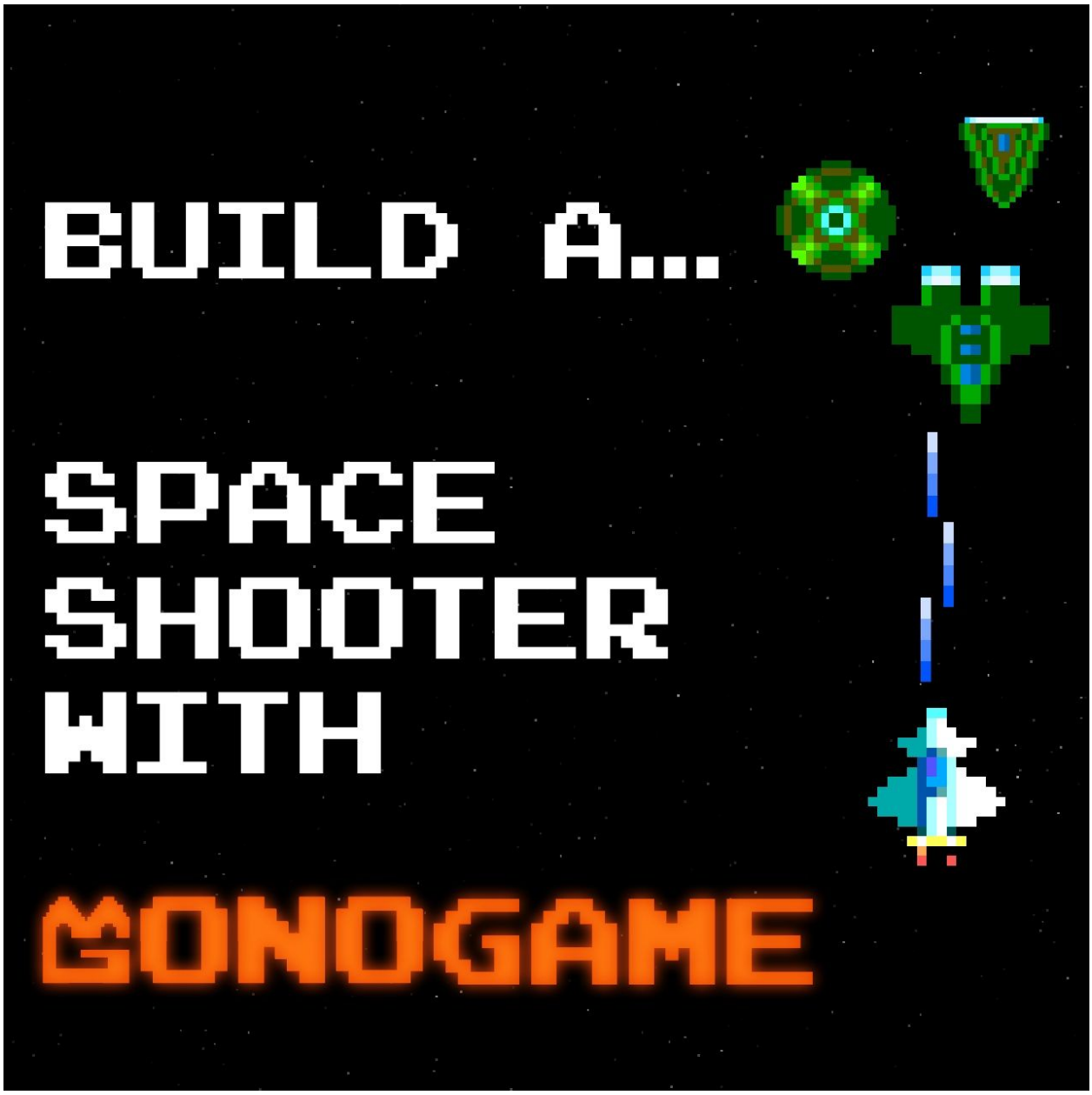


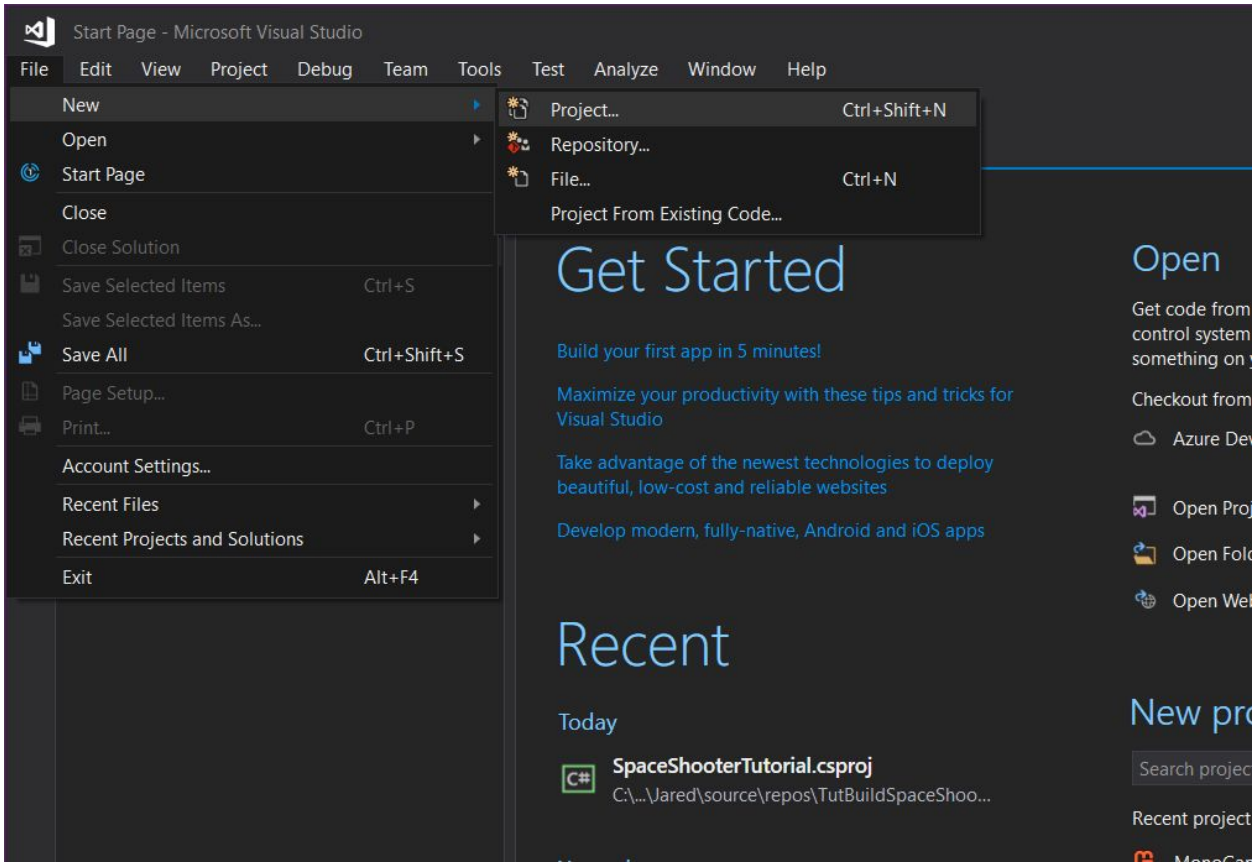
Build a Space Shooter with MonoGame
Version 1.0
Written by Jared York

INTRODUCTION

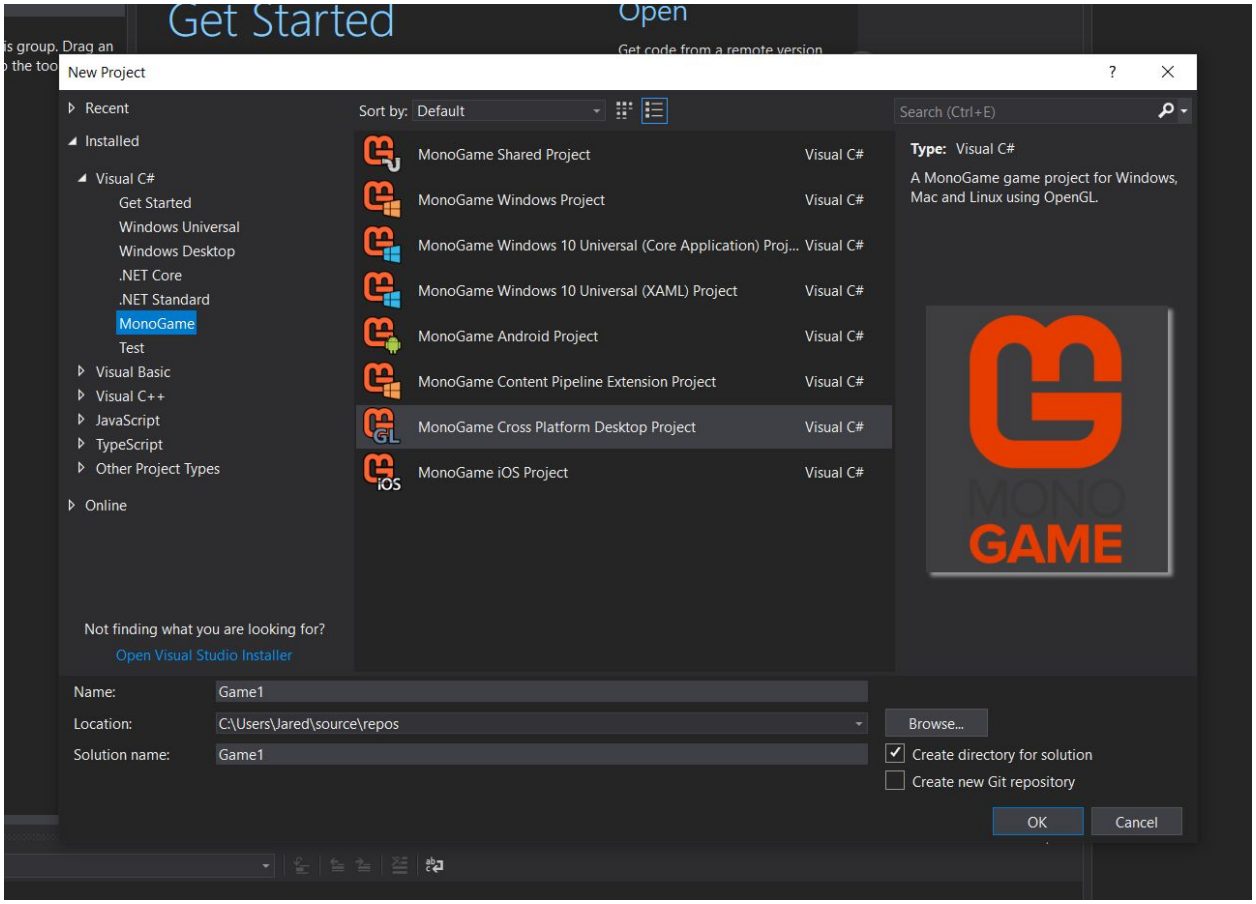
In this course, we will be creating a space shoot-em-up game with MonoGame! MonoGame is an open-source implementation of XNA, and has a very active community surrounding it. Before we get started, it will be important that you have MonoGame installed. If you need to install MonoGame on Windows, you can check out our installation guide, otherwise the installers for other platforms should be similar. If you are using Mac OS or Linux, you can find the download links on this page (for version 3.7.1 as of May 4, 2019), here.

