XNA Programming Lecture 5

Contents

Introduction 1
Design Discussion. 2
A new project 4
Loading Sprites. 5
Displaying the scene. 8
Onscreen Display 9
Adding the Ship Class. 13
Design Discussion. 13
Ship Methods. 16
Physics 101. 17
Drawing the ship. 20
Implementing ShipClass1 21
Input 22
Running the game. 23

Introduction

First off grab this zip file from Blackboard, or my website , make a new folder on your hard drive called XNA_Week5 and unpack it to the folder.

My website:

http://drewfx.com/TAFE/XNA/LunarLanderSprites.zip

Blackboard:

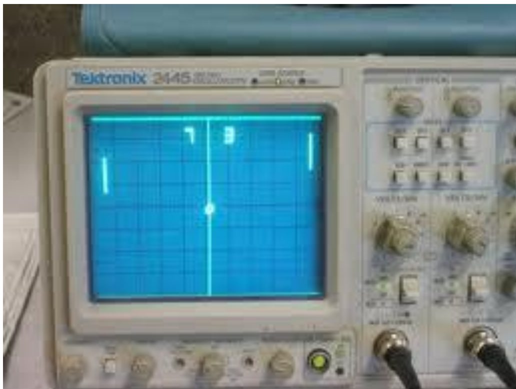
Apply introductory object-oriented language skills [D0053|ICAB4219B]

Look under Week 5 in Blackboard

Design Discussion.

Lunar Lander is a class game from way back video displays could be controlled and vector graphics could draw some rudimentary images on the screen. You'll probably seen an oscilloscope's output?

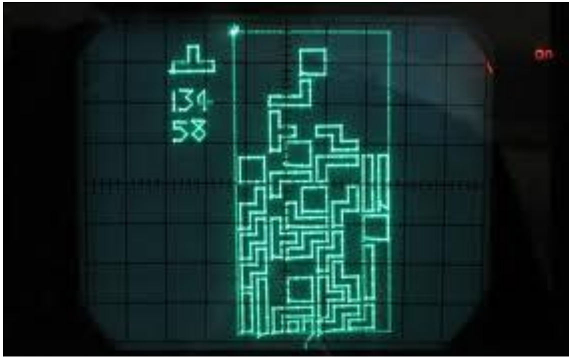
That's essentially how pong started for example.



Some dude figured out how to reorganise the display in a meaningful way and started the craze of video games.

Yes, they are freaks. But they are smart-ass freaks that made more money than we probably will.

Here's another few vectorised images that are games…



Recognise this one? Tetris.

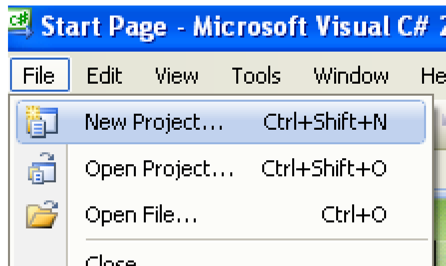
And finally we come to Lunar Lander...

The object of the game was to land on the pads and using a bit of fuel and gravity, attempt to land somewhere near them at a low enough speed not to impact the ground. We are going to recreate that game.

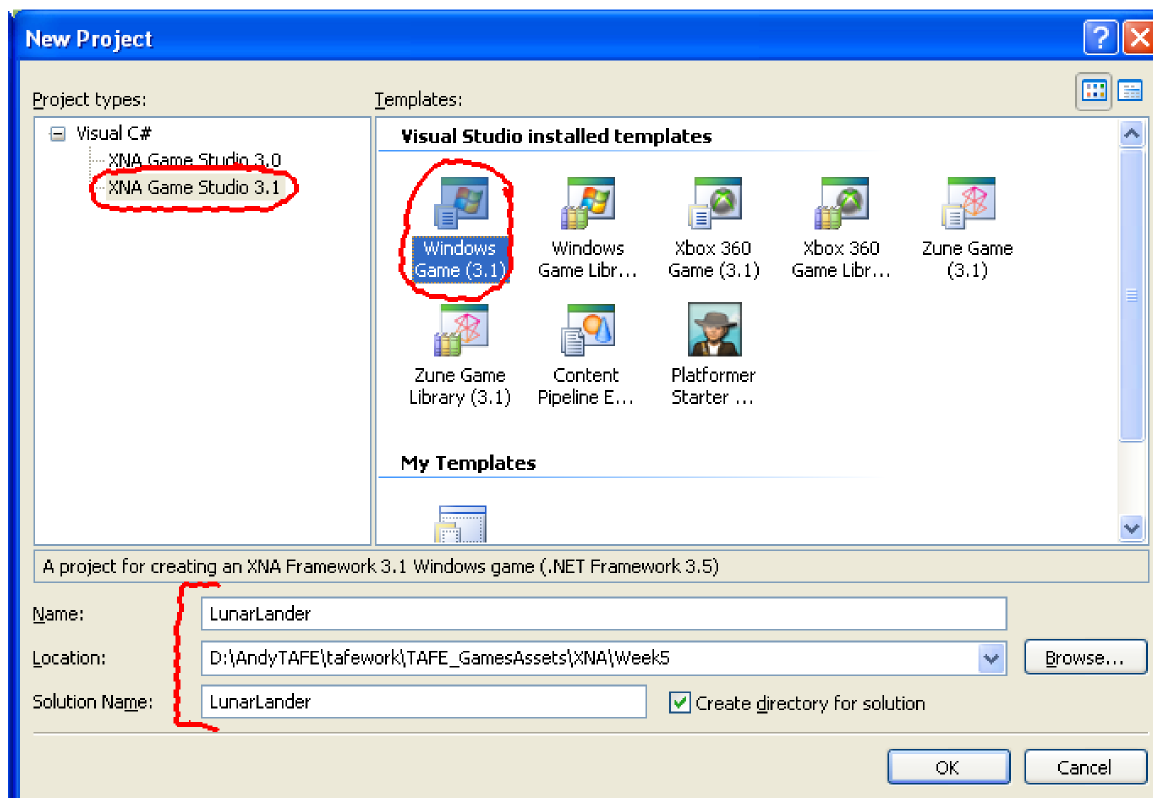
We need some graphics first so grab them from the link at the start (if you haven't already) and unpack them to a folder called XNA_Week5. You will import these graphics soon enough.

