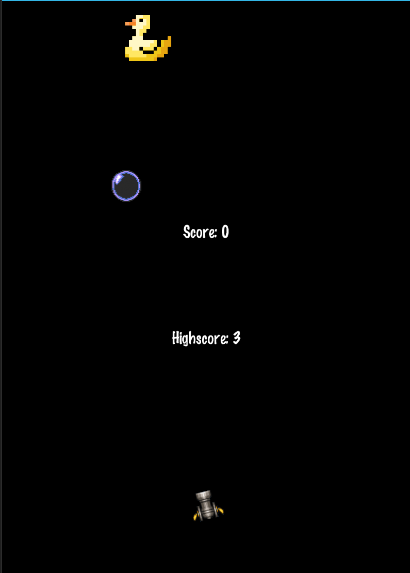
**How to build a basic Android game**

By Andres Felipe Rojas Jaramillo, March 2017

**Introduction to problem:**

Making videogame apps has been a recently increasing trend in the last couple of years. With Xamarin, it is easy to get started in game development thanks to the many engines it implements. For this small game in particular, we will be working with CocosSharp.

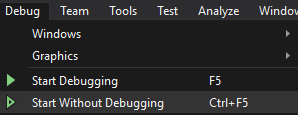
This guide should give you an idea on how games work: the logic, physics... By the end we will have a game which consists of shooting a duck which is constantly moving, as you can see in the image below. We will get started by creating a new project for CocosSharp Android. Lets name it “CannonGame”.

**Note:**

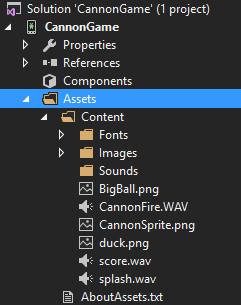
You can get the required image files and sounds files (as well as the finished project) at : <https://github.com/AndresFRJ98/Xamarin-CannonGame/tree/master/RequiredFiles> .

Also, to test our current code, firstly want to emulate an Android device. Make sure Xamarin is emulating an Android device. Then, to actually run our project, we can go over to debug -> Start Without Debugging. I recommend remembering the shortcut: **CRTL + F5**.



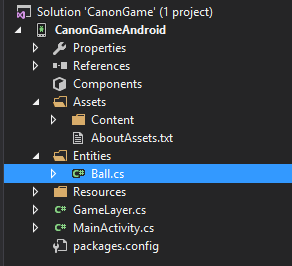
Setting up

1. Once you have created a new CocosSharp project, the first thing we want to do is to change our screen resolution values, so our game runs in portrait mode and not landscape and the background color.
   1. Go into the MainActivity.cs file located in the Solution Explorer (usually on right hand side).
   2. Scroll down to the “void loadGame” method, and inside the ‘if’ statement, we will see two variables (Ints): width and height. We want to make sure we have width set as 768, and height as 1027.
   3. Now for the background color, we simply go to “GameLayer.cs”, and on the public GameLayer method, we switch the color from Blue to Black.



1. Games require image and sound files to improve the user experience. We need to make sure Xamarin knows what files we are working with, so we need to locate them in the “Assets/Content” folder. You can just drag the files directly there inside Xamarin, or add them manually to the folder.

It is advisable to download the same files and keep the same names as shown. On the image to the right.

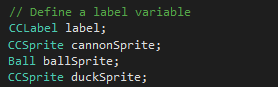
1. Finally, we need to create a directory for our “ball” entity. Entity is game design terminology. As defined in the Xamarin website: game entities are classes that define objects needing rendering, collision, physics, or artificial intelligence logic.

So lets go ahead and create the directory, and the file which we will use to define the ball.

Create a new folder, with the name Entities, and then add the C# class named “Ball.cs” to it. We need to make “Ball.cs” refer to the same namespace as our “GameLayer.cs”.

**Adding our objects to the project**

If we go over to our “GameLayer.cs” file (what we are mainly going to work in), we can see that the template includes quite a lot of functions already, which will come in very useful.

Inside the “GameLayer”, we see that the template includes a “CCLabel label”, which refers to a CocosSharp Label object. Next to it, we want to declare the objects we are going to work with: sprites and labels.

