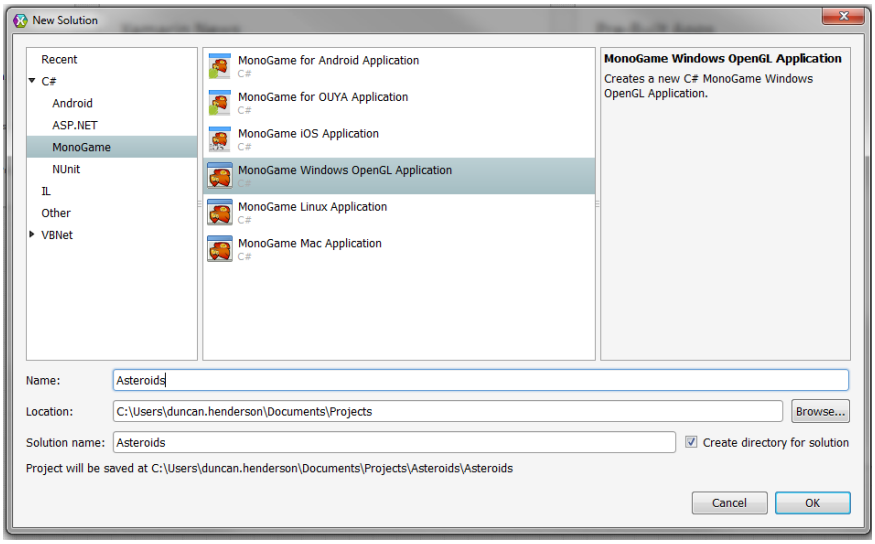
Asteroids

Our First Game, Awesome, let's do this.

Contents

Create a new Monogame project in Xamarin

1. File >New >Solution >C# > MonoGame Windows OpenGL Application
2. Enter your name for your project
3. Click ok.

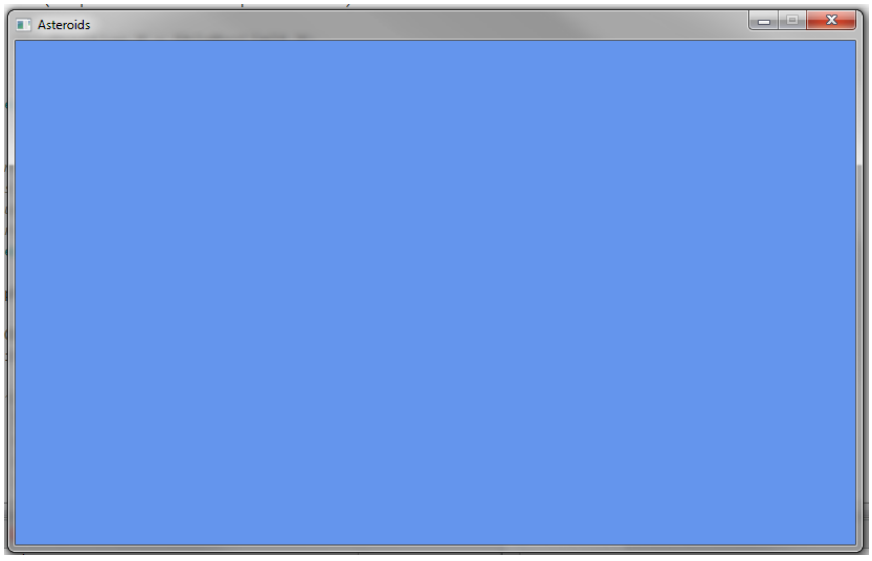


Have a look at your Solution Explorer window and note all the stuff that MonoGame adds to your project. MonoGame provides an excellent framework for us to build our game in, but it takes some getting used to.

If you haven't used MonoGame before, you're going to have to accept that you are not going to understand every line of code in your project.

The file that we will be coding in is Game1.cs. Have a look through this file and read all the comments that MonoGame has left for us. MonoGame does a lot of stuff in the background for us; some of the code in this file will be executed just once when the game launches, and others will be executed frequently, or as frequently as our computer can handle.

On line 23 of Game1.cs we want to change `graphics.IsFullScreen = true;` to `graphics.IsFullScreen = false;`



Click "Start Debugging" to run the project and you should get an empty blue window.

Draw the Player's Ship

The player's ship is just a texture/image/picture/sprite that we will draw to the screen. Later we will make this sprite move and shoot in response to keyboard input, collide with asteroids, etc. For now though we just want it to draw to screen.

Add a Texture to the Project

In order to draw a texture to the screen, we need to add it to our project.

1. Find or create a texture that we want to use for the player's ship.

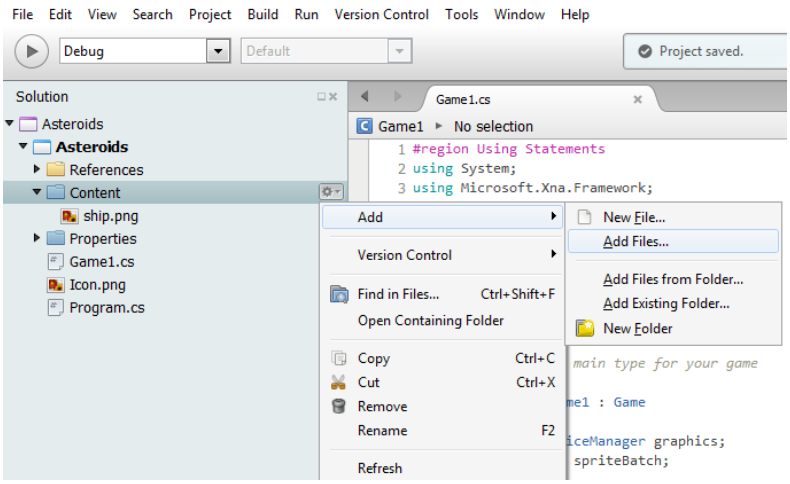
    a. The texture should be in a format that supports transparency, otherwise we'll have a square ship, which doesn't have efficient wind resistance in space. I recommend PNG files.

