CONTENTS

OVERVIEW ........................................................................................................................................... 3

EXERCISE 1: XNA GAME STUDIO GAMES ON THE WINDOWS PHONE 7 ........................................... 6
   Task 1 – XNA Game Studio Game Basics ......................................................................................... 8
   Task 2 – XNA Framework Game Resources ................................................................................... 18
   Task 3 – XNA Game Studio Game Loop ......................................................................................... 30
   Task 4 – XNA Game Studio Game Input ......................................................................................... 38
   Task 5 – Alien Game Specific Logic ............................................................................................... 43

SUMMARY ............................................................................................................................................ 69
Overview

This lab introduces you to XNA Game Studio game development on Windows Phones, as well as to the basics of XNA Game Studio game development. During the lab you will build a simple XNA Game Studio game application that introduces key concepts in XNA Game Studio game development and learn how to use Microsoft Visual 2010 Express for Windows Phone to build and design your XNA Game Studio games for Windows Phones.

Objectives

At the end of the lab you will have:

- A high-level understanding of the XNA Game Studio game engine model within a Windows Phone 7 application
- Learned how to use resources (images, fonts, etc.) in your XNA Game Studio game
- Learned how to add game logic
- Learned about the drawing mechanism for Windows Phone XNA Game Studio games

Prerequisites

The following is required in order to complete this hands-on lab:

- Microsoft Visual Studio 2010 Express for Windows Phone or Microsoft Visual Studio 2010
- Windows Phone Developer Tools

Note: All of these Tools can be downloaded together in a single package from http://developer.windowsphone.com

Setup

For convenience, much of the code used in this hands-on lab is available as Visual Studio code snippets. To install the code snippets:

1. Run the .vsi installer located in the lab's Source\Setup folder.

Note: If you have issues running the code snippets installer you can install the code snippets manually by copying all the .snippet files located in the Source\Setup\CodeSnippets folder of the lab to the following folder:
Using the Code Snippets

With code snippets, you have all the code you need at your fingertips. The lab document will tell you exactly when you can use them. For example,

![Code Snippet](image)

Figure 1
Using Visual Studio code snippets to insert code into your project

To add this code snippet in Visual Studio, you simply place the cursor where you would like the code to be inserted, start typing the snippet name (without spaces or hyphens), watch as IntelliSense picks up the snippet name, and then press the Tab key twice when the snippet you want is selected. The code will be inserted at the cursor location.

```
private void ClickMeButton_Click(object sender, RoutedEventArgs e)
{
    Hello
}
```

Figure 2
Start typing the snippet name
private void ClickMeButton_Click(object sender, RoutedEventArgs e)
{
    HelloPhoneEx1Task4Step4ClickMeButtonEventHandler;
}

Figure 3
Press Tab to select the highlighted snippet

private void ClickMeButton_Click(object sender, RoutedEventArgs e)
{
    BannerTextBlock.Text = MessageTextBox.Text;
    MessageTextBox.Text = String.Empty;
}

Figure 4
Press Tab again to expand the snippet

To insert a code snippet using the mouse rather than the keyboard, right-click where you want to insert the code snippet, select Insert Snippet followed by My Code Snippets and then pick the relevant snippet from the list.

To learn more about Visual Studio IntelliSense Code Snippets, including how to create your own, see http://msdn.microsoft.com/en-us/library/ms165392.aspx.

Tasks
This hands-on lab includes the following tasks:

1. XNA Game Studio Game Basics
2. XNA Framework Resources
3. XNA Game Studio Game Loop
4. XNA Game Studio Game Input
5. Alien Game Specific Logic

Estimated time to complete this lab: 60 minutes.
Exercise 1: XNA Game Studio Games on the Windows Phone 7

If you have ever wanted to make your own games, Microsoft® XNA™ Game Studio 4.0 is for you. Student, hobbyist, independent game developer — anybody can create and share great games using XNA Game Studio.

XNA Game Studio 4.0 is a game development product from Microsoft that is built on top of the Microsoft Visual Studio 2010 Express for Windows Phone, giving game developers the power and simplicity of C# as a programming language. XNA Game Studio 4.0 includes the XNA Framework and the XNA Framework Content Pipeline, which provide an easy and flexible way to import three-dimensional (3D) models, textures, sounds, and other assets into your game, and a game-focused application programming interface (API) that simplifies development for Xbox 360®, Windows® and now Windows Phone 7®.

The XNA Framework is an application programming interface (API). What that means is that it is a framework developed by Microsoft to help you make games faster. However, it’s not a drag and drop game maker and you will need to learn how to program before you can use it. It is easy to use, but you will have to be somewhat technical to develop games with it.

The XNA Framework is not a game engine. It does not include physics, collision detection, or other things often found in game engines. It is a game development framework, but how the game works is programmed entirely by you.

During this lab you will build a full XNA Game Studio game for the Windows Phone — “Alien Game” — a simple shooter game. The goal in Alien Game is simple: Protect earth against the invading aliens for as long as possible. The longer you last, the more difficult the game becomes. Watch out for the smaller aliens that come out at night!

General Architecture

Alien Game uses the game screen management architecture from the Game State Management sample (originally found at http://creators.xna.com/en-US/samples/gamestatemanagement), which provides the assets for this lab. The game includes three possible states:

- Main menu (MainMenuScreen class)
- Playing the game (GameplayScreen class)
- Paused (PauseScreen class)

Alien Game performs all content-loading at startup. The first thing it does is load and display the BackgroundScreen. Then it loads and displays the LoadingScreen. The LoadingScreen loads the content. The screen doesn't draw anything itself, so the user sees the Background screen (which also includes the
The LoadingScreen could draw or display some form of progress, but generally the loading is so quick that you won't see a progress display. Once all of the content is loaded, the MainMenuScreen is loaded and displayed, and the menus animate onto the screen. This allows for quicker transitions between the main menu and gameplay, without a lengthy pause as the content is loaded. This helps the hard-drive-based devices especially, as the hard drive may spin down.

**GameplayScreen and Game Classes**

The design and implementation of Alien Game is relatively simple. All logic and drawing is contained in GameplayScreen. There are a few game-specific types found at the end of GameplayScreen.cs, namely Bullet, Alien, and Player. The code inside of GameplayScreen.cs is grouped by functionality: Loading and Unloading Content, Update and Game Simulation, and Drawing.

**ParticleSystem**

Alien Game includes a simple sprite-based particle system that is used for the explosion and dust effects. The definition and creation of the effects is hard coded into the ParticleSystem, and is accessed using the factory CreateXXXEffect methods.

The completed game will look as follows:

![Alien Game running on Windows Phone](image)

**Figure 5**

*Alien Game running on Windows Phone*

**XNA Game Studio Game Basics**

The game is built in levels. Levels are connected to each other by the game plot, game player, enemies, etc. In the simple case, each level could be treated as a whole game.
A level usually has 3 states:

- **Loading** – In this state, the system loads resources, sets up level-related variables, calculates the game-world (game-world is the place where all the game process occurs), and performs any other tasks that must be performed before the game actually begins. This state occurs only once in the level/game life-cycle.

- **Update** – In this state, the system needs to update the game-world state. Usually this means calculating new positions of the acting persons (player(s) and enemies), updating health, ammo, and other status, recalculating the score and other game logic according to the game. This state occurs throughout the time that the game engine is active.

- **Draw** – In this state, the system draws the changes calculated in the update state to the output graphics device. This state occurs throughout the time that the game engine is active.

In the XNA Framework, the last two stages could occur up to 60 times per second on a PC or Xbox 360 and up to 30 times per second on a Zune, Zune HD or Windows Phone 7 device.

**Task 1 – XNA Game Studio Game Basics**

In this section, you will create your first XNA Game Studio game for the Windows Phone. The game will be a simple one, but you will add functionality throughout the lab.

**Note:** The steps in this hands-on lab illustrate procedures using Microsoft Visual Studio 2010 Express for Windows Phone, but they are equally applicable to Microsoft Visual Studio 2010 with the Windows Phone Developer Tools. Instructions that refer generically to Visual Studio apply to both products.

1. Open Microsoft Visual Studio 2010 Express for Windows Phone from Start | All Programs | Microsoft Visual Studio 2010 Express | Microsoft Visual Studio 2010 Express for Windows Phone.

   **Visual Studio 2010:** Open Visual Studio 2010 from Start | All Programs | Microsoft Visual Studio 2010.

2. In the **File** menu, choose **New Project**.

   **Visual Studio 2010:** In the File menu, point to New and then select Project.

3. In the **New Project** dialog, select the **XNA Game Studio game** for Windows Phone category and, from the list of installed templates, select **Windows Phone Game (4.0)**; then set the name to
**AlienGame** and solution name to **Begin**. Set the location to **Ex1-AlienGame** in the **Source** folder of the lab. Click **OK**.