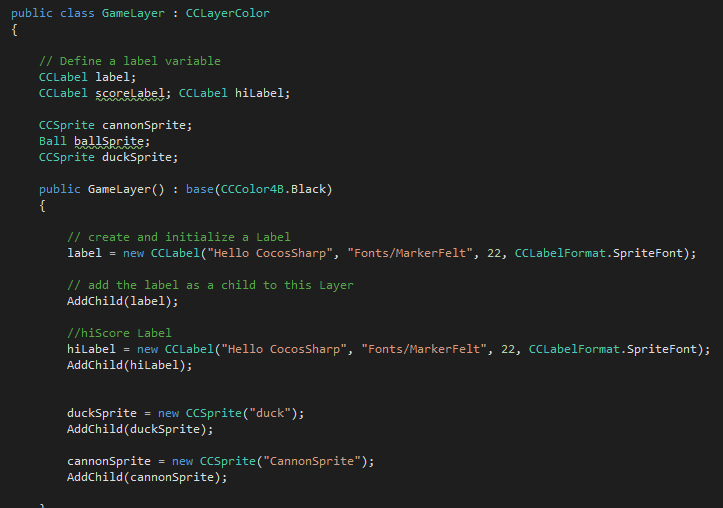
Sprites. Just next to the label declaration, we want to go ahead declare our sprites in the following way: “CCSprite cannonSprite;” ”CCSprite duckSprite” and finally “Ball ballSprite”. Ball is an object of a class, this allows us to treat it like an entity.

Under the “AddChild(label)” statement, we want to create our duck. So we do: “duckSprite = new CCSprite("duck");” followed by “AddChild(duckSprite);”.

Similarly, just after that, we write: “cannonSprite = new CCSprite(“CannonSprite”);” followed by “AddChild(cannonSprite);”.

Note that “duck” and “CannonSprite” in the parenthesis after CCSprite requests a file with that same name in the Assets folder.

Labels. For labels, we will be displaying two values: score, and high score. We have one label already (will be used for score) so lets add another one for high score. Next to the previous declarations, add a “CCLabel hiLabel;”, and for now we can initialize them by just copy and pasting the “label” initialization and changing the variable name to “hiLabel”.

****

**Adding our objects to the scene**

Labels

Right now, if we run our game, we will only see the default label being displayed. We need to add our second label which will display the high score. To do this we do the following:

1. In “GameLayer.cs” find the AddedToScene method. Here we can find the code which adds the default label onto the scene, so now we want to do the same thing, but for our own label.
2. Right after the default label statement, we add our own: hiLabel.Position = bounds.Center;
3. As this would make the labels appear right on top of each other, we want to separate them a bit so we can do: hiLabel.PositionY -= 100;

Cannon & Duck sprites

The cannon will be positioned at the bottom of the screen, in the center of the X axis. Our duck however, will be at the top of the screen. Right underneath our label statements we add the following:

// Position sprites

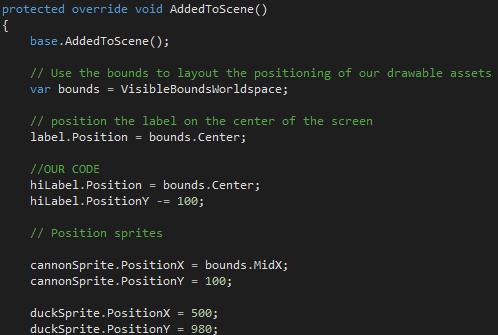
cannonSprite.PositionX = bounds.MidX;

cannonSprite.PositionY = 100;

duckSprite.PositionX = 500;

duckSprite.PositionY = 980;





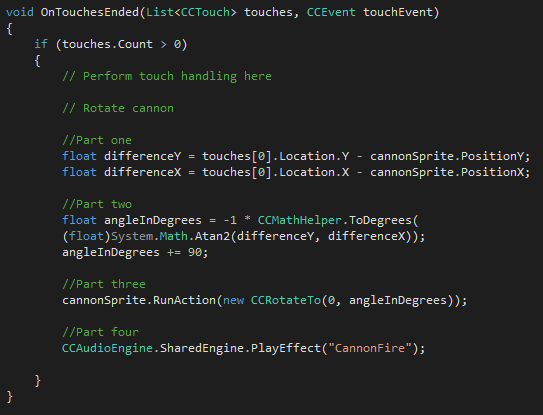
**Handling touch events to our project**

The code from this part, and what the solution looks like at this point.

The way our player interacts with the game is through tapping the screen. Right now, our project does not account user touches in any way, so that’s what we will take care of now, in void OnTouchesEnded (right after our last section).

The first thing we want, is the user touch to make the cannon rotate around. This concept requires some understanding of maths (trigonometry). Briefly, what we require is the following:

1. Identify where the user has touched on the screen with regards to our cannon.
2. Calculate the angle between that point and our cannon.
3. Rotate our cannon.
4. Play a sound to give feedback to the user that the cannon is shooting.