A new project

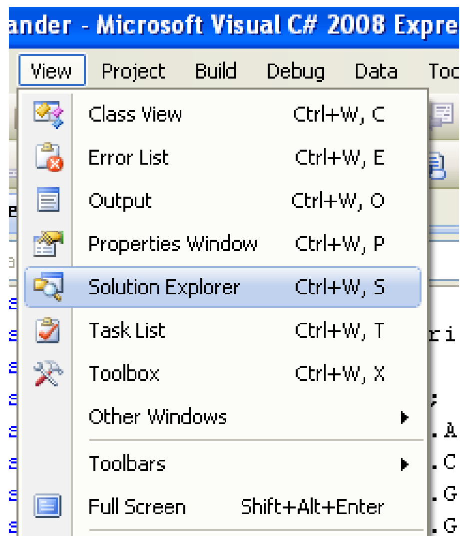
Let's start a brand new project. With Game Studio 3.1 open click on File->New Project.



Then select Project Type (on the left) as XNA Game Studio 3.1, select Windows Game 3.1 – this still uses XNA. Give it the name LunarLander, select a new folder on YOUR hard drive to output to and click on OK.

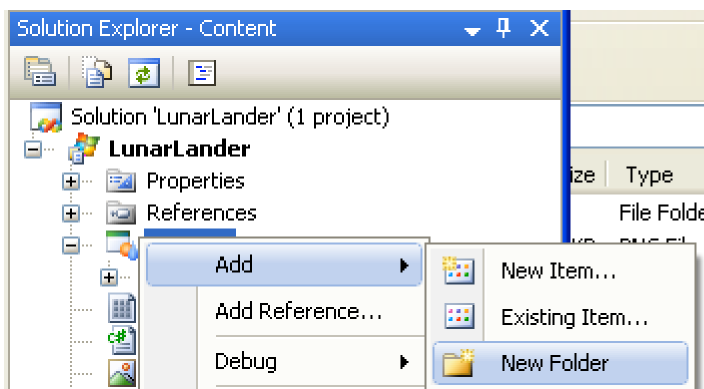


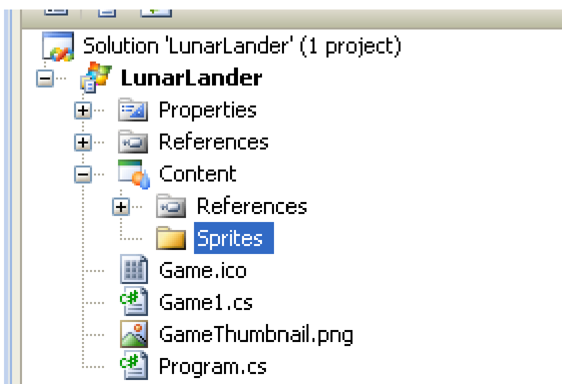
It may take a little while to create the project. Once it's loaded make sure you can see Solution Explorer on your right. If you can't, click on View->Solution Explorer...



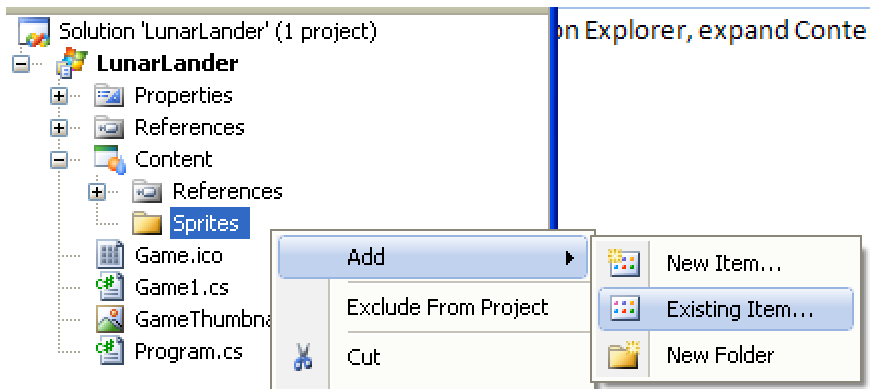
Loading Sprites.

Now the first you wanna add is the graphics. So in the Solution Explorer, expand Content and Add New Folder-> and call it Sprites...

