3/30/2017

Relatório

Capítulos 3 e 4

Fernando Nogueira Camilo Nº13233
André Rodrigues Nº13231

ESCOLA SUPERIOR DE TECNOLOGIA IPCA

Técnicas de desenvolvimento de jogos

André Rodrigues Nº13231
Fernando Camilo Nº13233

Introdução

Relatório

Capitulo 3

Este relatório é referente ao Livro "Learn 2D Game Development with C#" dos autores Kelvin Sung, Jebediah Pavleas, Rob Zhu, Jack Keng-Wei Chang. Abrange os capítulos 3 e 4 deste mesmo livro e tem como objetivo familiarizar os alunos com técnicas e padrões frequentemente vistos na codificação de um videojogo.

No terceiro capitulo a ideia é dotar os alunos da capacidade de usar a framework do monogame, na linguagem de C#, de forma a puderem controlar o tamanho da janela e colocar a janela de jogo em full screen, usar "assests" visuais de maneira a criar objetos de jogo e com isto estados de jogo.

No quarto capitulo serão abordados processos de colisão de objetos, e as redes matemáticas que as permitem, tal como comportamentos mais complexos que já poderiam pertencer a simples objetos de um jogo comercial, como a perseguição de um objeto pelo de outro apos este te mudado a sua direção para conseguir alinhar-se com o seu objeto alvo.

Por ultimo vão ser também indicadas boas praticas a ter, como aumentar a abstração dos objetos para estes possuírem um grande numero de características mas que foram herdadas de outras classe e assim ter um objeto da união de classe ao em vez de um que é formado por fragmentos de varias classe, para assim ter um código mais legível, portável e principalmente, um código capaz de ser atualizado, melhorando um qualquer sistema ou acrescentando comportamentos mais modernos e atualizados com um novo possível contexto de jogo.

IPCA INSTITUTO POLITÉCNICO DO CÁVADO E DO AVE
Licenciatura em Engenharia em desenvolvimento de jogos de digitais
2016/2017

ESCOLA SUPERIOR DE TECNOLOGIA IPCA

Técnicas de desenvolvimento de jogos

André Rodrigues Nº13231
Fernando Camilo Nº13233

Capitulo 3

Começando então pelo controlo da janela de jogo, declarou-se então duas variáveis (kWindowWidth e kWindowHeight) constantes de forma a serem acedidas por todo o código e os valores atribuídos a essas variáveis são utilizados para definir um tamanho preferencial da janela. Depois adiciona-se três condições à função Update de forma a ser possível sair do jogo, colocar em full screen, caso não esteja em full screen, e voltar a colocar em modo janela com o tamanho definido anteriormente.

Defina assim a capacidade para modificar a janela de jogo, verifica-se agora a necessidade de usar objetos visuais, desde texturas à arte do jogo, para este ser uma aplicação interativa. É necessário com isto criar uma classe (Textured Primitive) que permita a manipulação dos "assests" visuais. Primeiro é necessário modificar as três variáveis (SpriteBatch, ContentManeger e GraphicsDeviceManeger) presentes na classe Game1 (classe que funciona como Main) para estáticas, para assim terem uma posição na memoria fixa e publicas para serem acedidas por outras classes do programa. No inicio da classe TexturePrimitive é necessário declarar três variáveis, Texture2D mImage para carregar as imagens externas, como JPG, a Vector2 mPosition que controla a posição central da imagem carregada anteriormente e por ultimo a variável do tipo Vector2 designada por mSize que regula o tamanho da imagem. Esta variável tem que ser inicializadas depois no método da classe em questão. Ainda dentro da classe TexturePrimitive tem que ser criado o método Update que modifica o tamanho e posição das imagens mediante os valores recebidos como argumento da função. Por ultimo o método draw que é responsável por criar um background retangular para cada imagem, declara-se um objeto do tipo Rectangle com a posição e tamanho definido no inicio da classe e usando a propriedade Draw do SpriteBatch que recebe como argumentos a imagem, o retângulo criado e a cor do background consegue-se então criar um fundo para cada imagem.

Para usar a classe TexturePrimitive é necessário declara três variáveis na classe Game1, uma que define o numero de objetos que vamos usar (kNumObjects), um array (mGraphicsObjects) do tipo TexturePrimitive com o tamanho definido na variável anterior e por ultimo a variável com o índice (mCurrentIndex) do objeto a ser usado no array e neste exemplo usamos quatro imagens, portanto o array terá um tamanho de quatro e iniciar no índice zero. Para usar o array anteriormente declarado utiliza-se o método LoadContent e preenche-se cada posição do array com um objeto do tipo TexturePrimitive que então recebe o nome da imagem que se pretende que seja carregado, posição e tamanho da imagem. Por ultimo utiliza-se o método draw com recurso a um foreach para colocar no ecrã as quatro imagens e o método update para permitir que se troque as imagens que podemos controlar mediante o pressionar da tecla X do gamepad (tecla j no teclado).

ESCOLA SUPERIOR DE TECNOLOGIA IPCA

Técnicas de desenvolvimento de jogos

André Rodrigues Nº13231
Fernando Camilo Nº13233

Agora é necessário criar uma classe chamada camera que permite criar um sistema de coordenadas cartesiano de para controlo do aspeto da aplicação e padronizar as posições e tamanhos dos objetos de jogo independente da resolução do ecrã em que é apresentada.