SETTING UP THE PROJECT

The next thing we have to do is setup the project. In Visual Studio, select File > New > Project…



A new window should pop up allowing you to choose a project template. What we want to do is expand the *Visual C#* dropdown on the left sidebar, then select “MonoGame Cross Platform Desktop Project.” This project template will tell MonoGame to use OpenGL meaning we can run the game on not only Windows, but also Mac and Linux! Your screen should now look similar to:

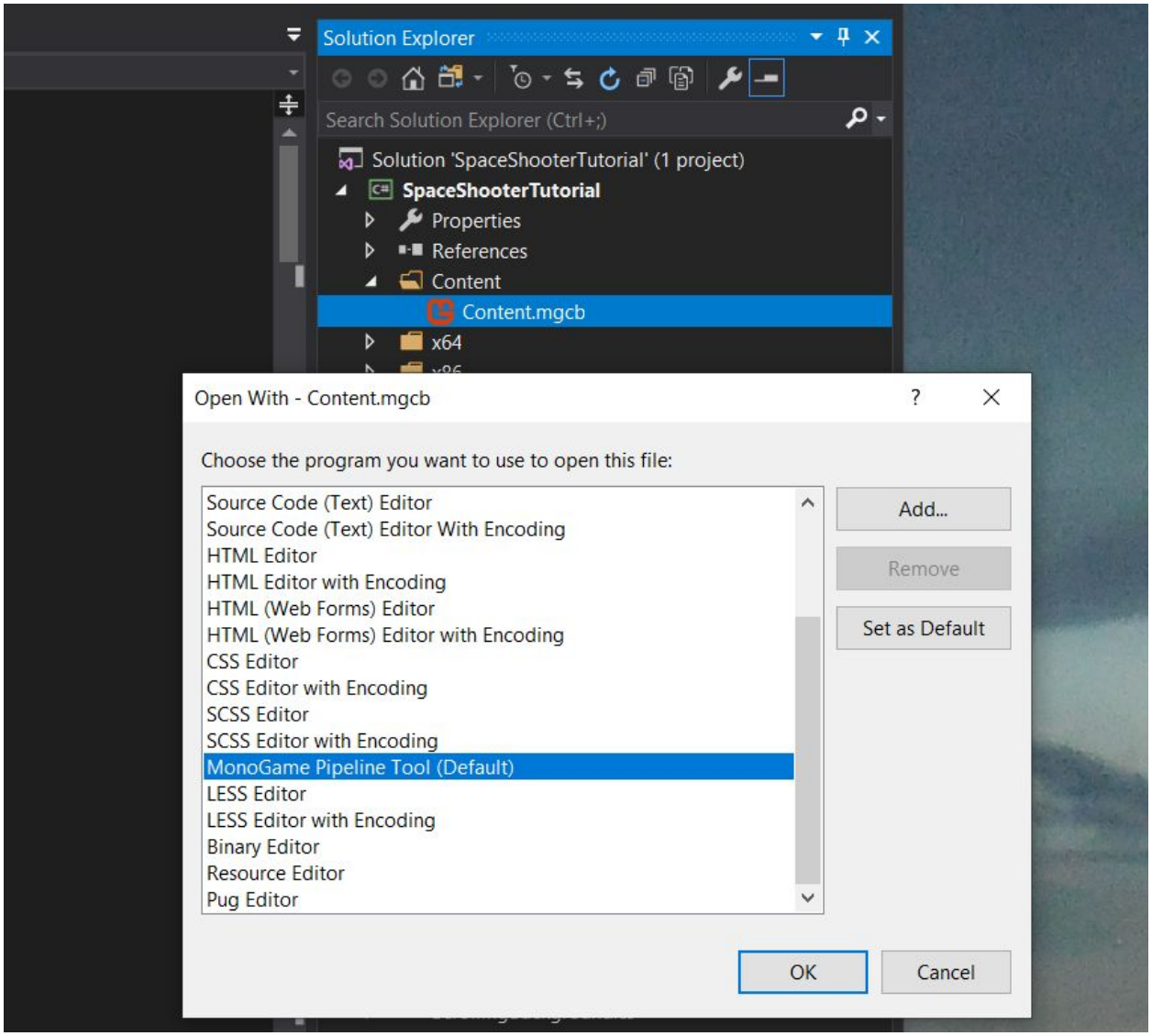


The next step is to name the project. To do this, you can fill out the text box next to the “Name” field on the bottom of this window. Don’t worry about the “Solution name” field, Visual Studio will fill this in for you. Once you’re ready, click the *OK* button.

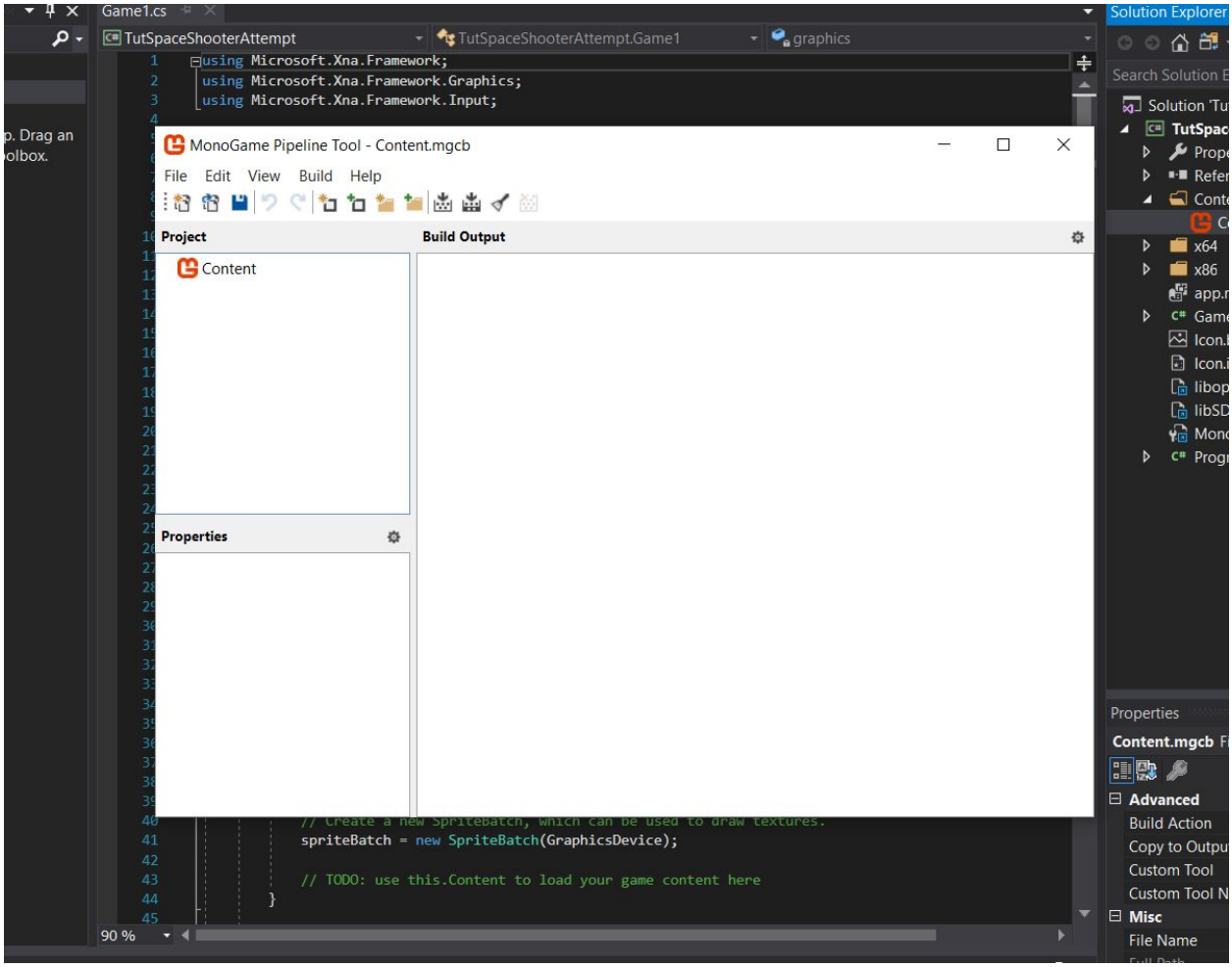
For this course, we will also need to download the content for our game (images and sounds.) The content is freely available for download here.

In order to add our content to the game, we will have to provide it to the MonoGame Pipeline Tool to add it to our project. I would recommend expanding the Content folder of our project (within the Solution Explorer of Visual Studio), right click “Content.mgcb”, then click Open With… A new dialog should appear, where you can click on “MonoGame Pipeline Tool.” Once that option is selected, if I were you, I would set that option as default via the associated button.

At this point, you should see the following in the “Open With” dialog:

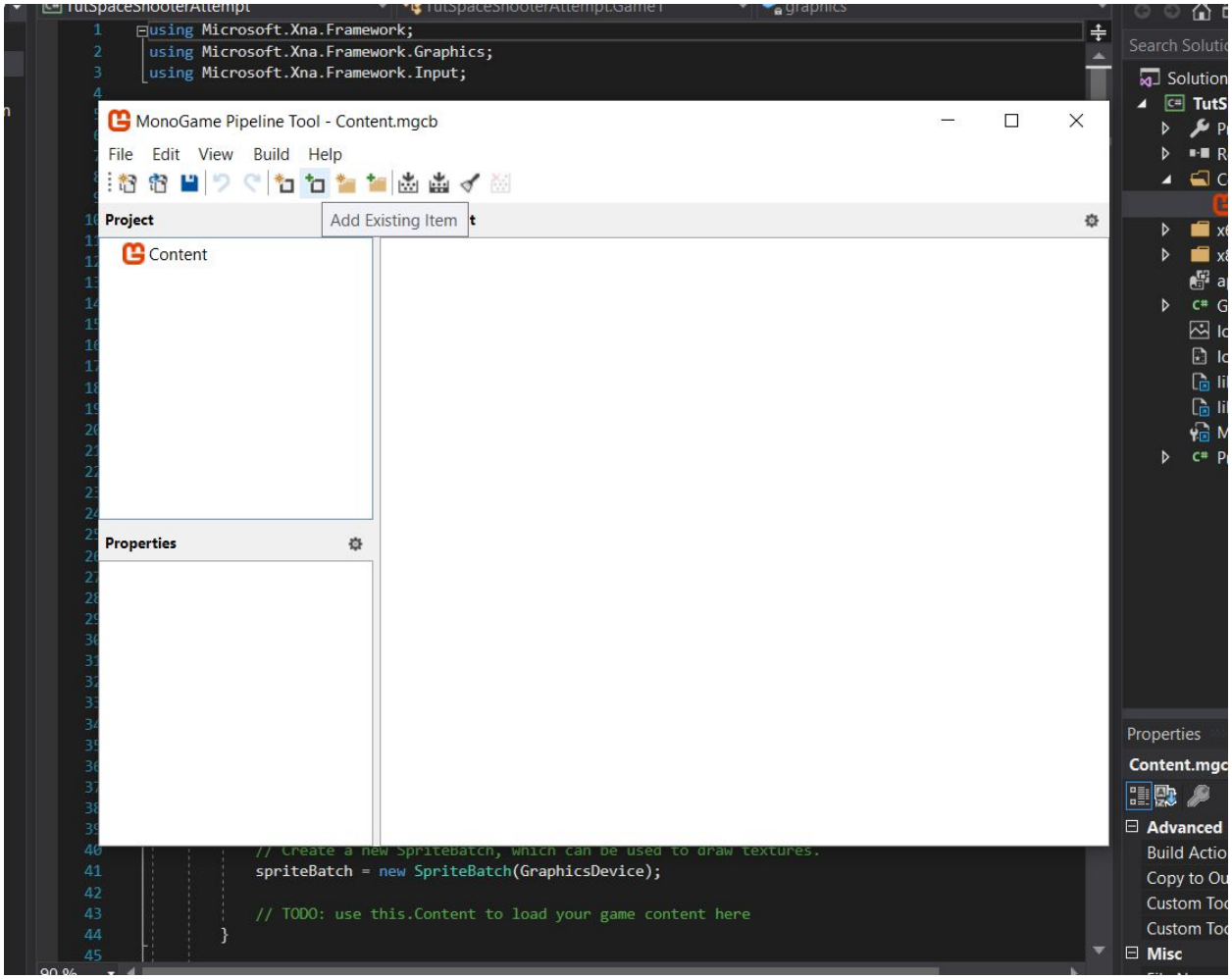


Once that's all set, just click the *OK* button, and you should be good to go! We can now add the content to our game. The first thing we'll need to do now, is double click the "Content.mgcb" file in the Solution Explorer pane. Now the MonoGame Pipeline Tool should open since we've set it as the default application to open *.mgcb* files.

