Desenvolvimento com Motores de Jogos I

Aula 08 - Animação de Sprites – Parte II

Apresentação

Fala, pessoal! Voltamos a nos encontrar, agora para a Aula 08 da disciplina de Desenvolvimento com Motores de Jogos I. Nesta aula, mais ainda que nas outras, podemos dizer que prosseguiremos de onde paramos com o nosso projeto, pois continuaremos o assunto iniciado exatamente na aula passada!

A parte de animação é bastante extensa e, mesmo que nós cubramos apenas uma parcela, em razão de estarmos trabalhando apenas com 2D, precisamos dividir todo o conteúdo ao longo de duas aulas a fim de não ficar muito cansativo para vocês. Para esta aula, então, sobrou o conteúdo referente a transições, incluindo, ainda, as Blend Trees!

Vimos, na aula passada, que a animação por *keyframes* segue um intervalo definido, de acordo com a sua *timeline*. Veremos, na aula de hoje, como podemos fugir um pouco desse modelo, criando uma nova animação, a qual reaja, por exemplo, à velocidade vertical na qual o usuário está no momento. São aí que surgem as Blend Trees.

Além disso, há um outro assunto muito importante que não tivemos tempo de abordar na primeira aula: as transições. Vimos como podemos importar três Sprite Sheets, mas só tivemos a oportunidade de utilizar uma. E, objetivando utilizá-la, trabalhamos apenas com a transição de **Entry** para a animação criada, ativada assim que o objeto é inicializado, sendo essa uma transição básica, criada automaticamente pelo Unity quando adicionamos uma nova animação em um controlador vazio.

Veremos, nesta aula, como podemos adicionar outras transições e como podemos configurá-las para as animações se interromperem e o personagem poder, na maior parte do tempo, estar fiel à movimentação que está sendo executada. Aprenderemos tanto as modificações que serão necessárias na própria máquina de estados de nosso Animation Controller quanto as modificações que precisaremos fazer no script para haver uma comunicação entre o controlador e o personagem.

Sobrou bastante assunto da aula passada, hein? Mas, agora, com outra aula inteirinha só para isso, com certeza teremos tempo de abordar tudo detalhadamente, para vocês saírem daqui craques da animação! Vamos juntos?

Objetivos

Ao final desta aula, você deverá ser capaz de:

- Criar transições entre os diversos estados do Animation Controller;
- Desenvolver scripts capazes de interagir com o Animator;
- Criar e utilizar Blend Trees para animar elementos baseados em parâmetros.

1. Recapitulando...

Na aula passada, conhecemos as animações através de *keyframes* e aprendemos como essas animações são desenvolvidas no Unity. Adicionamos ao nosso personagem um Animator, componente responsável por lidar com animações no Unity 3D. Em seguida, a fim de esse Animator poder funcionar adequadamente, criamos, para adicionar a ele como um elemento de controle, um Animator Controller. Também alteramos as configurações do animator para que ele se tornasse relativo à física, tendo suas atualizações juntamente ao FixedUpdate.

Em seguida, importamos três Sprites Sheets para o nosso jogo, configurando-as para serem do tipo **Multiple** e, na sequência, cortamos cada um dos sprites na Sprite Sheet de acordo com o Grid e o tamanho individual de cada imagem utilizada para compô-la. Com isso, tivemos acesso a um conjunto de sprites, parte de cada Sprite Sheet, a partir dos quais poderíamos criar as nossas animações.

