RC\_Framework – Student Notes

Version 1.0.0 (2018 – Feb 14)

Table of Contents

[What is RC\_Framework 3](#_Toc506368376)

[Why use RC\_Framework 3](#_Toc506368377)

[Documentation for RC\_Framework 3](#_Toc506368378)

[Getting Started with RC\_Framework 3](#_Toc506368379)

[RC\_Framework files 4](#_Toc506368380)

[Glossary 6](#_Toc506368381)

## What is RC\_Framework

The RC\_framework is a teaching framework used at University of Canberra to teach game programming concepts. Students are free to modify and take away the framework and use it for any purpose they wish including publishing games made with it. The RC\_Framework is software in the Public Domain.

I do not maintain a git repository but several students have put various versions (some forked) of it publicly available on the web. Previously it was known as the UC\_Framework but that is an older though very similar version.

I release a new version yearly at the start of the subject Games programming and maintain that version for the rest of the year at the time of writing its version number is officially V3.18.1

The version is stored in Sprite3.cs as: public const string SpriteVersion ="V3.18.1";

RC\_Framework is distributed as source code in C#.

## Why use RC\_Framework

The main reason for the RC\_framework is that it is a teaching framework designed to be both a useful full games development tool but also to allow students a slow introduction to the many concepts in game programming without overwhelming them.

By insisting that students use the RC\_Framework it makes plagiarism more difficult and gives me good options for testing coding in the exam.

## Documentation for RC\_Framework

Er yes – good point

The main documentation is HTML extracted from the source code, using the usual C# notation, in recent years the code base has been fairly stable and much additional documentation was added to the code base in 2016.

Additional documentation is this manual and also the many tutorials, slides and examples provided during the course.

I use Doxygen to generate the documentation. The root HTML is : /Doc/html/index.html

## Getting Started with RC\_Framework

Because the versions of Monogame and Visual studio change a start-up tutorial is given to the class for that years configuration.

But basically to use the RC\_Framework you do the following

1. Load Visual Studio (a recent version – see start-up tutorial)
2. Perhaps at this time you will need to install xna since monogame uses some xna classes; I am advised that future versions of monogame will include the xna files.
3. Load MonoGame (a recent version – see start-up tutorial
4. Create a C# project using the “MonoGame Windows Project” option
5. Import the RC\_Framework source files into your project
6. Add the line “using RC\_Framework;” to your source files
7. Write and compile your code
8. Run your code

## Bugs in RC\_Framework

Any software of this complexity has a mix of bugs and design flaws. I do release patches each year with new features and bug fixes.

## RC\_Framework the future

Two things are definitely to be changed in future versions

1. Some kind of unification of SpriteList and RenderableList these two have so many common methods its kind of embarrassing that they are not unified in some way.
2. The removal of purpose built GUi objects and replacing them with a child of renderable or bounded renderable.

In general as I add more functionality to the framework, the abstraction level increases, thus making how it works more opaque to students (which is a bad thing). However it does mean less code to do complicated things (which is a good thing); SO! – it’s a balance.

I have several forks of the main RC\_Framework these include versions that embody much game functionality specifically:

* A microgame concept - a renderable that is itself a complete game (great for subgame in game);
* Versions with Tile handling (very ancient);
* Versions with more advanced animation systems (eg bounding box per frame);
* Versions that can re-size themselves to multiple screen resolutions (does not work for all; objects)
* Version with pixel perfect collision detection (incompatible with re-sizing version);

Some of these forks are very old and cannot easily be re-added to the main tree, or are only partial implementations.

Importantly some of these have legacy code in the main tree – variables with interesting names and nothing that uses them. So don’t stress if you come across an unused variable.

NOTE: I used to release the micro game source to students and am happy to do so late in the semester. When I did it early in the semester students would use it to shortcut their main assignment (and then complain when I docked them marks).

## RC\_Framework files

The source files for the RC\_Framework are:

|  |  |  |
| --- | --- | --- |
| **File name** | **Purpose** | **Notes** |
| RC\_RenderableParents.cs | The king of all things put on the screen abstract class – a must have file |  |
| RC\_Renderables.cs | Useful things to display that don’t need all the functionality of a sprite and do not need to worry about a render bounding rectangle. |  |
| RC\_RenderableAttached.cs | Renderables that attach to sprites like health bars and names |  |
| RC\_RenderableBounded.cs | Renderables that have or need a render bounding rectangle the rule is that the renderable does not draw outside its rectangle | \*1 |
| RC\_RenderableList.cs | Just a list of renderables for convenience, sometimes I use these as the basis for ‘Render layers’ |  |
|  |  |  |
| RC\_Sprite3.cs | In some ways the master file for the entire suite, this contains the most used class the sprite class (Sprite3). | \*2 |
| RC\_SpriteList.cs | Just a list of sprites for convenience.  While I also use these as the basis for ‘Render layers’; Sprite lists are also useful in collision detection, and many other aspects of graphic management. |  |
| RC\_Waypoint.cs | Classes waypoint and waypoint list – used by sprite3 |  |
|  |  |  |
| RC\_Utils2.cs | Methods and classes that are useful – see the pages in this manual |  |
| RC\_Utils3.cs | More useful methods and classes |  |
| RC\_LineBatch.cs |  |  |
|  |  |  |
| RC\_GameState.cs |  |  |
|  |  |  |
| RC\_Frame.cs |  |  |
|  |  |  |
| RC\_GUI.cs |  |  |
| RC\_GUIMenuRenderable.cs |  |  |
|  |  |  |
| RC\_PanZoom.cs |  |  |
|  |  |  |
| RC\_Particle.cs |  |  |
| RC\_PositionFactory.cs |  |  |
|  |  |  |
| RC\_UtilText.cs |  |  |
| RC\_RenderablesUtilText.cs |  |  |
| RC\_RenderableMulti.cs |  |  |
|  |  |  |
| RC\_Sound.cs |  |  |
|  |  |  |
| RC\_Sprite12Step.cs |  |  |
|  |  |  |
| RC\_StringList.cs |  |  |
|  |  |  |
| RC\_Surface.cs |  |  |
|  |  |  |
| RC\_Texture.cs |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