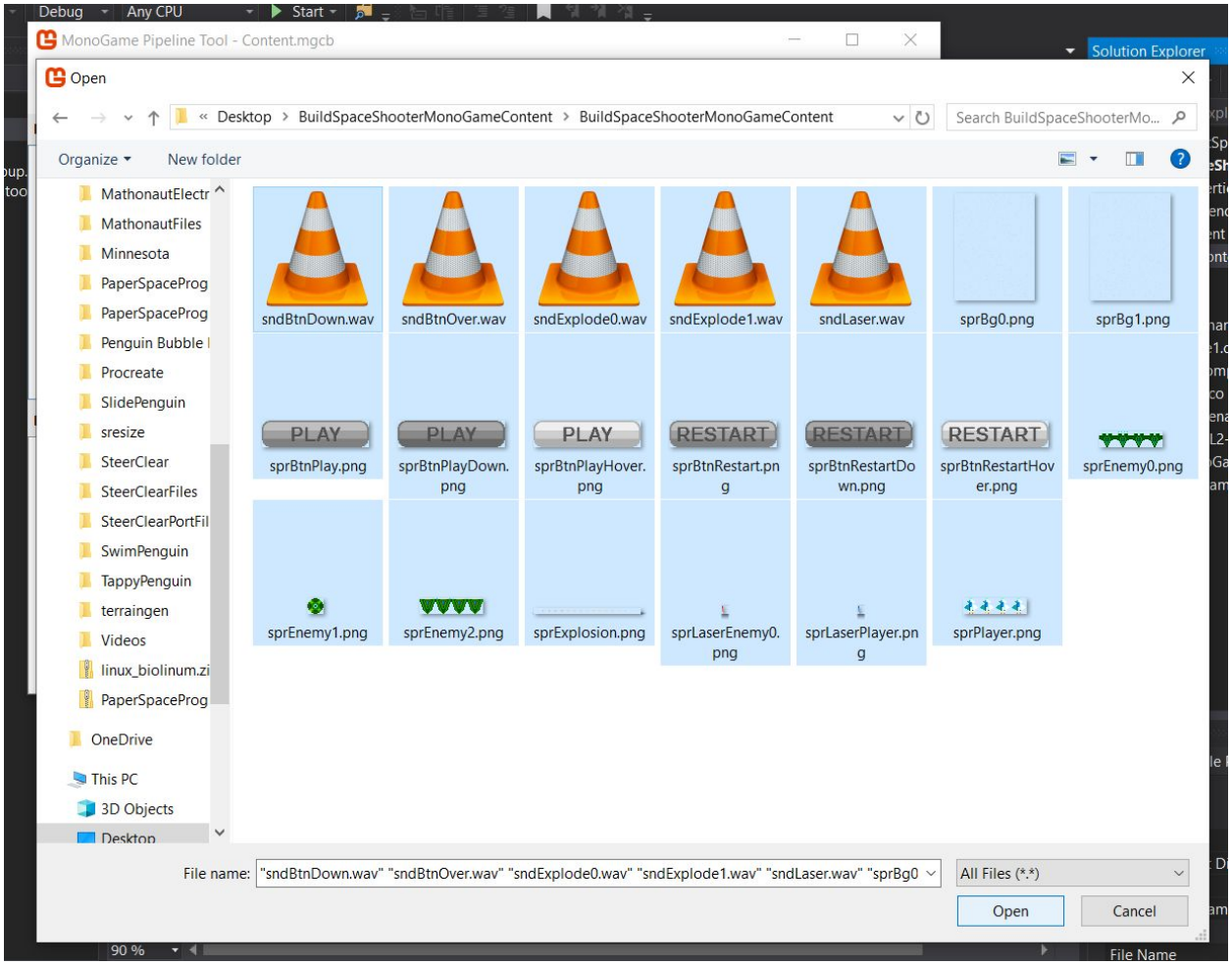


You should see something like the above window once the pipeline tool has opened.

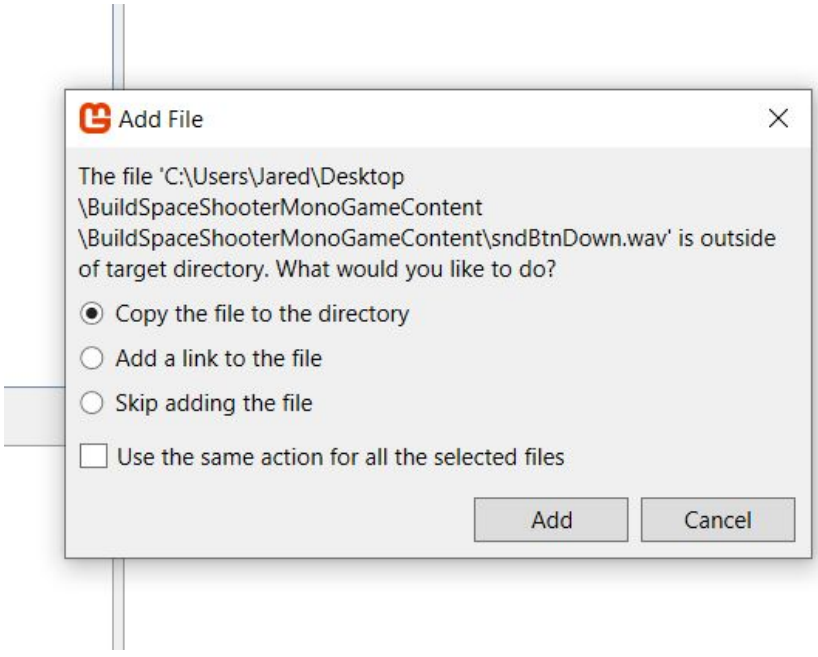
Unfortunately, we won't be able to just select our content, and drag and drop it in. So, we will have to click "Add Existing Item." This button should be on the row of buttons at the top of the window:



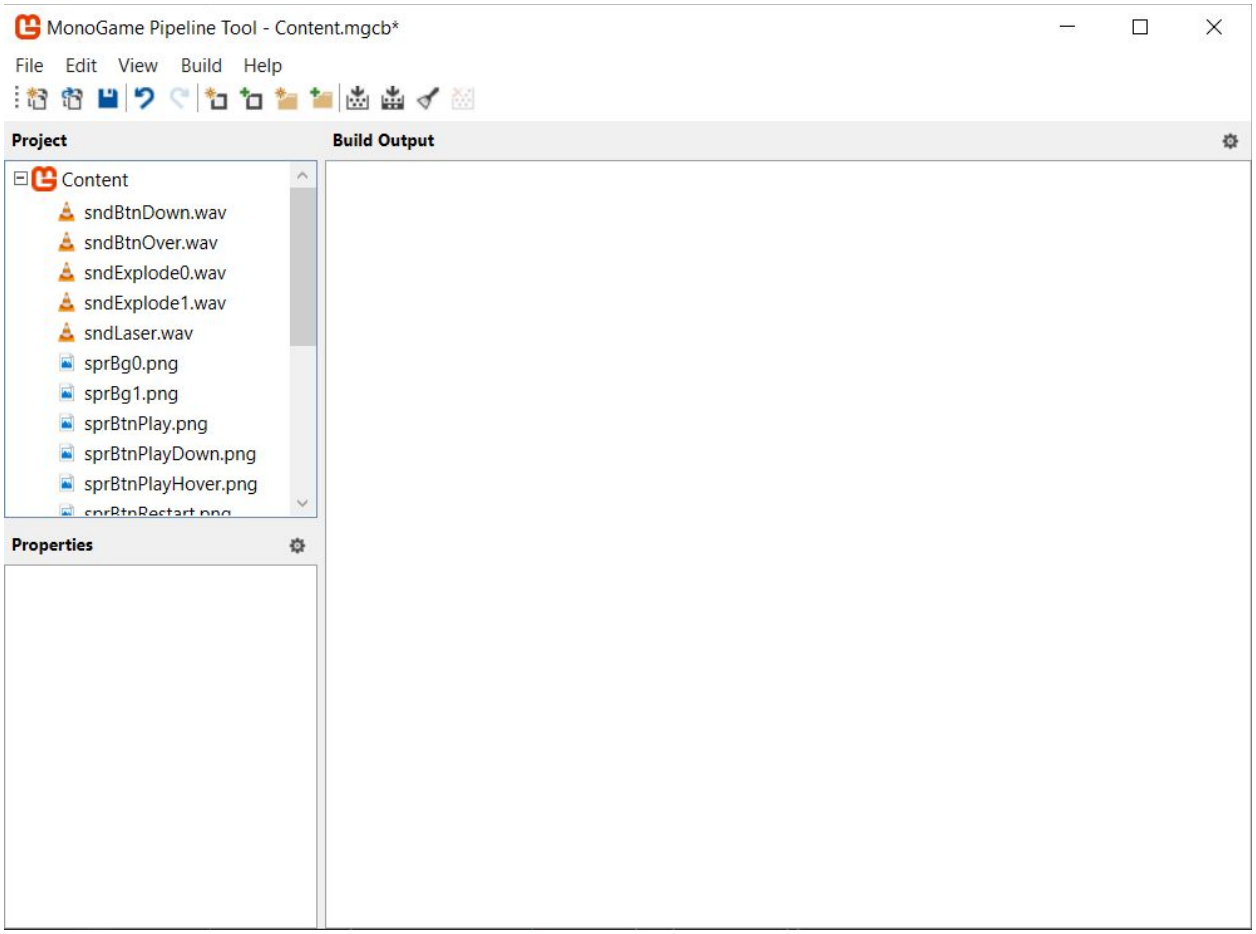
An open dialog should now be displayed. The next step is to find the ZIP file you downloaded containing the content and extract it. Then, click on the newly extracted folder and select all the files like so:



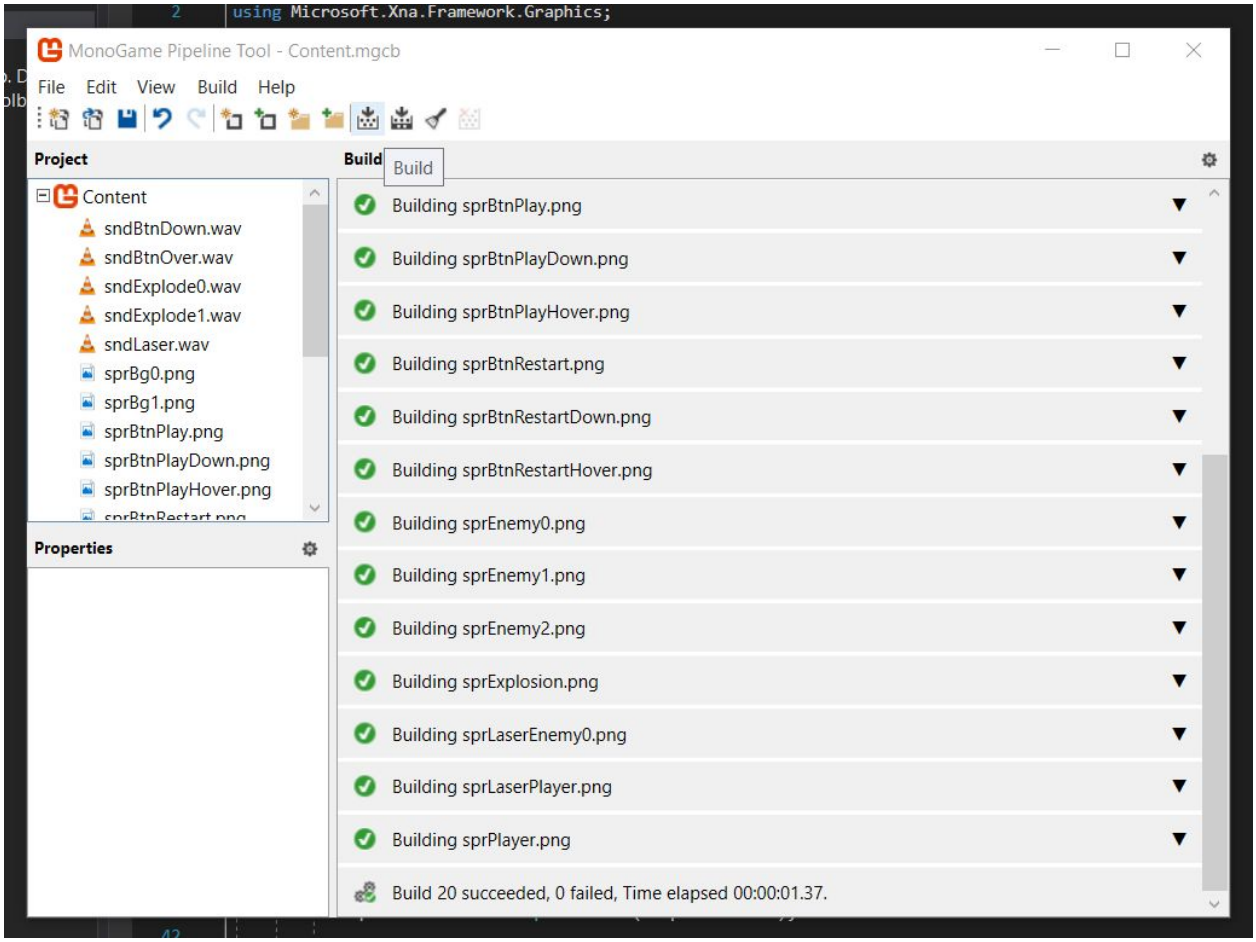
Once you've selected all of the content, click the *Open* button. If all goes right, you should now see another prompt on your screen asking you what you want to do:



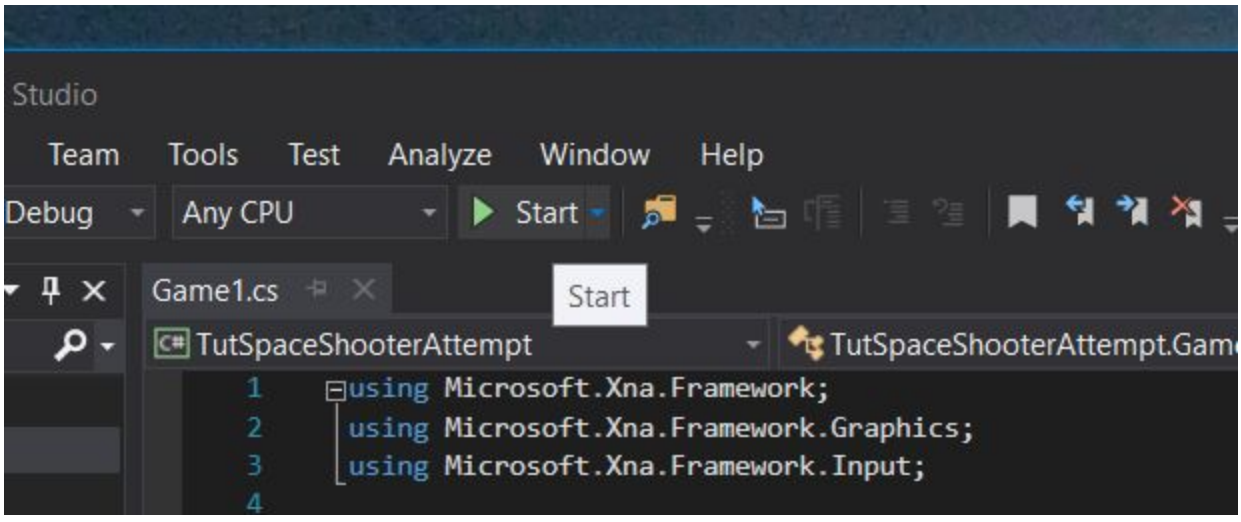
Ensure the option, “Copy the file to the directory” is selected. I would also mark the checkbox labeled, “Use the same action for all the selected files.” After that, click *Add* and the content should then be added to our project’s pipeline!