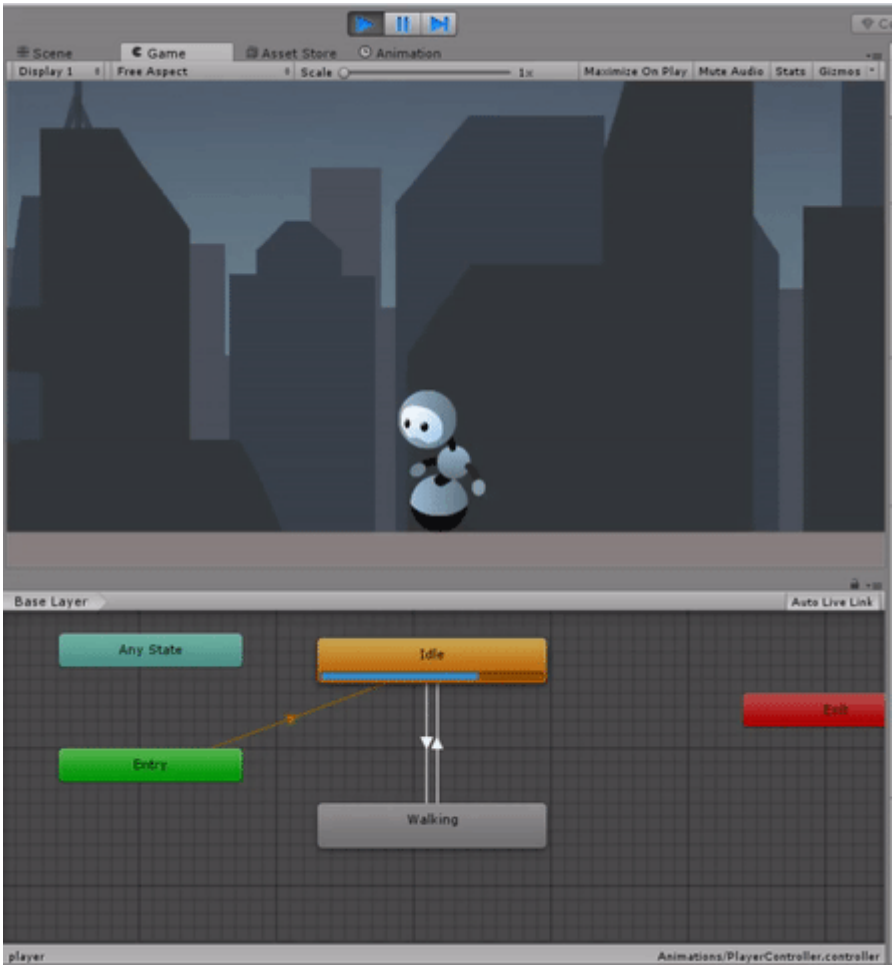
Pegamos a primeira Sprite Sheet, a da animação de Idle, e desenvolvemos, a partir dessa Sprite Sheet, a nossa primeira animação. Para isso, criamos uma nova animação chamada Idle e adicionamos a ela cada um dos sprites recortados de nossa Sprite Sheet como sendo um *keyframe* cada. Em seguida, já no final da aula, reconfiguramos o sample para um valor menor, a fim de diminuir a velocidade da animação, que estava baseada em 60 samples, o valor padrão do Unity.

Com isso, concluímos a aula passada e chegamos até o ponto que estamos. O projeto da aula passada foi disponibilizado neste link e deve ser utilizado como base, caso você não tenha desenvolvido o seu até a etapa atual, para acompanhar esta aula. A primeira modificação que faremos, a partir de agora, é a adição da segunda animação ao nosso personagem – a animação de caminhada.

2. Transição entre Animações

A próxima animação adicionada ao nosso personagem é a animação de caminhada, sendo essa animação utilizada toda vez que ele começar a se mover. A animação de *Idle* se manterá enquanto ele estiver parado e, quando começar a se mover horizontalmente, de acordo com o comando do usuário, o robô receberá a animação de caminhada, até voltar a estar parado. Veja na **Figura 1** como acontecerão essas animações:

Figura 01 - Transição entre os estados Idle e Walking do robô.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/. Acesso em: 24 de fev de 2017.

Percebe haver na definição desse comportamento, claramente, uma transição de estados declarada? Nós estamos no estado base, que é o estado de parado, ou Idle, e então passamos a um outro estado quando algo acontece. Uma transição, dentro de uma máquina de estados, é justamente isso: uma condição para, dada uma nova situação ou uma nova entrada, haver uma mudança de estados dentro da máquina. É importante também, no entanto, haver uma maneira de a máquina *sair* do estado no qual acaba de entrar, a fim de evitar que ela fique presa lá para sempre.

No caso descrito, a condição de saída para o nosso robô parar de se movimentar é justamente a movimentação se encerrar. Quando ele voltar ao estado de parado, a animação de caminhada deve se encerrar e dar lugar, mais uma vez, à animação de repouso. Parece bem simples, não? E é! Como fazemos, em qualquer um de nossos códigos, quando queremos depender de um estado? Criamos uma variável que monitore isso, correto? E é exatamente o que faremos! Teremos, em nossa máquina de estados, uma variável responsável por monitorar se há ou não movimento em nosso personagem. Caso o movimento se inicie, a alteração dessa variável indicará para o nosso controlador que deve haver também uma alteração na animação executada por ele.