Note1 - There may be some exceptions to this with semitransparent objects.

Note2 – the sprite3 is a child of 3 other classes (renderable, bounded renderable and sprite3 parent) this is done for polymorphic extensibility, but is not used by most students and can be ignored.

Other Files and Directories:

|  |  |  |
| --- | --- | --- |
| **File or Directory** | **Purpose** | **Notes** |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

## Glossary

|  |  |
| --- | --- |
| **Term** | **Meaning** |
| Render Bounding Rectangle | A rectangle structure (usually stored in a variable called bounds) that defines a rectangle in which a graphic object is rendered. This may be different from a collision rectangle or bounding box |
| Collision Rectangle | See Bounding Box |
| Bounding Box | A rectangle that represents the points at which two graphic objects (usually sprites) collide (or are considered to be in a state of collision).  Bounding boxes come in several forms but most commonly are Axis aligned or Arbitarily aligned – see the lecture on collision detection for details. |
| Render layers | A graphic object or group of graphic objects that have the same render order (or z order), for example all the background objects, or all the explosions. See the lecture on layers for details |
| Render order | The order in which graphic objects are rendered which causes one object to appear in front of another because of occlusion. |
| Z order | A more mechanistic form of Render order usually using a number to represent render order. |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

## Screen Co-ordinates

## Help with screen co-ordinates

I set the screen to usually 800 by 600 pixels in size using lines like:

// to be put in game1 constructor after Content.RootDirectory = "Content";

graphics.PreferredBackBufferHeight = 600;

graphics.PreferredBackBufferWidth = 800;



Location 0,600

Location 0,0

Approximate Location 10,100

Location 799,0

Location 799,599

NOTE on a PC screen co-ordinates are slightly more straightforward than on a console attached to a TV, since a TV has a ‘Viewable Area’ (called a safe zone)which is not consistent from TV to TV. So when programming for the XBOX (which runs using a TV) there are additional issues concerning screen size vs usable/visible screen size, but on a PC keep it simple (eg 800 by 600 as shown above).