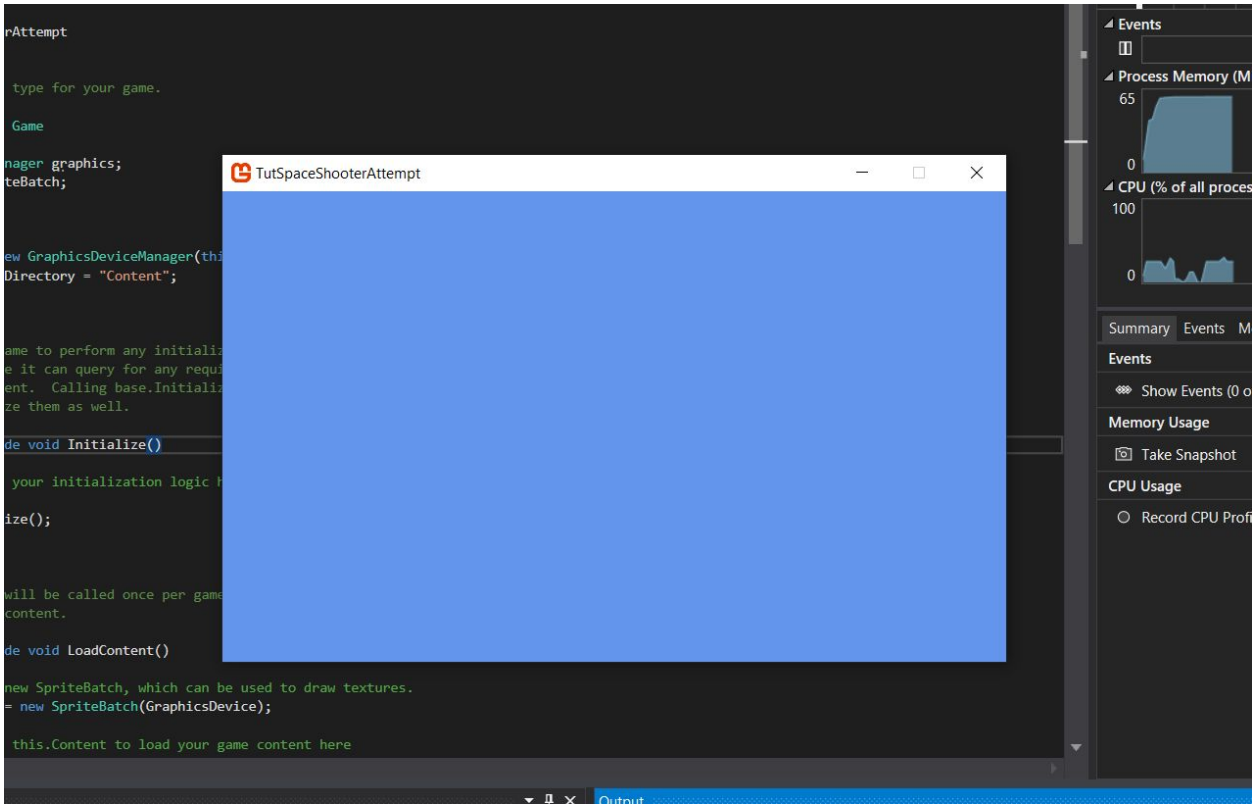
Of course, we still can’t use it in our game yet. Let’s click the *Build* button on the top of the pipeline tool and wait for our content to build.



At this point, we can close the pipeline (for now). We will be coming back to it once we add text to our game. Let's have a go at running the game and see what we get. At the top of Visual Studio, you should see the *Start* button.



If we click the button, our game will compile and we should see the following:



This looks pretty eventful, huh? We'll spruce it up, but we have to get some of the more boring work out of the way first. This includes loading our content.

If *Game1.cs* is not already open in your code window, you can navigate to it via the Solution Explorer. Once *Game1.cs* is displayed, add the following to the *using* statements at the top:

```
using Microsoft.Xna.Framework.Audio;
using System.Collections.Generic;
using System;
```

We will need the audio part of MonoGame in order to load and play our sounds. We also specify that we want to use "System.Collections.Generic". We will need this to create lists to hold objects in our game. We will be using "System" for randomly generating numbers. Before we get too ahead of ourselves, let's define the fields we'll use for referencing our images (textures). Add the following under the *SpriteBatch* definition, "SpriteBatch spriteBatch;":

```
private List<Texture2D> texBgs = new List<Texture2D>();
private Texture2D texBtnPlay;
private Texture2D texBtnPlayDown;
private Texture2D texBtnPlayHover;
```

```
private Texture2D texBtnRestart;
private Texture2D texBtnRestartDown;
private Texture2D texBtnRestartHover;
private Texture2D texPlayer;
private Texture2D texPlayerLaser;
private Texture2D texEnemyLaser;
private List<Texture2D> texEnemies = new List<Texture2D>();
private Texture2D texExplosion;

public SoundEffect sndBtnDown;
public SoundEffect sndBtnOver;
public List<SoundEffect> sndExplode = new List<SoundEffect>();
public SoundEffect sndLaser;

private SpriteFont fontArial;
```

From this point on, I will be calling the images we've added to our pipeline as textures. When we reference textures in our code, they will be the correct type, "Texture2D". This is very important. Notice how all of the texture fields are *private*. We won't be accessing these fields from within other classes, so we don't have to bother making them public.

In order to grab the textures and sounds from our project's pipeline, we will have to interface with the *ContentManager*, which allows MonoGame to interact with the pipeline. In our case, we will be using the *Load* method of the ContentManager, which we can access using the *Content* instance. This might sound pretty confusing. No worries though, we will simply be loading our content like so:

```
<field name> = Content.Load<Texture2D>("filename without extension");
```

Not too bad, right? We will be loading our sound effects and SpriteFonts (our game's font(s)) quite similarly, but instead of using the *Texture2D* class, we will utilize *SoundEffect* and *SpriteFont*. Let's put this into practice! In the *LoadContent* method, right under the line:

```
// TODO: use this.Content to load your game content here
```

Add:

```
// Load textures

for (int i = 0; i < 2; i++)
{
        texBgs.Add(Content.Load<Texture2D>("sprBg" + i));
}
```

```
texBtnPlay = Content.Load<Texture2D>("sprBtnPlay");
texBtnPlayDown = Content.Load<Texture2D>("sprBtnPlayDown");
texBtnPlayHover = Content.Load<Texture2D>("sprBtnPlayHover");

texBtnRestart = Content.Load<Texture2D>("sprBtnRestart");
texBtnRestartDown = Content.Load<Texture2D>("sprBtnRestartDown");
texBtnRestartHover = Content.Load<Texture2D>("sprBtnRestartHover");

texPlayer = Content.Load<Texture2D>("sprPlayer");
texPlayerLaser = Content.Load<Texture2D>("sprLaserPlayer");
texEnemyLaser = Content.Load<Texture2D>("sprLaserEnemy0");

for (int i = 0; i < 3; i++)
{
        texEnemies.Add(Content.Load<Texture2D>("sprEnemy" + i));
}

texExplosion = Content.Load<Texture2D>("sprExplosion");
```

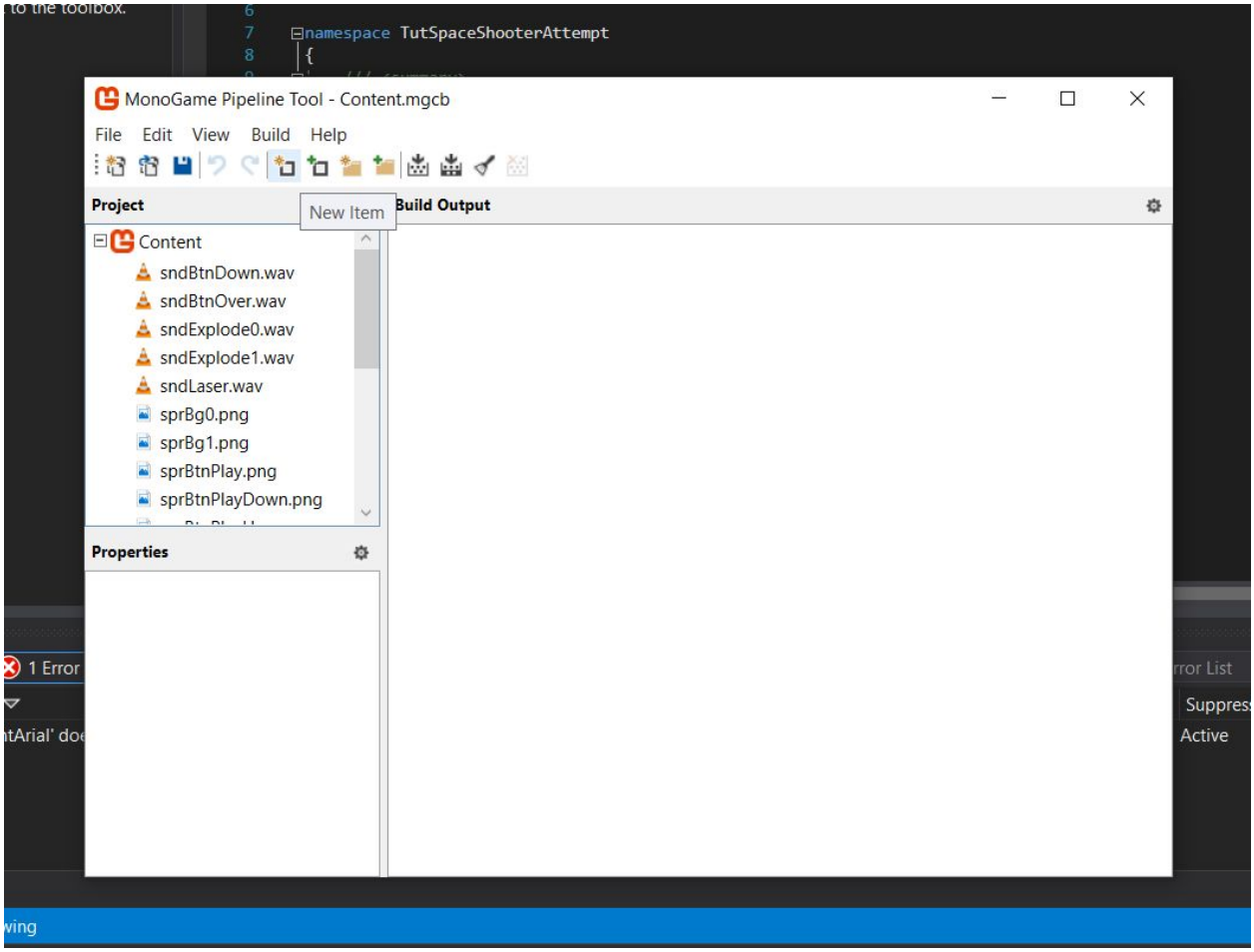
As I mentioned before, we can load our sounds in the same way. Add the following:

```
// Load sounds
sndBtnDown = Content.Load<SoundEffect>("sndBtnDown");
sndBtnOver = Content.Load<SoundEffect>("sndBtnOver");
for (int i = 0; i < 2; i++)
{
        sndExplode.Add(Content.Load<SoundEffect>("sndExplode" + i));
}
sndLaser = Content.Load<SoundEffect>("sndLaser");
```