Portanto esta classe tem como objetivo transformar um objeto em pixel space (em que o inicio do sistema de coordenadas é no canto superior esquerdo da janela e o eixo dos y esta invertido) para um sistema de coordenadas cartesiano. Para conseguirmos obter isto temos que primeiro colocar a janela de pixel space para a origem de um referencial cartesiano, isto é conseguido subtraindo o valor da altura da janela (Hc) pela posição da janela no eixo do y (Yc) e a largura da janela (Wc) pela posição relativa ao eixo do x (Xc) e assim colocamos a janela na origem do referencial depois para redimensionar a escala é necessário fazer o produto do Wc pela razão entre a nova largura (Wp) e a largura inicial (Wc) e repetir o processo para a altura mas invertendo sentido do eixo dos y.

Px= (Cx - Xc) *(Wp/Wc)

Py= Hp - (Cy - Yc) *(Hp/Hc)

Em que os valores obtidos são as coordenadas no pixel space.

Para criar a classe então começamos por defini-la como estática para ter apenas uma instancia, depois declaramos três variáveis, uma com as coordenadas da origem da janela, neste exemplo iniciada com o vetor (0,0), outra com a largura da janela e por ultimo uma variável que define o ratio entre a camera e a janela em pixel.

Depois cria-se um método que devolve o ratio de redimensionamento (Wp/Wc) sendo o Wp representado Game1.sGraphics. PreferredBackBufferWidth e o Wc é o valor da largura da janela atribuído anteriormente depois criou-se um método que define a posição e tamanho da janela camera(setcameraWindow), sendo que este método recebe os valores, portanto permite alterar os valores da posição e tamanho. Os dois últimos métodos da classe camera devolvem a posição em pixel space e o ultimo converte a área retangular para pixel space. Para usar a class basta chamar a função setcamerawindow dentro do loadcontent da classe game1 e colocar a origem para (10,20) e a largura para 100. Por ultimo é necessário converter todos “assets” usados para pixel space e isto é conseguido no método draw da classe TexturePrimitive ao chamar o método ComputePixelRectangle da classe camera.

ESCOLA SUPERIOR DE TECNOLOGIA IPCA

Técnicas de desenvolvimento de jogos

André Rodrigues Nº13231
Fernando Camilo Nº13233

Define SoccerBall Class:

-Função que herda todas a funcionalidades da class TexturedPrimitive e que tem como objetivo passar a informação sobre o tamanho e posição da bola.

-Variáveis:

-**MDeltaPosition**: Variável usada para representar as mudanças na posição do objeto (SoccerBall);

-**Radius**: Variável para mudar e receber o tamanho do raio da bola;

-Funções:

-**SoccerBall**: Passa a informação do tamanho e posição da bola para a class TexturedPrimitive, com o uso da variável mDeltaPosition.

-**Update**: Define a localização da bola após colidir, também com o uso da variável mDeltaPosition.

Using SoccerBall Class:

-Primeiro declaramos uma variável do tipo SoccerBall chamada mBall e outra variável chamada mUWBLogo do tipo TexturedPrimitive.

-De seguida damos *Load* da bola e do Logo, já definindo as suas posições, tamanhos e imagens, com o uso da função LoadContent () na class Game1.

-Por fim com o uso da função Draw são desenhados os objetos na janela (SoccerBall e o Logo), tal como são escritas as informações sobre a posição destes objetos no canto superior esquerdo.

Adding TexturedPrimitive collision detection support:

-Com a adição da função PrimitivesTouches conseguimos detetar se há sobreposição de objetos. Isto é feito através da comparação da distância entre a posição de dois objetos e metade da sua largura.

ESCOLA SUPERIOR DE TECNOLOGIA IPCA

Técnicas de desenvolvimento de jogos

André Rodrigues Nº13231
Fernando Camilo Nº13233

Creating BasketBall class:

-Função que herda funcionalidades da class TexturePrimitive, e define as condições essências das bolas de basket (onde dão load e quando explodem).

-**Variáveis:**

-KIncreaseRate – rácio de aumento do tamanho da bola.

-KInitSize – tamanho inicial da bola.

-KFinalSize - tamanho máximo da bola, quando explode.

-Funções:

-BasketBall (): base("BasketBall"): inicializa a posição da bola num ponto aleatório assim como o seu tamanho usando como base a função disponibilizada por TexturePrimitive.

-**UpdateAndExplode:** aumenta o tamanho da bola até esta estourar.

Creating the game state object:

-Variáveis:

-KHeroSize: Tamanho do herói;

-KHeroPosition: Posição inicial em que o herói aparece;

-MTotalBBallCreated: número de bolas de basket geradas;

-KBballMSecInterval: Tempo de diferença entre spawn de bolas de basket;

-MBBallMissed: Bolas de basket que explodiram;

-MBBallHit: Bolas apanhas pelo herói;

-Funções:

-**MyGame ():** desenha o herói e as bolas de Basket;

-**UpdateGame ():**

-Vai atualizando o jogo até chegar ao fim;

-Verifica o número de bolas explodidas e atualiza o score;

-Verifica o número de bolas capturadas;

André Rodrigues Nº13231
Fernando Camilo Nº13233

-Gera novas bolas, dependendo do tempo atual e do tempo da última da bola criada;

-E verifica a condição final de jogo, se o jogador ganhou ou perdeu;

-DrawGame (): desenha o herói, as bolas, e a imagem correspondente a ganhar ou perder o jogo. Também mostra o score do jogo.

