _selectedShip Then
                Graphics.DrawBitmap(GameImage("SelectedShip"), SHIPS_LEFT, _
                    SHIPS_TOP + i * SHIPS_HEIGHT)
            End If
        End If
    Next

    If HumanPlayer.ReadyToDeploy Then
        Graphics.DrawBitmap(GameImage("PlayButton"), PLAY_BUTTON_LEFT, _
            TOP_BUTTONS_TOP)
    End If
End Sub
```

Step 9: Drawing Ships

In **GameResources.vb** the following changes need to be made to **LoadImages()**:

```
'Ships
For i As Integer = 1 To 5
    NewImage("ShipLR" & i, "ship_deploy_horiz_" & i & ".png")
    NewImage("ShipUD" & i, "ship_deploy_vert_" & i & ".png")
Next
```

In the **UtilityFunctions.vb** file in **DrawShips(...)**.

```
If Not small Then
    Graphics.DrawBitmap(GameImage(shipName), colLeft, rowTop)
Else
    Graphics.FillRectangle(SHIP_FILL_COLOR, colLeft, rowTop, _
        shipWidth, shipHeight)
    Graphics.DrawRectangle(SHIP_OUTLINE_COLOR, colLeft, rowTop, _
        shipWidth, shipHeight)
End If
```


Step 11: Adding Animations

The following changes are from **LoadImages()** in **GameResources.vb**.

```
'Explosions
NewImage("Explosion", "explosion.png")
NewImage("Splash", "splash.png")
```

These changes are made in the **GameController.vb** file.

```
Private Sub PlayHitSequence(ByVal row As Integer, _
    ByVal column As Integer, ByVal showAnimation As Boolean)

    If showAnimation Then
        Call AddExplosion(row, column)
    End If

    Audio.PlaySoundEffect(GameSound("Hit"))

    DrawAnimationSequence()
End Sub
```

```
Private Sub PlayMiss(ByVal row As Integer, _
    ByVal column As Integer, ByVal showAnimation As Boolean)

    If showAnimation Then
        AddSplash(row, column)
    End If

    Audio.PlaySoundEffect(GameSound("Hit"))

    DrawAnimationSequence()
End Sub
```

In **UtilityFunctions.vb**:

```
Private Sub AddAnimation(ByVal row As Integer, _
    ByVal col As Integer, ByVal image As String)
    Dim s As Sprite

    s = Graphics.CreateSprite(GameImage(image), FRAMES_PER_CELL, _
        ANIMATION_CELLS, 40, 40)
    s.EndingAction = SpriteEndingAction.Stop
    s.X = FIELD_LEFT + col * (CELL_WIDTH + CELL_GAP)
    s.Y = FIELD_TOP + row * (CELL_HEIGHT + CELL_GAP)

    _Animations.Add(s)
End Sub

Public Sub DrawAnimations()
    For Each s As Sprite In _Animations
        Graphics.DrawSprite(s)
    Next
End Sub
```