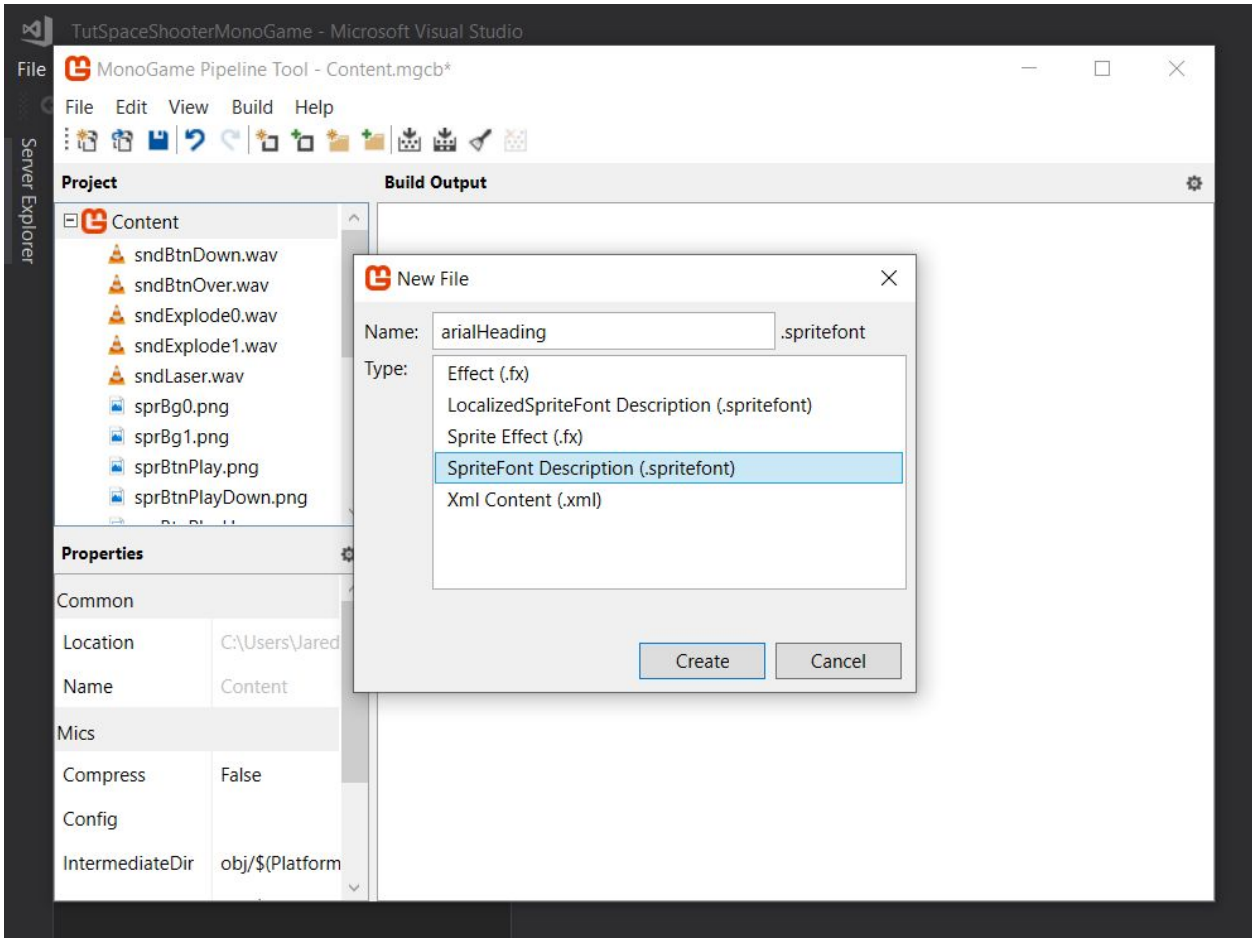
Plus, we can also load our single sprite font we will use:

```
// Load sprite fonts

fontArial = Content.Load<SpriteFont>("arialHeading");
```

If we run the game now, we'll see an error claiming 'arialHeading' does not exist. This is where sprite fonts come into play. Let's open up the content pipeline for our project, and click the "New Item" button in the toolbar.



A popup window should appear prompting you to name and select the type of file you wish to add. Choose "SpriteFont Description (.spritefont)," and name the file, "arialHeading".



Once you've selected the SpriteFont option, and named the file "fontArial," we can click the *Create* button. Our new SpriteFont file will be added to our project list. We will want to increase the font size though, since this font will be utilized for our title. Right click "arialHeading.spritefont" in our pipeline project, then click "Open With". Choose a text editor or code editor of your choice, anything works. Then click *OK* or the equivalent on your operating system, and the file should open in the editor you chose. I myself has chosen to use Visual Studio Code, which is a fantastic, fast code editor. At this point we should be seeing the following text:

```xml
<?xml version="1.0" encoding="utf-8"?>
<!--
This file contains an xml description of a font, and will be read by the XNA
Framework Content Pipeline. Follow the comments to customize the appearance
of the font in your game, and to change the characters which are available to draw
with.
-->
<XnaContent xmlns:Graphics="Microsoft.Xna.Framework.Content.Pipeline.Graphics">
  <Asset Type="Graphics:FontDescription">

    <!--
    Modify this string to change the font that will be imported.
    -->
    <FontName>Arial</FontName>

    <!--
    Size is a float value, measured in points. Modify this value to change
    the size of the font.
    -->
    <Size>12</Size>

    <!--
    Spacing is a float value, measured in pixels. Modify this value to change
    the amount of spacing in between characters.
    -->
    <Spacing>0</Spacing>

    <!--
    UseKerning controls the layout of the font. If this value is true, kerning information
    will be used when placing characters.
    -->
    <UseKerning>true</UseKerning>

    <!--
    Style controls the style of the font. Valid entries are "Regular", "Bold", "Italic",
    and "Bold, Italic", and are case sensitive.
    -->
    <Style>Regular</Style>

    <!--
    If you uncomment this line, the default character will be substituted if you draw
    or measure text that contains characters which were not included in the font.
    -->
    <!-- <DefaultCharacter>*</DefaultCharacter> -->

    <!--
    CharacterRegions control what letters are available in the font. Every
    character from Start to End will be built and made available for drawing. The
    default range is from 32, (ASCII space), to 126, ('~'), covering the basic Latin
    character set. The characters are ordered according to the Unicode standard.
    See the documentation for more information.
    -->
    <CharacterRegions>
      <CharacterRegion>
        <Start>&#32;</Start>
        <End>&#126;</End>
      </CharacterRegion>
    </CharacterRegions>
  </Asset>
</XnaContent>
```

Change the value between the <*Size*> tags from *12* to *32*. We can also change the value of the <*Style*> element to *Bold*. Save this file, and exit out of the editor you were using and let's go back to the pipeline tool.

Build the pipeline project, then let's head back to *Game1.cs* in Visual Studio. Now, we know every game worth it's money has a game state system of some sort. I mean this as in, most games have a main menu, a play state, and a game over state. Something like that. We will have to build this sort of system. For this, we will be utilizing the *enum* type in C#. Just after where we initialize the field *arialHeading*, add the following to define an enum named *GameState*:

```
enum GameState
{
	MainMenu,
	Gameplay,
	GameOver
}
```

We will also want to initialize a property to keep track of the current game state:

```
private GameState _gameState;
```

The next step to setup our game state system is to add some logic to the *Update* method of *Game1.cs*. After the comment, "TODO: Add your update logic here," add the following:

```
switch (_gameState) {
	case GameState.MainMenu:
	{
		UpdateMainMenu(gameTime);
		break;
	}

	case GameState.Gameplay:
	{
		UpdateGameplay(gameTime);
		break;
	}

	case GameState.GameOver:
	{
		UpdateGameOver(gameTime);
		break;
```

```
        }
}
```

The switch statement allows us to check which game state is currently set a little more compactly than an if statement. Let's define the methods that we specified for each case in our switch statement. Add the following methods after the *Update* method but before the *Draw* method:

```
private void UpdateMainMenu(GameTime gameTime) {

}

private void UpdateGameplay(GameTime gameTime) {

}

private void UpdateGameOver(GameTime gameTime) {

}
```

We will also want to add two additional methods for resetting gameplay and changing the game scene:

```
private void resetGameplay()
{

}

private void changeGameState(GameState gameState)
{

}
```

In the Draw method of *Game1.cs*, add the following code after the comment, "TODO: Add your drawing code here":

```
spriteBatch.Begin(SpriteSortMode.Deferred, BlendState.AlphaBlend, SamplerState.PointWrap);

switch (_gameState)
{
        case GameState.MainMenu:
        {
```

```
                DrawMainMenu(spriteBatch);
                break;
        }

        case GameState.Gameplay:
        {
                DrawGameplay(spriteBatch);
                break;
        }

        case GameState.GameOver:
        {
                DrawGameOver(spriteBatch);
                break;
        }
}

spriteBatch.End();
```

It's important to point out that in order to draw anything in MonoGame, you need to have all of your draw calls between the calls: *spriteBatch.Begin* and *spriteBatch.End*. It's also worth mentioning that the last parameter of our *spriteBatch.Begin* call tells the game we don't want to smooth images, in order to preserve our crisp pixel art. While we're at it, change the following line at the top of the *Draw* method:

```
GraphicsDevice.Clear(Color.CornflowerBlue);
```

To

```
GraphicsDevice.Clear(Color.Black);
```

As we did with the update methods for our game states, we will add draw methods for each of our game states. Add the following methods after the *Draw* method:

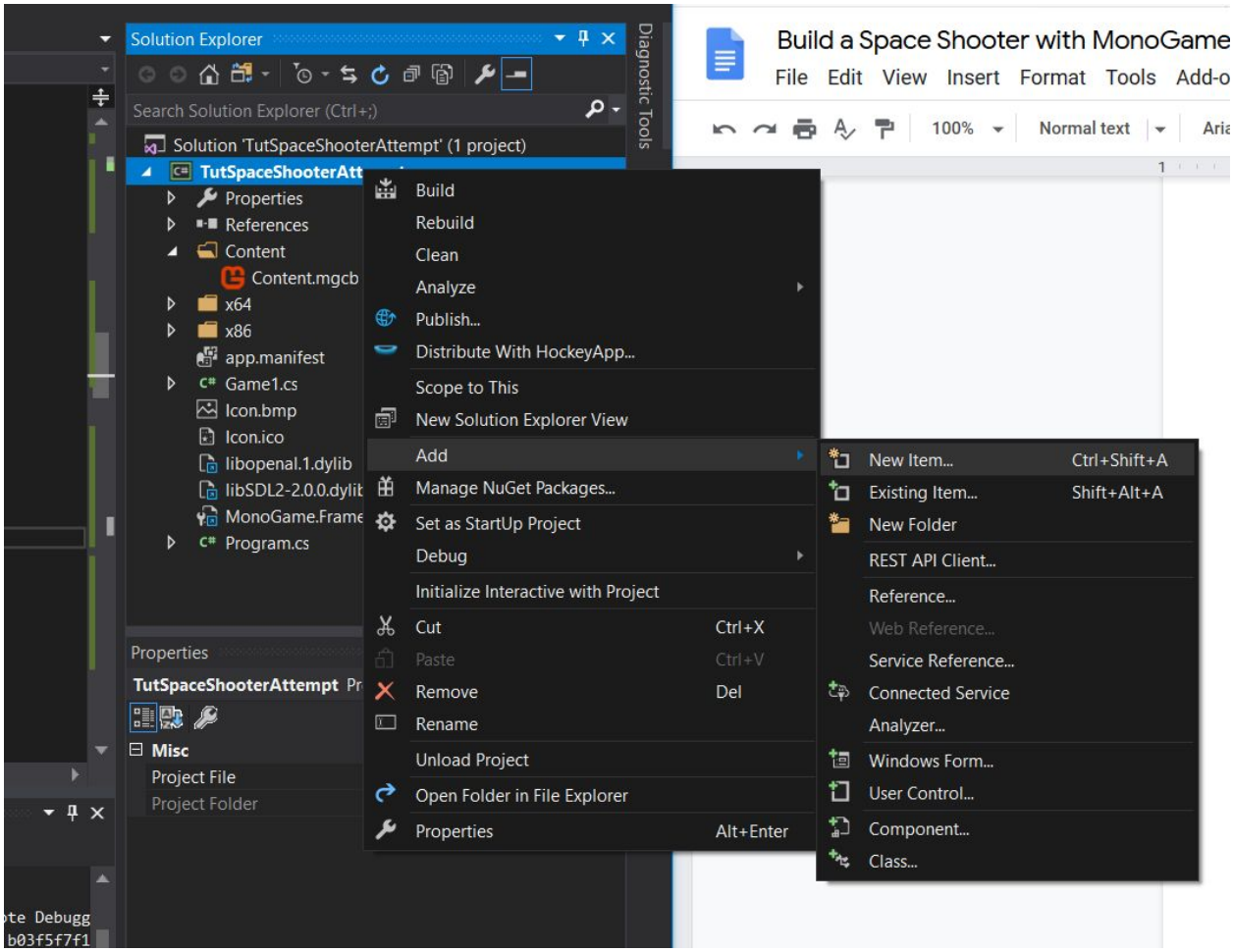
```
private void DrawMainMenu(SpriteBatch spriteBatch) {

}

private void DrawGameplay(SpriteBatch spriteBatch) {

}
```

```
private void DrawGameOver(SpriteBatch spriteBatch) {

}
```

Finally, we will need to add a few methods we will be using in our game class as well as others:

```
public static int randInt(int minNumber, int maxNumber)
{
	return new Random().Next(minNumber, maxNumber);
}

public static float randFloat(float minNumber, float maxNumber)
{
	return (float)new Random().NextDouble() * (maxNumber - minNumber) + minNumber;
}
```

The next thing we will do is define our other classes. We will start by creating a new class named *AnimatedSprite.cs*. We can add a new class by going to the Solution Explorer, right click our project directory, move the mouse over the add option, then click “New Item…”



A new prompt will appear allowing us to choose the file template we want, as well as name the file. By default, Visual Studio should have the class template selected, so we can just enter the name we want. Like I said before, we will be naming this class, *AnimatedSprite.cs*. Once you're done with that, click the add button, and our new class should be added to our project. Let's get started diving deep into these classes!

