Técnicas de Desenvolvimento de Jogos

FrameWork MonoGame

Engenharia Desenvolvimento de Jogos Digitais

Instituto Politécnico do Cavado e Ave

Trabalho realizado por:

- Carlos Alves a13219
- Bruno Rodrigues a13220

IPCA | INSTITUTO POLITÉCNICO DO CÁVADO E DO AVE
ESCOLA SUPERIOR DE TECNOLOGIA

Índice

Introdução

No âmbito da disciplina Técnicas de Desenvolvimento de Jogos, foi nos proposto realizar um relatório de um exercício realizado durante as aulas, que consiste na utilização do Visual Studio e do MonoGame (Framework) para a criação de um jogo.

Com este trabalho de grupo é pretendido melhor as nossas capacidades de programar em C-Sharp, como também a inicialização à criação de jogos através de MonoGame e do VisualStudio.

Sucintamente, o MonoGame é uma implementação open-source do Microsoft XNA 4 Framework. Permite que desenvolvedores de jogos portem seus jogos de Windows, Windows Phone e Xbox 360 para iOS, Android, Mac OS X, Linux e Windows 8 Apps.

Neste relatório iremos descrever todos os exercícios feitos, as variáveis, os métodos/funções, as classes e objetos, a hierarquia e a ordem entre cada um dos elementos, assim como a visibilidade. Também será descrito todas as dificuldades encontradas no decorrer dos exercícios, como a sua resolução e melhoramento.

Desenvolvimento

Inicialmente, começamos por conhecer um pouco a ***Framework MonoGame***, como também a sua instalação de forma que fosse compatível com a versão do ***VisualStudio*** já instalado.

Após a instalação e a verificação das versões entre a versão da Framework ***MonoGame*** e a versão do ***VisualStudio*** de maneira que fossem compatíveis, achamos melhor conhecermos e informarmo-nos melhores sobre o que cada método/função do Game1 fazia.

Métodos:

- **The Initialize()** – Esta função inicializa os gráficos necessários para o jogo funcionar, como o tamanho da janela.

- **LoadContent()** – Esta função carrega todo o conteúdo ou os recurso do projeto, como a arte do jogo e o áudio. Função é chamada apenas no início do jogo, sendo importante referir que apenas é chamada uma vez por execução.

- **UnloadContent()** – Esta função é parecida com a função acima referida e descrita, pois ela apenas é chamada no fim do jogo para descarregar todo o conteúdo que o método *LoadContent*() carregou no inicio do jogo.

- **Update()** – Esta função comporta-se de maneira diferente das duas funções descritas em cima, pois esta é chamada muitas vezes por segundo durante a execução do jogo. O método *Update*() processa qualquer alteração no estado do jogo e também é responsável por procurar os dispositivos de entrada.

- **Draw()** – A função *Draw*() também é chamada muitas vezes por segundo durante a execução do jogo, tal como o método anterior. Ela tem como função desenhar o "mundo" do jogo para a tela cada vez que é chamada, também limpa a tela a cada ciclo e redesenha todos os objetos compilados para o jogo.

É importante salientar que a função ***Update*()** e a ***Draw*()** tem as suas semelhanças, mas acabam por ser bastantes distintas e definidas. Isto, porque o método ***Update*()** deve apenas atualizar o estado do jogo(alterando o valor das variáveis) e nunca desenha nada, enquanto o método ***Draw***() exibe apenas o estado do jogo e nunca altera o estado do jogo.

Na resolução de um pequeno exercício, onde envolvia essas duas funções podemos verificar que por caso os dois métodos se juntem causaram imensos conflitos no jogo, como na própria classe. Este problema foi encontrado e solucionado através de tentativa e erro.

DrawAndControl Projeto

Depois de reunida a informação do que cada uma das funções fazia, seguimos para a criação de um novo projeto intitulado ***DrawAndControl,*** onde foi necessário adicionar duas imagens ao projeto. Este exercício tinha como objetivo demonstrar como carregar imagens para um projeto ***MonoGame*** e também como manipular as suas posições.

Primeiramente, declaramos as variáveis:

- *mJPGImage, mJPGPosition* – relativas à imagem JPG
- *mPNGImage, mPNGPosition* – relativas à imagem em PNG

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

        // Support for loading and drawing the JPG image
        Texture2D mJPGImage; // The UWB-JPG.jpg image to be loaded
        Vector2 mJPGPosition; // Top-Left pixel position of UWB-JPG.jpg
                              // Support for loading and drawing of the PNG image
        Texture2D mPNGImage; // The UWB-PNG.png image to be loaded
        Vector2 mPNGPosition; // Top-Left pixel position of UWB-PNG.png
```

Fig1.Variaveis

De seguida, na função ***Initialize*()** definimos a posição inicial das imagens que foram inseridas ao projeto anteriormente e na função ***LoadContent*()** carregamos essas mesmas imagens que se encontravam numa pasta do projeto(Pasta "*Content"*).

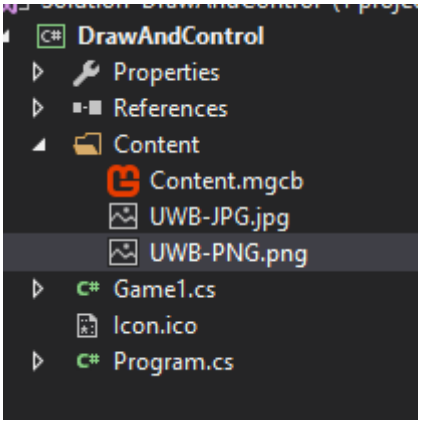


Fig2.Solution Explorer do projeto

Com as imagens carregadas e prontas a ser usadas no jogo, foi necessário desenhá-las no mundo do jogo, para isso usamos o objeto *SpriteBatch*. Com isso, pode-mos inicializar o processo de desenho com **SpriteBatch.Begin()** e adicionamos as imagens ao pacote a ser processado com **SpriteBatch.Draw()**. Depois de as imagens terem sido desenhadas foi necessário usar a função **SpriteBatch.End()** para descarregar e exibir as imagens no jogo.

```
// Limpa a cor de fundo
GraphicsDevice.Clear(Color.CornflowerBlue);
spriteBatch.Begin(); // Inicializa o processo de desenhar
// Desenha JPGImagem
spriteBatch.Draw(mJPGImage, mJPGPosition, Color.White);
// Desenha PNGImagem
spriteBatch.Draw(mPNGImage, mPNGPosition, Color.White);
spriteBatch.End(); // Informa o sistema que os desenhas estam prontos
base.Draw(gameTime);
```

Fig3. Método *Draw*() com as devidas funções comentadas;

Por fim, o último passo é atualizar as posições das imagens com base na entrada do utilizador usando o teclado para alterar as posições das imagens, e para isso modificamos a função **Update**() que tem como utilidade atualizar o estado do jogo apenas alterando as variáveis instanciadas no início do projeto, de maneira em que o utilizador possa controlar livremente as posições das imagens no jogo.

```
// Segue o jogo até ao fim
if (Keyboard.GetState().IsKeyDown(Keys.Escape))
    this.Exit();
// Carrega as posiçoes das imagens com as setas
if (Keyboard.GetState().IsKeyDown(Keys.Left))
    mJPGPosition.X--;
if (Keyboard.GetState().IsKeyDown(Keys.Right))
    mJPGPosition.X++;
if (Keyboard.GetState().IsKeyDown(Keys.Up))
    mJPGPosition.Y--;
if (Keyboard.GetState().IsKeyDown(Keys.Down))
    mJPGPosition.Y++;
```

Fig4.Função *Update*() – Ciclo para usar as setas como forma de manipular as posições das imagens;

Como mostra a Fig.4, podemos ver os ciclos (*if*) necessários relacionar as setas do teclado, de maneira que o utilizador possa modificar as posições das imagens enquanto joga.

Depois de verificar e solucionar os erros de sintaxe existentes no projeto, como algumas variáveis mal declaradas, podemos "correr" o programa para observar o resultado de todo o trabalho feito em cada uma das funções.

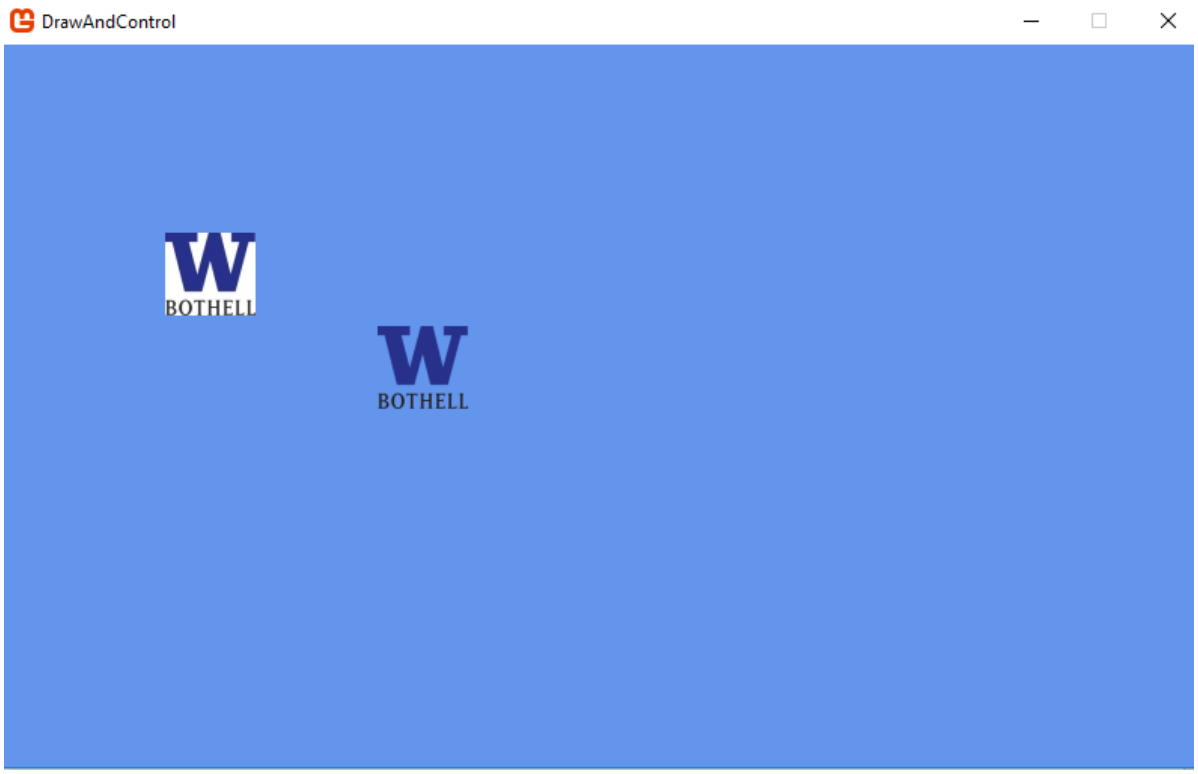


Fig5. DrawAndControl

Input Wrapper Projeto

Feito o primeiro exercício, podemos obter algumas bases para o próximo projeto, sendo assim usamos o projeto *DrawAndControl*, que consistia em carregar duas imagens na tela e configurar algumas teclas do teclado para que o utilizador pudesse modificar livremente as posições das imagens. Com esse projeto realizamos um outro nomeado *InputWrapper,* que tem como utilidade encapsular os dispositivos de entrada do *joystick* e teclado. Desta forma pode ser possível usar qualquer tipo de dispositivo sem ter de existir mais trabalho no código.

Este projeto tem como objetivo:

- Terminar o programa com a tecla F1 (teclado) ou o botão Back(*joystick*);
- Mover a imagem JPG com as teclas WSAD ou com o analógico esquerdo;
- Mover a imagem PNG com as setas do teclado ou com o analógico direito.

Inicialmente, começamos por criar uma nova classe (InputWrapper.cs), adicionamos as bibliotecas necessárias que armazenam imensas funções utilizadas no projeto. Sequencialmente, antes da classe ***InputWrapper*** definimos as estruturas de dados que vão receber os "*inputs*" dos botões do joystick. Nessa estrutura classificamos a estrutura como "internal" pois serve para assegurar que o conteúdo da estrutura feita só é acessível nesta biblioteca, limitando assim o acesso a este tipo de dados. Dentro da estrutura de dados ***AllinputButtons*** definimos as teclas que correspondem aos botões.

Ainda dentro da estrutura de dados ***AllInputButtons*** definimos a função **GetState**() que retorna a tecla ou o botão pressionado. Como se pode observar pelo código, a ordem de leitura é primeiro o teclado e só depois o *joystick*.