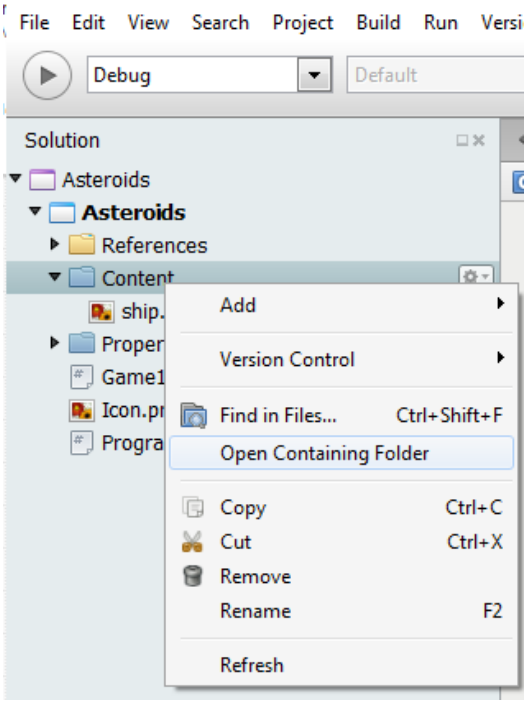
    b. You can download and use the sample textures from AIEportal.

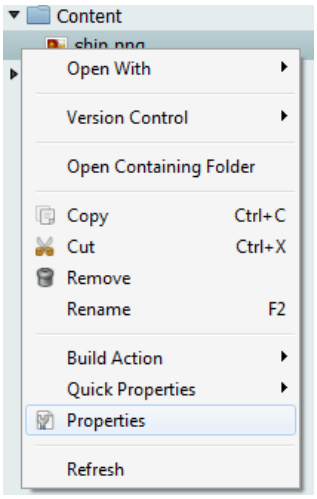
2. Open the containing folder of the content.

3. Drag the texture from its location on your disk to the Content Folder.

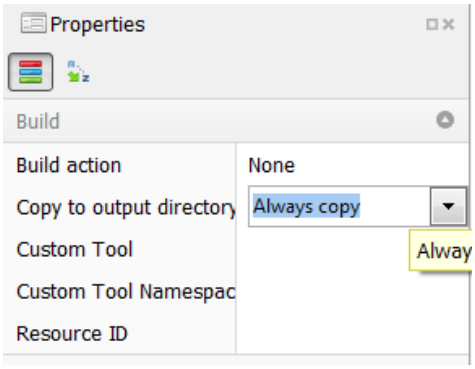
4. Add Files… into content to add your textures.







5. We also need to change the properties to always copy the picture into the build folder. Right click the newly added content and select properties. The properties window will show up on the right.



Declare, Initialise, Use

Now that we have added the texture to the project, we need to write the code to make it draw to screen. The way we do this is by using a Texture2D variable (Texture2D is a type given to us by MonoGame). We do this in 3 steps, which is a common sequence for all variables: Declare, Initialise, Use.

1. Declare the variable.

This is the place where we first mention the name and type of the new variable. After this, when the C# compiler encounters this variable name again, it will be all like "OK, I know what that name means. I know what data type it is, and where it is stored in memory".

A. We generally create new variables near the top of the file or class in which we are working. In this case, we will declare the variable right below the GraphicsDeviceManager and SpriteBatch variables which MonoGame has already declared for us. This code is executed once when the code is compiled and run.

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

        Texture2D ShipTexture;

        public Game1 ()
        {
            graphics = new GraphicsDeviceManager (this);
            Content.RootDirectory = "Content";
            graphics.IsFullScreen = true;
        }

        /// <summary>
        /// Allows the game to perform any initialization it needs to before starting to run.
        /// This is where it can query for any required services and load any non-graphic
        /// related content.  Calling base.Initialize will enumerate through any components
        /// and initialize them as well.
        /// </summary>
        protected override void Initialize ()
        {
            // TODO: Add your initialization logic here
            base.Initialize ();

        }

        /// <summary>
```

2. Initialise the variable.

All we are doing here is assigning a value to the variable for the first time.

A. A good place to do content loading in MonoGame is in the LoadContent function. That's probably why it's called LoadContent, 'cause it loads content. This function is called just once per game.

B. Note that the string "Ship" is the same as the name of the texture that we added to the project. You do not need to add the file name extension (.png), the Load function can find the texture without it.

```
/// <summary>
/// LoadContent will be called once per game and is the place to load
/// all of your content.
/// </summary>
protected override void LoadContent ()
{
    // Create a new SpriteBatch, which can be used to draw textures.
    spriteBatch = new SpriteBatch (GraphicsDevice);

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

    ShipTexture = Content.Load<Texture2D> ("ship");
}

/// <summary>
/// Allows the game to run logic such as updating the world,
/// checking for collisions, gathering input, and playing audio.
/// </summary>
/// <param name="gameTime">Provides a snapshot of timing values.</param>
protected override void Update (GameTime gameTime)
{
```

Texture2D ShipTexture

3. Use the variable.

We created and initialised the variable, so now it's ready to be used to draw it to the screen. But remember, games create the illusion of animation and movement by drawing static images to the screen constantly, and very quickly.

So, we need to draw the texture to the screen every time a new frame is drawn. This happens very fast, and multiple times per second, so we need to place our draw code somewhere where it will be called/invoked/executed repeatedly.

A. Well whaddya know, MonoGame has provided us with a Draw function which is automatically called as frequently as possible (up to 60 times per second by default).

Each time this function is called, the old frame is cleared from the screen, and the new one is drawn.

```
        /// <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)
        {
            graphics.GraphicsDevice.Clear (Color.CornflowerBlue);

            //TODO: Add your drawing code here
            spriteBatch.Begin ();

            spriteBatch.Draw (ShipTexture,
                new Vector2 (100, 100),
                null,
                Color.White,
                0,
                new Vector2 (ShipTexture.Width / 2, ShipTexture.Height / 2),
                new Vector2 (1, 1),
                SpriteEffects.None,
                 0);

            spriteBatch.End ();

            base.Draw (gameTime);
        }
    }
}
```

B. I have written the Draw call over multiple lines because there are so many parameters that it exceeds the width of the page. To see what each parameter does, click on the parameter, and press Ctrl+Shift+Space. This should bring up a window showing the type and name of each parameter, as well as any comments associated with that parameter:

```
spriteBatch.Draw (ShipTexture,
    new Vector2(100,100),
    null,
    Color.White,
    0,
    new Vector2                                    .Height / 2),
    new Vector2
    SpriteEffect
     0);

spriteBatch.End
```

```
▲ 5 of 7 ▼
public void Draw (
  Texture2D texture,
  Rectangle destinationRectangle,
  Rectangle? sourceRectangle,
  Color color,
  float rotation,
  Vector2 origin,
  SpriteEffects effect,
  float depth
)
```

C. Note that we have to calls spriteBatch.Begin() before spriteBatch.Draw, and spriteBatch.End() afterwards. Test out what happens if you don't do this.

Move the Ship

Let's fire this baby up and see what she can do!

We want the ship to respond to keyboard input in the following way:

- Right Arrow: Rotate ship clockwise.
- Left Arrow: Rotate ship anticlockwise.
- Up Arrow: Increase the velocity of the ship in the direction of its facing.
- Down Arrow: Reduce the velocity of the ship in the direction of its facing.

It would be easy to just moving the ship at a constant speed, and make it stop dead as soon as we release the movement key, but that's pretty boring behaviour for a spaceship, so shame on you for even considering that easy option.

Instead, we're going to simulate momentum, weight, and thrust by giving the ship a velocity. As Bert Newton once said, "Every body remains in a state of rest or uniform motion (constant velocity) unless it is acted upon by an external unbalanced force. This means that in the absence of a non-zero net force, the center of mass of a body either remains at rest, or moves at a constant speed in a straight line."

The ship's velocity will remain constant until we press the up or down arrow. When we do this, the velocity is increased or reduced.

Remember, a change in position over time is called velocity, and a change in velocity over time is called acceleration, so that's what we're going to call our variables.

The rotation of the ship involves some relatively complex math and code, but as every good programmer knows, you don't have to understand exactly how and why a piece of code works to use it.

Let's get cracking.

Declare the Variables we Need

Just as we did with the texture, we're going to declare, initialise, and use the variables used in the movement of the ship.

Declare the movement variables at the top of the class, just below the texture variable.

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

    Texture2D ShipTexture;
    Vector2 ShipPosition;
    Vector2 ShipVelocity; // the change in the ships position every update
    float ShipAcceleration;//the change in the ships velocity every update
                           //this should be +ve when the up arrow is pressed,
                           //and -ve when the down arrow is pressed.

    float ShipRotation; // ships rotation in radians
    float ShipRotationDelta;//the change in the ships rotation every update

    public Game1 ()
```

Initialise the Variables

Set the initial values for these variables, we only want to do this once at the start of the game. The comment above the Initialize function explains why this is a good place to put our initialisation code.

```
/// <summary>
/// Allows the game to perform any initialization it needs to before star
/// This is where it can query for any required services and load any non
/// related content.  Calling base.Initialize will enumerate through any
/// and initialize them as well.
/// </summary>
protected override void Initialize ()
{
    // TODO: Add your initialization logic here

    //position the ship in the middle of the screen
    ShipPosition = new Vector2 (graphics.PreferredBackBufferWidth / 2,
                                graphics.PreferredBackBufferHeight / 2);
    ShipVelocity = new Vector2 (0, 0);
    ShipAcceleration = 0;
    ShipRotation = 0;
    ShipRotationDelta = 0;

    base.Initialize ();

}

/// <summary>
/// LoadContent will be called once per game and is the place to load
/// all of your content.
/// </summary>
protected override void LoadContent ()
{
    // Create a new SpriteBatch, which can be used to draw textures.
    spriteBatch = new SpriteBatch (GraphicsDevice);
```

Respond to Keyboard Input

Key presses could happen at any time, so we need to continuously check for them while the game is running. Just as MonoGame continuously calls the Draw function while the game is running to handle everything to do with drawing the game to screen, it also continuously calls the Update function to handle updates to the game logic in the background.

```
protected override void Update (GameTime gameTime)
{
    // For Mobile devices, this logic will close the Game when the Back button is pressed
    if (GamePad.GetState (PlayerIndex.One).Buttons.Back == ButtonState.Pressed) {
        Exit ();
    }
    // TODO: Add your update logic here

    //reset values at the start of each update
    ShipAcceleration = 0;
    ShipRotationDelta = 0;

    if (Keyboard.GetState ().IsKeyDown (Keys.Up))
    {
        ShipAcceleration = -0.05f;
    }
    if (Keyboard.GetState ().IsKeyDown (Keys.Down))
    {
        ShipAcceleration = 0.05f;
    }
    if (Keyboard.GetState ().IsKeyDown (Keys.Left))
    {
        ShipRotationDelta = -0.05f;
    }
    if (Keyboard.GetState ().IsKeyDown (Keys.Right))
    {
        ShipRotationDelta = 0.05f;
    }

    ShipRotation += ShipRotationDelta;//calculate the change in rotation
    ////// Magic matrix math:
    /// create a matrix which describes the new rotation of the ship
    Matrix PLayerRotationMatrix = Matrix.CreateRotationZ (ShipRotation);
    //update the velocity using the new aceleration and rotation matrix
    ShipVelocity += Vector2.Transform (new Vector2 (0, ShipAcceleration), PLayerRotationMatrix);
    //update the position by the ships new velocity
    ShipPosition += ShipVelocity;


    base.Update (gameTime);
}
```

The matrix math used in the application of a rotation (in radians) to a velocity is beyond the scope of this course. It's important to know how to do this, but now why it works.

Update the Draw Function

All these calculations may be awesomely correct, but they won't do anything unless we use them to change where and how we draw the ship sprite.

Add the position and rotation values in to the Draw function for the ship texture.

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

    //TODO: Add your drawing code here
    spriteBatch.Begin ();

    spriteBatch.Draw (ShipTexture,
        ShipPosition, //<< add position here
        null,
        Color.White,
        ShipRotation, // <<< add rotation here
        new Vector2 (ShipTexture.Width / 2, ShipTexture.Height / 2),
        new Vector2 (1, 1),
        SpriteEffects.None,
         0);

    spriteBatch.End ();

    base.Draw (gameTime);
}
```

Run the game and check that you can move the ship with the arrow keys on the keyboard. If you're not completely clear on what all that code is doing, a good way to learn more about it is to make some changes to the code, and see how that affects the game.

Make the Ship Wrap Around the Screen

Have you noticed that you can lose the ship when it leaves the visible area of the screen space? In asteroids, when the ship reaches the end of the screen space at some limit, it should wrap or teleport to the opposite limit, making it impossible for the ship to ever leave the visible area of the screen.

For example, if the ship leaves the top of the screen, it should appear at the bottom, and similarly for the limits on the horizontal plane.

1. Create a size variable for the ship.

> The size of the ship is important in the wrap calculations, because we want to initiate the 'teleport' as soon as the ship is no longer visible on screen. Obviously, for larger ships, this will happen later than smaller ships.

The ship's position is the centre of the ship texture on screen.

> **A.** Declare the variable as a Vector2 with the other ship variables.
>
> **B.** Initialise the variable in the Initialize function. Set it equal to a new Vector2 of whatever size (in pixels) that you want.