![Figure 6](image)

*Creating a new Windows Phone Game application project in Microsoft Visual Studio 2010 Express for Windows Phone*

4. In **Solution Explorer**, review the structure of the solution generated by the Windows Phone Application template. Any Visual Studio solution is a container for related projects; in this case, it contains an XNA Game Studio game for Windows Phone project named **AlienGame** and a related games resources project named **AlienGameContent**.
5. The generated project includes a default game implementation that contains the basic XNA Game Studio game loop. It is located in the Game1.cs file.

6. Open the Game1.cs file. We recommend that you change the default name to the name that reflects your game.

7. Rename the main game class (default name Game1) to AlienGame. To rename it, right click on the class name, select Refactor | Rename.

Note: Solution Explorer allows you to view items and perform item management tasks in a solution or a project. To show Solution Explorer, press CTRL + W, S or in the View menu, select Other Windows | Solution Explorer.
8. In the Rename dialog window's **New name** field, enter AlienGame and click **OK**.

![Figure 9](image)

*Figure 9*

*Giving the name to the main game class*

9. Review changes suggested by Visual Studio and click **Apply**.
Figure 10
Apply changes to main game class

10. Rename the filename to match the new class name. Right-click on Game1.cs in Solution Explorer and choose Rename. Give the class the new name AlienGame.cs.
11. The `GameThumbnail.png` file contains the icon that identifies the application in the quick launch screen of the phone device. You can double-click the item in Solution Explorer to open the file in a registered application on your machine that can edit image files, for example, `Paint.exe`. You will change it later in lab.

   **Note:** In Visual Studio 2010, double-clicking the icon file in Solution Explorer opens the built-in image editor.

   A XNA Game Studio game for Windows Phone application typically takes advantage of services provided by the underlying platform or by other libraries. To use this functionality, the application needs to reference the corresponding assemblies that implement these services.

12. To display the assemblies referenced by the project, expand the `References` node in Solution Explorer and examine the list. It contains regular XNA Framework assemblies as well as assemblies specific to the Windows Phone platform.
Currently, the application does not do much, but it is ready for its first test run. In this step, you build the application, deploy it to the Windows Phone Emulator, and then execute it to understand the typical development cycle.

13. In the **Debug** menu, select **Windows | Output** to open the Output window.

14. Select **Build Solution** in the **Debug** menu or press the **SHIFT + F6** key combination to compile the projects in the solution.

**Visual Studio 2010:** Select **Build Solution** in the **Build** menu or press **CTRL + SHIFT + B** to compile the projects in the solution.
15. Observe the **Output** window and review the trace messages generated during the build process, including a final message with its outcome.

![Output window](image)

**Figure 13**

*Building the application in Visual Studio*

You should not observe any errors at this stage but, if the project were to contain compilation errors, these would appear in the **Output** window. To deal with these kinds of errors, you can take advantage of the **Error List** window. This window displays errors, warnings, and messages produced by the compiler in a list that you can sort and filter based on the severity of the error. Moreover, you can double-click an item in the list to automatically open the relevant source code file and navigate to the source of the error.

16. To open the Error List window, in the **View** menu, point to **Other Windows** and select **Error List**.

**Visual Studio 2010**: To open the Error List window, in the **View** menu select **Error List**.

![Error List window](image)

**Figure 14**

*Error List window shows errors during the build process*
17. Verify that the target of the deployment is the Windows Phone Emulator. To do this, ensure that Windows Phone 7 Emulator is selected in the **Select Device** drop down next to the **Start Debugging** button on the toolbar.

![Choosing the target device to deploy the application](Image)

**Figure 15**  
*Choosing the target device to deploy the application*

**Note:** When you deploy your application from Visual Studio, you have the option to deploy it to a real device or to the Windows Phone Emulator.

18. Press **F5** to launch the application in the Windows Phone Emulator. Notice that a device emulator window appears and there is a pause while Visual Studio sets up the emulator environment and deploys the image. Once it is ready, the emulator shows the Start page and shortly thereafter, your application appears in the emulator window.

The application will display a simple blue screen with nothing else shown. This is normal to an application in such early stages.
Until you create the user interface and program the application logic, there is very little that you can do with the application.
19. Press \texttt{SHIFT + F5} or click the \textbf{Stop} button in the toolbar to detach the debugger and end the debugging session. Do not close the emulator window.

![AlienGame (Running) - Microsoft Visual Studio 2010 Express for Windows Phone](image)

**Figure 17**  
Ending the debugging session

\begin{quote}
\textbf{Note:} When you start a debugging session, it takes a perceptible amount of time to set up the emulator environment and launch the application. To streamline your debugging experience, avoid closing the emulator while you work with the source code in Visual Studio. Once the emulator is running, it takes very little time to stop the current session, edit the source code, and then build and deploy a new image of your application to start a new debugging session.
\end{quote}

\textbf{Task 2 – XNA Framework Game Resources}

Many games using pre-defined images to present the game, process sound and other resources to play as a part of gameplay. This lab provides you with a number of such resources to make the game development process easier. During this task you will add those resources to the game. This lab also provides a number of code files to handle the complexity of menu and screen changes during the game. You will add those files to the game also.

You are building the game. The game will present some graphics. In this task you will add some resources to the application and some existing game logic (ScreenManager).

\begin{quote}
\textbf{Note:} All the game resources are provided in the lab install folder under the following locations of this lab:

Source\Assets\Code – all CSharp code files  
Source\Assets\Media – all graphics, fonts and sounds
\end{quote}
1. Close the project in Visual Studio. Switch to Windows Explorer, navigate to the project location and copy two files from the Source\Assets\Media\Images\Icons folder of this lab into the “AlienGame” directory, replacing the existing files:
   - Game.ico
   - GameThumbnail.png

2. Re-open Visual Studio 2010 and open the AlienGame project.

   Most games use art in the form of models, meshes, sprites, textures, effects, terrains, animations, and so on. Such art assets can be created in many different ways and stored in many different file formats. They tend to change frequently in the course of game development. The Content Pipeline is designed to help you include such art assets in your game easily and automatically. An artist working on a car model can add the resulting file to the XNA Game Studio game project, assign the model a name, and choose an importer and content processor for it. Then, a developer who wants to make the car drive can load it by name using a call to ContentManager.Load. This simple flow lets the artist focus on creating assets and the developer focus on using them, without either having to spend time worrying about content transformation.

   The XNA Content Pipeline is readily integrated into your XNA Game Studio project. You just add the resource to your project and when you compile it, the data is imported and converted in a XNB (XNA Binary File) using a Content Importer. This XNB file is generated for the right platform. Content Importers are implemented as assemblies. In addition to the standard ones provided by XNA Game Studio, you can also use custom importers and processors that you or other third parties develop. Some of standard Content Importers include the following file types (partial list):
   - Autodesk FBX format (.fbx)
   - DirectX Effect file format (.fx)
   - Font description specified in a .spritefont file
   - Texture file. The following types are supported: .bmp, .dds, .dib, .hdr, .jpg, .pfm, .png, .ppm, and .tga
   - Game audio specified in the Microsoft Cross-Platform Audio Creation Tool (XACT) format (.xap)

3. This lab provides a number of media resources such as fonts, sounds, and images. Add the following items to the AlienGameContent project:
   - All the fonts from the Source\Assets\Media\Fonts folder
   - All the images from the Source\Assets\Media\Images\Content folder
   - All the images from the Source\Assets\Media\Sounds folder
4. To add existing items right-click on the AlienGameContent project name and select Add | Existing Items:

![Image of adding existing items to the project](image.png)

**Figure 18**
Adding existing items to the project

5. Navigate to the file locations and select the files. In some cases you will not see all the files. In this case change the filter in the file selection dialog to show all the files in the directory:

![Image of file selection dialog](image2.png)
Figure 19  
*Changing file types filter to show all the files in the folder*

6. The resulting **AlienGameContent** project should look as follows:

![Image of AlienGameContent project structure]

**Figure 20**  
*Resulting Contents project structure*

7. Add a new project folder – right-click on AlienGame, select **Add | New Folder**.

![Image of adding a new project folder]

**Figure 21**  
*Adding New Project Folder*
8. Name the folder **ScreenManager**.

This folder will hold the source files provide by the lab. Those files will help to manage the complexity of creation game screens, menus, and changing between them.