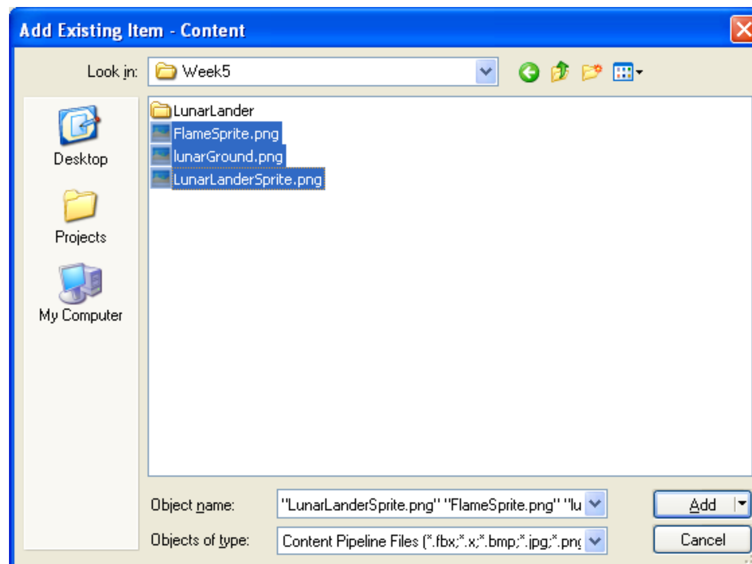




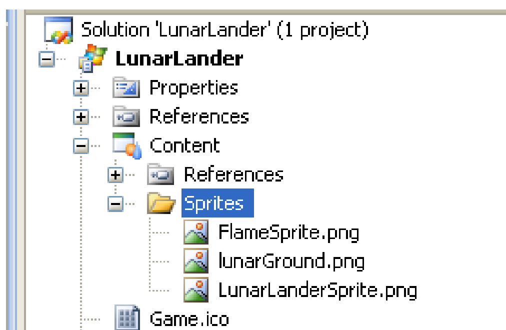
Now right click on Sprites and Add->Existing Item...



And browse to where you unpacked the sprites...

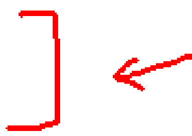


Now if you expand the Sprites folder under Content you will see them.

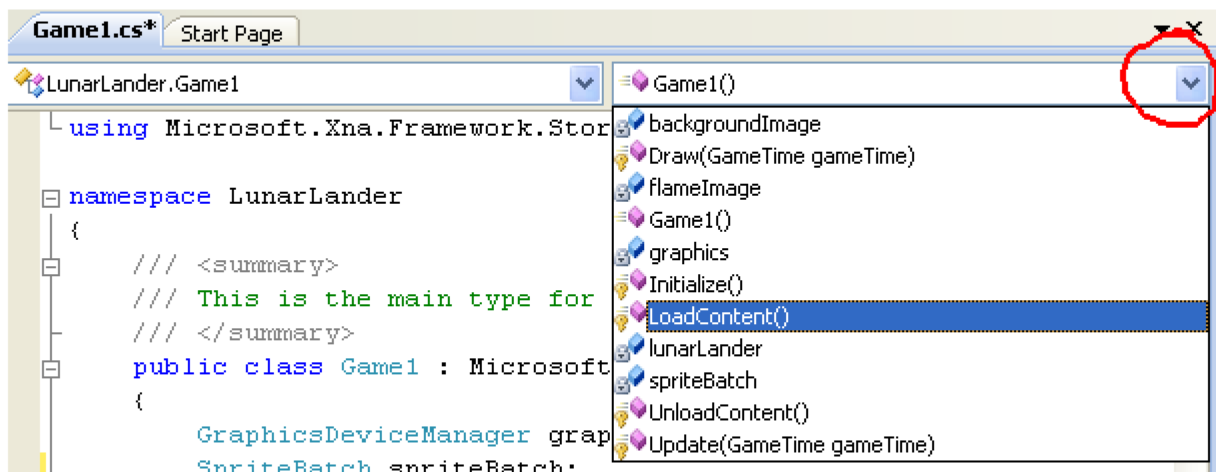


The name of the folder is what you use in the method LoadContent. So to proceed in a straight forward manner, let's add that code now. In the Solution Explorer double left click on Game1.cs. Scroll down until you find the declaration of SpriteBatch and add the following code...

```
namespace LunarLander
{
    /// <summary>
    /// This is the main type for your game
    /// </summary>
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        private Texture2D lunarLander;
        private Texture2D backgroundImage;
        private Texture2D flameImage;
```

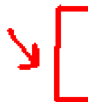


Then using the member browser (top right of the source code) select LoadContent...



In LoadContent you set up sprites and sounds required for the game. Let's load those sprites as follows...

```
protected override void LoadContent()
{
    // Create a new SpriteBatch, which can be used to draw textures.
    spriteBatch = new SpriteBatch(GraphicsDevice);

    backgroundImage = this.Content.Load<Texture2D>("Sprites\\lunarGround");
    lunarLander = this.Content.Load<Texture2D>("Sprites\\LunarLanderSprite");
    flameImage = this.Content.Load<Texture2D>("Sprites\\FlameSprite");
```



Notice that you don't need .png or .jpg. It works it out. You are only allowed jpg, pngs and tga's for Xbox anyway. Notice also that it has used the Sprites folder by specifying "Sprites\\" in the filename. If you change your Sprites folder you will need to update this as well.

Displaying the scene.

We'll need to know the screen metrics so at the top where you defined the sprites scroll up and define screen height and width here...

```
private Texture2D lunarLander;
private Texture2D backgroundImage;
private Texture2D flameImage;

private static int screenWidth;
private static int screenHeight;


public Game1()
{
```

We'll need to grab them the first time the screen is created so using the method browser find Initialize and type in the following code...

