Hands-on Lab

Multi-platform Development with the XNA Development Framework

Lab Version: 1.0.0

Last updated: 12/9/2010

Contents

[Overview 3](#_Toc278487450)

[Objectives 3](#_Toc278487451)

[Prerequisites 3](#_Toc278487452)

[Lab Structure 3](#_Toc278487453)

[Lab Contents 4](#_Toc278487454)

[Estimated completion time 4](#_Toc278487455)

[Exercise 1 - Windows Version 5](#_Toc278487456)

[Creating a copy of the Windows Phone project 5](#_Toc278487457)

[Handling keyboard input outside the gameplay screen 6](#_Toc278487458)

[Handling keyboard input in the gameplay screen 11](#_Toc278487459)

[Saving and loading high-score data – the high-score screen 25](#_Toc278487460)

[Saving and loading high-score data – other screens 33](#_Toc278487461)

[Introducing high-definition assets 39](#_Toc278487462)

[Adding scaling capabilities to all drawing operations 41](#_Toc278487463)

[Modifying the game to scale all drawing operations 42](#_Toc278487464)

[Fitting visual element positions to the target platform – the gameplay screen 45](#_Toc278487465)

[Fitting element rendering and logic to the target platform – bee component 55](#_Toc278487466)

[Fitting visual element positions to the target platform – game components 59](#_Toc278487467)

[Fitting visual element positions to the target platform – additional screens 62](#_Toc278487468)

[Exercise 2 – Xbox 360 Version 70](#_Toc278487469)

[Creating a copy of the Windows project 70](#_Toc278487470)

[Handling gamepad input 71](#_Toc278487471)

[Fitting visual element positions to the target platform 81](#_Toc278487472)

[Handling Guide Events 84](#_Toc278487473)

[Final touchups 86](#_Toc278487474)

Overview

* 1. Microsoft® XNA® Game Studio 4.0 and the Windows® Phone 7 present many new possibilities for game developers. XNA Game Studio 4.0 enables you to create games for Xbox 360®, for the Windows computer, and for Windows Phone 7. In addition, it enables you to port a game from one platform to another.
  2. This lab shows you how simple it is to port a Windows Phone 7 game to both Xbox 360 and the computer while retaining the original game’s feel and functionality. The game this lab ports is the Honeycomb Rush game. This game is part of the performance heavy lab (<http://create.msdn.com/en-US/education/catalog/lab/honeycomb_rush>).

# Objectives

* 1. This lab helps you do the following:
  + Port your game between the available platforms
  + Write code to work across all available platforms
  + Make your code aware of the display’s safe area

# Prerequisites

* 1. The following prerequisites ensure you get the most from this hands-on lab:
  + You should install Microsoft Visual Studio® 2010 or Microsoft Visual C#® Express 2010, and the Windows Phone Developer Tools.
  + You should already know how to work with the XNA Game Studio 4.0 to create a basic game project. For an example, see the Catapult Wars sample (<http://create.msdn.com/en-US/education/catalog/lab/catapult_wars>).
  + You should be familiar with the performance heavy lab. Specifically, it is important to be aware of the Honeycomb Rush game’s architecture as described in the performance heavy lab document at the beginning of “Exercise 1,” and under “Task 4 – Creating the gameplay screen.”

# Lab Structure

* 1. This lab includes two exercises. Each exercise has the following tasks:

1. Port the game to Windows.
2. Create a Windows copy of the phone project.
3. Handle keyboard input.
4. Introduce high-definition graphics assets.
5. Reposition the game elements.
6. Port the game to Xbox 360.
7. Create an Xbox 360 copy of the phone project.
8. Introduce high-definition graphics assets.
9. Reposition the game elements, using the safe area.
10. Handle gamepad input.
11. Handle Xbox 360 guide events.

# Lab Contents

* 1. The lab installation folder contains the following sub-folders:
  + Assets\Code—Contains code assets containing additional code required by this lab.
  + Assets\Media—Contains media assets that you can use to compose the visual aspects and the project sounds created in this lab.
  + Sources\Starter—Contains a Honeycomb Rush game version that is identical to the version created in the performance heavy lab, with one exception. This game version contains more code, which integrates performance measurement tools into the game. For more information about the performance measurement tools, see <http://create.msdn.com/en-US/education/catalog/utility/performance_utility>.
  + Sources\Ex1 - Windows Version—Contains the solution for exercise 1, which serves as the starter for exercise 2.
  + Sources\Ex1 - Xbox Version—Contains the solution for exercise 2.

# Estimated completion time

* 1. You should complete this lab in 90 minutes.

Exercise 1 - Windows Version

* 1. This exercise shows you how to port the game to Windows.

# Creating a copy of the Windows Phone project

* 1. The first step is to create a Windows copy of the Windows Phone 7 game project. Use the new project to compile and run the game in a Windows environment.
     1. **Note** You should create and work on a copy of the starter solution instead of on the original found in the lab installation folder. If you do work on a copy, remember to take that into account in future steps because the future steps refer to paths that relate to the starter solution.
     2. **Note** Game projects define a symbol based on the platform they target. Windows Phone 7 projects define the WINDOWS\_PHONE symbol, Windows projects define the Windows symbol, and Xbox 360 projects define the XBOX symbol.

To create the new project

1. In the lab installation folder at **Sources\Starter**, in the same directory as the supplied starter solution, create a new solution named **HoneycombRush - Windows**.
2. From the lab installation folder under **Sources\Starter\HoneycombRush\HoneycombRush,** add the **HoneycombRush** project to the new solution.
3. In **Solution Explorer**, right-click the **HoneycombRush** project, and then click **Create Copy of Project for Windows**.
4. In **Solution Explorer**, review the solution structure, and make sure that a new project named **Windows Copy of HoneycombRush** is now part of the solution.
5. Rename the project from **Windows Copy of HoneycombRush** to **HoneycombRush - Windows**.
6. Delete the **HoneycombRush** project from the solution.
7. On the **Build** menu, select **Configuration Manager**, and thenfrom the **Active solution platform** combo-box in the **Configuration Manager** dialog box, select **<Edit…>**.
8. In the list on the left, select **Mixed Platforms** and **Windows Phone,** click **Remove**, and then close all the dialog boxes.
9. From the **HoneycombRush - Windows** project’s references, remove **Microsoft.Phone,** and add a reference to **Microsoft.Xna.Framework.Net**.
10. Open the **HoneycombRush.cs** file, and delete the **LoadContent** method from the **HoneycombRush** class.

**Note** The **LoadContent** method uses isolated storage to load the high-score data. Isolated storage is not available on Windows. You need to disable it to get the game running.

1. In the same class as in the previous step, replace the constructor with the following code:
   * 1. C#
     2. public HoneycombRush()  
        {  
            // Initialize sound system  
            AudioManager.Initialize(this);              
            graphics = new GraphicsDeviceManager(this);  
            Content.RootDirectory = "Content";  
           
            // Frame rate is 30 fps by default for Windows Phone.  
            TargetElapsedTime = TimeSpan.FromTicks(333333);  
           
        #if WINDOWS\_PHONE  
            graphics.IsFullScreen = true;  
        #else  
            graphics.IsFullScreen = false;  
        #endif  
           
            // Create a new instance of the Screen Manager  
            screenManager = new ScreenManager(this);  
           
            screenManager.AddScreen(new BackgroundScreen("titleScreen"), null);  
            screenManager.AddScreen(new MainMenuScreen(), PlayerIndex.One);  
            Components.Add(screenManager);  
        }

# Handling keyboard input outside the gameplay screen

* 1. Currently, the new project uses touch input. Touch input is not a viable option in most Windows environments. Add input support for all screens other than the gameplay screen.

To add keyboard input support to the game

1. From the **ScreenManager** project folder, open the **MenuScreen.cs** file.
2. Modify the **HandleInput** method.

**Note** This new version of the method enables menus to react to keyboard input. Compare the new method version with the previous **one. This will help you** understand the exact scope of the changes made. Even though there are few differences, we provide the entire method body for easier execution of the lab.

* + 1. C#
    2. public override void HandleInput(GameTime gameTime, InputState input)  
       {  
           // we cancel the current menu screen if the user presses the back button  
           PlayerIndex player;  
           if (input.IsNewButtonPress(Buttons.Back, ControllingPlayer, out player))  
           {  
               OnCancel(player);  
           }  
          
           // Take care of Keyboard input  
           if (input.IsMenuUp(ControllingPlayer))  
           {  
               selectedEntry--;  
          
               if (selectedEntry < 0)  
                   selectedEntry = menuEntries.Count - 1;  
           }  
           else if (input.IsMenuDown(ControllingPlayer))  
           {  
               selectedEntry++;  
          
               if (selectedEntry >= menuEntries.Count)  
                   selectedEntry = 0;  
           }  
           else if (input.IsNewKeyPress(Keys.Enter, ControllingPlayer, out player) ||  
               input.IsNewKeyPress(Keys.Space, ControllingPlayer, out player))  
           {  
               OnSelectEntry(selectedEntry, player);  
           }  
           // look for any taps that occurred and select any entries that were tapped  
           foreach (GestureSample gesture in input.Gestures)  
           {  
               if (gesture.GestureType == GestureType.Tap)  
               {  
                   // convert the position to a Point that we can test against a
    3. // Rectangle  
                   Point tapLocation = new Point((int)gesture.Position.X,
    4. (int)gesture.Position.Y);  
          
                   // iterate the entries to see if any were tapped  
                   for (int i = 0; i < menuEntries.Count; i++)  
                   {  
                       MenuEntry menuEntry = menuEntries[i];  
          
                       if (GetMenuEntryHitBounds(menuEntry).Contains(tapLocation))  
                       {  
                           OnSelectEntry(i, PlayerIndex.One);  
                       }  
                   }  
               }  
           }  
       }

1. From the **ScreenManager** project folder, open the **InputState.cs** file.
2. Locate the **InputState** class’s **IsMenuUp** property, and then modify it.
   * 1. C#
     2. public bool IsMenuUp(PlayerIndex? controllingPlayer)  
        {  
            PlayerIndex playerIndex;  
           
            return IsNewKeyPress(Keys.Up, controllingPlayer, out playerIndex) ||  
                    IsNewKeyPress(Keys.Left, controllingPlayer, out playerIndex) ||  
                    IsNewButtonPress(Buttons.DPadLeft, controllingPlayer,
     3. out playerIndex) ||  
                    IsNewButtonPress(Buttons.LeftThumbstickLeft, controllingPlayer,
     4. out playerIndex);  
        }
3. Locate the **IsMenuDown** property, and then modify it.
   * 1. C#
     2. public bool IsMenuDown(PlayerIndex? controllingPlayer)  
        {  
            PlayerIndex playerIndex;  
           
            return IsNewKeyPress(Keys.Down, controllingPlayer, out playerIndex) ||  
                    IsNewKeyPress(Keys.Right, controllingPlayer, out playerIndex) ||  
                    IsNewButtonPress(Buttons.DPadRight, controllingPlayer,
     3. out playerIndex) ||  
                    IsNewButtonPress(Buttons.LeftThumbstickRight, controllingPlayer,
     4. out playerIndex);  
        }
4. Open the **LevelOverScreen.cs** file, and add a **using** statement for **Microsoft.Xna.Framework.Input**.
5. In the same file, modify the **LevelOverScreen** class’s **HandleInput** method.

**Note This new version of the method** includes an additional block for handling keyboard input.

* + 1. C#
    2. public override void HandleInput(GameTime gameTime, InputState input)  
       {  
           if (input == null)  
           {  
               throw new ArgumentNullException("input");  
           }  
          
           PlayerIndex player;  
          
           // Return to the main menu when a tap gesture is recognized  
           if (input.Gestures.Count > 0)  
           {  
               GestureSample sample = input.Gestures[0];  
               if (sample.GestureType == GestureType.Tap)  
               {  
                   StartNewLevelOrExit(input);  
                   input.Gestures.Clear();  
               }  
           }              
           // Handle keyboard  
           else if (input.IsNewKeyPress(Keys.Enter, ControllingPlayer, out player) ||  
               input.IsNewKeyPress(Keys.Space, ControllingPlayer, out player))  
           {  
               StartNewLevelOrExit(input);  
           }  
          
           base.HandleInput(gameTime, input);  
       }

1. Open the **LoadingAndInstructionScreen.cs** file, and add a **using** statement for **Microsoft.Xna.Framework.Input**.
2. In the same file, modify the **LoadingAndInstructionScreen** class’s **HandleInput** method.
   * 1. C#
     2. public override void HandleInput(GameTime gameTime, InputState input)  
        {  
            if (!isLoading)  
            {  
                PlayerIndex player;  
           
                // Handle touch input  
                if (input.Gestures.Count > 0)  
                {  
                    if (input.Gestures[0].GestureType == GestureType.Tap)  
                    {  
                        LoadResources();  
                    }  
                }  
                else if (input.IsNewKeyPress(Keys.Escape, ControllingPlayer,
     3. out player))  
                {  
                    foreach (GameScreen screen in ScreenManager.GetScreens())  
                    {  
                        screen.ExitScreen();  
                    }  
           
                    ScreenManager.AddScreen(new BackgroundScreen("titleScreen"),
     4. null);  
                    ScreenManager.AddScreen(new MainMenuScreen(), PlayerIndex.One);  
                }  
                else if (input.IsNewKeyPress(Keys.Enter, ControllingPlayer,
     5. out player) || input.IsNewKeyPress(Keys.Space,
     6. ControllingPlayer, out player))  
                {  
                    LoadResources();  
                }  
            }  
           
            base.HandleInput(gameTime, input);  
        }
3. Open the **HighScoreScreen.cs** file, and add a **using** statement for **Microsoft.Xna.Framework.Input**.
4. In the same file, modify the **HighScoreScreen** class’s **HandleInput** method.
   * 1. C#
     2. public override void HandleInput(GameTime gameTime, InputState input)  
        {  
            if (input == null)  
            {  
                throw new ArgumentNullException("input");  
            }  
           
            if (input.IsPauseGame(null))  
            {  
                Exit();  
            }  
           
            // Return to the main menu when a tap gesture is recognized  
            if (input.Gestures.Count > 0)  
            {  
                GestureSample sample = input.Gestures[0];  
                if (sample.GestureType == GestureType.Tap)  
                {  
                    Exit();  
           
                    input.Gestures.Clear();  
                }  
            }  
            // Handle gamepad input  
            PlayerIndex player;              
            // Handle keyboard input  
            if (input.IsNewKeyPress(Keys.Enter, ControllingPlayer, out player) ||  
                input.IsNewKeyPress(Keys.Space, ControllingPlayer, out player))  
            {  
                Exit();  
            }  
        }
5. Compile and run the game.

Using the arrow keys and the **Enter** key, you should now be able to navigate to the gameplay screen. The game is not playable yet because you have not modified the gameplay screen to react to user input. Furthermore, when you exit the game, the game shuts down because the high-score data uses isolated storage to save. This lab fixes the shutdown issue at a later stage.

# Handling keyboard input in the gameplay screen

* 1. To have the game function as designed, you must handle the keyboard input in the gameplay screen. The gameplay screen is more challenging than the screens revised in the previous task. Previously, it assumed that all input comes directly from the game’s virtual thumbstick. You need to add keyboard support, and you must make sure the code is unaware of the input’s source.

To add keyboard support to the gameplay screen

1. From the **Screens** project folder, open the **GameplayScreen.cs** file.
2. Rename the **GameplayScreen** class member from **userTapToExit** to **userInputToExit**, and add an additional field to the class.
   * 1. C#

Vector2 movementVector;

1. Modify the **GameplayScreen** class’s **HandleInput** method.
   * 1. **Note** As recommended previously, compare this method with the original version to see the exact scope of your changes.
     2. **Note** At the bottom of the method, there is now a call to a **SetMotion** method, which receives the input as a parameter. Furthermore, collision handling is now part of input handling, using a different version of the **HandleCollision** method.
     3. C#
     4. public override void HandleInput(GameTime gameTime, InputState input)  
        {  
            if (IsActive)  
            {  
                if (input == null)  
                {  
                    throw new ArgumentNullException("input");  
                }  
           
                if (input.IsPauseGame(null))  
                {  
                    PauseCurrentGame();  
                }  
            }  
           
            if (input.TouchState.Count > 0)  
            {  
                foreach (TouchLocation touch in input.TouchState)  
                {  
                    lastTouchPosition = touch.Position;  
                }  
            }  
           
            isSmokebuttonClicked = false;  
           
            PlayerIndex player;  
           
            VirtualThumbsticks.Update(input);  
           
            if (input.Gestures.Count > 0)  
            {  
                if (input.Gestures[0].GestureType == GestureType.DoubleTap)  
                {  
                    showDebugInfo = !showDebugInfo;  
                }  
            }  
           
            if (isLevelEnd)  
            {  
                if (input.Gestures.Count > 0)  
                {  
                    if (input.Gestures[0].GestureType == GestureType.Tap)  
                    {  
                        userInputToExit = true;  
                    }  
                }                  
           
                if (input.IsNewKeyPress(Keys.Enter, ControllingPlayer, out player) ||   
                    input.IsNewKeyPress(Keys.Space, ControllingPlayer, out player))  
                {  
                    userInputToExit = true;  
                }  
            }  
           
            if (!IsStarted)  
            {  
                return;  
            }  
           
            // If there was any touch  
            if (VirtualThumbsticks.RightThumbstickCenter.HasValue)  
            {  
                // Button Bounds  
                Rectangle buttonRectangle = new Rectangle((int)smokeButtonPosition.X,
     5. (int)smokeButtonPosition.Y, smokeButton.Width / 2,
     6. smokeButton.Height);  
           
                // Touch Bounds  
                Rectangle touchRectangle =
     7. new Rectangle(
     8. (int)VirtualThumbsticks.RightThumbstickCenter.Value.X,  
                        (int)VirtualThumbsticks.RightThumbstickCenter.Value.Y,  
                        1, 1);
     9. // If the touch is in the button  
                if (buttonRectangle.Contains(touchRectangle) &&

            !beeKeeper.IsCollectingHoney && !beeKeeper.IsStung)  
        {  
            isSmokebuttonClicked = true;  
        }  
    }              
   
    // Handle keyboard  
    if (input.IsNewKeyPress(Keys.Y, ControllingPlayer, out player))  
    {  
        showDebugInfo = !showDebugInfo;  
    }  
    if (input.IsKeyDown(Keys.Space, ControllingPlayer, out player) &&

        !beeKeeper.IsCollectingHoney &&  
        !beeKeeper.IsStung)  
        {  
        isSmokebuttonClicked = true;  
    }  
   
    movementVector = SetMotion(input);  
    beeKeeper.SetDirection(movementVector);  
    HandleCollision(gameTime);  
}

1. Modify the **GameplayScreen**’s **Update** method by changing the following block of code contained inside the method:
   * 1. C#
     2. HandleThumbStick();  
           
        HandleSmoke();  
           
        beeKeeper.SetDirection(VirtualThumbsticks.LeftThumbstick);  
           
        HandleCollision(gameTime);
     3. To the following:
     4. C#
     5. #if WINDOWS\_PHONE  
        HandleThumbStick();  
        #endif  
           
        HandleSmoke();
     6. HandleCollision(gameTime);
     7. These instructions remove the logic now performed by **HandleInput** and cause the virtual-thumbstick-related logic to compile when targeting the Windows Phone 7 only.
2. Modify the **SetMotion** method.

