FTT JOURNAL
of Engineering and Business

Desenvolvimento de jogo Platformer em Unity: Vlad, the Platformer

Development of a platformer game in Unity: Vlad, the Platformer

Fernando Sarturi Prass *(fprass@gmail.com)*

Mestre em Ciência da Computação pela Universidade Federal de Santa Catarina (UFSC). Professor do Centro Universitário Franciscano (Unifra) e diretor administrativo da FP2 Tecnologia.

Marcos Righi Brasil *(marcos_righi@yahoo.com.br)*

Bacharel em Ciência da Computação pelo Centro Universitário Universitário Franciscano (Unifra)

FTT Journal of Engineering and Business. • SÃO BERNARDO DO CAMPO, SP

SETEMBRO 2017 • ISSN 2525-8729

Submissão: 14 abr. 2017. **Aceitação:** 26 jun.. 2017

Sistema de avaliação: às cegas dupla (*double blind review*).

FACULDADE TECNOLOGIA TERMOMECANICA, p. 79 - 94

Engenharia de Computação

Resumo

Jogos são uma das formas de entretenimento modernas mais populares no mundo. A indústria de jogos exige o trabalho de profissionais de diversas áreas para que os jogos mais ambiciosos possam se tornar realidade, como computação gráfica, programação, som e design. O desenvolvimento de um jogo é um processo longo, complexo e difícil. Visando conhecer e aprender com o desenvolvimento de um jogo inspirado em Castlevania, do gênero platformer, foi criado um protótipo de um jogo utilizando o motor de jogos Unity. Este trabalho apresenta o processo de desenvolvimento do protótipo, que tem uma fase demonstrativa e as principais mecânicas de jogo implementadas.

Palavras-chave: Desenvolvimento. Jogos. Arcade. Unity.

Abstract

Games are one of the world's most popular modern forms of entertainment. The games industry requires the work of professionals from several areas so that the most ambitious games can come true, in areas like computer graphics, programming, sound and design. The development of a game is a long, complex and hard process. Aiming to know and learn with the development of a game inspired in Castlevania, from the genre platform, a prototype of a game was created utilizing the Unity game engine. This paper shows the development process of the prototype, who has a demonstration level and the main game mechanics implemented.

Keywords: Development. Games. Arcade. Unity.

Introdução

Jogos são uma das formas de entretenimento mais populares do mundo e uma das indústrias que mais movimentam dinheiro, ultrapassando até mesmo a indústria do cinema (CHATFIELD, 2009). Em 2014, consumidores gastaram US$ 22 bilhões em jogos, hardware e acessórios, incluindo compras de jogos para dispositivos móveis, assinaturas e jogos em redes sociais. No Brasil, considerando-se os moradores das principais regiões metropolitanas, 41% afirmaram possuir um console de *videogame* (IBOPE, 2012) e a previsão é que sejam gastos US$ 844 milhões nesse mercado (FLEURY; NAKANO; CORDEIRO, 2014, p. 33).

O desenvolvimento de jogos envolve processos criativos e tecnológicos, exigindo dos desenvolvedores não apenas a capacidade de idealizar e criar os elementos que definirão o jogo, mas também a capacidade de unir todos os elementos em um produto final. É um processo complexo, cujo trabalho pode ser dividido entre membros de uma equipe com diferentes especialidades. Apesar da dificuldade, equipes pequenas, e até mesmo desenvolvedores individuais, também conseguiram produzir jogos, denominados jogos *indies*, ou independentes. São jogos que, muitas vezes, obtêm destaque por possuírem características inovadoras (DUTTON, 2012).

Este trabalho apresenta o desenvolvimento de um protótipo de um jogo com mecânicas de *gameplay* (jogabilidade) de jogos do gênero plataforma inspirado em jogos da franquia Castlevania, chamado Vlad, the Platformer, cuja temática envolve vampiros, monstros e castelos góticos.

Para o desenvolvimento do protótipo foi utilizado o Unity, um *game engine* (motor de jogos) que fornece bibliotecas gráficas, físicas e um sistema de *scripts* (LEWIS E JACOBSON, 2002), necessários para o desenvolvimento do jogo.

Referencial teórico

Esta seção apresenta brevemente alguns gêneros e motores de jogos, com maior ênfase para o gênero plataforma e para o motor Unity.

Gêneros de jogos

Para facilitar a classificação de jogos, eles são divididos por gênero. Existem gêneros de natureza classificativa mais abrangente, como ação, aventura, estratégia; e também mais específicos, também chamados de subgêneros, como *survival* (sobrevivência) ou *tower defense* (defesa de torres). O gênero reflete as principais características do jogo (RABIN, 2012).

O jogo desenvolvido neste trabalho é do gênero plataforma, ou *platformer*, com perspectiva 2D e fases no qual o jogador utiliza movimentos e habilidades para ultrapassar obstáculos (RABIN, 2012). São exemplos de jogos do tipo *platformer* o Super Mario World (Nintendo) e o Castlevania: Symphony of the Night (Konami), sendo que este último é resultado da combinação de *platformer* com ação e aventura.