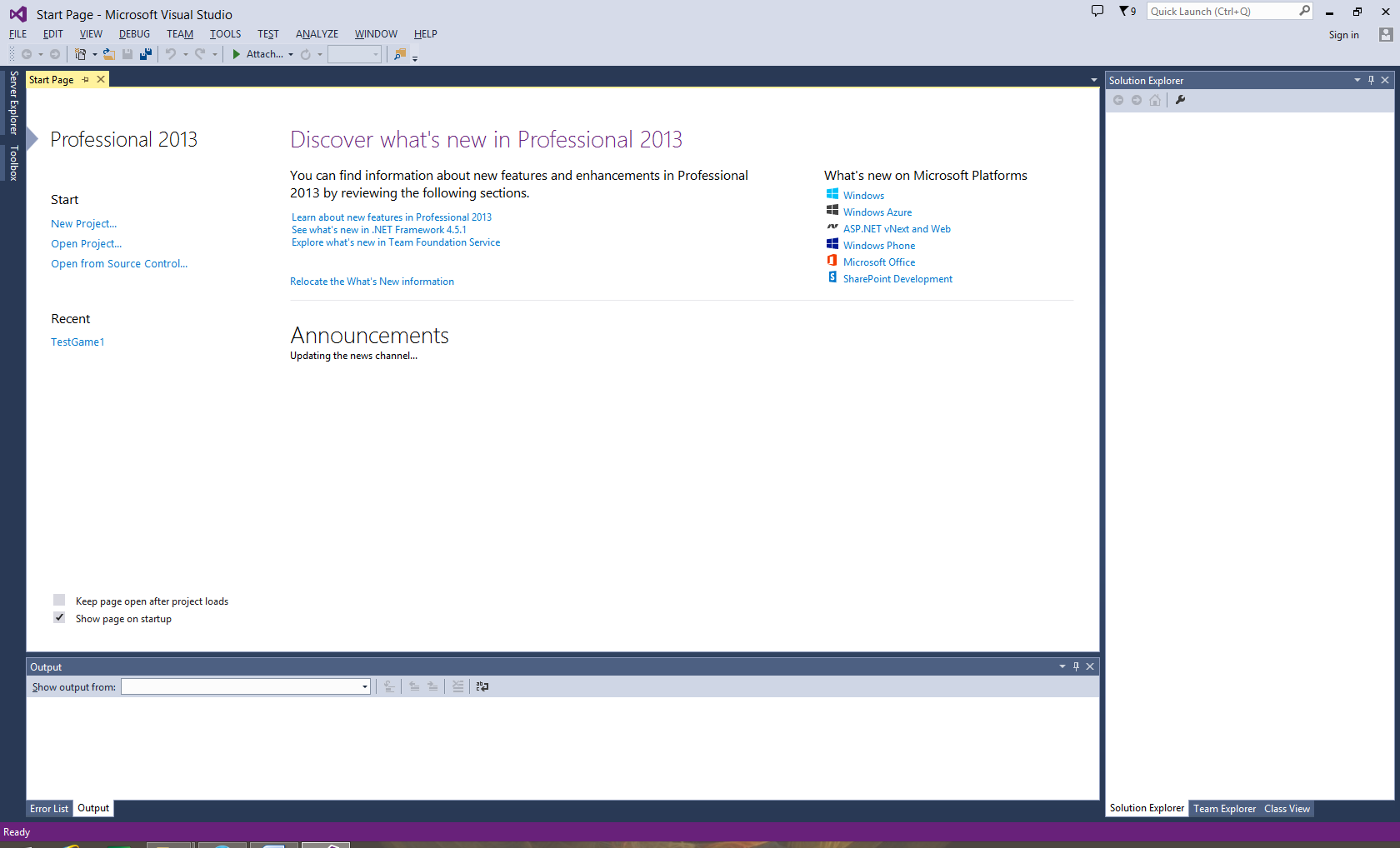
## Tutorials

### Tutorial 1 – Graphics without the RC\_Framework

Tutorial By Nicholas Londey and R. Cox

# Start Visual Studio 2015

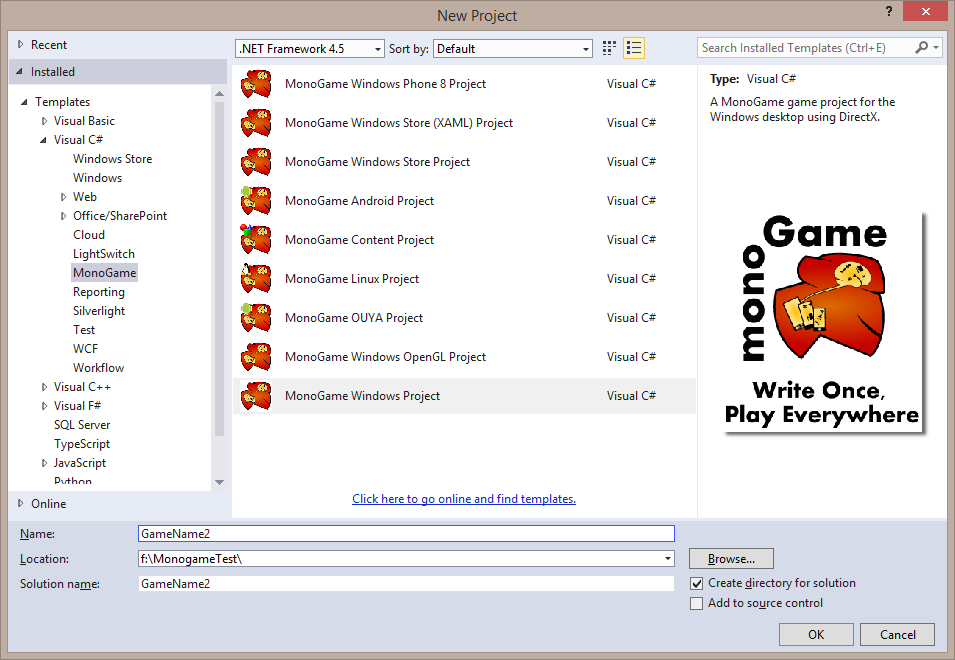
Start Visual Studio 2015 and you should see a screen that looks something like this:



Select the option New Project:

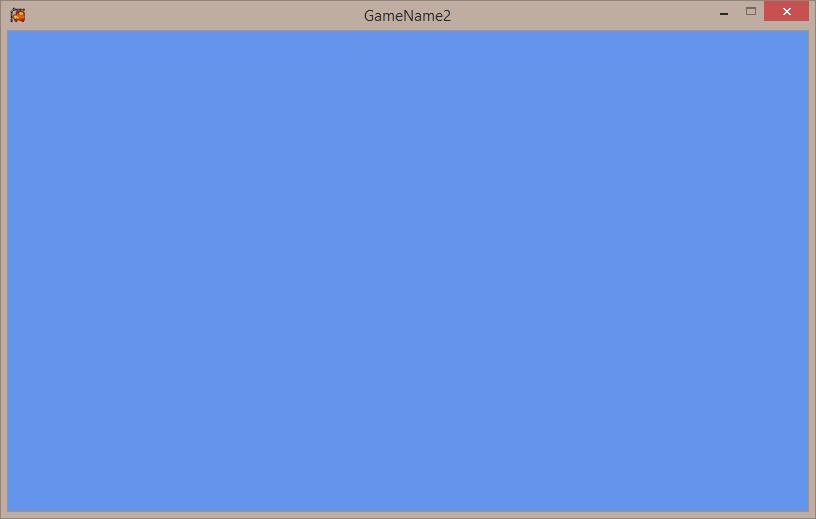
And select sensible Name, location and Solution name – I don’t know how the student labs are set up this year; so you may or may not be able to put your code on c: but wherever you put your code/solution, be sure to back it up to your pen drive regularly.