We will start by opening *AnimatedSprite.cs*. The following two using statements will be required in this class, so add them to the other using statements at the top of the file:

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
```

Next, we will need to add some fields and properties that all animated sprites will have in common. What are they? We will need to store the width and height of each frame, the duration of each frame, the amount of frames, the current frame, and the different fields for the timer. We will also have to keep track if an animation can repeat, and if that animation can't repeat, we need to know if the animation has finished playing.

```
private Texture2D texture;
public int frameWidth { get; set; }
public int frameHeight { get; set; }
public int duration { get; set; }
public Rectangle sourceRect { get; set; }
public int amountFrames { get; set; }
public int currentFrame { get; set; }
private int updateTick = 0;
private bool _repeats = true;
public void setCanRepeat(bool canRepeat)
{
        _repeats = canRepeat;
}
private bool _finished = false;
public bool isFinished()
{
        return _finished;
}
```

The next step is to add the constructor. We will be taking in a texture, the frame width, the frame height, and the duration of the animation. We can figure out the rest inside the class once it's instantiated. Add the following code to create our constructor:

```
public AnimatedSprite(Texture2D texture, int frameWidth, int frameHeight, int duration)
{
	this.texture = texture;
	this.frameWidth = frameWidth;
	this.frameHeight = frameHeight;
	this.duration = duration;
	amountFrames = this.texture.Width / this.frameWidth;
	sourceRect = new Rectangle(currentFrame * this.frameWidth, 0, this.frameWidth,
this.frameHeight);
}
```

Finally we will need to add an *Update* method to our *AnimatedSprite* class. The instructions we're trying to execute each frame looks kind of like the following:

```
IF updateTick < duration THEN
	Add 1 to updateTick
ELSE
	IF currentFrame < amountFrames - 1 THEN
		Add 1 to currentFrame
	ELSE
		IF _repeats THEN
			Set currentFrame to 0.
		ELSE
			Set finished to true

	Set the source rectangle to the update rectangle
	Set updateTick to 0
```

Let's add the following to write our *Update* method and above with real C#:

```
public void Update(GameTime gameTime)
{
	if (updateTick < duration)
	{
		updateTick++;
	}
	else
	{
		if (currentFrame < amountFrames - 1)
		{
			currentFrame++;
		}
```

```
                else
                {
                        if (_repeats)
                        {
                                currentFrame = 0;
                        }
                        else
                        {
                                _finished = true;
                        }
                }

                sourceRect = new Rectangle(currentFrame * this.frameWidth, 0, this.frameWidth,
        this.frameHeight);
                updateTick = 0;
        }
}
```

We are now finished writing the *AnimatedSprite* class! Next let's create a new class named *Entity.cs*. The player spaceship and any other enemies in the game with inherit the properties of this class. In our new *Entity* class, add a using statement for *Microsoft.Xna.Framework*. Every entity will store the following information: is rotatable, scale, position, source origin, destination origin, and physics body. Add the following to initialize these fields and properties:

```
protected bool isRotatable = false;
public Vector2 scale = new Vector2(1.5f, 1.5f);
public Vector2 position = new Vector2(0, 0);
protected Vector2 sourceOrigin = new Vector2(0, 0);
public Vector2 destOrigin = new Vector2(0, 0);
public PhysicsBody body { get; set; }
```

Let's add our constructor, then instantiate a physics body inside it. In other words, when another class inherits this *Entity* class, a physics body will automatically be created for that class. It will all make sense shortly since it can be a bit confusing. Add the following to add our constructor:

```
public Entity() {
        body = new PhysicsBody();
}
```

We will also want a way to set up the bounding box for our entities. Add the following method:

```
 public void setupBoundingBox(int width, int height)
{
        body.boundingBox = new Rectangle((int)(position.X - destOrigin.X), (int)(position.Y -
destOrigin.Y), (int)(width * scale.X), (int)(height * scale.Y));
}
```

Finally, we will need to add an *Update* method. This method will be called for every entity. Let's add the *Update* method:

```
public void Update(GameTime gameTime)
{
        if (body != null)
        {
        position.X += body.velocity.X;
        position.Y += body.velocity.Y;

        body.boundingBox = new Rectangle((int)position.X - (int)destOrigin.X, (int)position.Y -
(int)destOrigin.Y, body.boundingBox.Width, body.boundingBox.Height);

        }
        else
        {
        Console.WriteLine("[BaseEntity] body not found, skipping position updates.");
        }
}
```

Our *Entity* class is finished! The next step is to add a new class named *PhysicsBody.cs*. This class will contain no constructor or any other methods. We will just be adding two fields: *velocity*, and *boundingBox*. But first, we will need to add a using statement pointing to *Microsoft.Xna.Framework*. After that, add the following code to specify our two fields:

```
public Vector2 velocity = new Vector2(0, 0);
public Rectangle boundingBox;
```

Boom. We are done with the *PhysicsBody* class. Next, let's create another class called *Enemy.cs*. This class will be inheriting the *Entity* class. Then, any enemy spaceship class will inherit from this *Enemy* class. Let's add *using Microsoft.Xna.Framework;* and *using Microsoft.Xna.Framework.Graphics;* to our using statements. The *Enemy* class will hold an instance of a *Texture2D*, an *AnimatedSprite*, and a float containing the angle.

Now we can add our couple fields and property:

```
protected Texture2D texture;
protected AnimatedSprite sprite;
public float angle { get; set; }
```

Note, the next chunk of code isn't your average constructor. Write the following:

```
public Enemy(Texture2D texture, Vector2 position, Vector2 velocity) : base()
{
        this.texture = texture;
        sprite = new AnimatedSprite(this.texture, 16, 16, 10);
        scale = new Vector2(Game1.randFloat(10, 20) * 0.1f);
        sourceOrigin = new Vector2(sprite.frameWidth * 0.5f, sprite.frameHeight * 0.5f);
        destOrigin = new Vector2((sprite.frameWidth * 0.5f) * scale.X, (sprite.frameHeight * 0.5f)
* scale.Y);
        this.position = position;
        body.velocity = velocity;

        setupBoundingBox(sprite.frameWidth, sprite.frameHeight);
}
```

Perhaps you notice the ": base()" after the ending parenthesis. You would normally put in the requirements required by the *Entity* class, but there are none, so we'll leave it empty. We will also have to change the class declaration from:

```
class Enemy
```

To:

```
class Enemy : Entity
```

This is what tells the compiler that we want our *Enemy* class to extend *Entity*. After our constructor, let's add an *Update* method. Every tick, we will be setting the destination origin and update the animated sprite. Further, we will also be calling the *Update* method of the *Entity* class. We can accomplish that by utilizing *base.Update(gameTime)*. Let's add this method:

```
public new virtual void Update(GameTime gameTime)
{
        destOrigin = new Vector2(
        (float)Math.Round((sprite.frameWidth * 0.5f) * scale.X),
        (float)Math.Round((sprite.frameHeight * 0.5f) * scale.Y)
        );

        sprite.Update(gameTime);

        base.Update(gameTime);
}
```

We will also add a *Draw* method, which will be called if any enemy spaceship class extending this class does not have it's own *Draw* method:

```
public virtual void Draw(SpriteBatch spriteBatch)
{
        if (isRotatable)
        {
        Rectangle destRect = new Rectangle((int)position.X, (int)position.Y,
(int)(sprite.frameWidth * scale.X), (int)(sprite.frameHeight * scale.Y));
        spriteBatch.Draw(texture, destRect, sprite.sourceRect, Color.White,
MathHelper.ToRadians(angle), sourceOrigin, SpriteEffects.None, 0);

        }
        else
        {
        Rectangle destRect = new Rectangle((int)position.X, (int)position.Y,
(int)(sprite.frameWidth * scale.X), (int)(sprite.frameHeight * scale.Y));
        spriteBatch.Draw(texture, destRect, sprite.sourceRect, Color.White,
MathHelper.ToRadians(angle), sourceOrigin, SpriteEffects.None, 0);
        }
}
```

Next, we will add our three enemy classes. We will start with a new class named, *GunShip*. In this new class, let's make this class extend *Entity* by changing our class declaration to:

```
class GunShip : Enemy
```

Before we continue, let's not forget to add two using statements like so:

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
```

Now let's add three fields to store the shoot delay, the shoot tick, and whether or not the gun ship can shoot:

```
private int shootDelay = 60;
private int shootTick = 0;
public bool canShoot = false;
```

Then, let's add the constructor, as well as provide the arguments to instantiate the *Enemy* class we're inheriting:

```
public GunShip(Texture2D texture, Vector2 position, Vector2 velocity) : base(texture, position, velocity)
{

}
```

Every tick, we want to update our shoot timer and the animated sprite. To accomplish, let's add the following *Update* method:

```
public override void Update(GameTime gameTime)
{
	if (!canShoot)
	{
		if (shootTick < shootDelay)
		{
			shootTick++;
		}
		else
		{
			canShoot = true;
		}
	}

	sprite.Update(gameTime);

	base.Update(gameTime);
}
```

Finally, we will want to add a method to reset our shoot fields, once this gun ship has shot a laser. Let's add the following method and call it *resetCanShoot*:

```
public void resetCanShoot()
{
        canShoot = false;
        shootTick = 0;
}
```

Now that the *GunShip* class is defined, we can add a new class named *ChaserShip.cs*. For the chaser ship, we want to check if the player is within a specified distance, then have the chaser ship chaser after the player. By now, you should know what to do in order to make this class inherit *Enemy*. Add the following using statements before we forget:

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
```

To start creating the state system for our *ChaserShip* class, let's define a enum with two values:

```
public enum States
{
        MOVE_DOWN,
        CHASE
}
```

After that, let's set the current state by creating a *state* field:

```
private States state = States.MOVE_DOWN;
```

Let's add two methods to set and retrieve the state property:

```
public void SetState(States state)
{
        this.state = state;
        isRotatable = true;
}

public States GetState()
{
        return state;
}
```

Now we can add our constructor, we'll provide the arguments to the *Enemy* class as usual. We will also set the *angle* to *0*:

```
public ChaserShip(Texture2D texture, Vector2 position, Vector2 velocity) : base(texture,
position, velocity)
{
        angle = 0;
}
```

We will also want to add our *Update* method that will call the *Update* method of *Enemy*:

```
public override void Update(GameTime gameTime)
{
        base.Update(gameTime);
}
```

The last class we are adding that inherits *Enemy* is *CarrierShip*. Add a new class and name it *CarrierShip.cs*, add the two usual using statements, and make this new class inherit *Enemy*. The *CarrierShip* really doesn't do anything special actually. We will just be taking arguments in and passing them to the *Enemy* class. Add the following code for our constructor:

```
public CarrierShip(Texture2D texture, Vector2 position, Vector2 velocity) : base(texture,
position, velocity)
{

}
```

That's it for the *CarrierShip* class. While we're at it, let's add a new class for enemy lasers. Name this new class, *EnemyLaser.cs*. Add the usual using statements, and make the class extend *Entity*. We will need to add a field to store the texture:

```
private Texture2D texture;
```

Let's add our constructor:

```
public EnemyLaser(Texture2D texture, Vector2 position, Vector2 velocity) : base()
{
        this.texture = texture;
        this.position = position;
        body.velocity = velocity;

        setupBoundingBox(this.texture.Width, this.texture.Height);
}
```

As I mentioned previously, *Entity* doesn't accept any arguments, so we don't need to provide any arguments within the *base* keyword. We will also add an *Update* method, which will call the *Update* method of *Entity*:

```
public new void Update(GameTime gameTime)
{
        base.Update(gameTime);
}
```

We will conclude writing this class by adding a *Draw* method:

```
public void Draw(SpriteBatch spriteBatch)
{
        spriteBatch.Draw(texture, position, Color.White);
}
```

Let's create a new class for the player's laser and name it *PlayerLaser.cs*. Add the two using statements, then make the class inherit *Entity*. Add a field for storing the texture:

```
private Texture2D texture;
```

After that we can add the constructor:

```
public PlayerLaser(Texture2D texture, Vector2 position, Vector2 velocity) : base()
{
        this.texture = texture;
        this.position = position;
        body.velocity = velocity;

        setupBoundingBox(this.texture.Width, this.texture.Height);
}
```

Then we can add our *Update* and *Draw* methods respectively:

```
public new void Update(GameTime gameTime)
{
        base.Update(gameTime);
}

public void Draw(SpriteBatch spriteBatch)
{
        spriteBatch.Draw(texture, position, Color.White);
```

```
}
```

The next class we will add is the *Player* class. Add a new class and name it *Player.cs*. Add the using statements, and make the class extend *Entity*. This class will store a texture, an animated sprite, the movement speed, and a property storing whether or not the player is dead.

Add the following code to create these fields and properties:

```
public Texture2D texture;
public AnimatedSprite sprite;
public float moveSpeed = 4;
private bool _dead = false;
public bool isDead() { return _dead; }
public void setDead(bool isDead) { _dead = isDead; }
```

After that, let's create the constructor:

```
public Player(Texture2D texture, Vector2 position) : base()
{
        this.texture = texture;
        sprite = new AnimatedSprite(this.texture, 16, 16, 10);
        this.position = position;
        sourceOrigin = new Vector2(sprite.frameWidth * 0.5f, sprite.frameHeight * 0.5f);
        destOrigin = new Vector2((sprite.frameWidth * 0.5f) * scale.X, (sprite.frameHeight * 0.5f)
* scale.Y);
        setupBoundingBox(sprite.frameWidth, sprite.frameHeight);
}
```

Now that we added our constructor, we will need to define four methods we will use for movement:

```
public void MoveUp()
{
        body.velocity.Y = -moveSpeed;
}

public void MoveDown()
{
        body.velocity.Y = moveSpeed;
}

public void MoveLeft()
```

```
{
        body.velocity.X = -moveSpeed;
}

public void MoveRight()
{
        body.velocity.X = moveSpeed;
}
```

Next, we'll add the *Update* function which will update the animated sprite, and call the base *Update* function of *Entity*:

```
public void Update(GameTime gameTime)
{
        sprite.Update(gameTime);

        base.Update(gameTime);
}
```

Finally, we will finish the *Player* class by adding the *Draw* method:

```
public void Draw(SpriteBatch spriteBatch)
{
        if (!_dead)
        {
        Rectangle destRect = new Rectangle((int)position.X, (int)position.Y,
(int)(sprite.frameWidth * scale.X), (int)(sprite.frameHeight * scale.Y));
        spriteBatch.Draw(texture, destRect, sprite.sourceRect, Color.White);
        }
}
```

We only want to draw the player if the player is not dead. The player instance won't actually be destroyed, but we'll fake it by making it invisible when it's hit.

Next we will create a class for explosions. Name this new class, *Explosion.cs* and make it extend *Entity*. Also add the usual two using statements. This class will contain three fields: *texture*, *sprite*, and *origin*. Let's add them!

```
private Texture2D texture;
public AnimatedSprite sprite;
public Vector2 origin;
```

Then let's add the constructor:

```
public Explosion(Texture2D texture, Vector2 position) : base()
{
	this.texture = texture;
	sprite = new AnimatedSprite(this.texture, 32, 32, 5);
	sprite.setCanRepeat(false);
	this.position = position;
	origin = new Vector2((sprite.frameWidth * 0.5f) * scale.X, (sprite.frameHeight * 0.5f) *
scale.Y);
	setupBoundingBox(sprite.frameWidth, sprite.frameHeight);
}
```

We can also add the *Update* method:

```
public new void Update(GameTime gameTime)
{
	sprite.Update(gameTime);

	base.Update(gameTime);
}
```

Let's finally add the *Draw* method:

```
public void Draw(SpriteBatch spriteBatch)
{
	Rectangle destRect = new Rectangle((int)position.X - (int)origin.X, (int)position.Y -
(int)origin.Y, (int)(sprite.frameWidth * scale.X), (int)(sprite.frameHeight * scale.Y));
	spriteBatch.Draw(texture, destRect, sprite.sourceRect, Color.White);
}
```

The last two classes we'll add will be devoted to the scrolling background system. Create a new class, and name it *ScrollingBackground.cs*. This class **does not** need to inherit anything. It will still need the two usual using statements. We will want to add two fields containing background textures and layers:

```
private List<Texture2D> textures = new List<Texture2D>();
private List<ScrollingBackgroundLayer> layers = new List<ScrollingBackgroundLayer>();
```

The step is to add the constructor. We will be creating a vertical “stack” of three backgrounds, but we’ll be creating two more layers of backgrounds on top. This will allow us to have a buffer of backgrounds that will display as the layers before move off screen. This is how we can scroll the backgrounds. Let’s add the construcotr:

```
public ScrollingBackground(List<Texture2D> textures)
{
	this.textures = textures;

	for (int i = -1; i < 2; i++) // position indexes
	{
		for (int j = 0; j < 3; j++) { // 3 layers
			Texture2D texture = textures[Game1.randInt(0, textures.Count - 1)];
			Vector2 position = new Vector2(0, texture.Height * i);
			Vector2 velocity = new Vector2(0, (j + 1) * 0.2f);
			ScrollingBackgroundLayer layer = new ScrollingBackgroundLayer(this,
		texture, j, i, position, velocity);
			layers.Add(layer);
		}
	}
}
```

It’s time to add the more complicated part. Let’s first define our *Update* method:

```
public void Update(GameTime gameTime) {

}
```

In the *Update* method, the first thing we’ll want to do is sort each background by depth. We will then put the layer in the list of it’s corresponding depth. Add the following to start our *Update* method:

```
List<ScrollingBackgroundLayer> layersDepth0 = new List<ScrollingBackgroundLayer>();
List<ScrollingBackgroundLayer> layersDepth1 = new List<ScrollingBackgroundLayer>();
List<ScrollingBackgroundLayer> layersDepth2 = new List<ScrollingBackgroundLayer>();
List<ScrollingBackgroundLayer> layersToReset = new List<ScrollingBackgroundLayer>();

for (int i = 0; i < layers.Count; i++)
{
	layers[i].Update(gameTime);

	switch (layers[i].depth)
	{
```

```
                case 0:
                {
                        layersDepth0.Add(layers[i]);
                        break;
                }

                case 1:
                {
                        layersDepth1.Add(layers[i]);
                        break;
                }

                case 2:
                {
                        layersDepth2.Add(layers[i]);
                        break;
                }
        }
}
```

Next, we will have to iterate through each depth list and check if the first background is more than Y coordinate zero. We don't want to run out of backgrounds so we will reset the position of each layer in the corresponding depth. Let's add some more code to the *Update* function:

```
bool resetLayersDepth0 = false;
bool resetLayersDepth1 = false;
bool resetLayersDepth2 = false;

// Loop through layers in depth 0
for (int i = 0; i < layersDepth0.Count; i++)
{
        if (layersDepth0[i].positionIndex == -1)
        {
                if (layersDepth0[i].position.Y > 0)
                {
                        resetLayersDepth0 = true;
                }
        }
}

// Loop through layers in depth 1
for (int i = 0; i < layersDepth1.Count; i++)
{
```

```
        if (layersDepth1[i].positionIndex == -1)
        {
                if (layersDepth1[i].position.Y > 0)
                {
                resetLayersDepth1 = true;
                }
        }
}

// Loop through layers in depth 2
for (int i = 0; i < layersDepth2.Count; i++)
{
        if (layersDepth2[i].positionIndex == -1)
        {
                if (layersDepth2[i].position.Y > 0)
                {
                resetLayersDepth2 = true;
                }
        }
}

if (resetLayersDepth0)
{
        for (int i = 0; i < layersDepth0.Count; i++)
        {
                layersDepth0[i].position = layersDepth0[i].initialPosition;
        }
}

if (resetLayersDepth1)
{
        for (int i = 0; i < layersDepth1.Count; i++)
        {
                layersDepth1[i].position = layersDepth1[i].initialPosition;
        }
}

if (resetLayersDepth2)
{
        for (int i = 0; i < layersDepth2.Count; i++)
        {
                layersDepth2[i].position = layersDepth2[i].initialPosition;
        }
```

```
}
```

Last, we will add the *Draw* function, which will call the *Draw* method of each layer:

```
public void Draw(SpriteBatch spriteBatch)
{
        for (int i = 0; i < layers.Count; i++)
        {
                layers[i].Draw(spriteBatch);
        }
}
```

The last class we need to add will represent a scrolling background layer. Create a new class and call it *ScrollingBackgroundLayer.cs*. This class will need to extend *Entity*. Make sure to add the two usual using statements. We will also have to add several fields:

```
private ScrollingBackground scrollingBackground;
private Texture2D texture;
public Texture2D getTexture() { return texture; }
public int depth = 0;
public int positionIndex = 0;
public Vector2 initialPosition;
```

Let's add the following constructor:

```
public ScrollingBackgroundLayer(ScrollingBackground scrollingBackground, Texture2D texture,
int depth, int positionIndex, Vector2 position, Vector2 velocity) : base()
{
        this.scrollingBackground = scrollingBackground;
        this.texture = texture;
        this.depth = depth;
        this.positionIndex = positionIndex;
        this.position = position;
        initialPosition = this.position;
        body.velocity = velocity;
}
```

Then we will need to add the usual *Update* method:

```
public new void Update(GameTime gameTime)
{
        base.Update(gameTime);
}
```

Finally, we'll add the *Draw* method, which will draw the background layer when called:

```
public void Draw(SpriteBatch spriteBatch)
{
        spriteBatch.Draw(texture, position, Color.White);
}
```

Now that we're done with the classes for our game object's, let's create one more class for our menu buttons. Add a new class named *MenuButton.cs*. In that class, add the usual two using statements. This class does not need to extend anything. Next, add the following fields and properties:

```
private Game1 game;
private Vector2 position;
private Texture2D texDefault;
private Texture2D texOnDown;
private Texture2D texOnHover;
private Texture2D currentTexture;
public Rectangle boundingBox;

public bool isActive { get; set; }
public bool lastIsDown = false;
private bool _isDown = false;
private bool _isHovered = false;
```

Then we will add two methods for setting whether the button is pushed down or not, and whether the mouse is hovering over the button or not. In addition we will be adding a third method to update the texture of the button:

```
public void SetDown(bool isDown)
{
        if (!_isDown && isDown)
        {
                game.sndBtnDown.Play();
        }
        _isDown = isDown;
```

```
        ChangeTexture();
}
public void SetHovered(bool isHovered)
{
        if (!_isHovered && !_isDown && isHovered)
        {
                game.sndBtnOver.Play();
        }
        _isHovered = isHovered;

        ChangeTexture();
}

private void ChangeTexture()
{
        if (_isDown)
        {
                currentTexture = texOnDown;
        }
        else
        {
                if (_isHovered)
                {
                currentTexture = texOnHover;
                }
                else
                {
                currentTexture = texDefault;
                }
        }
}
```

After that, let's add our constructor:

```
public MenuButton(Game1 game, Vector2 position, Texture2D texDefault, Texture2D
texOnDown, Texture2D texOnHover)
{
        this.game = game;
        this.position = position;
        this.texDefault = texDefault;
        this.texOnDown = texOnDown;
        this.texOnHover = texOnHover;
        currentTexture = this.texDefault;
```

```
        boundingBox = new Rectangle((int)position.X, (int)position.Y, this.texDefault.Width,
this.texDefault.Height);
}
```

Finally, we can conclude *MenuButton.cs* with the *Draw* method:

```
public void Draw(SpriteBatch spriteBatch)
{
        if (isActive)
        {
                spriteBatch.Draw(currentTexture, position, Color.White);
        }
}
```

At this point, we can hop back over to *Game1.cs*. In *Game1.cs*, right under where we set the current game state, add the following:

```
private KeyboardState keyState = Keyboard.GetState();
```

The above line is used for the movement logic. After this line, we will want to define two menu buttons for the play button and the restart button:

```
private MenuButton playButton;
private MenuButton restartButton;
```

Then, we will add several lists for keeping track of explosions, enemies, lasers, and the like:

```
private List<Explosion> explosions = new List<Explosion>();
private List<Enemy> enemies = new List<Enemy>();
private List<EnemyLaser> enemyLasers = new List<EnemyLaser>();
private List<PlayerLaser> playerLasers = new List<PlayerLaser>();
private Player player = null;
private ScrollingBackground scrollingBackground;
```

Next, let's add two lines for the restart timer, which will be used when the player is destroyed:

```
private int restartDelay = 60 * 2;
private int restartTick = 0;
```

After that, we need to add two more lines for the enemy spawner timer:

```
private int spawnEnemyDelay = 60;
private int spawnEnemyTick = 0;
```

Then, let's write two more lines for the player shoot timer:

```
private int playerShootDelay = 15;
private int playerShootTick = 0;
```

In the Initialize method of *Game.cs*, we can set the mouse to be visible when you move it over the game window:

```
IsMouseVisible = true;
```

We can also set the width and height of the game window:

```
graphics.PreferredBackBufferWidth = 480;
graphics.PreferredBackBufferHeight = 640;
```

Finally, we have to apply the changes in order for these properties to take effect.