```
if (Keyboard.GetState().IsKeyDown(key))
    return ButtonState.Pressed;
  //Joystick
if ((GamePad.GetState(PlayerIndex.One).IsConnected))
    return gamePadButtonState;

return ButtonState.Released;
```

Fig.5 – Função GetState()

Continuando na estrutura de dados ***AllInputButtons*** definimos avaliadores dos *inputs* recebidos correspondentes aos botões que serão usados na função anteriormente referida. Exemplo abaixo de como foram definidos esses avaliadores.

```
///Estado do botao (pressionado ou liberado
public ButtonState A { get { return GetState(GamePad.GetState(PlayerIndex.One).Buttons.A, kA_ButtonKey); } }
```

Fig.6 – Exemplo dos avaliadores usados no projeto;

Após isso, definimos outras duas estruturas ***AllInputTriggers*** *e* ***AllThumbSticks*** para encapsular a ativação do *game controller* e dos analógicos.

Na estrutura ***AllInputTriggers*** implementamos três variáveis que correspondiam aos ativadores (*trigger*) que fariam corresponder às teclas N e M e outra que retorna um valor de 0.75 para garantir a prioridade de verificação em relação ao *joystick*.

Passando para a estrutura de dados ***AllThumbSticks***, é preciso fazer com que os movimentos no eixo dos X do analógico esquerdo serão iguais as teclas A e D do teclado, já os movimentos correspondentes ao eixo dos Y serão iguais as teclas W e S. Os movimentos do analógico direito devem corresponder as setas do teclado.

Para fazer isso, foi necessário usar a função ***ThumbStickState*****(),** pois ela tem como utilidade receber informação do *game controller* e do teclado para retornar os valores dos movimentos realizados.

```
public Vector2 Right
{
    get
    {
        return ThumbStickState(GamePad.GetState(PlayerIndex.One).ThumbSticks.Right,
              kRightThumbStickUp, kRightThumbStickDown, kRightThumbStickLeft, kRightThumbStickRight);
    }
}
```

Fig.7 – Exemplo função Right;

Feitas todas as estruturas de dados que vão dar suporte à classe ***InputWrapper***, esta classe vai apenas limitar a instanciar as estruturas com nome parecidos à classe **GamePad**. A classe **InputWrapper** terá de ser "*static*", pois se for declarada apenas como "*public*" as novas instâncias poderão ser a cessadas posteriormente em outros métodos, mas não se manterão estáticas. Devido a isto tivemos alguns problemas com os botões.

```
static class InputWrapper
{
    static public AllInputButtons Buttons = new AllInputButtons();
    static public AllThumbSticks ThumbSticks = new AllThumbSticks();
    static public AllInputTriggers Triggers = new AllInputTriggers();
}
```

Fig.8 – Classe InputWrapper;

Finalizando o projeto, a nova classe ***InputWrapper*** serviu para simplificar o sistema de verificar as classes ***GamePad*** e ***Keyboard***, apenas tivemos de substituir as funções correspondentes na classe Game1 na função ***Update***(). Ficando desta forma:

```
protected override void Update(GameTime gameTime)
{
    #region Game Controller
    // Segue o jogo até ao fim
    if (InputWrapper.Buttons.Back == ButtonState.Pressed)
        this.Exit();

    // Carrega a posiçao da imagem com analogico esquerdo/direito
    mJPGPosition += InputWrapper.ThumbSticks.Left;
    mPNGPosition += InputWrapper.ThumbSticks.Right;
    #endregion

    base.Update(gameTime);
}
```

Fig.9 – Função *Update*() após as modificações;

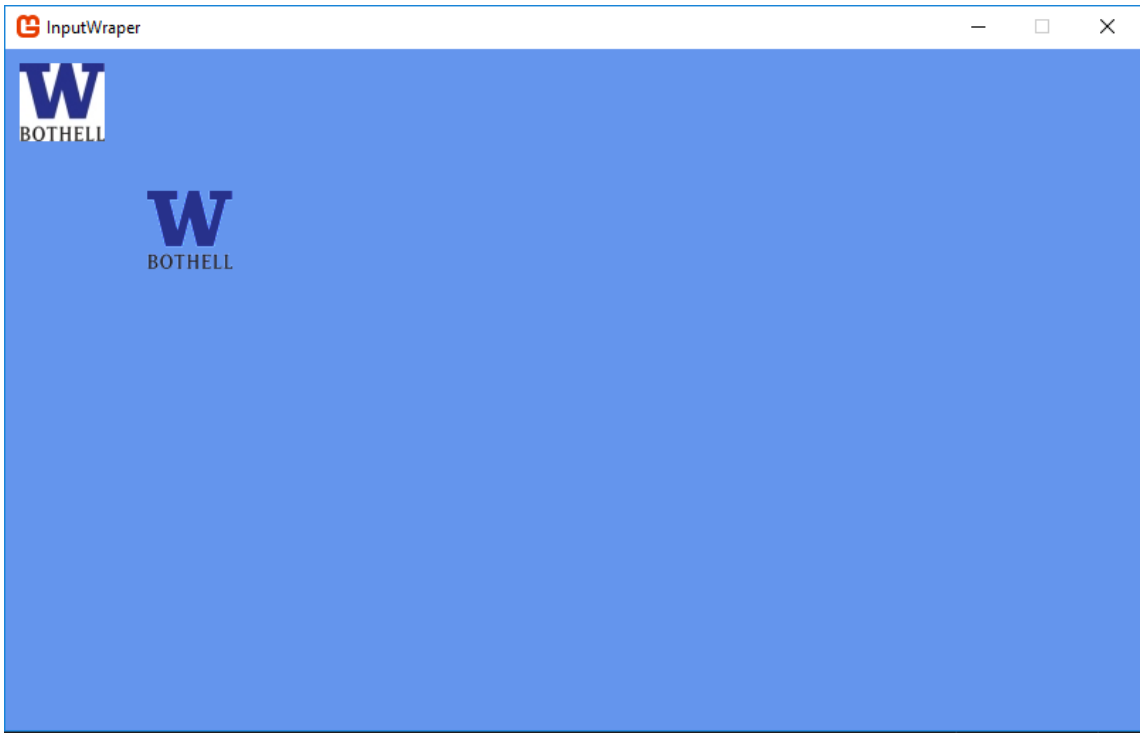


Fig.10 – Programa InputWrapper a funcionar;

Em resumo, este exercício não nos deu grandes problemas talvez devido à simplicidade do codigo.

Game Window Size Projet

O seguinte projeto consiste na modificação do projeto anterior (DrawAndControl) de maneira a poder suportar janela, isto é, o seguinte código vai permitir ao utilizador minimizar e maximizar a janela do jogo.Para tal, devem ser feitas certas alterações ao código.

Para começar, na classe Game1 deve ser adicionado as variáveis que indicam o tamanho da janela que o utilizador pretende:

```
#region Preferred Window Size
// Prefer window size
// Convention: "k" to begin constant variable names
const int kWindowWidth = 1000;
const int kWindowHeight = 700;
#endregion
```

FIg.11 – Variáveis do tamanho da janela;

E uma que permita o seu acesso e inicialização:

```
// Create graphics device to access window size
Game1.sGraphics = new GraphicsDeviceManager(this);
// set prefer window size
Game1.sGraphics.PreferredBackBufferWidth = kWindowWidth;
Game1.sGraphics.PreferredBackBufferHeight = kWindowHeight;
```

Fig.12 – Inicialização das variáveis;

De seguida, para permitir a mudança entre *full screen* e janela normal, alterou-se a função ***Update***(), adicionando um *if statement* para permitir sair de *full screen* e mais dois ciclos *if's* que permitem trocar entre janelas.

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

    #region Toggle full screen and window size
    // "A" to toggle full screen
    if (InputWrapper.Buttons.A == ButtonState.Pressed)
    {
        if (!Game1.sGraphics.IsFullScreen)
        {
            Game1.sGraphics.IsFullScreen = true;
            Game1.sGraphics.ApplyChanges();
        }
    }

    // "B" toggles back to window
    if (InputWrapper.Buttons.B == ButtonState.Pressed)
    {
        if (Game1.sGraphics.IsFullScreen)
        {
            Game1.sGraphics.IsFullScreen = false;
```

Na execução do exercício, não aparentava ter erros, pelo menos na *console* do *Visual Studio* mas ao tentar maximizar a janela, a janela apenas perdia as "bordas" e não maximizava. Este problema foi resolvido num dos *if's* da função ***Update***().