Game Engines

Um *game engine*, em português, motor de jogos, pode ser definido como uma "coleção de módulos de códigos de simulação que não especificam diretamente o comportamento de um jogo (lógica do jogo) ou o ambiente do jogo (dados de níveis)" (LEWIS; JACOBSON, 2002). Em outras palavras, *engines* auxiliam o desenvolvimento de jogos, proporcionando as ferramentas necessárias para trabalhar os vários aspectos de um jogo, como cenários, áudio, animações e simulação de física.

Para o desenvolvimento de jogos, existem as opções de construir o próprio motor de jogos ou utilizar um através da aquisição de uma licença. Um exemplo de *engine* é a Unreal Engine 3 (EPIC GAMES, 2016), que possibilita a criação de jogos de inúmeros gêneros e estilos diferentes, porém a complexidade da ferramenta a torna difícil de ser utilizada por desenvolvedores menos experientes. Outro exemplo é a Source que, apesar de completa, é mais acessível que a primeira. Ambas são focadas no desenvolvimento de jogos 3D. O Unity, *engine* utilizado neste trabalho, oferece suporte ao desenvolvimento tanto de jogos 3D como de 2D, com um módulo e ferramentas específicas para facilitar o trabalho com 2D.

Unity

Unity é uma *engine* desenvolvida pela Unity Technologies. Este projeto utilizou a versão 4.6.3, a mais atualizada disponível no início do desenvolvimento do protótipo. Não foram realizadas atualizações durante o desenvolvimento para se evitar problemas de incompatibilidade com o que já havia sido finalizado. Duas características foram consideradas para levar à escolha da Unity como *engine*: 1) o fato de ela possibilitar o desenvolvimento de jogos de diferentes gêneros e graus de complexidade; e 2) o módulo dedicado especificamente ao desenvolvimento de jogos 2D.

Entre as ferramentas e componentes disponíveis estão as que lidam com física, gráficos, *scripting*, áudio, animação e UI (*User Interface*). A parte gráfica é composta de vários recursos que são utilizados para compor o aspecto visual do jogo, como texturas, iluminação, a câmera do jogo e *shaders*. *Shaders* definem como a superfície de objetos reage aos elementos externos; através de *shaders* é possível gerar efeitos de reflexão em superfícies de vidros e de espelhos, ou efeitos dinâmicos onde a água de um cenário reflete a iluminação do mundo à sua volta através de suas ondas.

Scripting refere-se à programação dentro do Unity, que é realizada através de *scripts*. Os *scripts* podem ser escritos em C#, Javascript ou Boo (removida a partir da versão 5.0).

Entre as várias finalidades de *scripts*, uma das principais é a leitura de *input* (entrada, ou seja, comandos) do jogador e a organização de eventos que deverão ocorrer dentro do jogo. Outros usos de *scripts* envolvem a manipulação do comportamento físico de objetos, ou mesmo a implementação de um sistema de inteligência artificial para personagens de um jogo.

A seguir, estão descritos alguns componentes e elementos básicos do Unity, necessários para a compreensão do processo de desenvolvimento deste trabalho:

- *Scene*: contém todos os objetos presentes em uma cena de um jogo; exemplo: várias *scenes* podem ser criadas para conter diferentes fases de um mesmo jogo, cabendo ao desenvolvedor implementar a transição de uma cena para outra. Uma vez adicionados objetos na cena, organizados e definidos seus comportamentos (propriedades), componentes e *scripts*, o Unity se encarrega de garantir que todos esses elementos funcionem em sintonia.
- *GameObject*: todo objeto que existe dentro de uma cena de um jogo é um *GameObject* como, por exemplo, o personagem controlado pelo jogador e elementos do cenário. Componentes podem ser acoplados a um *GameObject*, por exemplo: se for adicionado um componente *RigidBody* a um objeto 3D (como um cubo ou uma esfera), é possível aplicar forças físicas, como a força gravitacional, nesse objeto.
- *Prefabs*: são *GameObjects* exportados para fora da hierarquia interna do projeto com a finalidade de facilitar a criação de objetos que serão utilizados múltiplas vezes. É um modelo de *GameObject,* no qual atributos e componentes já estão configurados e definidos, eliminando a necessidade de programar cada objeto a partir do zero.

Metodologia

O processo de planejamento de um jogo é conhecido por *game design* e tem como objetivo concretizar as principais ideias acerca do jogo proposto. Porém, *game design* engloba muitos elementos que são necessários somente em um jogo completo, tais como personagens, enredo e temática. Como a proposta aqui era o desenvolvimento de um protótipo, muitos desses elementos seriam desperdiçados. Portanto, foram utilizados somente alguns dos princípios de *game design* para um jogo de escopo reduzido; são eles: definir as mecânicas de jogo; definir o personagem que será controlado pelo jogador e suas características e habilidades; elaborar uma fase para ser implementada; definir a interface do jogo e definir um objetivo simples para ser atingido. Esses e outros princípios para o desenvolvimento de jogos são apresentados por Guimarães Neto e Ernane (2014), Tonéis e Frant (2015) e Rabin (2012).

