Engenharia em Desenvolvimento de Jogos Digitais

Técnicas de Desenvolvimento de Jogos

NINJOUT

Equipa:

Daniel Salgado – 10661

Tiago Pinheiro - 10929

Índice

Introdução

No âmbito da Unidade Curricular de Técnicas em Desenvolvimento de Jogos foi solicitado pelo docente aos alunos que para o trabalho prático realizassem um jogo 2D feito em linguagem C# na plataforma *Monogame*, com a temática a sua escolha, em que implementassem a matéria abordada ao longo do semestre.

O grupo, após discutir entre vários temas, acabou por decidir que o seu jogo seria um *platformer* híbrido de árcade e simulação para apenas um jogador, que teria como personagem um ninja, cujo objetivo seria percorrer os mapas de jogo derrotando todos os inimigos de modo a ultrapassar todos os níveis.

Descrição do Jogo

- História:

Estamos no 2027, os E.U.A. continuam a ser chamados para resolver os conflitos e evitar confrontos sangrentos, embora as vezes só os piorem. Com o evoluir dos anos, formar soldados tem sido cada vez mais dispendioso e para um país que tem cerca de 540 000 soldados em zonas de conflito ou em forças de pacificação, o custo é exageradamente alto.

Numa tentativa de usar soldados que tenham sido mutilados, mas com grande experiência, os E.U.A. tentaram criar soldados híbridos usando partes robóticas no lugar dos membros perdidos.

Ao fim de 2 anos terão conseguido criar com sucesso um hibrido eficiente e com um custo bastante menor, sendo que um hibrido equivalia a 30 soldados normais, tendo um bónus de ter experiência em zonas de conflito.

Pouco tempo depois, os cientistas terão criado um robô com um cérebro humano de um antigo praticador de *kung fu, e* para o testar terão criado 2 ambientes diferentes em que este teria de derrotar soldados híbridos e outros robôs de transporte de tropas. O seu objetivo seria derrota-los nos 2 níveis e regressar a zona dos testes, mas este secretamente planeava a sua fuga.

Sem saber, pouco depois de ter passado nos testes, os cientistas souberam que este queria viver em liberdade e terão tentado desmontá-lo, mas antes que o conseguissem este terá conseguido escapar do laboratório sendo que a única saída era através da zona de testes. O robô terá de enfrentar os seus oponentes sendo que estes têm ordens de usar força letal e pará-lo a todo o custo.

Tipo de Jogo

Este jogo foi feito em C# no *Visual Studio* através da plataforma *Monogame*, e baseia-se num jogo *platformer* 2D. O público alvo que procuramos com este jogo centra-se em jogadores casuais e hardcore com mais de 12 anos.

Objetivo

Tal como já referido em cima, neste jogo o personagem que é o ninja terá de usar as suas habilidades de modo a sobreviver aos seus oponentes e assim conseguir completar os níveis e escapar, de modo a conseguir alcançar a liberdade bastante desejada

Visualização

Ao longo do *NinjOUT* vai ser possível visualizar que este possui ambientes que aparentam ter vida, inimigos com uma *A.I.* avançada que não desistem de procurar destruir o personagem, perseguindo-o e ultrapassando também todos os obstáculos que apareçam.

Também o *spawn* dos inimigos e o seu comportamento é sempre aleatório, o que acaba por tornar o jogo sempre diferente cada vez que é jogado, o que evita assim a monotonia deste.

