DEVELOPMENT TUTORIAL ON PIXELSENSE

SUMMARY

WHAT IS PIXELSENSE?

TECHNOLOGY

Microsoft PixelSense is an interactive surface computing platform that allows one or more people to use touch and real world objects, and share digital content at the same time. The PixelSense platform consists of software and hardware products that combine vision based multi-touch PC hardware, 360-degree multiuser application design, and Windows software to create a natural user interface (NUI).

SAMSUNG SUR40 WITH PIXELSENSE

Software development kit (SDK): Microsoft Surface 2.0

Operating system: Windows 7 Professional for Embedded Systems (64-bit)

START PROGRAMMING FOR PIXELSENSE

REQUIRED SOFTWARE

To start a program for PixelSense you have to work on a Windows 7 32-bit (x86), use the C# language and to install the Surface 2.0 SDK.

To use this SDK you need several software:

- Microsoft Visual Studio 2010 (Professional or Express)
- The XNA Framework Redistributable 4.0
- The XNA Game Studio 4.0

The new versions of Microsoft Visual Studio are not supported by the 2.0 SDK, it's better to stay on the 2010 version.

Links to download the software:

- Microsoft Surface SDK 2.0 : http://www.microsoft.com/en-us/download/details.aspx?id=26716
- XNA Game Studio 4.0 : http://www.microsoft.com/en-us/download/details.aspx?id=23714
- XNA Framework Redistributable 4.0 : http://www.microsoft.com/en-us/download/details.aspx?id=20914
- Microsoft Visual Studio 2010 (Professional or Express) : http://msdn.microsoft.com/fr-fr/vstudio/dd582936.aspx

If you want to test your future application a simulator is included in the SDK.

A LITTLE HELLO WORLD

All our codes are located in Github at this address:
https://github.com/D33D33/PixelSense/tree/master/Tuto/Hello

C# example:

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

namespace Hello_World
{
    class Program
    {
        static void Main(string[] args)
        {

           Console.WriteLine("hello world");
        }
    }
}
```

A short program regrouping some bases of C# is located in the GitHub's folder Tuto/Hello.

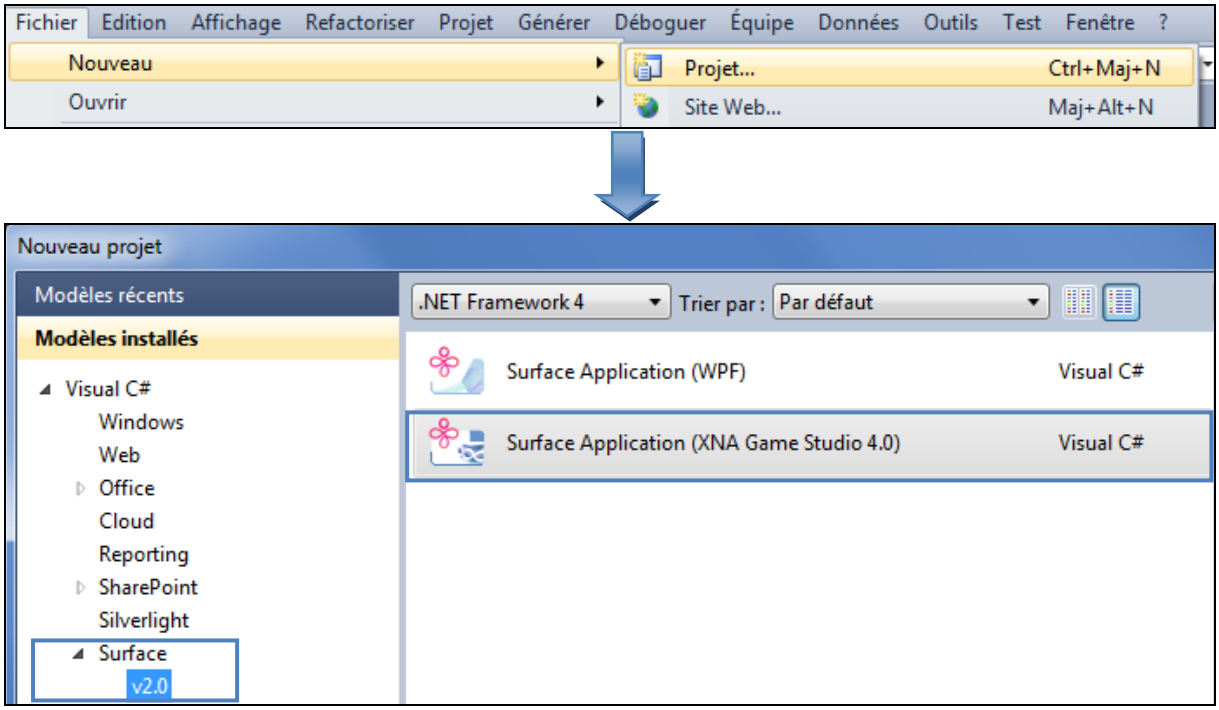
This program includes lists: here a garage, which is a list consisting of several elements of the class car and a test game.

Warning!!

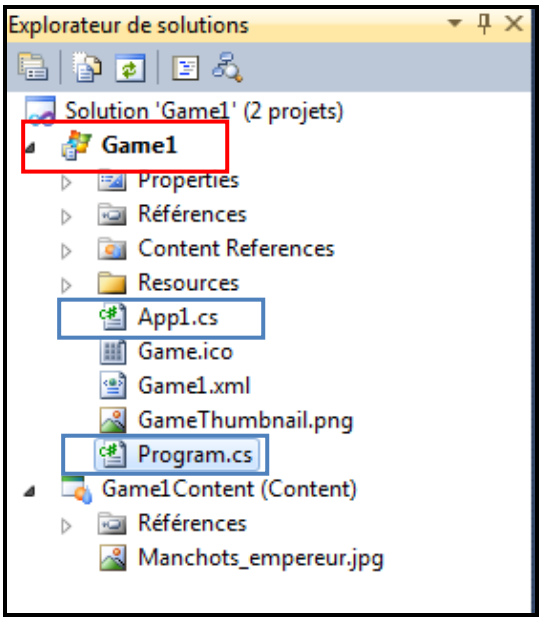
If an assembly reference is missing you have to add it by yourself by right-clicking on your project's Reference Folder and browse the different folders to find the dll.

CREATE A NEW PROJECT FOR PIXELSENSE

To create a project you have to click on the menu File->New->Project and choose Visual C#->Surface->v2.0->Surface Application (here we chose the XNA Game Studio Application because we want to create a little game).



Once we have finished this step Visual Studio will create a project with some code separated in two files. We can find these files in the tree on the right (here App1.cs and Program.cs):



Program.cs contains the basic code of our new window and the main, let's take a look!

In this part we can find all the libraries we'll use in this File.