//Part one

float differenceY = touches[0].Location.Y - cannonSprite.PositionY;

float differenceX = touches[0].Location.X - cannonSprite.PositionX;

//Part two

float angleInDegrees = -1 \* CCMathHelper.ToDegrees(

(float)System.Math.Atan2(differenceY, differenceX));

angleInDegrees += 90;

//Part three

cannonSprite.RunAction(new CCRotateTo(0, angleInDegrees));

//Part four

CCAudioEngine.SharedEngine.PlayEffect("CannonFire");

**Implementing frame logic to our project**

If you are unfamiliar with the term “frames”, it refers to the still images which when displayed in quick succession, create the illusion of movement. With frame logic, we mean conditions which are considered in every frame to create its contents, in our case for example, we want the duck to move left and right across the screen. For this, the duck moves every frame, and we need to take care of it in every frame.

To begin implementing frame logic, we need to return to the public GameLayer() method and at the end of it add: Schedule(RunGameLogic); .

Now right after the void OnTouchesEnded method closes, we want to create a new method: void RunGameLogic(float frameTimeInSeconds) . This is method is where we take care of all the frame logic.

Now, to move the duck we can simply increment/decrement its position on the x-axis by an amount every frame. Right at the start of our method, we can type:

duckSprite.PositionX += 3;

We want our duck to bounce on the edges of the screen. For convenience, we will store the position of the right-most and left-most part of our duck under two variables:

// Duck edges

float duckRight = duckSprite.BoundingBoxTransformedToParent.MaxX;

float duckLeft = duckSprite.BoundingBoxTransformedToParent.MinX;

Similarly, we do the same for all the four edges of the screen:

// Screen edges

float screenRight = VisibleBoundsWorldspace.MaxX;

float screenLeft = VisibleBoundsWorldspace.MinX;

float screenTop = VisibleBoundsWorldspace.MaxY;

float screenBot = VisibleBoundsWorldspace.MinY;

Perhaps you have noticed already, but if we run our solution right now, our duck will move along the x-axis and disappear off-screen. However, this is not what we want. What we want is: if the duck is going off-screen on the right side, start moving towards the left and vice versa. We can check if this condition is true or false every frame with the following statement:

bool turn = (duckRight > screenRight) || (duckLeft < (screenLeft));

Now, we can simply make an if statement which depends on our “turn” Boolean, and it will move the duck in the correct direction. For this, we will create a global variable which will be -1 or 1 depending on which direction the duck needs to move, and we will multiply our statement which modified the duck’s position by this value.

1. At the top of our “GameLayer.cs”, next to our sprite declarations, we will include an additional variable: int dir = 1; .
2. Then, we need to make the if statement which depends on our bool turn , back down in our RunGameLogic method. As well as changing the direction the duck is moving, we want to flip its sprite so it faces the direction of its movement.

if (turn)

{

dir \*= -1;

if (dir < 0)

{

duckSprite.FlipX = false;

}

else

{

duckSprite.FlipX = true;

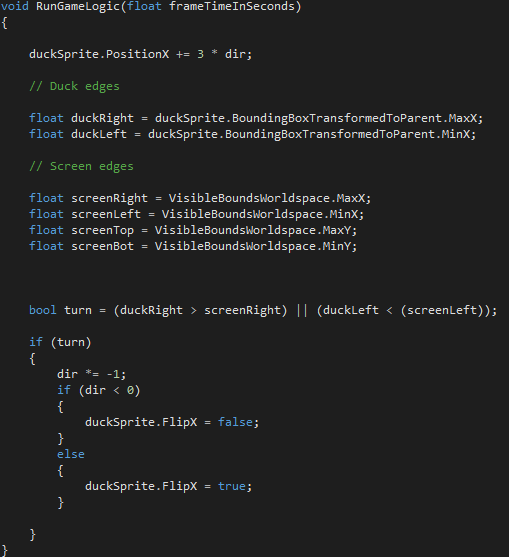
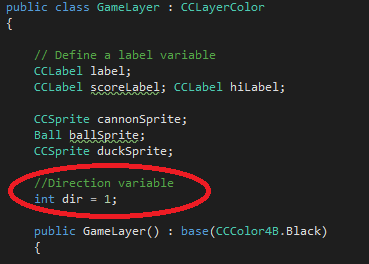
}

}

1. Finally, we need to go back to our statement where we make the duck move and multiply it by our direction value:

duckSprite.PositionX += 3 \* dir;

Now we have only a couple of things to take care of now: creating the cannon balls and the score system!