Proposta geral do jogo

O jogo, nomeado Vlad, the Platformer, tem seu *gameplay* inspirado em jogos do gênero *platformer*. Esses gêneros remetem a jogos em 2D, cujo foco é a exploração de ambientes fazendo uso de habilidades específicas para vencer obstáculos. Vlad, the Platformer foca no elemento de exploração, dando ao jogador habilidades específicas para serem utilizadas em fases com obstáculos diferentes. O protótipo contém uma fase demonstrativa, na qual o jogador deve fazer uso das habilidades do personagem para chegar ao final da fase.

Personagem e cenário

O protagonista do jogo, controlado pelo jogador, é um vampiro chamado Vlad, que pode andar em sua forma humana ou se transformar em um morcego, sendo possível atravessar trechos do cenário estreitos demais para serem percorridos na forma normal.

O cenário do jogo é composto por campos e plataformas ao ar livre, em um ambiente noturno. No final do cenário há um castelo, e o objetivo do protótipo se resume a chegar à porta desse castelo.

Movimentação e habilidades

A movimentação básica do personagem é semelhante à dos jogos da série *Super Mario World*: andar para a esquerda (seta para esquerda), para a direita (seta para direita), e pular (barra de espaço). Quando o jogador utilizar a ação de pular, ele terá controle sobre a direção do salto no ar. O jogador tem liberdade para ir e voltar à vontade pelo cenário. A câmera do jogo acompanha o personagem centralizado na tela.

Além da capacidade de movimentação, o personagem dispõe de habilidades que devem ser específicas para ultrapassar os obstáculos. São elas:

Pulo duplo: ao apertar a barra de espaço durante um salto, é realizado um segundo salto. Utilizado para alcançar lugares mais altos.

Salto em paredes: se, durante um salto, o personagem encostar em alguma parede, ao se apertar a barra de espaço o personagem utiliza a parede para impulsionar e realizar um outro salto na direção oposta à parede.

Dash: é uma arrancada veloz que o personagem pode realizar para uma direção específica (esquerda ou direita) ao se apertar a tecla *shift*. Essa arrancada é feita na forma de morcego, ou seja, Vlad se transformará em morcego, e nessa forma ele arrancará velozmente para a direção escolhida e, ao final do voo, voltará à forma normal de vampiro (vide esquema na Figura 10). O voo tem uma trajetória limitada.

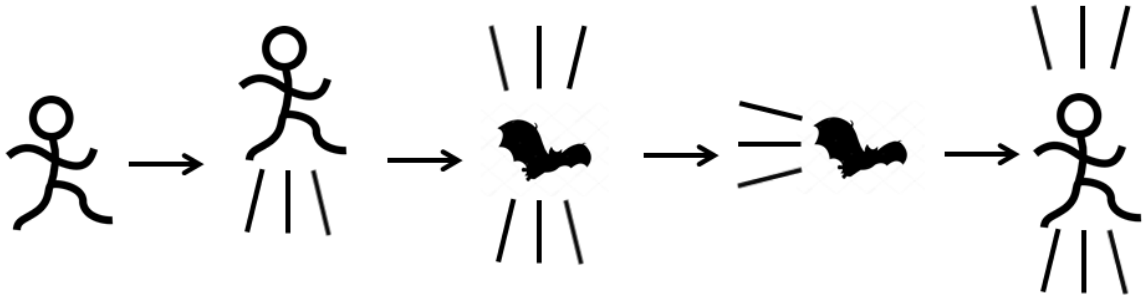

Figura 10. Desenho representando a ação da habilidade *Dash*.

Descrição da fase do jogo

Para demonstrar o uso das mecânicas do jogo em ação, foi desenhada a fase do protótipo do jogo. Pode-se observar na Figura 2 que o personagem do jogador inicia a ação onde está indicada a letra A. O objetivo da fase é chegar até a porta do castelo, localizada na parte superior da fase, próximo à letra F.

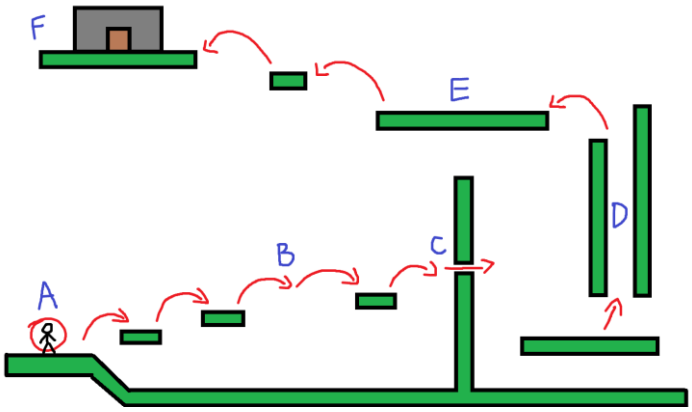

Figura 11. Rascunho da fase protótipo do jogo.

A fase pode ser finalizada da seguinte forma: o jogador (A) pula nas duas primeiras plataformas e utiliza a habilidade "pulo duplo" para alcançar a terceira (B). Para ultrapassar o obstáculo seguinte (C), é preciso usar a habilidade de transformação em morcego, pois o caminho é muito estreito. No próximo obstáculo (D), o jogador deve utilizar a habilidade de salto em paredes, pois existem duas colunas próximas entre si que permitem que o jogador as escale até a parte superior. Chegando à plataforma maior (E), o jogador deve seguir para a esquerda, pulando a última plataforma e, por fim, alcançando a porta do castelo (F), quando é exibida uma mensagem informando o fim da fase.