```
spriteBatch.Draw (ShipTexture,
    ShipPosition,
    null,
    Color.White,
    ShipRotation,
    new Vector2 (ShipTexture.Width / 2, ShipTexture.Height / 2),
    new Vector2 (ShipSize.X / ShipTexture.Width, ShipSize.Y /  ShipTexture.Height),
    SpriteEffects.None,
     0);
```

> **c.** In the draw function, replace the scaling of the ship to use the new size variable:

2. Create two new Vector2 variables to store the min and max limits of the ship.

A. You know what to do to declare them.

B. Initialize them in the usual place

```
public class Game1 : Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    Texture2D ShipTexture;
    Vector2 ShipPosition;
    Vector2 ShipVelocity; // the change in the ships position every update
    float ShipAcceleration;//the change in the ships velocity every update
                           //this should be +ve when the up arrow is pressed,
                           //and -ve when the down arrow is pressed.

    float ShipRotation; // ships rotation in radians
    float ShipRotationDelta;//the change in the ships rotation every update

    Vector2 ShipSize;
    Vector2 ShipMaxLimit;
    Vector2 ShipMinLimit;

    public Game1 ()
    {
        graphics = new GraphicsDeviceManager (this);
        Content.RootDirectory = "Content";
        graphics.IsFullScreen = false;
    }
```

```
protected override void Initialize ()
{
    // TODO: Add your initialization logic here

    //position the ship in the middle of the screen
    ShipPosition = new Vector2 (graphics.PreferredBackBufferWidth / 2,
                                graphics.PreferredBackBufferHeight / 2);
    ShipVelocity = new Vector2 (0, 0);
    ShipAcceleration = 0;
    ShipRotation = 0;
    ShipRotationDelta = 0;

    ShipSize = new Vector2 (32.0f, 32.0f);
    ShipMaxLimit = new Vector2 (graphics.PreferredBackBufferWidth + (ShipSize.X / 2),
        graphics.PreferredBackBufferHeight + (ShipSize.Y / 2));
    ShipMinLimit = new Vector2 (0 - (ShipSize.X / 2), 0 - (ShipSize.Y / 2));

    base.Initialize ();

}
```

This is much easier to understand when drawn. If you are having trouble understanding how it works, ask your teacher to explain it.

C. In the update function, use these limits to constrain the ship's position.

```
    ShipRotation += ShipRotationDelta;//calculate t
    ////// Magic matrix math:
    /// create a matrix which describes the new rot
    Matrix PLayerRotationMatrix = Matrix.CreateRota
    //update the velocity using the new aceleration
    ShipVelocity += Vector2.Transform (new Vector2
    //update the position by the ships new velocity
    ShipPosition += ShipVelocity;

    //keep the ship within the screen limits.
    if (ShipPosition.X > ShipMaxLimit.X)
    {
        ShipPosition.X = ShipMinLimit.X;
    }
    else if(ShipPosition.X < ShipMinLimit.X)
    {
        ShipPosition.X = ShipMaxLimit.X;
    }

    if (ShipPosition.Y > ShipMaxLimit.Y)
    {
        ShipPosition.Y = ShipMinLimit.Y;
    }
    else if(ShipPosition.Y < ShipMinLimit.Y)
    {
        ShipPosition.Y = ShipMaxLimit.Y;
    }

    base.Update (gameTime);
}
```

Add an Asteroid

Some of you may have already suspected that this was coming, but our game is going to have... ASTEROIDS!

An asteroid is like our ship; you can see it on screen, and it moves around. The differences are that we can't control the movement of asteroids, and they don't wrap around the screen area like the ship.

Classes

The similarities between ships and asteroids mean that an asteroid is going to have many of the same properties as a ship. Texture, position, rotation, velocity, size, etc. But these names are already all being used by the ship, and we can't define two variables with the same name or the compiler won't know which one we are referring to in the code. You won't like the compiler when he's angry.

One solution is to give unique names to each variable. Instead of calling the asteroid's position variable "position", we can call it "asteroidPosition", and similarly for "asteroidVelocity", "asteroidRotation", etc.

This will work, but it requires longer variable names, and we have to do the same thing if we add bullets, enemies, missiles, explosions, etc. What I'm saying is that this solution is to good programming as Justin Bieber is to good music.

A better solution is to use structures or classes. Structures and classes allow us to group properties/variables together in a way that is distinct from other variables. Using structures or classes, our ship and asteroid can have variables with the same names without conflict.

It is not all that important to know the difference between structures and classes right now. For the purposes of this project, they will serve almost exactly the same purpose. I have chosen to use classes just because they work better with 'foreach' loops.

Let's start by...

Classifying our Ship Variables

```
GraphicsDeviceManager graphics;
SpriteBatch spriteBatch;

class ShipClass
{
    public Texture2D Texture;
    public Vector2 Position;
    public Vector2 Velocity; // the change in the ships position every update
    public float Acceleration;//the change in the ships velocity every update
                              //this should be +ve when the up arrow is pressed,
                              //and -ve when the down arrow is pressed.

    public float Rotation; // ships rotation in radians
    public float RotationDelta;//the change in the ships rotation every update

    public Vector2 Size;
    public Vector2 MaxLimit;
    public Vector2 MinLimit;
}

public Game1 ()
{
```

Note that we made each variable public. This allows us to access the variables from outside the class. It's kinda hard to explain, so just try and remove the word public later and see what happens.

Well that was pretty easy… what's that compiler? I now have 50 errors? That's cool, I know what I'm doing, I was trying to get 50 errors in the first place.

Now that we've put all these variables into a class we can't simply refer to them by name any more. First we have to create a ship object using the ship class, and then we have to reference each variable via that object.

Declare an Initialise a Ship Object

The ship class we defined above is just the blueprint for a ship object. All is does is tell the compiler what variables are associated with ships. Until we actually create a ship object there is no ship or ship variables stored in memory.

Declare a new ship object right below the ship class. This creates a block of memory assigned to this ship variable that is large enough to store a value for each of the variables in the ship class.