Modifying Game1 to support the game state:

-Apenas é necessário conectar a class MyGame à class Game1.

André Rodrigues Nº13231
Fernando Camilo Nº13233

Conclusão

Capitulo 3

Este capitulo serviu para nos integrarmos nos processos de fabrico de simples objetos de jogo e aquisição de boas praticas para nos guiar nos processos de execução do mesmo, indo de processos mais específicos de uma plataforma como manipular o tamanho da janela de jogo e colocar essa janela em "fullscreen", a características mais universais da criação de um videojogo como a área de jogo que representada por camera de jogo e definida por coordenadas que foi também um conceito revisto neste capitulo, sendo que também foi possível verificar um processo para prestar os objetos de jogo com comportamentos simples, como movimento ou "respawn". Com isto as mais dificuldades sentidas prenderam-se mais com questões técnicas como incompatibilidades de versões de "frameworks" e "IDE" usados, mas foi estabelecida de uma base sólida de para avançar na complexidade de processos e procedimentos que vão estar presentes no próximo capitulo.

ESCOLA SUPERIOR DE TECNOLOGIA IPCA

Técnicas de desenvolvimento de jogos

André Rodrigues Nº13231
Fernando Camilo Nº13233

Introdução

Capitulo 4

Com este trabalho foi nos dado a experiência de trabalhar com alguns, dos muitos, conceitos básicos dos jogos.

Com o suporte da ferramenta MonoGame e do livro "Learn 2D Game Development in C#" de Jebediah Pavleas, Jack Keng-Wei Chang, Kelvin Sung e Robert Zhu, mais especificamente o capítulo 4 deste, pudemos trabalhar com Rotações, Vetores, Front Direction, Game Objects e Chasers.

Para mostrar a nossa compreensão perante estes conceitos apresentamos mais abaixo um resumo de cada parte, com o seu devido código.

ESCOLA SUPERIOR DE TECNOLOGIA IPCA

Técnicas de desenvolvimento de jogos

André Rodrigues Nº13231
Fernando Camilo Nº13233

Rotações

Como introdução ao capitulo somos apresentados a trabalhar com rotações que irá nos facilitar a criar animações mais naturais.

Começamos então por modificar a classe TexturedPrimitive, ao declarar uma variável (mRotateAngle) do tipo float que irá servir para guardar os valores das rotações das texturas, em radianos, e logo a seguir inicializamos a variável no construtor a zero, para que o objeto criado não sofra logo uma rotação após dar Load.

```
protected float mRotateAngle; // In radians, clockwise rotation
```

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

    mRotateAngle = 0f;//inicialização da variável para 0
}
public TexturedPrimitive(String imageName)
{
    mRotateAngle = 0f;
}
```

Criamos um getter e setter para a variável mRotateAngle, para ser facilmente modificada.

```
/*Getter e Setter da variável que faz com que certo objeto rode*/
public float RotateAngleInRadian
{
    get { return mRotateAngle; }
    set { mRotateAngle = value; }
}
```

Modificamos a função Update da classe TexturedPrimitive para ir dando update à variável de rotação, sendo que isto foi feito ao adicionar outro parâmetro a especificar o ângulo de rotação.

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

Técnicas de desenvolvimento de jogos

André Rodrigues Nº13231
Fernando Camilo Nº13233

De seguida modificamos a função Draw para converter o tamanho e dimensão do objeto para o pixel space, calcular o centro das imagens dos objetos para definir como ponto de rotação e desenhar as texturas com o uso da função SpriteBatch.Draw da classe Game1:

```
virtual public void Draw()
{
    // Define location and size of the texture
    Rectangle destRect = Camera.ComputePixelRectangle(mPosition, mSize);
    // Define the rotation origin
    Vector2 org = new Vector2(mImage.Width / 2, mImage.Height / 2);
    // Draw the texture
    Game1.sSpriteBatch.Draw(
    mImage,
    destRect, // Area to be drawn in pixel space
    null,
    Color.White,
    mRotateAngle, // Angle to rotate (clockwise)
    org, // Image reference position
    SpriteEffects.None, 0f);
}
```

a) Conversão das coordenadas colocadas pelo o utilizador para pixel space com a função ComputePixelRectangle criada na classe Camera, passa o tamanho e posição da textura para a função e depois armazena os resultados na variável destRect;

```
// Define location and size of the texture
Rectangle destRect = Camera.ComputePixelRectangle(mPosition, mSize);
```

b) Cálculo do ponto central da textura para definir o ponto de rotação, sendo que as informações das coordena das irão ser guardadas na variável org;

```
// Define the rotation origin
Vector2 org = new Vector2(mImage.Width / 2, mImage.Height / 2);
```

C) com o uso da informação nas três variáveis já referidas (mRotateAngle, destRect e org) são desenhadas as texturas;

```
// Draw the texture
Game1.sSpriteBatch.Draw(
mImage,
destRect, // Area to be drawn in pixel space
null,
Color.White,
mRotateAngle, // Angle to rotate (clockwise)
org, // Image reference position
SpriteEffects.None, 0f);
```

Técnicas de desenvolvimento de jogos

André Rodrigues Nº13231
Fernando Camilo Nº13233

Depois da classe TexturedPrimitive estiver feita podemos modificar a classe GameState (MyGame). Adicionamos as variáveis mBall, mUWBLogo e mWorkPrim ligadas à classe TexturedPrimitive e inicializadas num construtor.