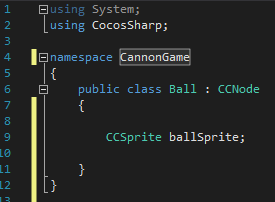
Note that we want to change the duck’s position and THEN define our duckRight and duckLeft variables.

**Adding the cannon balls to our project**

The only sprite we are missing is the cannon ball. In one of the previous sections we created a new Folder which contained the Ball.cs file. We are going to add the cannon ball’s functions into that file.

At the top of our Ball.cs file, make sure that Xamarin knows this class file belongs to a CocosSharp project by making sure the statement: using CocosSharp; is at the top. Also make sure the namespace is the same as our other files.

Now, we begin to write our class. What we want is a class which behaves like a sprite, in which we can alter the position of the ball.

1. All of our code will be contained inside a public class Ball : CCNode. This is declared right after our namespace, as we can see in the image to the right.
2. Now, inside our Ball class, we want to declare a CCSprite ballSprite; similar to what we have done previously.
3. After this, we need to create getter/setter methods for two variables which will be contained inside our class: the velocity in the X axis and the Y axis. Right after our sprite declaration we place this:

//Getter Setter

public float VelocityY

{

get;

set;

}

public float VelocityX

{

get;

set;

}

1. Finally, we need to add the constructor which selects our sprite image, adds it to our scene and runs the Ball’s frame logic (which is moving it by the velocity). (Note that in new CCSprite("BigBall") “BigBall” corresponds to the filename for our ball sprite)

public Ball() : base()

{

//Our frame by frame action

void ApplyVelocity(float time)

{

PositionX += VelocityX\*time;

PositionY += VelocityY\*time;

}

ballSprite = new CCSprite("BigBall");

// Making the Sprite be centered makes

// positioning easier.

ballSprite.AnchorPoint = CCPoint.AnchorMiddle;

this.AddChild(ballSprite);

//Starts our frame logic

this.Schedule(ApplyVelocity);

}

Our Ball class is now finished, we need to make sure our “GameLayer.cs” file uses this to create the ball now. Before doing this, we want to create a Boolean variable next to our previous global variables at the top: bool active = false; This variable lets our game know if the first ball has exists in the scene.

As we want our ball to be created by the touch of the user, we need to go over to our void OnTouchesEnded method, where we rotated the cannon before. After the cannon rotation, we place the following:

//Create and shoot ball

if (active)

{

ballSprite.RemoveFromParent(true);

}

ballSprite = new Ball();

AddChild(ballSprite);

active = true;

ballSprite.VelocityX = 0;

ballSprite.VelocityY = 0;

ballSprite.Position = cannonSprite.Position;

CCVector2 velocity = new CCVector2(0, 650);

RotateVector(ref velocity, angleInDegrees);

ballSprite.VelocityX = velocity.X;

ballSprite.VelocityY = velocity.Y;

As you can see, there is method which we have not added here yet: RotateVector . We need to add it to our file, I recommend placing it after the void RunGameLogic method.

RotateVector(ref CCVector2 vector, float cocosSharpDegrees)

{

// Invert the rotation to get degrees as is normally

// used in math (counterclockwise)

float mathDegrees = -cocosSharpDegrees;

// Convert the degrees to radians, as the System.Math

// object expects arguments in radians

float radians = CCMathHelper.ToRadians(mathDegrees);

// Calculate the "up" and "right" vectors. This is essentially

// a 2x2 matrix that we'll use to rotate the vector

float xAxisXComponent = (float)System.Math.Cos(radians);

float xAxisYComponent = (float)System.Math.Sin(radians);

float yAxisXComponent = (float)System.Math.Cos(radians + CCMathHelper.Pi / 2.0f);

float yAxisYComponent = (float)System.Math.Sin(radians + CCMathHelper.Pi / 2.0f);

// Store the original vector values which will be used

// below to perform the final operation of rotation.

float originalX = vector.X;

float originalY = vector.Y;

// Use the axis values calculated above (the matrix values)

// to rotate and assign the vector.

vector.X = originalX \* xAxisXComponent + originalY \* yAxisXComponent;