Faremos melhor ainda! A fim de não ficar tão brusca a mudança, de modo que qualquer coisinha faça o nosso robozinho começar a se mover “loucamente”, utilizaremos, em vez de uma booleana, uma variável do tipo ponto flutuante, para, somente ao passar de um limite mínimo de velocidade, o robozinho começar de fato a realizar o movimento de caminhada e, também, para o movimento se encerrar um pouco antes de o robô chegar ao repouso total. Isso evita que a animação do robô pareça estar desproporcional, com muito movimento sendo realizado, mas com o robô praticamente sem sair do canto (com uma velocidade muito baixa).

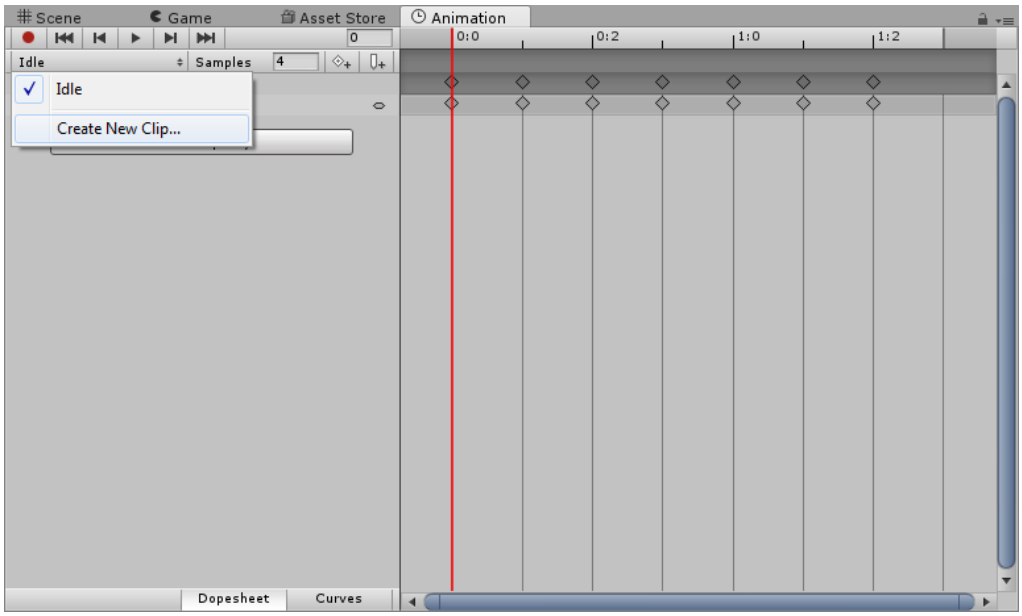
A partir dessa nossa conversa inicial, podemos observar quatro pontos importantes: precisamos criar a nova animação do robô, criar uma variável dentro de nossa máquina de estados para registrar o movimento ou não do nosso robô, adicionar a transição entre os estados criados para o nosso robô e, por fim, fazer um script que, de alguma maneira, seja capaz de se comunicar com o nosso Animator, alterando a variável criada para a animação poder realizar suas transições adequadamente. A seguir, realizaremos esses quatro passos, nessa ordem!

2.1 Criando a Animação de Caminhada

A criação da animação de caminhada não será diferente da criação da animação de Idle, vista na aula passada. Precisaremos criar um novo Animation Clip e, então, nesse novo clip, adicionar como um *keyframe* cada um dos sprites recortados da Sprite Sheet de caminhada, da mesma maneira que vimos anteriormente.

A diferença, no entanto, será na criação do clipe de animação. Dessa vez, não será mais possível simplesmente utilizar o botão create, uma vez que já temos um clipe definido e a aba Animation o carrega quando selecionamos o objeto player. Precisaremos, então, clicar no menu dropdown que contém o nome da animação e selecionar a opção “Create New Clip...” a partir disso, o que nos levará ao mesmo caminho de criação já encontrado antes. A **Figura 2** exibe esse menu.

Figura 02 - Criando um novo clipe a partir do menu dropdown.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/. Acesso em: 24 de fev de 2017.