```
using System;
using System.Drawing;
using System.Windows.Forms;
using Microsoft.Surface;
using Microsoft.Surface.Core;
using Microsoft.Xna.Framework;

namespace Game1
{
    static class Program
    {
        // Hold on to the game window.
        static GameWindow Window;

        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]

static void Main(string[] args)
        {
            // Disable the WinForms unhandled exception dialog.
            // SurfaceShell will notify the user.

Application.SetUnhandledExceptionMode(UnhandledExceptionMode.ThrowException);

            // Apply Surface globalization settings
            GlobalizationSettings.ApplyToCurrentThread();

            using (App1 app = new App1())
            {
                app.Run();
            }
        }....
```

Here is our Main! It's the first executed when we launch the game.

The void Main() instantiates Game1 class with a "using" clause (to ensure the complete release of resources at the end of its use). Finally, once Game1 is instantiated, it launches the Run() method of the latter, which will trigger the process that we have seen previously (InitializeLoadContent, Update ...).

In this file are the definitions of three functions we can use : PositionWindow() (will position the window and size it), SetWindowStyle() and SetWindowSize().

Also, we can find three functions: Size WindowSize (which gets the size of the main window), InitializeWindow (GameWindow window) (which position and decorate our beautiful window) and OnFormLocationChanged (object sender, EventArgs e) (to Respond to changes in the form location, and adjust if necessary).

Now, let's analyze the content of our second file: Game1.cs.

First, there are many "using" clauses that integrate the use of XNA and the namespace in which we work.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;
using Microsoft.Surface;
using Microsoft.Surface.Core;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Media;

namespace Game1
{

...
```

```
public class App1 : Microsoft.Xna.Framework.Game
    {

private readonly GraphicsDeviceManager graphics;
private SpriteBatch spriteBatch;

private TouchTarget touchTarget;
private Color backgroundColor = new Color(81, 81, 81);
private bool applicationLoadCompleteSignalled;

private UserOrientation currentOrientation = UserOrientation.Bottom;
private Matrix screenTransform = Matrix.Identity;
```

Our class App1 inherits from the XNA Game

Here are the attributes generated

In these attributes we can find a GraphicsDeviceManager and a SpriteBatch.

- The GraphicsDeviceManager: This item will make the interface between us and our graphics card. You will be able, for example, to set the size of the window in which draw the game;
- The SpriteBatch: This item will simply serve to draw pictures or strings on your game screen.

```
public App1()
        {
            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
        }
```

The constructor instantiates the GraphicsDeviceManager

The Content Directory is the directory in which we search for any resources we want to load in our game!

Here is the list of methods, used in the game loop, which can be found in the file:

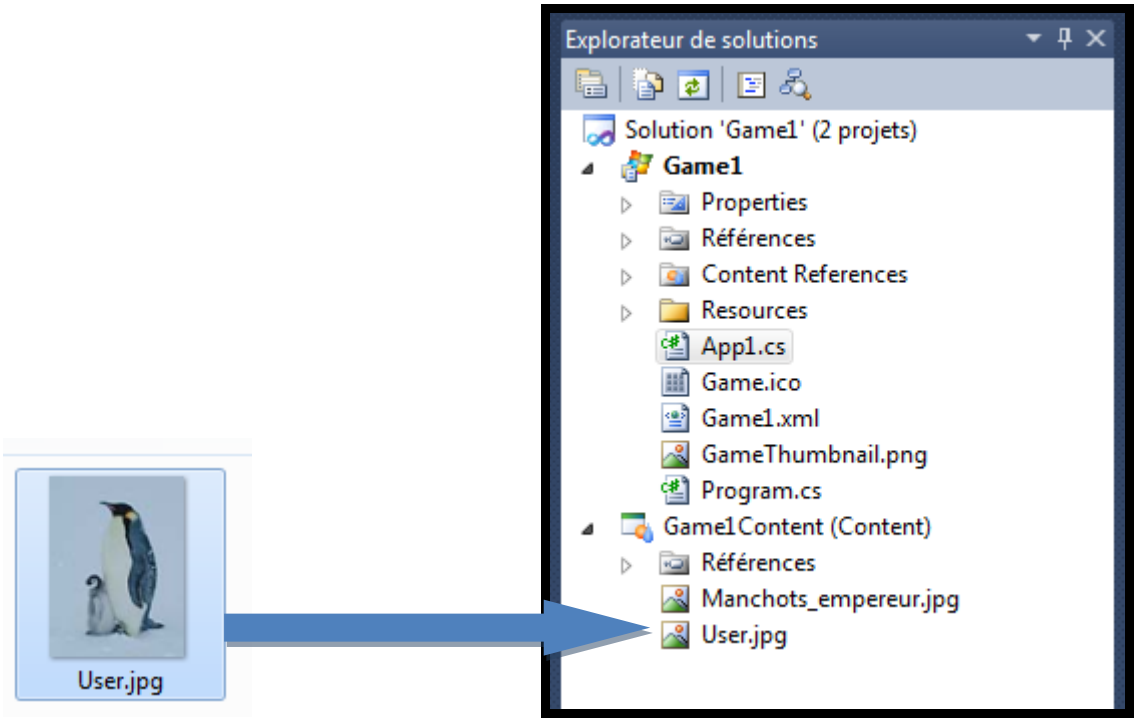
- Protected override void Initialize (): First Method called when the Run method of the Game class is executed (). It allows implement any logic out of our game by initializing variables to the desired values.
- Protected override void LoadContent (): Second method of Run () called. It allows us to load the resources we need to launch our game. This ranges from simple small picture that represents a number to a large 3D model through the sounds and music. Here SpriteBatch is instantiated because it is a pretty important resource that will draw pictures.
- Protected override void UnLoadContent (): Method called when closing the game. It allows us to unload all the content that would not loaded via the Content Manager offered by XNA and remove any memory variable which can’t be handled by the Garbage Collector.
- Protected override void Update (): A part of the two methods of the game loop. This is used to update the logic. The logic game includes for example all that is the player's position, position monsters, calculation of collisions, etc ...
- Protected override void Draw (): This is the second method of the game loop. Unlike Update, this method does not handle the game logic but draw the result of the logical calculated by Update. So she will be in charge of "communicating" with the graphics card to give him things to draw both 2D and 3D. By default in this method, only one action is performed: cleaning your screen. The Clear () method takes a colour parameter (the pretty blue we see at the launch of the game).