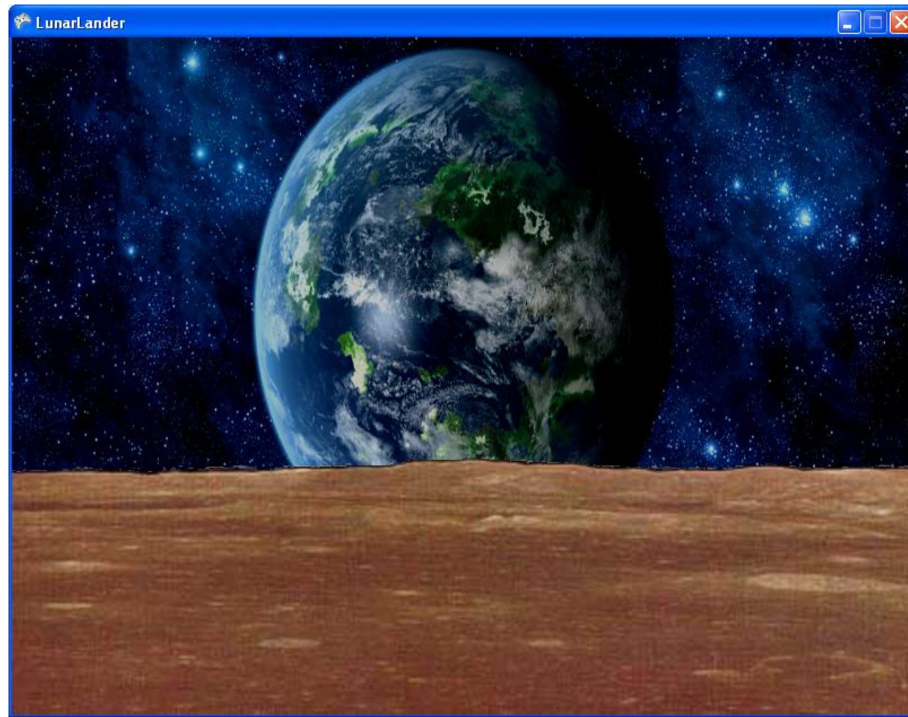
```
protected override void Initialize()
{
    screenHeight = graphics.GraphicsDevice.Viewport.Height;
    screenWidth = graphics.GraphicsDevice.Viewport.Width;

    base.Initialize();
}
```

Use the member browser and find the Draw method. Add the following code to draw the background scene.

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    // begin drawing sprites
    spriteBatch.Begin();

    // draw the starfield
    spriteBatch.Draw(backgroundImage,
                        new Rectangle(0, 0, screenWidth, screenHeight),
                        Color.White);

    // stop drawing sprites.
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Let's give a test now we have something to see. Hit F6 to compile and if you have any errors double click on the error report and see if you can figure out why it's not working. Just trace back through this doc and check your spelling.

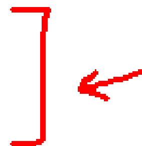
Next press CTRL + F5 to run a Release version of the game. You should see this – my own programmer art made from a bunch of Google'd images – pretty sweet hey? Shut down the game by hitting the X.



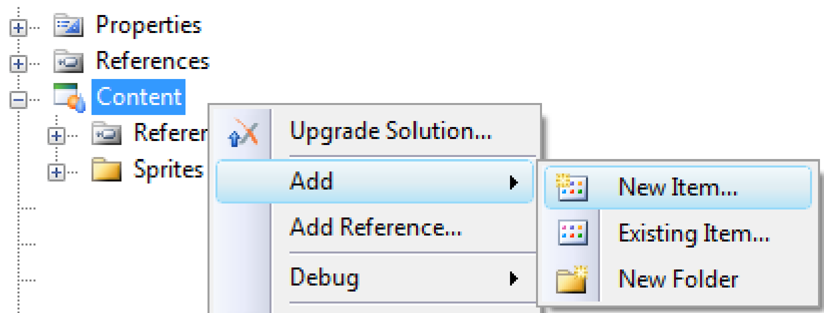
Onscreen Display

We'll need some text on the screen to display our velocity, fuel remaining and a status of whether we've crashed or not. Let's define some attributes (variables) first. Scroll to the top of Game1.cs and enter these definitions.

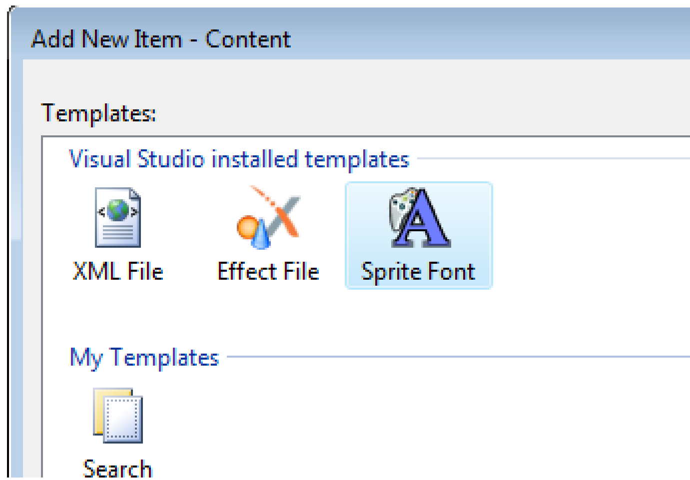
```
private static int screenWidth;
private static int screenHeight;

private float gravity = 9.8f;

SpriteFont Font1;
Vector2 FontPos;
```



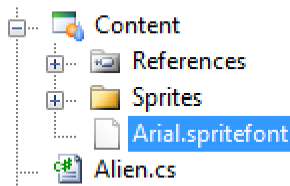
Next we need to set up the fonts. Firstly, go to the Content tree in Solution Explorer, the right click on Content, select Add->New Item...



Now select Sprite Font...



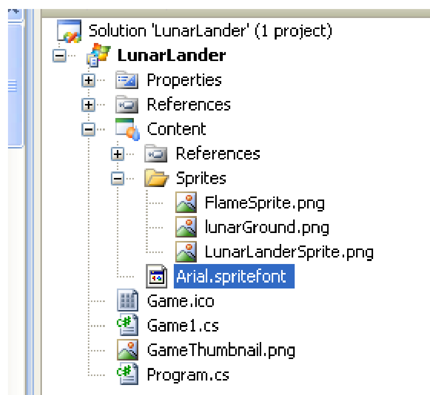
Find the item in the Content tree and rename it Arial...