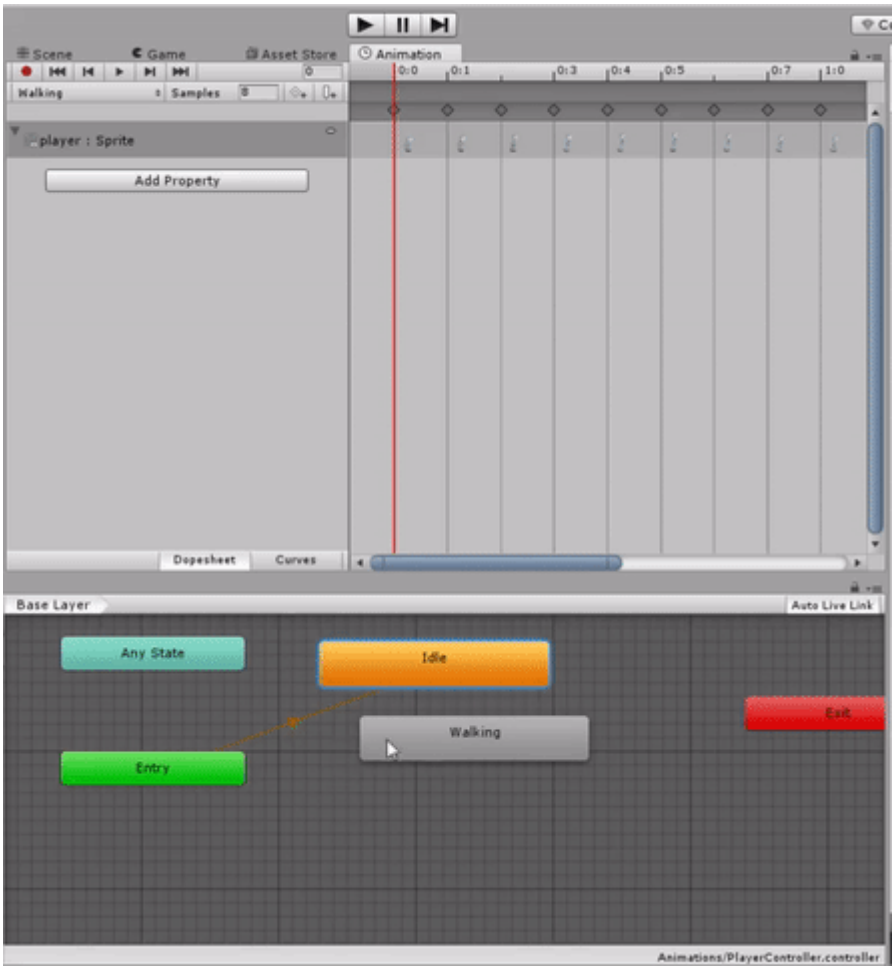
Utilizaremos o nome “Walking” para o clipe de caminhada e o salvaremos na pasta Animations, juntamente ao clipe Idle. Após criar o clipe, perceba que uma nova timeline zerada é criada e que a taxa de samples volta a ser 60. Adicionaremos, mais uma vez, os sprites recortados à nossa janela de Animation, como vimos aula passada, selecionando todos os sprites robotWalking e arrastando-os até essa janela. Isso deve criar um novo conjunto de *keyframes* nessa *timeline*, indicando cada um dos sprites de caminhada que temos em nosso projeto.

Adicionados os sprites, o próximo passo é diminuir a quantidade de samples utilizados para essa animação. Novamente, reduziremos a quantidade bastante, pois a nossa animação possui 16 frames e deve durar 2 segundos em uma rodada completa. Utilizaremos 8 para o valor de samples. Experimente com esse valor e veja, também, outro de seu agrado! Perceba, ainda, que se você utilizar velocidades diferentes para o robô, a velocidade da animação também deve ser influenciada para evitar que o movimento fique muito discrepante.

Note, na aba de Animator, que, ao criarmos uma nova animação através da aba Animation, essa nova animação é adicionada automaticamente ao nosso controlador, criando um novo quadrinho para representar o novo estado. Para fins de testes, tornaremos agora o Walking como nossa animação padrão, de modo a vê-la assim que o jogo for iniciado, a partir de Entry.

Para tanto, basta ir até a aba Animator e clicar com o botão direito no objeto Walking, fazendo um menu aparecer. Selecione a opção “Set as Layer Default State”. Ela fará o estado mudar a cor para laranja. Além disso, a transição, que antes era entre Entry e Idle, agora será entre Entry e Walking. Desse modo, ao iniciar o objeto, a animação de Walking será tocada, em vez da animação de Entry. Dê o Play no seu jogo para fazer o teste!

Figura 03 - Definindo o clipe Walking como Default e vendo sua execução no jogo.