**Note:** This code implements the standard approach for creating XNA Game Studio menus and screens.

9. Select the folder and add all existing Screen Manager files from the `Source\Assets\Code\ScreenManager` folder of this lab.

10. Add the `ParticleSystem.cs` file from the `Source\Assets\Code` folder to the root of the `AlienGame` project.

11. The resulting project should look as follows:

![Figure 22](image)

**Figure 22**

*Resulting AlienGame project structure*

12. Add a new class to the AlienGame project and name it **BackgroundScreen**:
Figure 23  
Adding a new class to the project

13. Open the new class and add the following using statements:

(Code Snippet – Game Development with XNA – Background Screen using statements)

```
C#
using AlienGameSample;
using Microsoft.Xna.Framework;
```

14. Derive a new class from the GameScreen class (the GameScreen class is defined in classes added previously to the ScreenManager folder):

```
C#
class BackgroundScreen : GameScreen
{
}
```

15. Add the following class variables to be used later for loading resources:

(Code Snippet – Game Development with XNA – Background Screen class variables)

```
C#
Texture2D title;
```
16. Define class constructor as follows:

(Code Snippet – Game Development with XNA – Background Screen Constructor)

```csharp
public BackgroundScreen()
{
    TransitionOnTime = TimeSpan.FromSeconds(0.0);
    TransitionOffTime = TimeSpan.FromSeconds(0.5);
}
```

17. The GameScreen Class defines some core game functionality according to what was described in the exercise preface: LoadContent, Update, and Draw. Override the base class LoadContent functionality:

(Code Snippet – Game Development with XNA – Background Screen LoadContent method)

```csharp
public override void LoadContent()
{
    title = ScreenManager.Game.Content.Load<Texture2D>("title");
    background = ScreenManager.Game.Content.Load<Texture2D>("background");
}
```

This code snippet loads the content from resources of the game. The content is loaded by name.

18. Now create a>LoadingScreen</code> class. The screen presented with the class will be presented while game resources being loaded.

**Note:** To create a new class, in Solution Explorer right-click the AlienGame project and select Add | Class.

19. Add following using statements to the LoadingScreen class:

(Code Snippet – Game Development with XNA – Loading Screen using statements)

```csharp
using AlienGameSample;
using System.Threading;
using Microsoft.Xna.Framework;
```
20. Derive a new class from the `GameScreen` base class (like you did for the previous class), and add the constructor as follows:

(Code Snippet – *Game Development with XNA – Loading Screen constructor*)

C#  
```csharp
class LoadingScreen : GameScreen
{
    public LoadingScreen()
    {
        TransitionOnTime = TimeSpan.FromSeconds(0.0);
        TransitionOffTime = TimeSpan.FromSeconds(0.0);
    }
}
```

21. Add a class variable to hold the Thread which will be used to load the components:

(Code Snippet – *Game Development with XNA – Loading Screen class variables*)

C#  
```csharp
private Thread backgroundThread;
```

22. Create a method to load the content. This approach is part of the standard loading procedures in XNA programming:

(Code Snippet – *Game Development with XNA – Loading Screen BackgroundLoadContent method*)

C#  
```csharp
void BackgroundLoadContent()
{
    ScreenManager.Game.Content.Load<object>("alien_hit");
    ScreenManager.Game.Content.Load<object>("alien1");
    ScreenManager.Game.Content.Load<object>("background");
    ScreenManager.Game.Content.Load<object>("badguy_blue");
    ScreenManager.Game.Content.Load<object>("badguy_green");
    ScreenManager.Game.Content.Load<object>("badguy_orange");
    ScreenManager.Game.Content.Load<object>("badguy_red");
    ScreenManager.Game.Content.Load<object>("bullet");
    ScreenManager.Game.Content.Load<object>("cloud1");
    ScreenManager.Game.Content.Load<object>("cloud2");
    ScreenManager.Game.Content.Load<object>("fire");
    ScreenManager.Game.Content.Load<object>("gamefont");
    ScreenManager.Game.Content.Load<object>("ground");
    ScreenManager.Game.Content.Load<object>("hills");
    ScreenManager.Game.Content.Load<object>("laser");
}````
ScreenManager.Game.Content.Load<object>("menufont");
ScreenManager.Game.Content.Load<object>("moon");
ScreenManager.Game.Content.Load<object>("mountains_blurred");
ScreenManager.Game.Content.Load<object>("player_hit");
ScreenManager.Game.Content.Load<object>("scorefont");
ScreenManager.Game.Content.Load<object>("smoke");
ScreenManager.Game.Content.Load<object>("sun");
ScreenManager.Game.Content.Load<object>("tank");
ScreenManager.Game.Content.Load<object>("tank_fire");
ScreenManager.Game.Content.Load<object>("tank_tire");
ScreenManager.Game.Content.Load<object>("tank_top");
ScreenManager.Game.Content.Load<object>("title");
ScreenManager.Game.Content.Load<object>("titlefont");
}

23. Override the base class **LoadContent** method and start the loader method as a new thread to achieve asynchronous loading of the resources:

```
public override void LoadContent()
{
    if (backgroundThread == null)
    {
        backgroundThread = new Thread(BackgroundLoadContent);
        backgroundThread.Start();
    }
    base.LoadContent();
}
```

**Note:** In our simple game the resources will be loaded momentarily, but for more complex games, this approach enables showing a progress indicator or a splash screen.

24. Override the base class **Update** method to wait for **LoadContent** to finish and jump to the MainMenu screen (added in next steps):

```
public override void Update(GameTime gameTime, bool otherScreenHasFocus, bool coveredByOtherScreen)
{
    if (backgroundThread != null && backgroundThread.Join(10))
    {
        // MainMenu screen logic
    }
}
```
25. Now add a new class and name it **MainMenuScreen**. When created, add the following *using* statement to it:

(Code Snippet – *Game Development with XNA – MainMenu Screen using statement*)

```csharp
using AlienGameSample;
```

26. Derive the new class from the **MenuScreen** base class. This class is also defined in classes added to **ScreenManager** folder and facilitates all typical functionality needed to show/interact with menu and menu items.

27. Create **MainMenuScreen** class constructor:

(Code Snippet – *Game Development with XNA – MainMenu Screen constructor*)

```csharp
class MainMenuScreen : MenuScreen
{
    public MainMenuScreen()
    : base("Main")
    {
        // Create our menu entries.
        MenuEntry startGameMenuEntry = new MenuEntry("START GAME");
        MenuEntry exitMenuEntry = new MenuEntry("QUIT");

        // Hook up menu event handlers.
        startGameMenuEntry.Selected += StartGameMenuEntrySelected;
        exitMenuEntry.Selected += OnCancel;

        // Add entries to the menu.
        MenuEntries.Add(startGameMenuEntry);
        MenuEntries.Add(exitMenuEntry);
    }
}
```
28. In the constructor you subscribed for two events, which will fire when the user selects the menu items. Create the event handler methods to handle those events:

(Code Snippet – Game Development with XNA – MainMenu Screen event handlers)

```csharp
void StartGameMenuEntrySelected(object sender, EventArgs e)
{
}

protected override void OnCancel()
{
    ScreenManager.Game.Exit();
}
```

29. Open the `AlienGame.cs` file and add the following using statement:

(Code Snippet – Game Development with XNA – AlienGame using statement)

```csharp
using AlienGameSample;
```

30. Add the following class variable:

(Code Snippet – Game Development with XNA – AlienGame variables)

```csharp
ScreenManager screenManager;
```

31. Delete the spriteBatch variable created by Visual Studio:

```csharp
SpriteBatch spriteBatch;
```

32. Delete all the code except for the class’s constructor and the variable declarations.

33. After initializing the main game class, you need to load the game resources and present the user with some background for the menu. The menu screen will follow the loading process. In addition, we recommend that you define the preferred resolution to the graphical device. To achieve this, change the constructor method according to the following code snippet:
(Code Snippet – *Game Development with XNA – AlienGame Constructor*)

```csharp
public AlienGame()
{
    graphics = new GraphicsDeviceManager(this);

    // Set the Windows Phone screen resolution
    graphics.PreferredBackBufferWidth = 480;
    graphics.PreferredBackBufferHeight = 800;

    Content.RootDirectory = "Content";

    // Frame rate is 30 fps by default for Windows Phone.
    TargetElapsedTime = TimeSpan.FromSeconds(1 / 30.0);

    // Create a new instance of the Screen Manager
    screenManager = new ScreenManager(this);
    Components.Add(screenManager);