**Note** Other than adding keyboard input support, you are also changing the logic that checks whether the beekeeper is leaving the screen bounds. The next exercise provides more information about the **safeArea** variable used in the new logic.

* + 1. C#
    2. private Vector2 SetMotion(InputState input)  
       {  
           // Calculate the beekeeper location, if allow moving  
           Rectangle safeArea = SafeArea;  
          
           PlayerIndex playerIndex;  
          
           Vector2 leftThumbstick = VirtualThumbsticks.LeftThumbstick;  
          
           // Move on to keyboard input if we still have nothing  
           if (leftThumbstick == Vector2.Zero)  
           {  
               float vecX = 0;  
               float vecY = 0;  
          
               if (input.IsKeyDown(Keys.Left, ControllingPlayer, out playerIndex))  
               {  
                   vecX--;  
               }  
               if (input.IsKeyDown(Keys.Right, ControllingPlayer, out playerIndex))  
               {  
                   vecX++;  
               }  
               if (input.IsKeyDown(Keys.Up, ControllingPlayer, out playerIndex))  
               {  
                   vecY--;  
               }  
               if (input.IsKeyDown(Keys.Down, ControllingPlayer, out playerIndex))  
               {  
                   vecY++;  
               }  
          
               leftThumbstick = new Vector2(vecX, vecY);  
           }  
          
           Vector2 movementVector = leftThumbstick \* 12f;  
          
           Rectangle futureBounds = beeKeeper.Bounds;  
           futureBounds.X += (int)movementVector.X;  
           futureBounds.Y += (int)movementVector.Y;  
          
           if (futureBounds.Left <= safeArea.Left ||
    3. futureBounds.Right >= safeArea.Right)  
           {  
               movementVector.X = 0;  
           }  
           if (futureBounds.Top <= safeArea.Top ||
    4. futureBounds.Bottom >= safeArea.Bottom)  
           {  
               movementVector.Y = 0;  
           }  
          
           if (movementVector == Vector2.Zero)  
           {  
               IsInMotion = false;  
               beeKeeper.SetMovement(Vector2.Zero);  
           }  
           else  
           {  
               Vector2 beekeeperCalculatedPosition =  
                   new Vector2(beeKeeper.CentralCollisionArea.X,
    5. beeKeeper.CentralCollisionArea.Y) + movementVector;  
          
               if (!CheckBeehiveCollision(beekeeperCalculatedPosition))  
               {  
                   beeKeeper.SetMovement(movementVector);  
                   IsInMotion = true;  
               }  
           }  
          
           return movementVector;  
       }

1. Modify the **HandleVatCollision** method.
   * 1. C#
     2. private bool HandleVatCollision()  
        {  
            if (beeKeeper.Bounds.HasCollision(vat.VatDepositArea))  
            {  
                if (jar.HasHoney && !beeKeeper.IsStung &&
     3. !beeKeeper.IsDepostingHoney && movementVector == Vector2.Zero)  
                {  
                    beeKeeper.StartTransferHoney(4, EndHoneyDeposit);  
                }  
           
                return true;  
            }  
           
            beeKeeper.EndTransferHoney();  
            return false;  
        }
2. Modify the **HandleBeeKeeperBeehiveCollision** method.
   * 1. C#
     2. private bool HandleBeeKeeperBeehiveCollision()  
        {  
            bool isCollidingWithBeehive = false;  
           
            Beehive collidedBeehive = null;  
           
            // Goes over all the beehives  
            foreach (Beehive beehive in beehives)  
            {  
                // If the beekeeper intersects with the beehive  
                if (beeKeeper.Bounds.HasCollision(beehive.Bounds))  
                {  
                    if (movementVector == Vector2.Zero)  
                    {  
                        collidedBeehive = beehive;  
                        isCollidingWithBeehive = true;  
                    }  
                }  
                else  
                {  
                    beehive.AllowBeesToGenerate = true;  
                }  
            }  
           
            if (collidedBeehive != null)  
            {  
                // The beehive has honey, the jar can carry more honey, and the
     3. // beekeeper is not stung  
                if (collidedBeehive.HasHoney && jar.CanCarryMore &&
     4. !beeKeeper.IsStung)  
                {  
                    // Take honey from the beehive and put it in the jar  
                    collidedBeehive.DecreaseHoney(1);  
                    jar.IncreaseHoney(1);  
                    beeKeeper.IsCollectingHoney = true;  
                    AudioManager.PlaySound("FillingHoneyPot\_Loop");  
                }  
                else  
                {  
                    beeKeeper.IsCollectingHoney = false;  
                }  
           
                // Bees are not allowed to regenerate while the beekeeper is colliding
     5. // with their beehive  
                isCollidingWithBeehive = true;  
                collidedBeehive.AllowBeesToGenerate = false;  
            }  
            else  
            {  
                beeKeeper.IsCollectingHoney = false;  
                AudioManager.StopSound("FillingHoneyPot\_Loop");  
            }  
           
            return isCollidingWithBeehive;  
        }
3. Replace the **HandleThumbStick** method.

**Note** We remove some logic that **HandleInput** now handles, and make the method exist only when targeting the Windows Phone 7.

* + 1. C#
    2. #if WINDOWS\_PHONE  
       /// <summary>  
       /// Handle thumbstick logic  
       /// </summary>  
       private void HandleThumbStick()  
       {  
           // Calculate the rectangle of the outer circle of the thumbstick  
           Rectangle outerControlstick = new Rectangle(0,
    3. (int)controlstickBoundryPosition.Y - 35,  
               controlstickBoundry.Width + 60, controlstickBoundry.Height + 60);  
          
           // Reset the thumbstick position when it is idle  
           if (VirtualThumbsticks.LeftThumbstick == Vector2.Zero)  
           {  
               IsInMotion = false;  
               controlstickStartupPosition = new Vector2(55, 369);  
           }  
           else  
           {  
               // If not in motion and the touch point is not in the control bounds -
    4. // there is no movement  
               Rectangle touchRectangle = new Rectangle((int)lastTouchPosition.X,
    5. (int)lastTouchPosition.Y, 1, 1);  
          
               if (!outerControlstick.Contains(touchRectangle))  
               {  
                   controlstickStartupPosition = new Vector2(55, 369);  
                   IsInMotion = false;  
                   return;  
               }  
          
               // Moves the thumbstick's inner circle  
               float radious = controlstick.Width / 2 + 35;  
               controlstickStartupPosition = new Vector2(55, 369) +
    6. (VirtualThumbsticks.LeftThumbstick \* radious);  
           }  
       }  
       #endif

1. In the **CheckIfCurrentGameFinished** method, modify all references to the variable from **userTapToExit** to **userInputToExit**.
2. From the **ScreenManager** project folder, open **GameScreen.cs,** and add a **using** statement for **Microsoft.Xna.Framework.Graphics**.
3. In the same file, add the **SafeArea**.
   * 1. C#
     2. public Rectangle SafeArea  
        {  
            get  
            {  
                Viewport viewport = ScreenManager.Game.GraphicsDevice.Viewport;  
           
                return viewport.TitleSafeArea;  
            }  
        }

Finally, the beekeeper needs to react to the new input method supported by the gameplay screen.

1. Open **BeeKeeper.cs,** and add a field definition to the **BeeKeeper** class.
   * 1. WalkingDirection newDirection = WalkingDirection.Up;
2. In the same class, modify the **GetSpriteEffect** method.
   * 1. C#
     2. private SpriteEffects GetSpriteEffect(Vector2 movementDirection)  
        {  
            // Checks if there is any movement input  
            if (movementDirection != Vector2.Zero)  
            {  
                // If beekeeper is facing left  
                if (movementDirection.X < 0)  
                {  
                    lastEffect = SpriteEffects.FlipHorizontally;  
                }  
                else if (movementDirection.X > 0)  
                {  
                    lastEffect = SpriteEffects.None;  
                }  
            }  
           
            return lastEffect;  
        }
3. Modify the **DetermineDirection**, **DetermineDirectionDominantX** and **DetermineDirectionDominantY** methods.
   * 1. C#
     2. private void DetermineDirection(Vector2 movement,
     3. ref WalkingDirection tempDirection, ref Vector2 smokeAdjustment)  
        {  
            if (movement == Vector2.Zero)  
            {  
                return;  
            }  
           
            if (Math.Abs(movement.X) > Math.Abs(movement.Y))  
            {  
                DetermineDirectionDominantX(movement, ref tempDirection,
     4. ref smokeAdjustment);  
            }  
            else  
            {  
                DetermineDirectionDominantY(movement, ref tempDirection,
     5. ref smokeAdjustment);  
            }  
        }  
           
        private void DetermineDirectionDominantX(Vector2 movement,
     6. ref WalkingDirection tempDirection, ref Vector2 smokeAdjustment)  
        {  
            if (movement.X > 0)  
            {  
                if (movement.Y > 0.25f)  
                {  
                    tempDirection = WalkingDirection.RightDown;  
                    smokeAdjustment = new Vector2(UIConstants.SprayRightOffset,
     7. UIConstants.SprayDownOffset);  
                }  
                else if (movement.Y < -0.25f)  
                {  
                    tempDirection = WalkingDirection.RightUp;  
                    smokeAdjustment = new Vector2(UIConstants.SprayRightOffset,
     8. UIConstants.SprayUpOffset);  
                }  
                else  
                {  
                    tempDirection = WalkingDirection.Right;  
                    smokeAdjustment = new Vector2(UIConstants.SprayRightOffset,
     9. UIConstants.SprayMiddleOffset);  
                }  
            }  
            else  
            {  
                if (movement.Y > 0.25f)  
                {  
                    tempDirection = WalkingDirection.LeftDown;  
                    smokeAdjustment = new Vector2(-UIConstants.SprayLeftOffset,
     10. UIConstants.SprayDownOffset);  
                 }  
                 else if (movement.Y < -0.25f)  
                 {  
                     tempDirection = WalkingDirection.LeftUp;  
                     smokeAdjustment = new Vector2(-UIConstants.SprayLeftOffset,
     11. UIConstants.SprayUpOffset);  
                 }  
                 else  
                 {  
                     tempDirection = WalkingDirection.Left;  
                     smokeAdjustment = new Vector2(-UIConstants.SprayLeftOffset,
     12. UIConstants.SprayMiddleOffset);  
                 }  
             }  
         }  
            
         private void DetermineDirectionDominantY(Vector2 movement,
     13. ref WalkingDirection tempDirection, ref Vector2 smokeAdjustment)  
         {  
             if (movement.Y > 0)  
             {  
                 if (movement.X > 0.25f)  
                 {  
                     tempDirection = WalkingDirection.RightDown;  
                     smokeAdjustment = new Vector2(UIConstants.SprayRightOffset,
     14. UIConstants.SprayDownOffset);  
                 }  
                 else if (movement.X < -0.25f)  
                 {  
                     tempDirection = WalkingDirection.LeftDown;  
                     smokeAdjustment = new Vector2(-UIConstants.SprayLeftOffset,
     15. UIConstants.SprayDownOffset);  
                 }  
                 else  
                 {  
                     tempDirection = WalkingDirection.Down;  
                     smokeAdjustment = Vector2.Zero;  
                 }  
             }  
             else  
             {  
                 if (movement.X > 0.25f)  
                 {  
                     tempDirection = WalkingDirection.RightUp;  
                     smokeAdjustment = new Vector2(UIConstants.SprayRightOffset,
     16. UIConstants.SprayUpOffset);  
                 }  
                 else if (movement.X < -0.25f)  
                 {  
                     tempDirection = WalkingDirection.LeftUp;  
                     smokeAdjustment = new Vector2(-UIConstants.SprayLeftOffset,
     17. UIConstants.SprayUpOffset);  
                 }  
                 else  
                 {  
                     tempDirection = WalkingDirection.Up;  
                     smokeAdjustment = Vector2.Zero;  
                 }  
             }  
         }
4. Replace the **Draw** method.
   * 1. **Note** Compare the new method with the previous method to understand the exact scope of the changes you made to the method. Even though there are few differences, it is easier to provide the entire method than provide instructions to make the changes.
     2. C#
     3. public override void Draw(GameTime gameTime)  
        {  
            if (!(gamePlayScreen.IsActive && gamePlayScreen.IsStarted))  
            {  
                base.Draw(gameTime);  
                return;  
            }  
           
            // Make sure not to draw the beekeeper while flashing  
            if (isStung || isFlashing)  
            {  
                if (stungDrawingCounter != stungDrawingInterval)  
                {  
                    if (isDrawnLastStungInterval)  
                    {  
                        return;  
                    }  
                }  
            }  
           
            spriteBatch.Begin();  
           
            // if stung we want to show another animation  
            if (isStung)  
            {  
                spriteBatch.Draw(Game.Content.Load<Texture2D>("Textures/hit"),
     4. position, Color.White);  
                spriteBatch.End();  
                return;  
            }  
           
            // If collecting honey, draw the appropriate animation  
            if (IsCollectingHoney)  
            {  
                AnimationDefinitions[BeekeeperCollectingHoneyAnimationKey].Draw(
     5. spriteBatch, position, SpriteEffects.None);  
                spriteBatch.End();  
                return;  
            }  
           
           
            if (isDepositingHoney)  
            {  
                if (velocity != Vector2.Zero)  
                {  
                    isDepositingHoney = false;  
                    AudioManager.StopSound("DepositingIntoVat\_Loop");  
                }  
           
                // We want the deposit duration to sync with the deposit    
                // animation  
                // So we manage the timing ourselves  
                if (depositHoneyUpdatingTimer != TimeSpan.Zero &&  
                    depositHoneyUpdatingTimer + depositHoneyUpdatingInterval <
     6. gameTime.TotalGameTime)  
                {  
                    depositHoneyTimerCounter++;  
                    depositHoneyUpdatingTimer = TimeSpan.Zero;  
                }  
           
                AnimationDefinitions[BeekeeperDesposingHoneyAnimationKey].Draw(
     7. spriteBatch, position, SpriteEffects.None);  
           
                if (depositHoneyTimerCounter == honeyDepositFrameCount - 1)  
                {  
                    isDepositingHoney = false;  
                    depositHoneyCallback.Invoke(null);  
                    AnimationDefinitions[
     8. BeekeeperDesposingHoneyAnimationKey].PlayFromFrameIndex(0);  
                }  
           
                spriteBatch.End();  
                return;  
            }  
           
            bool hadDirectionChanged = false;  
           
            // See if the direction changed  
            if (newDirection != direction)  
            {  
                hadDirectionChanged = true;  
                direction = newDirection;  
            }  
           
            if (hadDirectionChanged)  
            {  
                // Update the animation  
                lastFrameCounter = 0;  
                AnimationDefinitions[LegAnimationKey].PlayFromFrameIndex(
     9. lastFrameCounter + (int)direction);  
                AnimationDefinitions[ShootingAnimationKey].PlayFromFrameIndex(
     10. lastFrameCounter + (int)direction);  
                 AnimationDefinitions[BodyAnimationKey].PlayFromFrameIndex(
     11. lastFrameCounter + (int)direction);  
             }  
             else  
             {  
                 // Because our animation is 8 cells, but the row is 16 cells,  
                 // we need to reset the counter after 8 rounds  
            
                 if (lastFrameCounter == 8)  
                 {  
                     lastFrameCounter = 0;  
                     AnimationDefinitions[LegAnimationKey].PlayFromFrameIndex(
     12. lastFrameCounter + (int)direction);  
                     AnimationDefinitions[ShootingAnimationKey].PlayFromFrameIndex(  
                         lastFrameCounter + (int)direction);  
                     AnimationDefinitions[BodyAnimationKey].PlayFromFrameIndex(
     13. lastFrameCounter + (int)direction);  
                 }  
                 else  
                 {  
                     lastFrameCounter++;  
                 }  
             }  
            
            
             AnimationDefinitions[LegAnimationKey].Draw(spriteBatch, position, 1f,
     14. SpriteEffects.None);  
            
             if (needToShootSmoke)  
             {  
                 // Draw the body  
                 AnimationDefinitions[ShootingAnimationKey].Draw(spriteBatch, position,
     15. 1f, SpriteEffects.None);  
            
                 // If true we need to draw smoke  
                 if (smokeAdjustment != Vector2.Zero)  
                 {  
                     AnimationDefinitions[SmokeAnimationKey].Draw(spriteBatch,
     16. position + smokeAdjustment, 1f, currentEffect);  
                 }  
             }  
             else  
             {  
                 AnimationDefinitions[BodyAnimationKey].Draw(spriteBatch, position,
     17. 1f, SpriteEffects.None);  
             }  
             spriteBatch.End();  
            
             base.Draw(gameTime);  
         }
5. Modify the **SetDirection** method.
   * 1. C#
     2. public void SetDirection(Vector2 movementDirection)  
        {  
            DetermineDirection(movementDirection, ref newDirection,
     3. ref smokeAdjustment);  
            currentEffect = GetSpriteEffect(movementDirection);  
        }
6. Compile and run the game.