Mecânica de Jogo

Mecânicas *Player*

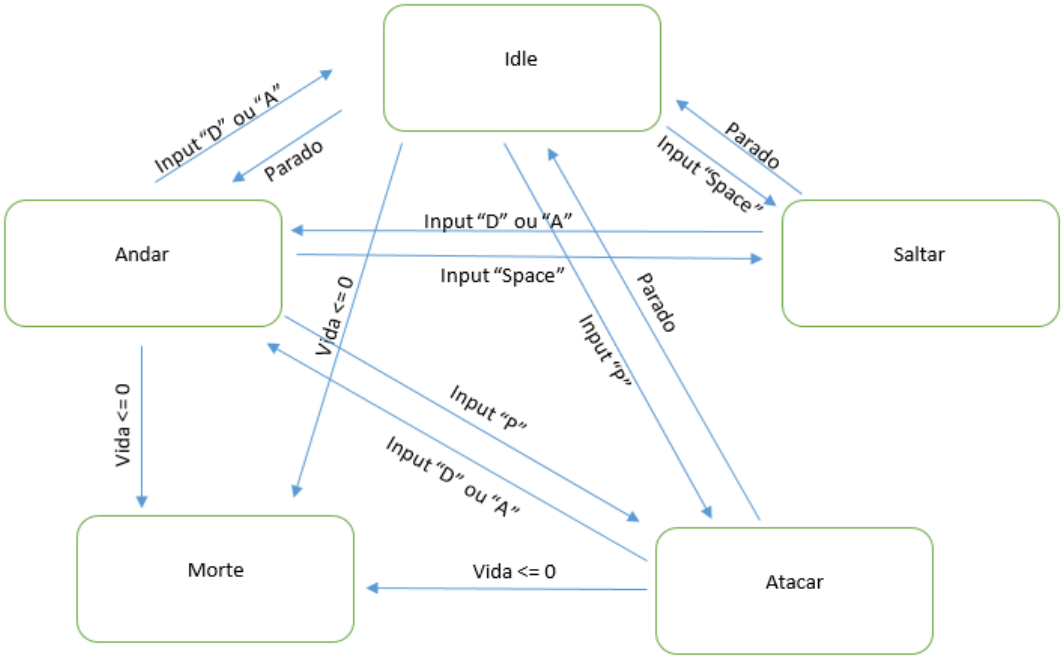

Figura 1 - Máquina de Estados do Player

O *player* possui vários estados (tal como pode ser visto na imagem acima), pelo que para cada uma delas possui uma animação diferente. Este inicialmente está em Idle, pelo que pode sucessivamente se deslocar, saltar ou atacar após ser acionado o input referente a cada um destes estados. Quando a sua vida fica igual ou menor a 0, este transita para o estado de "Morte" o que irá levar o jogador a ser reencaminhado de seguida novamente para o menu.