Projeto TexturedPrimitive

O seguinte projeto tem como funcionalidade demostrar como se pode carregar imagens e manipular o seu tamanho e posição utilizando o teclado.

Para por o projeto a suportar *sprites*, primeiramente tivemo que modificar novamente a classe Game1 de forma a permitir a utilização de *sprites*.

```
public class Game1 : Game
{
    #region Class variables defined to be globally accessible!!
    // for drawing support
    // Convention: staticClassVariable names begin with "s"
    /// <summary>
    /// sGraphicsDevice - reference to th graphics device for current display size
    /// sSpriteBatch - reference to the SpriteBatch to draw all of the primitives
    /// sContent - reference to the ContentManager to load the textures
    /// </summary>
    static public SpriteBatch sSpriteBatch;  // Drawing support
    static public ContentManager sContent;   // Loading textures
    static public GraphicsDeviceManager sGraphics; // Current display size
```

FIg.13 – Class Game1;

De seguida, foi necessário criar uma classe ***TexturedPrimitive*** que foi necessária para o suporte das texturas.

A classe ***TexturedPrimitive*** vai permitir ser usada mais tarde noutros projetos, tendo como função carregar texturas e fontes.

Declarou-se três variáveis, uma para suportar a imagem (*Texture2d*) e outras duas para a posição e tamanho (*Vector2*), sendo depois inicializadas através de um construtor (*public* ***TexturedPrimitive***) com parâmetros correspondentes, que vão permitir a sua utilização.

```
class TexturedPrimitive
{

    protected Texture2D mImage; // The UWB-JPG.jpg image to be loaded
   public Vector2 mPosition; // Center position of image
    protected Vector2 mSize; // Size of the image to be drawn
```

Fig.14 – Variáveis na Classe TexturedPrimitive;

```
public TexturedPrimitive(String imageName, Vector2 position, Vector2 size)
{
    mImage = Game1.sContent.Load<Texture2D>(imageName);
    mPosition = position;
    mSize = size;
}
```

A função a criar a seguir vai permitir, ao ser chamada, alterar o tamanho e posição da textura, onde as variáveis *deltaTranslate* e *deltaScale* são adicionadas ao tamanho e posição.

Caso *deltaTranslate* e *deltaScale* variem, posição e tamanho também variam, se se mantiverem estáticas, a posição e tamanho mantêm-se.

```
public void Update(Vector2 deltaTranslate, Vector2 deltaScale)
{
    mPosition += deltaTranslate;
    mSize += deltaScale;
}
```

Fig.16 – Função **Update**();

A próxima função será responsável pelo aparecimento da textura no ecrã.

A função **Draw**() irá ditar aonde estará e qual será o tamanho da textura, criando um objeto retângulo para o tamanho e posição, e chama-se o *sSpriteBatch.Draw* do Game1 para textura do objeto, e a cor.

```
public void Draw()
{

    // Defines where and size of the texture to show
    Rectangle destRect = Camera.ComputePixelRectangle(mPosition, mSize);
    Game1.sSpriteBatch.Draw(mImage, destRect, Color.White);
```

Fig.17 – Função **Draw**();

Para utilizar a classe **TexturedPrimitive**, tem que se realizar alterações nas funções **Update**(), **Draw**(), e **LoadContent**() da classe Game1.

O código adicionado no **LoadContent** vai criar um *array* de textura, com imagem, posição e tamanho inicial. Posteriormente têm que ser adicionadas ao projeto as imagens para serem carregadas, de maneira a evitar erros.

```
protected override void LoadContent()
{
    // Create a new SpriteBatch, which can be used to draw textures.
    Game1.sSpriteBatch = new SpriteBatch(GraphicsDevice);

    // Define camera window bounds
    Camera.SetCameraWindow(new Vector2(10f, 20f), 100f);

    // Create the primitives
    mGraphicsObjects = new TexturedPrimitive[kNumObjects];
    mGraphicsObjects[0] = new TexturedPrimitive( "UWB-JPG",new Vector2(15f, 25f), // Position to draw
     new Vector2(10f, 10f)); // Size to draw
    mGraphicsObjects[1] = new TexturedPrimitive( "UWB-JPG", new Vector2(35f, 60f),new Vector2(50f, 50f));
    mGraphicsObjects[2] = new TexturedPrimitive( "UWB-PNG", new Vector2(105f, 25f),
   new Vector2(10f, 10f));
    mGraphicsObjects[3] = new TexturedPrimitive("UWB-PNG",new Vector2(90f, 60f),new Vector2(35f, 35f));
}
```

Fig.18 – Função **LoadContent**();

A alteração realizada na função **Draw**() vai permitir carregar as imagens no ecrã, o ciclo *foreach* vai permitir chamar a função **Draw**() da classe **TexturedPrimitive.**

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

    Game1.sSpriteBatch.Begin(); // Initialize drawing support
    // Clear to background color
    GraphicsDevice.Clear(Color.CornflowerBlue);
    // Loop over and draw each primitive
    foreach (TexturedPrimitive p in mGraphicsObjects)
    {
        p.Draw();
    }
```

Fig.19 – Função **Draw**();

A alteração da função **Update**() permite alterar por entre *sprites*, e alterar o seu tamanho e posição através de inputs do *gamepad*.

```
#region Select object and control selected object
// Button-x to select the next object to work with
if (InputWrapper.Buttons.X == ButtonState.Pressed)
    mCurrentIndex = (mCurrentIndex + 1) % kNumObjects;

// Update currently working object with thumb sticks.
mGraphicsObjects[mCurrentIndex].Update(
        InputWrapper.ThumbSticks.Left,
        InputWrapper.ThumbSticks.Right);
#endregion
```

A criação da classe **Camera** vai permitir a navegação do *sprite* pelo sistema de coordenadas, para tal, serão efetuadas algumas alterações na classe Game1 para permitir o suporte da **Camera**.

Então na classe **Camera**, declaramos variáveis que vão representar a origem, proporção e largura entre a janela da camera e a janela de pixel;

```
static public class Camera
{
    static private Vector2 sOrigin = Vector2.Zero; // Origin of the world
    static private float sWidth = 100f; // Width of the world
    static private float sRatio = -1f; // Ratio between camera window and pixel

    static private float cameraWindowToPixelRatio()
```

Fig.20 – Classe **Camera;**

Criamos uma função que retorna a diferença entre as duas funções (cameraWindowToPixelRatio()), que vai ditar a proporção da janela,

```
static private float cameraWindowToPixelRatio()
{
    if (sRatio < 0f)
        sRatio = (float)Game1.sGraphics.PreferredBackBufferWidth / sWidth;
    return sRatio;
}
```

Fig.21 – Método cameraWindowToPixelRatio();

e outra função que vai ditar o tamanho e posição da janela da camera(SetCameraWindow).

```
static public void SetCameraWindow(Vector2 origin, float width)
{
    sOrigin = origin;
    sWidth = width;
}
```

Fig.22 – Método SetCameraWindow;

De seguida, a implementação da função **ComputePixelPosition**() converte a posição determinada pelo utilizador para pixel,

```
        static public void ComputePixelPosition(Vector2 cameraPosition,
out int x, out int y)
        {
            float ratio = cameraWindowToPixelRatio();

            // Convert the position to pixel space
            x = (int)(((cameraPosition.X - sOrigin.X) * ratio) + 0.5f);
            y = (int)(((cameraPosition.Y - sOrigin.Y) * ratio) + 0.5f);

            y = Game1.sGraphics.PreferredBackBufferHeight - y;
        }
```

e a **ComputePixelRectangle** converte uma área retangular inserida pelo utilizador através do sistema de coordenadas para espaço pixel.

```
        static public Rectangle ComputePixelRectangle(Vector2 position,
Vector2 size)
        {
            float ratio = cameraWindowToPixelRatio();

            // Convert size from camera window space to pixel space.
            int width = (int)((size.X * ratio) + 0.5f);
            int height = (int)((size.Y * ratio) + 0.5f);

            // Convert the position to pixel space
            int x, y;
            ComputePixelPosition(position, out x, out y);

            // Reference position is the center
            y -= height / 2;
            x -= width / 2;

            return new Rectangle(x, y, width, height);
        }
```

Fig.23 – Função **ComputePixelRectangle();**

Para utilizar a classe **Camera**, altera-se a função **LoadContent**, definindo uma janela para a camera, definindo também o seu limite.

```
// Define camera window bounds
Camera.SetCameraWindow(new Vector2(10f, 20f), 100f);
```

Fig.24 – Limite da janela;

A utilização da função **Draw**() da classe **TexturedPrimitives** irá servir para converter todas as posições inseridas pelo utilizador para espaço em pixel.

Para acrescentar texto ao jogo for necessário criar uma nova classe, a classe **FontSupport** que permitiu a utilização de texto no jogo, para implementa-la, primeiramente criamos a classe, onde declaramos três variáveis que ditam a fonte do texto, a cor e a posição no ecrã.

```
class FontSupport
{
    static private SpriteFont sTheFont = null;
    static private Color sDefaultDrawColor = Color.Black;
    static private Vector2 sStatusLocation = new Vector2(5, 5);
```

FIg.25 – Classe **FontSupport** e as suas variáveis;

Ao implementarmos a função **LoadFont**(), permite-nos inicializar a fonte do texto, que para efeitos demostrativos, foi utilizada a fonte "*Arial*".

```
static private void LoadFont()
{
    // For demo purposes, loads Arial.spritefont
    if (null == sTheFont)
        sTheFont = Game1.sContent.Load<SpriteFont>("Arial");
}
```

Fig.26 – Função **LoadFont**()

E a função **ColorToUse**() vai permitir alterar a cor do texto para qualquer cor especificada, caso nenhuma cor seja especificada, a cor retornada vai ser a *default*, que é a cor preta.