The game should now play just as it did on the phone, although the game still shuts down when you try to save high-score data. We address this issue in the next task.

# Saving and loading high-score data—the high-score screen

* 1. Isolated storage stores and loads the high-score data on the Windows Phone 7. Do not use isolated storage when you target a Windows environment or Xbox 360. Instead, make your alterations for the Windows and Xbox 360 versions by using the proper method of storage, named “Storage.” This task focuses on the logic contained in the **HighScoreScreen** class.
     1. **Note** For more information about using isolated storage on Windows Phone 7, see <http://msdn.microsoft.com/en-us/library/ff402541%28VS.92%29.aspx>. For more information about the Storage mechanism on Windows and Xbox 360, see <http://msdn.microsoft.com/en-us/library/bb200105.aspx>

To add Storage support

1. To the **HoneycombRush – Windows** project, add a reference to **Microsoft.Xna.Framework.Storage**.
2. From the Screens project folder open the **HighScoreScreen.cs** file, and add the following **using** statements:
   * 1. C#
     2. using Microsoft.Xna.Framework.GamerServices;
     3. #if !WINDOWS\_PHONE  
        using Microsoft.Xna.Framework.Storage;  
        #endif
3. Add the following fields and properties to the **HighScoreScreen** class:
   * 1. C#
     2. Rectangle safeArea;  
           
        public static bool HighscoreLoaded { get; private set; }
     3. public static bool HighscoreSaved { get; private set; }  
           
        #if !WINDOWS\_PHONE  
        static bool shouldSaveHighScore;  
        static bool savingHighscore;  
        static bool loadingHighscore;  
        static bool deviceSelectorLaunched;  
           
        public static StorageDevice Storage { get; set; }  
        #endif
4. Add a static constructor.
   * 1. C#
     2. static HighScoreScreen()  
        {  
            HighscoreLoaded = false;
     3. HighscoreSaved = false;  
        }
5. Modify the **HighScoreScreen** class’s constructor.

**Note** The new constructor ensures that the screen does not appear before high-score data is loaded.

* + 1. C#

public HighScoreScreen()  
{  
    EnabledGestures = GestureType.Tap;  
    if (HighscoreLoaded == false)  
    {  
        throw new InvalidOperationException("Missing highscore data");  
    }  
   
    numberPlaceMapping = new Dictionary<int, string>();  
    InitializeMapping();  
}

1. Modify the **LoadContent** method.
   * 1. C#
     2. public override void LoadContent()  
        {  
            highScoreFont = Load<SpriteFont>(@"Fonts\HighScoreFont");  
           
            safeArea = SafeArea;  
           
            base.LoadContent();  
        }
2. Override the **Update** method.
   * 1. C#
     2. public override void Update(GameTime gameTime, bool otherScreenHasFocus,
     3. bool coveredByOtherScreen)  
        {  
        #if !WINDOWS\_PHONE  
            if (shouldSaveHighScore)  
            {  
                SaveHighscore();  
            }  
        #endif  
           
            base.Update(gameTime, otherScreenHasFocus, coveredByOtherScreen);  
        }
3. Modify the **Draw** method.
   * 1. C#
     2. public override void Draw(GameTime gameTime)  
        {  
            if (HighscoreLoaded == false)  
            {  
                base.Draw(gameTime);  
                return;  
            }  
           
            ScreenManager.SpriteBatch.Begin();  
           
            // Draw the highscores table  
            for (int i = 0; i < highScore.Count; i++)  
            {  
                if (!string.IsNullOrEmpty(highScore[i].Key))  
                {  
                    // Draw place number  
                    ScreenManager.SpriteBatch.DrawString(highScoreFont,
     3. GetPlaceString(i), new Vector2(20, i \* 72 + 86), Color.Black);  
           
                    // Draw Name  
                    ScreenManager.SpriteBatch.DrawString(highScoreFont,
     4. highScore[i].Key, new Vector2(210, i \* 72 + 86),
     5. Color.DarkRed);  
           
                    // Draw score  
                    ScreenManager.SpriteBatch.DrawString(highScoreFont,
     6. highScore[i].Value.ToString(), new Vector2(560, i \* 72 + 86),
     7. Color.Yellow);  
                }  
            }  
           
            ScreenManager.SpriteBatch.End();  
           
            base.Draw(gameTime);  
        }
4. Add the **HighScoreChanged** method.
   * 1. C#
     2. public static void HighScoreChanged()  
        {  
            HighscoreSaved = false;  
        }
5. Add the **InitializeStorageDevice** and **GetDevice** methods.

**Note** These methods enable the asynchronous initialization of the storage device used in storage operations.

* + 1. C#
    2. #if !WINDOWS\_PHONE  
       public static void InitializeStorageDevice()  
       {  
           if (Storage == null && Guide.IsVisible == false &&
    3. deviceSelectorLaunched == false)  
           {  
               deviceSelectorLaunched = true;  
               StorageDevice.BeginShowSelector(PlayerIndex.One, GetDevice, null);  
           }  
       }  
          
       static void GetDevice(IAsyncResult result)  
       {  
           Storage = StorageDevice.EndShowSelector(result);  
           deviceSelectorLaunched = false;  
       }  
       #endif

1. Modify the **SaveHighscore** method, and add two more methods, **SaveHighscoreToStorage** and **SaveContainerOpened**.

**Note** These new methods support the asynchronous nature of the high-score saving operation on Windows and Xbox 360.

* + 1. C#
    2. public static void SaveHighscore()  
       {  
       #if WINDOWS\_PHONE  
           // Get the place to store the data  
           using (IsolatedStorageFile isf =
    3. IsolatedStorageFile.GetUserStoreForApplication())  
           {  
               // Create the file to save the data  
               using (IsolatedStorageFileStream isfs =
    4. isf.CreateFile(HighScoreScreen.HighScoreFilename))  
               {  
                   using (StreamWriter writer = new StreamWriter(isfs))  
                   {  
                       for (int i = 0; i < highScore.Count; i++)  
                       {  
                           // Write the scores  
                           writer.WriteLine(highScore[i].Key);  
                           writer.WriteLine(highScore[i].Value.ToString());  
                       }  
                   }  
               }  
           }  
          
           HighscoreSaved = true;  
       #else  
           if (Storage == null || Storage.IsConnected == false)  
           {  
               shouldSaveHighScore = true;  
               // We do not have a storage device, initialize it                  
               InitializeStorageDevice();  
           }  
           else if (!savingHighscore)  
           {  
               shouldSaveHighScore = false;  
               savingHighscore = true;  
               SaveHighscoreToStorage();  
           }  
       #endif  
       }  
          
       #if !WINDOWS\_PHONE  
       private static void SaveHighscoreToStorage()  
       {  
           Storage.BeginOpenContainer(HoneycombRush.GameName, SaveContainerOpened,
    5. null);  
       }  
          
       private static void SaveContainerOpened(IAsyncResult result)  
       {  
           StorageContainer container = Storage.EndOpenContainer(result);  
          
           if (container.FileExists(HighScoreFilename))  
           {  
               container.DeleteFile(HighScoreFilename);  
           }  
          
           Stream stream = container.CreateFile(HighScoreFilename);  
          
           using (StreamWriter writer = new StreamWriter(stream))  
           {  
               for (int i = 0; i < highScore.Count; i++)  
               {  
                   // Write the scores  
                   writer.WriteLine(highScore[i].Key);  
                   writer.WriteLine(highScore[i].Value.ToString());  
               }  
           }  
          
           HighscoreSaved = true;  
           savingHighscore = false;  
       }  
       #endif

1. Modify the **LoadHighscores** method, and add two more methods, **LoadHighscoreFromStorage** and **LoadContainerOpened**.

**Note** These new methods support the asynchronous nature of the high-score loading operation on Windows and Xbox 360.

* + 1. C#
    2. public static void LoadHighscores()  
       {  
       #if WINDOWS\_PHONE  
           // Get the place the data stored  
           using (IsolatedStorageFile isf =
    3. IsolatedStorageFile.GetUserStoreForApplication())  
           {  
               // Try to open the file  
               if (isf.FileExists(HighScoreScreen.HighScoreFilename))  
               {  
                   using (IsolatedStorageFileStream isfs = isf.OpenFile(
    4. HighScoreScreen.HighScoreFilename, FileMode.Open))  
                   {  
                       // Get the stream to read the data  
                       using (StreamReader reader = new StreamReader(isfs))  
                       {  
                           // Read the highscores  
                           int i = 0;  
                           while (!reader.EndOfStream)  
                           {  
                               string name = reader.ReadLine();  
                               string score = reader.ReadLine();  
                               highScore[i++] = new KeyValuePair<string, int>(name,
    5. int.Parse(score));  
                           }  
                       }  
                   }  
               }  
           }  
          
           OrderGameScore();  
          
           HighscoreLoaded = true;  
       #else  
           if (Storage == null || Storage.IsConnected == false)  
           {  
               // We do not have a storage device, initialize it                  
               InitializeStorageDevice();  
           }  
           else if (!loadingHighscore)  
           {  
               loadingHighscore = true;  
               LoadHighscoreFromStorage();  
           }  
       #endif  
       }  
          
       #if !WINDOWS\_PHONE  
       public static void LoadHighscoreFromStorage()  
       {  
           Storage.BeginOpenContainer(HoneycombRush.GameName, LoadContainerOpened,
    6. null);  
       }  
          
       public static void LoadContainerOpened(IAsyncResult result)  
       {  
           StorageContainer container = Storage.EndOpenContainer(result);  
          
           if (container.FileExists(HighScoreFilename))  
           {  
               Stream stream = container.OpenFile(HighScoreFilename, FileMode.Open);  
          
               using (StreamReader reader = new StreamReader(stream))  
               {  
                   // Read the highscores  
                   int i = 0;  
                   while (!reader.EndOfStream)  
                   {  
                       string name = reader.ReadLine();  
                       string score = reader.ReadLine();  
                       highScore[i++] = new KeyValuePair<string, int>(name,
    7. int.Parse(score));  
                   }  
               }  
           }  
          
           OrderGameScore();  
          
           HighscoreLoaded = true;  
           loadingHighscore = false;  
       }  
       #endif

1. Open **HoneycombRush.cs,** and add a field to the **HoneycombRush**.
   * 1. C#
     2. public static String GameName = "Honeycomb Rush";

# Saving and loading high-score data—other screens

* 1. This task is a continuation of the previous task. The high-score screen now operates with the correct storage mechanism when it targets the Windows platform. However, keep in mind that storage operations on the Windows platform are asynchronous. This means you need to modify parts of the game to manage this issue. The most notable change is that now the player enters a name when transitioning from the level-over screen to the high-score screen, instead of entering a name directly after finishing the last level.

To use the new high-score screen

1. From the **Screens** project folder, open the **MainMenuScreen.cs** file, and add the following **using** statements:
   * 1. C#
     2. using Microsoft.Xna.Framework.GamerServices;  
        #if !WINDOWS\_PHONE  
        using Microsoft.Xna.Framework.Storage;  
        #endif
2. In the same file, add a field to the **MainMenuScreen** class.
   * 1. C#
     2. bool isExiting = false;
3. Override the **Update** and **HandleInput** methods.
   * 1. C#
     2. public override void Update(GameTime gameTime, bool otherScreenHasFocus,
     3. bool coveredByOtherScreen)  
        {  
            if (isExiting)  
            {  
                if (!HighScoreScreen.HighscoreSaved)  
                {  
                    HighScoreScreen.SaveHighscore();  
                }  
                else  
                {  
                    isExiting = false;  
                    ScreenManager.Game.Exit();  
                }  
            }  
           
            base.Update(gameTime, otherScreenHasFocus, coveredByOtherScreen);  
        }  
           
        public override void HandleInput(GameTime gameTime, InputState input)  
        {  
            if (isExiting)  
            {  
                return;  
            }  
           
            base.HandleInput(gameTime, input);  
        }
4. Modify the **OnCancel** method.
   * 1. C#
     2. protected override void OnCancel(PlayerIndex playerIndex)  
        {  
            isExiting = true;              
           
            AudioManager.StopSound("MenuMusic\_Loop");  
        }
5. From the **Screens** project folder, open **LevelOverScreen.cs**, and add the following **using** statement:
   * 1. C#
     2. using Microsoft.Xna.Framework.GamerServices;
6. Modify the **LevelOverScreen** class by changing the **StartNewLevelOrExit** method.
   * 1. C#
     2. private void StartNewLevelOrExit(InputState input)  
        {  
            // If there is no next level - go to high score screen  
            if (!difficultyMode.HasValue)  
            {  
                // If is in high score, gets is name  
                if (GameplayScreen.FinalScore != 0 &&
     3. HighScoreScreen.IsInHighscores(GameplayScreen.FinalScore))  
                {  
                    Guide.BeginShowKeyboardInput(PlayerIndex.One,  
                        "Player Name", "What is your name (max 15 characters)?",
     4. "Player", AfterPlayerEnterName, null);  
                }  
                else  
                {  
                    foreach (GameScreen screen in ScreenManager.GetScreens())  
                    {  
                        screen.ExitScreen();  
                    }  
           
                    ScreenManager.AddScreen(new BackgroundScreen("highScoreScreen"),
     5. null);  
                    ScreenManager.AddScreen(new HighScoreScreen(), null);  
                }  
            }  
            // If not already loading  
            else if (!isLoading)  
            {  
                // Start loading the resources in an additional thread  
                thread = new Thread(new ThreadStart(gameplayScreen.LoadAssets));  
           
                isLoading = true;  
                thread.Start();  
            }  
        }
7. Replace the **Update** method.
   * 1. C#
     2. public override void Update(GameTime gameTime, bool otherScreenHasFocus, bool coveredByOtherScreen)  
        {  
            // If necessary, load the highscore. Remember that calling LoadHighscores multiple times does not have  
            // and adverse effect. Highscore will only be loaded once.  
            if (!HighScoreScreen.HighscoreLoaded)  
            {  
                HighScoreScreen.LoadHighscores();  
            }  
            // If additional thread is running, do nothing  
            else if (null != thread)  
            {  
                // If additional thread finished loading and the screen is not exiting  
                if (thread.ThreadState == ThreadState.Stopped && !IsExiting)  
                {  
                    // Move on to the game play screen once highscore data is loaded                      
                    foreach (GameScreen screen in ScreenManager.GetScreens())  
                    {  
                        screen.ExitScreen();  
                    }  
           
                    ScreenManager.AddScreen(gameplayScreen, null);  
                }  
            }  
           
            base.Update(gameTime, otherScreenHasFocus, coveredByOtherScreen);  
        }
8. Add the **AfterPlayerEnterName** method.
   * 1. C#
     2. private void AfterPlayerEnterName(IAsyncResult result)  
        {  
            // Gets the name entered  
            string playerName = Guide.EndShowKeyboardInput(result);  
            if (!string.IsNullOrEmpty(playerName))  
            {  
                // Ensure that it is valid  
                if (playerName != null && playerName.Length > 15)  
                    playerName = playerName.Substring(0, 15);  
           
                // Puts it in high score  
                HighScoreScreen.PutHighScore(playerName, GameplayScreen.FinalScore);  
                HighScoreScreen.HighScoreChanged();  
            }  
           
            // Moves to the next screen  
            foreach (GameScreen screen in ScreenManager.GetScreens())  
            {  
                screen.ExitScreen();  
            }  
           
            ScreenManager.AddScreen(new BackgroundScreen("highScoreScreen"), null);  
            ScreenManager.AddScreen(new HighScoreScreen(), null);  
        }
9. Open the **GameplayScreen.cs** file, remove the **Score** property from the **GameplayScreen** class, and replace it with the following field:
   * 1. C#
     2. public static int FinalScore;
10. In the same class, modify the **DrawLevelEndIfNecessary** method.
    * 1. C#
      2. private void DrawLevelEndIfNecessary()  
         {  
             if (isLevelEnd)  
             {  
                 string stringToDisplay = string.Empty;  
            
                 if (isUserWon)  
                 {  
                     if (FinalScore != 0 && HighScoreScreen.IsInHighscores(FinalScore))  
                     {  
                         stringToDisplay = "It's a new\nHigh-Score!";  
                     }  
                     else  
                     {  
                         stringToDisplay = "You Win!";  
                     }  
                 }  
                 else  
                 {  
                     stringToDisplay = "Time Is Up!";  
                 }  
            