MAKE SURE TO KEEP THE .SPRITEFONT suffix.

Now let's tweak the XML for the font to look like it should, Arial, Bold, Italic.

Find the font in the Content tree...



Double left click on it to bring up the XML. Now find the first entry called <FontName> and change it to this...

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

Scroll down and find <Style> and change it to Bold, Italic...

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

Next we need to load the font into the game so go to Game1.cs (from Solution Explorer) and select LoadContent by using the member browser and type in the following...

```
lunarLander = this.Content.Load<Texture2D>("Sprites\\LunarLanderSprite")
flameImage = this.Content.Load<Texture2D>("Sprites\\FlameSprite");

// set up the fonts.
Font1 = Content.Load<SpriteFont>("Arial");

FontPos = new Vector2(screenWidth / 2, screenHeight / 2);
```

We need a generalised print function (you can use this in the future for other situations) so using the member browser select the Draw method and just above it make this new method...

```
private void PrintString(string myString, Vector2 position)
{
    // Find the center of the string
    Vector2 FontOrigin = Font1.MeasureString(myString) / 2;
    // Draw the string
    spriteBatch.DrawString(Font1, myString, position, Color.LightGreen,
        0, FontOrigin, 1.0f, SpriteEffects.None, 0.5f);
}

/// <summary>
/// This is called when the game should draw itself.
/// </summary>
/// <param name="gameTime">Provides a snapshot of timing values.</param>
protected override void Draw(GameTime gameTime)
{
```

This method takes a string of characters and measures their width to determine where to place the string on screen taking into account the screen coordinates passed in using position as well. Then using a lengthy spriteBatch call it uses the Font object we've created to draw the text on screen. We make this a method because in future we don't want to stuff around writing this again everytime we want something printed on the screen. We just want to say PrintString("something", where).

Directly below that add a new method called DrawText which basically handles all the text we want to draw all in one go...

```
private void DrawText()
{
    // Draw Strings
    string velocity = "Velocity: " + Convert.ToString(myPlayer.Speed);
    PrintString(velocity, new Vector2(100, 50));
    string fuelRemaing = "Fuel: " + Convert.ToString(myPlayer.FuelRemains);
    PrintString(fuelRemaing, new Vector2(100, 150));

    if (myPlayer.bCrashed)
    {
        PrintString("Crashed!!!", new Vector2(screenWidth / 2, screenHeight / 2));
    }
}

/// <summary>
/// This is called when the game should draw itself.
/// </summary>
/// <param name="gameTime">Provides a snapshot of timing values.</param>
protected override void Draw(GameTime gameTime)
```

You'll notice it uses myPlayer attributes, such as myPlayer.Speed – which is from the ShipClass we haven't written yet. Just type them in for now and we'll add that soon. However let's have a look at what this method is doing to understand a few things. We cast a string called velocity and format a string to include "Velocity: " and then we concatenate the myPlayer.Speed to it. We have to convert Speed from a number to a string and that's what the Convert.ToString does. We must do it this way in order to create a string of characters otherwise it would generate an error when adding the two together.

Notice how we use new Vector2 and that a convenient way to make an x/y coordinate to place the text. We do the same things again with FuelRemains.

At the end we need to know if the player has crashed (which will write later) but it the Boolean bCrashed is true then it must update the onscreen status to say "Crashed!". We place that in the middle of the screen.

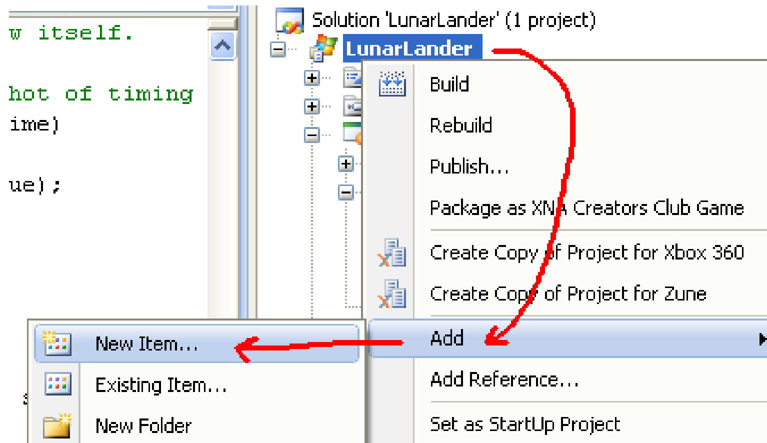
Finally we need to call the DrawText method during our applications Draw method, so use the member browser to find Draw and add this code...

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    // begin drawing sprites
    spriteBatch.Begin();

    // draw the starfield
    spriteBatch.Draw(backgroundImage,
                     new Rectangle(0, 0, screenWidth, screenHeight),
                     Color.White);

    // draw all text now
    DrawText();