The project type will be C#, Monogame Windows Project



When you hit OK it will create a set of files for an empty game.

If you hit the compikle and run button (the green triangle) you should see something like this.



This is an empty game and indicates all is ok so far.

Well done you now have an empty game – a game that has a shell for future development but no actual code that does anything meaningfull.

Try changing the background colour in the file Game1.cs

Change the line ‘GraphicsDevice.Clear(Color.CornflowerBlue);’ to something like ‘GraphicsDevice.Clear(Color.Red);’

This should make it clear that this line controls the background colour of the screen.

## Program.cs – Main()

This file contains the program entry point Main(). Generally you will not need to modify this file. The only addition you might consider is to pass the command line arguments (args) to the game object’s constructor however this won’t be necessary for this tutorial. Command line arguments can often be a useful way to for example automatically load a specific level to improve iteration speed while debugging.

|  |
| --- |
| static void Main(string[] args)  {  using (Game1 game = new Game1())  {  game.Run();  }  } |

## Game1.cs - Game1 class

This file is created by the wizard and is prefilled with an outline for your game code. This file will contain the implementation of a class ‘Game1’ that inherits from Xna Framework’s Game class which forms the basis for all Xna games. The Game1 class contains a number of members and methods to override which will be discussed here

|  |
| --- |
| public class Game1 : Microsoft.Xna.Framework.Game  {  ... |

### Pre-implemented members of Game1 class

#### GraphicsDeviceManager graphics;

This member object is used to pre-configure the GraphicsDevice object that will be created by the base Game class. We will not need to do anything additional with this object for this tutorial however it can be used to do things such as start the game in full screen mode.

|  |
| --- |
| GraphicsDeviceManager graphics;  ...  graphics = new GraphicsDeviceManager(this); |

#### SpriteBatch spriteBatch;

The initial game wizard also adds a member SpriteBatch object. This is included purely as an example and is not actually used by the initial program other than to be constructed. An introduction to sprites will be covered in a latter tutorial.

|  |
| --- |
| SpriteBatch spriteBatch;  ...  spriteBatch = new SpriteBatch(GraphicsDevice); |

### Inherited Properties

#### Components Property

Components are a way of organising game code into reusable size chunks such as a menu screen or a collision manager. An in depth discussion of game components will be left for a latter tutorial however they are a concept in Xna to be worth mentioning at this time if for no other reason than you will come across them in examples from the web.

For software engineers and game programming students is better to construct our own game framework object classes since the Microsoft ones tend to be very abstract, they are a lot easier to undersnd once you have built a few classes of your own and seen how things work.

#### Content Property

This property provides an initial ContentManager object for convenience. ContentManagers are the mechanism used to load data at runtime from the content project. A detailed explanation of content loading will be covered in a latter tutorial.

#### GraphicsDevice Property

The base Game class creates graphics device object exposed though this property. The graphics device object is the object though which we control the video card. When one of its draw function are called the graphics device object it saves out a number of commands into a command buffer that the video card will later read back and execute in the order they were issued. This command buffer is the mechanism used to allow the CPU and the graphics chip to work concurrently and is more explicitly exposed in game console programming and newer versions of DirectX.

### Overridden methods

Game code for Xna games is implemented by overriding functions from the framework’s Game class. These provide the means to provide custom code for initialisation and the game loop.

#### Constructor

The constructor is the first stage of start-up and is where you should construct any objects that will register as service providers. Note that at this time the GraphicsDevice and Services will not yet exist.

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

#### Initialize()

Initialize is where you should put most of the code to setup your game. At this time game services should already be registered and can be used. When base.Initialize() is called the Intalize() functions of all current game components is also called.

|  |
| --- |
| protected override void Initialize()  {  // TODO: Add your initialization logic here  base.Initialize();  } |

#### LoadContent()

This function is where you should load an game assets you will need to run your game. The construction of the sprite batch is included only as an example.

The game assets will include sound and image files of various kinds, and a future tutorial will demonstrate how we can load these assets.

|  |
| --- |
| 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  } |

#### UnloadContent()

For the most part you will not need to worry about adding clean up code to this function. As most of the content you will load will be done using content managers the cleanup will be done for you.

|  |
| --- |
| protected override void UnloadContent()  {  // TODO: Unload any non ContentManager content here  } |

#### Update()

This is the function where you should put all your frame to frame game logic such as responding to controller input, moving around enemies, etc. The default behaviour of Xna is for this function to be called 60 times a second (60Hz) with a fixed time step (16.6ms). It is recommended that you keep this configuration as programming game logic for a variable time step (i.e. a inconsistent frame rate) is significantly more difficult. If the game does run into a performance trouble spot, Update() will continue to be called with an elapsed game time of 16.6ms even if the elapsed real time is greater.