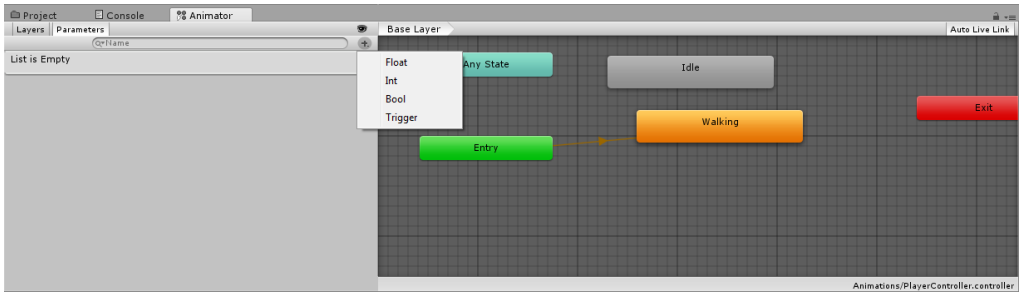
Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/. Acesso em: 24 de fev de 2017.

Agora já sabemos como podemos utilizar uma animação ou outra, a partir do estado inicial de entrada, utilizando a animação padrão escolhida. O próximo passo é saber como podemos utilizar as duas ao mesmo tempo, com base em uma transição diferente da padrão! Para fazer isso, precisaremos, antes, criar as variáveis necessárias a fim de as transições ocorrerem com sucesso. Essas variáveis são criadas na própria aba do Animator. Vamos ver como?

2.2 Criando Variáveis para as Transições

A criação de variáveis, ou parâmetros, para as transições poderem acontecer, é parte importante do desenvolvimento de animações no Unity. A fim de facilitar, foi criado um menu específico dentro da aba de Animator para lidar com essas variáveis. No lado esquerdo da camada da máquina de estados, há uma caixinha selecionada chamada Parameters. Será nesse local que criaremos os novos parâmetros necessários a fim de as transições acontecerem. Mais precisamente, há, no lado direito desse menu, próximo ao centro da View, um +, o qual nos dá as possibilidades de variáveis que temos ao trabalhar com esses parâmetros. Veja, na **Figura 4**, esse menu, com as quatro opções de variáveis trazidas por ele.

Figura 04 - Menu de criação de parâmetros com os quatro tipos disponíveis.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/. Acesso em: 24 de fev de 2017.

Float, Int, Bool e Trigger: esses são os quatro tipos de parâmetros disponíveis. Float, Int e Bool são bem claros, representando, respectivamente, ponto flutuante, inteiros e booleanos, respectivamente. A novidade está justamente no parâmetro do tipo Trigger, o qual não está diretamente relacionado a um tipo básico de dados que já conhecemos.

Um parâmetro do tipo Trigger funciona, na verdade, como um booleano, mas possui uma especificidade bem interessante. Ele se mantém sempre em falso, até algum evento ocorrer e o disparar para verdadeiro. Quando isso acontecer, o parâmetro Trigger já voltará a ser falso logo no frame seguinte, mantendo-se assim até que um novo evento qualquer ocorra. Isso pode ser interessante ao pensarmos em animações que são iniciadas a partir de qualquer estado e devem ser executadas apenas uma vez. Para esses casos, os parâmetros do tipo Trigger são muito adequados!

No caso específico do nosso jogo, no entanto, não trabalharemos com esse tipo de parâmetro. Nossas animações serão todas controladas apenas por parâmetros de velocidade vertical ou horizontal e de posicionamento. Sendo assim, para essa primeira transição, criaremos somente um parâmetro do tipo ponto flutuante, sendo esse parâmetro responsável por armazenar a velocidade na qual o personagem está se movendo, para gerenciar as transições entre o estado de parado e caminhando.

Cliquemos, então, no botão +, demonstrado na **Figura 4**, e, em seguida, selecionemos a opção Float. Isso criará uma nova variável na lista e solicitará que demos um nome a essa variável. Utilizaremos o nome *speed* para tal. A caixinha à direita do nome da variável indica seu valor inicial. Como nosso personagem começará parado, manteremos a nossa variável *speed* com o valor 0.0. Assim, já temos um parâmetro para basearmos nossas transições nele.

2.3 Criando as Transições

No começo da aula, discutimos o esboço básico da ideia de nossa animação. Vimos, a partir disso, que transições entre os estados de animação criados seriam necessárias para podermos manter o robô se movendo adequadamente. A partir da variável *speed*, criaremos, então, essas transições, para finalmente podermos trabalhar com os dois estados disponíveis.