USING PICTURES IN OUR PROJECT 1/2: IMPORT

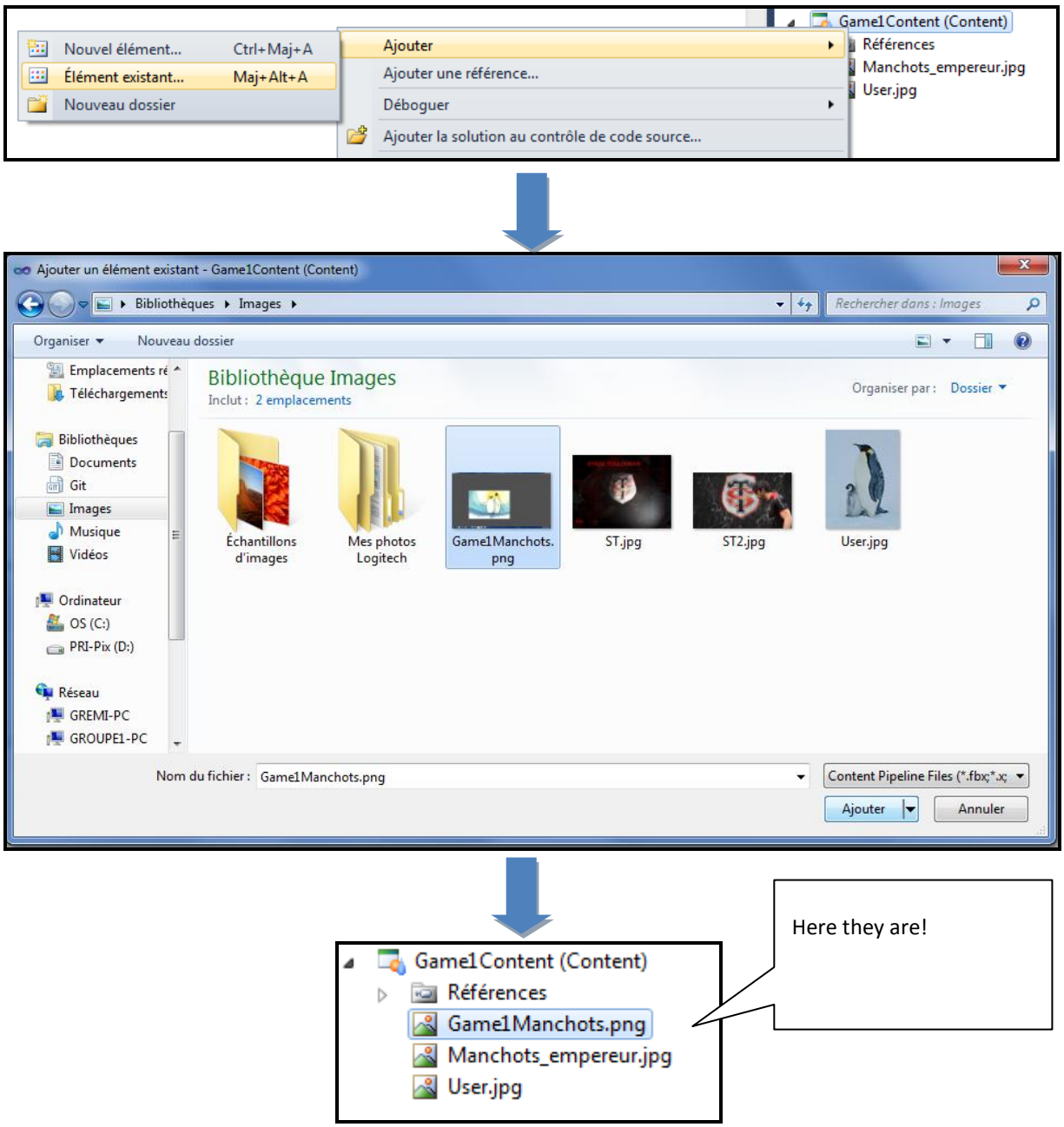
The import process will allow you to simply add your resource file in your project content. The file formats that are taken into account by default in XNA are:

- Pictures : .bmp, .jpg, .png, .dds and .tga ;
- Audio : .mp3, .wav, .wma and .xap ;
- Video : .wmv ;
- 3D models : .fbx and .x ;
- Effects: .fx.

There are two simple methods available to import pictures in our project. The first is really simple, we just have to do a drag and drop from the folder which contains the picture we want to import.



The Second method is as easy as the first, we just need to do a right-click on the Game1Content (Content) of the tree, and click on Add/Existing item and browse to select the desired file.



(We have to give these pictures with the executable)

USING PICTURES IN OUR PROJECT 2/2: DISPLAY

A picture is represented in the code by a Texture2D object types. So we have to create an attribute to our class App1:

```
private Texture2D _manchots;
```

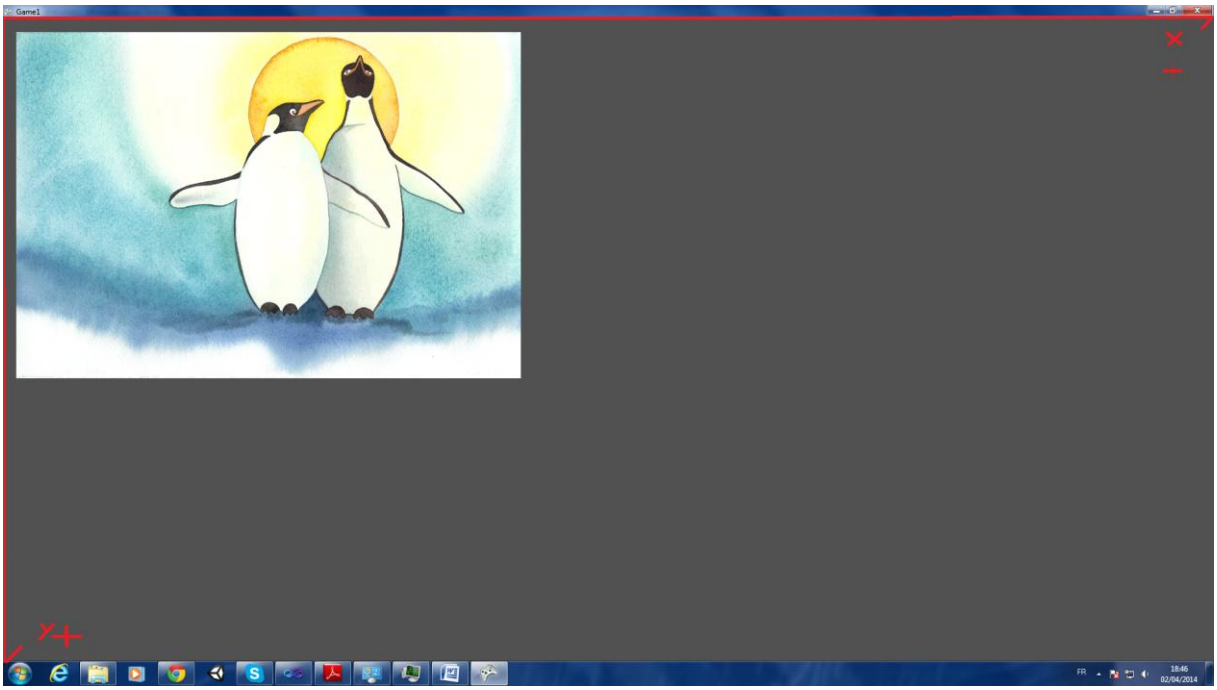
Then we have to load our picture in this attribute using the LoadContent method:

```
protected override void LoadContent()
        {
            // Create a new SpriteBatch, which can be used to draw textures.
            spriteBatch = new SpriteBatch(GraphicsDevice);
            _manchots = Content.Load<Texture2D>("Manchots_empereur");
            // TODO: use this.Content to load your application content here
        }
```