```
    public Vector2 Size;
    public Vector2 MaxLimit;
    public Vector2 MinLimit;
}

ShipClass Ship;

public Game1 ()
{
    graphics = new GraphicsDeviceManager (this);
```

Then initialise the object and all its properties / variables inside the initialise function:

```
protected override void Initialize ()
{
    // TODO: Add your initialization logic here


    Ship = new ShipClass();

    //position the ship in the middle of the screen
    Ship.Position = new Vector2 (graphics.PreferredBackBufferWidth / 2,
                                 graphics.PreferredBackBufferHeight / 2);
    Ship.Velocity = new Vector2 (0, 0);
    Ship.Acceleration = 0;
    Ship.Rotation = 0;
    Ship.RotationDelta = 0;

    Ship.Size = new Vector2 (32.0f, 32.0f);
    Ship.MaxLimit = new Vector2 (graphics.PreferredBackBufferWidth + (Ship.Size.X / 2),
        graphics.PreferredBackBufferHeight + (Ship.Size.Y / 2));
    Ship.MinLimit = new Vector2 (0 - (Ship.Size.X / 2), 0 - (Ship.Size.Y / 2));

    base.Initialize ();

}
```

All the remaining errors can be fixed by referencing the variables via the ship object.

Create an asteroid

Most of the asteroid code can be created by simply copying and modifying our ship code.

```
class AsteroidClass
{
    public Vector2 Position;
    public Vector2 Velocity;
    public float Rotation;
    public float RotationDelta;

    public Vector2 Size;

    public Vector2 MaxLimit;
    public Vector2 MinLimit;
}

public Texture2D AsteroidTexture;
AsteroidClass Asteroid;

Random RandNum;

public Game1 ()
```

Note that I have not included the asteroid texture in the class. Although we have multiple asteroids, the texture will be the same for each one so it would be a waste of memory space to load the same data more than once. A texture should only be loaded once in the load content function. And when we come to draw each asteroid we can draw the same texture multiple times at the position and orientation defined by each asteroid.

Note also the randNum variable. This will be used to give the asteroid a random position, size and velocity.

Initialize in the initialize function.

```
protected override void Initialize ()
{
    // TODO: Add your initialization logic here
    RandNum = new Random ();

    Asteroid = new AsteroidClass ();
    Asteroid.Position = new Vector2 (RandNum.Next(graphics.PreferredBackBufferWidth),
        RandNum.Next(graphics.PreferredBackBufferHeight));
    Asteroid.Velocity = new Vector2 (RandNum.Next(-3,3),RandNum.Next(-3,3));
    Asteroid.RotationDelta = RandNum.Next(-100,100);

    int RandSize = RandNum.Next(32,256);
    Asteroid.Size = new Vector2 (RandSize, RandSize);

    Asteroid.MaxLimit = new Vector2 (graphics.PreferredBackBufferWidth + (Asteroid.Size.X +100),
        graphics.PreferredBackBufferHeight + (Asteroid.Size.Y + 100));
    Asteroid.MinLimit = new Vector2 (0 - (Asteroid.Size.X - 100), 0 - (Asteroid.Size.Y - 100));

    Ship = new ShipClass();
```

In the update function simply update the asteroids position by its velocity.

In the draw function, draw the asteroid in the same way that you draw the ship. Note that you do not need to call the spritebatch Begin() and End() functions again if you draw the asteroid between the existing calls.

Run the game and we should see an asteroid flying through space.

Bouncy Asteroids

Once the asteroid leaves the screen, it will continue travelling forever, or until its position value exceeds the size limits of a Vector2 type in c#. If you've got a couple of trillion years to kill, that's cool, but in the meantime, let's try to develop a more immediate solution.

We could do the same thing we did for the ship and wrap it around some world limits (though I recommend making the limits for asteroids greater than the visible screen area). However, Another easy solution is to bounce them off some world limits. This is easy done by negating its velocity on the plane whose limit it has crossed.

For example, if the asteroid's horizontal velocity if negative, it will eventually cross the minimum limit on the X axis. When this happens, all we need to do is multiply the horizontal velocity by -1 and the asteroid will move in the other direction.

```
Asteroid.Rotation += Asteroid.RotationDelta;
Asteroid.Position += Asteroid.Velocity;

if (Asteroid.Position.X > Asteroid.MaxLimit.X)
{
    //Asteroid.Position.X = Ship.MaxLimit.X;
    Asteroid.Velocity.X *= -1;
}
else if(Asteroid.Position.X < Asteroid.MinLimit.X)
{
    //Asteroid.Position.X = Ship.MinLimit.X;
    Asteroid.Velocity.X *= -1;
}
```

You are welcome to wrap the asteroids around the world limits (like we do with the ship), or give it a new random position, or any other solution that you like.

Here's a challenge: try to figure out a way to give the asteroid a new random position that is outside the visible screen area, but within some other world bounds. Let me know if you work it out.

More asteroids

Hey, you know what's better than one asteroid. That's right TEN THOUSAND asteroids, or however many you choose to put in your game.

I recommend ten thousand.

Now you might be thinking " but I don't have the patience to declare, initialize update and draw ten thousand asteroids", and that's ok because we're not going to create ten thousand asteroid variables we're going to create a single list of asteroids.

1. Change your asteroid variable to a List of Asteroids, and add a constant variable to store the number of asteroids that we want to create. We also need to add a using system.collection.generic to get access to the List<>.

```
#region Using Statements
using System;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Storage;
using Microsoft.Xna.Framework.Input;
using System.Collections.Generic;

#endregion
namespace Asteroids
{
```

```
public Texture2D AsteroidTexture;
//AsteroidClass Asteroid;   REMOVE THIS LINE
List<AsteroidClass> MyAsteroids;
const int NUM_ASTEROIDS = 10000;

Random RandNum;

public Game1 ()
```

2. Initialise the asteroids using a loop and add each one to the asteroids list

```
protected override void Initialize ()
{
    // TODO: Add your initialization logic here
    RandNum = new Random ();

    MyAsteroids = new List<AsteroidClass> ();

    for (int i = 0; i < NUM_ASTEROIDS; ++i)
    {
        AsteroidClass Asteroid = new AsteroidClass ();
        Asteroid.Position = new Vector2 (RandNum.Next (graphics.PreferredBackBufferWidth),
            RandNum.Next (graphics.PreferredBackBufferHeight));
        Asteroid.Velocity = new Vector2 (RandNum.Next (-3, 3), RandNum.Next (-3, 3));
        Asteroid.RotationDelta = RandNum.Next (-100, 100);

        int RandSize = RandNum.Next (32, 256);
        Asteroid.Size = new Vector2 (RandSize, RandSize);