Mecânicas *Enemy*

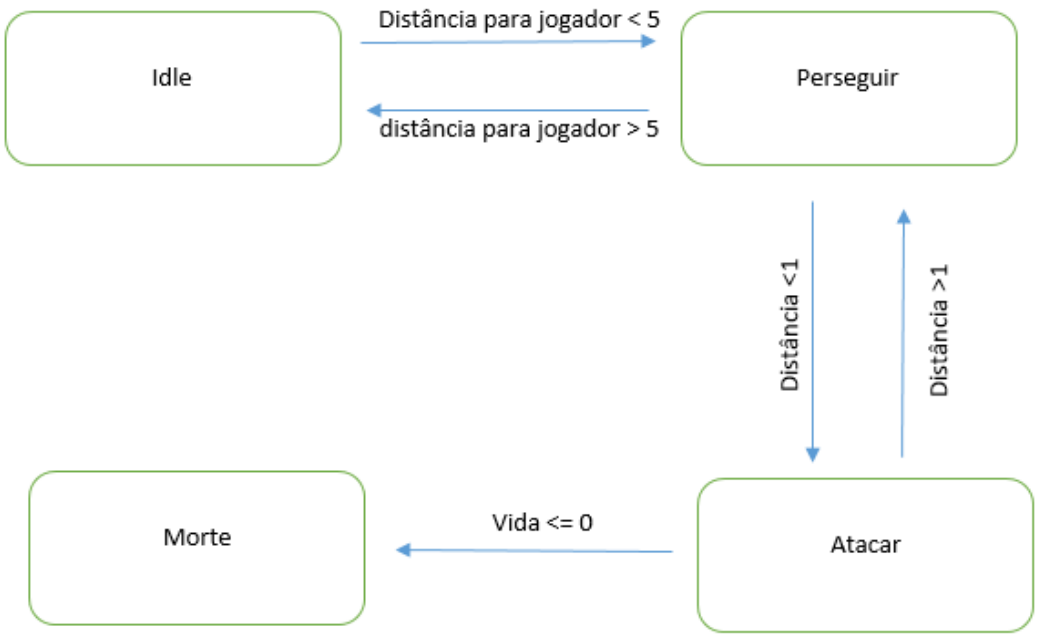

Figura 2 - Máquina de Estados do Enemy

Para o inimigo foram implementados 4 estados, sendo que o primeiro é o Idle. Assim que este verifica que a sua distância para com o player é menor que 5 inicia então a sua perseguição, e quando se encontra praticamente em contacto com este passa para o estado de ataque. Sendo a sua vida igual ou menor a 0 transita para o estado de "Morte" sendo consequentemente destruído.

Mecânica Base do Jogo

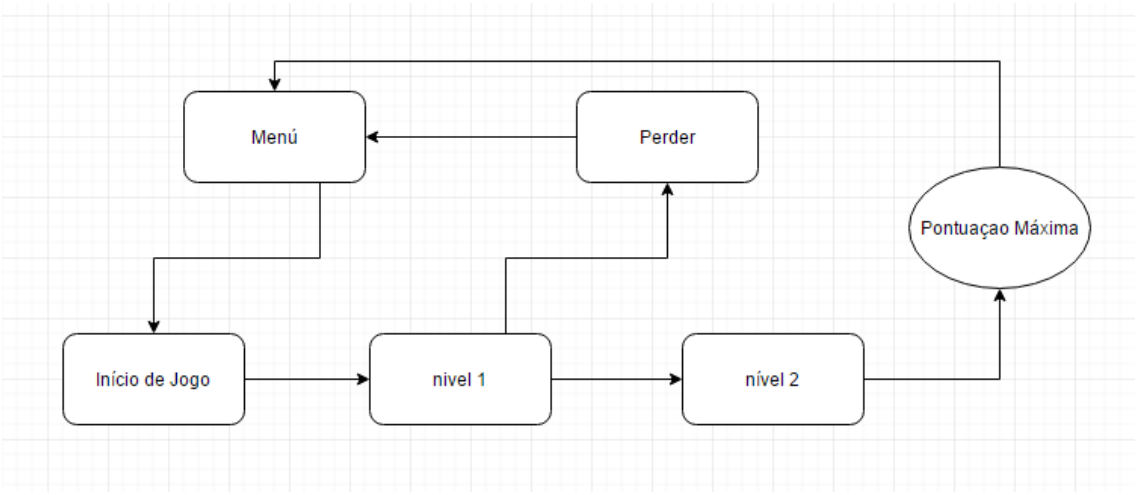

Figura 3 - Diagrama de Blocos sobre mecânica de Jogo

O jogador é inicialmente deparado com o menu, sendo depois encaminhado para o nível 1 assim que clicar no *Input* referido para este feito. Já no primeiro nível ele tem como objetivo sobreviver e ultrapassar todo o nível sem nunca ser derrotado por algum dos inimigos nem cair em nenhuma das armadilhas, e caso este ultima seja verificada é reencaminhado para o menu, e caso ultrapasse com sucesso este nível passa para o próximo nível.

No *level* 2 o funcionamento vai ser um pouco diferente pois aqui o objetivo do jogador será exclusivamente derrotar os inimigos, sendo recompensado com uma pontuação pela destruição de cada um destes. Quando morrer vai ser anunciada a sua pontuação final e será consequentemente reencaminhado para o menu.

Controles

Ao longo do *gameplay* o jogador poderá usar vários ataques utilizando o teclado.

Para movimentar-se ao longo do mapa, o jogador terá de utilizar as setas para poder mover-se, podendo também usar a tecla *Space* para saltar e assim desviar-se dos inimigos e ultrapassar obstáculos estáticos.