When the base.Update(gameTime) is called the update functions of all current game components will also be called.

|  |
| --- |
| protected override void Update(GameTime gameTime)  {  // Allows the game to exit  if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)  this.Exit();  // TODO: Add your update logic here  base.Update(gameTime);  } |

#### Draw()

During this function is the only time you should do any rendering using the graphics device. At the same time you should be careful not to place any game logic in this function. This is because if the game is running slower than its target framerate Xna will start dropping frames (skipping calls to Draw()) in order to catch up while continuing to call Update().

The only pre-implemented code here is a call to the graphics device to clear the section of the screen controlled by our graphics device object and fill it with the colour CornflowerBlue. If we do not explicitly clear the screen it may still contain of the contents from the previous frame. Clearing also resets various internal video card states and so should always be called once per frame.

An important concept to grasp in real time computer graphics is that for the most part we redraw the entire scene from scratch every frame. Any appearance of consistency between frames is entirely in our own game code and is not understood by Xna, DirectX or the video card.

|  |
| --- |
| protected override void Draw(GameTime gameTime)  {  GraphicsDevice.Clear(Color.CornflowerBlue);  // TODO: Add your drawing code here  base.Draw(gameTime);  } |

## Now lets set the screen resoloution

In the constructor after the line

Content.RootDirectory = "Content";

Add the following lines

graphics.PreferredBackBufferHeight = 600;

graphics.PreferredBackBufferWidth = 800;

This should ask Monogame to provide you with a suitable 800 by 600 screen.

Build and run to be sure it still works

## Now to import a background image

There are two ways to import a background image it can be read from a data file (this is the way I usually use) or it can be included in the game resources in content and loaded by the content manager (which I usually do before releasing the game, in final builds).

To do this I uses a helper routine (which I later supply in the framework file Utils2.cs) but for now we will copy it into the program so add it now.

// to be put into Game1 as a method to allow

// loading textures from disk

public static Texture2D texFromFile(GraphicsDevice gd, String fName)

{

// note needs :using System.IO;

Stream fs = new FileStream(fName, FileMode.Open);

Texture2D rc = Texture2D.FromStream(gd, fs);

fs.Close();

return rc;

}

NOTE You will need to add the line using System.IO; as well.

Done that – Good !

Now lets add the variable we will need to store the texture which is :

Texture2D texBack = null;

I put it after the line SpriteBatch spriteBatch;

Now in the LoadContent() routine put the call to load in the texture it should be after the spritebatch is created as shown below;

spriteBatch = new SpriteBatch(GraphicsDevice);

texBack = texFromFile(GraphicsDevice, @"F:\MonogameTest\TestGame1\TestGame1\back3.png");

NOTE: edit the path to where you put the back3.png file

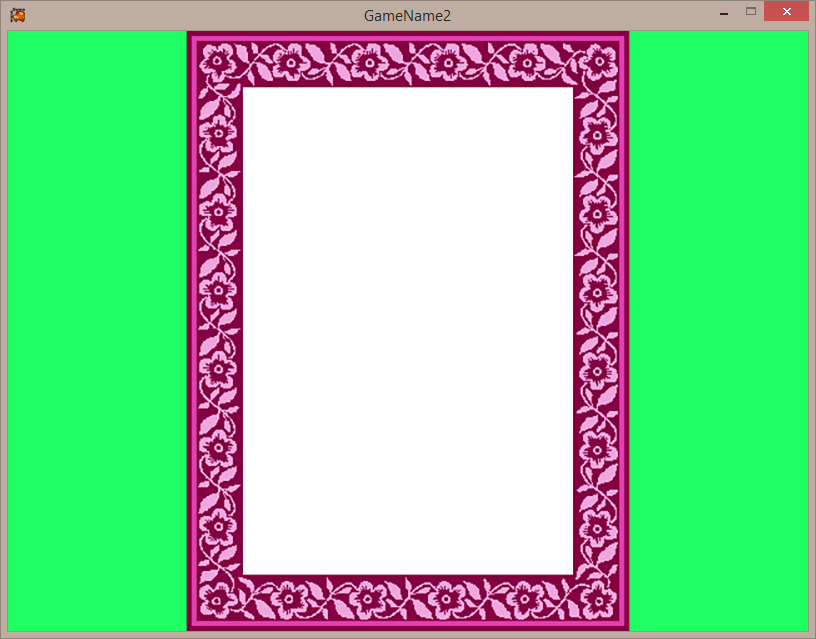
Now that we have the texture read in we need to display it so add the following to the draw routine:

spriteBatch.Begin();

spriteBatch.Draw(texBack, new Vector2(0,0), Color.White);

spriteBatch.End();

Now run the program it should look something like this :



Which is the start of a simple breakout game that we will build in tutorials:

## Lets continue and add a paddle controlled by the keyboard

We need a texture to contain the paddle (which is a simple red brick) and also some variables to contain its position, and an object to access the keyboard so add the following to the program:

Texture2D texpaddle = null;

float xx = 350;

float yy = 500;

KeyboardState k;

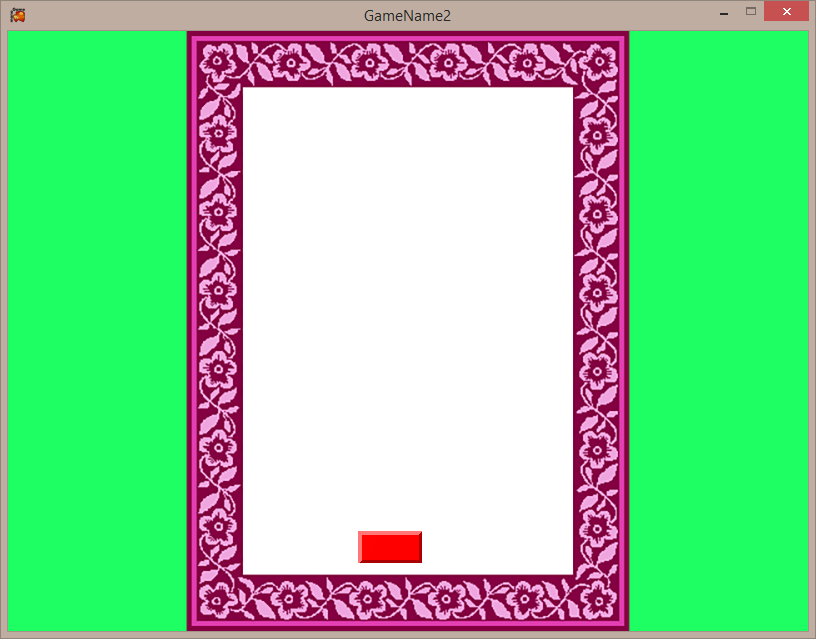
Ok now add code to load the texture texpaddle from the file red64x32.png

texpaddle = texFromFile(GraphicsDevice, @"F:\MonogameTest\TestGame1\TestGame1\red64x32.png");

and add code to draw the paddle this will use spritebatch as before but it should occur after you draw the background.

spriteBatch.Draw(texpaddle, new Vector2(xx,yy), Color.White);

build and run the program the result should look something like this:



As a final exercise lets make the paddle move to the commands of the keyboard

## Moving the paddle:

In the update routine put the following code which detects if the right arrow key is depressed and moves the paddle 3 if it is – remember this routine runs many times a second so the effect is that of a constantly moving paddle

k = Keyboard.GetState();

if (k.IsKeyDown(Keys.Right))

{

xx = xx + 3;

}

Given that ‘Keys.Right’ represents the right arrow and that ‘Keys.Left’ represents the left arrow add code to move the paddle both right and left.

Also vary the xx+3 (eg xx+1 , and xx+5) to vary paddle speed

## Moving the paddle to the edge:

The edges of the flower frame are at 235 and 565 modify the code with an extra if statement or two so the paddle wont move out of the frame. (Hint you can get the width of the paddle with texpaddle.Width)

Finally move the paddle down to the bottom of the frame (y coordinate 543) so we are ready to start the next step in creating the breakout game.

For convenience I added the following variables.

int paddleSpeed = 3;

int lhs = 236;

int rhs = 564;

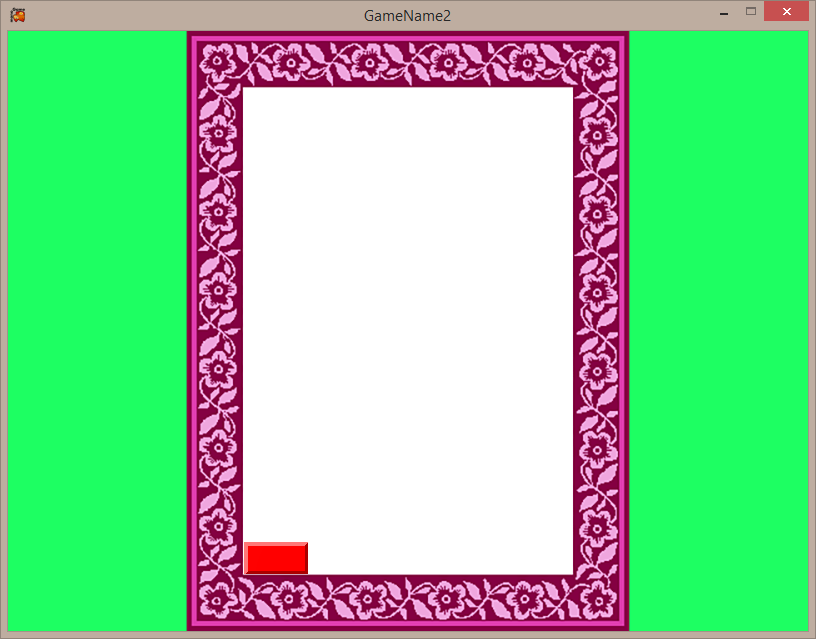
int bot = 543;

Next week we will rewrite this uysing sprites which wont change much – but till then be patient.