        Asteroid.MaxLimit = new Vector2 (graphics.PreferredBackBufferWidth + (Asteroid.Size.X + 100),
            graphics.PreferredBackBufferHeight + (Asteroid.Size.Y + 100));
        Asteroid.MinLimit = new Vector2 (0 - (Asteroid.Size.X - 100), 0 - (Asteroid.Size.Y - 100));

        MyAsteroids.Add (Asteroid);
    }
```

3. Load, Update, and Draw each asteroid using foreach loops

```
foreach(AsteroidClass Asteroid in MyAsteroids)
{
    spriteBatch.Draw (AsteroidTexture,
        Asteroid.Position,
        null,
        Color.White,
        Asteroid.Rotation,
        new Vector2 (AsteroidTexture.Width / 2, AsteroidTexture.Height / 2),
        new Vector2 (Asteroid.Size.X / AsteroidTexture.Width, Asteroid.Size.Y /  AsteroidTexture.Height),
        SpriteEffects.None,
        0);
}
```

Shooting

What's the point of building a spaceship and launching into the unknown if you're not going to shoot stuff? Am I right?

I'm right.

Again, you're going to notice a lot of similarities between our missiles and asteroids and ship. We want to have multiple missiles alive at once, so we're going to create a list of em, just like we did for the asteroids.

Missiles will be created at the ship's position (and with the ship's rotation) when the user presses the space bar. Missiles will travel in the direction of their facing until they hit either an asteroid, or their world limits, which will destroy the missile.

1. Create our missile class and list of missiles
   - Note the Timespan variables will be used to limit our rate of fire

```
class MissleClass
{
    public Vector2 Position;
    public Vector2 Velocity;
    public float Rotation;

    public Vector2 Size;

    public Vector2 MaxLimit;
    public Vector2 MinLimit;
}

Texture2D MissleTexture;
List<MissleClass> MyMissles;

//the time that we fired our last shot
TimeSpan LastShot = new TimeSpan (0, 0, 0, 0, 0);
//the minimum time between shots
TimeSpan ShotCoolDown = new TimeSpan (0, 0, 0, 0, 100);
```

2. Initialise the missile list in the Initialise function, but do not add any missiles to the list yet. We will create missiles and add them to the list when the spacebar is pressed.

```
    MyMissles = new List<MissleClass> ();
```

3. Load the MissleTexture into the LoadContect() Function.

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

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

    Ship.Texture = Content.Load<Texture2D> ("ship");
    AsteroidTexture = Content.Load<Texture2D> ("asteroid");
    MissleTexture = Content.Load<Texture2D> ("missle");

}
```

4. In the update function, check if the space key has been pressed. If it has, check if enough time has passed since our last shot before we can fire again. If enough time has passed, then fire another missile.

```
if (Keyboard.GetState ().IsKeyDown (Keys.Space))
{
    //get the time since the last shot
    TimeSpan timeSincelastShot = gameTime.TotalGameTime - LastShot;

    //if the time Since last Shot is longer than the cooldown time then shoot again
    if(timeSincelastShot > ShotCoolDown)
    {
        MissleClass Missle = new MissleClass ();

        Missle.Position = Ship.Position;

        Missle.Rotation = Ship.Rotation;

        Matrix MissleRotationMatrix = Matrix.CreateRotationZ (Missle.Rotation);
        Missle.Velocity = new Vector2 (0, -10);
        Missle.Velocity = Vector2.Transform (Missle.Velocity, MissleRotationMatrix);
        Missle.Velocity = Missle.Velocity + Ship.Velocity;

        Missle.Size = new Vector2 (16, 16);

        Missle.MaxLimit = new Vector2 (graphics.PreferredBackBufferWidth + 500,
            graphics.PreferredBackBufferHeight + 500);
        Missle.MinLimit = new Vector2 (-500, -500);

        MyMissles.Add (Missle);

        //store the time of this shot
        LastShot = gameTime.TotalGameTime;
    }
}
```

5. In the draw function, draw each missile in the missiles list.
6. Don't forget to add a loop in the update function to apply the Missiles velocity to its position.

The Great Functionarizing*

* According to the internet, "Functionarizing" is and never will be a word. I guess that proves that me/humans are still smarter than the internet.

Well I dunno about you guys but I'm pretty impressed with what we've gotten so far. We've written hundreds of lines of code, and the foundation of a cool little game.

However, you may have noticed that it's getting increasingly more difficult to navigate around our code. The update function, for example has expanded to over a page in length and its doing a lot of different stuff.

This is where functions come to the rescue. We've been working a lot with functions but until not we haven't created one on our own. A function should have a single purpose, which is described by its name, and a rule of thumb is that when a function starts to exceed about half a page in length, we should consider breaking the code up into different functions.

Our game code should be broken down into al least the following extra functions:

- InitialiseShip
- InitialiseAsteroids
- CheckInput
- CreateMissiles
- UpdateShip
- UpdateAsteroids
- UpdateMissles
- DrawShip
- DrawAsteroids
- DrawMissiles