```
static private Color ColorToUse(Nullable<Color> c)
{
    return (null == c) ? sDefaultDrawColor : (Color)c;
}
```

Fig.27 – Função **ColorToUse**();

Para suportar a impressão da fonte para a janela de jogo, adicionamos duas funções:

A função **PrintStatus**(), que contem dois parâmetros, um para a mensagem que vai ser imprimida, e outra para a cor da fonte. Esta função coloca a mensagem no canto superior esquerdo da janela.

```
static public void PrintStatus(String msg, Nullable<Color> drawColor)
{
    LoadFont();
    Color useColor = ColorToUse(drawColor);
    // Compute top-left corner as the reference for output status
    Game1.sSpriteBatch.DrawString(sTheFont, msg, sStatusLocation, useColor);
}
```

Fig.28 – Função **PrintStatus**();

A função **PrintStatusAt**() é semelhante a função **PrintStatus**(), sendo que esta permite colocar a mensagem num local especifico, utilizando a função **ComputePixelPosition**(), transformando o parâmetro posição para coordenadas especificadas pelo utilizador.

```
static public void PrintStatusAt(Vector2 pos, String msg, Nullable<Color> drawColor)
{
    LoadFont();
    Color useColor = ColorToUse(drawColor);
    int pixelX, pixelY;
    Camera.ComputePixelPosition(pos, out pixelX, out pixelY);
    Game1.sSpriteBatch.DrawString(sTheFont, msg,
   new Vector2(pixelX, pixelY), useColor);
}
```

Fig.29 – Função **PrintStatusAt**(),

Para por a classe **FontSupport** a funcionar, foi necessario alterar a função **Draw**() da classe Game1, chamando a função **PrintStatus()** e **PrintStatusAt()**.

```
// Print out text message to echo status
FontSupport.PrintStatus("Selected object is:" + mCurrentIndex +
" Location=" + mGraphicsObjects[mCurrentIndex].mPosition, null);
FontSupport.PrintStatusAt(mGraphicsObjects[mCurrentIndex].mPosition, "Selected",
Color.Red);
```

Fig.30 – Código que mostra a localização e o objeto selecionado;

Infelizmente, colocar texto no jogo tornou-se um desafio para nós, pois o livro pelo qual seguimos as instruções continha imensos erros, mas com persistência encontramos o erro que fazia com que não carregasse, este erro estava na função PrintStatusAt() e não reconhecia a variável mPosition.

SimpleGameProject

O **SimpleGameProject** consiste num jogo simples que já inclui mecânicas de colisão simples, para tal, e aproveitando as classes anteriores, será preciso de fazer algumas alterações no código para poder suportar as novas mecânicas.

Para começar, é necessário adicionar duas novas variáveis na classe **TexturedPrimitives**, nomeadamente a ***MaxBound*** e ***MinBound***. Estas duas variáveis permitem utilizar a posição e tamanho da textura para calcular os limites, permitindo que estes estejam sempre corretos.

```
public Vector2 MinBound { get { return mPosition - (0.5f * mSize); } }
public Vector2 MaxBound { get { return mPosition + (0.5f * mSize); } }
```

Fig.31 – Variáveis *MixBound* e *MaxBound*;

Na classe ***Camera***, são adicionados limites semelhantes. Utilizando a sua posição (*sOrigin*) e tamanho (*sWidth*, sHeight), calcula-se o *MinBound* da *camera* e o *MaxBound*. Como a origem (*sOrigin*) encontra-se no fundo da janela, este será o limite mínimo, e somando o tamanho (*sWidth*, *sHeight*) a origem, obtém-se o limite máximo.

```
static public class Camera
{
    static private Vector2 sOrigin = Vector2.Zero; // Origin of the world
    static private float sWidth = 100f; // Width of the world
    static private float sRatio = -1f; // Ratio between camera window and pixel
    static private float sHeight = 70f;
    /// Accessors to the camera window bounds
    static public Vector2 CameraWindowLowerLeftPosition
    { get { return sOrigin; } }
    static public Vector2 CameraWindowUpperRightPosition
    { get { return sOrigin + new Vector2(sWidth, sHeight); } }
```

Fig.32 – Classe Camera modificada;

Para podermos adicionar o suporte de colisão de objetos, foi necessário primeiro criar a função ***CameraWindowCollisionStatus***(), que basicamente representa os cinco estados, ou as cinco posições que um objeto ao colidir se encontra, isto é, ou no limite mínimo (o fundo), no limite máximo (topo), na esquerda, na direita, ou interior.

```
// Support collision with the camera bounds
public enum CameraWindowCollisionStatus
{
    CollideTop = 0,
    CollideBottom = 1,
    CollideLeft = 2,
    CollideRight = 3,
    InsideWindow = 4
};
```

Fig.33 – Função **CameraWindowCollisionStatus**();

Adicionamos então a função que nos dará deteção de colisão para a *camera*. A função **CollidedWithCameraWindow**() aceita um objeto da **TexturedPrimitives**, testando a colisão com os limites da *camera*, e retornando o resultado. Os 4 *if's* cobrem as colisões que são possíveis de acontecer, caso bata na esquerda, direita, cima ou baixo.

```
static public CameraWindowCollisionStatus CollidedWithCameraWindow(TexturedPrimitive prim)
{
    Vector2 min = CameraWindowLowerLeftPosition;
    Vector2 max = CameraWindowUpperRightPosition;
    if (prim.MaxBound.Y > max.Y)
        return CameraWindowCollisionStatus.CollideTop;
    if (prim.MinBound.X < min.X)
        return CameraWindowCollisionStatus.CollideLeft;
    if (prim.MaxBound.X > max.X)
        return CameraWindowCollisionStatus.CollideRight;
    if (prim.MinBound.Y < min.Y)
        return CameraWindowCollisionStatus.CollideBottom;
    return CameraWindowCollisionStatus.InsideWindow;
}
```

Fig.34 – Função **CameraWindowCollisionStatus**();

Por fim adiciona-se uma variável *random* na classe Game1, que ira gerar números aleatórios.

```
static public Random sRan; // For generating random numbers
public Game1()
{
    Game1.sRan = new Random();
```

Fig.35 – Uso do *Random;*

De seguida, foi necessário criar a função **soccer**() – que irá permitir inserir e trabalhar com a bola, o seu tamanho e etc.

Fazemos com que a classe Soccer receba como herança a classe **TexturedPrimitives** de maneira a podermos aproveitar as suas funcionalidades.

Adicionamos de seguida a variável *mDeltaPosition*, que vai representar as mudanças de posição. Esta função serviu para mover a bola.

```
public class SoccerBall : TexturedPrimitive
{
    private Vector2 mDeltaPosition; // Change current position by this amount
```

Fig.36 – Declaração da variável vetor mDeltaPosition;

Criamos o construtor, de maneira a podermos depois utilizar os parâmetros especificados dentro dela, que vão ditar o tamanho e a posição da bola ao serem chamadas.

```
public SoccerBall(Vector2 position, float diameter) :
base("Soccer", position, new Vector2(diameter, diameter))
{
    mDeltaPosition.X = (float)(Game1.sRan.NextDouble()) * 2f - 1f;
    mDeltaPosition.Y = (float)(Game1.sRan.NextDouble()) * 2f - 1f;
}
```

Fig.37 – SoccerBall;

Para definir o raio que a bola vai ter, foi criada uma função que ira determina-lo, e visto que a classe soccer é considerada como um retângulo, alterar o raio da bola significara alterar a altura e largura do retângulo.

```
public float Radius
{
    get { return mSize.X * 0.5f; }
    set { mSize.X = 2f * value; mSize.Y = mSize.X; }
}
```

Fig.38 – Definir o raio da bola;

Para definir o comportamento da bola no momento de impacto, criamos uma função **Update**(), que vai ditar o posicionamento da bola a seguir ao impacto.

```
public void Update()
{
    Camera.CameraWindowCollisionStatus status =
    Camera.CollidedWithCameraWindow(this);
    switch (status)
    {
        case Camera.CameraWindowCollisionStatus.CollideBottom:
        case Camera.CameraWindowCollisionStatus.CollideTop:
            mDeltaPosition.Y *= -1;
            break;

    case Camera.CameraWindowCollisionStatus.CollideLeft:
        case Camera.CameraWindowCollisionStatus.CollideRight:
            mDeltaPosition.X *= -1;
            break;
    }
    mPosition += mDeltaPosition;
}
```

Fig.39 – Função **Update**() modificada;

Assim ficou concluída a classe Soccer. De maneira a implementar estas alterações e podermos por o jogo a rodar, foram necessárias algumas alterações na classe Game1.

Primeiro, declaramos duas variáveis, uma para o logo, **TexturedPrimitive**, para carregar a textura, e a outra vai ditar a posição inicial e tamanho da bola (*SoccerBall mball*).

```
TexturedPrimitive mUWBLogo;
SoccerBall mBall;
Vector2 mSoccerPosition = new Vector2(50, 50);
float mSoccerBallRadius = 3f;
#endregion
```

Fig.40 – Declaração das variáveis;

Seguidamente, alteramos a função **LoadContent**() de maneira a permitir carregar a imagem pretendida, a bola de futebol.

```
// Create the primitives
mUWBLogo = new TexturedPrimitive("UWB-PNG", new Vector2(30, 30), new Vector2(20, 20));
mBall = new SoccerBall(mSoccerPosition, mSoccerBallRadius * 2f);
```

Fig.41 – Alterações na função **LoadContent**();

Alteramos a função **Update**(), para alterar as funcionalidades dos comandos, de maneira ao carregar no botão especifico, uma nova bola é instanciada.

```
mUWBLogo.Update(InputWrapper.ThumbSticks.Left, Vector2.Zero);
mBall.Update();
mBall.Update(Vector2.Zero, InputWrapper.ThumbSticks.Right);

if (InputWrapper.Buttons.A == ButtonState.Pressed)
    mBall = new SoccerBall(mSoccerPosition, mSoccerBallRadius * 2f);
```

Fig.42 – Código responsável pela funcionalidade do botão;

E por ultimo, alteramos a função **Draw**() para a textura aparecer no ecrã, completando assim o código funcional de um jogo simples.