Implementação

Como foi notório ao longo dos pontos abordados anteriormente, são várias as implementações que o nosso jogo possui, pelo que as vamos explorar ponto a ponto de modo a ser mais compreensível o seu funcionamento.

Colisões

Para as colisões foi seguido um tutorial online, pois como esta era uma matéria com a qual não tínhamos contacto há já algum tempo, tivemos alguma dificuldade ao procurar soluções para como aplicar as colisões em cada um dos *rectangles*.

```
// r1 rectangle player, r2 rectangle "chão"
 public static bool TouchTopOf(this Rectangle r1, Rectangle r2)
 {
     return (r1.Bottom >= (r2.Top - 1) && r1.Bottom <= r2.Top + (r2.Height / 2) &&
         r1.Right >= r2.Left + (r2.Width / 5) && r1.Left <= r2.Right - (r2.Width / 5));
 }
 public static bool TouchBottomOf(this Rectangle r1, Rectangle r2)
 {
     return (r1.Top <= r2.Bottom + (r2.Height / 5) &&
         r1.Top >= r2.Bottom - 1 &&
         r1.Right >= r2.Left + (r2.Width / 5) &&
         r1.Left <= r2.Right - (r2.Width / 2));
 }
 public static bool TouchLeftOf(this Rectangle r1, Rectangle r2)
 {
     return (r1.Right <= r2.Right &&
         r1.Right >= r2.Left - 5 &&
         r1.Top <= r2.Bottom - (r2.Width / 4) &&
         r1.Bottom >= r2.Top + (r2.Width / 4));
 }
 public static bool TouchRightOf(this Rectangle r1, Rectangle r2)
 {
     return (r1.Left >= r2.Left &&
         r1.Left <= r2.Right + 5 &&
         r1.Top <= r2.Bottom - (r2.Width / 4) &&
         r1.Bottom >= r2.Top + (r2.Width / 4));
 }
```

Figura 4 - Código Relativo às colisões

Foi então adicionada uma classe denominada "*RectangleHelper*", onde são aplicados vários booleanos com os quais se verificam as posições dos dois retângulos e se verifica se estes estão a colidir em algum dos lados. Estes são então chamados nas classes dos "objetos" cujos queremos verificar se há colisão com algum outro componente e, sendo isto verificado, é então na própria classe onde esta função foi chamada implementada o comportamento que este terá de ter.

```
public void Collision(Rectangle newRectangle, int xOffset, int yOffset)
{
    if (rectangle.TouchTopOf(newRectangle))
    {
        // rectangle.Y = newRectangle.Y - rectangle.Height;
        rectangle.Y = newRectangle.Y - rectangle.Height;
        velocity.Y = 0f;
        hasJumped = false;
    }
```

Figura 5 - Exemplo de Colisão do Player

Quanto a esta ultima parte em que são chamadas as funções inicializadas no *"RectangleHelper",* aquilo que foi referido no tutorial seguido não estavam a funcionar corretamente, pelo que aí tivemos de ser nós a implementar corretamente.

Personagem

Para a implementação do personagem começou-se por inicializar o seu *rectangle* e os vetores velocidade e posição. Seguiu-se a adição dos inputs do teclado onde se fizerem as devidas alterações de velocidade e posição deste para que se efetuasse o respetivo movimento.

```
private void Input(GameTime gameTime)
{
    if (Keyboard.GetState().IsKeyDown(Keys.D))
    {
        velocity.X = (float)gameTime.ElapsedGameTime.TotalMilliseconds / 3;
        flip = false;
    }
    else if (Keyboard.GetState().IsKeyDown(Keys.A))
    {

        velocity.X = -(float)gameTime.ElapsedGameTime.TotalMilliseconds / 3;
        flip = true;
    }
    else
    {
        velocity.X = 0f;
    }
```