Desenvolvimento do jogo

O primeiro passo do desenvolvimento foi criar a *scene* principal, na qual posteriormente foi construída a fase jogável do protótipo. Com a cena criada, o objetivo seguinte foi obter texturas para poder representar graficamente os elementos principais do jogo: o personagem e o cenário. Todas as texturas utilizadas no protótipo (ver Figura 12) foram obtidas do site OpenGameArt.org, que disponibiliza recursos gráficos e sonoros para uso, alguns com licenças gratuitas. As texturas obtidas foram divididas e organizadas conforme a necessidade, utilizando o *Sprite Editor* da Unity. Para estabelecer um padrão nas dimensões dos *sprites*, foram utilizadas texturas divisíveis por 16 *pixels*.

Figura 12. Algumas das texturas obtidas para serem utilizadas no jogo.

Finalizados os ajustes das texturas do cenário e do personagem, iniciou-se o desenvolvimento dos primeiros objetos do jogo. Primeiramente, como já haviam sido adicionadas texturas dedicadas ao cenário, foi preciso trabalhar com as dimensões definidas previamente: 16x16 *pixels*. Na construção de um elemento do cenário, é possível que ele tenha dimensões maiores: uma plataforma pode ser composta de várias texturas unidas, por exemplo. Ao adicionar uma textura na cena, ela surge na forma de um objeto. Todos os objetos presentes em uma cena aparecem organizados em uma hierarquia. Para tornar a criação de cenários mais organizada, é possível tornar um objeto em um *child* (filho) de outro objeto.

Portanto, para criar uma plataforma, primeiro cria-se um objeto vazio na cena; todas as texturas que irão compor essa plataforma são adicionadas como filhos desse objeto. É importante salientar que, se alguma propriedade de um objeto *parent* (pai) for alterada, todos os objetos filhos serão influenciados por essa alteração. Se um objeto pai for transladado, todos os objetos filhos serão movidos junto com ele. Isso acaba facilitando a manipulação dos elementos do cenário dentro da cena.

Concluída a parte gráfica da plataforma, foram adicionados dois componentes para que ela possa ter presença física dentro do jogo, ou seja, para que o personagem possa caminhar e permanecer sobre ela. O primeiro componente foi um *Rigidbody 2D* (corpo rígido 2D), que faz o objeto ser influenciado pela física da *engine*. Como o objeto em si é uma plataforma que faz parte do cenário e é fixa, ela não deve ser movida pela gravidade nem pelo contato com outros objetos; então alguns parâmetros foram

alterados para que a plataforma ignore essas forças e permaneça fixa em sua posição. O segundo componente foi um *Box Collider 2D* (colisor de caixa em 2D), que define as dimensões do *Rigidbody 2D* do objeto. Um *rigidbody* precisa de um *collider* para especificar quais as dimensões exatas do corpo do objeto; no caso de uma plataforma, um *collider* no formato de um retângulo foi a escolha ideal. Com esses componentes adicionados e ajustados, a plataforma estava finalizada.

O próximo elemento a ser criado foi o personagem, cujas texturas obtidas continham vários quadros de animação, entre eles: parado, caminhando, pulando e sendo atingido. Foi criado o objeto do jogador, adicionadas as texturas do personagem parado e adicionados os componentes *Rigidbody 2D* e um *Box Collider 2D*. Um ajuste feito no *Rigidbody* fixou o ângulo do personagem, para ele não cair quando estiver na beira de uma plataforma. Depois iniciou-se a programação do personagem, com a implementação dos movimentos mais básicos: caminhar e pular. A programação dessas funcionalidades foi realizada juntamente com o desenvolvimento das suas animações.

Para lidar com animações, a Unity possui dois módulos: *Animator* (animador) e *Animation* (animação); com o *Animator* é possível criar estados de animação de um objeto e estabelecer relações entre esses estados, e com o *Animation* é possível editar a ordem e o fluxo dos quadros de cada animação específica. Para o personagem, foram criados quatro estados de animação: *Idle*, quando o personagem está parado; *Moving*, quando o personagem está caminhando; *Rising*, quando o personagem está subindo (início de um pulo) e *Falling*, quando o personagem está caindo (depois de um pulo ou após cair de uma plataforma). O relacionamento entre os estados de animação pode ser observado na Figura 13.