```
mUWBLogo.Draw();
mBall.Draw();
// Print out text message to echo status
FontSupport.PrintStatus("Ball Position:" + mBall.mPosition, null);
FontSupport.PrintStatusAt(mUWBLogo.mPosition,
mUWBLogo.mPosition.ToString(), Color.White);
FontSupport.PrintStatusAt(mBall.mPosition, "Radius" + mBall.Radius, Color.Red);
```

Fig.43

Com este exercício não tivemos qualquer problema na execução do mesmo.

The Simple Game State Project

Este projeto tem como objetivo demonstrar como se move o personagem através do ecrã e apanhar bolas de basquete. As bolas de basquete vão ser geradas aleatoriamente e aumentarão de tamanha continuadamente ao longo do tempo. Quando as bolas chegam a um determinado tamanho elas explodem. Caso não explodam, o jogador ganha o jogo.

Inicialmente, modificamos a classe **TexturePrimituve** e adicionamos a função **PrimitiveTouches**() que deteta se dois objetos estão sobrepostos no espaço. A função compara a distância entre os dois objetos com metade da largura dos dois objetos.

```
public bool PrimitivesTouches(TexturedPrimitive otherPrim)
{
    Vector2 v = mPosition - otherPrim.mPosition;
    float dist = v.Length();
    return (dist < ((mSize.X / 2f) + (otherPrim.mSize.X / 2f)));
}
```

Fig.44

Seguidamente, criamos a classe **BasketBall** que vai receber como herança a classe **TexturePrimitive**, depois adicionamos três variáveis; uma vai determinar a velocidade com que a bola aumenta de tamanho, a outra vai ditar o tamanho inicial da bola e por fim uma que determina com que tamanho a bola arrebenta.

Posteriormente, desenvolvemos um construtor que inicializa a posição da bola de basquete num lugar aleatório e determina o seu tamanho inicial. Também, criamos a função **UpdateAndExplode**() que dita quando a bola vai explodir ou desaparecer ao chegar ao seu tamanho final.

```
public bool UpdateAndExplode()
{
    mSize *= kIncreaseRate;

    return mSize.X > kFinalSize;
}
```

Fig.45

Para inicializar os elementos descritos anteriormente, foi necessário a criação de uma classe **MyGame**, para tal declaramos três variáveis que irão servir para o personagem; uma para a seu a textura; uma para o seu tamanho e outra para a posição.

```
// Hero stuff ...
TexturedPrimitive mHero;
Vector2 kHeroSize = new Vector2(15, 15);
Vector2 kHeroPosition = Vector2.Zero;
```

Fig.46 – Variáveis relativas ao heroi;

Para as bolas de basquete foi necessário a criação de uma lista de objetos, para permitir o aparecimento de várias bolas no ecrã ao longo do jogo. Adicionalmente, desenvolveu-se um *TimeSpan* para permitir se deve aparecer uma nova bola ou não.

```
// Basketballs ...
List<Basketball> mBBallList;
TimeSpan mCreationTimeStamp;
int mTotalBBallCreated = 0;
// this is 0.5 seconds
const int kBballMSecInterval = 500;
```

Fig.47 – Variáveis relativas ao aparecimento das bolas;

As últimas variáveis inseridas servem para a pontuação do jogo e para mostrar as texturas de quando o jogador ganha ou perde.

```
// Game state
int mScore = 0;
int mBBallMissed = 0, mBBallHit = 0;
const int kBballTouchScore = 1;
const int kBballMissedScore = -2;
const int kWinScore = 10;
const int kLossScore = -10;
TexturedPrimitive mFinal = null;
```

Fig.48 – Estado do Jogo;

De seguida, criou-se um construtor para inicializar a textura, a posição, a lista de basquete e o *TimeStamp*.

```
public MyGame()
{
    // Hero ...
    mHero = new TexturedPrimitive("Me", kHeroPosition, kHeroSize);
    // Basketballs
    mCreationTimeStamp = new TimeSpan(0);
    mBBallList = new List<Basketball>();
}
```

Depois foi preciso criar a função **UpdateGame**() que irá verificar se o jogo acabou, verifica-se se a bola explodiu e se sim remove-a do jogo, também verifica se o jogador apanhou a bola de basquete, se sim faz um *update* da lista de bolas e do resultado, faz com que apareçam novas bolas de basquete e por ultimo esta função verifica se tens pontuação para ganhar ou para perder.

Por fim, criamos a função **DrawGame**() – esta função faz com que as texturas sejam desenhadas no ecrã que correspondem ao personagem, as bolas de basquete e à imagem de vitoria ou derrota. Imprimi também o resultado o *status*.

```
public void DrawGame()
{
    mHero.Draw();
    foreach (Basketball b in mBBallList)
        b.Draw();
    if (null != mFinal)
        mFinal.Draw();
    // Drawn last to always show up on top
    FontSupport.PrintStatus("Status: " +"Score=" + mScore + " Basketball: Generated( " +
   mTotalBBallCreated + ") Collected(" + mBBallHit + ") Missed(" +mBBallMissed + ")", null);
}
```

Fig.50 – Função DrawGame();

Para que possamos fazer uso das funções e classe acima descritas, simplesmente chamamos as funções criadas na classe **MyGame** nas funções correspondentes.

Neste exercício, foram encontrados muitos erros relativos à classe ***TexturePrimitive,*** que impossibilitou a execução do programa. Mas com a ajuda do livro e de alguns colegas conseguimos resolver o problema, mesmo não sabendo qual era.

The Rotate Textured Primitive project

Este projeto tem como objetivo mover um objeto em torno da janela do jogo e girá-la no sentido horário ou anti-horário. Também tem como objetivo demonstrar como selecionar o objeto *Ball* ou o objeto *Logo* e dimensiona-lo.

Inicialmente modificamos a classe **TexturedPrimitive** adicionando a variável *mRotateAngle*. Esta variável tem como função manter o ângulo de rotação no sentido horário da textura em radianos.

```
mRotateAngle = 0f;
```

Fig.51

Depois de adicionada a variável de ângulo de rotação, foi preciso inicializa-la dentro do construtor. A classe **TexturedPrimitive** contém dois construtores; Portanto, o ângulo de rotação tem de ser inicializado dentro de ambos. Ambos os construtores inicializam a variável a zero, pois o objeto ***TexturedPrimitive*** não tem ângulo de rotação após a criação.

De seguida, foi preciso fornecer um "*accessor*", pois vamos modificar o ângulo de rotação.

```
public float RotateAngleInRadian { get { return mRotateAngle; }
```

Fig.52 – *Get* do RotateAngleInRadian;

Foi também necessário modificar a função **Update**() da classe **TexturedPrimitive** para dar suporte para utilizar o angulo de rotação. Para isto apenas foi preciso adicionar outro parâmetro para especificar o angulo de rotação.(ex. *mPosition* é igual à soma *mPosition* mais o *deltaTranslate.*)

```
public void Update(Vector2 deltaTranslate, Vector2 deltaScale,
{
    mPosition += deltaTranslate;
    mSize += deltaScale;
    mRotateAngle += deltaAngleInRadian;
}
```

Fig.53 – Função Update()

A ultima função que tivemos de modificar foi a função **Draw**() da classe **TexturedPrimitive**. Esta função inclui três passos importantes: converte a posição e o tamanho da **TexturedPrimitive** em

"*pixel space*" para desenhar; Calcula a rotação de origem como a do pivô; e desenha a textura para a tela usando o *SpriteBatch*.

Para converter do espaço de coordenadas definido pelo usuário para o espaço de pixels, usamos a função **ComputePixelRectangle**() criada na classe **Camera** no exercício passado. Para isso bastou passar a posição e o tamanho da textura e armazenar os resultados numa variável *Rectangle Local* chamada *destRect*, que usaremos posteriormente para desenhar.

Agora que a classe **TexturedPrimitive** suporta a rotação, podemos usa-la dentro da classe **GameState** para demonstrar que está a funcionar corretamente. Primeiro adicionamos as variáveis **TexturedPrimitive** e inicializamos dentro do construtor.

```
public class GameState
{
    // Work with Textured Primitive and
    TexturedPrimitive mBall, mUWBLogo;
    TexturedPrimitive mWorkPrim;

    /// <summary>
    /// Constructor
    /// </summary>
    public GameState()
    {
        // Create the primitives
        mBall = new TexturedPrimitive("Soccer", new Vector2(30, 30), new Vector2(10, 15));
        mUWBLogo = new TexturedPrimitive("UWB-JPG", new Vector2(60, 30), new Vector2(20, 20));
        mWorkPrim = mBall;
    }
```

Fig.54 – GameState;

Depois, mudamos a função **Update**() para suportar a seleção primitiva atual e também a rotação dessa mesma primitiva, isto no caso dos botões X ou Y forem pressionados.

```
    public void UpdateGame()
    {
        #region Select which primitive to work on
        if (InputWrapper.Buttons.A == ButtonState.Pressed)
            mWorkPrim = mBall;
        else if (InputWrapper.Buttons.B == ButtonState.Pressed)
            mWorkPrim = mUWBLogo;
        #endregion

        #region Update the work primitive
        float rotation = 0;
        if (InputWrapper.Buttons.X == ButtonState.Pressed)
            rotation = MathHelper.ToRadians(1f); // 1 degree per-press
        else if (InputWrapper.Buttons.Y == ButtonState.Pressed)
            rotation = MathHelper.ToRadians(-1f); // 1 degree per-press
        mWorkPrim.Update(
            InputWrapper.ThumbSticks.Left,
            InputWrapper.ThumbSticks.Right,
            rotation);
        #endregion
    }
```

Fig.55 – UpdateGame()

Na fig.55 podemos ver pelo código, se o botão **A** tiver sido carregado, *mWorkPrim* será igual ao *mBall*, isto acontece de maneira parecida se o botão **B** for carregado apenas *mWorkPrim* será igual ao *mLogo*. Por fim, modificamos a função **DrawGame**() para desenhar ambos os objetos da classe **TexturedPrimitive** e exibir o seus ângulos de rotação através da classe **FontSupport**.