    // Add two new screens
    screenManager.AddScreen(new BackgroundScreen());
    screenManager.AddScreen(new LoadingScreen());
}
```

34. Compile and run the application. After the application loads, the main menu screen should appear:
35. Stop the debugging and return to editing the application.

During this task you added provided resources to the game, created a number of screens to present the user with some basic user interfaces while loading the game, and created a main menu.

**Task 3 – XNA Game Studio Game Loop**

In this task you will focus on two remaining parts of the game – overriding the **Update** and **Draw** functionalities.

1. Open `BackgroundScreen.cs`.
2. Override the base class **Update** method as follows:
3. Override the base class `Draw` method. The `Draw` method will use the `SpriteBatch` class from the `Microsoft.Xna.Framework.Graphics` namespace to draw on the graphics device. It enables a group of sprites to be drawn using the same settings. Change the `Draw` method to match the following code snippet:

```
public override void Draw(GameTime gameTime) {
    SpriteBatch spriteBatch = ScreenManager.SpriteBatch;

    // Make the menu slide into place during transitions, using a
    // power curve to make things look more interesting (this makes
    // the movement slow down as it nears the end).
    float transitionOffset = (float)Math.Pow(TransitionPosition, 2);

    spriteBatch.Begin();

    // Draw Background
    spriteBatch.Draw(background, new Vector2(0, 0),
                      new Color(255, 255, 255, TransitionAlpha));

    // Draw Title
    spriteBatch.Draw(title, new Vector2(60, 55),
                     new Color(255, 255, 255, TransitionAlpha));

    spriteBatch.End();
}
```

4. Press `F5` to compile and run the application.
5. Stop the debugging (SHIFT+F5) and return to editing the application.

6. Add an additional class to the application and set its name to `GameplayScreen`.

   **Note:** To create a new class, in Solution Explorer right-click the AlienGame project and select Add | Class.

7. Add the following `using` statements to the new class:

   (Code Snippet – Game Development with XNA – Gameplay Screen using statements )

   ```c#
   using AlienGameSample;
   using Microsoft.Xna.Framework;
   ```

   ![Figure 25](image)

   *Running application after changing Update and Draw methods*
using Microsoft.Xna.Framework.Audio;
using System.IO.IsolatedStorage;
using System.IO;

8. Derive the class from `GameScreen`.

```csharp
class GameplayScreen : GameScreen
{
}
```

9. Add following class variables (you will use them in the game). Those variables will be used later in the lab to handle game logic, user input, drawing, etc.:

(Code Snippet – *Game Development with XNA – Gameplay Screen variables*)

```csharp
// // Game Play Members
// Rectangle worldBounds;
bool gameOver;
int baseLevelKillCount;
int levelKillCount;
float alienSpawnTimer;
float alienSpawnRate;
float alienMaxAccuracy;
float alienSpeedMin;
float alienSpeedMax;
int alienScore;
int nextLife;
int hitStreak;
int highScore;
Random random;

// // Rendering Members
// Texture2D cloud1Texture;
Texture2D cloud2Texture;
Texture2D sunTexture;
Texture2D moonTexture;
Texture2D groundTexture;
Texture2D tankTexture;
```
Texture2D alienTexture;
Texture2D badguy_blue;
Texture2D badguy_red;
Texture2D badguy_green;
Texture2D badguy_orange;
Texture2D mountainsTexture;
Texture2D hillsTexture;
Texture2D bulletTexture;
Texture2D laserTexture;

SpriteFont scoreFont;
SpriteFont menuFont;

Vector2 cloud1Position;
Vector2 cloud2Position;
Vector2 sunPosition;

// Level changes, nighttime transitions, etc
float transitionFactor; // 0.0f == day, 1.0f == night
float transitionRate; // > 0.0f == day to night

ParticleSystem particles;

// Audio Members
//
SoundEffect alienFired;
SoundEffect alienDied;
SoundEffect playerFired;
SoundEffect playerDied;

//Screen dimension consts
const float screenHeight = 800.0f;
const float screenWidth = 480.0f;
const int leftOffset = 25;
const int topOffset = 50;
const int bottomOffset = 20;

10. The **GamePlay** class constructor defines the speed of screen transitions (between the Gameplay Screen and other screens in the game) and the size of “game world” – the place where all the game actions are handled. Add this class constructor as follows:

(Code Snippet – *Game Development with XNA – Gameplay Screen Constructor*)

```csharp
public GameplayScreen()
{
```
random = new Random();

worldBounds = new Rectangle(0, 0, (int)setWidth, (int)setHeight);

gameOver = true;

TransitionOnTime = TimeSpan.FromSeconds(0.0);
TransitionOffTime = TimeSpan.FromSeconds(0.0);

11. Now let’s create content Loading and Unloading functionality. Override the base class
LoadContent and UnloadContent methods.

Add the LoadContent code snippet:

(Code Snippet – Game Development with XNA – Gameplay Screen LoadContent method)

C#

public override void LoadContent()
{
    cloud1Texture = ScreenManager.Game.Content.Load<Texture2D>("cloud1");
    cloud2Texture = ScreenManager.Game.Content.Load<Texture2D>("cloud2");
    sunTexture = ScreenManager.Game.Content.Load<Texture2D>("sun");
    moonTexture = ScreenManager.Game.Content.Load<Texture2D>("moon");
    groundTexture = ScreenManager.Game.Content.Load<Texture2D>("ground");
    tankTexture = ScreenManager.Game.Content.Load<Texture2D>("tank");
    mountainsTexture = ScreenManager.Game.Content.Load<Texture2D>("mountains_blurred");
    hillsTexture = ScreenManager.Game.Content.Load<Texture2D>("hills");
    alienTexture = ScreenManager.Game.Content.Load<Texture2D>("alien1");
    badguy_blue = ScreenManager.Game.Content.Load<Texture2D>("badguy_blue");
    badguy_red = ScreenManager.Game.Content.Load<Texture2D>("badguy_red");
    badguy_green = ScreenManager.Game.Content.Load<Texture2D>("badguy_green");
    badguy_orange = ScreenManager.Game.Content.Load<Texture2D>("badguy_orange");
    bulletTexture = ScreenManager.Game.Content.Load<Texture2D>("bullet");
    laserTexture = ScreenManager.Game.Content.Load<Texture2D>("laser");
    alienFired = ScreenManager.Game.Content.Load<SoundEffect>("Tank_Fire");
    alienDied = ScreenManager.Game.Content.Load<SoundEffect>("Alien_Hit");
    playerFired = ScreenManager.Game.Content.Load<SoundEffect>("Tank_Fire");
    playerDied = ScreenManager.Game.Content.Load<SoundEffect>("Player_Hit");
    scoreFont = ScreenManager.Game.Content.Load<SpriteFont>("ScoreFont");
    menuFont = ScreenManager.Game.Content.Load<SpriteFont>("MenuFont");

    cloud1Position = new Vector2(224 - cloud1Texture.Width, 32);
    cloud2Position = new Vector2(64, 80);
12. Add the `UnloadContent` code snippet:

(Code Snippet – *Game Development with XNA – Gameplay Screen Unload method*)

```csharp
public override void UnloadContent()
{
    particles = null;
    base.UnloadContent();
}
```

13. Override the base class `Update` functionality:

(Note: This method will be changed later during the lab to provide the game logic.)

(Code Snippet – *Game Development with XNA – Gameplay Screen Update method*)

```csharp
/// <summary>
/// Runs one frame of update for the game.
/// </summary>
/// <param name="gameTime">
/// Provides a snapshot of timing values.
/// </param>
public override void Update(GameTime gameTime, 
bool otherScreenHasFocus, bool coveredByOtherScreen)
{
    float elapsed = (float)gameTime.ElapsedGameTime.TotalSeconds;
    base.Update(gameTime, otherScreenHasFocus, coveredByOtherScreen);
}
```

14. Override the base class `Draw` functionality that draws the current “game world” state up to 30 times per second.

(Code Snippet – *Game Development with XNA – Gameplay Screen Draw region*)