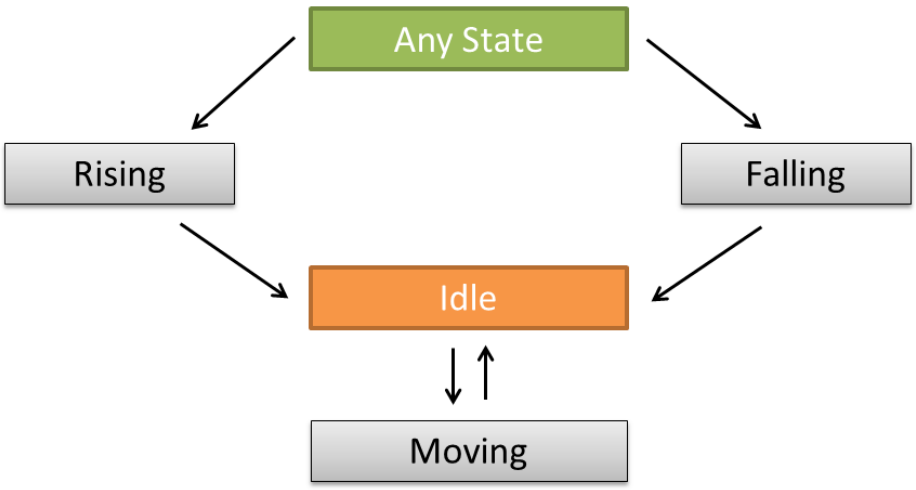

Figura 13. Fluxograma de estados de animação e seus movimentos relacionados no módulo Animator do Unity.

A programação de objetos em uma cena é feita através de *scripts*, que suportam múltiplas linguagens de programação. Por questão de familiaridade do autor, a linguagem escolhida para este projeto foi C#. Todo *script* novo criado inclui algumas importações básicas, uma classe com o mesmo nome do *script* herdado da classe

MonoBehaviour, e dois métodos, *Start* e *Update*. A classe *MonoBehaviour* é a base para todos os scripts da Unity e contém diversas funções e variáveis necessárias para o funcionamento de objetos no jogo. Os métodos *Start* e *Update* são derivados da classe *MonoBehaviour*; elas funcionam da seguinte forma: o *Start* é chamado na primeira vez que um objeto é ativado dentro do jogo, e somente essa única vez, funcionando de modo semelhante a um construtor; já o método *Update* é chamado a cada *frame* de execução do jogo, então é um método muito útil para, por exemplo, capturar o *input* do jogador.

Para fazer o personagem se movimentar, no entanto, não pode ser utilizado o método *Update*, pois, dependendo da taxa de quadros da execução do jogo em um determinado computador ou dispositivo, o resultado será diferente. Como é necessário fazer com que o jogo funcione da mesma forma, independentemente do dispositivo, foi utilizada a função similar *FixedUpdate*, a qual é utilizada em uma taxa de quadros fixa pela própria Unity. Em vista disso, a melhor prática é sempre utilizar esta função quando for trabalhar com a física do objeto, e a função *Update* para outros propósitos.

É preciso obter o *input* do jogador para que o personagem reproduza algum movimento dentro do jogo. Para o personagem caminhar, como a perspectiva do jogo é 2D, o tipo de *input* que deve ser obtido é aquele que indica movimento horizontal dentro do eixo do jogo. Para isso, foi utilizada a função *Input.GetAxisRaw ("Horizontal")*, que retorna a 0 (zero) caso nenhum *input* tenha sido recebido, a -1 (um negativo) quando indicar movimento para a esquerda e a 1 (um) quando indicar movimento para a direita. Com esses valores, é possível utilizá-los na função de translação da Unity para mover o *Transform* do personagem dentro da cena. O *Transform* é um componente que existe em todos os objetos contidos em uma cena, e armazena todos os valores de posicionamento de um objeto, sendo eles: posição, rotação e escala. A função *Translate* serve para mover a posição de um *Transform*. Entretanto, cabe frisar que utilizar somente o resultado do *input* para mover o objeto não é recomendado, pois o intervalo de valores é muito baixo (entre um negativo e um). Para se ter mais controle sobre a velocidade de movimento do personagem, cria-se uma variável *float* (que pode ser chamada de *speed*, velocidade) para multiplicar o valor de *input* e definir o quão rápido o personagem caminha dentro do jogo, conforme mostra a Figura 14.

```
85      void Move (float horizontal) {
86          movement.Set (horizontal, 0f);
87
88          playerTransform.Translate (movement * speed * Time.deltaTime);
89
90          moving = horizontal != 0;
91      }
```

Figura 14. Função para mover o personagem no eixo horizontal.

A função *Move* é chamada uma vez a cada quadro de execução. Como a taxa de quadros pode variar de uma execução para outra, a quantidade de chamadas da função *Move* é inconsistente. Multiplicar os valores de translação pela variável *Time.deltaTime*, utilizada na linha 88 da Figura 5, possibilita que o resultado de múltiplas chamadas a essa função seja constante e independente da taxa de quadros de cada execução.

Para o personagem poder pular, a programação foi feita da seguinte forma: caso o jogador tenha apertado a barra de espaço, o botão definido para pular, é aplicada uma força direcionada para cima ao *Rigidbody* do personagem (ver Figura 9). Uma das habilidades do personagem é realizar um pulo duplo; portanto, foi usado um contador para verificar quantos saltos o personagem já fez até tocar novamente o chão. A função *AddForce* soma uma força sobre todas as forças físicas que já estão em ação sobre o personagem. Essas forças que agem sobre um objeto podem ser obtidas e alteradas no atributo *velocity* (velocidade), presente no *Rigidbody* do objeto. Sabendo disso, para que o segundo pulo não fique mais forte ou mais fraco que o primeiro e tenha um resultado padronizado, é necessário zerar a *velocity* do personagem, como pode ser visto na linha 95 da Figura 15.

```
93      void Jump () {
94          if (jumpCommand) {
95              playerRigidBody.velocity = Vector2.zero;
96              playerRigidBody.AddForce (new Vector2 (0f, 10f) * jumpForce);
97              jumpCommand = false;
98          }
99      }
```

Figura 15. Função para o personagem pular.