```
public void DrawGame()
{
    mBall.Draw();
    FontSupport.PrintStatusAt(mBall.Position, mBall.RotateAngleInRadian.ToString(), Color.Red);

    mUWBLogo.Draw();
    FontSupport.PrintStatusAt(mUWBLogo.Position, mUWBLogo.Position.ToString(), Color.Black);

    // Print out text message to echo status
    FontSupport.PrintStatus("A-Soccer B-Logo LeftThumb:Move RightThumb:Scale X/Y:Rotate", null);
}
```

Fig.56 - DrawGame()

PrintStatusAt é a função da classe **FontSupport** que usamos para imprimir as posições, as cores e os restantes objetos. Já a função **PrintStatus**() é responsável por imprimir o texto desejado. Com este exercício podemos observar na sua execução que as texturas podem ser giradas no sentido horário e anti-horário e que giram em torno do centro da imagem. Na resolução do exercício não obtivemos qualquer tipo de erros/problemas.

Fig.57-Exercicio a executar

The Show Vector Project

Neste projeto temos como objetivo entender como representar e manipular vetores dentro da janela do jogo. No fim poderemos selecionar e manipular cada um dos vetores alterando os seus atributos, como direção e tamanho.

Primeiramente, criamos uma nova classe **ShowVector** dentro da pasta **GraphicsSuppor**. Depois declaramos duas variáveis como *static*, que oferecem suporte para desenhar o objeto **ShowVector**.

De seguida, adicionamos uma nova função para carregar a variável da imagem com a textura desejada. Game1.sContent.Load<Texture2D>(“”) tem como objetivo carregar a tal textura.

```
static private void LoadImage()
{
    if (null == sImage)
        ShowVector.sImage = Game1.sContent.Load<Texture2D>("Arrow");
}
```

Fig.58 LoadImage();

Posteriormente, adicionamos um nova função que pudesse desenhar corretamente o vetor para o jogo. Para isto criamos a função **DrawPointVector**() que aceita dois argumentos, o primeiro argumento é a posição inicial e a segunda é a direção. Na criação desta função tivemos alguns problemas devido ao facto de não declararmos os argumentos como Vector2.

No desenvolvimento desta função, primeiramente carregamos a imagem vetorial para ser desenhada usando a função anteriormente criada, função **LoadImage**(). De seguida, foi preciso calcular o angulo de rotação correto para a imagem e para isso calculamos o ângulo entre a direção do vetor e o eixo x. Depois de calculado o ângulo de rotação correto, fizemos uso do valor de **Y** para que nos indica-se o sentido da rotação. Caso **Y** seja positivo a rotação será feita no sentido anti-horário se for negativo será no sentido horário. Toda esta explicação está resumida na figura 59.

```
#region Step 4b. Compute the angle to rotate
float length = dir.Length();

float theta = 0f;

if (length > 0.001f)
{
    dir /= length;
    theta = (float)Math.Acos((double)dir.X);
    if (dir.X < 0.0f)
    {
        if (dir.Y > 0.0f)
            theta = -theta;
    }
    else
    {
        if (dir.Y > 0.0f)
            theta= -theta;
    }
}
#endregion
```

Fig.59 - DrawPointVector step4b;

Depois da rotação correta para a imagem for descoberta, podemos desenha-la chamando a função **Game1.sSpriteBatch.Draw().** A última parte desta função imprime a direção e o tamanho do vetor e para isso usamos a função **PrintStatusAt**() da classe **FontSupport**.

```
#region Step 4d. Print status message
String msg;
msg = "Direction=" + dir + "\nSize=" + length;
FontSupport.PrintStatusAt(from + new Vector2(2, 5), msg, Color.White);
#endregion
```

Fig.60 - DrawPointVector step4d;

Com a função **DrawPointVector**() completa, precisamos de adicionar outra função para desenhar um vetor entre dois pontos. Então usamos essa mesma função criada anteriormente numa outra função nomeada **DrawFromTo**()- (from, to – front) significa entre os tais dois pontos).

```
static public void DrawFromTo(Vector2 from, Vector2 to)
{
    DrawPointVector(from, to - from);
}
```

Fig.61- DrawFromTo;

A ultima função que necessitamos de criar é uma função de rotação, sendo então criamos a função **RotateVectorByAngle**(), que tem como tarefa aceitar um vetor e a rotação desejada. Depois da função calcular a rotação ((x= cosTheta * v.X + sinTheta * v.Y) e (y=-sinTheta * v.X + cosTheta *v.Y)), ela retorna o novo vetor(x ,y).

```
static public Vector2 RotateVectorByAngle(Vector2 v, float angleInRadian)
{
    float sinTheta = (float)(Math.Sin((double)angleInRadian));
    float cosTheta = (float)(Math.Cos((double)angleInRadian));
    float x, y;
    x =  cosTheta * v.X + sinTheta * v.Y;
    y = -sinTheta * v.X + cosTheta * v.Y;
    return new Vector2(x, y);
}
```

Fig.62 – RotateVectorByAngle**();**

Após a classe **ShowVector** ser implementada, nós modificamos a classe **GameState** para dar suporte aos vetores desejados. Adicionamos as variáveis necessárias e modificamos o construtor **GameState** de modo que inicializa-se os "localizadores" da imagem, posição, tamanho e nome.

```
public GameState()
{
    // Create the primitives
    mPa = new TexturedPrimitive("Position", new Vector2(30, 30), kPointSize, "Pa");
    mPb = new TexturedPrimitive("Position", new Vector2(60, 30), kPointSize, "Pb");
    mPx = new TexturedPrimitive("Position", new Vector2(20, 10), kPointSize, "Px");
    mPy = new TexturedPrimitive("Position", new Vector2(20, 50), kPointSize, "Py");
    mCurrentLocator = mPa;
}
```

Fig.63 - GameState();

Com as variáveis exigidas declaradas e inicializadas, tivemos de acrescentar código para alterar os seus estados na função **Update**(). Isto permitiu que o usuário altera-se o vetor selecionado apenas carregando nos botões A,B,X e Y do *gamepad*.

Este código permite que o utilizador mude o comprimento do vetor através do analógico esquerdo e rodar o vetor através do analógico direito.

```
#region Step 3c. Rotate Vector
// Left thumbstick-X rotates the vector at Py
float rotateYByRadian = MathHelper.ToRadians(
            InputWrapper.ThumbSticks.Left.X);
#endregion
```

Fig.64 -GameState step 3c.

A última função que precisamos de modificar dentro da classe **GameState** foi a função **DrawGame**(), nela os vetores serão desenhados através da classe **ShowVector** e os "localizadores" serão desenhados usando a função **TexturedPrimitiveDraw**().

Relativamente a problemas, fomos encontrando diversos tipos de erros, alguns explicados acima, como erros de sintaxe e problemas nas bibliotecas.

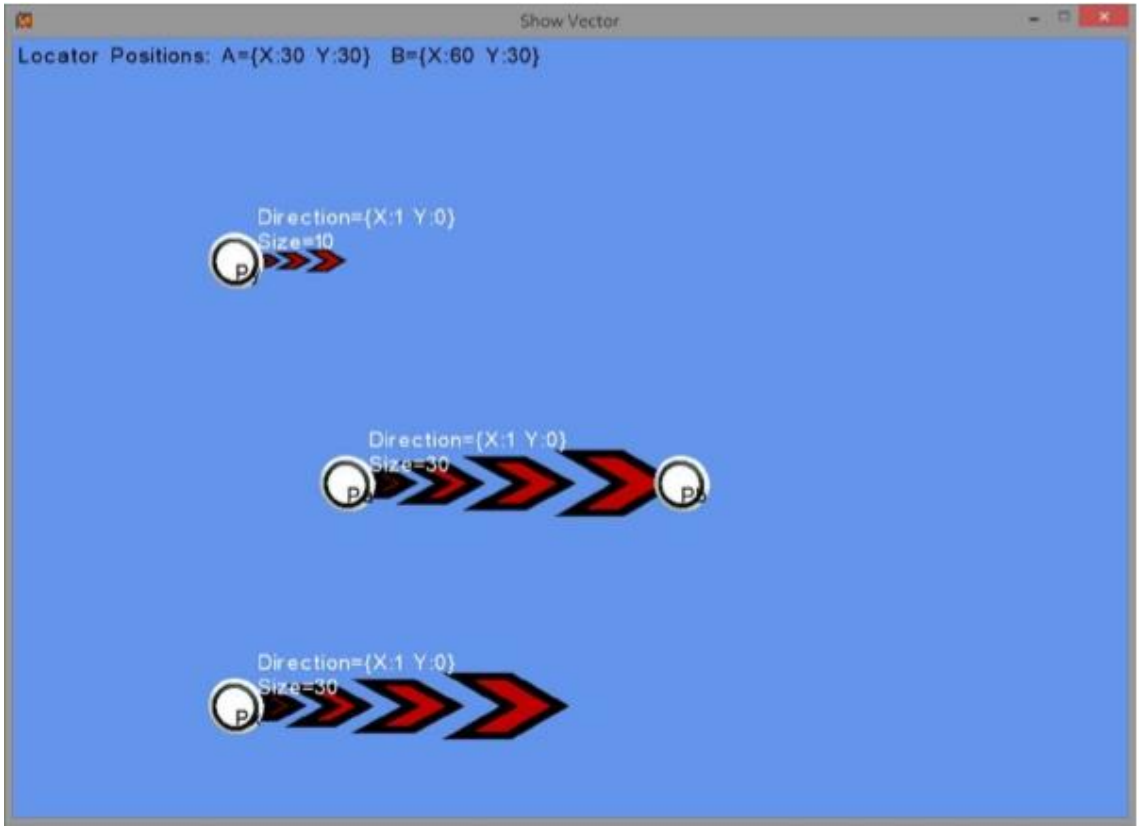


Fig.65 – ShowVector;

Front Direction

Este projeto tem como objetivo demonstrar como usar o conceito de "*front direction*" controlando um foguete, a direção que ele percorre e, a direção dos seus projeteis. O foguete poderá mover-se, girar e disparar. O objetivo do utilizador é acertar com um projétil na abelha, caso o projétil acerta a abelha, faz com que a abelha reapareça numa outra posição.

Inicialmente, declaramos as variáveis necessárias para suportar o estado de jogo desejado. Depois de declaradas as variáveis necessárias, inicializamos as variáveis com os valores que nos foram indicados no livro na **GameState**(). A seguir foi preciso carregar cada um dos *game objets*, nesta parte começamos por carregar o foguete e para isso usamos o analógico direito (X) e movemos o foguete com o analógico esquerdo (Y).