                 Vector2 stringVector = font36px.MeasureString(stringToDisplay);  
            
                 ScreenManager.SpriteBatch.DrawString(font36px, stringToDisplay,  
                     new Vector2(ScreenManager.GraphicsDevice.Viewport.Width / 2 -
      3. stringVector.X / 2,
      4. ScreenManager.GraphicsDevice.Viewport.Height / 2 -
      5. stringVector.Y / 2), Color.White);  
             }  
         }
11. Remove the **CheckIsInHighScore** method.
12. Modify the **CheckIfCurrentGameFinished** method.
    * 1. C#
      2. private bool CheckIfCurrentGameFinished()  
         {  
             levelEnded = false;  
             isUserWon = vat.CurrentVatCapacity >= vat.MaxVatCapacity;  
            
             // If the vat is full, the player wins  
             if (isUserWon || gameElapsed <= TimeSpan.Zero)  
             {  
                 levelEnded = true;  
            
                 if (gameDifficultyLevel == DifficultyMode.Hard)  
                 {  
                     FinalScore = ConfigurationManager.ModesConfiguration[
      3. gameDifficultyLevel].HighScoreFactor \*
      4. (int)gameElapsed.TotalMilliseconds;  
                 }  
                 else  
                 {  
                     FinalScore = 0;  
                 }  
             }  
            
             // if true, game is over  
             if (gameElapsed <= TimeSpan.Zero || levelEnded)  
             {  
                 isLevelEnd = true;  
            
                 if (userInputToExit)  
                 {  
                     ScreenManager.RemoveScreen(this);  
            
                     if (isUserWon) // True - the user won  
                     {  
                         AudioManager.PlaySound("Victory");  
                     }  
                     else  
                     {  
                         AudioManager.PlaySound("Defeat");  
                     }  
            
                     MoveToNextScreen(isUserWon);  
                 }  
             }  
            
             return false;  
         }
13. Modify the **UnloadContent** method.
    * 1. C#
      2. public override void UnloadContent()  
         {  
             var componentList = ScreenManager.Game.Components;  
            
             for (int index = 0; index < componentList.Count; index++)  
             {  
                 if (componentList[index] is TexturedDrawableGameComponent ||
      3. componentList[index] is ScoreBar)  
                 {  
                     componentList.RemoveAt(index);  
                     index--;  
                 }  
             }  
            
             base.UnloadContent();  
         }
14. Remove the **AfterPlayerEnterName** method.
15. Open the **HoneycombRush.cs** file, and add the following **using** statement:
    * 1. C#
      2. using Microsoft.Xna.Framework.GamerServices;
16. In the same file, add the following statement to the **HoneycombRush** class’s constructor:

**Note** **GamerServicesComponent** is required to display the prompt where the player enters a name after achieving a high score.

* + 1. C#

#if WINDOWS  
            Components.Add(new GamerServicesComponent(this));  
#endif

1. Compile and run the game.

The game should now be operational and free of errors. Note that since the gameplay elements are not positioned properly, part of the game will appear off screen.

# Introducing high-definition assets

* 1. At this stage, while the game now functions on a Windows platform, all of the game’s assets have a low resolution—because the original display is for a Windows Phone 7 device. All the game elements are positioned at hard-coded locations based on the resolution of Windows Phone 7. This task’s objective is to improve the game’s presentation on Windows, while preserving its presentation on Windows Phone 7.

To add high-definition assets to the project

1. In **Solution Explorer**, right-click the **HoneycombRushContent** content project, and then click **Remove**.
2. In **Solution Explorer**, right-click the **HoneycombRush** solution, point to **Add**, and then click **New Project**.
3. In the Add New Project dialog box, select **XNA Game Studio 4.0** in the left, and then click **Empty Content Project (4.0)**.
4. In the **Name** box, type **HoneycombRushContentHD, an**d then click **OK**.
5. In **Solution Explorer**, right-click the **HoneycombRushContentHD** project, and press **F4** to open the new content project’s properties.
6. Modify the **Content Root Directory** to “Content.”
7. In the lab’s installation folder under **Assets\Media,** add all the assets to the new content project, while preserving the directory structure.
8. In **Solution Explorer** right-click **HoneycombRush – Windows**, select **Add Content Reference**, and then point to the new content project.
9. In **Solution Explorer**, expand **HoneycombRushContentHD**, expand the **Configuration** project folder, click **Configuration.xml,** and **press F4.**
10. Change **Build Action** to **None,** and **Copy to Output Directory** to **Copy if newer**.
11. In **Solution Explorer**, expand **HoneycombRushContentHD**, expand the **Textures** project folder, click **AnimationsDefinition.xml,** and then press **F4**.
12. Change **Build Action** to **None,** and **Copy to Output Directory** to **Copy if newer**.
13. In **Solution Explorer**, expand **HoneycombRushContentHD**, expand the **Sounds** project folder**,** click **InGameSong\_Loop.wav,** and then press **F4**.
14. Change **Content Processor** to **Song - XNA Framework**.
15. In **Solution Explorer**, expand **HoneycombRushContentHD**, expand the **Sounds** project folder**,** click **MenuMusic\_Loop.wav,** and then press **F4**.
16. Change **Content Processor** to **Song - XNA Framework**.
17. From the **Screens** project folder, open the **MainMenuScreen.cs** file.
18. Modify the **MainMenuScreen** class’s **StartGameMenuEntrySelected** method.
    * 1. C#
      2. void StartGameMenuEntrySelected(object sender, EventArgs e)  
         {  
             foreach (GameScreen screen in ScreenManager.GetScreens())  
                 screen.ExitScreen();  
         #if WINDOWS\_PHONE  
             ScreenManager.AddScreen(new BackgroundScreen("Instructions"), null);
      3. #elseif XBOX  
             ScreenManager.AddScreen(new BackgroundScreen("InstructionsXbox"), null);
      4. #else  
             ScreenManager.AddScreen(new BackgroundScreen("InstructionsPC"), null);  
         #endif  
            
             ScreenManager.AddScreen(new LoadingAndInstructionScreen(), null);  
            
             AudioManager.StopSound("MenuMusic\_Loop");  
         }
19. Compile and run the game.

While the game now uses higher resolution assets, it looks worse than before since the hard-coded positions do not fit the new assets.

# Adding scaling capabilities to all drawing operations

* 1. You added high-definition assets in the previous task, but you cannot use them in their current form when you run in a Windows computer environment. As it is preferable to support monitors with limited capabilities, the target for the game is to run at 800x480. This means you must scale down all of the high-definition assets to fit onto the screen.

To make it easy to scale all drawing operations

1. From the **Misc** project folder, open the **Animation.cs** file, and use Visual Studio’s refactoring capabilities to rename **Animation** to **ScaledAnimation**.
2. From the **Misc** project folder, remove the file **Animation.cs**.
3. From the lab’s installation folder under **Assets\Code**, add the files **ScaledAnimation.cs** and **ScaledSpriteBatch.cs** to the **Misc** project folder.

**Note** **ScaledSpriteBatch.cs** defines an alternate sprite batch that hides all **Draw** and **DrawString** methods, replacing them with alternate versions that are scaled according to the class’s **ScaleVector** property. **ScaledAnimation.cs** is a version of **Animation.cs**, which uses a scaled sprite batch instead of the regular **SpriteBatch** class.

1. From the **ScreenManager** project folder, open the file **ScreenManager.cs**.
2. Locate the **spriteBatch** field, and modify its type to **ScaledSpriteBatch**.
3. Add a field to the **ScreenManager** class.
   * 1. C#
     2. Vector2 drawingScale;
4. Locate the **SpriteBatch** property, and modify its type to **ScaledSpriteBatch**.
5. Modify the **ScreenManager** class’s constructor.
   * 1. C#
     2. public ScreenManager(Game game, Vector2 drawingScale)  
            : base(game)  
        {  
            // we must set EnabledGestures before we can query for them, but  
            // we don't assume the game wants to read them.  
            TouchPanel.EnabledGestures = GestureType.None;  
           
            this.drawingScale = drawingScale;  
        }
6. Modify the **LoadContent** method.
   * 1. C#
     2. protected override void LoadContent()  
        {  
            // Load content belonging to the screen manager.  
            ContentManager content = Game.Content;  
           
            spriteBatch = new ScaledSpriteBatch(Game.GraphicsDevice, drawingScale);  
            Game.Services.AddService(typeof(ScaledSpriteBatch), spriteBatch);   
           
            font = content.Load<SpriteFont>("Fonts/MenuFont");  
            blankTexture = content.Load<Texture2D>("Textures/Backgrounds/blank");  
            buttonBackground = content.Load<Texture2D>(
     3. "Textures/Backgrounds/buttonBackground");  
           
            // Tell each of the screens to load their content.  
            foreach (GameScreen screen in screens)  
            {  
                screen.LoadContent();  
            }  
        }
7. Open the **HoneycombRush** solution, which houses the Windows Phone 7 version of the game, and repeat steps 2 and 3. Note that when you do step 3, you must add the items already in the **Misc** folder.

# Modifying the game to scale all drawing operations

* 1. The previous task used the **ScaledSpriteBatch class** to scale all drawing operations.

To integrate the scaled sprite batch into the game code

1. Open the **HoneycombRush - Windows** solution, and then open the **HoneycombRush.cs** file.
2. Modify the **HoneycombRush** class’s constructor.
   * 1. C#
     2. public HoneycombRush()  
        {  
            // Initialize sound system  
            AudioManager.Initialize(this);  
           
            graphics = new GraphicsDeviceManager(this);  
            Content.RootDirectory = "Content";  
           
        #if WINDOWS\_PHONE  
            // Frame rate is 30 fps by default for Windows Phone.  
            TargetElapsedTime = TimeSpan.FromTicks(333333);  
           
            graphics.IsFullScreen = true;  
           
            screenManager = new ScreenManager(this, Vector2.One);  
        #elseif WINDOWS  
            graphics.PreferredBackBufferHeight = 480;  
            graphics.PreferredBackBufferWidth = 800;              
           
            // Make the game windowed  
            graphics.IsFullScreen = false;  
            IsMouseVisible = true;  
           
            Components.Add(new GamerServicesComponent(this));  
           
            Vector2 scaleVector =
     3. new Vector2(graphics.PreferredBackBufferWidth / 1280f,   
                graphics.PreferredBackBufferHeight / 720f);  
           
            UIConstants.SetScale(scaleVector);  
           
            // Create a new instance of the Screen Manager. Have all drawing scaled
     4. // from 720p to the PC's resolution  
            screenManager = new ScreenManager(this, scaleVector);  
        #endif  
           
            screenManager.AddScreen(new BackgroundScreen("titleScreen"), null);  
            screenManager.AddScreen(new MainMenuScreen(), PlayerIndex.One);  
            Components.Add(screenManager);  
        }
3. From the **Object** project folder, open the **TexturedDrawableGameComponent.cs** file.
4. Use the refactoring capabilities of Visual Studio to rename the **spriteBatch** field to **scaledSpriteBatch**.
5. Modify the type of the **scaledSpriteBatch** field to **ScaledSpriteBatch**.
6. Modify the **Bounds** property.
   * 1. C#
     2. public virtual Rectangle Bounds  
        {  
            get  
            {  
                if (texture == null)  
                {  
                    return default(Rectangle);  
                }  
                else  
                {  
                    return new Rectangle((int)position.X, (int)position.Y,   
                        (int)(texture.Width \* scaledSpriteBatch.ScaleVector.X),   
                        (int)(texture.Height \* scaledSpriteBatch.ScaleVector.Y));  
                }  
            }  
        }
7. Modify the class’s constructor.
   * 1. C#
     2. public TexturedDrawableGameComponent(Game game, GameplayScreen gamePlayScreen)  
            : base(game)  
        {  
            this.gamePlayScreen = gamePlayScreen;  
           
            scaledSpriteBatch = (ScaledSpriteBatch)game.Services.GetService(
     3. typeof(ScaledSpriteBatch));  
        }
8. From the **Object** project folder, open the **ScoreBar.cs** file.
9. Using the refactoring capabilities of Visual Studio, modify the **ScoreBar** class’s **spriteBatch** field to **scaledSpriteBatch**.
10. Modify the type of the **scaledSpriteBatch** field to **ScaledSpriteBatch**.
11. Modify the class’s constructor.
    * 1. C#
      2. public ScoreBar(Game game, int minValue, int maxValue, Vector2 position,
      3. int height, int width,  
             Color scoreBarColor, ScoreBarOrientation scoreBarOrientation,
      4. int initialValue, GameplayScreen screen, bool isAppearAtCountDown)  
             : base(game)  
         {  
             this.MinValue = minValue;  
             this.MaxValue = maxValue;  
             this.Position = position;  
             this.ScoreBarColor = scoreBarColor;  
             this.scoreBarOrientation = scoreBarOrientation;  
             this.currentValue = initialValue;  
             this.width = width;  
             this.height = height;  
             this.gameplayScreen = screen;  
             this.isAppearAtCountDown = isAppearAtCountDown;  
            
            
             scaledSpriteBatch = (ScaledSpriteBatch)Game.Services.GetService(
      5. typeof(ScaledSpriteBatch));  
            
             GetSpaceFromBorder();  
         }

# Fitting visual element positions to the target platform—the gameplay screen

* 1. In this task, you will modify the game’s visual element positioning code to consider the target platform.

To improve the game’s display

1. In **Solution Explorer**, right-click the **Misc** project folder, point to add, and then click **Class**.
2. In the Add New Item dialog box, type **UIConstants** in the **Name** box, and then click **Add**.
3. Change the namespace in the new class file to **HoneycombRush**.
4. Modify the class created in the previous steps.

**Note** This class defines various constants that position the various game elements on the screen. Notice that when you target a Windows computer, you can change the constants to match the scaling introduced in the previous tasks. You will use these values in later steps.

* + 1. C#

static class UIConstants  
{  
#if WINDOWS\_PHONE  
    public const float BeehiveLeftMargin = 85;  
    public const float BeehiveRightMargin = 85;  
    public const float BeehiveTopMargin = 10;  
    /// <summary>  
    /// An additional offset used for positioning the bottom two beehives.  
    /// </summary>  
    public const float BeehiveMiddleOffset = 0;  
   
    public const float HoneyJarTopMargin = 8;  
    public const float HoneyJarLeftMargin = 20;  
   
    public const float VatArrowOffset = -20;  
    public const float VatBottomMargin = 25;  
    public const int VatScorebarHeight = 10;  
    public const int VatScorebarWidth = 190;  
   
    /// <summary>  
    /// The size of a single frame in the smoke button's two-state texture.  
    /// </summary>  
    public const int SmokeButtonSize = 109;  
   
    /// <summary>  
    /// This margin takes the button's own size in consideration.  
    /// </summary>  
    public const float SmokeButtonRightAbsoluteMargin = 137;  
    /// <summary>  
    /// This margin takes the button's own size in consideration.  
    /// </summary>  
    public const float SmokeButtonBottomAbsoluteMargin = 144;  
   
    public const float SprayUpOffset = 0;  
    public const float SprayMiddleOffset = 25;  
    public const float SprayDownOffset = 30;  
    public const float SprayRightOffset = 75;  
    public const float SprayLeftOffset = 75;  
   
    public const float HighScorePlaceLeftMargin = 15;  
    public const float HighScoreNameLeftMargin = 190;  
    public const float HighScoreScoreLeftMargin = 545;  
    public const float HighScoreTopMargin = 86;  
    public const float HighScoreVerticalJump = 72;  
#else  
    /// <summary>  
    /// Updates all constants returned by the class according to a specified

    /// scale factor. Always call this method at least once before retrieving

    /// constants from the class.  
    /// </summary>  
    /// <param name="scaleVector">Vector depicting the scale used.</param>  
    public static void SetScale(Vector2 scaleVector)  
    {  
        BeehiveLeftMargin = 130 \* scaleVector.X;  
        BeehiveRightMargin = 130 \* scaleVector.X;  
        BeehiveTopMargin = 30 \* scaleVector.Y;              
        BeehiveMiddleOffset = 70 \* scaleVector.Y;  
   
        HoneyJarTopMargin = 12 \* scaleVector.Y;  
        HoneyJarLeftMargin = 30 \* scaleVector.Y;  
   
        VatArrowOffset = -35 \* scaleVector.Y;  
        VatBottomMargin = 50 \* scaleVector.Y;  
        VatScorebarHeight = (int)(20 \* scaleVector.Y);  
        VatScorebarWidth = (int)(210 \* scaleVector.X);  
   
        SmokeButtonSize = 137;  
        SmokeButtonRightAbsoluteMargin = 170 \* scaleVector.X;              
        SmokeButtonBottomAbsoluteMargin = 200 \* scaleVector.Y;  
   
        SprayUpOffset = 24 \* scaleVector.Y;  
        SprayMiddleOffset = 65 \* scaleVector.Y;  
        SprayDownOffset = 110 \* scaleVector.Y;  
        SprayRightOffset = 230 \* scaleVector.X;  
        SprayLeftOffset = 130 \* scaleVector.X;  
   