Now our picture is loaded, we have to draw it. To do so we have to use the XNA SpriteBatch item. This one can draw pictures with common properties. We have to signal to our app that we will draw. So in the Draw () method we have to add: spriteBatch.Begin ();

Then we draw the picture: spriteBatch.Draw(_manchots, Vector2.Zero, Color.White), and end the SpriteBatch by: spriteBatch.End();

Vector2.Zero represents the position of the picture in the window (Top-Left Corner).



If we want to have our picture centered in the window we have to go down on Y axis by increasing its value.

The use of Vector2:

- **Vector2.Zero** : corresponds to the coordinates (0, 0) in the top-left corner of our window ;
- **Vector2.UnitX** : corresponds to the coordinates (1, 0) ;
- **Vector2.UnitY** : corresponds to the coordinates (0, 1)
- **Vector2.One**: corresponds to the coordinates (1, 1).
- **Others**: Vector2 vector2 = **new** Vector2(2, 3); Set the coordinates to (2,3).

Now we want to move our picture dynamically, in our main class App1 we will add these attributes:

```
// La position des manchots
private Vector2 _manchotsPosition;
// Le déplacement des manchots
private Vector2 _manchotsDisplacement;
```

And we initialize them in the Initialize () method:

```
protected override void Initialize()
    {
        // TODO: Add your initialization logic here
        _manchotsPosition = Vector2.Zero;
        _manchotsDisplacement = Vector2.One;
```

To reposition _machots according to its displacement we just have to put this code in the Update () method:

```
_manchotsPosition += _manchotsDisplacement;
```

The last thing to do now is to modify our Draw () method, we have to change the spriteBatch.Draw(_manchots, Vector2.Zero, Color.White) in spriteBatch.Draw(_manchots, _manchotsPosition, Color.White).

Now our picture is moving on the screen, but goes out and disappears.

To fix this problem we have to modify Update(). We'll have to reverse either X or Y movement if an edge is touched and if your picture is heading in the wrong direction. This can be done directly by modifying _ `manchotsDisplacement`.X and _ `manchotsDisplacement`.Y.

We have to remember that the image position is relative to the upper left edge, if we forget to take our picture could disappear and we don't want that.

How to do it:

First we'll create two attributes: _screenWidth and _screenHeight :

```
        private Texture2D _manchots;
         // La position des manchots
         private Vector2 _manchotsPosition;
         // Le déplacement des manchots
         private Vector2 _manchotsDisplacement;
         //attributes to get Height and width of the screen
         private int _screenWidth;
         private int _screenHeight;
```

Then in the void Initialize () we will initialize them to get dimensions of the screen:

```
protected override void Initialize()
        {
            // TODO: Add your initialization logic here
            _manchotsPosition = Vector2.Zero;
            _manchotsDisplacement = Vector2.One;

            IsMouseVisible = true; // easier for debugging not to "lose" mouse
            SetWindowOnSurface();
            InitializeSurfaceInput();
            //Initialize the window size
            _screenWidth=Program.WindowSize.Width;
            _screenHeight = Program.WindowSize.Height;
              ……
```

And in Update() we'll test if our manchots goes out of the screen, and replace it :

```
protected override void Update(GameTime gameTime)
        {
            …… // we move manchots by adding it the displacement
            _manchotsPosition += _manchotsDisplacement;
//here we test whether Zozor moves to the left, if this is the case
/ / We check that it does not come out of the screen to the left. Same thing for the
right.
            if ((_manchotsDisplacement.X < 0 && _manchotsPosition.X <= 0)
             || (_manchotsDisplacement.X > 0 && _manchotsPosition.X + _manchots.Width
>= _screenWidth))
            {
                // if we are both in case right and left, we have to reverse
displacement on the X axis
               _manchotsDisplacement.X = - _manchotsDisplacement.X;
            }

            // Same thing for UP/DOWN
            if ((_manchotsDisplacement.Y < 0 && _manchotsPosition.Y <= 0)
                 || (_manchotsDisplacement.Y > 0 && _manchotsPosition.Y +
_manchots.Height >= _screenHeight))
            {
                // Reverse displacement on the Y axis
                _manchotsDisplacement.Y = - _manchotsDisplacement.Y;
            }...
```

CREATE A LITTLE GAME

WHICH GAME?

We have done a little program before with our manchots, we'll use this program as base of our new game.

This new game is the "jeu du palet":

"JEU DU PALET" 1/4: APP1'S ATTRIBUTES

First of all, In the App1.cs file, our "palet" will be named "bignou", so we can replace manchots by it. And we need a little sound effect (when the "palet" will hit the edge of the screen).

```
SoundEffect _ballBounceWall;
private Texture2D bignou; // palet
private Vector2 BignouPosition;
private Vector2 BignouDep;

private KeyboardState _keyboardState;
```

And we have to declare the different hitboxes of bignou (and of our future bats).

```
private Rectangle hitboxRaquetteGauche;
private Rectangle hitboxRaquetteDroite;
private Rectangle hitboxBignou;
```

The attributes below are used later to print scores:

```
private SpriteFont _font;
private int scoreA = 0;
private int scoreB = 0;
```

To finish with our attributes, here are the bats:

```
private Bat batLeft;
private Bat batRight;
```

These bats are described in another Class (we have to create it: right-click on project Add/new item/C# Class) that we'll describe below.