Com as funcionalidades de caminhar e pular finalizadas, é realizada a implementação das animações para essas respectivas ações do personagem. Como foi descrito anteriormente, animações possuem estados e esses estados possuem relações entre si. Essas relações permitem que um objeto transite de um estado de animação para outro, como, por exemplo, do estado de animação “parado” para o estado “em movimento”. Para transitar entre um estado e outro, são utilizados parâmetros que podem ser alterados nos *scripts* conforme a necessidade. Neste projeto, foram utilizados parâmetros booleanos para realizar essas transições. São eles: *IsMoving* (está se movendo), *IsGrounded* (está em contato com o chão) e *IsFalling* (está caindo). A lógica utilizada para organizar as animações pode ser analisada na seguinte tabela verdade (estados de animação na Figura 16):

IsMoving	IsGrounded	IsFalling	Estado de Animação
0	0	0	*Rising* (subindo)
0	0	1	*Falling* (caindo)
0	1	0	*Idle* (parado)
0	1	1	*Idle* (parado)
1	0	0	*Rising* (subindo)
1	0	1	*Falling* (caindo)
1	1	0	*Moving* (movendo)
1	1	1	*Idle* (parado)

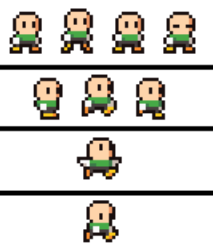

Figura 16. À esquerda, tabela verdade dos estados de animação; à direita, os estados de animação do personagem, de cima para baixo: *Idle, Moving, Rising e Falling*.

Para verificar se o personagem está em contato com o chão foi utilizado um *collider* configurado como um *trigger* (gatilho), chamado *floor trigger*. Um *trigger collider* funciona da seguinte forma: ele não possui *Rigidbody*, ou seja, objetos com corpo sólido podem passar pela área do *trigger*. Sempre que um objeto em uma cena entrar na área do *trigger*, são ativadas algumas funções. Verifica-se se o jogador está em contato com o chão da seguinte maneira: enquanto o *trigger* detectar que um objeto que faz parte do cenário entrou em sua área (através do método *OnTriggerStay2D*), ele mantém o valor do parâmetro *grounded* (que está no *script* do personagem) como *true*; e sempre que o *trigger* detectar que um objeto identificado como parte do cenário saiu de sua área (através do método *OnTriggerExit2D*), ele define *grounded* como *false*. A área do *trigger collider* foi posicionada aos pés do personagem, como pode ser visto na Figura 17.

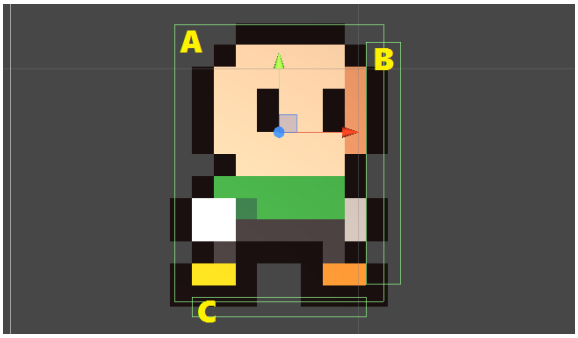

Figura 17. Os três *colliders* do personagem, representados por retângulos. Collider A: *Rigidbody* do personagem. Collider B: *wall trigger*. Collider C: *floor trigger*.

Um ponto importante a ser destacado é como um *trigger* identifica que um objeto que entrou em sua área faz parte do cenário. Todo *GameObject* possui um atributo opcional chamado *tag* (etiqueta), e *tags* podem ser determinadas pelo desenvolvedor. O personagem controlado pelo jogador tem a *tag Player*, que é utilizada somente por ele. Já os elementos do cenário, como plataformas e paredes, compartilham a *tag Level Block* (bloco de fase). No caso do *trigger* citado no parágrafo acima, ele irá reagir sempre que identificar que o objeto que entrou ou saiu de sua área possui essa *tag*.

Com o personagem em movimento, surge a necessidade de a câmera acompanhá-lo enquanto percorre a fase. A implementação foi realizada visando a um movimento suave da câmera acompanhando o personagem. Para isso, primeiro foi preciso obter duas posições: a posição atual da câmera e a nova posição do jogador em movimento. Em seguida, foi utilizada a função *Vector3.SmoothDamp*, que compara as duas posições recebidas e executa o movimento suavizado de acordo com o valor recebido no parâmetro *smoothness* (suavidade), conforme a

Figura 18.

```
15          Vector3 currentPosition = new Vector3 (
16                                        transform.position.x,
17                                        transform.position.y,
18                                        -10f
19                                    );
20          Vector3 targetPosition = new Vector3 (
21                                       playerTransform.position.x,
22                                       playerTransform.position.y,
23                                       -10f
24                                   );
25
26          transform.position = Vector3.SmoothDamp (
27                                   currentPosition,
28                                   targetPosition,
29                                   ref velocity,
30                                   smoothness
31                               );
```

Figura 18. Comandos utilizados no método *FixedUpdate* da câmera que acompanha o jogador, com o propósito de suavizar o movimento da câmera.