```csharp
```
/// <summary>
/// Draw the game world, effects, and HUD
/// </summary>
public override void Draw(GameTime gameTime)
{
    float elapsedTime = (float)gameTime.ElapsedGameTime.TotalSeconds;

    ScreenManager.SpriteBatch.Begin();
    ScreenManager.SpriteBatch.End();
}

Note: The GameTime could be used to calculate the drawing locations of various game items.

15. Open MainMenuScreen.cs, locate the StartGameMenuEntrySelected method, which should be still empty, and add the following code. This will add the GameplayScreen screen to the ScreenManager when user clicks “START GAME” button:

(Code Snippet – Game Development with XNA – MainMenu Screen – GameMenuEntrySelected handler)

C#

void StartGameMenuEntrySelected(object sender, EventArgs e)
{
    ScreenManager.AddScreen(new GameplayScreen());
}

16. Compile and run the application. Click the “START GAME” menu entry and observe the main menu items scrolling down from the screen.
Figure 26
Running the game

**Note:** The GameplayScreen is still empty, thus you will not see any additional change at the current stage of the game development.

17. Stop the debugging and return to editing the application.

During this task you created a main game class, and overrode basic game functionality.

**Task 4 – XNA Game Studio Game Input**

In this task, you will add an input for the game. In the Windows Phone, input is done via the Touch panel and Accelerometer. As accelerometer is not supported in the Windows Phone emulator, this lab
prepares the solution with keyboard support to simulate and substitute the functionality of the Accelerometer. This will not work on the real device, but will work on the emulator.


![Adding a reference to the Microsoft.Device.Sensors assembly](image)

**Note:** To add a reference, in Solution Explorer right-click the References node, under the AlienGame project and select Add References.

2. Open `GameplayScreen.cs` (if not opened already).

3. Add the following additional `using` statements to the class:

```csharp
using Microsoft.Xna.Framework.Input;
using Microsoft.Devices.Sensors;
```
4. Add additional class variables to hold Touch and Accelerometer states:

(Code Snippet – *Game Development with XNA – Gameplay Screen – more class variables*)

```csharp
//Input Members
AccelerometerReadingEventArgs accelState;
TouchCollection touchState;
Accelerometer Accelerometer;
```

5. Initialize accelerometer and subscribe to its events. To do this add the following code snippet to the class constructor:

(Code Snippet – *Game Development with XNA – Gameplay Screen – Accelerometer Initialization*)

```csharp
Accelerometer = new Accelerometer();
if (Accelerometer.State == SensorState.Ready)
{
    Accelerometer.ReadingChanged += (s, e) =>
    {
        accelState = e;
    };
    Accelerometer.Start();
}
```

6. Create the “Input” region within the `GameplayScreen` class:

```csharp
#region Input
#endregion
```

7. Add an override to the base class `HandleInput` method inside the `Input` region:

This method will read the current user’s input and be used later to respond with changes in game variables.

**Note:** In the emulator, mouse clicks will be exposed as touches and keyboard input. In case of the Windows Phone device, the keyboard input will never occur.

(Code Snippet – *Game Development with XNA – Gameplan Screen – HandleInput method*)

```csharp
```
/// <summary>
/// Input helper method provided by GameScreen. Packages up the various input
/// values for ease of use. Here it checks for pausing and handles controlling
/// the player's tank.
/// </summary>
/// <param name="input">The state of the gamepads</param>
public override void HandleInput(InputState input)
{
    if (input == null)
        throw new ArgumentNullException("input");

    if (input.PauseGame)
    {
        if (gameOver == true)
            finishCurrentGame();
    }
    else
    {
        touchState = TouchPanel.GetState();
        bool buttonTouched = false;

        //interpret touch screen presses
        foreach (TouchLocation location in touchState)
        {
            switch (location.State)
            {
                case TouchLocationState.Pressed:
                    buttonTouched = true;
                    break;
                case TouchLocationState.Moved:
                    break;
                case TouchLocationState.Released:
                    break;
            }
        }

        float movement = 0.0f;
        if (accelState != null)
        {
            if (Math.Abs(accelState.X) > 0.10f)
            {
                if (accelState.X > 0.0f)
                    movement = 1.0f;
                else
                    movement = -1.0f;
            }
        }
//TODO: Update player velocity over X axis #1

//This section handles tank movement. We only allow one "movement" action
//to occur at once so that touchpad devices don't get double hits.
KeyboardState keyState = Keyboard.GetState();

if (input.CurrentGamePadStates[0].DPad.Left == ButtonState.Pressed || keyState.IsKeyDown(Keys.Left))
{
    //TODO: Update player velocity over X axis #2
}
else if (input.CurrentGamePadStates[0].DPad.Right == ButtonState.Pressed || keyState.IsKeyDown(Keys.Right))
{
    //TODO: Update player velocity over X axis #3
}
else
{
    //TODO: Update player velocity over X axis #4
}

// B button, or pressing on the upper half of the pad or space on keyboard or touching the touch panel fires the weapon.
if (input.CurrentGamePadStates[0].IsButtonDown(Buttons.B) || input.CurrentGamePadStates[0].IsButtonDown(Buttons.A) || input.CurrentGamePadStates[0].ThumbSticks.Left.Y > 0.25f || keyState.IsKeyDown(Keys.Space) || buttonTouched)
{
    if (!gameOver)
    {
        //TODO: Fire the bullet
    }

    else if (gameOver)
    {
        finishCurrentGame();
    }
}
### Task 5 – Alien Game Specific Logic

In this task, you will create game-specific logic, helper methods, and classes.

1. In `GameplayScreen.cs` file, create a new helper class (outside of the GameplayScreen class) according to the following code snippet:

   (Code Snippet – Game Development with XNA – Gameplay Screen – Bullet class)

   ```csharp
   /// <summary>
   /// Represents either an alien or player bullet
   /// </summary>
   public class Bullet
   {
       public Vector2 Position;
       public Vector2 Velocity;
       public bool IsAlive;
   }
   ```

2. Add two helper classes after the Bullet class:

   (Code Snippet – Game Development with XNA – Gameplay Screen – Player and Alien classes)

   ```csharp
   /// <summary>
   /// The player's state
   /// </summary>
   ```

9. Compile the application.

During this task you created an input-handling sub-system of the game. It will be used in the next task to create the game logic.
public class Player
{
    public Vector2 Position;
    public Vector2 Velocity;
    public float Width;
    public float Height;
    public bool IsAlive;
    public float FireTimer;
    public float RespawnTimer;
    public string Name;
    public Texture2D Picture;
    public int Score;
    public int Lives;
}

/// <summary>
/// Data for an alien. The only difference between the ships
/// and the badguys are the texture used.
/// </summary>
public class Alien
{
    public Vector2 Position;
    public Texture2D Texture;
    public Vector2 Velocity;
    public float Width;
    public float Height;
    public int Score;
    public bool IsAlive;
    public float FireTimer;
    public float Accuracy;
    public int FireCount;
}

3. Add the following GameplayScreen class variables:

(Code Snippet – Game Development with XNA – Gameplay Screen – even more variables)

C#

Player player;
List<Alien> aliens;
List<Bullet> alienBullets;
List<Bullet> playerBullets;

4. Initialize the variables in the class constructor as shown in the following code snippet:
5. Initialize the player variables Width and Height in the `LoadContent` method (after initializing the `ParticleSystem`):

(Code Snippet – Game Development with XNA – Gameplay Screen – Player Initialization in LoadContent method)

```csharp
public override void LoadContent()
{
    ...
    particles = new ParticleSystem(ScreenManager.Game.Content, ScreenManager.SpriteBatch);
    player.Width = tankTexture.Width;
    player.Height = tankTexture.Height;
    base.LoadContent();
}
```

6. The next few code snippets will add the logic of the game. They will change the way “player1” moves in response to the user’s input. Navigate to the `HandleInput` method, and locate following line:
Add the next code snippet after it:

(Code Snippet – Game Development with XNA – Gameplay Screen – Player Movements 1 in HandleInput method)

```csharp
player.Velocity.X = movement;
```

7. Locate following line (in the HandleInput method):

```csharp
//TODO: Update player velocity over X axis #1
```

Add the following code snippet after it:

(Code Snippet – Game Development with XNA – Gameplay Screen – Player Movements 2 in HandleInput method)

```csharp
player.Velocity.X = -1.0f;
```

8. Locate the following line (in HandleInput method):

```csharp
//TODO: Update player velocity over X axis #2
```

Add the following code snippet after it:

(Code Snippet – Game Development with XNA – Gameplay Screen – Player Movements 3 in HandleInput method)

```csharp
player.Velocity.X = 1.0f;
```

9. Locate the following line (in HandleInput method):

```csharp
```
// TODO: Update player velocity over X axis #4

Add the following code snippet after it:

(Code Snippet – Game Development with XNA – Gameplay Screen – Player Movements 4 in HandleInput method)

**C#**

```csharp
player.Velocity.X = MathHelper.Min(input.CurrentGamePadStates[0].ThumbSticks.Left.X * 2.0f, 1.0f);
```

10. Locate following line:

**C#**

```csharp
// TODO: Fire the bullet
```

Change the “if” statement according to the following code snippet:

(Code Snippet – Game Development with XNA – Gameplay Screen – HandleInput firing the bullet code)

**C#**