```
// Work with TexturedPrimitive
TexturedPrimitive mBall, mUWBLogo;
TexturedPrimitive mWorkPrim;
```

```
// 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;
```

Depois alteramos a função UpdateGame para ser possível alterar qual objeto queremos ir mexendo.

```
/*Escolhe qual o objeto que vai mexer*/
#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

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

Por ultimo alteramos a função DrawGame para desenhar os objetos da TexturedPrimitive e enviar os seus ângulos de rotação através da função FontSupport.

```
mBall.Draw();
FontSupport.PrintStatusAt(mBall.Position,mBall.RotateAngleInRadian.ToString(),Color.Red);
mUWBLogo.Draw();
FontSupport.PrintStatusAt(mUWBLogo.Position,mUWBLogo.Position.ToString(),Color.Black);
FontSupport.PrintStatus("A-Soccer B-Logo LeftThumb:Move RightThumb:Scale X/Y:Rotate",null);
```

Técnicas de desenvolvimento de jogos

André Rodrigues Nº13231
Fernando Camilo Nº13233

Vetores

Nesta parte do capítulo entendemos o funcionamento dos vetores nos jogos, que são usados principalmente para representar o deslocamento e direção de um objeto.

Começamos então por criar a classe ShowVector e adicionamos as variáveis sImage do tipo Texture2D e KLenToWidthR do tipo float.

```
protected static Texture2D sImage = null; // Singleton for the class
private static float kLenToWidthRatio = 0.2f;
```

Adicionamos uma função static chamada LoadImage para dar Load à variável da imagem com a textura correta, que neste caso é a seta;

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

Adicionamos uma função DrawPointVector que irá desenhar o vetor para a janela de jogo (game window), para fazer isto a função recebe 2 argumentos do tipo Vector2, sendo o primeiro a posição inicial e o segundo a direção.

a) Usar a função LoadImage, criada anteriormente, para dar load à imagem do vetor para ficar pronta para ser desenhada;

```
static public void DrawPointVector(Vector2 from, Vector2 dir)
{
    LoadImage();
}
```

b) Depois é necessário calcular o ângulo de rotação para a imagem, que se encontra entre o vetor e o eixo dos X, para isto é necessário dividir o vetor dado com o comprimento, normalizando assim o vetor, agora que o comprimento do vetor tem o mesmo tamanho da hipotenusa e tem de valor 1, sabemos que o valor de X do vetor é igual à adjacente ao ângulo e o valor de Y é igual ao lado oposto ao ângulo, desta maneira usamos o arccos do valor de X do vetor para descobrir o angulo de rotação (angulo de rotação= arccos(Valor de X)) e por último descobrimos o sinal do valor de Y para indicar para que lado irá rodar, se positivo irá rodar para o lado contrário ao sentido dos ponteiros do relógio, mas caso seja negativo irá rodar para o lado do sentido dos ponteiros do relógio ;

Técnicas de desenvolvimento de jogos

André Rodrigues Nº13231
Fernando Camilo Nº13233

```
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;
     }
}
```

c) agora falta nos apenas desenhar o vetor, fazemos isso ao chamar a função Game1.sSpriteBatch. Draw e ao passar os valores da rotação do ângulo, a imagem e a área de jogo (hitbox do vetor), que pode ser calculada através da posição e tamanho do vetor.

```
// Define location and size of the texture to show
Vector2 size = new Vector2(length, kLenToWidthRatio * length);
Rectangle destRect = Camera.ComputePixelRectangle(from, size);
// destRect is computed with respect to the "from" position
// on the left side of the texture.
// We only need to offset the reference
// in the y from top left to middle left.
Vector2 org = new Vector2(0f, ShowVector.sImage.Height / 2f);
Game1.sSpriteBatch.Draw(ShowVector.sImage, destRect, null, Color.White,
theta, org, SpriteEffects.None, 0f);
```

d) por ultimo a parte que iremos inserir escreve a direção e posição do vetor, sendo isto possível apenas com a ajuda da função PrintStatus que se encontra na classe FontSupport;

```
String msg;
msg = "Direction=" + dir + "\nSize=" + length;
FontSupport.PrintStatusAt(from + new Vector2(2, 5), msg, Color.Black);
```

Criamos uma função chamada DrawFromTo que desenha um vetor entre 2 pontos, esta função recebe 2 argumentos a posição do ponto inicial e a posição do ponto de chegada, para que a função funcione usamos a função anteriormente criada (DrawPointVector)

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

Por ultimo criamos da função RotateVectorByAngle que recebe um vetor e o seu angulo de rotação, basicamente a função serve para rodar o vetor recebido com o angulo recebido e depois dá return a esse vetor gerado pela rotação.

```
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);
}
```

Depois da classe ShowVector estiver feita podemos modificar a classe GameState (MyGame).