Durante o período do desenvolvimento, foram realizadas algumas expansões no tamanho da fase. As texturas do jogo foram modificadas para serem influenciadas por iluminação. Por padrão, texturas bidimensionais não têm essa característica. Para mudar isso, foi preciso criar um novo material no projeto. Na Unity, materiais são utilizados para aplicar texturas em objetos 3D ou para modificar suas propriedades gráficas. Objetos 2D, por padrão, não precisam utilizar materiais para terem texturas no jogo. Foi criado um novo material, utilizando o tipo de *shader Sprites/Diffuse*, e esse material foi adicionado a todas as texturas utilizadas no jogo. A partir disso, foi possível adicionar várias fontes de iluminação ao longo do cenário, iluminando não somente o cenário, mas também o personagem.

O próximo objetivo de desenvolvimento foi implementar uma forma de o jogador se transformar em morcego. Nesse ponto, percebeu-se que haveria a necessidade de separar a forma padrão do personagem (a forma humana) da nova forma de morcego. Foram criados dois *GameObjects* como filhos do objeto *Player*, um deles chamado *Human Form* (forma humana) e o outro chamado *Bat Form* (forma morcego). Os componentes do personagem foram divididos entre esses dois objetos filhos, de forma que cada um contivesse somente seus respectivos componentes. Como a forma humana tem dimensões maiores que a forma de morcego, cada objeto ficou com seu respectivo *collider*. O *script* com as mecânicas do jogador foi movido para a forma humana, e um novo *script* foi criado para a forma de morcego. Ambas as formas compartilham o mesmo *Rigidbody* do seu objeto pai, o *Player*.

Com essa divisão dos componentes, realizar a transição entre a forma humana e a forma de morcego do personagem se tornou mais fácil. Na forma humana, foi programado que, quando o jogador apertar a tecla *shift*, o personagem realize a transformação, que ocorre da seguinte forma: o objeto *Bat Form* é ativado; no seu script é executado o método *StartFlight* (iniciar voo), e o objeto *Human Form* se desativa. No *script* da forma de morcego, o método *StartFlight* altera os valores do *Rigidbody* do personagem, de modo que o personagem transformado em morcego realize um movimento de voo em linha reta, no sentido no qual o personagem está direcionado. A duração do voo é de valor fixo, ou seja, quando a duração acaba, o personagem volta a se transformar em humano.

Wall grab (segurar-se em paredes) e *wall jump* (pular em paredes) foram as últimas habilidades implementadas no personagem. *Wall grabbing* permite que o personagem, ao entrar em contato com uma parede durante um salto, se agarre na parede e use esse atrito para diminuir sua velocidade de queda. *Wall jumping* é a habilidade que permite que o personagem, enquanto está utilizando a habilidade *wall grabbing*, utilize a parede na qual está se segurando para se impulsionar e pular na direção oposta. Por conseguinte, a segunda habilidade é dependente da primeira, então a implementação iniciou-se com a habilidade de segurar-se em paredes.

Para desenvolver a habilidade *wall grab*, foi criado um novo *trigger collider* para detectar quando o personagem entra em contato com alguma parede. Esse *collider*, chamado *wall trigger*, pode ser observado na Figura 10. O *wall grab* funciona da seguinte forma: é desativado o efeito da gravidade no *Rigidbody* do personagem e, ao invés do personagem cair com a força gravitacional, ele cai com uma velocidade fixa, mais lenta do que cai normalmente, para simular o atrito que o personagem está gerando ao segurar-se na parede. Além de estar próximo a uma parede, também é preciso satisfazer outros requisitos para que o personagem possa utilizar a *wall grab*, tais como: estar se movendo em direção à parede, estar no ar (ou seja, durante um salto) e estar caindo. Com esses requisitos cumpridos, o jogador pode realizar o *wall jump* apertando a barra de espaço.

A implementação da habilidade *wall jump* é diferente da implementação do pulo normal. Como é uma habilidade que deve ser utilizada mais de uma vez em uma única jogada, é preciso fornecer um certo tempo após seu uso para que o jogador possa se preparar para o próximo *wall jump*. Essa função, conforme mostra a Figura 19, executa uma série de comandos. Primeiro, a gravidade do jogador é reativada, pois a habilidade a desativa, e o controle do personagem é removido do jogador. A remoção do controle é feita para garantir que o *wall jump* atinja o máximo de altura do salto antes de retornar o controle ao jogador. Após isso, é alterada a direção a qual o personagem está se dirigindo. Em seguida, são aplicadas as forças que fazem o personagem saltar adiante. Finalmente, depois do tempo de espera haver passado, o controle é finalmente retornado ao jogador.

```
37    IEnumerator WallJump () {
38        if (wallJumpCommand) {
39            ResetPlayerGravity ();
40
41            playerScript.SetFreeControl (false);
42
43            int directionMultiplier = playerScript.IsFacingRight () ? -1 : 1;
44
45            playerScript.FaceDirection (directionMultiplier);
46
47            playerRigidBody.velocity = Vector2.zero;
48            playerTransform.Translate (new Vector3 (0.1f * directionMultiplier, 0f, 0f));
49            playerRigidBody.AddForce (new Vector2 (4f * directionMultiplier, 10f) * wallJumpForce);
50
51            wallJumpCommand = false;
52
53            yield return new WaitForSeconds(0.4f);
54
55            playerRigidBody.velocity = Vector2.zero;
56            playerScript.SetFreeControl (true);
57        }
58    }
```

Figura 19. Função que executa a habilidade *wall jump* do personagem.