"JEU DU PALET" 2/4: BAT CLASS

In our game, the bat will have to hit "bignou" ("bignou" = "palet").

```
/// <summary>
/// Getter and of texture
/// </summary>
public Texture2D Texture
{
    get { return _texture; }
}
private Texture2D _texture;

/// <summary>
/// Getter of size
/// </summary>
public Rectangle Size
{
    get { return _texture.Bounds; }
}
```

This notation gives us the way to generate our getter and setter.

We will use the same method to all the other attributes:

```
public Vector2 Position
    {
        get { return _position; }
        set { _position = value;
        Console.WriteLine(_prevPosition + " - " + _position);
        }
    }
    private Vector2 _position;
    private Vector2 _prevPosition;


    public Vector2 Direction
    {
        get { return _direction; }
        set { _direction = value; }
    }
    private Vector2 _direction;

    /// <summary>
    /// Getter and setter of speed
    /// </summary>
    public Vector2 Speed
    {
        get { return _speed; }
        //set { _speed = value; }
    }
```

The bat's Position is driven by two vectors: the current position: _position, and its previous position: _prevPosition

```
private Vector2 _speed;

        private float _speedMax;
        public float SpeedMax
        {
            get { return _speedMax; }
            set { _speedMax = value; }
        }
```

The _speedMax is the Maximum value of speed we authorize.

Now that we have finished to create our attributes we have to initialize them in the Initialize() function:

```
    /// <summary>
    /// Bat initialisation
    /// </summary>
    public virtual void Initialize()
    {
        _position = Vector2.Zero;
        _prevPosition = Vector2.Zero;
        _direction = Vector2.Zero;
        _speedMax = 5;
        _speed = Vector2.Zero;
    }
```

The following function LoadContent will allow us to load the pictures of our bats:

```
public virtual void LoadContent(ContentManager content, string assetName)
{
    _texture = content.Load<Texture2D>(assetName);
}
```

This function will be called in App1.cs's LoadContent function.

In the same way the App1's draw function will call this draw function of bat.cs:

```
public virtual void Draw(SpriteBatch spriteBatch, GameTime gameTime)
        {
            spriteBatch.Draw(_texture, _position, Color.White);
        }
```

The last Bat's function called by App1 is Update(), in it we update the position, direction and speed vectors.

```
public virtual void Update(GameTime gameTime)
        {
            _direction = _position - _prevPosition;


            if (_direction.Length() > 0)
            {
                _direction.Normalize();
                _speed = _direction * _speedMax
            }
            else
            {
                _speed *= 0;
            }
            _prevPosition = _position;
        }
```

The normalize is here to get a vector's norm = 1 .

Now we have finished with Bat. Let's go back to App1. And take a coffee break.

"JEU DU PALET" 3/4: APP1 METHODS

Now, we have to initialize our attributes in the Initialize() function:

```
        screenWidth = Program.WindowSize.Width;
        screenHeight = Program.WindowSize.Height;

        BignouPosition = Vector2.Zero;
        BignouDep = Vector2.Zero;

        batLeft = new Bat();
        batLeft.Initialize();
        batRight = new Bat();
        batRight.Initialize();
        batRight.Position = new Vector2(screenWidth - 140, screenHeight - 455);
```

And to load the different pictures:

```
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 application content here
            bignou = Content.Load<Texture2D>("paletGame");

            BignouPosition.X = (screenWidth - bignou.Width) / 2;
            BignouPosition.Y = (screenHeight - bignou.Height) / 2;

            batLeft.LoadContent(Content, "raquette1");
            batRight.LoadContent(Content, "raquette2");
            _ballBounceWall = Content.Load<SoundEffect>("bahhhhh");

            _font = Content.Load<SpriteFont>("MaPolice");
        }
```

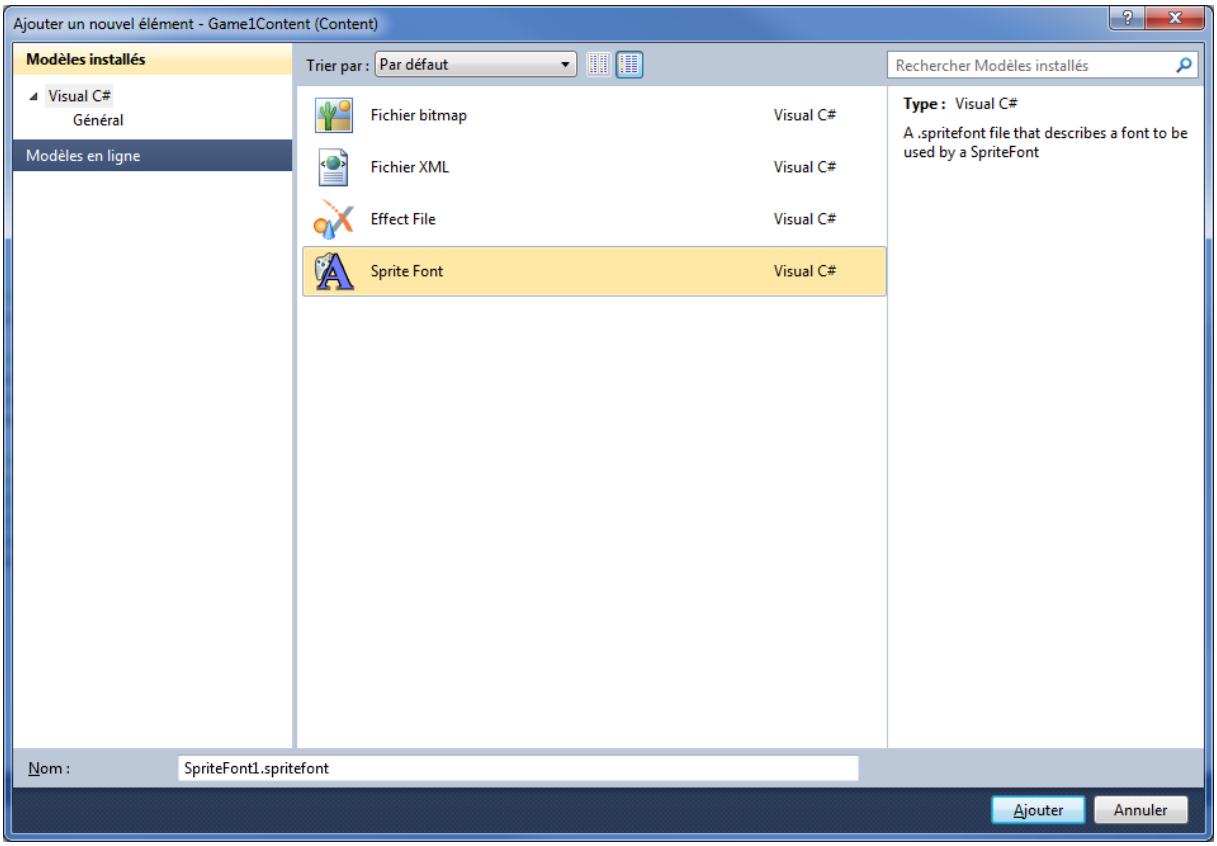
Here the sound effect « bahhhhh » is an mp3 file reference (see using picture part)

This line add the font in memory

Here we can see that we add a SpriteFont file because we want to display the score in the game.