    // stop drawing sprites.
```

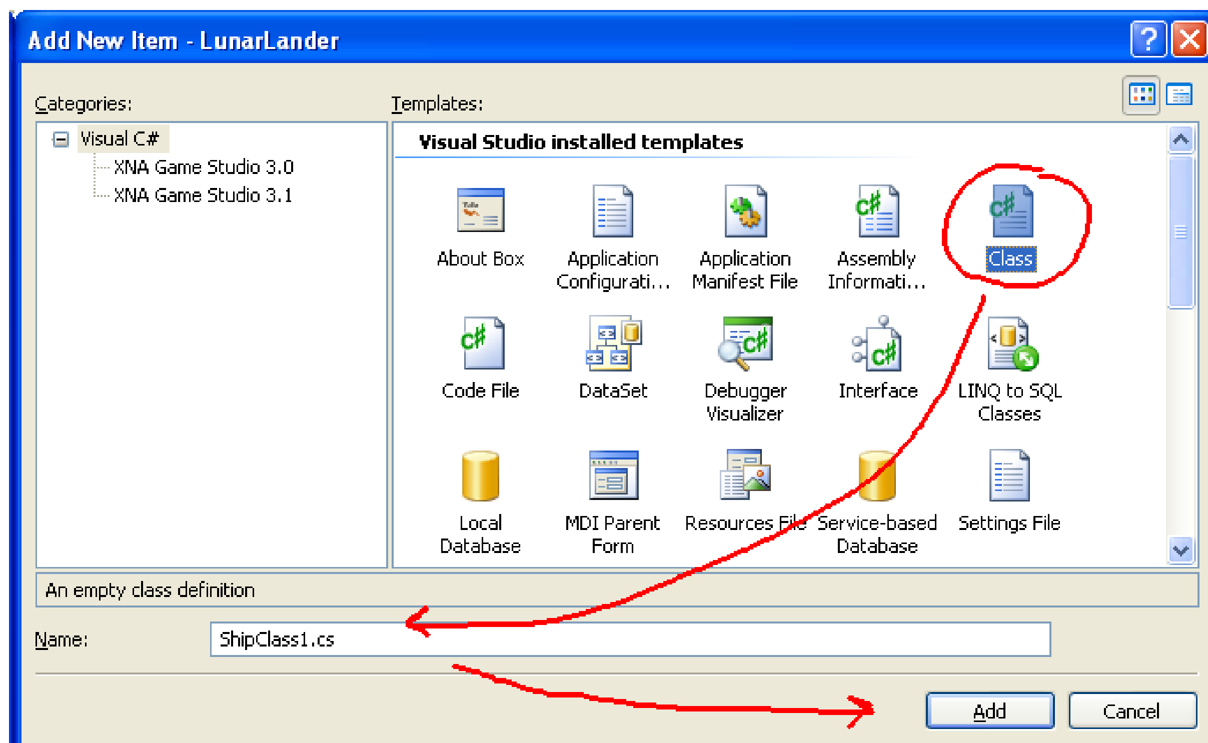
Don't bother running it yet, because we need to define the player class.

Adding the Ship Class.

We need to add a new class so right click on the project name Lunar Lander in the Solution Browser and click on Add->New Item...



In the dialog specify a class as the item and name it ShipClass1.cs...



Now double left click on the file ShipClass1.cs in the Solution Browser.

Design Discussion.

The ship needs a few things to work. Some sort of way to track it's propulsion, a sprite to draw it, some coordinates to place it, record the fuel remaining and some wave to move it.

We will of course need to access the Microsoft framework (that's what we're learnin' here) so add these two lines at the top...

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

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

namespace LunarLander
{
    class ShipClass1
    {
```

Let’s start by defining some attributes at the top. Type these attribute inside the {} braces...

```
namespace LunarLander
{
    class ShipClass1
    {
        private double ThrustSpeed = 20;
        private float sX, sY;
        public bool bCrashed = false;
    }
}
```

Now we need to track thrust, gravity and velocity and the way we do this is by tracking it over time. We’ll need a time keeper to work out home much time has passed since last time we updated it. Remember we want to protect data so we need to have a get and set method and this is how we write it. The upper case version of the attribute is the one we can use publicly which exposes the private lower case version. Make sure you put this inside the curly braces for the class...

```
        public bool bCrashed = false;

        // expose timePassed to external code
        public float TimePassed
        {
            get { return timePassed; }
            set { timePassed = value; }
        }
        private float timePassed;
    }
```

Next we want to access fuel for the drawing text section in Game1.cs that we spoke of before, so in ShipClass1.cs just below what we have typed, add this get / set method. I’ve added a comment there to remind you to keep all this code inside the class...

```
private float timePassed;

// expose fuel to external code
public int FuelRemains
{
    get { return fuelRemains; }
    set { fuelRemains = value; }
}
private int fuelRemains;

} // dont type code below this line
```

Let’s do two at a time now, so we need to expose Speed and Position to external code. Type this in below the code you just added...

```
private int fuelRemains;

// expose speed to external code
public float Speed
{
    get { return sX + sY; }
    set { }
}

public Vector2 Position
{
    get { return position; }
    set { position = value; }
}
private Vector2 position;

} // dont type code below this line
```

Notice that Speed is a little different and returns two summed attributes sX and sY which we had previously defined so there is no need to return speed. sX and sY are going to be used to move the spaceship along the x and y axis, and it’s this movement we determine as speed.

We’ll also need to expose Gravity and Velocity, so add these after Position...

```
private Vector2 position;

public float Gravity
{
    get { return gravity; }
    set { gravity = value; }
}
private float gravity;

public Vector2 Velocity
{
    get { return velocity; }
    set { velocity = value; }
}
private Vector2 velocity;

} // dont type code below this line
```

And finally we will expose the ScreenSize (so we can pass the screen size to this class) and a hook into the texture so we can send that in here too. We use the ScreenSize to know when we've hit the ground. The texture is just managed in Game1.cs and passed into the Ship class when loaded.

```
    private Vector2 velocity;

    // need this to know where we are on the screen
    public Vector2 ScreenSize
    {
        get { return screenSize; }
        set { screenSize = value; }
    }
    private Vector2 screenSize;

    public Texture2D Sprite
    {
        get { return shipSprite; }
        set { shipSprite = value; }
    }
    private Texture2D shipSprite;

} // dont type code below this line
```

All good. Now everything's defined, let's start adding some functionality to the ship class.