```
public GameState()
{
    // Create and set up the primitives
    mRocket = new TexturedPrimitive("Rocket", new Vector2(5, 5), new Vector2(3, 10));
    mRocketInitDirection = Vector2.UnitY; // initially umbrella is pointing in the pos

    mNet = new TexturedPrimitive("Net", new Vector2(0, 0), new Vector2(2, 5));
    mNetInFlight = false; // until user press "A", Foguetao nao esta a voar
    mNetVelocity = Vector2.Zero;
    mNetSpeed = 0.5f;

    // Inicia uma nova insect
    mInsect = new TexturedPrimitive("Insect", Vector2.Zero, new Vector2(5, 5));
    mInsectPreset = false;

    // Inicia o status do jogo
    mNumInsectShot = 0;
    mNumMissed = 0;
}
```

Fig.66 - GameState();

```
#region Step 4a. Update Rocket control
mRocket.RotateAngleInRadian +=
        MathHelper.ToRadians(InputWrapper.ThumbSticks.Right.X);
mRocket.Position += InputWrapper.ThumbSticks.Left;
#endregion
```

Fig.67 – UpdateGame() 1ºpasso;

Após feito o primeiro passo, avançamos para o próximo que consiste em atualizar o comportamento do projétil e para tal modificamos as suas variáveis quando o botão **A** é pressionado.

```
if (InputWrapper.Buttons.A == ButtonState.Pressed)
{
    mNetInFlight = true;
    mNet.RotateAngleInRadian = mRocket.RotateAngleInRadian;
    mNet.Position = mRocket.Position;
    mNetVelocity = ShowVector.RotateVectorByAngle(
                    mRocketInitDirection, mNet.RotateAngleInRadian) * mNetSpeed;
}
```

Fig.68 – UpdateGame() 2ºpasso;

O Ultimo passo era carregar a abelha, e facilmente a carregamos com o código a baixo.

```
if (!mInsectPreset)
{
    float x = 15f + ((float)Game1.sRan.NextDouble() * 30f);
    float y = 15f + ((float)Game1.sRan.NextDouble() * 30f);
    mInsect.Position = new Vector2(x, y);
    mInsectPreset = true;
}
```

A ultima alteração que tivemos de fazer na classe **GameState** foi na função **DrawGame**(), nela nós simplesmente chamamos as funções de desenho dos objetos se o seu estado for definido como verdadeiro.

```
public void DrawGame()
{

    mRocket.Draw();
    if (mNetInFlight)
        mNet.Draw();

    if (mInsectPreset)
        mInsect.Draw();
```

Fig.69 – DrawGame();

Terminados todas as alterações necessárias e todo o código acrescentado, executamos o programa para analisar o comportamento dos objetos em causa. Concluímos que o programa funciona corretamente, porém achamos que em termos de arquitetura de jogo, a aplicação ainda está um pouco simples. Também podemos retirar deste exercício que criar objetos torna-se um pouco aborrecido e tedioso, já que as propriedades de cada objeto também precisam de ser declaradas. Na resolução deste exercício, foram encontrados diversos erros de sintaxe que com ajuda da consola de erros do *Visual Studio* foram resolvidos.

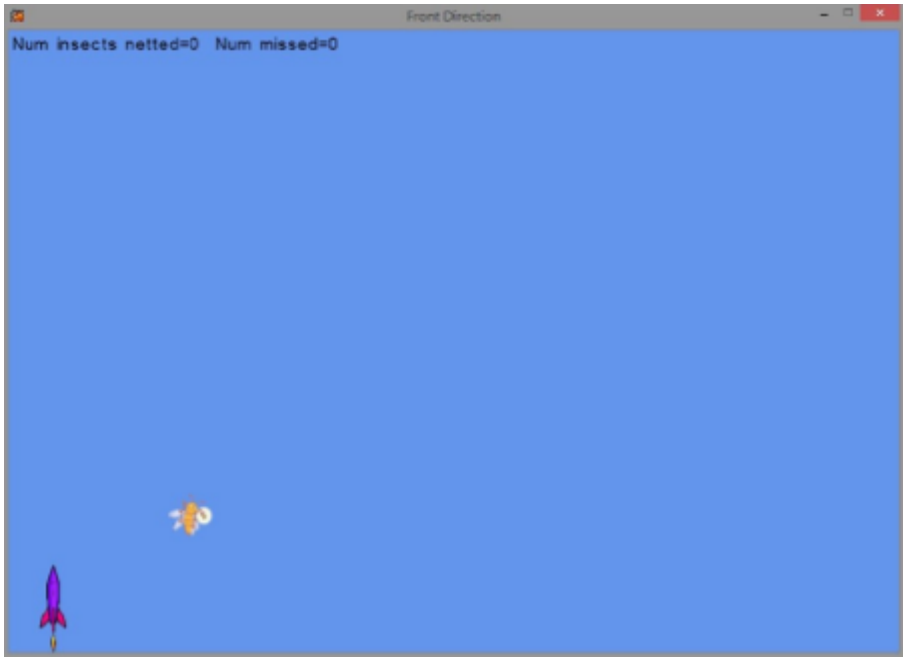


Fig.70 – Front Direction;

Projeto Game Object

Este projeto serve para demonstrar como mover um objeto livremente pela janela do jogo, permitindo ajustar a sua velocidade e a sua direção.

Para poder aplicar tais mecanismos no projeto, tivemos que criar a classe **GameObject**, esta vai permitir adicionar comportamentos específicos ao objeto, que neste caso será a alteração da velocidade e direção.

Para isso criamos três variáveis:

- Uma que inicia o movimento frontal;
- Uma que irá alterar a rapidez da direção;
- E por ultimo, uma que permita aumentar a velocidade do objeto.

```
public class GameObject : TexturedPrimitive
{
    // Inicializa a posição inicial frontal (quando a variavel RotateAngle é 0)
    protected Vector2 mInitFrontDir = Vector2.UnitY;

    // Comportamento da velocidade
    protected Vector2 mVelocityDir; // Se não for zero, a velocidade é normalizada
    protected float mSpeed; // velocidade
```

Fig.71 – Declarar variáveis;

Para inicializar a classe, criamos um construtor que permite a utilização dos argumentos do objeto, o nome, posição, tamanho e a sua qualificação respetivamente.

```
public GameObject(String imageName, Vector2 position, Vector2 size, String label = null)
        : base(imageName, position, size, label)
{
    InitGameObject();
}
```

Fig.72

Colocamos as variáveis *mVelocityDir* e *mSpeed* a zero.

```
protected void InitGameObject() {
    mVelocityDir = Vector2.Zero;
    mSpeed = 0f;
}
```

Fig.73 – InitGameObject();

E declaramos uma variável que altera a posição do objeto, quando as variáveis "*mVelocityDir*" e "*mSpeed*" variam.

```
virtual public void Update()
{
    mPosition += (mVelocityDir * mSpeed);
}
```

Fig.74

Adicionamos de seguida, *getters* e *setters* que vão permitir a outras classes fazer alterações às variáveis declaradas.

```
public Vector2 InitialFrontDirection
{
    get { return mInitFrontDir; }
    set
    {
        float len = value.Length();
        if (len > float.Epsilon) // If the input vector is well defined
            mInitFrontDir = value / len;
        else
            mInitFrontDir = Vector2.UnitY;
    }
}
```

Fig.75 - A função InitialFrontDirection(), que vai permitir obter a direção inicial.

```
public Vector2 FrontDirection {
    get
    {
        return ShowVector.RotateVectorByAngle(mInitFrontDir, RotateAngleInRadian);
    }
    set
    {
        float len = value.Length();
        if (len > float.Epsilon)
        {
            value *= (1f / len);
            double theta = Math.Atan2(value.Y, value.X);
            mRotateAngle = -(float)(theta - Math.Atan2(mInitFrontDir.Y, mInitFrontDir.X));
        }
    }
}
```

Fig.76 - A função FrontDirection() que irá retornar o valor de rotação do objeto.

```
public Vector2 Velocity
{
    get { return mVelocityDir * Speed; }
    set
    {
        mSpeed = value.Length();
        if (mSpeed > float.Epsilon)
            mVelocityDir = value / mSpeed;
        else
            mVelocityDir = Vector2.Zero;
    }
}

public float Speed {
    get { return mSpeed; }
    set { mSpeed = value; }
}
```

Fig.77 - A função Velocity(), que obtém a velocidade, e Speed, que obtém a rapidez.

```
public Vector2 VelocityDirection
{
    get { return mVelocityDir; }
    set
    {
        float s = value.Length();
        if (s > float.Epsilon)
        {
            mVelocityDir = value / s;
        }
        else
            mVelocityDir = Vector2.Zero;
    }
}
```

Fig.78 - A função VelocityDirection(), que obtém a direção da velocidade.

Para podermos implementar a classe **GameObject** e as suas funcionalidades, tivemos que modificar a classe **GameState**.

Começamos por implementar três variáveis, uma para a posição do objeto inicial, que será o foguete, e uma para a seta.

```
public class GameState
{
    Vector2 kInitRocketPosition = new Vector2(10, 10);
    // Rocket support
    GameObject mRocket;
    // The arrow
    GameObject mArrow;
```

Fig. 79 – Classe GameState;

Inicializamos as variáveis num construtor já com valores predefinidos.

```
public GameState()
{
    mRocket = new GameObject("Rocket", kInitRocketPosition, new Vector2(3, 10));

    mArrow = new GameObject("Arrow", new Vector2(50, 30), new Vector2(10, 4));
    mArrow.InitialFrontDirection = Vector2.UnitX; // Inicialmente aponta para a posição x
}
```

Fig.80 – Função GameState();

Modificamos depois a função ***UpdateGame***() para que seja possível controlar o foguete, e também para a seta apontar para o foguete.

A função vai permitir controlar o movimento do foguete com as setas do teclado (da esquerda e da direita) e a velocidade do foguete (W para acelerar e S desacelerar/andar para trás) para além de colocar o foguete na posição inicial sempre que colida com a borda da janela de jogo

```
public void UpdateGame()
{
    #region Step 3a. Controla e mexe o foguete
    mRocket.RotateAngleInRadian +=
        MathHelper.ToRadians(InputWrapper.ThumbSticks.Right.X);

    mRocket.Speed += InputWrapper.ThumbSticks.Left.Y * 0.1f;

    mRocket.VelocityDirection = mRocket.FrontDirection;

    if (Camera.CollidedWithCameraWindow(mRocket) !=
                Camera.CameraWindowCollisionStatus.InsideWindow)
    {
        mRocket.Speed = 0f;
        mRocket.Position = kInitRocketPosition;
    }

    mRocket.Update();
    #endregion
```

Fig.81 – Função UpdadeGame();

E direciona a seta para o foguete, encontrando o vetor que é formado pela posição do foguete e a posição da seta em si, aplicando esse vetor na sua posição frontal.