XNA provides a simple way of writing the text where we want in our game. In order to write strings on the game window, XNA provides us SpriteFont class. An object of this Type in our code will represent the style that will be used to write. This defines the font to use (Arial, Verdana ...), the pixel size of the letters or the spacing between two characters.

To add a SpriteFont file, just right-click on the content folder, click on Add/new item and a window will appear. We just have to choose Sprite Font:



The SpriteFont file end in .spritefont but is an xml file so there are tags that we will modify:

- **FontName**: The name of the font we want to use, it must be installed on the computer that compiles the project but not necessarily on the computer that runs the game.
- **Size**: Size of the letters.
- **Style**: Allows you to specify if you want your text bold, italic or regular.
- **Spacing**: defines the space between two characters, it's a float.
- **CharacterRegion**: Allows, using its <start> <end> tags, to determine what range of characters must be load for SpriteFont. By default <start> tag is "** **" and the tag <end> to "**~**" representing the ASCII characters between the "space" and the tilde ('~'). This range does not include accents to use them we must change the value of <end> and give for example the value "**Ā**" which includes all characters ASCII (ASCII extended Table: http://www.table-ascii.com/).

This is our “MaPolice” file used to display score in the game:

```
<?xml version="1.0" encoding="utf-8"?>
<!--
This file contains an xml description of a font, and will be read
by the XNA
Framework Content Pipeline. Follow the comments to customize the
appearance
of the font in your game, and to change the characters which are
available to draw
with.
-->
<XnaContent
xmlns:Graphics="Microsoft.Xna.Framework.Content.Pipeline.Graphics">
<Asset Type="Graphics:FontDescription">
<!--
Modify this string to change the font that will be imported.
-->
<FontName>Impact</FontName>
<!--
Size is a float value, measured in points. Modify this value to
change
the size of the font.
-->
<Size>72</Size>
<!--
Spacing is a float value, measured in pixels. Modify this value to
change
the amount of spacing in between characters.
-->
<Spacing>0</Spacing>
<!--
UseKerning controls the layout of the font. If this value is true,
kerning information
will be used when placing characters.
-->
<UseKerning>true</UseKerning>
<!--
Style controls the style of the font. Valid entries are "Regular",
"Bold", "Italic",
and "Bold, Italic", and are case sensitive.
-->
<Style>Bold</Style>
<!--
If you uncomment this line, the default character will be
substituted if you draw
or measure text that contains characters which were not included in
the font.
-->
<!-- <DefaultCharacter>*</DefaultCharacter> -->
<!--
CharacterRegions control what letters are available in the font.
Every
character from Start to End will be built and made available for
drawing. The
default range is from 32, (ASCII space), to 126, ('~'), covering
the basic Latin
character set. The characters are ordered according to the Unicode
standard.
See the documentation for more information.
-->
```

```
<CharacterRegions>
<CharacterRegion>
<Start>&#32;</Start>
<End>&#256;</End>
</CharacterRegion>
</CharacterRegions>
</Asset>
</XnaContent>
```

The Font is used to draw in the draw() function.Below, the scoreA and scoreB are incremented in the update function with the game logic(hitbox/collision…).

```
protected override void Draw(GameTime gameTime)
        {
…

             spriteBatch.Begin();
             batLeft.Draw(spriteBatch, gameTime);
             batRight.Draw(spriteBatch, gameTime);

              spriteBatch.Draw(bignou, BignouPosition,
       Microsoft.Xna.Framework.Color.White);

              spriteBatch.Draw(batRight.Texture,hitboxRaquetteDroite,Microsoft.Xna.Fram
              ework.Color.Orange);
               Vector2 textSize = _font.MeasureString(scoreA + " - " + scoreB );
               spriteBatch.DrawString(_font, scoreA + " - " + scoreB, new
Vector2((screenWidth - textSize.X) / 2, 20), Microsoft.Xna.Framework.Color.White);
               spriteBatch.End();


            base.Draw(gameTime);
        }
```

The game Logic is very important for our game, it’s set in the Update() function and contains all the collision logic of the “palet” and the bats.

"JEU DU PALET" 4/4: GAME LOGIC

All the game logic is located in the Update() function. Let's take a look step by step:

```
protected override void Update(GameTime gameTime)
        {
            batLeft.Update(gameTime);
            batRight.Update(gameTime);
            // Hitbox des raquettes
            hitboxRaquetteGauche = new Rectangle((int)batLeft.Position.X,
(int)batLeft.Position.Y, batLeft.Size.Width, batLeft.Size.Height);
            hitboxRaquetteDroite = new Rectangle((int)batRight.Position.X,
(int)batRight.Position.Y, batRight.Size.Width, batRight.Size.Height);
            hitboxBignou = new
Rectangle((int)BignouPosition.X,(int)BignouPosition.Y,bignou.Width,bignou.Height);
……
```

Here we call the bat's update() function that update the bat's vector. The hitbox above are rectangles and are here to manage the collision calculations we need to do after.

```
 BignouPosition += BignouDep;

// logique du bignou : collision
if((BignouDep.X < 0 && BignouPosition.X <= 0) ||
 (BignouDep.X > 0 && BignouPosition.X + bignou.Width >= screenWidth)){
_ballBounceWall.Play();

if ((BignouDep.X < 0 && BignouPosition.X <= 0))
{
     scoreB++;
}


else
{
     scoreA++;
}


BignouPosition.X = (screenWidth - bignou.Width) / 2;
BignouPosition.Y = (screenHeight - bignou.Height) / 2;
BignouDep *= 0;
}

else if((BignouDep.Y < 0 && BignouPosition.Y <= 0) ||
(BignouDep.Y > 0 && BignouPosition.Y + bignou.Height >= screenHeight))
{
      BignouDep.Y = -BignouDep.Y;
      _ballBounceWall.Play();
}
```

« Palet » displacement as "_manchots" before.

ballBounceWall play the sound effect

Here we manage the players' score, if the position. X is negative or > screenWidth the score is incremented (there are goals when the "palet" hit the left or right border of the screen)

After goal we put the "palet" on the center of the screen.