Começamos por declarar os vetores e os pontos que vamos pretender controlar (mPa, mPb, mPx, mPy). Isto inclui um TexturedPrimitive mCurrentLocator que vai ser responsável por selecionar o vetor que que remos controlar.

```
// Size of all the positions
Vector2 kPointSize = new Vector2(5f, 5f);
// Work with TexturedPrimitive
TexturedPrimitive mPa, mPb; // The locators for showing Point A and Point B
TexturedPrimitive mPx; // to show same displacement can be applied to any position
TexturedPrimitive mPy; // To show we can rotate/manipulate vectors independently
Vector2 mVectorAtPy = new Vector2(10, 0); // Start with vector in the X direction;
TexturedPrimitive mCurrentLocator;
```

Modificamos a classe GameState para inicializar os vetores com uma imagem, posição, tamanho e nome e ainda indicamos quais dos pontos queremos controlar primeiro ao igualar algum dos vetores à variável mCurrentLocator.

```
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;
}
```

Técnicas de desenvolvimento de jogos

André Rodrigues Nº13231
Fernando Camilo Nº13233

Modificamos a função UpdateGame para que fosse possível movimentar, rodar os vetores, aumenta-los ou trocar o vetor pretendido. Desta maneira podemos testar as funções criadas em ShowVector.

```
if (InputWrapper.Buttons.A == ButtonState.Pressed)
    mCurrentLocator = mPa;
else if (InputWrapper.Buttons.B == ButtonState.Pressed)
    mCurrentLocator = mPb;
else if (InputWrapper.Buttons.X == ButtonState.Pressed)
    mCurrentLocator = mPx;
else if (InputWrapper.Buttons.Y == ButtonState.Pressed)
    mCurrentLocator = mPy;
// Change the current locator position
mCurrentLocator.Position +=
InputWrapper.ThumbSticks.Right;
// Left thumbstick-X rotates the vector at Py
float rotateYByRadian = MathHelper.ToRadians(
InputWrapper.ThumbSticks.Left.X);
// Left thumbstick-Y increase/decrease the length of vector at Py
float vecYLen = mVectorAtPy.Length();
vecYLen += InputWrapper.ThumbSticks.Left.Y;
// Compute the rotated direction of vector at Py
mVectorAtPy = ShowVector.RotateVectorByAngle(mVectorAtPy, rotateYByRadian);
mVectorAtPy.Normalize(); // Normalize vectorAtPy to size of 1f
mVectorAtPy *= vecYLen; // Scale the vector to the new size
```

Por fim mudamos a função DrawGame para desenhar os vetores no ecrã. Sendo que os vetores são desenhados através da class ShowVector.

```
// Drawing the vectors
Vector2 v = mPb.Position - mPa.Position; // Vector V is from Pa to Pb
                                        // Draw Vector-V at Pa, and Px
ShowVector.DrawFromTo(mPa.Position, mPb.Position);
ShowVector.DrawPointVector(mPx.Position, v);
// Draw vectorAtPy at Py
ShowVector.DrawPointVector(mPy.Position, mVectorAtPy);
mPa.Draw();
mPb.Draw();
mPx.Draw();
mPy.Draw();
// Print out text message to echo status
FontSupport.PrintStatus(
"Locator Positions: A=" + mPa.Position +
" B=" + mPb.Position,
null);
```

ESCOLA SUPERIOR DE TECNOLOGIA IPCA

Técnicas de desenvolvimento de jogos

André Rodrigues Nº13231
Fernando Camilo Nº13233

Font Diretion

No que toca à direção em que a personagem de jogo está virada, é um conceito simples que esta relacionado com os elementos vetoriais do jogo.

O objetivo deste projeto é criar um foguete capaz de se movimentar em todas as direções e que dispara projeteis de forma a acertar numa abelha que é colocada na janela de jogo de forma aleatória. Para conseguir isto é necessário então criar uma variável do tipo "TexturedPrimitive" designada de mRocket e uma outra do tipo Vector2 (mRocketInitDirection) que define a direção inicial do foguete. Para o projétil é necessário uma variável também do tipo TexturePrimitive (mNet), uma do tipo bool para verificar se esta a voar ou não, uma do tipo float para a rapidez(mNetSpeed) e uma variável do tipo vetorial para a velocidade do projétil, enquanto que a abelha necessita apenas de uma variável TexturedPrimitive(mInsect) e uma bool para verificar o estado em que se encontra o inseto e por ultimo são necessárias duas variáveis para estatísticas do jogo como tiros falhados e insetos "mortos".

Começando então por inicializar as variáveis referentes aos foguetes sendo que é necessário criar uma instancia da classe TexturedPrimitive utilizando a imagem denominada de "Rocket", com a posição inicial do vetor no ponto (10, 10) e com o tamanho de um vetor com as coordenadas (3,10), já no que toca à direção inicial do foguete, essa é definida pela variável "mRocketInitDirection" que toma o valor de um vetor com o sentido do eixo vertical do y (Vector2.UnitY).

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

    GameObject mArrow;

    Chaser
    // Sim
    int mChaserHit, mChaserMissed;

    /// <summary>
    /// Constructor
    /// </summary>
```

class BookExample.GameObject
GameObject class: TexturedPrimitive with behavior

Técnicas de desenvolvimento de jogos

André Rodrigues Nº13231
Fernando Camilo Nº13233

```
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; // initially pointing in the x direction

    mChaser = new ChaserGameObject("Chaser", Vector2.Zero, new Vector2(6f, 1.7f), null);
    mChaser.InitialFrontDirection = -Vector2.UnitX; // initially facing in the negative x direction
    mChaser.Speed = 0.2f;