```
Vector2 toRocket = mRocket.Position - mArrow.Position;
mArrow.FrontDirection = toRocket;
#endregion
```

Fig.82

De seguida, para as imagens aparecerem no ecrã, alteramos a função ***DrawGame***(), adicionando os objetos do foguete e da seta e fazemos o *update* aos controlos.

```
public void DrawGame()
{
    mRocket.Draw();
    mArrow.Draw();

    FontSupport.PrintStatus("Rocket Speed(LeftThumb-Y)=" + mRocket.Speed +
        "  VelocityDirection(RightThumb-X):" + mRocket.VelocityDirection, null);

    FontSupport.PrintStatusAt(mRocket.Position, mRocket.Position.ToString(), Color.White);
}
```

Fig.83 – Função DrawGame();

Na realização deste exercício podemos observar e analisar como fazer com que um objeto se mova livremente pela janela do jogo e ainda permitir ajustar a sua velocidade e a sua direção. Na resolução deste mesmo exercício não encontramos qualquer tipo de erros, apenas alguns de sintaxe.

Projeto Chaser Object

Este projeto tem como objetivo demostrar como controlar um objeto (foguete) usando dois controlos diferentes, um para controlar a velocidade (W e S) e outro para controlar a rotação (seta esquerda e seta direita).

O objetivo do jogo consiste no lançamento de um projétil (cobra) e dirigir o foguete de maneira a evitar ser atingido.

Para implementar estas alterações, primeiramente criamos a classe **ChaserGameObject**.

Esta classe vai receber como herança a classe **GameObject**, de maneira a podermos utilizar as suas potencialidades.

Criamos três variáveis, *mTarget*, que representa o alvo a ser atingido, *mHitTarget*, uma variável booleana que determinara se o alvo foi atingido ou não e *mHomeInRate* determina o quão rápido o projétil se dirige para o alvo.

```
public class ChaserGameObject : GameObject
{
    // Representa o alvo
    protected TexturedPrimitive mTarget;
    // Se atingiu o alvo. Sim/Não
    protected bool mHitTarget;
    // Velocidade a que se aproxima do alvo
    protected float mHomeInRate;
```

Fig. 84 – Variáveis na Classe ChaseGameObject;

Criamos de seguida um construtor que vai inicializar as variáveis com valores que desejamos.

```
public ChaserGameObject(String imageName, Vector2 position, Vector2 size, TexturedPrimitive target)
        : base(imageName, position, size, null)
{
    Target = target;
    mHomeInRate = 0.05f; // homes in 5% at each update
    mHitTarget = false;

    mSpeed = 0.1f;
}
```

Fig.85 – Construtor ChaserGameObject();

Depois, para o projétil ajustar a sua direção de acordo com o movimento do foguete (controlado pelo jogador), criamos a função **ChaseTarget**().

A função vai verificar se existiu colisão entre os dois objetos (alvo e projétil), se a colisão é inexistente, então é calculado o ângulo entre a direção frontal e a direção do alvo, que depois

de ser processado, o produto é lado para determinar se o alvo se dirige na direção do relógio ou sentido contrário do relógio. Para realizar o cálculo foi necessário usar *Epsilon* pois este usa o menor número positivo possível com vírgula flutuante.

```
if (!mHitTarget)
{
    #region Calculate angle
    Vector2 targetDir = mTarget.Position - Position;
    float distToTargetSq = targetDir.LengthSquared();

    targetDir /= (float) Math.Sqrt(distToTargetSq);
    float cosTheta = Vector2.Dot(FrontDirection, targetDir);
    float theta = (float)
    Math.Acos(cosTheta);
    #endregion

    #region Calculate rotation direction
    if (theta > float.Epsilon)
    { // not quite aligned ...
        Vector3 fIn3D = new Vector3(FrontDirection, 0f);
        Vector3 tIn3D = new Vector3(targetDir, 0f);
        Vector3 sign = Vector3.Cross(tIn3D, fIn3D);

        RotateAngleInRadian += Math.Sign(sign.Z) * theta * mHomeInRate;
        VelocityDirection = FrontDirection;
    }
```

Fig.86

Para podermos aceder ou modificar as variáveis instanciadas fora da classe, adicionamos *getters* e *setters*.

```
public float HomeInRate { get { return mHomeInRate; } set { mHomeInRate = value; } }
public bool HitTarget { get { return mHitTarget; } }
public bool HasValidTarget { get { return null != mTarget; } }
public TexturedPrimitive Target
{
    get { return mTarget; }
    set
    {
        mTarget = value;
        mHitTarget = false;
        if (null != mTarget)
        {
            FrontDirection = mTarget.Position - Position;
            VelocityDirection = FrontDirection;
        }
    }
```

Fig.87

Para podermos utilizar e inserir no jogo as funcionalidades criadas na classe **ChaserGameObject**, tivemos que modificar a classe **GameState**.

Começamos por criar duas variáveis, uma para o projétil, e outra que vai permitir seguir o número de vezes que o projétil atingiu ou falhou o alvo.

```
public class GameState
{

    ChaserGameObject mChaser;

    // Simple game status
    int mChaserHit, mChaserMissed;
```

Fig.88 – Variáveis GameState;

No construtor, inicializamos o projétil com valores específicos e colocamos os valores de acertos ou falhas (*hit or miss*) a zero.

```
public GameState()
{
    mChaser = new ChaserGameObject("Chaser", Vector2.Zero, new Vector2(6f, 1.7f), null);
    mChaser.InitialFrontDirection = -Vector2.UnitX; // inicialmente na direção x negativo
    mChaser.Speed = 0.2f;

    // Inicializa o status do jogo
    mChaserHit = 0;
    mChaserMissed = 0;
```

Fig.89

De seguida modificamos a função **Update**().

Adicionamos um comportamento à função que irá verificar se o alvo é válido (se existe). Se isso se confirmar, então o projétil vai ser acionado e perseguir o alvo. O número de acertos ou falhas baseia-se caso o alvo acerte no alvo será um ponto ganho, e se acertar nas bordas da janela é um ponto falhado.

```
#region Step 3: Check/launch the chaser?
if (mChaser.HasValidTarget)
{
    mChaser.ChaseTarget();

    if (mChaser.HitTarget)
    {
        mChaserHit++;
        mChaser.Target = null;
    }

    if (Camera.CollidedWithCameraWindow(mChaser) !=
            Camera.CameraWindowCollisionStatus.InsideWindow)
    {
        mChaserMissed++;
        mChaser.Target = null;
    }
}

if (InputWrapper.Buttons.A == ButtonState.Pressed)
{
    mChaser.Target = mRocket;
    mChaser.Position = mArrow.Position;
}
#endregion
```

Fig.90

De seguida modificamos a função **Draw**(), que vai imprimir o projétil no ecrã e também as estatísticas (*hit/miss*).

```
public void DrawGame()
{
    mRocket.Draw();
    mArrow.Draw();
    if (mChaser.HasValidTarget)
        mChaser.Draw();

    // Print out text messsage to echo status
    FontSupport.PrintStatus("Chaser Hit=" + mChaserHit + "   Missed=" + mChaserMissed, null);
}
```

Fig.91

Finalizando o ultimo exercício e por sua vez o capítulo 4 do livro "*Learn 2D Game Development with C#",* conseguimos "correr" o jogo sem problemas, e com isto aprender novas mecânicas que podemos a vir inserir nos nossos próprios jogos.

Conclusão

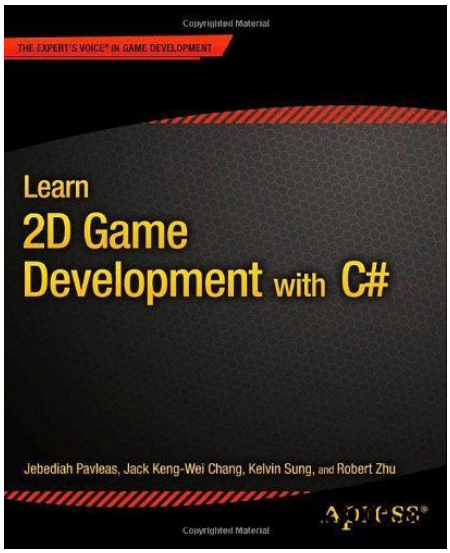
Concluímos que através do livro *Learn 2D Game Development with C-sharp*, fomos capazes de realizar grande parte dos exercícios exigidos, infelizmente na resolução dos mesmos foram encontrados imensos erros, dos quais alguns não fomos capazes de os solucionar.

Estes exercícios permitiram-nos criar e explorar um jogo/programa básico usando um projeto ***MonoGame***. Ao usar as funções pré-definidas da *framework* foi possível criar um programa que permite ao utilizador interagir com objetos gráficos no ecrã como também explorar certas mecânicas da própria *framework*.

Podemos observar e concluir que as funções *Draw/Update* são basicamente o núcleo de qualquer tipo de jogo, sendo eles programados em C#, java ou etc.

Com a realização dos exercícios até ao capítulo 4, conseguimos concluir que muitas das mecânicas implementadas nos mesmos poderão e serão bem empregues (adaptados) nos nossos jogos, caso os façamos.

Por fim, concluímos que através da ***FrameWork MonoGame*** e do ***VisualStudio***, ambos são excelentes ferramentas que tornam possível criar imensos projetos de diversas maneiras.



Bibliografia

Jebediah Pavleas, J. K.-W. (2013). *Learn 2D Game Development with C#* (Vol. 1). U.S.A: Apress.

Micorsoft. (s.d.). *msdn Microsoft*. Obtido de Microsoft : https://msdn.microsoft.com/en-us/library/microsoft.xna.framework.game.initialize.aspx

Team, M. (2 de Setembro de 2009). *MonoGame* . Obtido de MonoGame : http://www.monogame.net/