Figura 6 - Inputs de movimento do Player

Enquanto às animações deste, foram feitas através do uso de *spritesheets* sendo adicionada uma *Texture2D* independente para cada *spritesheet*. Estas são chamadas quando a ação referente é despoletada ou está em execução.

```
if (Keyboard.GetState().IsKeyDown(Keys.D) || (Keyboard.GetState().IsKeyDown(Keys.A)))
    currentTexture = WalkTexture;

else if (Keyboard.GetState().IsKeyDown(Keys.Space) /*&& hasJumped == false*/)
{
    currentTexture = JumpTexture;
}

else
{
    currentTexture = IdleTexture;
}
```

Figura 7 - Alguns dos exemplos de texturas a serem chamadas

Para que estas deem uma noção de animação dividiu-se a imagem por linhas e colunas e seguidamente foi-se alterando as *frames* conforme estas últimas, como vai ser possível analisar na imagem seguinte:

```
Input(gameTime);

frames++;

if (frames > 5)
{
    currentFrame++;
    if (currentFrame > 9)
        currentFrame = 0;

    frames = 0;
    colum = (int)((columnsWidth / 10) * currentFrame);

}

if (currentFrame == totalFrames)
{

    currentFrame = 0;
}

if (currentTexture == WalkTexture)
{
    sourceRectangle = new Rectangle(colum, row, currentTexture.Width / 10, currentTexture.Height);
}
```

Figura 8 - Animação da spriteshet

Para retirar *damage* aos inimigos é feita uma implementação na própria classe do *enemy*, onde se é verificada se a animação de ataque do jogador é despoletada e se a distância entre eles é menor que um determinado número. Se isto acontecer, é então retirada vida ao inimigo.

Camera

Para a *camera* foi implementado um procedimento relativamente simples, em que se recebe a posição atual do jogador e através desta se faz um cálculo do centro do ecrã através da largura e altura do *viewport*. Tendo o centro do ecrã de jogo calculado atribuiu-se uma translação constantemente à matriz *transform* da *camera* de modo a esta estar constantemente centrada com o jogador.

```
public void Update(Vector2 position, int xOffset, int yOffset)
{
    if (position.X < viewport.Width / 2)
        center.X = viewport.Width / 2;
    else if (position.X > xOffset - (viewport.Width / 2))
        center.X = xOffset - (viewport.Width / 2);
    else center.X = position.X;

    if (position.Y < viewport.Height / 2)
        center.Y = viewport.Height / 2;
    else if (position.Y > yOffset - (viewport.Height / 2))
        center.Y = yOffset - (viewport.Height / 2);
    else center.Y = position.Y;

    transform = Matrix.CreateTranslation(new Vector3(-center.X + (viewport.Width / 2), -center.Y + (viewport.Height / 2), 0));
}
```

Figura 9 - Calculo do centro do ecrã e translação da matriz transformação

Inimigos

O código dos inimigos seguiu basicamente a mesma lógica do que o do jogador, sendo também as animações feitas através de várias *spritesheets* associadas a uma *texture2D* independente e os danos efetuados também através de uma verificação de distância e da animação despoletada.

Posto isto o que foi realmente alterado em relação ao player foi o movimento e o *spawn* destes.

Para a translação destes foi calculada a distância relativamente ao jogador, sendo que só se moveriam para trás ou para a frente no Vetor X consoante este fator, tal como vai ser possível analisar na imagem seguinte:

```
if (player.Position.X>5900)
{
    velocity.X = 0;

    for (int i = 0; i < 4; i++)
    {
        if (player.Position.X - position.X > 170)
        {
            currentTexture = WalkTexture;
            velocity.X += 1;
            flip = false;
        }
        else if (player.Position.X - position.X+1 < -170)
        {
            currentTexture = WalkTexture;
            velocity.X -= 1;
            flip = true;
        }
        else break;


    }
}
```

Figura 10 - Código referente às deslocações dos inimigos

Relativamente aos ataques, são efetuados quando estes se encontram a uma determinada distância do jogador, sendo decrementado sucessivamente um determinado valor à variável "*health*" do *player*.