    // Initialize game status
    mChaserHit = 0;
    mChaserMissed = 0;
}
```

No que toca ao projétil o processo é idêntico na criação de uma instancia da classe TexturedPrimitive, mas com uma imagem diferente, posição inicial de zero e um tamanho vetorial de (2, 5), depois coloca-se a variável de estado de voo como falsa ate a tecla “A” do teclado for pressionada, a velocidade é iniciada a zero e a rapidez a 0.5. já o “asset” da abelha é uma instancia da classe TexturedPrimitive que usa a imagem com o nome “Insect”, na posição inicial zero (0, 0) e com o tamanho de um vetor de (5, 5) e a variável de estado a falso, por ultimo no que a inicializações de variáveis diz respeito, as duas que guardam as estatísticas de jogo tomam como valor inicial zero.

Agora que as variáveis podem ser usadas verifica-se a necessidade de modificar a função Update atribuído o movimento do objeto de jogo ao manipulo esquerdo do comando e a rotação do foguete ao manipulo direito. Depois coloca-se uma clausula “if” para que quando o botão A seja pressionado se verifiquem modificações nas variáveis do projétil, em que a variável de estado passa a verdadeira, o projétil toma o valor de rotação do foguete, tal como a sua posição e a velocidade do projétil é igual a metade (valor da rapidez mNetSpeed) do valor do vetor inicial do foguete com o angulo de rotação já atribuído anteriormente à propriedade “RotateAngleInRadian” da “mNet”.

Por ultimo é necessário definir a abelha em termos de código, ao colocar uma condição que verifica se a abelha esta no ecrã de jogo e caso não esteja coloca-se de maneira aleatória e altera-se o estado para verdadeiro.

André Rodrigues Nº13231
Fernando Camilo Nº13233

```
public void UpdateGame()
{

    #region Control and fly the rocket
    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

    Set the arrow to point towards the rocket

    Step 3. Check/launch the chaser!
}
```

Por ultimo, no que a alterações à função Update da classe GameState diz respeito, é necessário modificar a função de forma a puder suportar interatividade entre objetos de jogo e isto é conseguido acrescentando uma condição “if” que verifica se o projétil esta em movimento, em que caso esteja é acrescentado o valor da velocidade à posição do projétil de forma a criar uma trajetória ao projeto e duas condições dentro da anterior referida (mNetInFlight == true) que verificam se o projétil toca na abelha e caso seja verdade altera-se, o estado das variáveis referentes à presença da abelha e ao estado de movimento do projétil, para “false” e acrescenta-se um à variável que guarda o numero de abelhas atingidas, já a segunda condição verifica se o projétil em movimento está dento das configuraçoes da área de jogo, em que caso não esteja aumenta-se em um valor, o que esta guardado na variável que sustem o valor dos disparos falhados (mNumMissed) e altera a variável de estado de movimento do projétil para falso.

Para os objetos aparecerem na janela de jogo acrescenta-se à função DrawGame a propriedade Draw de cada variável que represente cada objeto, mas colocando condições ao projétil, para que apareça em jogo quando a sua variável de estado seja verdadeira tal como a abelha só é “desenhada” quando “mInsectPreset” for verdadeira. Para mostrar as estatísticas de jogo usa-se a propriedade “PrintStatus” da classe FontSupport.

Agora como os projetos criados ate ao momento usam propriedades das mais variadas classes, existe a necessidade de criar uma classe que aglomera todas características importantes para um novo projeto de forma a ter um código mais “limpo” e otimizado.

André Rodrigues Nº13231
Fernando Camilo Nº13233

Esta nova classe chama-se GameObject que herda as propriedades da classe TexturedPrimitive, com três novas variáveis, mInitFrontDir que guarda a posição inicial do objeto que é sempre iniciado na direção do eixo do y, mVelocityDir e mSpeed que controlam a velocidade do objeto.

```
public class GameObject : TexturedPrimitive
{
    // Initial front direction (when RotateAngle is 0)
    protected Vector2 mInitFrontDir = Vector2.UnitY;

    // GameObject behavior: velocity
    protected Vector2 mVelocityDir; // If not zero, always normalized
    protected float mSpeed;


    /// <summary>
    /// Initialize the game object
    /// </summary>
    protected void InitGameObject() {
        mVelocityDir = Vector2.Zero;
        mSpeed = 0f;
    }
```

Agora é necessário criar um construtor que inicializa a velocidade do objeto a zero e outro que recebe as características do objeto como nome, posição e tamanho do objeto e chama o construtor referente à velocidade. Depois tem que se criar uma função que permite modificar a posição dos objetos mediante o produto da velocidade com a rapidez (mVelocityDir*mSpeed). É preciso definir os "get/set" de forma a ser possível modificar a direção e velocidade dos objetos de jogo a partir de outras classes, começando então por definir um para a direção inicial que permite obter a direção do vetor inicial e altera-la caso o vetor não seja nulo, ou seja cria uma variável float chamada len que guarda o valor do comprimento do vetor da direção inicial, depois verifica se o comprimento é superior a zero pois esta variável vai ser o denominador, esta comparação é feita com o uso da propriedade Epsilon (valor mais próximo de zero sem ser zero) do tipo float, isto porque em operações com virgula flutuante são muito mais complexas em binário que em decimal sendo assim necessário usar um numero o mais próximo possível de zero sem ser um zero absoluto, então caso o comprimento seja superior a zero divide-se pelo valor que foi obtido a partir do construtor da classe mas caso seja inferior a zero a "mInitFrontDir" toma o valor do vetor com a direção do eixo dos yy, o próximo retorna um objeto do tipo ShowVector, ou seja usa a direção inicial e aplica um angulo de rotação, sendo este angulo obtido calculando primeiro o arco tangente desse vetor que é obtido pelo construtor da classe, através da propriedade Atan2 da biblioteca "Math" e depois ao calcular a diferença entre o arco de

André Rodrigues Nº13231
Fernando Camilo Nº13233

tangente do vetor com a sua posição inicial e converte para float obtendo-se assim o angulo de rotação do foguete, continuando agora cria-se um "get/set" para a velocidade, obtendo-se a direção do vetor velocidade ao dividir esse valor obtido pela rapidez(mSpeed) que tem o valor do comprimento do vetor obtido do construtor e verifica se é superior à propriedade Epsilon, caso seja mVelocityDir toma o valor da razão entre o que é obtido do construtor pela rapidez e caso não seja passa a vetor nulo, passando agora a definir o penúltimo "get/set" para a rapidez e o ultimo para a direção do vetor velocidade que é obtido ao dividir o valor que se vai buscar a dividir por uma variável s que tomou o valor do comprimento do vetor de velocidade antes de ser modificado é então comparado a float.Epsilon e caso seja superior o valor do vetor é dividido pelo comprimento caso não seja o vetor passa ser nulo.

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

/// <summary>
/// Update the game object, by default, move by velocity
/// </summary>
virtual public void Update()
{
    mPosition += (mVelocityDir * mSpeed);
}

/// <summary>
/// Get/Set the Initial FrontDirection of the GameObject
/// </summary>
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;
    }
}
```