```csharp
if (player.FireTimer <= 0.0f && player.IsAlive && !gameOver)
{
    Bullet bullet = CreatePlayerBullet();
    bullet.Velocity = new Vector2(0, -256.0f);
    player.FireTimer = 1.0f;

    particles.CreatePlayerFireSmoke(player);
    playerFired.Play();
}
else if (gameOver)
{
    finishCurrentGame();
}
```

11. Create the following method inside the **GameplayScreen** class:

This method will create an instance of Bullet class (defined before). This instance was used in a previous code snippet.

(Code Snippet – Game Development with XNA – Gameplay Screen – CreatePlayerBullet method)

**C#**

```csharp
C#
/// <summary>
/// Returns an instance of a usable player bullet. Prefers reusing an existing (dead) bullet over creating a new instance.
/// </summary>
/// <returns>A bullet ready to place into the world.</returns>
Bullet CreatePlayerBullet()
{
    Bullet b = null;

    for (int i = 0; i < playerBullets.Count; ++i)
    {
        if (playerBullets[i].IsAlive == false)
        {
            b = playerBullets[i];
            break;
        }
    }

    if (b == null)
    {
        b = new Bullet();
        playerBullets.Add(b);
    }

    b.IsAlive = true;

    return b;
}

12. Change the Update method. Add the following blue-highlighted code snippet before the “base.Update(...)” method call:

This block of the code actually provides the “game logic” – it moves the player and calls the methods to update the Aliens and recalculate the Bullets' positions.

(Code Snippet – Game Development with XNA – Gameplay Screen – Update method)

C#

public override void Update(GameTime gameTime, bool otherScreenHasFocus, bool coveredByOtherScreen)
{
    float elapsed = (float)gameTime.ElapsedGameTime.TotalSeconds;

    if (IsActive)
    {
        // Move the player
    }
if (player.IsAlive == true)
{
    player.Position += player.Velocity * 128.0f * elapsed;
    player.FireTimer -= elapsed;
    
    if (player.Position.X <= 0.0f)
        player.Position = new Vector2(0.0f, player.Position.Y);
    
    if (player.Position.X + player.Width >= worldBounds.Right)
        player.Position = new Vector2(worldBounds.Right -
            player.Width, player.Position.Y);
}

Respawn(elapsed);

UpdateAliens(elapsed);

UpdateBullets(elapsed);

CheckHits();

if (player.IsAlive && player.Velocity.LengthSquared() > 0.0f)
    particles.CreatePlayerDust(player);

particles.Update(elapsed);

base.Update(gameTime, otherScreenHasFocus, coveredByOtherScreen);

13. Add following helper methods to the GameplayScreen class:

Note: The following code snippet adds a number of helper methods. The method purposes are
as follows:

Respawn: Checks if the player is “dead”, and the game is not over. If this is the case, it waits for
respawnTimer to finish and creates a new player instance in the middle of the screen.

UpdateBullets: Checks and updates the positions of the Player’s and Alien’s bullets on the
screen.

UpdateAliens: Moves the Aliens and calculates if they should fire a bullet and in which
direction.

CheckHits: Checks for all bullet and player/alien collisions. In addition, it handles game logic
when a hit occurs, such as killing something, adding to the score, ending the game, etc.
C#  

/// <summary>
/// Handles respawning the player if we are playing a game and the player is dead.
/// </summary>
/// <param name="elapsed">Time elapsed since Respawn was called last.</param>
void Respawn(float elapsed)
{
    if (gameOver)
    {
        return;
    }
    if (!player.IsAlive)
    {
        player.RespawnTimer -= elapsed;
        if (player.RespawnTimer <= 0.0f)
        {
            // See if there are any bullets close...
            int left = worldBounds.Width / 2 - tankTexture.Width / 2 - 8;
            int right = worldBounds.Width / 2 + tankTexture.Width / 2 + 8;

            for (int i = 0; i < alienBullets.Count; ++i)
            {
                if (alienBullets[i].IsAlive == false)
                {
                    continue;
                }

                if (alienBullets[i].Position.X >= left ||
                    alienBullets[i].Position.X <= right)
                {
                    return;
                }
            }
            player.IsAlive = true;
            player.Position = new Vector2(worldBounds.Width / 2 - player.Width / 2, worldBounds.Bottom - groundTexture.Height + 2 - player.Height);
            player.Velocity = Vector2.Zero;
            player.Lives--;
        }
    }
}

/// <summary>
/// Moves all of the bullets (player and alien) and prunes "dead" bullets.
/// </summary>
/// <param name="elapsed"></param>
void UpdateBullets(float elapsed)
```csharp
for (int i = 0; i < playerBullets.Count; ++i)
{
    if (playerBullets[i].IsAlive == false)
        continue;

    playerBullets[i].Position += playerBullets[i].Velocity * elapsed;
    if (playerBullets[i].Position.Y < -32)
    {
        playerBullets[i].IsAlive = false;
        hitStreak = 0;
    }
}

for (int i = 0; i < alienBullets.Count; ++i)
{
    if (alienBullets[i].IsAlive == false)
        continue;

    alienBullets[i].Position += alienBullets[i].Velocity * elapsed;
    if (alienBullets[i].Position.Y > worldBounds.Height -
        groundTexture.Height - laserTexture.Height)
        alienBullets[i].IsAlive = false;
}

/// <summary>
/// Moves the aliens and performs their "thinking" by determining if they
/// should shoot and where.
/// </summary>
/// <param name="elapsed">The elapsed time since UpdateAliens was called last.</param>
private void UpdateAliens(float elapsed)
{
    // See if it's time to spawn an alien;
    alienSpawnTimer -= elapsed;
    if (alienSpawnTimer <= 0.0f)
    {
        SpawnAlien();
        alienSpawnTimer += alienSpawnRate;
    }

    for (int i = 0; i < aliens.Count; ++i)
    { 
        if (aliens[i].IsAlive == false)
            continue;
```
aliens[i].Position += aliens[i].Velocity * elapsed;
    if ((aliens[i].Position.X < -aliens[i].Width - 64 &&
    aliens[i].Velocity.X < 0.0f) ||
    (aliens[i].Position.X > worldBounds.Width + 64 &&
    aliens[i].Velocity.X > 0.0f))
    {
        aliens[i].IsAlive = false;
        continue;
    }

    aliens[i].FireTimer -= elapsed;

    if (aliens[i].FireTimer <= 0.0f && aliens[i].FireCount > 0)
    {
        if (player.IsAlive)
        {
            Bullet bullet = CreateAlienBullet();
            if ((float)random.NextDouble() <= aliens[i].Accuracy)
            {
                bullet.Velocity = Vector2.Normalize(player.Position - aliens[i].Position) * 64.0f;
            }
            else
            {
                bullet.Velocity = new Vector2(-8.0f + 16.0f *
                (float)random.NextDouble(), 64.0f);
            }
            alienFired.Play();
        }
        aliens[i].FireCount--;
    }

    /// <summary>
    /// Performs all bullet and player/alien collision detection. Also handles
game logic
    /// when a hit occurs, such as killing something, adding score, ending the
game, etc.
    /// </summary>
    void CheckHits()
if (gameOver)
    return;

for (int i = 0; i < playerBullets.Count; ++i)
{
    if (playerBullets[i].IsAlive == false)
        continue;

    for (int a = 0; a < aliens.Count; ++a)
    {
        if (aliens[a].IsAlive == false)
            continue;

        if ((playerBullets[i].Position.X >= aliens[a].Position.X
                && playerBullets[i].Position.X <= aliens[a].Position.X + aliens[a].Width)
        {
            playerBullets[i].IsAlive = false;
            aliens[a].IsAlive = false;

            hitStreak++;

            player.Score += aliens[a].Score * (hitStreak / 5 + 1);

            if (player.Score > highScore)
                highScore = player.Score;

            if (player.Score > nextLife)
            {
                player.Lives++;
                nextLife += nextLife;
            }

            levelKillCount--;
            if (levelKillCount <= 0)
                AdvanceLevel();


            alienDied.Play();
        }
    }
}

if (player.IsAlive == false)
return;

    for (int i = 0; i < alienBullets.Count; ++i)
    {
        if (alienBullets[i].IsAlive == false)
            continue;

        if ((alienBullets[i].Position.X >= player.Position.X + 2 &&
            alienBullets[i].Position.X <= player.Position.X + player.Width - 2) &&
        {
            alienBullets[i].IsAlive = false;

            player.IsAlive = false;
            hitStreak = 0;

            player.RespawnTimer = 3.0f;
            particles.CreatePlayerExplosion(new Vector2(player.Position.X +
                                                        player.Width / 2, player.Position.Y + player.Height / 2));
            playerDied.Play();

            if (player.Lives <= 0)
            {
                gameOver = true;
            }
        }
    }
}