Ship Methods.

As with all classes we need to define the default constructor. In this case on creation we can set the fuel limit as well...

```
    private Texture2D shipSprite;

    // the default constructor
    public ShipClass1()
    {
        fuelRemains = 1000;
    }

} // dont type code below this line
```

Next we want to be able to move the ship. Type this in and I'll clarify what it does afterwards.

```
        fuelRemains = 1000;
    }

    // move the ship with a bit of math
    public void Move(GameTime gameTime)
    {
        TimeSpan ts = gameTime.ElapsedGameTime;
        timePassed = (float)ts.TotalSeconds;

        sY += gravity * (float)ts.TotalSeconds;

        if (position.Y > screenSize.Y - 100)
        {
            position.Y = screenSize.Y - 100;
            if (sX + sY > 35) bCrashed = true;
            sX = 0;
            sY = 0;

        }
        position.X += sX * (float)ts.TotalSeconds;
        position.Y += sY * (float)ts.TotalSeconds;
    }

} // dont type code below this line
```

GameTime is an object that handles units of time in various forms. We use it on the next couple of lines to determine how much time has passed since the last update cycle. Remember Updates and Draws rarely happen at the same time. Draw calls happen when the raster beam hits the bottom of the screen which is dependent on the GPU (Graphics Processing Unit) and the Update calls happen when the code has done a full cycle based on the CPU (Central Processing Unit).

What we are doing with simulations such as these where time is part of the calculation we can't allow the drawing to get in the way otherwise time can essentially slow down or speed up depending on the GPU clock speed.

Physics 101.

Anyway the calculation for ***displacement = gravity * time***...

$$\Delta x = v\,t$$

... derived from ...

$$\text{velocity} = \frac{s}{t}$$

... or ***velocity = displacement / time***.

Let’s have a look at what’s going on here so you can learn to grab physics calculation off the ‘net and use them in code. I scoured the ‘net looking for equations of motion and found this first one.

$$\Delta x = v\,t$$

The triangle symbol is universal in mathematics for DELTA. Delta refers to a shift in position, or displacement from an old position. In this case the equation I found was dealing with movement along the x axis. Next in the equation is ***= v t***

This means multiply ***v by t*** which can sometimes be represented as ***v.t***

If you want to convert this to computer code you would do this(DONT TYPE THIS IN – JUST READ), and this is what you will always be struggling to do over your career in graphics programming especially.

```
float displacement = 0;
float velocity = 0.5f;
float time = 0.1f;

// work out displacement
displacement = velocity * time; // 0.05

// work out velocity
velocity = displacement / time;

// work out time
time = displacment / velocity;
```

The last equation for time is an over simplification because you really need to know two positions for an object to work out the time take to get from one to the other, but the principal is the same.

As I’ve pointed out in our first exercise with Physics, there really are only 3 fundamental elements of measurement to deal with the displacement of objects, M(Mass), (A)Acceleration and (T)Time. Any part of displacement equations must be a derivative of these. So in our case gravity = acceleration. I’ll show the method again to refresh your memory.

```
        fuelRemains = 1000;
    }

    // move the ship with a bit of math
    public void Move(GameTime gameTime)
    {
        TimeSpan ts = gameTime.ElapsedGameTime;
        timePassed = (float)ts.TotalSeconds;

        sY += gravity * (float)ts.TotalSeconds;

        if (position.Y > screenSize.Y - 100)
        {
            position.Y = screenSize.Y - 100;
            if (sX + sY > 35) bCrashed = true;
            sX = 0;
            sY = 0;

        }
        position.X += sX * (float)ts.TotalSeconds;
        position.Y += sY * (float)ts.TotalSeconds;
    }

} // dont type code below this line
```

So now we know all that, the line...

```
sY += gravity * (float)ts.TotalSeconds)
```

... is doing this. It's displacing Y by the acceleration of gravity over the time passed since the last update. The `(float)` is type-casting `ts.TotalSeconds` to a float so that gravity and time become both floats and are compatible. You will see type-casting a lot in your coding career.

Moving on the next thing we do is test the position of the Lunar Lander with the height of the screen minus 100. This is the landing point. If this has been achieved we do a bit of math to work out the final speed of impact and if it's over 35 then set `bCrashed` to `true`, then reset the speeds (or velocities or displacements) or x and y.

At the end of the method we update the position of the lunar Lander with the final velocities along x and y scaled by the time since last update. Notice that all essential calculations that affect the movement of the Lander are affected by time. This creates a realistic simulation of motion for the user no matter how fast or slow their graphics card is.

Right! So how do you feel now? If you understood the last bit then you have a brilliant mind – well done. If you are like the rest of us and just got totally confused then welcome to the human race. We really aren't meant to understand this maths gibberish in normal day-to-day life, that's why we write this stuff down and work out calculations to handle the numbers for us.

So now the hard bit's over let's finish the code off.

The next bit we need to add is thrust over time. We know are all knowledge-empowered with our new physics brains so go ahead and see if you can figure out what's going on as you type. This goes just after the last function.

```
        position.Y += sY * (float)ts.TotalSeconds;
    }

    // calculate the thrust
    public void Thrust(double totalSeconds)
    {
        if (fuelRemains <= 0) return;
        //lander rotation
        //add thrust
        bCrashed = false;
        sX -= (float)Math.Sin(0) * (float)totalSeconds * (float)ThrustSpeed;
        sY -= (float)Math.Cos(0) * (float)totalSeconds * (float)ThrustSpeed;

        //decrease fuel
        fuelRemains--;
        if (fuelRemains < 0) fuelRemains = 0;
    }

} // dont type code below this line
```

If there is no fuel left then we can't thrust so first we check that and return out of the method if that is the case. However if there is fuel left we reset the check for crashed because we can thrust again, and work out the displacement of sX and sY (speed on x and y) using thrust speed (20 always) multiplied by time and a rotation. This week we are just dealing with up and down so leave the rotation at 0. We then reduce our fuel by 1 unit and make sure we can't have negative fuel.