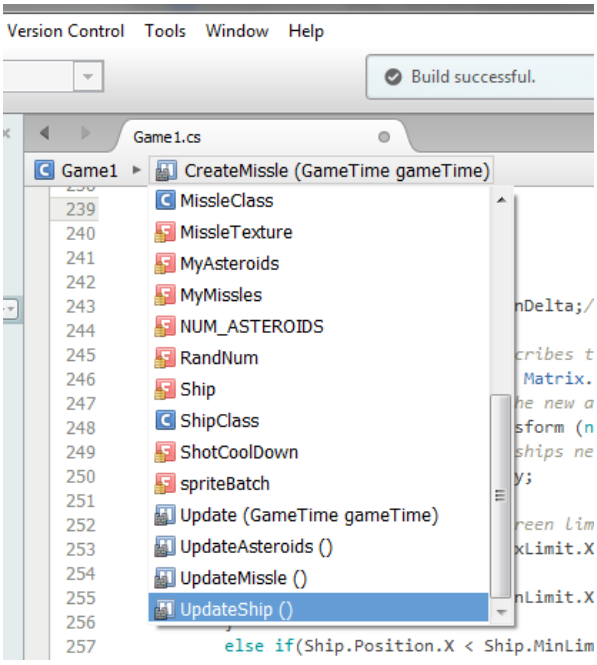
Some examples of these:

```
if (Keyboard.GetState ().IsKeyDown (Keys.Space))
{
    //get the time since the last shot
    TimeSpan timeSincelastShot = gameTime.TotalGameTime - LastShot;

    //if the time Since last Shot is longer than the cooldown time then shoot again
    if(timeSincelastShot > ShotCoolDown)
    {
        MissleClass Missle = new MissleClass ();

        Missle.Position = Ship.Position;

        Missle.Rotation = Ship.Rotation;

        Matrix MissleRotationMatrix = Matrix.CreateRotationZ (Missle.Rotation);
        Missle.Velocity = new Vector2 (0, -10);
        Missle.Velocity = Vector2.Transform (Missle.Velocity, MissleRotationMatrix);
        Missle.Velocity = Missle.Velocity + Ship.Velocity;

        Missle.Size = new Vector2 (16, 16);

        Missle.MaxLimit = new Vector2 (graphics.PreferredBackBufferWidth + 500,
            graphics.PreferredBackBufferHeight + 500);
        Missle.MinLimit = new Vector2 (-500, -500);

        MyMissles.Add (Missle);

        //store the time of this shot
        LastShot = gameTime.TotalGameTime;
    }
}
```

```
protected override void Initialize ()
{
    // TODO: Add your initialization logic here
    RandNum = new Random ();

    MyMissles = new List<MissleClass> ();

    InitializeASteroids ();
    InitializeShip ();

    base.Initialize ();
}

private void InitializeShip()
{
    Ship = new ShipClass();

    //position the ship in the middle of the screen
    Ship.Position = new Vector2 (graphics.PreferredBackBufferWidth / 2,
        graphics.PreferredBackBufferHeight / 2);
    Ship.Velocity = new Vector2 (0, 0);
    Ship.Acceleration = 0;
    Ship.Rotation = 0;
    Ship.RotationDelta = 0;

    Ship.Size = new Vector2 (32.0f, 32.0f);
    Ship.MaxLimit = new Vector2 (graphics.PreferredBackBufferWidth + (Ship.Size.X / 2),
        graphics.PreferredBackBufferHeight + (Ship.Size.Y / 2));
    Ship.MinLimit = new Vector2 (0 - (Ship.Size.X / 2), 0 - (Ship.Size.Y / 2));
}

protected override void Update (GameTime gameTime)
{
    CheckInput ();
    UpdateShip ();
    UpdateAsteroids ();
    UpdateMissle ();

    base.Update (gameTime);
}

protected void CheckInput(GameTime gameTime)
{
    // For Mobile devices, this logic will close the Game when the Back button is pressed
    if (GamePad.GetState (PlayerIndex.One).Buttons.Back == ButtonState.Pressed) {
        Exit ();
    }
    // TODO: Add your update logic here

    //reset values at the start of each update
    Ship.Acceleration = 0;
    Ship.RotationDelta = 0;

    if (Keyboard.GetState ().IsKeyDown (Keys.Up))
    {
        Ship.Acceleration = -0.05f;
    }
    if (Keyboard.GetState ().IsKeyDown (Keys.Down))
    {
        Ship.Acceleration = 0.05f;
    }
    if (Keyboard.GetState ().IsKeyDown (Keys.Left))
    {
        Ship.RotationDelta = -0.05f;
    }
    if (Keyboard.GetState ().IsKeyDown (Keys.Right))
    {
        Ship.RotationDelta = 0.05f;
    }

    if (Keyboard.GetState ().IsKeyDown (Keys.Space))
    {
        CreateMissle (gameTime);
    }
}
protected void CreateMissle(GameTime gameTime)
```

```
}
protected void CreateMissle(GameTime gameTime)
{
    //get the time since the last shot
    TimeSpan timeSincelastShot = gameTime.TotalGameTime - LastShot;

    //if the time Since last Shot is longer than the cooldown time then shoot again
    if (timeSincelastShot > ShotCoolDown) {
        MissleClass Missle = new MissleClass ();

        Missle.Position = Ship.Position;

        Missle.Rotation = Ship.Rotation;

        Matrix MissleRotationMatrix = Matrix.CreateRotationZ (Missle.Rotation);
        Missle.Velocity = new Vector2 (0, -10);
        Missle.Velocity = Vector2.Transform (Missle.Velocity, MissleRotationMatrix);
        Missle.Velocity = Missle.Velocity + Ship.Velocity;

        Missle.Size = new Vector2 (16, 16);

        Missle.MaxLimit = new Vector2 (graphics.PreferredBackBufferWidth + 500,
            graphics.PreferredBackBufferHeight + 500);
        Missle.MinLimit = new Vector2 (-500, -500);

        MyMissles.Add (Missle);

        //store the time of this shot
        LastShot = gameTime.TotalGameTime;
    }
}
protected void UpdateShip()
```

Notice how the dropdown list in the top left of Xamarin contains a list of all your functions and variables. You can use this to quickly jump between functions.



Collision

You know why God created asteroids, don't you? To smash into stuff obviously. I reckon it's our divine duty to add some collision to our game.

There are two types of collision in asteroids; player – asteroid collisions, and asteroid – missile collisions. We are going to do simple circle – circle type collisions for both.

Let's add a generic function that will return true if a collision has occurred between two objects, and false if no collision has occurred.

```
private bool CircleCollisionCheck(Vector2 Object1Pos,float Object1Radius,
    Vector2 Object2Pos,float Object2Radius)
{
    float DistanceBetweenObjects = (Object1Pos - Object2Pos).Length ();
    float SumOfRadii = Object1Radius + Object2Radius;

    if(DistanceBetweenObjects > SumOfRadii)
    {
        return true;
    }
    return false;
}
```

We can then use this function to check for collisions between the ship and asteroids, and between missiles and asteroids:

```
private void CheckCollisons()
{
    //remove these objects at the end of the check
    //we have to di the removal like this because
    //c# will not allow us to modify a list while we are
    //itterating over it. there is a good reason for this
    List<AsteroidClass> AsteroidDeathRow = new List<AsteroidClass> ();
    List<MissleClass> MissleDeathRow = new List<MissleClass> ();


    foreach(AsteroidClass Asteroid in MyAsteroids)
    {
        bool PlayerCollisionCheck = CircleCollisionCheck (Ship.Position, Ship.Size.X / 2,
            Asteroid.Position, Asteroid.Size.X / 2);
        if(PlayerCollisionCheck)
        {
            Ship.Die ();
            AsteroidDeathRow.Add (Asteroid);
        }

        foreach(MissleClass Missle in MyMissles)
        {
            bool MissleCollisionCheck = CircleCollisionCheck (Missle.Position, Missle.Size.X / 2,
                Asteroid.Position, Asteroid.Size.X / 2);

            if(MissleCollisionCheck)
            {
                MissleDeathRow.Add (Missle);
                AsteroidDeathRow.Add (Asteroid);
            }
        }
    }

    //removing a missle or asteroid from its
    //associated list will result in it no
    // longer updating , drawing and eventually
    //from existing (as the c# garbage collector will
    // come around adn remove it).
    foreach(AsteroidClass Asteroid in AsteroidDeathRow)
    {
        MyAsteroids.Remove (Asteroid);
    }
    foreach(MissleClass Missle in MissleDeathRow)
    {
        MyMissles.Remove (Missle);
    }
}
```

When the ship dies make it invisible, and prevent it from colliding with the asteroids for a short time. Respawn it and give it invulnerability for a short duration after afterwards.