/// <summary>
/// Advances the difficulty of the game one level.
/// </summary>
void AdvanceLevel()
{
    baseLevelKillCount += 5;
    levelKillCount = baseLevelKillCount;
    alienScore += 25;
    alienSpawnRate -= 0.3f;
    alienMaxAccuracy += 0.1f;
    if (alienMaxAccuracy > 0.75f)
        alienMaxAccuracy = 0.75f;

    alienSpeedMin *= 1.35f;
    alienSpeedMax *= 1.35f;
14. And the following code snippet to the `GameplayScreen` class:

```csharp
/// <summary>
/// Returns an instance of a usable alien bullet. Prefers reusing an existing (dead)
/// bullet over creating a new instance.
/// </summary>
/// <returns>A bullet ready to place into the world.</returns>
Bullet CreateAlienBullet()
{
    Bullet b = null;
    for (int i = 0; i < alienBullets.Count; ++i)
    {
        if (alienBullets[i].IsAlive == false)
        {
            b = alienBullets[i];
            break;
        }
    }

    return b;
}
```
if (b == null)
{
    b = new Bullet();
    alienBullets.Add(b);
}

b.IsAlive = true;
return b;

/// <summary>
/// Creates an instance of an alien, sets the initial state, and places it into the world.
/// </summary>
private void SpawnAlien()
{
    Alien newAlien = CreateAlien();

    if (random.Next(2) == 1)
    {
        newAlien.Position.X = -64.0f;
        newAlien.Velocity.X = random.Next((int)alienSpeedMin, (int)alienSpeedMax);
    }
    else
    {
        newAlien.Position.X = worldBounds.Width + 32;
        newAlien.Velocity.X = -random.Next((int)alienSpeedMin, (int)alienSpeedMax);
    }

    newAlien.Position.Y = 24.0f + 80.0f * (float)random.NextDouble();

    // Aliens
    if (transitionFactor > 0.0f)
    {
        switch (random.Next(4))
        {
            case 0:
                newAlien.Texture = badguy_blue;
                break;
            case 1:
                newAlien.Texture = badguy_red;
                break;
        }
    }
```csharp
    case 2:
        newAlien.Texture = badguy_green;
        break;
    case 3:
        newAlien.Texture = badguy_orange;
        break;
    }
}
else
{
    newAlien.Texture = alienTexture;
}
newAlien.Width = newAlien.Texture.Width;
newAlien.Height = newAlien.Texture.Height;
newAlien.IsAlive = true;
newAlien.Score = alienScore;

float duration = screenHeight / newAlien.Velocity.Length();

newAlien.FireTimer = duration * (float)random.NextDouble();
newAlien.FireCount = 1;

newAlien.Accuracy = alienMaxAccuracy;
}

/// <summary>
/// Returns an instance of a usable alien instance. Prefers reusing an existing (dead) alien over creating a new instance.
/// </summary>
/// <returns>An alien ready to place into the world.</returns>
Alien CreateAlien()
{
    Alien b = null;

    for (int i = 0; i < aliens.Count; ++i)
    {
        if (aliens[i].IsAlive == false)
        {
            b = aliens[i];
            break;
        }
    }

    if (b == null)
    {
        b = new Alien();
    }
```
aliens.Add(b);
}

b.IsAlive = true;
return b;

15. Navigate to Draw method and add the following blue-highlighted code snippet between the calls to Screen.SpriteBatch.Begin() and Screen.SpriteBatch.End():

This change to the Draw method will call to the helper methods in order to draw changes calculated by Update on the screen.

(Code Snippet – Game Development with XNA – Gameplay Screen – Draw method update)

C#

public override void Draw(GameTime gameTime)
{
    float elapsedTime = (float)gameTime.ElapsedGameTime.TotalSeconds;

    ScreenManager.SpriteBatch.Begin();

    DrawBackground(elapsedTime);
    DrawAliens();
    DrawPlayer();
    DrawBullets();
    particles.Draw();
    DrawForeground(elapsedTime);
    DrawHud();

    ScreenManager.SpriteBatch.End();
}

16. Add the following methods after the Draw method:

**Note:** The following code snippet adds a number of drawing-related helper methods. The method purposes are as follows:

*DrawPlayer:* Draws the player’s tank.

*DrawAliens:* Draws all aliens.

*DrawBullets:* Draws all bullets (both the player’s and the alien’s).

*DrawForeground:* Draws the clouds as the foreground and moves them.
DrawBackground: Draws the grass, hills, mountains, and sun/moon. Also handles transitioning between day and night.

DrawHud: Draws the score elements, lives remaining, and the “GAME OVER” when needed.

DrawString: Generic method to draw shadowed text.

(Code Snippet – Game Development with XNA – Gameplay Screen – Draw methods)

C#

/// <summary>
/// Draws the player's tank
/// </summary>
void DrawPlayer()
{
    if (!gameOver && player.IsAlive)
    {
        ScreenManager.SpriteBatch.Draw(tankTexture, player.Position, Color.White);
    }
}

/// <summary>
/// Draws all of the aliens.
/// </summary>
void DrawAliens()
{
    for (int i = 0; i < aliens.Count; ++i)
    {
        if (aliens[i].IsAlive)
    }
}

/// <summary>
/// Draw both the player and alien bullets.
/// </summary>
private void DrawBullets()
{
    for (int i = 0; i < playerBullets.Count; ++i)
    {
        if (playerBullets[i].IsAlive)
        ScreenManager.SpriteBatch.Draw(bulletTexture, playerBullets[i].Position, Color.White);
    }
for (int i = 0; i < alienBullets.Count; ++i)
{
    if (alienBullets[i].IsAlive)
    {
        ScreenManager.SpriteBatch.Draw(laserTexture, alienBullets[i].Position, Color.White);
    }
}

/// <summary>
/// Draw the foreground, which is basically the clouds. I think I had planned on one point
/// having foreground grass that was drawn in front of the tank.
/// </summary>
/// <param name="elapsedTime">The elapsed time since last Draw</param>
private void DrawForeground(float elapsedTime)
{
    // Move the clouds. Movement seems like an Update thing to do, but this animations
    // have no impact over gameplay.
    cloud1Position += new Vector2(24.0f, 0.0f) * elapsedTime;
    if (cloud1Position.X > screenWidth)
    {
        cloud1Position.X = -cloud1Texture.Width * 2.0f;
    }
    cloud2Position += new Vector2(16.0f, 0.0f) * elapsedTime;
    if (cloud2Position.X > screenWidth)
    {
        cloud2Position.X = -cloud1Texture.Width * 2.0f;
    }

    ScreenManager.SpriteBatch.Draw(cloud1Texture, cloud1Position, Color.White);
    ScreenManager.SpriteBatch.Draw(cloud2Texture, cloud2Position, Color.White);
}

/// <summary>
/// Draw the grass, hills, mountains, and sun/moon. Handle transitioning
/// between day and night as well.
/// </summary>
/// <param name="elapsedTime">The elapsed time since last Draw</param>
private void DrawBackground(float elapsedTime)
{
    transitionFactor += transitionRate * elapsedTime;
    if (transitionFactor < 0.0f)
    {
        transitionFactor = 0.0f;
        transitionRate = 0.0f;
    }
if (transitionFactor > 1.0f)
{
    transitionFactor = 1.0f;
    transitionRate = 0.0f;
}

Vector3 day = Color.White.ToVector3();
Vector3 night = new Color(80, 80, 180).ToVector3();
Vector3 dayClear = Color.CornflowerBlue.ToVector3();
Vector3 nightClear = night;

Color clear = new Color(Vector3.Lerp(dayClear, nightClear, transitionFactor));
Color tint = new Color(Vector3.Lerp(day, night, transitionFactor));

    // Clear the background, using the day/night color
    ScreenManager.Game.GraphicsDevice.Clear(clear);

    // Draw the mountains
    ScreenManager.SpriteBatch.Draw(mountainsTexture, new Vector2(0, screenHeight - mountainsTexture.Height), tint);

    // Draw the hills
    ScreenManager.SpriteBatch.Draw(hillsTexture, new Vector2(0, screenHeight - hillsTexture.Height), tint);

    // Draw the ground
    ScreenManager.SpriteBatch.Draw(groundTexture, new Vector2(0, screenHeight - groundTexture.Height), tint);

    // Draw the sun or moon (based on time)
    ScreenManager.SpriteBatch.Draw(sunTexture, sunPosition, (new Color(255, 255, 255, (byte)(255.0f * (1.0f - transitionFactor)))));
    ScreenManager.SpriteBatch.Draw(moonTexture, sunPosition, (new Color(255, 255, 255, (byte)(255.0f * transitionFactor))));
}