Para indicar o final da fase, foi adicionado na última plataforma do cenário um castelo com um portão. Para chegar até esse castelo, o jogador precisa fazer uso de todas as habilidades disponíveis ao personagem para poder ultrapassar os obstáculos distribuídos ao longo da fase.

Considerações finais

Os objetivos definidos na proposta geral do jogo foram alcançados. As mecânicas de *gameplay* referentes às habilidades do personagem foram implementadas e o design da fase cujo objetivo era exigir o uso de todas essas habilidades para completá-la cumpriu esse papel em sua implementação. Ao final deste trabalho, obteve-se o protótipo finalizado de Vlad, the Platformer, que serve como base de um jogo 2D do gênero plataforma.

A experiência obtida com a realização deste trabalho proporcionou uma nova perspectiva sobre o que consiste o desenvolvimento de jogos. Jogos são sistemas complexos e, assim como qualquer *software*, quanto mais ambiciosa for a sua proposta, maior será o grau de complexidade da sua implementação.

O que mais diferencia um jogo de um *software* tradicional em seu desenvolvimento é o fator criativo. O objetivo principal de um jogo é entreter, e isso é feito através de aspectos que se beneficiam de um fator criativo de qualidade: o visual, o auditivo e o interativo. Esse objetivo não é cumprido somente com esses três aspectos trabalhados individualmente, mas em conjunto. Trabalhar em um jogo exige que o desenvolvedor busque soluções criativas e formas de fazer com que diferentes sistemas e funcionalidades se comuniquem corretamente e permaneçam em sintonia. Em conclusão, a etapa de criação de um jogo não reside somente em sua concepção, mas perdura durante todo o seu desenvolvimento.

Neste trabalho, as áreas que obtiveram maior foco foram programação, criação de fase e animações do personagem. Como foram utilizadas texturas disponibilizadas gratuitamente, o desenvolvimento do aspecto gráfico do jogo teve seu esforço reduzido. Se os recursos gráficos utilizados no projeto fossem criados a partir do zero, o tempo e esforço necessários para alcançar o que foi concluído seria inevitavelmente maior. A parte sonora do jogo não foi trabalhada, pois o foco do protótipo foi sobre as mecânicas do jogo.

Como sugestão de trabalhos futuros, pode-se citar: o desenvolvimento de inimigos, que envolveria trabalhar com inteligência artificial, além da implementação de um sistema de combate e de pontos de vida para o personagem principal, para que ele possa enfrentar tais inimigos. Também podem ser implementadas novas fases, menus de interface e outros elementos aos cenários, como plataformas móveis e armadilhas.

Referências

CHATFIELD, T. Videogames now outperform Hollywood movies [Internet]. *The Guardian,* 2009. Disponível em: www.theguardian.com/technology/gamesblog/2009/sep/27/videogames-hollywood. Acesso em 8, nov, 2016.

DUTTON, F. What is Indie? [Internet]. Eurogamer, 2012. Disponível em: www.eurogamer.net/articles/2012-04-16-what-is-indie. Acesso em 8, nov, 2016.

EPIC GAMES. Epic Games [Internet]. 2016. Disponível em: www.epicgames.com. Acesso em 10, out, 2016.

FLEURY, Afonso; NAKANO, Davi; CORDEIRO, José H. D. *Mapeamento da Indústria Brasileira de Jogos Digitais* [Internet]. Pesquisa do GEDIGames, NPGT, Escola Politécnica, USP, para o BNDES, 2014.

GUIMARÃES NETO, Ernane; LIMA, Leonardo. *Narrativas e personagens para jogos*. São Paulo: Érica, 2014, 152p.

IBOPE. Conheça as principais características de quem joga videogame no Brasil [Internet]. 2012. Disponível em: www.ibope.com.br/pt-br/noticias/Paginas/Conheca-as-caracteristicas-de-quem-joga-videogame-no-Brasil.aspx. Acesso em 10, out, 2016.

LEWIS, M. e Jacobson, J. Game Engines in Scientific Research, *ACM*, Vol. 45, No. 1. 2002.

RABIN, S. Introdução ao desenvolvimento de games, v. 1. Cengage Learning, 2ª ed. 2012.

TONÉIS, Cristiano N.; FRANT, Janete Bolite. *Do desenvolvimento e criação de puzzles para a produção de conhecimentos nos jogos digitais.* Teccogs: Revista Digital de Tecnologias Cognitivas, TIDD | PUC-SP, São Paulo, n. 11, p. 95-114, jan-jun. 2015.