        HighScorePlaceLeftMargin = 50 \* scaleVector.X;  
        HighScoreNameLeftMargin = 300 \* scaleVector.X;  
        HighScoreScoreLeftMargin = 960 \* scaleVector.X;  
        HighScoreTopMargin = 147 \* scaleVector.Y;  
        HighScoreOddVerticalJump = 74 \* scaleVector.Y;  
        HighScoreEvenVerticalJump = 69 \* scaleVector.Y;  
    }  
   
    public static float BeehiveLeftMargin { get; private set; }  
    public static float BeehiveRightMargin { get; private set; }  
    public static float BeehiveTopMargin { get; private set; }  
    /// <summary>  
    /// An additional offset used for positioning the bottom two beehives.  
    /// </summary>  
    public static float BeehiveMiddleOffset { get; private set; }  
   
    public static float HoneyJarTopMargin { get; private set; }  
    public static float HoneyJarLeftMargin { get; private set; }  
   
    public static float VatArrowOffset { get; private set; }  
    public static float VatBottomMargin { get; private set; }  
    public static int VatScorebarHeight { get; private set; }  
    public static int VatScorebarWidth { get; private set; }  
   
    /// <summary>  
    /// Sets the size of the smoke button. This does not take the scale

    /// factor into account.  
    /// </summary>  
    public static float SmokeButtonSize { get; private set; }  
   
    /// <summary>  
    /// This margin takes the button's own size in consideration.  
    /// </summary>  
    public static float SmokeButtonRightAbsoluteMargin { get; private set; }  
    /// <summary>  
    /// This margin takes the button's own size in consideration.  
    /// </summary>  
    public static float SmokeButtonBottomAbsoluteMargin { get; private set; }  
   
    public static float SprayUpOffset { get; private set; }  
    public static float SprayMiddleOffset { get; private set; }  
    public static float SprayDownOffset { get; private set; }  
    public static float SprayRightOffset { get; private set; }  
    public static float SprayLeftOffset { get; private set; }  
   
    public static float HighScorePlaceLeftMargin { get; private set; }  
    public static float HighScoreNameLeftMargin { get; private set; }  
    public static float HighScoreScoreLeftMargin { get; private set; }  
    public static float HighScoreTopMargin { get; private set; }  
    public static float HighScoreOddVerticalJump { get; private set; }  
    public static float HighScoreEvenVerticalJump { get; private set; }  
#endif  
}

1. Open the **GameplayScreen.cs** file.
2. Add two fields to the **GameplayScreen** class.
   * 1. C#
     2. Vector2 smokeTextLocation;
     3. Vector2 vatArrowPosition;
3. Modify the **GameplayScreen** class’s **Draw** method.
   * 1. C#
     2. public override void Draw(GameTime gameTime)  
        {  
            ScreenManager.SpriteBatch.Begin();  
           
            ScreenManager.SpriteBatch.Draw(background,
     3. ScreenManager.GraphicsDevice.Viewport.Bounds, null,  
                Color.White, 0f, Vector2.Zero, SpriteEffects.None, 1);  
           
           
            // Draw count down screen  
            if (isAtStartupCountDown)  
            {  
                DrawStartupString();  
            }  
           
            if (IsActive && IsStarted)  
            {  
                DrawSmokeButton();  
           
        #if WINDOWS\_PHONE  
                // Only draw the virtual thumbstick on the phone  
                ScreenManager.SpriteBatch.Draw(controlstickBoundry,
     4. controlstickBoundryPosition, Color.White);  
                ScreenManager.SpriteBatch.Draw(controlstick,
     5. controlstickStartupPosition, Color.White);  
        #endif  
           
                ScreenManager.SpriteBatch.DrawString(font16px, SmokeText,
     6. smokeTextLocation, Color.White);  
           
                DrawVatHoneyArrow();  
            }  
           
            DrawLevelEndIfNecessary();  
           
            ScreenManager.SpriteBatch.End();  
           
            base.Draw(gameTime);  
        }
4. Modify the **CreateGameComponents** method.

**Note** Unlike other method changes shown previously, we change this method extensively to remove all hard coded placements of the game’s visual elements.

* + 1. C#
    2. private void CreateGameComponents()  
       {  
           Vector2 scaleVector = ScreenManager.SpriteBatch.ScaleVector;  
          
           Rectangle safeArea = SafeArea;  
          
           Texture2D jarTexture = ScreenManager.Game.Content.Load<Texture2D>(
    3. "Textures/honeyJar");  
          
           Vector2 honeyJarLocation =  
               safeArea.GetVector() + new Vector2(UIConstants.HoneyJarLeftMargin,
    4. UIConstants.HoneyJarTopMargin);  
          
           Vector2 jarBarLocation = honeyJarLocation + new Vector2(0,
    5. jarTexture.Height \* scaleVector.Y + 7);  
          
           ScoreBar scoreBar = new ScoreBar(ScreenManager.Game, 0, 100,
    6. jarBarLocation, (int)(jarTexture.Height / 6 \* scaleVector.Y),
    7. (int)(jarTexture.Width \* scaleVector.X), Color.Blue,   
               ScoreBar.ScoreBarOrientation.Horizontal, 0, this, true);  
           ScreenManager.Game.Components.Add(scoreBar);  
          
           // Create the honey jar  
           jar = new HoneyJar(ScreenManager.Game, this, honeyJarLocation, scoreBar);  
           ScreenManager.Game.Components.Add(jar);  
          
           // Create all the beehives and the bees  
           CreateBeehives(safeArea, jar);  
          
           // We only initialize the smoke button position here since we need access  
           // to the screen manager in order to do so (and it is null in the   
           // constructor)  
           smokeButtonPosition =  
               new Vector2(safeArea.Right -             UIConstants.SmokeButtonRightAbsoluteMargin,  
                   safeArea.Bottom - UIConstants.SmokeButtonBottomAbsoluteMargin);  
          
           // Create the smoke gun's score bar  
           int totalSmokeAmount = ConfigurationManager.ModesConfiguration[
    8. gameDifficultyLevel].TotalSmokeAmount;  
                     
           Vector2 smokeBarLocation = smokeButtonPosition +   
               new Vector2(UIConstants.SmokeButtonSize \* scaleVector.X / 8,  
               UIConstants.SmokeButtonSize \* scaleVector.Y);  
          
           smokeButtonScorebar = new ScoreBar(ScreenManager.Game, 0,
    9. totalSmokeAmount, smokeBarLocation,
    10. (int)(UIConstants.SmokeButtonSize \* scaleVector.X / 10),   
                (int)(UIConstants.SmokeButtonSize \* scaleVector.Y \* 3 / 4),
    11. Color.White,  ScoreBar.ScoreBarOrientation.Horizontal,
    12. totalSmokeAmount, this, false);  
           
            smokeTextLocation = smokeButtonPosition +  
                    new Vector2(  
                        UIConstants.SmokeButtonSize \* scaleVector.X / 2 -   
                            font16px.MeasureString(SmokeText).X \* scaleVector.X / 2,  
                        UIConstants.SmokeButtonSize \* scaleVector.Y \* 11 / 10);  
           
            ScreenManager.Game.Components.Add(smokeButtonScorebar);  
           
            // Creates the BeeKeeper  
            beeKeeper = new BeeKeeper(ScreenManager.Game, this);  
            beeKeeper.AnimationDefinitions = animations;  
            beeKeeper.ThumbStickArea =
    13. new Rectangle((int)controlstickBoundaryPosition.X,
    14. (int)controlstickBoundaryPosition.Y, controlstickBoundary.Width,
    15. controlstickBoundary.Height);  
            ScreenManager.Game.Components.Add(beeKeeper);  
           
            // Creates the vat  
            Texture2D vatTexture = ScreenManager.Game.Content.Load<Texture2D>(
    16. "Textures/vat");  
           
            Vector2 vatLocation =
    17. new Vector2(safeArea.Center.X – vatTexture.Width \* scaleVector.X / 2,
    18. safeArea.Bottom - vatTexture.Height \* scaleVector.Y -
    19. UIConstants.VatBottomMargin);  
           
            Vector2 vatScorebarLocation = vatLocation +  
                new Vector2((vatTexture.Width \* scaleVector.X –
    20. UIConstants.VatScorebarWidth) / 2, vatTexture.Height \*
    21. scaleVector.Y \* 7 / 10);  
           
            scoreBar = new ScoreBar(ScreenManager.Game, 0, 300, vatScorebarLocation,
    22. UIConstants.VatScorebarHeight, UIConstants.VatScorebarWidth,
    23. Color.White, ScoreBar.ScoreBarOrientation.Horizontal, 0, this, true);  
           
            vat = new Vat(ScreenManager.Game, this, vatTexture, vatLocation,
    24. scoreBar);  
            ScreenManager.Game.Components.Add(vat);  
           
            vatArrowPosition =  
                vatLocation + new Vector2(vatTexture.Width \* scaleVector.X / 2 -
    25. arrowTexture.Width \* scaleVector.X / 2, UIConstants.VatArrowOffset);  
           
            ScreenManager.Game.Components.Add(scoreBar);  
           
            scoreBar.DrawOrder = vat.DrawOrder + 1;  
        }

1. Modify the **CreateBeehives** method.
   * 1. C#
     2. private void CreateBeehives(Rectangle safeArea, HoneyJar jar)  
        {  
            Vector2 scaleVector = ScreenManager.SpriteBatch.ScaleVector;  
           
            // Init position parameters  
            Vector2 scorebarPosition =   
                new Vector2(beehiveTexture.Width \* scaleVector.X / 4,
     3. beehiveTexture.Height \* scaleVector.Y \* 9 / 10);  
           
            Vector2[] beehivePositions = new Vector2[5]   
            {             
                // top left  
                new Vector2(safeArea.Left + UIConstants.BeehiveLeftMargin,   
                    safeArea.Top + UIConstants.BeehiveTopMargin),  
                // top middle  
                new Vector2(safeArea.Center.X - beehiveTexture.Width \*
     4. scaleVector.X / 2, safeArea.Top + UIConstants.BeehiveTopMargin),  
                // top right  
                new Vector2(safeArea.Right - beehiveTexture.Width \* scaleVector.X -
     5. UIConstants.BeehiveRightMargin, safeArea.Top +
     6. UIConstants.BeehiveTopMargin),  
                // left  
                new Vector2(safeArea.Left + UIConstants.BeehiveLeftMargin,   
                    safeArea.Center.Y - beehiveTexture.Height \* scaleVector.Y / 2
     7. + UIConstants.BeehiveMiddleOffset),  
                // right  
                new Vector2(safeArea.Right - beehiveTexture.Width \* scaleVector.X -
     8. UIConstants.BeehiveRightMargin, safeArea.Center.Y -
     9. beehiveTexture.Height \* scaleVector.Y / 2 +
     10. UIConstants.BeehiveMiddleOffset)   
                 };  
            
             // Create the beehives  
             for (int beehiveCounter = 0; beehiveCounter < beehivePositions.Length;
     11. beehiveCounter++)  
             {  
                 ScoreBar scoreBar = new ScoreBar(ScreenManager.Game, 0, 100,
     12. beehivePositions[beehiveCounter] + scorebarPosition,
     13. (int)(beehiveTexture.Height \* scaleVector.Y / 10),  
                     (int)(beehiveTexture.Width \* scaleVector.X / 2), Color.Green,
     14. ScoreBar.ScoreBarOrientation.Horizontal, 100, this, false);  
                 ScreenManager.Game.Components.Add(scoreBar);  
            
                 Beehive beehive =  
                     new Beehive(ScreenManager.Game, this, beehiveTexture, scoreBar,
     15. beehivePositions[beehiveCounter]);  
            
                 beehive.AnimationDefinitions = animations;  
            
                 ScreenManager.Game.Components.Add(beehive);  
                 beehives.Add(beehive);  
                 scoreBar.DrawOrder = beehive.DrawOrder;  
             }  
            
             for (int beehiveIndex = 0; beehiveIndex < beehivePositions.Length;
     16. beehiveIndex++)  
             {  
                 // Create the Soldier bees  
                 for (int SoldierBeeCounter = 0;
     17. SoldierBeeCounter < amountOfSoldierBee;
     18. SoldierBeeCounter++)  
                 {  
                     SoldierBee bee = new SoldierBee(ScreenManager.Game, this,
     19. beehives[beehiveIndex]);  
                     bee.AnimationDefinitions = animations;  
                     ScreenManager.Game.Components.Add(bee);  
                     bees.Add(bee);  
                 }  
            
                 // Creates the worker bees  
                 for (int workerBeeCounter = 0; workerBeeCounter < amountOfWorkerBee;
     20. workerBeeCounter++)  
                 {  
                     WorkerBee bee = new WorkerBee(ScreenManager.Game, this,
     21. beehives[beehiveIndex]);  
                     bee.AnimationDefinitions = animations;  
                     ScreenManager.Game.Components.Add(bee);  
                     bees.Add(bee);  
                 }  
             }  
         }
2. Modify the **DrawVatHoneyArrow** method.
   * 1. C#
     2. private void DrawVatHoneyArrow()  
        {  
            // If the arrow needs to be drawn, and it is not invisible during the
     3. // current interval  
            if (drawArrow && drawArrowInInterval)  
            {  
                ScreenManager.SpriteBatch.Draw(arrowTexture, vatArrowPosition,
     4. Color.White);  
           
                if (arrowCounter == 20)  
                {  
                    drawArrowInInterval = false;  
                    arrowCounter = 0;  
                }  
                arrowCounter++;  
            }  
            else  
            {  
                if (arrowCounter == 20)  
                {  
                    drawArrowInInterval = true;  
                    arrowCounter = 0;  
                }  
                arrowCounter++;  
            }  
        }
3. Modify the **DrawSmokeButton** method.

**Note** Platform-specific code has been introduced into this method as the smoke button behaves quite differently on the Windows Phone 7 version when compared to the Windows version of the game. On the phone, the smoke button lights up when you press it. However, that is not the case in the Windows version, where the smoke button is a static asset.

* + 1. C#
    2. private void DrawSmokeButton()  
       {  
       #if WINDOWS\_PHONE  
           int buttonSize = UIConstants.SmokeButtonSize;  
          
           if (isSmokebuttonClicked)  
           {  
               ScreenManager.SpriteBatch.Draw( smokeButton, smokeButtonPosition,             new Rectangle(buttonSize, 0, buttonSize, buttonSize),             Color.White);  
           }  
           else  
           {  
               ScreenManager.SpriteBatch.Draw( smokeButton, smokeButtonPosition,
    3. new Rectangle(0, 0, buttonSize, buttonSize), Color.White);  
           }  
       #else  
           ScreenManager.SpriteBatch.Draw(smokeButton,   
               new Rectangle(  
                   (int)smokeButtonPosition.X,  
                   (int)smokeButtonPosition.Y,   
                   (int)(UIConstants.SmokeButtonSize \*
    4. ScreenManager.SpriteBatch.ScaleVector.X),  
                   (int)(UIConstants.SmokeButtonSize \*
    5. ScreenManager.SpriteBatch.ScaleVector.Y)),   
               Color.White);  
       #endif  
       }

1. Replace the **DrawLevelEndIfNecessary** method.
   * 1. C#
     2. private void DrawLevelEndIfNecessary()  
        {  
            if (isLevelEnd)  
            {  
                string stringToDisplay = string.Empty;  
           
                if (isUserWon)  
                {  
                    if (FinalScore != 0 && HighScoreScreen.IsInHighscores(FinalScore))  
                    {  
                        stringToDisplay = "It's a new\nHigh-Score!";  
                    }  
                    else  
                    {  
                        stringToDisplay = "You Win!";  
                    }  
                }  
                else  
                {  
                    stringToDisplay = "Time Is Up!";  
                }  
           
                Vector2 stringVector = font36px.MeasureString(stringToDisplay) \*
     3. ScreenManager.SpriteBatch.ScaleVector;  
           
                ScreenManager.SpriteBatch.DrawString(font36px, stringToDisplay,
     4. new Vector2(ScreenManager.GraphicsDevice.Viewport.Width / 2 -
     5. stringVector.X / 2, ScreenManager.GraphicsDevice.Viewport.Height /
     6. 2 - stringVector.Y / 2), Color.White);  
            }  
        }
2. In the **SetMotion** method, go to this line:
   * 1. C#
     2. Vector2 movementVector = leftThumbstick \* 12f;
     3. And change it to this:
     4. C#
     5. Vector2 movementVector = leftThumbstick \* 12f \*
     6. ScreenManager.SpriteBatch.ScaleVector;

# Fitting element rendering and logic to the target platform—bee component

The bee component requires special care when you modify it to perform as designed on all platforms targeted by the game. Besides the changes required to properly position it on the screen, this lab modifies the bee component’s update logic to use the passage of time, not update cycles, as a trigger for various operations.