/// <summary>
/// Draw the hud, which consists of the score elements and the GAME OVER tag.
/// </summary>
void DrawHud()
{
    float scale = 2.0f;
    if (gameOver)
{  
    Vector2 size = menuFont.MeasureString("GAME OVER");  
}

else {
    int bonus = 100 * (hitStreak / 5);
    // Score
    DrawString(scoreFont, "SCORE: " + player.Score.ToString(System.Globalization.CultureInfo.CurrentCulture) + bonusString, new Vector2(leftOffset, topOffset), Color.Yellow, scale);

    Vector2 size = scoreFont.MeasureString(text);
    size *= scale;

    // Lives
    DrawString(scoreFont, text, new Vector2(screenWidth - leftOffset - (int)size.X, topOffset), Color.Yellow, scale);

    DrawString(scoreFont, "LEVEL: " + (((baseLevelKillCount - 5) / 5) + 1).ToString(System.Globalization.CultureInfo.CurrentCulture), new Vector2(leftOffset, screenHeight - bottomOffset), Color.Yellow, scale);

    text = "HIGH SCORE: " + highScore.ToString(System.Globalization.CultureInfo.CurrentCulture);
    size = scoreFont.MeasureString(text);

    DrawString(scoreFont, text, new Vector2(screenWidth - leftOffset - (int)size.X * 2, screenHeight - bottomOffset), Color.Yellow, scale);
}

/// <summary>
/// A simple helper to draw shadowed text.
/// </summary>
void DrawString(SpriteFont font, string text,
vector2 position, color color) {
    screenmanager.spritebatch.drawstring(font, text, new vector2(position.X + 1, position.Y + 1), color.black);
    screenmanager.spritebatch.drawstring(font, text, position, color);
}

/// <summary>
/// A simple helper to draw shadowed text.
/// </summary>
void drawstring(spritemenu font, string text, vector2 position, color color, float fontscale) {
    screenmanager.spritebatch.drawstring(font, text, new vector2(position.X + 1, position.Y + 1), color.black, 0, new vector2(0, font.lineSpacing / 2), fontscale, spriteeffects.none, 0);
    screenmanager.spritebatch.drawstring(font, text, position, color, 0, new vector2(0, font.lineSpacing / 2), fontscale, spriteeffects.none, 0);
}

17. Locate the LoadContent method and add the following code snippet after the base.LoadContent() method call:

C#

public override void LoadContent()
{
    ...
    player.Width = tankTexture.Width;
    player.Height = tankTexture.Height;

    base.LoadContent();

    LoadHighscore();
    Start();
}

18. Locate the UnloadContent method and add the following code snippet before the "particles = null;" statement:

C#

public override void UnloadContent()
{
    SaveHighscore();
}
19. Create a new region with loading/unloading high scores logic. To save/load data to the Windows Phone file system, you have to use the Isolated Storage that is provided for each application. Use the following code snippet to create this logic in the **GameplayScreen** class:

(Code Snippet – *Game Development with XNA – Gameplay Screen – Highscore storage methods*)

```csharp
#region Highscore loading/saving logic
/// <summary>
/// Saves the current highscore to a text file. The StorageDevice was selected during screen loading.
/// </summary>
private void SaveHighscore()
{
    using (IsolatedStorageFile isf = IsolatedStorageFile.GetUserStoreForApplication())
    {
        using (IsolatedStorageFileStream isfs = new IsolatedStorageFileStream("highscores.txt", FileMode.Create, isf))
        {
            using (StreamWriter writer = new StreamWriter(isfs))
            {
                writer.Write(highScore.ToString(System.Globalization.CultureInfo.InvariantCulture));
                writer.Flush();
                writer.Close();
            }
        }
    }
}

/// <summary>
/// Loads the high score from a text file. The StorageDevice was selected during the loading screen.
/// </summary>
private void LoadHighscore()
{
    using (IsolatedStorageFile isf = IsolatedStorageFile.GetUserStoreForApplication())
    {
        if (isf.Exists("highscores.txt"))
        {
            using (IsolatedStorageFileStream isfs = new IsolatedStorageFileStream("highscores.txt", FileMode.Open, isf))
            {
                using (StreamReader reader = new StreamReader(isfs))
                {
                    highScore = int.Parse(reader.ReadToEnd());
                }
            }
        }
    }
}
```
{ using (IsolatedStorageFileStream isfs = new IsolatedStorageFileStream("highscores.txt", FileMode.Open, isf))
  {
    using (StreamReader reader = new StreamReader(isfs))
    {
      try
      {
        highScore = Int32.Parse(reader.ReadToEnd(), System.Globalization.CultureInfo.InvariantCulture);
      }
      catch (FormatException)
      {
        highScore = 10000;
      }
      finally
      {
        if (reader != null)
          reader.Close();
      }
    }
  }
}
#endregion

20. Create the `Start` method to start a new game inside the `GameplayScreen` class:

(Code Snippet – Game Development with XNA – Gameplay Screen – Start method)

C#

    /// <summary>
    /// Starts a new game session, setting all game states to initial values.
    /// </summary>
    void Start()
    {
      if (gameOver)
      {
        player.Score = 0;
        player.Lives = 3;
        player.RespawnTimer = 0.0f;
        gameOver = false;
        aliens.Clear();
        alienBullets.Clear();
      }
21. Open **ParticleSystem.cs**.

22. Add following *using* statement:

   (Code Snippet – *Game Development with XNA – ParticleSystem – using statement*)

   ```c#
   using AlienGame;
   ```

23. During this step you will add two methods to create the mud/dust effect when the player’s tank moves and fire effect when the player fires a bullet. Add the following methods to the **ParticleSystem** class:

   (Code Snippet – *Game Development with XNA – ParticleSystem – Helper player effects methods*)

   ```c#
   /// <summary>
   /// Creates the mud/dust effect when the player moves.
   /// </summary>
   /// <param name="position">Where on the screen to create the effect.</param>
   public void CreatePlayerDust(Player player)
   {
       for (int i = 0; i < 2; ++i)
   ```
```csharp
{
    Particle p = CreateParticle();
    p.Texture = smoke;
    p.Color = new Color(125, 108, 43);
    p.Alpha = 1.0f;
    p.AlphaRate = -2.0f;
    p.Life = 0.5f;
    p.Rotation = 0.0f;
    p.RotationRate = -2.0f + 4.0f * (float)random.NextDouble();
    p.Scale = 0.25f;
    p.ScaleRate = 0.5f;
    p.Velocity.X = -4 + 8.0f * (float)random.NextDouble();
    p.Velocity.Y = -8 + 4.0f * (float)random.NextDouble();
}

/// <summary>
/// Creates the effect for when the player fires a bullet.
/// <summary>
/// <param name="position">Where on the screen to create the effect.</param>
public void CreatePlayerFireSmoke(Player player)
{
    for (int i = 0; i < 8; ++i)
    {
        Particle p = CreateParticle();
        p.Texture = smoke;
        p.Color = Color.White;
        p.Alpha = 1.0f;
        p.AlphaRate = -1.0f;
        p.Life = 1.0f;
        p.Rotation = 0.0f;
        p.RotationRate = -2.0f + 4.0f * (float)random.NextDouble();
        p.Scale = 0.25f;
        p.ScaleRate = 0.25f;
        p.Velocity.X = -4 + 8.0f * (float)random.NextDouble();
    }
}
```
Note: To use the desktop keyboard for moving the vehicle in the Emulator you need to first press the PAUSE/BREAK key. This will toggle between the Emulator Software Input Panel (SIP) and the desktop keyboard, which cannot be active at the same time. This is a known issue and will be fixed in future releases.

![Finished game](image)

**Figure 28**  
*Finished game*

This step concludes the lab.

During this task you created the AlienGame logic, including the player’s and alien’s movement calculations, hit detection, screen draw, and others.

Note: The complete solution for this exercise is located at the `Source\Ex1-AlienGame\End` folder of this lab.
Summary

This lab introduced you to developing applications for the Windows Phone platform using the XNA Framework. From this lab you created an XNA Game Studio game for Windows Phone, loaded the game’s resources, took care of the input, updated the game state, and added game specific logic.

By completing this hands-on lab, you also became familiar with the tools required to create and test a Windows Phone XNA Game Studio game. In this lab, you created a new XNA Game Studio game for Windows Phone application using Microsoft Visual Phone Developer 2010 Express, and then used this free tool to create the application logic and the layout of the user interface.