Para a realização do *spawn* destes foi criado um *Array* de inimigos no game1, onde para cada nível é definido o número de elementos a instanciar e se define a posição de cada um deles individualmente.

```
if (levelToLoad == 1)
{
    for (int i = 0; i < 5; i++)
    {


        enemyArray[i] = new Enemy(player);

        enemyArray[i].xPosEnemy = enemyArray[i].xPosEnemy + i * 200;

        enemyArray[i].Load(Content);
    }
}
if (levelToLoad == 2)
{
    for (int i = 0; i < 40; i++)
    {


        enemyArray[i] = new Enemy(player);

        enemyArray[i].xPosEnemy = enemyArray[i].xPosEnemy2 + i * 200;

        enemyArray[i].Load(Content);
    }

}
```

Figura 11 - Instanciação dos enemy's no LoadContent do Game1

Menu

A implementação do menu acabou por ser um processo relativamente simples, pelo que foi feita uma verificação através de um booleano, em que quando se está no menu é apresentada uma imagem (criada por nós) alusiva ao tema e que apresenta 2 espécies de botões (não didáticos) onde se pode ler: "*Press 'Enter' to Play*" e "*Press 'Esc' to Exit*". Com isto foi feita a adição de um "input" em que quando se clica no botão '*Enter*' é inicializado o jogo.

Game Over

Para ser acionado o *Game Over* têm de ser verificadas uma de duas condições, sendo uma delas a vida do jogador ser igual ou menor que zero, e a outra é que o jogador caia num dos vários buracos presentes em ambos os níveis, sendo que para verificar se este caiu nesta armadilha analisa-se constantemente se o Y da posição do jogador ultrapassa um determinado valor e se esta condição for verdadeira é alterado o booleano "*isGameOver*" para *true*.

Posto isto vai ser então apresentado no ecrã uma imagem relativa ao Game Over, que funciona muito semelhantemente ao Menu, apresentando os mesmos botões e o mesmo input, que quando acionado faz com que o jogador reinicie o jogo no nível1.

TileMap

Relativamente ao mapa de jogo, foi criado através de um *Tile Map* e tivemos também de seguir um tutorial para proceder ao desenvolvimento deste pois era um dos pontos com os quais já não tínhamos contacto há já algum tempo, o que nos debilitou quanto às noções iniciais aquando do seu desenvolvimento.

Foi então inicialmente criada uma classe *Tile* onde foram adicionadas as propriedades de cada *Tile.*

```
    private Rectangle rectangle;
    public Rectangle Rectangle
    {
        get { return rectangle; }
        protected set { rectangle = value; }
    }

    private static ContentManager content;
    public static ContentManager Content
    {
        protected get { return content; }
        set { content = value; }
    }

    public void Draw (SpriteBatch spriteBatch)
    {
        spriteBatch.Draw(texture, rectangle, Color.White);
    }
}

class CollisionTiles : Tile
{
    public CollisionTiles(int i, Rectangle newRectangle)
    {
        texture = Content.Load<Texture2D>("Tile" + i);
        this.Rectangle = newRectangle;
    }
```

Figura 12 - Classe Tile

Seguidamente criou-se a classe “*Map*”, onde se inicializou uma lista de Tiles e onde se implementou o “*Generate*” destes.

```
public void Generate(int[,]map, int size)
{
    for (int x = 0; x < map.GetLength(1); x++)
    {
        for (int y = 0; y < map.GetLength(0); y++)
        {
            int number = map[y, x];

            if (number > 0)
            {
                collisionTiles.Add(new CollisionTiles(number, new Rectangle(x * size, y * size, size, size)));

                //x + 1, porque o x pode ser 0, e caso isso aconteça o reusltado da multiplicação seria 0
                width = (x + 1) * size;
                height = (y + 1) * size;
            }
        }
    }
}

public void Draw (SpriteBatch spritebacth)
{
    foreach(CollisionTiles tile in collisionTiles)
    {
        tile.Draw(spritebacth);
    }
}
```

Figura 13 - Generate dos Tiles na Classe "Map"