ESCOLA SUPERIOR DE TECNOLOGIA IPCA

Técnicas de desenvolvimento de jogos

André Rodrigues Nº13231
Fernando Camilo Nº13233

```
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));
        }
    }
}

/// <summary>
/// Get/Sets the velocity of the game object
/// </summary>
public Vector2 Velocity
{
    get { return mVelocityDir * Speed; }
    set
    {
        mSpeed = value.Length();
        if (mSpeed > float.Epsilon)
            mVelocityDir = value / mSpeed;
        else
            mVelocityDir = Vector2.Zero;
    }
}
```

IPCA INSTITUTO POLITÉCNICO DO CÁVADO E DO AVE

Licenciatura em Engenharia em desenvolvimento de jogos de digitais
2016/2017

Técnicas de desenvolvimento de jogos

André Rodrigues Nº13231
Fernando Camilo Nº13233

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

        /// <summary>
        /// Get/Sets the Direction of the velocity.
        /// Ensures vector is normalized
        /// </summary>
        public Vector2 VelocityDirection
        {
            get { return mVelocityDir; }
            set
            {
                float s = value.Length();
                if (s > float.Epsilon)
                {
                    mVelocityDir = value / s;
                }
                else
                    mVelocityDir = Vector2.Zero;
            }
        }
    }
}
```

Terminados os “get/set” é agora utilizar todas as propriedades da classe GameObject, mas é também preciso alterar a classe GameState para utilizar estas propriedades, começando então por criar três variáveis dentro desta classe, duas do tipo Gameobject para criar o foguete e a seta de jogo e uma do tipo Vector2 para definir a posição inicial do foguete e estas variáveis tem que ser iniciadas dentro do construtor da classe em que a seta tem a direção inicial do eixo dos xx.

Por ultimo há que modificar a função UpdateGame para ser possível então controlar o foguete com os “ThumbSticks” sendo que o direito controla a rotação do objeto e o esquerdo controla as modificações de velocidade do foguete e uma condição “if” que trata do “respawn” do foguete ao verificar se esta fora da área de jogo e caso esteja então a velocidade do foguete é colocada a zero e a posição passa para a posição definida inicialmente. Para colocar a seta a apontar para o foguete é preciso definir uma nova variável chamada “ToRocket” que é a diferença de valores entre a posição do foguete e da seta, tendo a propriedade “FrontDirection” da seta que ter o valor da ultima variável definida. E agora para aparecerem os objetos na área de jogo usa-se a propriedade draw de cada objeto de jogo, na função DrawGame e mensagem de texto com as estatísticas de jogo.

Técnicas de desenvolvimento de jogos

André Rodrigues Nº13231
Fernando Camilo Nº13233

```
#region Control and fly the rocket
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
```

```
    #region Set the arrow to point towards the rocket
    Vector2 toRocket = mRocket.Position - mArrow.Position;
    mArrow.FrontDirection = toRocket;
    #endregion

    Step 3. Check/launch the chaser!
}

/// <summary>
/// This is called when the game should draw itself.
/// </summary>
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);
}
```

A ultima classe que se precisa de criar para o jogo é então a de ChaserGameObject que vai corresponder a um objeto que persegue o jogador, mas é necessário falar em dois conceitos de álgebra vetorial, começando pelo produto entre os pontos de dois vetores normalizados que fornece o angulo entre os dois vetores:

V1 = (X1, Y1);

V2= (X2, Y2);

V1. V2 = V2. V1 = X1.X2 + Y1. Y2;

V1. V2 = cos(Angulo);

André Rodrigues Nº13231
Fernando Camilo Nº13233

E caso o produto seja nulo então angulo entre estes dois vetores é reto, e os vetores são perpendiculares.

E o outro conceito matemático é então o de produto cruzado de vetores, visto que este produto entre dois vetores bidimensionais, cria um terceiro vetor perpendicular que entra na terceira dimensão e que neste caso serve para ver se objeto de jogo roda no sentido dos ponteiros do relógio ou contrarrelógio e usados os dois vetores anteriores, se o produto destes for superior a zero a rotação é no sentido dos ponteiros do relógio do vetor V2 para V1 e se for inferior a zero verifica-se o inverso.

É necessário agora criar a classe ChaserGameObject que herda as características da classe GameObject, com mais três variáveis, uma que é o próprio objeto que se pretende atingir(mTarget), outra que detém os estados se foi atingido(mHitTarget) ou não e a ultima controla a velocidade a que o projétil localiza o alvo(mHomeInRate). Depois cria-se um construtor da classe para inicializar as variáveis, colocando a velocidade a 0.05, o estado a falso e a rapidez a 0.1.