Drawing the ship.

We just have the final update and draw methods to add after that previous method.

```
        if (fuelRemains < 0) fuelRemains = 0;
    }

    // update per clock cycle.
    public void Update(GameTime gameTime)
    {

        Move(gameTime);

    }
    // update per gpu cycle.
    public void Draw(SpriteBatch spriteBatch)
    {
        int x = (int)position.X;
        int y = (int)position.Y;
        spriteBatch.Draw(shipSprite, new Rectangle(x, y, 96, 96), Color.White);
    }

} // dont type code below this line
```

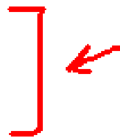
Using the new Rectangle command we can resize our ship sprite to whatever size we want. And we're done for the ShipClass.

Let's finish off Game1.cs to allow for ship class to be used.

Implementing ShipClass1

Open Game1.cs from the Solution Explorer and find the top of the code, then scroll down to where we defined the fonts and add this...

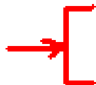
```
SpriteFont Font1;
Vector2 FontPos;

// player stuff
private static ShipClass1 myPlayer;
KeyboardState keyState;
bool bFlameIsOn = false;


public Game1()
```



This creates a handler to a potential instance of the ship class, a keystate handler for input and a Boolean to know when to draw the flame.

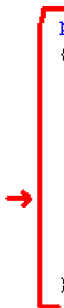
We want to initialise the font system now and the player, so using the member browser in Game1.cs find Initialize() and type in the following changes...

```
protected override void Initialize()
{
    screenHeight = graphics.GraphicsDevice.Viewport.Height;
    screenWidth = graphics.GraphicsDevice.Viewport.Width;

    base.Initialize();

    FontPos = new Vector2(100, 20);
    CreatePlayer();

}
```



Next we need to define the CreatePlayer method so scroll down after this method and add it...

```
    CreatePlayer();

}

private void CreatePlayer()
{
    myPlayer = new ShipClass1();
    myPlayer.Sprite = lunarLander;
    myPlayer.ScreenSize = new Vector2(screenWidth, screenHeight);
    myPlayer.Gravity = gravity;
    myPlayer.Position = new Vector2((screenWidth / 2) - (myPlayer.Sprite.Width / 2),
                                    screenHeight / 8);

}

/// <summary>
```



In this method we first instantiate (or create a new copy) of the ShipClass using the new keyword. This calls the default constructor in ShipClass1.cs which sets fuelRemaining to 100, as we know.

Next we assign the lunarLander sprite to this class using the Sprite set command we have already defined in ShipClass1.cs

We pass in the screen dimensions, gravity and place the lunar lander on the screen taking into account the width of the sprite.

Input

We now want to control the thrust by allowing the player to interact. Pressing the A button on the joystick or keyboard will make this happen. Using the member browser in Game1.cs find the Update method and type in the following...(remember we want to process maths at the speed of the cpu not the gpu so thrust control is done here)

```
protected override void Update(GameTime gameTime)
{
    // Allows the game to exit
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    keyState = Keyboard.GetState();

    // Allows the default game to exit on Xbox 360 and Windows
    if ((GamePad.GetState(PlayerIndex.One).Buttons.A == ButtonState.Pressed ||
        keyState.IsKeyDown(Keys.A)))
    {
        myPlayer.Thrust(gameTime.ElapsedGameTime.TotalSeconds);
        if (myPlayer.FuelRemains > 0) bFlameIsOn = true;
    }
    else
        bFlameIsOn = false;

    // now move the player
    myPlayer.Move(gameTime);

    base.Update(gameTime);
}
```

When you press A it calls the Thrust method and passes in the time since last update so thrust is calculated correctly over time. We can check if there is any fuel left by using the publicly accessible `FuelRemains` set method of the class and if it is we toggle the `bFlameOn` to true, otherwise it is false. Then finally we tell the ship class to move using the time since last update as well because we want it to work out the math independent of how fast it can draw the graphics.

Okay, one last thing remains... drawing it all...

Using the member browser in Game1.cs find the Draw method (not the DrawText method), and add the following changes...

```
/// <param name="gameTime">Provides a snapshot of timing values.</param>
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    // begin drawing sprites
    spriteBatch.Begin();

    // draw the starfield
    spriteBatch.Draw(backgroundImage,
                    new Rectangle(0, 0, screenWidth, screenHeight),
                    Color.White);

    // are we thrusting?
    if (bFlameIsOn)
    {
        spriteBatch.Draw(flameImage, new Rectangle((int)myPlayer.Position.X + 22,
                                                   (int)myPlayer.Position.Y + 64,
                                                   64, 64), Color.White);
    }

    // draw the player
    myPlayer.Draw(spriteBatch);

    // draw all text now
    DrawText();
```

We add two command to draw the flame if we are thrusting and then call the draw method on the player class with a pointer to the spriteBatch. Notice the bFlameIsOn drawing method rescales the sprite using the new Rectangle again. I don't really like this too much as art should always be scaled to suit and then imported, so we shouldn't need to scale. But this will do for this quick example.

Running the game.

Okay press F6 to compile, fix any errors (will be typos) and then run using CTRL+F5. You will see the lander drop. Hit A to add thrust and see the velocity change. If you hit the floor too hard you get the CRASH!! Status update comes up. If you run out of fuel you can no longer thrust. Have a go. Once you have finished, save this away and continue with the 2D video tutorials I have provided you.