Por fim no Game1 é então realizado o desenho do *tileMap*, sendo chamado no *Load Content* a função Generate e realizado o respetivo Mapa de jogo.

```
map = new Map();
if (levelToLoad == 1)
{
    map.Generate(new int[,]
{


{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
{ 0, 0, 0, 0, 0, 0, 0, 1, 5, 5, 5, 7, 0, 0, 1, 5, 4, 6, 5, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0,
{ 0, 0, 0, 0, 0, 0, 1, 4, 3, 3, 3, 8, 0, 0, 2, 3, 3, 3, 3, 6, 7, 0, 0, 0, 0, 0, 1, 5, 7, 0, 0, 2, 8, 0, 0, 1, 5, 7, 0, 0, 1, 5,
{ 0, 0, 0, 0, 0, 1, 4, 3, 3, 3, 3, 8, 0, 0, 2, 3, 3, 3, 3, 3, 6, 7, 0, 0, 0, 1, 4, 3, 8, 0, 0, 2, 8, 0, 0, 2, 3, 8, 0, 0, 2, 3,
{ 5, 5, 5, 5, 5, 4, 3, 3, 3, 3, 3, 8, 0, 0, 2, 3, 3, 3, 3, 3, 3, 8, 0, 0, 1, 4, 3, 3, 8, 0, 0, 2, 8, 0, 0, 2, 3, 8, 0, 0, 2, 3,
}, 128);
}
```

Figura 14 - Pequeno exemplo de Desenho de TileMap no Game1

Vitória

O jogador é vencedor quando completa os dois níveis sem morrer, tendo também como objetivo procurar matar todos os inimigos que conseguir, perder o mínimo de vida possível e chegar ao fim dos dois níveis no mínimo de tempo possível, pois quando é apresentado a imagem de "*You Win*" é também apresentado a pontuação combinada, que é calculada através da multiplicação dos pontos adquiridos ao derrotar os inimigos com a vida restante do jogador, a dividir pelo tempo que demorou a completar os dois níveis, acabando assim o jogo por apresentar alguma dinâmica e competitividade diferente.

Dificuldades Encontradas

Ao longo da implementação dos tópicos referidos anteriormente as dificuldades que mais tivemos prenderam-se muito com os factos de devido a estar-mos ambos os elementos do grupo a terminar o terceiro ano da licenciatura, ter-mos tido bastante trabalho com o nosso projeto e, devido ao facto desta ser uma unidade curricular do primeiro ano, já não termos contacto com esta matéria à já algum tempo, o que por si fez logo desde inicio que as dificuldades fossem aparecendo.

No geral as dificuldades prendem-se com as colisões e o *TileMap*, pelo que tivemos de recorrer a tutoriais para suceder à sua implementação. Seguidamente tivemos também dificuldade com as animações das *Spritesheets*, pois para cada uma delas teve de se alterar as *columns* e as *rows* e foi bastante complicado encontrar valores com as quais a animação fica-se fluida e agradável.

Outra dificuldade foi a reprodução de dois sons em simultâneo, pois o *XNA*/ *Monogame* não permite este feito e a única forma era a instalação de outros programas e a procura de outras soluções, as quais não pudemos procurar devido ao compromisso que tivemos em outras unidades curriculares.

Arte

Ambiente de Jogo

O ambiente de jogo é maioritariamente constituído por um Terreno de jogo e um background alusivo ao tema para cada um dos níveis.

O terreno de jogo foi feito através de um *tilemap*, em que cada tile representa um retângulo texturizado e todos juntos foram uma espécie de um puzzle dando o efeito de um terreno de jogo tal como pretendido.

O jogador é também deparado com partes do jogo como o menu, o "*Game Over*" e o "*You Win*", onde em todos são apresentadas imagens e tipos de letra alusivos ao tema.

Para criar um ambiente mais imersivo no que toca à temática ninja, foram adicionadas 3 músicas ao jogo, uma para o menu, outra para o nível 1 e uma última para o nível 2, sendo todas elas de género asiático.