PLAYER LIVES

Now that we're able to kill the player , let's give him some lives, a nice way to visually indicate to the player how many lives they has left is to draw one spaceship per life in one of the corners of the game screen.

```
private void DrawLives()
{
    for(int i = 0; i < PlayerLives; ++i)
    {
        spriteBatch.Draw (Ship.Texture,
            new Vector2(Ship.Size.X * (i + 1),Ship.Size.Y),
            null,
            Color.White,
            0,
            new Vector2 (Ship.Texture.Width / 2, Ship.Texture.Height / 2),
            new Vector2 (Ship.Size.X / Ship.Texture.Width,
                Ship.Size.Y /  Ship.Texture.Height),
            SpriteEffects.None,
            0);
    }
}
```

Score

Keeping track of the player's score is pretty straight forward. Just create a new score variable in the ship class, and increase it by whatever each time an asteroid is destroyed.

Display the score to the screen is done like so:

1. Add a new spritefont to your project's content directory the same way you add .png's. (there is one on AIEportal for you)

2. Create a spritefont variable
```
SpriteFont ScoreText;
```

3. and initialise it in the load content function
```
Ship.Texture = Content.Load<Texture2D> ("ship");
AsteroidTexture = Content.Load<Texture2D> ("asteroid")
MissleTexture = Content.Load<Texture2D> ("missle");
ScoreText = Content.Load<SpriteFont > ("scoreFont");
```

4. Draw the test to the screen
```
private void DrawScore()
{
    spriteBatch.DrawString (ScoreText, "SCORE : ", new Vector2(10,10), Color.White);
    spriteBatch.DrawString (ScoreText, Ship.Score.ToString(), new Vector2(120,10), Color.White);
}
```

Explosions

So far all of our sprites have been static textures. They can move around the screen but not animate themselves.

To do animation we have to use sprite sheets. These are bit textures containing a bunch of different images. We only draw one part of the texture (called a frame) for any given draw call, but if each frame is similar to the previous one and we switch them fast enough, the animation should look seamless.

Unfortunately, we have to handle the drawing, timing, and updating of each frame manually, so here's how it's done:

1. Define an explosion call, a list of explosions, an enumeration to differentiate two different types of explosions, and two new textures to use for the explosions, one texture will be used for asteroid explosion and the other for ship explosions.

```
public enum ExplosionType
{
    ASTEROID,
    SHIP,
}
class ExplosionsClass
{
    public float Timer;
    public float Interval;

    public int FrameCount;
    public int CurrentFrame;

    public int FrameWidth;
    public int FrameHeight;

    public Rectangle CurrentSprite;
    public Vector2 Size;
    public Vector2 Position;
    public ExplosionType MyType;
}

List<ExplosionsClass> Explosions;

Texture2D ShipExplosionTexture;
Texture2D AsteroidExplosiontexture;
```

2. Create an explosion when a ship or asteroid dies.

```
protected void CreateExplosion(Vector2 SpawnPosition, ExplosionType SpawnedExplosionType)
{
    ExplosionsClass NewExplosion = new ExplosionsClass ();

    NewExplosion.CurrentFrame = 0;
    NewExplosion.FrameCount = 12;
    NewExplosion.FrameWidth = 128;
    NewExplosion.FrameHeight = 128;
    NewExplosion.Size = new Vector2 (128, 128);
    // (1000ms / 30frames) = time per frame for 30 frames per second
    NewExplosion.Interval = 1000.0f / 30.0f;
    NewExplosion.Position = SpawnPosition;
    NewExplosion.CurrentSprite = new Rectangle (0, 0, 128, 128);
    NewExplosion.MyType = SpawnedExplosionType;

    Explosions.Add (NewExplosion);
}
```

3. Update explosions

```
private void UpdateExplosions(GameTime gameTime)
{
    List<ExplosionsClass> ToRemove = new List<ExplosionsClass> ();

    foreach(ExplosionsClass Explosion in Explosions)
    {
        if(Explosion.CurrentFrame > Explosion.FrameCount -1)
        {
            ToRemove.Add (Explosion);
            continue;
        }
        else
        {
            Explosion.Timer += (float)gameTime.ElapsedGameTime.TotalMilliseconds;

            if(Explosion.Timer > Explosion.Interval)
            {
                ++Explosion.CurrentFrame;

                Explosion.CurrentSprite = new Rectangle (
                    Explosion.CurrentFrame * Explosion.FrameWidth, 0,
                    Explosion.FrameWidth, Explosion.FrameHeight);

                Explosion.Timer = 0;
            }
        }
    }

    foreach(ExplosionsClass Explosion in ToRemove)
    {
        Explosions.Remove (Explosion);
    }
}
```

4. Draw explosions

```
private void DrawExplosions()
{
    Texture2D TempText;

    foreach(ExplosionsClass Explosion in Explosions)
    {
        if(Explosion.MyType == ExplosionType.ASTEROID)
        {
            TempText = AsteroidExplosiontexture;
        }
        else
        {
            TempText = ShipExplosionTexture;
        }

        spriteBatch.Draw (TempText,
            Explosion.Position,
            Explosion.CurrentSprite,
            Color.White, 0 ,
            new Vector2 (Explosion.FrameWidth / 2, Explosion.FrameHeight / 2),
            new Vector2 (Explosion.Size.X / Explosion.FrameWidth, Explosion.Size.Y /  Explosion.FrameHeight),
            SpriteEffects.None,
            0);
    }
}
```

And we're done. Now it's up to you to make your own changes to the game. Here are some suggestions

- Powerups
- Create more asteroids as the game progresses
- Big asteroids split into smaller ones when shot
- Multiple levels
- Sound (google xna / monogame sound FX for some ideas)
- New weapons
- Evil asteroid follows ship and stabs it with a sharpened toothbrush