To update the bee component

1. From the **Objects** project folder, open the **Bee.cs** file.
2. Replace the **Bee** class’s **velocityChangeCounter** field.
   * 1. C#
     2. TimeSpan velocityChangeTimer = TimeSpan.Zero;
3. In the same class, modify the **VelocityChangeInterval** property.
   * 1. C#
     2. protected virtual TimeSpan VelocityChangeInterval  
        {  
            get  
            {  
                return TimeSpan.FromMilliseconds(500);  
            }  
        }
4. Modify the **Bounds** property.
   * 1. C#
     2. public override Rectangle Bounds  
        {  
            get  
            {  
                if (texture == null)  
                {  
                    return default(Rectangle);  
                }  
                else  
                {  
                    // The bee's texture is an animation strip, so we must devide the
     3. // texture's width by three to get the bee's actual width  
                    return new Rectangle((int)position.X, (int)position.Y,   
                        (int)(texture.Width / 3 \* scaledSpriteBatch.ScaleVector.X),  
                        (int)(texture.Height \* scaledSpriteBatch.ScaleVector.Y));  
                }  
            }  
        }
5. Modify the **Update** method.
   * 1. C#
     2. public override void Update(GameTime gameTime)  
        {  
            if (!(gamePlayScreen.IsActive && gamePlayScreen.IsStarted))  
            {  
                base.Update(gameTime);  
                return;  
            }  
           
            // This method will handle the regeneration of bees that were hit by    
            // smoke  
            if (!HandleRegeneration(gameTime))  
            {  
                return;  
            }  
           
            if (!string.IsNullOrEmpty(AnimationKey))  
            {  
                AnimationDefinitions[AnimationKey].Update(gameTime, true);  
            }  
           
            // If a bee is hit by smoke, it doesn't have random movement until    
            //  regeneration  
            if (!isHitBySmoke)  
            {  
                SetRandomMovement(gameTime);  
            }  
           
            // Moving the bee according to its velocity  
            position += velocity \* scaledSpriteBatch.ScaleVector;  
           
            // If the bee is hit by smoke make it bee move faster  
            if (isHitBySmoke)  
            {  
                position += velocity \* scaledSpriteBatch.ScaleVector;  
            }  
           
            // If the bee is out of screen  
            if (position.X < 0 ||
     3. position.X > Game.GraphicsDevice.Viewport.Width - Bounds.Width ||  
                position.Y < 0 ||
     4. position.Y > Game.GraphicsDevice.Viewport.Height - Bounds.Height)  
            {  
                if (isHitBySmoke)  
                {  
                    // Reset the bee's position  
                    SetStartupPositionWithTimer();  
                }  
                else  
                {  
                    // When hit by the screen bounds, we want the bee to move  
                    // longer than usual before picking a new direction  
                    velocityChangeTimer = TimeSpan.FromMilliseconds(-160);  
                    if (position.X < Bounds.Width ||
     5. position.X >
     6. Game.GraphicsDevice.Viewport.Width - Bounds.Width)  
                    {  
                        velocity = new Vector2(velocity.X \*= -1, velocity.Y);  
                    }  
                    else  
                    {  
                        velocity = new Vector2(velocity.X, velocity.Y \*= -1);  
                    }  
                }  
            }  
           
            base.Update(gameTime);  
        }
6. Modify the **Collide** method.
   * 1. C#
     2. public void Collide(Rectangle bounds)  
        {  
            // Check if this collision is new  
            if (!isGotHit)  
            {  
                // Moves to new dircetion calculted by the "wall" that the bee
     3. // collided with.  
                velocityChangeTimer = TimeSpan.FromMilliseconds(-300);  
                if (position.X < bounds.X || position.X > bounds.X + bounds.Width)  
                {  
                    velocity = new Vector2(velocity.X \*= -1, velocity.Y);  
                }  
                else  
                {  
                    velocity = new Vector2(velocity.X, velocity.Y \*= -1);  
                }  
           
                isGotHit = true;  
            }  
        }
7. Modify the **SetRandomMovement** method.
   * 1. C#
     2. private void SetRandomMovement(GameTime gameTime)  
        {  
            velocityChangeTimer += gameTime.ElapsedGameTime;  
            if (velocityChangeTimer >= VelocityChangeInterval)  
            {  
                velocity = new Vector2(random.Next(-MaxVelocity \* 100,
     3. MaxVelocity \* 100) / 100, random.Next(-MaxVelocity \* 100,
     4. MaxVelocity \* 100) / 100);  
           
                velocityChangeTimer = TimeSpan.Zero;  
           
                if (isGotHit)  
                {  
                    isGotHit = false;  
                }  
            }  
        }

# Fitting visual element positions to the target platform—game components

* 1. Continuing the change started in the previous task, this task’s focus is on additional game components.

To have all game components positioned as designed in the game’s Windows version

1. From the **Object** project folder, open the **BeeKeeper.cs** file.
2. Modify the **BeeKeeper** class’s **Initialize** method.
   * 1. C#

public override void Initialize()  
{  
    // Initialize the animation   
    AnimationDefinitions[LegAnimationKey].PlayFromFrameIndex(0);  
    AnimationDefinitions[BodyAnimationKey].PlayFromFrameIndex(0);  
    AnimationDefinitions[SmokeAnimationKey].PlayFromFrameIndex(0);  
    AnimationDefinitions[ShootingAnimationKey].PlayFromFrameIndex(0);  
    AnimationDefinitions[

        BeekeeperCollectingHoneyAnimationKey].PlayFromFrameIndex(0);  
    AnimationDefinitions[

        BeekeeperDesposingHoneyAnimationKey].PlayFromFrameIndex(0);  
   
    Point bodyAnimationFrameSize =

        AnimationDefinitions[LegAnimationKey].frameSize;  
    bodySize =

        new Vector2(bodyAnimationFrameSize.X, bodyAnimationFrameSize.Y) \*

            scaledSpriteBatch.ScaleVector;  
   
    isStung = false;  
    stungDuration = TimeSpan.FromSeconds(1);  
    flashingDuration = TimeSpan.FromSeconds(2);  
   
    availableSmokePuffs = new Stack<SmokePuff>(MaxSmokePuffs);  
    FiredSmokePuffs = new Queue<SmokePuff>(MaxSmokePuffs);  
   
    base.Initialize();  
}

1. From the Objects project folder, open the **HoneyJar.cs** file.
2. Modify the **HoneyJar** class’s **Draw** method.
   * 1. C#
     2. public override void Draw(GameTime gameTime)  
        {  
            if (!gamePlayScreen.IsActive)  
            {  
                base.Draw(gameTime);  
                return;  
            }  
           
            scaledSpriteBatch.Begin();  
            scaledSpriteBatch.Draw(texture, position, Color.White);  
            scaledSpriteBatch.DrawString(font16px, HoneyText, position +  
                new Vector2(Bounds.Width / 2 - honeyTextSize.X / 2,
     3. Bounds.Height \* 4 / 3), Color.White);  
            scaledSpriteBatch.End();  
           
            base.Draw(gameTime);  
        }
3. Modify the **LoadContent** method.
   * 1. C#
     2. protected override void LoadContent()  
        {  
            font16px = Game.Content.Load<SpriteFont>("Fonts/GameScreenFont16px");  
            texture = Game.Content.Load<Texture2D>("Textures/HoneyJar");  
            honeyTextSize =
     3. font16px.MeasureString(HoneyText) \* scaledSpriteBatch.ScaleVector;  
           
            base.LoadContent();  
        }
4. From the **Objects** projectfolder,open the **Vat.cs** file.
5. Modify the **Vat** class’s **Draw** method.
   * 1. C#
     2. public override void Draw(GameTime gameTime)  
        {  
            if (!gamePlayScreen.IsActive)  
            {  
                base.Draw(gameTime);  
                return;  
            }  
           
            // Draws the texture  
            scaledSpriteBatch.Begin();  
            scaledSpriteBatch.Draw(texture, position, Color.White);  
           
            // Draws the "time left" text  
            scaledSpriteBatch.DrawString(font16px, TimeLeftString,  
                position + new Vector2(Bounds.Width / 2 - timeleftStringSize.X / 2,
     3. timeleftStringSize.Y - 8), Color.White, 0, Vector2.Zero, 0,
     4. SpriteEffects.None, 2f);  
           
            // Draws how much time is left  
            timeDigStringSize = font36px.MeasureString(timeLeftString) \*
     5. scaledSpriteBatch.ScaleVector;  
            Color colorToDraw = Color.White;  
           
            if (timeLeft.Minutes == 0 &&
     6. (timeLeft.Seconds == 30 || timeLeft.Seconds <= 10))  
            {  
                colorToDraw = Color.Red;  
            }  
           
            scaledSpriteBatch.DrawString(font36px, timeLeftString, position +
     7. new Vector2(Bounds.Width / 2 - timeDigStringSize.X / 2,  
                Bounds.Height / 2 - timeDigStringSize.Y / 2), colorToDraw);  
           
            // Draws the "full" and "empty" strings  
            scaledSpriteBatch.DrawString(font14px, EmptyString,  
                new Vector2(position.X, position.Y + Bounds.Height -
     8. emptyStringSize.Y), Color.White);  
           
            scaledSpriteBatch.DrawString(font14px, FullString,  
                new Vector2(position.X + Bounds.Width - fullStringSize.X,  
                            position.Y + Bounds.Height - emptyStringSize.Y),
     9. Color.White);  
           
            scaledSpriteBatch.End();  
           
            base.Draw(gameTime);  
        }
6. Modify the **LoadContent** method.
   * 1. C#
     2. protected override void LoadContent()  
        {  
            font14px = Game.Content.Load<SpriteFont>("Fonts/GameScreenFont14px");  
            font16px = Game.Content.Load<SpriteFont>("Fonts/GameScreenFont16px");  
            font36px = Game.Content.Load<SpriteFont>("Fonts/GameScreenFont36px");  
           
            fullStringSize = font14px.MeasureString(FullString) \*
     3. scaledSpriteBatch.ScaleVector;  
            emptyStringSize = font14px.MeasureString(EmptyString) \*
     4. scaledSpriteBatch.ScaleVector;  
            timeleftStringSize = font16px.MeasureString(TimeLeftString) \*
     5. scaledSpriteBatch.ScaleVector;  
           
            base.LoadContent();  
        }
7. From the **Objects** project folder, open the **Beehive.cs** file.
8. Modify the **Beehive** class’s **Bounds** property.
   * 1. C#
     2. public override Rectangle Bounds  
        {  
            get  
            {  
                Rectangle baseBounds = base.Bounds;  
                int widthMargin = baseBounds.Width / 10;  
                int width = baseBounds.Width - widthMargin;  
                int height = baseBounds.Height / 3;  
           
                return new Rectangle(baseBounds.X + widthMargin,
     3. baseBounds.Y + height, width - widthMargin, height);  
            }  
        }
9. From the **Objects** project folder,open the **SmokePuff.cs** file.
10. Modify the **SmokePuff** class’s **CentralCollisionArea** property.
    * 1. C#
      2. public override Rectangle CentralCollisionArea  
         {  
             get  
             {  
                 int boundsWidth = (int)(texture.Width \* spreadFactor \* 1.5f \*
      3. scaledSpriteBatch.ScaleVector.X);  
                 int boundsHeight = (int)(texture.Height \* spreadFactor \* 1.5f \*
      4. scaledSpriteBatch.ScaleVector.Y);  
            
                 return new Rectangle((int)position.X - boundsWidth / 4,
      5. (int)position.Y - boundsHeight / 4, boundsWidth, boundsHeight);  
             }  
         }

# Fitting visual element positions to the target platform—additional screens

* 1. This task finalizes the positioning changes required to have the game look good on both the Windows Phone 7 and on a Windows-enabled computer. This task focuses on the screens that you have not seen yet.

To finalize support for high-definition graphics in the game

1. From the **Screens** project folder, open the **BackgroundScreen.cs** file.
2. Change the **BackgroundScreen** class’s **Draw** method.
   * 1. C#
     2. public override void Draw(GameTime gameTime)  
        {  
            SpriteBatch spriteBatch = ScreenManager.SpriteBatch;  
           
            spriteBatch.Begin();  
           
            // Draw background  
            spriteBatch.Draw(background, ScreenManager.GraphicsDevice.Viewport.Bounds,
     3. Color.White \* TransitionAlpha);  
           
            spriteBatch.End();  
        }
3. From the **Screens** project folder, open the **HighScoreScreen.cs** file.
4. Examine the **HighScoreScreen** class, and then modify two of the class’s fields, **highscorePlaces** and **highScore**.

**Note** When you run on a Windows computer, these changes are to manage the additional vertical space available by adding more high-score lines to the screen.

* + 1. C#

#if WINDOWS\_PHONE  
const int highscorePlaces = 5;  
#else  
const int highscorePlaces = 7;  
#endif  
public static List<KeyValuePair<string, int>> highScore = new List<KeyValuePair<string, int>>(highscorePlaces)  
{  
    new KeyValuePair<string,int>  
        ("Jasper",55000),  
    new KeyValuePair<string,int>  
        ("Ellen",52750),  
    new KeyValuePair<string,int>  
        ("Terry",52200),  
    new KeyValuePair<string,int>  
        ("Lori",50200),  
    new KeyValuePair<string,int>  
        ("Michael",50750),  
#if !WINDOWS\_PHONE  
    new KeyValuePair<string,int>  
        ("Frodo",49550),  
    new KeyValuePair<string,int>  
        ("Chuck",46750),  
#endif  
};

1. Modify the **Draw** method.
   * 1. C#
     2. public override void Draw(GameTime gameTime)  
        {  
            if (HighscoreLoaded == false)  
            {  
                base.Draw(gameTime);  
                return;  
            }  
           
            ScreenManager.SpriteBatch.Begin();  
           
            // Draw the highscores table  
        #if WINDOWS\_PHONE  
            for (int i = 0; i < highScore.Count; i++)  
            {  
                if (!string.IsNullOrEmpty(highScore[i].Key))  
                {  
                    // Draw place number  
                    ScreenManager.SpriteBatch.DrawString(highScoreFont,
     3. GetPlaceString(i), new Vector2(safeArea.Left +
     4. UIConstants.HighScorePlaceLeftMargin, safeArea.Top + i \*
     5. UIConstants.HighScoreVerticalJump +
     6. UIConstants.HighScoreTopMargin), Color.Black);  
           
                    // Draw Name  
                    ScreenManager.SpriteBatch.DrawString(highScoreFont,
     7. highScore[i].Key, Vector2(safeArea.Left +
     8. UIConstants.HighScoreNameLeftMargin, safeArea.Top + i \*
     9. UIConstants.HighScoreVerticalJump +
     10. UIConstants.HighScoreTopMargin), Color.DarkRed);  
            
                     // Draw score  
                     ScreenManager.SpriteBatch.DrawString(highScoreFont,
     11. highScore[i].Value.ToString(), new Vector2(safeArea.Left +
     12. UIConstants.HighScoreScoreLeftMargin, safeArea.Top + i \*
     13. UIConstants.HighScoreVerticalJump +
     14. UIConstants.HighScoreTopMargin), Color.Yellow);  
                 }  
             }  
         #else  
             float verticalPosition = UIConstants.HighScoreTopMargin;  
            
             for (int i = 0; i < highScore.Count; i++)  
             {  
                 if (!string.IsNullOrEmpty(highScore[i].Key))  
                 {  
                     // Draw place number  
                     ScreenManager.SpriteBatch.DrawString( highScoreFont,
     15. GetPlaceString(i), new Vector2(safeArea.Left +
     16. UIConstants.HighScorePlaceLeftMargin, safeArea.Top +
     17. verticalPosition), Color.Black);  
            
                     // Draw Name  
                     ScreenManager.SpriteBatch.DrawString(highScoreFont,
     18. highScore[i].Key,  new Vector2(safeArea.Left +
     19. UIConstants.HighScoreNameLeftMargin, safeArea.Top +
     20. verticalPosition), Color.DarkRed);  
            
                     // Draw score  
                     ScreenManager.SpriteBatch.DrawString(highScoreFont,
     21. highScore[i].Value.ToString(), new Vector2(safeArea.Left +
     22. UIConstants.HighScoreScoreLeftMargin, safeArea.Top +
     23. verticalPosition), Color.Yellow);  
                 }  
            
                 // Odd and even lines have different height. Remember that "i" is an
     24. // index so even i's are actually odd lines.  
                 if (i % 2 == 0)  
                 {  
                     verticalPosition += UIConstants.HighScoreOddVerticalJump;  
                 }  
                 else  
                 {  
                     verticalPosition += UIConstants.HighScoreEvenVerticalJump;  
                 }  
             }  
         #endif  
            
             ScreenManager.SpriteBatch.End();  
            
             base.Draw(gameTime);  
         }
2. Modify the **InitializeMapping** method.
   * 1. C#
     2. private void InitializeMapping()  
        {  
            numberPlaceMapping.Add(0, "1ST");  
            numberPlaceMapping.Add(1, "2ND");  
            numberPlaceMapping.Add(2, "3RD");  
            numberPlaceMapping.Add(3, "4TH");  
            numberPlaceMapping.Add(4, "5TH");  
            numberPlaceMapping.Add(5, "6TH");  
            numberPlaceMapping.Add(6, "7TH");  
        }
3. From the **Screens** project folder, open the **LevelOverScreen.cs** file.
4. Add a field to the **LevelOverScreen** class.
   * 1. C#
     2. Rectangle safeArea;
5. Replace the **LoadContent** method.
   * 1. C#
     2. public override void LoadContent()  
        {  
            if (difficultyMode.HasValue)  
            {  
                gameplayScreen = new GameplayScreen(difficultyMode.Value);  
                gameplayScreen.ScreenManager = ScreenManager;  
            }  
            font36px = ScreenManager.Game.Content.Load<SpriteFont>(
     3. "Fonts/GameScreenFont36px");  
            font16px = ScreenManager.Game.Content.Load<SpriteFont>(
     4. "Fonts/GameScreenFont16px");  
            textSize = font36px.MeasureString(text);  
            safeArea = SafeArea;  
           
            base.LoadContent();  
        }
6. Modify the **Draw** method.
   * 1. C#
     2. public override void Draw(GameTime gameTime)  
        {  
            SpriteBatch spriteBatch = ScreenManager.SpriteBatch;  
           
            spriteBatch.Begin();  
           
            // Draw the footer text  
             if (difficultyMode.HasValue)  
            {  
        #if WINDOWS\_PHONE  
                string actionText = "Touch to start next level";  
        #else  
                string actionText = "Press space to start next level";  
        #endif  
           
                spriteBatch.DrawString(font16px, actionText,  
                    new Vector2(ScreenManager.GraphicsDevice.Viewport.Width / 2 -  
                        font16px.MeasureString(actionText).X / 2,  
                        safeArea.Bottom - font16px.MeasureString(actionText).Y - 4),  
                    Color.Black);  
            }  
            else  
            {  
        #if WINDOWS\_PHONE  
                string actionText = "Touch to end game";  
        #else  
                string actionText = "Press space to end game";  
        #endif  
                spriteBatch.DrawString(font16px, actionText,  
                    new Vector2(safeArea.Left + safeArea.Width / 2 -
     3. font16px.MeasureString(actionText).X / 2,  
                        safeArea.Top + safeArea.Height -
     4. font16px.MeasureString(actionText).Y - 4),  
                    Color.Black);  
            }  
           
            spriteBatch.End();  
        }
7. From the **Screens** project folder, open the **MainMenuScreen.cs** file.
8. Modify the **MainMenuScreen** class’s constructor and **LoadContent** method.