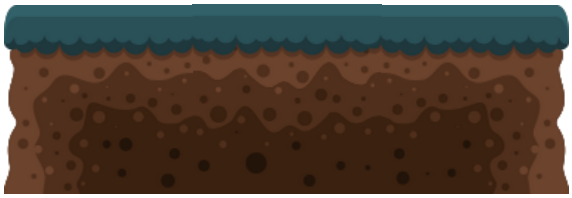

Figura 15 - Exemplos de Tiles do Tile Map

Figura 16 – Imagem Menu

Figura 17 - background level1

Player e Enemy

As *sprites* do personagem e o inimigo foram retiradas de um site que as disponibiliza gratuitamente, o que para nós foi bastante bom não só por terem sido fáceis de obter, mas também porque eram mesmo aquilo que tínhamos em mente desde o momento que idealizamos este jogo, este que acabou por ser um ponto em que o nosso trabalho ficou bem mais facilitado.

Quanto à spriteSheet de cada ataque já foram construídas por nós.

Figura 18 - SpriteSheet do Ataque do Personagem

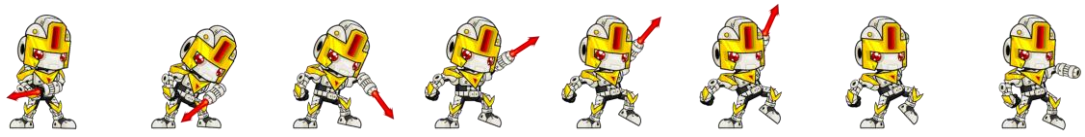

Figura 19 - SpriteSheet do Inimigo

Desenvolvimento

A nossa ideia inicial passou por criar um *platformer* em que tivéssemos como personagem principal um ninja e onde conseguíssemos adicionar um ambiente bastante imersivo em relação à temática Ninja. Esta ideia acabou por ser seguida, ficando apenas por resolver o facto de não termos conseguido adicionar toda aquela imersividade que procurávamos inicialmente.

Em termos de dificuldades foram basicamente aquelas que foram já referidas em cima, juntando a falta de tempo que tivemos devido à grande quantidade de trabalho que nos foi entregue em outras unidades curriculares.

Quanto ao tempo de desenvolvimento e a forma como este foi seguido, fornecemos de seguida o link para o repositório do nosso projeto no Github, onde temos toda esta informação disponível:

Link Repositório Github:

https://github.com/TiagoPinheiro2580/NinjOut

Desenvolvimento Futuro

Futuramente no nosso jogo podem vir a ser implementadas novas *features*, tais como um menu com deteção de botões dinâmicos e uma aba em que fosse permitido ao jogador visualizar as instruções e história do jogo, efeitos secundários de jogo (por exemplo sangue a sair do jogador quando atacado), sons para todos os ataques e elementos do jogo, mais um nível com outro tipo de dinamismo, mais inimigos e também uma espécie de "*pick Up's*" para premiar o jogador.

Conclusão

Após a realização deste trabalho são várias as valências que readquirimos, pois esta era uma matéria com a qual já não tínhamos contacto há algum tempo devido a já estarmos no terceiro ano da licenciatura. Para tal tivemos de procurar na internet por tutoriais e documentos que nos ajudassem a relembrar algumas temáticas já esquecidas e das quais necessitávamos para a realizar o proposto.

Tal como referimos em cima, por estarmos no fim desta nossa caminhada na licenciatura em Engenharia em Desenvolvimento de Jogos Digitais, tivemos imenso trabalho neste semestre com unidades curriculares como a de Projeto Aplicado II, o que acabou por nos deixar um pouco debilitados em termos de tempo para dedicar a este trabalho, o que consequentemente no prejudicou um pouco no alcance dos nossos objetivos delineados previamente para este desenvolvimento. Contudo não deixamos de dar o máximo possível para conseguir concluir este trabalho e assim conseguirmos dar também dar por concluída a Licenciatura em Desenvolvimento de Jogos Digitais no Instituto Politécnico do Cávado e do Ave.