## For the Super Keen:

Try using the graphic ‘white64x32.png’ instead of the ‘red64x32.png’ but vary the color in the spritebatch draw command, consider how this could be used to change the paddle in play without requiring an additional graphic object.

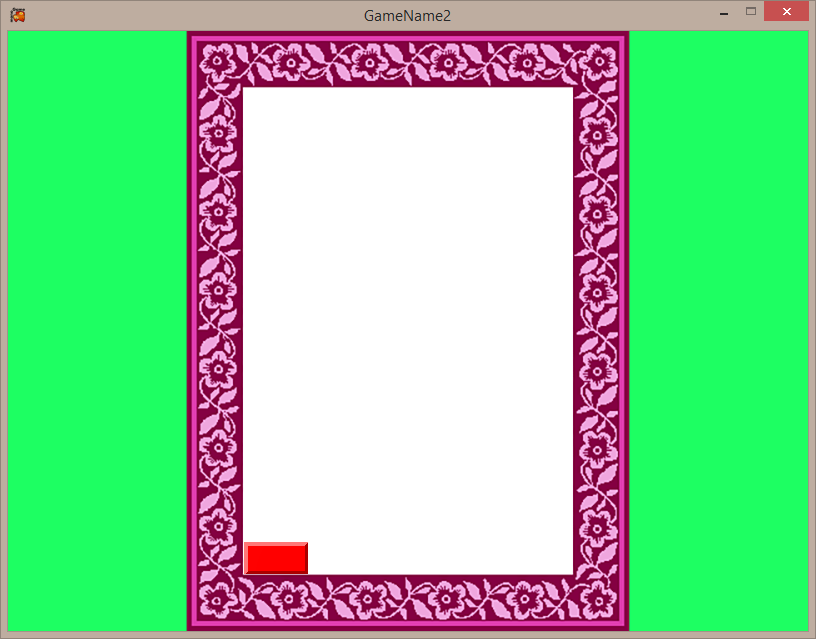
My final screen



### Tutorial 2

Today we will modify the tutorial from last week so that it uses sprites this will bring us no real benefit due to the simplicity of this initial program, but you will see that it brings us many benefits as the program complexity increases and the game becomes more complex.

You should create a copy of last weeks tutorial – then make the following changes



## Step 1

Include the following files:

RC\_Linebatch, RC\_renderables, RC\_RenderablesBounded, RC\_renderableParent, RC\_Sprite, RC\_Waypoint, RC\_Utils2, RC\_FrameSource and RC\_Utils3

|  |  |
| --- | --- |
| **File** | **Purpose** |
| RC\_Linebatch, | Draws lines, rectangles and circles |
| RC\_renderables, | The parent class for all renderables |
| RC\_RenderablesBounded, | The parent class for all renderables that have a render boundary |
| RC\_renderableParent, | The parent class for all renderables |
| RC\_Sprite, | The general sprite class |
| RC\_Waypoint, | Waypoint movement system – will be explained when I cover movement |
| RC\_Utils2 | Some utility routines see slides for hints |
| RC\_Utils3 | For the future |

Once added please add the line ‘using RC\_Framework;’ then recompile – it should be error free

## Step 2

Now we will replace the paddle with a sprite and also the background with a renderable, and finally add a play area rectangle.

Subsequent tutorials will cover adding the blocks and movement of the ball

You will need to add the following variables

* string dir = @"F:\MonogameTest\TestGame1\TestGame1\"; //\*\*\*
  + this is just for the futre so we have a singe directory so we need to change less code when we move the program to new machines or directories
* Sprite3 paddle = null;
  + This will become the paddle sprite
* ImageBackground back1 = null;
  + This will be used to display the background
* Rectangle playArea;
  + This will define the inside rectangle of the play area
* bool showbb = false;
  + Just a falag to help us see bounding information and hotspots
* KeyboardState prevK;
  + Somewhere to store previous keyboard information so we can detect a single key click (as opposed to the 30 a second we get on a key depress)

* int top = 56; // \*\*\*
  + Bounds the top of the play area

Then compile – keep it error free;

#### Step 3

First initialise linebatch in the LoadContent routine

LineBatch.init(GraphicsDevice); // \*\*\*

Then optionally add a directory to the texture file paths

texBack = Util.texFromFile(GraphicsDevice, dir+"back3.png"); //\*\*\*

texpaddle = Util.texFromFile(GraphicsDevice, dir+"red64x32.png"); //\*\*\*

Now create the sprite that will become the paddle

paddle = new Sprite3(true,texpaddle,xx,yy);

paddle.setBBToTexture();

Now create the renderable that will become image background

back1 = new ImageBackground(texBack,Color.White,GraphicsDevice);