**Note** This repositions the main menu entries so they are placed relative to the screen center instead of at a set location.

* + 1. C#

public MainMenuScreen()  
    : base("")  
{  
}  
   
public override void LoadContent()  
{  
    // Create our menu entries.  
    MenuEntry startGameMenuEntry = new MenuEntry("Start");  
    MenuEntry exitMenuEntry = new MenuEntry("Exit");  
   
    // Calculate menu positions - we do this here since we want the screen  
    // manager to be available  
    int quarterViewportWidth =

        ScreenManager.GraphicsDevice.Viewport.Width / 4;  
    int menuEntryHeight = SafeArea.Bottom -  
        ScreenManager.ButtonBackground.Height \* 2;  
    startGameMenuEntry.Position = new Vector2(quarterViewportWidth -  
        ScreenManager.ButtonBackground.Width / 2, menuEntryHeight);  
    exitMenuEntry.Position = new Vector2(3 \* quarterViewportWidth -  
        ScreenManager.ButtonBackground.Width / 2, menuEntryHeight);  
   
    // Hook up menu event handlers.  
    startGameMenuEntry.Selected += StartGameMenuEntrySelected;  
    exitMenuEntry.Selected += OnCancel;  
   
    // Add entries to the menu.  
    MenuEntries.Add(startGameMenuEntry);  
    MenuEntries.Add(exitMenuEntry);              
   
    AudioManager.LoadSounds();  
    AudioManager.LoadMusic();  
   
    AudioManager.PlayMusic("MenuMusic\_Loop");  
   
    base.LoadContent();  
}

1. From the **Screens** project folder, open the **PauseScreen.cs** file.
2. Modify the **PauseScreen** class’s constructor, and override the **LoadContent** method.
   * 1. C#
     2. public PauseScreen() : base(string.Empty)  
        {  
            IsPopup = true;  
        }  
           
        /// <summary>  
        /// Load screen resources  
        /// </summary>  
        public override void LoadContent()  
        {  
            AudioManager.PlaySound("menu");  
           
            MenuEntry returnGameMenuEntry = new MenuEntry("Resume");  
            returnGameMenuEntry.Scale = 0.7f;  
            MenuEntry exitMenuEntry = new MenuEntry("Exit");  
           
            // Calculate menu positions - we do this here since we want the screen  
            // manager to be available  
            int quarterViewportWidth =
     3. ScreenManager.GraphicsDevice.Viewport.Width / 4;  
            int menuEntryHeight = SafeArea.Bottom -  
                ScreenManager.ButtonBackground.Height \* 2;  
            returnGameMenuEntry.Position = new Vector2(quarterViewportWidth -  
                ScreenManager.ButtonBackground.Width / 2, menuEntryHeight);  
            exitMenuEntry.Position = new Vector2(3 \* quarterViewportWidth -  
                ScreenManager.ButtonBackground.Width / 2, menuEntryHeight);  
           
            // Hook up menu event handlers.  
            returnGameMenuEntry.Selected += ReturnGameMenuEntrySelected;  
            exitMenuEntry.Selected += OnCancel;  
           
            //// Add entries to the menu.  
            MenuEntries.Add(returnGameMenuEntry);  
            MenuEntries.Add(exitMenuEntry);  
           
            base.LoadContent();  
        }
3. Compile and run the game.

The game should now run as expected on Windows.

1. Open the starter solution (or the copy you made of it), and add the **UIConstants.cs** file created during this exercise to the **Misc** project folder.
2. Run the altered game on the phone by using the starter solution (or its copy, if you made one).

Notice how the same code now runs on both Windows and Windows Phone 7.

Exercise 2 – Xbox 360 Version

* 1. In the previous exercise, you altered the Honeycomb Rush game for Windows Phone 7 so you can run the same code on both the Windows Phone 7 and on Windows-enabled computers, while maintaining feature parity. This exercise focuses on having the code run on Xbox 360.

# Creating a copy of the Windows project

* 1. First, you need to create an Xbox 360 copy of the Windows game project created in the previous exercise. You will use the new project to compile and run the game on Xbox 360.
     1. **Note** While we could have created Xbox 360 project by creating a copy of the Windows Phone 7 project, the Windows project is a more natural choice as the Xbox 360 project requires high-definition assets as well.

To create the new project

1. Create a copy of the solution created in the first task of the previous exercise, name it **HoneycombRush - Xbox 360**, and then open the new solution.

**Note** The solution created during the previous exercise also is available in the lab installation folder under **Sources\ Ex1 - Windows Version**.

1. In **Solution Explorer**, right-c**l**ick the **HoneycombRush - Windows** project, and then from the context menu, click **Create Copy of Project for Xbox 360**.
2. In **Solution Explorer**, review the solution structure.

A new project named **Xbox 360 Copy of HoneycombRush - Windows** should now be part of the solution.

1. Rename the project **Xbox 360 Copy of HoneycombRush - Windows** to **HoneycombRush – Xbox 360**.
2. Remove the **HoneycombRush - Windows** project from the solution.
3. From the **Build** menu, select **Configuration Manager,** and then in the **Configuration Manager** dialog box, select **<Edit…>** from the **Active solution platform** combo box.
4. From the list on the left, select the **Mixed Platforms**, **Any CPU,** and **x86** entries, and then click **Remove**. Close all dialog boxes after you remove both entries.
5. Currently, the game does not compile, as the user interface (UI) positioning constants for this new platform have not been defined. For more information, see “Fitting visual element positions to the target platform – the gameplay screen” in the previous exercise.

# Handling gamepad input

* 1. When you implemented keyboard support in the previous exercise, you ensured that the code would not be aware of the input method. This makes the task of adding support for the Xbox’s gamepad much simpler.

To add gamepad support to the game

1. From the **ScreenManager** project folder, open the **InputState.cs** file.
2. Add the **IsButtonDown** and **LeftThumbstick** methods to the **InputState** class.

**Note** These methods indicate whether a gamepad button is down. They enable you to know the state of the gamepad’s left thumbstick.

* + 1. C#

public bool IsButtonDown(Buttons button, PlayerIndex? controllingPlayer,  
                                                out PlayerIndex playerIndex)  
{  
    if (controllingPlayer.HasValue)  
    {  
        // Read input from the specified player.  
        playerIndex = controllingPlayer.Value;  
   
        int i = (int)playerIndex;  
   
        return (CurrentGamePadStates[i].IsButtonDown(button));  
    }  
    else  
    {  
        // Accept input from any player.  
        return (IsButtonDown(button, PlayerIndex.One, out playerIndex) ||  
                IsButtonDown(button, PlayerIndex.Two, out playerIndex) ||  
                IsButtonDown(button, PlayerIndex.Three, out playerIndex) ||  
                IsButtonDown(button, PlayerIndex.Four, out playerIndex));  
    }  
}  
   
public Vector2 LeftThumbstick(PlayerIndex? controllingPlayer,  
                                                out PlayerIndex playerIndex)  
{  
    if (controllingPlayer.HasValue)  
    {  
        // Read input from the specified player.  
        playerIndex = controllingPlayer.Value;  
   
        int i = (int)playerIndex;  
   
        return CurrentGamePadStates[i].ThumbSticks.Left;  
    }  
    else  
    {  
        Vector2 result = LeftThumbstick(PlayerIndex.One, out playerIndex);  
        if (result != Vector2.Zero)  
        {  
            return result;  
        }  
        result = LeftThumbstick(PlayerIndex.Two, out playerIndex);  
        if (result != Vector2.Zero)  
        {  
            return result;  
        }  
        result = LeftThumbstick(PlayerIndex.Three, out playerIndex);  
        if (result != Vector2.Zero)  
        {  
            return result;  
        }  
        result = LeftThumbstick(PlayerIndex.Four, out playerIndex);  
   
        return result;  
    }  
}

1. Modify the **IsPauseGame** method.
   * 1. C#
     2. public bool IsPauseGame(PlayerIndex? controllingPlayer)  
        {  
            PlayerIndex playerIndex;  
           
            return
     3. IsNewKeyPress(Keys.Escape, controllingPlayer, out playerIndex) ||  
                IsNewButtonPress(Buttons.Back, controllingPlayer, out playerIndex) ||  
                IsNewButtonPress(Buttons.Start, controllingPlayer, out playerIndex) ||  
                IsNewButtonPress(Buttons.BigButton, controllingPlayer, out
     4. playerIndex);  
        }
2. From the **ScreenManager** project folder, open the **MenuScreen.cs** file.
3. Modify the **MenuScreen** class’s **HandleInput** method.

**Note** The method is hardly changed. The only change is that the gamepad’s “A” button now selects the current menu entry.

* + 1. C#

public override void HandleInput(GameTime gameTime, InputState input)  
{  
    // we cancel the current menu screen if the user presses the back button  
    PlayerIndex player;  
    if (input.IsNewButtonPress(Buttons.Back, ControllingPlayer, out player))  
    {  
        OnCancel(player);  
    }  
   
    // Take care of Keyboard/gamepad input  
    if (input.IsMenuUp(ControllingPlayer))  
    {  
        selectedEntry--;  
   
        if (selectedEntry < 0)  
            selectedEntry = menuEntries.Count - 1;  
    }  
    else if (input.IsMenuDown(ControllingPlayer))  
    {  
        selectedEntry++;  
   
        if (selectedEntry >= menuEntries.Count)  
            selectedEntry = 0;  
    }  
    else if (input.IsNewKeyPress(Keys.Enter, ControllingPlayer, out player) ||  
        input.IsNewKeyPress(Keys.Space, ControllingPlayer, out player))  
    {  
        OnSelectEntry(selectedEntry, player);  
    }  
    else if (input.IsNewButtonPress(Buttons.A, ControllingPlayer, out player))  
    {  
        OnSelectEntry(selectedEntry, player);  
    }  
    // look for any taps that occurred and select any entries that were tapped  
    foreach (GestureSample gesture in input.Gestures)  
    {  
        if (gesture.GestureType == GestureType.Tap)  
        {  
            // convert the position to a Point that we can test against a

            // Rectangle  
            Point tapLocation = new Point((int)gesture.Position.X,

                (int)gesture.Position.Y);  
   
            // iterate the entries to see if any were tapped  
            for (int i = 0; i < menuEntries.Count; i++)  
            {  
                MenuEntry menuEntry = menuEntries[i];  
   
                if (GetMenuEntryHitBounds(menuEntry).Contains(tapLocation))  
                {  
                    // Select the entry. since gestures are only available on

                    // Windows Phone, we can safely pass PlayerIndex.One to                     // all entries since there is only one player on Windows                     // Phone.  
                    OnSelectEntry(i, PlayerIndex.One);  
                }  
            }  
        }  
    }  
}

1. From the **Screens** project folder, open the **GameplayScreen.cs** file.
2. Modify the **GameplayScreen** class’s **HandleInput** method.
   * 1. **Note** Compare the new method with the previous method to understand the exact scope of the changes you made to the method.
     2. C#
     3. public override void HandleInput(GameTime gameTime, InputState input)  
        {  
            if (IsActive)  
            {  
                if (input == null)  
                {  
                    throw new ArgumentNullException("input");  
                }  
           
                if (input.IsPauseGame(null))  
                {  
                    PauseCurrentGame();  
                }  
            }  
           
            if (input.TouchState.Count > 0)  
            {  
                foreach (TouchLocation touch in input.TouchState)  
                {  
                    lastTouchPosition = touch.Position;  
                }  
            }  
           
            isSmokebuttonClicked = false;  
           
            PlayerIndex player;  
           
            VirtualThumbsticks.Update(input);  
           
            if (input.Gestures.Count > 0)  
            {  
                if (input.Gestures[0].GestureType == GestureType.DoubleTap)  
                {  
                    showDebugInfo = !showDebugInfo;  
                }  
            }  
           
            if (isLevelEnd)  
            {  
                if (input.Gestures.Count > 0)  
                {  
                    if (input.Gestures[0].GestureType == GestureType.Tap)  
                    {  
                        userInputToExit = true;  
                    }  
                }  
                else if (input.IsNewButtonPress(Buttons.A, ControllingPlayer,  
                    out player))  
                {  
                    userInputToExit = true;  
                }  
           
                if (input.IsNewKeyPress(Keys.Enter, ControllingPlayer, out player) ||  
                    input.IsNewKeyPress(Keys.Space, ControllingPlayer, out player))  
                {  
                    userInputToExit = true;  
                }  
            }  
           
            if (!IsStarted)  
            {  
                return;  
            }  
           
            // If there was any touch  
            if (VirtualThumbsticks.RightThumbstickCenter.HasValue)  
            {  
                // Button Bounds  
                Rectangle buttonRectangle = new Rectangle((int)smokeButtonPosition.X,
     4. (int)smokeButtonPosition.Y, smokeButton.Width / 2,
     5. smokeButton.Height);  
           
                // Touch Bounds  
                Rectangle touchRectangle = new Rectangle(
     6. (int)VirtualThumbsticks.RightThumbstickCenter.Value.X,
     7. (int)VirtualThumbsticks.RightThumbstickCenter.Value.Y,
     8. 1, 1);
     9. // If the touch is in the button  
                if (buttonRectangle.Contains(touchRectangle) &&
     10. !beeKeeper.IsCollectingHoney && !beeKeeper.IsStung)  
                 {  
                     isSmokebuttonClicked = true;  
                 }  
             }  
            
             if (input.IsNewButtonPress(Buttons.Y, ControllingPlayer, out player))  
             {  
                 showDebugInfo = !showDebugInfo;  
             }  
             else if (input.IsButtonDown(Buttons.A, ControllingPlayer, out player) &&
     11. !beeKeeper.IsCollectingHoney && !beeKeeper.IsStung)  
             {  
                 isSmokebuttonClicked = true;  
             }  
            
             // Handle keyboard  
             if (input.IsNewKeyPress(Keys.Y, ControllingPlayer, out player))  
             {  
                 showDebugInfo = !showDebugInfo;  
             }  
             if (input.IsKeyDown(Keys.Space, ControllingPlayer, out player) &&
     12. !beeKeeper.IsCollectingHoney &&  
                 !beeKeeper.IsStung)  
             {  
                 isSmokebuttonClicked = true;  
             }  
            
             movementVector = SetMotion(input);  
             beeKeeper.SetDirection(movementVector);  
         }
3. Replace the **SetMotion** method.
   * 1. C#
     2. private Vector2 SetMotion(InputState input)  
        {  
            // Calculate the beekeeper location, if allow moving  
            Rectangle safeArea = SafeArea;  
           
            PlayerIndex playerIndex;  
           
            Vector2 leftThumbstick = VirtualThumbsticks.LeftThumbstick;  
           
            // If we have no phone directional input, try the gamepad  
            if (leftThumbstick == Vector2.Zero)  
            {  
                leftThumbstick = input.LeftThumbstick(ControllingPlayer,
     3. out playerIndex) \* new Vector2(1, -1);  
            }  
           
            // Move on to keyboard input if we still have nothing  
            if (leftThumbstick == Vector2.Zero)  
            {  
                float vecX = 0;  
                float vecY = 0;  
           
                if (input.IsKeyDown(Keys.Left, ControllingPlayer, out playerIndex))  
                {  
                    vecX--;  
                }  
                if (input.IsKeyDown(Keys.Right, ControllingPlayer, out playerIndex))  
                {  
                    vecX++;  
                }  
                if (input.IsKeyDown(Keys.Up, ControllingPlayer, out playerIndex))  
                {  
                    vecY--;  
                }  
                if (input.IsKeyDown(Keys.Down, ControllingPlayer, out playerIndex))  
                {  
                    vecY++;  
                }  
           
                leftThumbstick = new Vector2(vecX, vecY);  
            }  
           
            Vector2 movementVector = leftThumbstick \* 12f;  
           