```
public class ChaserGameObject : GameObject
{
    // The target to go toward
    protected TexturedPrimitive mTarget;
    // Have we hit the target yet?
    protected bool mHitTarget;
    // How rapidly the chaser home in on the target
    protected float mHomeInRate;


    /// <summary>
    /// Constructor of ChaserGameObject
    /// </summary>
    /// <param name="imageName">name of the image to be loaded as texture.</param>
    /// <param name="position">center position of the texture to be drawn</param>
    /// <param name="target">the target to chase towards</param>
    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;
    }
```

Começando agora o código para o projétil em si é preciso verificar se existe objeto a atingir, caso exista e não tenham colidido com o "chaser" é necessário depois calcular o angulo entre as duas direções dos vetores do chaser e do foguete e isto é feito através do produto dos pontos de dois vetores e depois o produto entre vetores é usado para definir se esse angulo foi formado no sentido dos ponteiros do relógio ou contra esse mesmo sentido. Verifica-se se o angulo é superior a Epsilon

e caso seja cria-se dois vetores com a direção dos dois objetos para fazer então o produto e saber em que sentido tem que se mover o projétil, apos tudo isto tem que se atribuir o valor de sinal do vetor, através da propriedade "Sign" da biblioteca Math, e fazer o produto deste valor pelo angulo obtido e pela velocidade de aquisição do alvo de forma a adicionar ao angulo que já possui o objeto de jogo, rodando assim o projétil. Agora há que criar quatro "get/set" em dois são bools que retornam o estado se atingido ou não e o do alvo altera a FrontDirection de acordo com a distancia entre objetos e atribui esse valor à "VelocityDirection".

```
public void ChaseTarget()
{
    #region Step 4a.
    if (null == mTarget)
        return;

    base.Update(); // Move the GameObject in the velocity direction
    #endregion

    Step 4b.
}

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;
        }
    }
}
```

Por ultimo é necessário alterar a classe GameState, adicionando duas variáveis, uma para o próprio projétil e outra para guardar as vezes que acerta ou falha o alvo. No construtor da classe tem que se inicializar as variáveis e o projétil esta na direção negativa do eixo dos xx. Ao acrescentarmos código à função "UpdateGame" verificamos se existe um alvo caso exista chama-se a função "ChaseTarget" do projétil e coloca-se três condições, uma que verifica se o alvo foi atingido e caso se verifique aumenta a estatística e faz "reset" ao alvo do projétil, outra condição que verifica se o projétil esta dentro da área de jogo e caso não esteja aumenta a estatística de disparos falhados e também faz "reset" e a terceira e ultima condição verifica se o Butão A do "GamePad" foi pressionado para assim definir como alvo o foguete e coloca-lo na direção da seta. Sendo necessário chamar as funções

André Rodrigues Nº13231
Fernando Camilo Nº13233

draw de cada objeto para estes aparecerem na área de jogo e também desenhar o projétil caso exista um alvo, tal como todas as estatísticas de jogo.

```
    ChaserGameObject mChaser;

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

```
    mChaser = new ChaserGameObject("Chaser", Vector2.Zero, new Vector2(6f, 1.7f), null);
    mChaser.InitialFrontDirection = -Vector2.UnitX; // initially facing in the negative x direction
    mChaser.Speed = 0.2f;

    // Initialize game status
    mChaserHit = 0;
    mChaserMissed = 0;
}
```

```
    #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
}
```

André Rodrigues Nº13231
Fernando Camilo Nº13233

```
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);
}
```

Conclusão

Capitulo 4

Neste capitulo foram revistos conceitos matemáticos relativos a vetores e operações com esses mesmos vetores de forma a ser possível criar interações entre objetos, como a perseguição e colisão dos objetos de jogo, sendo que estas propriedades que foram adquiridas pelos objetos de jogo deve-se a calculo do produto dos pontos dos vetores, que definem os objetos de jogo, para definir o angulo de diferença entre esses dois objetos e assim usar esse mesmo angulo para modificar a direção em que um objeto se encontra e dar ilusão de perseguição de um objeto pelo outro. E com o aumento da complexidade e volume das propriedades desses mesmo objetos, verifica-se a necessidade criar objetos que aglomerem todas estas características de forma a tornar o código mais organizado, percetível e acima de tudo mais acessível para o acréscimo de novas “features” ao jogo. As dificuldades sentidas foram idênticas ao do capitulo anterior, maioritariamente devidas à incompatibilidade de novas versões da “framework” com as mais recentes versões do “IDE” que foi usada (“Visual Studio 2015”) com o acréscimo da compreensão matemática que é pouco intuitiva no que a compreensão e abstração deste exemplo, diz respeito mas apesar de tudo todo este processo e execução deste relatório serviu em grande parte para ligar conhecimentos de programação em linguagem C# com as técnicas mais usuais para fazer um jogo em MonoGame e assim dar bases fortes e um rumo mais clarividente para o que será importante fazer no segundo projeto da cadeira e qualquer futuro videojogo que façamos.