And finally create a rectangle for the future which bounds the actual play area of the breakout game

playArea = new Rectangle(lhs,top,rhs-lhs,bot-top); // width and height

Then compile keep it error free

## Step 4

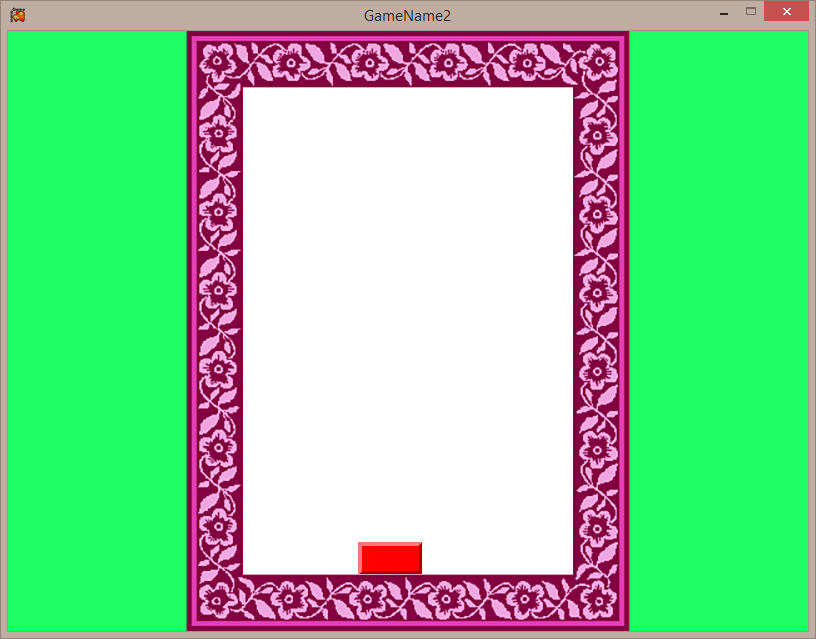
Delete the previous spritebatch.draw commands from the routine Draw

Update the draw routine to draw the background then the paddle

back1.Draw(spriteBatch); //\*\*\*

paddle.Draw(spriteBatch); //\*\*\*

Compiling and running you should see something like this



## Step 5

Now we will modify the paddle movement routines in Update so they work in exactly the same way as they did before but act on a sprite. This is not necessarily how it would be done if you started with sprites but should help to avoid confusion until we cover movement in a lecture.

k = Keyboard.GetState();

if (k.IsKeyDown(Keys.Right))

{

if (paddle.getPosX() < rhs - texpaddle.Width) paddle.setPosX(paddle.getPosX() + paddleSpeed);

}

if (k.IsKeyDown(Keys.Left))

{

if (paddle.getPosX() > lhs) paddle.setPosX(paddle.getPosX() - paddleSpeed);

}

Compile and run – you should be able to move the paddle

## Step 6

Now lets look at bounding and hotspot information but we need to be able to detect a single keystroke to switch on and off the variable showbb.

To do this we must know the prevous state of the keyboard so modify update

prevK = k;

k = Keyboard.GetState();

and also add into update the following code to flip the state of showbb

if (k.IsKeyDown(Keys.B) && prevK.IsKeyUp(Keys.B)) // \*\*\*

{

showbb = !showbb;

}

Cool does it still compile – nearly done

Finally add the hotspot and bounding information display into draw

if (showbb)

{

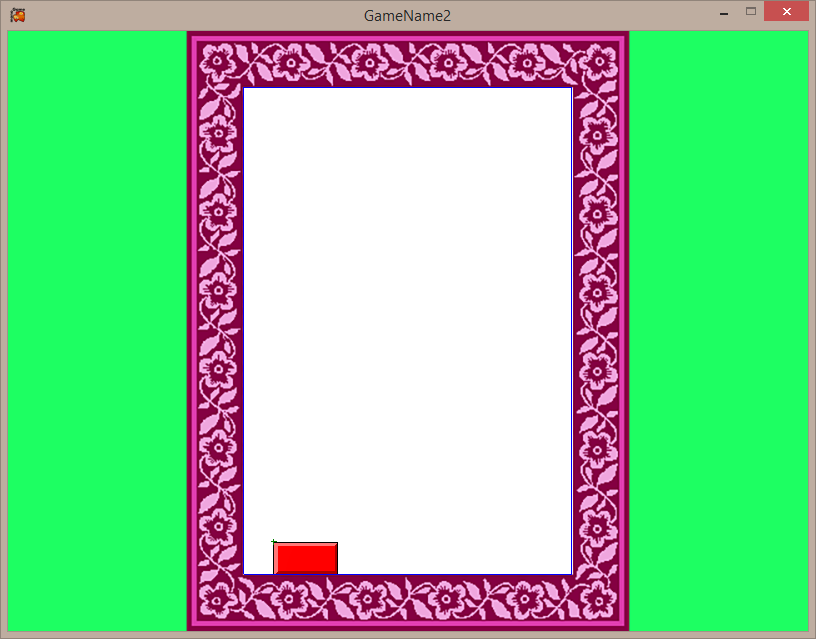
paddle.drawBB(spriteBatch, Color.Black);

paddle.drawHS(spriteBatch, Color.Green);

LineBatch.drawLineRectangle(spriteBatch, playArea, Color.Blue);

}

When run you should be able to see the hotspot , and bounding information displayed after you hit the b key as shown below



## Step 7

Now if you are keen add a second sprite that is the ball – and make it sit on and move with the paddle ready to hit space and launch it into the yet to be coded bricks