```
graphics.ApplyChanges();
```

Let's take another look at the *LoadContent* method. At the bottom after we load our SpriteFont, add the following few lines to instantiate our scrolling background, create our menu buttons, and change the game scene to the main menu:

```
scrollingBackground = new ScrollingBackground(texBgs);

playButton = new MenuButton(this, new Vector2(graphics.PreferredBackBufferWidth * 0.5f - (int)(texBtnPlay.Width * 0.5), graphics.PreferredBackBufferHeight * 0.5f), texBtnPlay, texBtnPlayDown, texBtnPlayHover);
restartButton = new MenuButton(this, new Vector2(graphics.PreferredBackBufferWidth * 0.5f - (int)(texBtnPlay.Width * 0.5), graphics.PreferredBackBufferHeight * 0.5f), texBtnRestart, texBtnRestartDown, texBtnRestartHover);

changeGameState(GameState.MainMenu);
```

Let's take another look at the *Update* method. Right after "TODO: Add your update logic here," but before the switch statement, add:

```
keyState = Keyboard.GetState();

scrollingBackground.Update(gameTime);
```

Next, let's fill in the *UpdateMainMenu* method. Pretty much, all we're doing in this method is to update the menu button state depending on the mouse state and position. If the mouse moves over the player button, it will play the hover sound and display the hover texture. If the mouse button is pressed while the mouse is over the play button, the button down sound will play and the corresponding texture will be show. Let's add the following to *UpdateMainMenu*:

```
if (playButton.isActive)
{
	MouseState mouseState = Mouse.GetState();

	if (playButton.boundingBox.Contains(mouseState.Position))
	{
		if (mouseState.LeftButton == ButtonState.Pressed)
		{
			playButton.SetDown(true);
			playButton.SetHovered(false);
		}
		else
		{
			playButton.SetDown(false);
			playButton.SetHovered(true);
		}

		if (mouseState.LeftButton == ButtonState.Released && playButton.lastIsDown)
		{
			changeGameState(GameState.Gameplay);
		}
	}
	else
	{
		playButton.SetDown(false);
		playButton.SetHovered(false);
	}
```

```
        playButton.lastIsDown = mouseState.LeftButton == ButtonState.Pressed ? true : false;
}
else
{
        playButton.isActive = true;
}
```

I'll give a brief rundown what we'll be doing in the *UpdateGameplay* method. If the player doesn't exist yet (when the game starts), we create an instance of it and assign it to the player field. At the same time we will be updating the restart timer. After that we update the player's movement (via the keyboard checks). Then we restrict the player position to the bounds of the screen. We also want to update all of the game object lists, as well as check for collisions. Let's start by the initial check testing whether the player is null or not. Add the following to the *UpdateGameplay* method:

```
if (player == null) {
        player = new Player(texPlayer, new Vector2(graphics.PreferredBackBufferWidth * 0.5f,
graphics.PreferredBackBufferHeight * 0.5f));
}
else {
        player.body.velocity = new Vector2(0, 0);

        if (player.isDead())
        {
                if (restartTick < restartDelay)
                {
                restartTick++;
                }
                else
                {
                changeGameState(GameState.GameOver);
                restartTick = 0;
                }
        }
        else
        {
                if (keyState.IsKeyDown(Keys.W))
                {
                player.MoveUp();
                }
```

```
            if (keyState.IsKeyDown(Keys.S))
            {
            player.MoveDown();
            }
            if (keyState.IsKeyDown(Keys.A))
            {
            player.MoveLeft();
            }
            if (keyState.IsKeyDown(Keys.D))
            {
            player.MoveRight();
            }
            if (keyState.IsKeyDown(Keys.Space))
            {
            if (playerShootTick < playerShootDelay)
            {
            playerShootTick++;
            }
            else
            {
            sndLaser.Play();
            PlayerLaser laser = new PlayerLaser(texPlayerLaser, new
        Vector2(player.position.X + player.destOrigin.X, player.position.Y), new Vector2(0, -10));
            playerLasers.Add(laser);
            playerShootTick = 0;
            }
            }
        }

        player.Update(gameTime);

        player.position.X = MathHelper.Clamp(player.position.X, 0,
        graphics.PreferredBackBufferWidth - player.body.boundingBox.Width);
        player.position.Y = MathHelper.Clamp(player.position.Y, 0,
        graphics.PreferredBackBufferHeight - player.body.boundingBox.Height);
}
```

After this check, we will be updating entity positions:

```
/**
* UPDATE ENTITY POSITIONS
**/
for (int i = 0; i < playerLasers.Count; i++)
```

```
{
        playerLasers[i].Update(gameTime);

        if (playerLasers[i].position.Y < 0)
        {
                playerLasers.Remove(playerLasers[i]);
                continue;
        }
}

for (int i = 0; i < enemyLasers.Count; i++)
{
        enemyLasers[i].Update(gameTime);

        if (player != null)
        {
                if (!player.isDead())
                {
                        if
                (player.body.boundingBox.Intersects(enemyLasers[i].body.boundingBox))
                        {
                                sndExplode[randInt(0, sndExplode.Count - 1)].Play();
                                Explosion explosion = new Explosion(texExplosion, new
                Vector2(player.position.X + player.destOrigin.X, player.position.Y +
                player.destOrigin.Y));
                                explosions.Add(explosion);

                                player.setDead(true);
                        }
                }
        }

        if (enemyLasers[i].position.Y > GraphicsDevice.Viewport.Height)
        {
                enemyLasers.Remove(enemyLasers[i]);
        }

}

for (int i = 0; i < enemies.Count; i++)
{
        enemies[i].Update(gameTime);
```

```
if (player != null)
{
	if (!player.isDead())
	{
		if (player.body.boundingBox.Intersects(enemies[i].body.boundingBox))
		{
			sndExplode[randInt(0, sndExplode.Count - 1)].Play();
			Explosion explosion = new Explosion(texExplosion, new
	Vector2(player.position.X + player.destOrigin.X, player.position.Y +
	player.destOrigin.Y));
			explosions.Add(explosion);

			player.setDead(true);
		}

		if (enemies[i].GetType() == typeof(GunShip))
		{
			GunShip enemy = (GunShip)enemies[i];

			if (enemy.canShoot)
			{
				EnemyLaser laser = new EnemyLaser(texEnemyLaser,
			new Vector2(enemy.position.X, enemy.position.Y), new Vector2(0, 5));
				enemyLasers.Add(laser);

				enemy.resetCanShoot();
			}
		}
		if (enemies[i].GetType() == typeof(ChaserShip))
		{
			ChaserShip enemy = (ChaserShip)enemies[i];

			if (Vector2.Distance(enemies[i].position, player.position +
	player.destOrigin) < 320)
			{
				enemy.SetState(ChaserShip.States.CHASE);
			}

			if (enemy.GetState() == ChaserShip.States.CHASE)
			{
				Vector2 direction = (player.position + player.destOrigin) -
			enemy.position;
				direction.Normalize();
```

```
                                        float speed = 3;
                                        enemy.body.velocity = direction * speed;

                                        if (enemy.position.X + (enemy.destOrigin.X) <
                                player.position.X + (player.destOrigin.X))
                                        {
                                                enemy.angle = enemy.angle - 5;
                                        }
                                        else
                                        {
                                                enemy.angle = enemy.angle + 5;
                                        }
                                }
                        }
                }
        }

        if (enemies[i].position.Y > GraphicsDevice.Viewport.Height)
        {
                enemies.Remove(enemies[i]);
        }
}

for (int i = 0; i < explosions.Count; i++)
{
        explosions[i].Update(gameTime);

        if (explosions[i].sprite.isFinished())
        {
                explosions.Remove(explosions[i]);
        }
}
```

We also want to add a collision check testing if each player laser has collided with an enemy:

```
for (int i = 0; i < playerLasers.Count; i++)
{
   bool shouldDestroyLaser = false;
   for (int j = 0; j < enemies.Count; j++)
   {
      if (playerLasers[i].body.boundingBox.Intersects(enemies[j].body.boundingBox))
```

```
        {
            sndExplode[randInt(0, sndExplode.Count - 1)].Play();

            Explosion explosion = new Explosion(texExplosion, new Vector2(enemies[j].position.X,
enemies[j].position.Y));
            explosion.scale = enemies[j].scale;

            Console.WriteLine("Shot enemy.  Origin: " + enemies[j].destOrigin + ", pos: " +
enemies[j].position);

            explosion.position.Y += enemies[j].body.boundingBox.Height * 0.5f;
            explosions.Add(explosion);

            enemies.Remove(enemies[j]);

            shouldDestroyLaser = true;
        }
    }

    if (shouldDestroyLaser)
    {
        playerLasers.Remove(playerLasers[i]);
    }
}
```

Finally, we want to add some logic for the enemy spawn timer. We will be selecting a random enemy to spawn, then we add the instance to the enemies list. Add the following code to conclude our *UpdateGameplay* method:

```
// Enemy spawning
if (spawnEnemyTick < spawnEnemyDelay)
{
        spawnEnemyTick++;
}
else
{
        Enemy enemy = null;

        if (randInt(0, 10) <= 3)
        {
                Vector2 spawnPos = new Vector2(randFloat(0,
        graphics.PreferredBackBufferWidth), -128);
```

```
            enemy = new GunShip(texEnemies[0], spawnPos, new Vector2(0, randFloat(1,
        3)));
        }
        else if (randInt(0, 10) >= 5)
        {
            Vector2 spawnPos = new Vector2(randFloat(0,
        graphics.PreferredBackBufferWidth), -128);
            enemy = new ChaserShip(texEnemies[1], spawnPos, new Vector2(0,
        randFloat(1, 3)));
        }
        else
        {
            Vector2 spawnPos = new Vector2(randFloat(0,
        graphics.PreferredBackBufferWidth), -128);
            enemy = new CarrierShip(texEnemies[2], spawnPos, new Vector2(0,
        randFloat(1, 3)));
        }

        enemies.Add(enemy);

        spawnEnemyTick = 0;
}
```

Now, we can move on to the *UpdateGameOver* method! This method will be very similar to the *UpdateMainMenu* method, but we'll be dealing with the restart button instead of the play button. In the *UpdateGameOver* method, add:

```
if (restartButton.isActive)
{
        MouseState mouseState = Mouse.GetState();

        if (restartButton.boundingBox.Contains(mouseState.Position))
        {
            if (mouseState.LeftButton == ButtonState.Pressed)
            {
                restartButton.SetDown(true);
                restartButton.SetHovered(false);
            }
            else
            {
                restartButton.SetDown(false);
                restartButton.SetHovered(true);
```

```
                }

                if (mouseState.LeftButton == ButtonState.Released && restartButton.lastIsDown)
                {
                        changeGameState(GameState.Gameplay);
                }
        }
        else
        {
                restartButton.SetDown(false);
                restartButton.SetHovered(false);
        }

        restartButton.lastIsDown = mouseState.LeftButton == ButtonState.Pressed ? true : false;
}
else
{
        restartButton.isActive = true;
}
```

Next, in the *resetGameplay* method, we will be setting the player to not be dead, then reset the player position. In the *resetGameplay* method, add the following:

```
if (player != null)
{
        player.setDead(false);
        player.position = new Vector2((int)(graphics.PreferredBackBufferWidth * 0.5),
(int)(graphics.PreferredBackBufferHeight * 0.5));
}
```

Then, in the *changeGameState* method, we want to clear all of the lists of game objects, call *resetGameplay*, then change the state. Add the following to *changeGameState*:

```
playButton.isActive = false;
restartButton.isActive = false;
explosions.Clear();
enemies.Clear();
playerLasers.Clear();
enemyLasers.Clear();
resetGameplay();
```

```
_gameState = gameState;
```

We have to make one quick addition to the *Draw* method, which is to add the draw call for the scrolling background. Between the *spriteBatch.Begin* call and the switch statement, let's add this line:

```
scrollingBackground.Draw(spriteBatch);
```

Now we can move on to the draw methods for our game states. Let's start with the main menu. In the *DrawMainMenu* method, add the following:

```
string title = "SPACE SHOOTER";
spriteBatch.DrawString(fontArial, title, new Vector2(graphics.PreferredBackBufferWidth * 0.5f -
(fontArial.MeasureString(title).X * 0.5f), graphics.PreferredBackBufferHeight * 0.2f),
Color.White);

playButton.Draw(spriteBatch);
```

After that, in the *DrawGameplay* method, add the following to draw each object in our lists of game objects:

```
for (int i = 0; i < enemies.Count; i++)
{
        enemies[i].Draw(spriteBatch);
}

for (int i = 0; i < playerLasers.Count; i++)
{
        playerLasers[i].Draw(spriteBatch);
}

for (int i = 0; i < enemyLasers.Count; i++)
{
        enemyLasers[i].Draw(spriteBatch);
}

for (int i = 0; i < explosions.Count; i++)
{
```

```
        explosions[i].Draw(spriteBatch);
}

if (player != null)
{
        player.Draw(spriteBatch);
}
```

Finally, in the *DrawGameOver* method, add the following to draw the elements on the game over game state:

```
string title = "GAME OVER";
spriteBatch.DrawString(fontArial, title, new Vector2(graphics.PreferredBackBufferWidth * 0.5f -
(fontArial.MeasureString(title).X * 0.5f), graphics.PreferredBackBufferHeight * 0.2f),
Color.White);

restartButton.Draw(spriteBatch);
```

And that concludes this course! If you have any questions, comments, or general feedback, I'd love to hear it. You can email me at jared.york@yorkcs.com, or tweet me at @jaredyork_.

If you found this course valuable, and would like to receive news about future tutorials and courses we release, please fill out the form.

You can find the full source code for this course on GitHub.