If the "palet" hit the up/down border it bounces.

When the bat hit the"palet"'s hitboxes, the "palet" will go in the opposite direction, when we push the "palet" with the bat, the bat will stick the "palet" until it gain speed. Float Y and X below get the direction of the "palet". Here is the code of our collision logic (the same method is used for both right and left bat):

```
/* collision Logic left bat:   */

                float Y = Math.Min(Math.Min(Math.Max(BignouPosition.Y + bignou.Height
- batLeft.Position.Y, 0), Math.Max(batLeft.Position.Y + batLeft.Size.Height -
BignouPosition.Y, 0)), bignou.Height);
                float X = Math.Min(Math.Min(Math.Max(BignouPosition.X + bignou.Width -
batLeft.Position.X, 0), Math.Max(batLeft.Position.X + batLeft.Size.Width -
BignouPosition.X, 0)), bignou.Width);

                //Console.WriteLine(X + " - " + Y);
                if (hitboxRaquetteGauche.Contains((int)BignouPosition.X,
(int)BignouPosition.Y) ||
                    hitboxRaquetteGauche.Contains((int)BignouPosition.X,
(int)BignouPosition.Y + bignou.Height))
                {
                    if (X < Y)
                    {
                        if (BignouDep.X <= 0)
                        {
                            BignouDep.X = Math.Max(-BignouDep.X, batLeft.Speed.X);
                            BignouDep.Y += batLeft.Speed.Y * (float)0.5;
                            _ballBounceWall.Play();
                        }
                        else if (batLeft.Direction.X > 0)
                        {
                            BignouPosition.X += batLeft.Direction.X;
                        }
                    }
                }
                else if (   hitboxRaquetteGauche.Contains((int)BignouPosition.X +
bignou.Width, (int)BignouPosition.Y) ||
                            hitboxRaquetteGauche.Contains((int)BignouPosition.X +
bignou.Width, (int)BignouPosition.Y + bignou.Height))
                {
                    if (X < Y)
                    {
                        if (BignouDep.X >= 0)
                        {
                            BignouDep.X = Math.Min(-BignouDep.X, batLeft.Speed.X);
                            BignouDep.Y += batLeft.Speed.Y * (float)0.5;
                            _ballBounceWall.Play();
                        }
                        else if (batLeft.Direction.X < 0)
                        {
                            BignouPosition.X += batLeft.Direction.X;
                        }
                    }
                }
                if (hitboxBignou.Contains((int)batLeft.Position.X,
(int)batLeft.Position.Y) ||
                        hitboxBignou.Contains((int)batLeft.Position.X +
batLeft.Size.Width, (int)batLeft.Position.Y))
                {
                    if (X > Y)
```

```
                    {
                        if (BignouDep.Y >= 0)
                        {
                            BignouDep.X += batLeft.Speed.X * (float)0.5;
                            BignouDep.Y = Math.Min(-BignouDep.Y, batLeft.Speed.Y);
                            _ballBounceWall.Play();
                        }
                        else if (batLeft.Direction.Y < 0)
                        {
                            BignouPosition.Y += batLeft.Direction.Y;
                        }
                    }
                }
                else if (hitboxBignou.Contains((int)batLeft.Position.X +
batLeft.Size.Width, (int)batLeft.Position.Y + batLeft.Size.Height) ||
                            hitboxBignou.Contains((int)batLeft.Position.X,
(int)batLeft.Position.Y + batLeft.Size.Height))
                {
                    if (X > Y)
                    {
                        if (BignouDep.Y <= 0)
                        {
                            BignouDep.X += batLeft.Speed.X * (float)0.5;
                            BignouDep.Y = Math.Max(-BignouDep.Y, batLeft.Speed.Y);
                            _ballBounceWall.Play();
                        }
                        else if (batLeft.Direction.Y > 0)
                        {
                            BignouPosition.Y += batLeft.Direction.Y;
                        }
                    }
                }
```

```
/* collision Logic right bat:   */

float Y2 = Math.Min(Math.Min(Math.Max(BignouPosition.Y + bignou.Height -
batRight.Position.Y, 0), Math.Max(batRight.Position.Y + batRight.Size.Height -
BignouPosition.Y, 0)), bignou.Height);
float X2 = Math.Min(Math.Min(Math.Max(BignouPosition.X + bignou.Width -
batRight.Position.X, 0), Math.Max(batRight.Position.X + batRight.Size.Width -
BignouPosition.X, 0)), bignou.Width);

                if (hitboxRaquetteDroite.Contains((int)BignouPosition.X +
bignou.Width, (int)BignouPosition.Y) ||
                    hitboxRaquetteDroite.Contains((int)BignouPosition.X +
bignou.Width, (int)BignouPosition.Y + bignou.Height))
                {
                    if (X2 < Y2)
                    {
                        if (BignouDep.X >= 0)
                        {
                            Console.WriteLine(batRight.Speed.X);
                            BignouDep.X = Math.Min(-BignouDep.X, batRight.Speed.X);
                            BignouDep.Y += batRight.Speed.Y * (float)0.5;
                            _ballBounceWall.Play();
                        }
                        else if (batRight.Direction.X < 0)
                        {
                            BignouPosition.X += batRight.Direction.X;
// = raquetteDroitePosition.X - bignou.Width;
                        }
                    }
                }
                else if(hitboxRaquetteDroite.Contains((int)BignouPosition.X,
(int)BignouPosition.Y) ||
                        hitboxRaquetteDroite.Contains((int)BignouPosition.X,
(int)BignouPosition.Y + bignou.Height))
                {
                    if (X2 < Y2)
                    {
                        if (BignouDep.X <= 0)
                        {
                            BignouDep.X = Math.Max(-BignouDep.X, batRight.Speed.X);
                            BignouDep.Y += batRight.Speed.Y * (float)0.5;
                            _ballBounceWall.Play();
                        }
                        else if (batRight.Direction.X > 0)
                        {
                            BignouPosition.X += batRight.Direction.X;
//  = raquetteDroitePosition.X + raquetteDroite.Width;
                        }
                    }
                }

                if (hitboxBignou.Contains((int)batRight.Position.X,
(int)batRight.Position.Y) ||
                     hitboxBignou.Contains((int)batRight.Position.X +
batRight.Size.Width, (int)batRight.Position.Y))
                {
                    if (X2 > Y2)
```

```
                    {
                        if (BignouDep.Y >= 0)
                        {
                            BignouDep.X += batRight.Speed.X * (float)0.5;
                            BignouDep.Y = Math.Min(-BignouDep.Y, batRight.Speed.Y);
                            _ballBounceWall.Play();
                        }
                        else if (batRight.Direction.Y < 0)
                        {
                            BignouPosition.Y += batRight.Direction.Y;
//  = raquetteDroitePosition.Y - bignou.Height;
                        }
                    }
                }
                else if (hitboxBignou.Contains((int)batRight.Position.X +
batRight.Size.Width, (int)batRight.Position.Y + batRight.Size.Height) ||
                        hitboxBignou.Contains((int)batRight.Position.X,
(int)batRight.Position.Y + batRight.Size.Height))
                {
                    if (X2 > Y2)
                    {
                        if (BignouDep.Y <= 0)
                        {
                            BignouDep.X += batRight.Speed.X * (float)0.5;
                            BignouDep.Y = Math.Max(-BignouDep.Y, batRight.Speed.Y);
                            _ballBounceWall.Play();
                        }
                        else if (batRight.Position.Y > 0)
                        {
                            BignouPosition.Y += batRight.Position.Y;//
raquetteDroitePosition.Y + raquetteDroite.Height;
                        }
                    }
                }
```