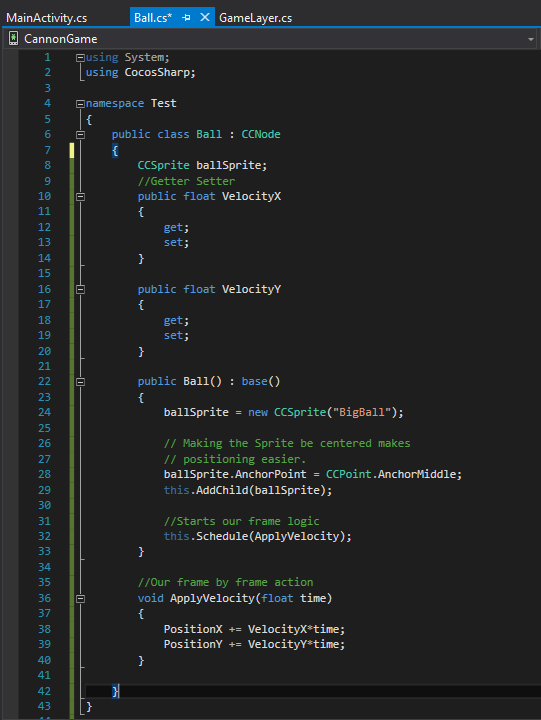
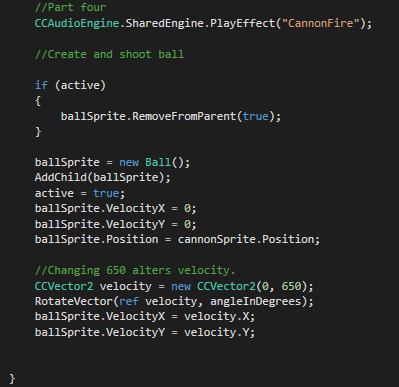
vector.Y = originalX \* xAxisYComponent + originalY \* yAxisYComponent;

}

}

This method takes care of all the maths behind knowing towards which direction to accelerate the ball in. If you wish to alter the balls speed, simply modify the second value in CCVector2 velocity = new CCVector2(0, 650); . For example, if we want to make it faster, we change 650 to 1000.

At this point we have practically everything complete. Now we just need to let our game know when we hit the duck with our ball and when the ball goes off-screen, and modify our score accordingly.

****

void OnTouchesEnded with the new additions, right after what we last added.

Ball.cs. Make sure the namespace is the same as your other files!

**Implementing score**

The way we want to give score points is simple: if the ball collides with the duck, award a point. We also need to remove the ball once it has collided with the duck. As we need to check for this every frame, we need to implement this in the void RunGameLogic method, after our last statement (duck movement). Before this however, we want to add 2 final global variables at the top of the file (with our other global variables) :

//Score and hiscore variables.

int score = 0;

int hiscore = 0;

Once these are declared, we can add our scoring system in void RunGameLogic .

//Score system

label.Text = "Score: " + score;

hiLabel.Text = "Highscore: " + hiscore;

if (active)

{

bool scored = ballSprite.BoundingBoxTransformedToParent.IntersectsRect

(duckSprite.BoundingBoxTransformedToParent);

if (scored)

{

ballSprite.Position = cannonSprite.Position;

ballSprite.RemoveFromParent(true);

active = false;

ballSprite.VelocityX = 0;

ballSprite.VelocityY = 0;

CCAudioEngine.SharedEngine.PlayEffect("score");

score++;

}

if (score >= hiscore)

{

hiscore = score;

}

//Remove ball if its not on the screen & reset score.

bool onscreen = ballSprite.BoundingBoxTransformedToParent.IntersectsRect(VisibleBoundsWorldspace);

if (onscreen == false)

{

ballSprite.RemoveFromParent(true);

active = false;

CCAudioEngine.SharedEngine.PlayEffect("splash");

score = 0;

}

}

As seen before, please note that both CCAudioEngine.SharedEngine.PlayEffect("score"); and CCAudioEngine.SharedEngine.PlayEffect("splash"); plays the sound file with the name “score” and “splash” respectively. Make sure your file names are equal to whatever you put inside the brackets!

Finally we only need to add two more things to finish!

1. While not mandatory, perhaps we want our duck to move faster depending on our score. We can go back to where we set its movement, at the start of the void RunGameLogic and change it to this:

//Increases speed by current score

duckSprite.PositionX += (3 + score) \* dir ;

1. Finally, we want to reset the score if the player fires a shot whilst the ball is still active, in order to make him do precise shots, as previously he could re fire without penalty. To do this, we need to go to void OnTouchesEnded and add one line to our previous if statement regarding the ball:

//Create and shoot ball

if (active)

{

ballSprite.RemoveFromParent(true);

//Resets score

score = 0;

}

Once this has been added our project is done! Congratulations for making it until the end, and I hope you are inspired to create some games on your own time!