            Rectangle futureBounds = beeKeeper.Bounds;  
            futureBounds.X += (int)movementVector.X;  
            futureBounds.Y += (int)movementVector.Y;  
           
            if (futureBounds.Left <= safeArea.Left ||
     4. futureBounds.Right >= safeArea.Right)  
            {  
                movementVector.X = 0;  
            }  
            if (futureBounds.Top <= safeArea.Top ||
     5. futureBounds.Bottom >= safeArea.Bottom)  
            {  
                movementVector.Y = 0;  
            }  
           
            if (movementVector == Vector2.Zero)  
            {  
                IsInMotion = false;  
                beeKeeper.SetMovement(Vector2.Zero);  
            }  
            else  
            {  
                Vector2 beekeeperCalculatedPosition =  
                    new Vector2(beeKeeper.CentralCollisionArea.X,
     6. beeKeeper.CentralCollisionArea.Y) + movementVector;  
           
                if (!CheckBeehiveCollision(beekeeperCalculatedPosition))  
                {  
                    beeKeeper.SetMovement(movementVector);  
                    IsInMotion = true;  
                }  
            }  
           
            return movementVector;  
        }
4. From the **Screens** project folder, open the **HighScoreScreen.cs** file.
5. Modify the **HighScoreScreen** class’s **HandleInput** method.
   * 1. C#
     2. public override void HandleInput(GameTime gameTime, InputState input)  
        {  
            if (input == null)  
            {  
                throw new ArgumentNullException("input");  
            }  
           
            if (input.IsPauseGame(null))  
            {  
                Exit();  
            }  
           
            // Return to the main menu when a tap gesture is recognized  
            if (input.Gestures.Count > 0)  
            {  
                GestureSample sample = input.Gestures[0];  
                if (sample.GestureType == GestureType.Tap)  
                {  
                    Exit();  
           
                    input.Gestures.Clear();  
                }  
            }  
            // Handle gamepad input  
            PlayerIndex player;  
            if (input.IsNewButtonPress(Buttons.A, ControllingPlayer, out player))  
            {  
                Exit();  
            }  
            // Handle keyboard input  
            if (input.IsNewKeyPress(Keys.Enter, ControllingPlayer, out player) ||  
                input.IsNewKeyPress(Keys.Space, ControllingPlayer, out player))  
            {  
                Exit();  
            }  
        }
6. From the **Screens** project folder, open the **LevelOverScreen.cs** file.
7. Modify the **LevelOverScreen** class’s **HandleInput** method.
   * 1. C#
     2. public override void HandleInput(GameTime gameTime, InputState input)  
        {  
            if (input == null)  
            {  
                throw new ArgumentNullException("input");  
            }  
           
            PlayerIndex player;  
           
            // Return to the main menu when a tap gesture is recognized  
            if (input.Gestures.Count > 0)  
            {  
                GestureSample sample = input.Gestures[0];  
                if (sample.GestureType == GestureType.Tap)  
                {  
                    StartNewLevelOrExit(input);  
                    input.Gestures.Clear();  
                }  
            }  
            // Handle gamepad  
            else if (input.IsNewButtonPress(Buttons.A, ControllingPlayer, out player))  
            {  
                StartNewLevelOrExit(input);  
            }  
            // Handle keyboard  
            else if (input.IsNewKeyPress(Keys.Enter, ControllingPlayer, out player) ||  
                input.IsNewKeyPress(Keys.Space, ControllingPlayer, out player))  
            {  
                StartNewLevelOrExit(input);  
            }  
           
            base.HandleInput(gameTime, input);  
        }
8. From the **Screens** project folder, open the **LoadingAndInstructionScreen.cs** file.
9. Modify the **LoadingAndInstructionScreen** class’s **HandleInput** method.
   * 1. C#
     2. public override void HandleInput(GameTime gameTime, InputState input)  
        {  
            if (!isLoading)  
            {  
                PlayerIndex player;  
           
                // Handle touch input  
                if (input.Gestures.Count > 0)  
                {  
                    if (input.Gestures[0].GestureType == GestureType.Tap)  
                    {  
                        LoadResources();  
                    }  
                }  
                else if (input.IsNewButtonPress(Buttons.Back, ControllingPlayer,
     3. out player) ||  
                    input.IsNewKeyPress(Keys.Escape, ControllingPlayer, out player))  
                {  
                    foreach (GameScreen screen in ScreenManager.GetScreens())  
                    {  
                        screen.ExitScreen();  
                    }  
           
                    ScreenManager.AddScreen(new BackgroundScreen("titleScreen"),
     4. null);  
                    ScreenManager.AddScreen(new MainMenuScreen(), PlayerIndex.One);  
                }  
                else if (
     5. input.IsNewKeyPress(Keys.Enter, ControllingPlayer, out player) ||  
                    input.IsNewKeyPress(Keys.Space, ControllingPlayer, out player) ||  
                    input.IsNewButtonPress(Buttons.A, ControllingPlayer, out player))  
                {  
                    LoadResources();  
                }  
            }  
           
            base.HandleInput(gameTime, input);  
        }
10. From the **Screens** project folder, open the **PauseScreen.cs** file, and add the following **using** statement:
    * 1. C#
      2. using Microsoft.Xna.Framework.Input;
11. In the same file, override the **HandleInput** in the **PauseScreen** class.
    * 1. C#
      2. public override void HandleInput(GameTime gameTime, InputState input)  
         {  
             PlayerIndex player;  
             if (input.IsNewButtonPress(Buttons.Start, ControllingPlayer, out player))  
             {  
                 ReturnGameMenuEntrySelected(this, new EventArgs());  
             }  
            
             base.HandleInput(gameTime, input);  
         }

# Fitting visual element positions to the target platform

* 1. The previous task modified the game to respond to gamepad input. This task is to make sure the game renders properly on Xbox 360. The Xbox 360 version presents an additional challenge as you try to connect it to different displays, some of which, such as certain television sets, will crop the image they display. This means that game-critical elements placed at the screen edges may not be visible to some players.
  2. Recall the **SafeArea** property you added to the **GameScreen** class during the previous exercise (see “Handling keyboard input in the gameplay screen”). This property contains the bounds of the *Title Safe Area*, an area that is supposed to be visible on all possible displays, and which equals the entire viewport on both Windows and the Windows Phone 7.
     1. **Note** For more information about the issues mentioned previously, see the Safe Area sample at <http://create.msdn.com/en-US/education/catalog/sample/safe_area>.
  3. Your game rendering code already uses the **SafeArea** property to position the game’s elements. Therefore, you just need to define specific UI positioning constants for Xbox 360.
     1. **Note** Due to the additional constraints posed by fitting the game inside the safe area, the game’s layout will be slightly different when you run it on Xbox 360.

To have the game display as designed on Xbox 360

1. From the **Misc** project folder, open the **UIConstants.cs** file, and locate the following segment of code:
   * 1. C#
     2. public const float HighScoreScoreLeftMargin = 545;  
            public const float HighScoreTopMargin = 86;  
            public const float HighScoreVerticalJump = 72;  
        #else  
            /// <summary>  
            /// Updates all constants returned by the class according to a specified
     3. /// scale factor. Always call this method at least once before retrieving
     4. /// constants from the class.  
            /// </summary>  
            /// <param name="scaleVector">Vector depicting the scale used.</param>  
            public static void SetScale(Vector2 scaleVector)  
            {  
                BeehiveLeftMargin = 130 \* scaleVector.X;
2. Add the following block of code just above the **#else** line in the previous step:
   * 1. C#
     2. #elseif XBOX  
            public const float BeehiveLeftMargin = 0;  
            public const float BeehiveRightMargin = 0;  
            public const float BeehiveTopMargin = -50;  
            /// <summary>  
            /// An additional offset used for positioning the bottom two beehives.  
            /// </summary>  
            public const float BeehiveMiddleOffset = 50;  
           
            public const float HoneyJarTopMargin = 440;  
            public const float HoneyJarLeftMargin = 265;  
           
            public const float VatArrowOffset = -35;  
            public const float VatBottomMargin = 40;  
            public const int VatScorebarHeight = 15;  
            public const int VatScorebarWidth = 230;  
           
            /// <summary>  
            /// The size of the smoke button.  
            /// </summary>  
            public const float SmokeButtonSize = 100;  
           
            /// <summary>  
            /// This margin takes the button's own size in consideration.  
            /// </summary>  
            public const float SmokeButtonRightAbsoluteMargin = 325;  
            /// <summary>  
            /// This margin takes the button's own size in consideration.  
            /// </summary>  
            public const float SmokeButtonBottomAbsoluteMargin = 145;  
           
            public const float SprayUpOffset = 24;  
            public const float SprayMiddleOffset = 65;  
            public const float SprayDownOffset = 110;  
            public const float SprayRightOffset = 230;  
            public const float SprayLeftOffset = 130;  
           
            public const float HighScorePlaceLeftMargin = 10;  
            public const float HighScoreNameLeftMargin = 220;  
            public const float HighScoreScoreLeftMargin = 680;  
            public const float HighScoreTopMargin = 75;  
            public const float HighScoreOddVerticalJump = 74;  
            public const float HighScoreEvenVerticalJump = 69;
3. Open the **HoneycombRush.cs** file.
4. Modify the **HoneycombRush** class’s constructor.
   * 1. C#
     2. public HoneycombRush()  
        {  
            // Initialize sound system  
            AudioManager.Initialize(this);  
           
            graphics = new GraphicsDeviceManager(this);  
            Content.RootDirectory = "Content";  
           
        #if WINDOWS\_PHONE  
            // Frame rate is 30 fps by default for Windows Phone.  
            TargetElapsedTime = TimeSpan.FromTicks(333333);  
           
            graphics.IsFullScreen = true;  
           
            screenManager = new ScreenManager(this, Vector2.One);  
        #elseif XBOX  
            graphics.PreferredBackBufferHeight = 720;  
            graphics.PreferredBackBufferWidth = 1280;  
           
            screenManager = new ScreenManager(this, Vector2.One);  
        #elseif WINDOWS  
            graphics.PreferredBackBufferHeight = 480;  
            graphics.PreferredBackBufferWidth = 800;                          
           
            // Make the game windowed  
            graphics.IsFullScreen = false;  
            IsMouseVisible = true;  
           
            Components.Add(new GamerServicesComponent(this));              
           
            Vector2 scaleVector =
     3. new Vector2(graphics.PreferredBackBufferWidth / 1280f,   
                graphics.PreferredBackBufferHeight / 720f);  
           
            UIConstants.SetScale(scaleVector);  
           
            // Create a new instance of the Screen Manager. Have all drawing scaled
     4. // from 720p to the PC's resolution  
            screenManager = new ScreenManager(this, scaleVector);  
        #endif  
           
            screenManager.AddScreen(new BackgroundScreen("titleScreen"), null);  
            screenManager.AddScreen(new MainMenuScreen(), null);  
           
            Components.Add(screenManager);  
        }

# Handling Guide Events

* 1. When running on Xbox 360, a game must take care to handle Guide Events. The user triggers these events when he or she presses the **Guide** button. When Guide launches, the game should pause or otherwise consume less processing power to enable the user smooth interaction with the Guide. In this game, the gameplay screen is the only screen that will present processing strain.

To handle Guide events in the gameplay screen, do the following:

* + Open the **GameplayScreen.cs** file under the **Screens** project folder.
  + Change the **GameplayScreen** class’s **Update** method.
    1. C#
    2. public override void Update(GameTime gameTime, bool otherScreenHasFocus,
    3. bool coveredByOtherScreen)  
       {  
           // When the game starts the first thing the user sees is the count down
    4. // before the game actually begins  
           if (isAtStartupCountDown)  
           {  
               startScreenTime -= gameTime.ElapsedGameTime;  
           }  
          
           // Check for and handle a game over  
           if (CheckIfCurrentGameFinished())  
           {  
               base.Update(gameTime, otherScreenHasFocus, coveredByOtherScreen);  
          
               return;  
           }  
          
           if (!(IsActive && IsStarted && !Guide.IsVisible))  
           {  
               if (Guide.IsVisible && IsActive)  
           {  
                   PauseCurrentGame();  
               }  
          
               base.Update(gameTime, otherScreenHasFocus, coveredByOtherScreen);  
               return;  
           }  
          
           // Show all diagnostic counters  
           debugSystem.FpsCounter.Visible = showDebugInfo;  
           debugSystem.TimeRuler.Visible = showDebugInfo;  
           debugSystem.TimeRuler.ShowLog = showDebugInfo;  
          
           gameElapsed -= gameTime.ElapsedGameTime;  
          
       #if WINDOWS\_PHONE  
           HandleThumbStick();  
       #endif  
          
           HandleSmoke();  
          
           HandleCollision(gameTime);  
          
           HandleVatHoneyArrow();  
          
           beeKeeper.DrawOrder = 1;  
           int beeKeeperY = (int)(beeKeeper.Position.Y +
    5. beeKeeper.Bounds.Height - 2);  
          
           // We want to determine the draw order of the beekeeper,  
           // if the beekeeper is under half the height of the beehive   
           // it should be drawn over the beehive.  
           foreach (Beehive beehive in beehives)  
           {  
               if (beeKeeperY > beehive.Bounds.Y)  
               {  
                   if (beehive.Bounds.Y + beehive.Bounds.Height / 2 < beeKeeperY)  
                   {  
                       beeKeeper.DrawOrder = Math.Max(beeKeeper.DrawOrder,
    6. beehive.Bounds.Y + 1);  
                   }  
               }  
           }  
          
           if (gameElapsed.Minutes == 0 && gameElapsed.Seconds == 10)  
           {  
               AudioManager.PlaySound("10SecondCountDown");  
           }  
           if (gameElapsed.Minutes == 0 && gameElapsed.Seconds == 30)  
           {  
               AudioManager.PlaySound("30SecondWarning");  
           }  
          
           // Update the time remaining displayed on the vat  
           vat.DrawTimeLeft(gameElapsed);  
          
           base.Update(gameTime, otherScreenHasFocus, coveredByOtherScreen);  
       }

# Final touchups

* 1. While the game is now fully playable on all platforms, you need to make two more changes to complete the exercise.

To complete the exercise

1. From the **Objects** project folder, open the **BeeKeeper.cs** file.
2. Modify the **BeeKeeper** class’s **GetSmokeVelocityVector method**.

**Note** This change makes it easier to shoot smoke on Windows and Xbox 360, where the movement speed is more difficult to control.

* + 1. C#
    2. private Vector2 GetSmokeVelocityVector()  
       {  
           Vector2 initialVector;  
          
           switch (direction)  
           {  
               case WalkingDirection.Down:  
                   initialVector = new Vector2(0, 1);  
                   break;  
               case WalkingDirection.Up:  
                   initialVector = new Vector2(0, -1);  
                   break;  
               case WalkingDirection.Left:  
                   initialVector = new Vector2(-1, 0);  
                   break;  
               case WalkingDirection.Right:  
                   initialVector = new Vector2(1, 0);  
                   break;  
               case WalkingDirection.LeftDown:  
                   initialVector = new Vector2(-1, 1);  
                   break;  
               case WalkingDirection.RightDown:  
                   initialVector = new Vector2(1, 1);  
                   break;  
               case WalkingDirection.LeftUp:  
                   initialVector = new Vector2(-1, -1);  
                   break;  
               case WalkingDirection.RightUp:  
                   initialVector = new Vector2(1, -1);  
                   break;  
               default:  
                   throw new InvalidOperationException(
    3. "Determining the vector for an invalid walking direction");  
           }  
          
       #if WINDOWS\_PHONE  
           return initialVector \* 2f + velocity \* 1f;  
       #else  
           // On the PC/Xbox, movement speed is more touchy, so we need the puffs to
    4. // behave differently  
           return initialVector \* 5f + velocity \* 0.2f;  
       #endif   
       }

1. From the **Screens** project folder, open the **LevelOverScreen.cs** file.
2. Modify the **LevelOverScreen** class’s **Draw** method.

**Note** This shows the appropriate prompts on Xbox 360 when you transition between levels.

* + 1. C#
    2. public override void Draw(GameTime gameTime)  
       {  
           SpriteBatch spriteBatch = ScreenManager.SpriteBatch;  
          
           spriteBatch.Begin();  
          
           // Draw the footer text  
          
           if (difficultyMode.HasValue)  
           {  
       #if WINDOWS\_PHONE  
               string actionText = "Touch to start next level";  
       #elseif XBOX  
               string actionText = "Press A to start next level";  
       #else  
               string actionText = "Press space to start next level";  
       #endif  
          
               spriteBatch.DrawString(font16px, actionText,  
                   new Vector2(ScreenManager.GraphicsDevice.Viewport.Width / 2 -  
                       font16px.MeasureString(actionText).X / 2,  
                       safeArea.Bottom - font16px.MeasureString(actionText).Y - 4),  
                   Color.Black);  
           }  
           else  
           {  
       #if WINDOWS\_PHONE  
               string actionText = "Touch to end game";  
       #elseif XBOX  
               string actionText = "Press A to end game";  
       #else  
               string actionText = "Press space to end game";  
       #endif  
               spriteBatch.DrawString(font16px, actionText,  
                   new Vector2(safeArea.Left + safeArea.Width / 2 - font16px.MeasureString(actionText).X / 2,  
                       safeArea.Top + safeArea.Height - font16px.MeasureString(actionText).Y - 4),  
                   Color.Black);  
           }  
          
           spriteBatch.End();  
       }

**Congratulations!** You have taken a Windows Phone 7 game and ported it to both Windows and Xbox 360, while maintaining complete feature parity between the various versions.

* + 1. **Note** The supplied end solution for this exercise contains additional code, which does not appear as part of the exercise. This code registers debug commands in the game. The debug console is only available on Windows by pressing tab. Try typing in the two available commands **ToggleBounds** and **ToggleCollisionAreas**.