In order to prevent the “palet” from going too fast we have added a speed reduction factor to decrease its speed:

```
/**
 * Bignou slowing
 */
double speedReductionFactor = 0.999;
BignouDep *= (float)speedReductionFactor;
```

In a first time we will use the keyboard to manage bats' movement: below is the code to manage a bat with the direction keys, the same method can be used for the other bat.

```
                _keyboardState = Keyboard.GetState();
                if (_keyboardState.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.Up)){
                    if (batRight.Position.Y >= 0)
                    {
                        Vector2 tmp = batRight.Position;
                        tmp.Y -= batRight.SpeedMax;
                        batRight.Position = tmp;
                    }
                    else
                    {
                     }
                }
                else if
(_keyboardState.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.Down))
                {
                    if (batRight.Position.Y <= screenHeight - batRight.Size.Height)
                    {
                        Vector2 tmp = batRight.Position;
                        tmp.Y += batRight.SpeedMax;
                        batRight.Position = tmp;
                    }
                    else
                    {
                    }
                }

                if
(_keyboardState.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.Right))
                {
                    if (batRight.Position.X < screenWidth - batRight.Size.Width)
                    {
                        Vector2 tmp = batRight.Position;
                        tmp.X += batRight.SpeedMax;
                        batRight.Position = tmp;
                    }
                    else{
                    }
                }
                else if
(_keyboardState.IsKeyDown(Microsoft.Xna.Framework.Input.Keys.Left))
                {
                    if (batRight.Position.X > ((screenWidth / 2) +
batRight.Size.Width) * 0.9)
                    {
                        Vector2 tmp = batRight.Position;
                        tmp.X -= batRight.SpeedMax;
                        batRight.Position = tmp;
                    }
                    else
                    {
                      }
            }
```

TACTILE EVENT

For now, our game is keyboard dependant on a tactile device… Let's make it tactile!

TACTILE "JEU DU PALET": BAT CLASS RETURNS

To make our bats tactile we have to add these attributes:

```
private int _touchId = 0;
private Vector2 _delta = new Vector2(0, 0);
/// <summary>
/// Getter and setter of touchTarget
/// </summary>
public TouchTarget TouchTarget
{
    get { return _touchTarget; }
    set { _touchTarget = value; }
}
private TouchTarget _touchTarget;
```

We create a TouchTarget which contains the state of the screen (coordinates of the touch point, screen touched or not…)

We'll now have to initialize the TouchTarget in the Initialize() function:

```
public virtual void Initialize( TouchTarget touchTarget)
  {
       …
      _touchTarget = touchTarget;

      _touchTarget.TouchDown += new
EventHandler<TouchEventArgs>(this.TouchedDown);
      _touchTarget.TouchMove += new
EventHandler<TouchEventArgs>(this.TouchedMove);
      _touchTarget.TouchUp += new EventHandler<TouchEventArgs>(this.TouchedUp);
  }
```

TouchTarget of the game

Actions linked to tactile table's interactions

```
public void TouchedUp(object sender, EventArgs e)
{
      TouchEventArgs args = (TouchEventArgs)e;
      TouchPoint touch = args.TouchPoint;

      if (_touchId == touch.Id)
      {
      Position = new Vector2(touch.CenterX - _delta.X, touch.CenterY - _delta.Y);
       }

       if (_touchId == touch.Id)
        {
        _touchId = 0;
        }
}
```

Verify if we release the touch on this bat and make the move corresponding.

```
public void TouchedDown(object sender, EventArgs e)
{
      TouchEventArgs args = (TouchEventArgs)e;
      TouchPoint touch = args.TouchPoint;

      float x = touch.CenterX;
      float y = touch.CenterY;
      if (x > Position.X && x < Position.X + Size.Width &&
          y > Position.Y && y < Position.Y + Size.Height)
      {
           _touchId = touch.Id;
           _delta = new Vector2(x - Position.X, y - Position.Y);
      }
}
```

Verify if we are touching any points of the object.

```
public void TouchedMove(object sender, EventArgs e)
{
      TouchEventArgs args = (TouchEventArgs)e;
      TouchPoint touch = args.TouchPoint;

      if (_touchId == touch.Id)
      {
      Position = new Vector2(touch.CenterX - _delta.X, touch.CenterY - _delta.Y);
       }
}
```

Specify that e is a TouchEvent (specification of event)

TACTILE “JEU DU PALET”: APP1 CLASS RETURNS

```
batLeft = new Bat();
batLeft.Initialize(touchTarget);
batRight = new Bat();
batRight.Initialize(touchTarget);
```

Now our bats need a touchTarget to be initialized (see App1 method part to find the first initialization of the bats).

Our game is now a tactile game!

CONCLUSION

Now, you got strong assets and good tools to work and create functional applications on Microsoft surface with XNA framework.

In this tutorial, we only get into basics such as displaying pictures, creating fonts and tactile interactions. You have to know surface can perform so many more actions like reading tags, detecting fingers or objects… It is also possible to develop using WPF and we did not get into it.

We hope this tutorial was useful to you and seduced you to program with XNA framework.