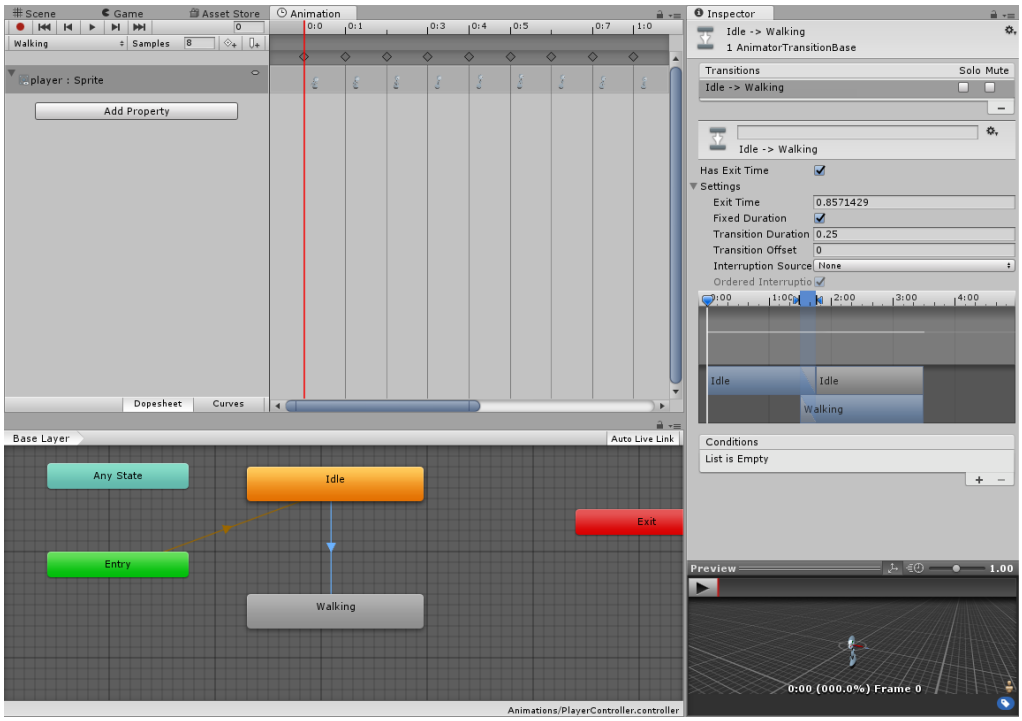
Primeiramente, alteraremos de volta o estado padrão de animação, o qual havíamos modificado para o Walking para testar a animação. É só clicar com o botão direito no bloquinho de Idle e selecionar a opção “Set as Layer Default State”. Feito isso, agora sabemos que a animação será iniciada sempre no bloquinho correto, uma vez que o personagem inicia parado.

Em seguida, criaremos nossa primeira transição entre estados. Essa transição nos levará do estado Idle ao estado Walking quando nossa velocidade for maior que 0.1. De onde vem esse valor 0.1? Bem, foi um chute! Depois, com a animação concluída, poderemos testar outros valores e ver se de fato esse foi uma boa tentativa. Para criar essa transição, devemos clicar com o botão direito no bloco do Idle e, então, selecionar a opção “Make Transition”. Ao fazer isso, uma reta cinza, com uma setinha começará a acompanhar o cursor do mouse, indicando que

devemos ligar esse bloco ao bloco para o qual desejamos fazer a transição. Nesse caso, cliquemos com o botão esquerdo no bloco Walking. Isso criará uma seta cinza entre os dois blocos, indicando uma transição.

Para facilitar a visualização do que estamos fazendo, espaçaremos um pouco mais os blocos. Para isso, basta clicarmos com o botão esquerdo em um bloco e arrastá-lo até a posição desejada. Após realizar esse procedimento da maneira que achar adequada, clique na nova transição que acabamos de criar. Isso fará o Inspector mostrar as propriedades dessa transição, como visto na **Figura 5**.

Figura 05 - Transição entre os estados Idle (padrão) e Walking selecionada e exibida no Inspector.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/. Acesso em: 24 de fev de 2017.

Perceba: a transição em si tem diversas propriedades que podem ser configuradas para tornar a transição o quão consistente desejarmos. Mas nós estamos trabalhando em 2D! E aí as transições, na verdade, servem somente para trocar a sequência de sprites com os quais estamos trabalhando (pelo menos no caso do nosso jogo). Então, nesse caso, podemos basicamente tirar a existência da transição em si e deixá-la apenas como uma mudança de estado! Para isso, a primeira coisa que devemos alterar é propriedade Has Exit Time. Podemos desmarcar essa opção, pois não há um momento definido em que devemos sair

dessa animação. Em seguida, podemos zerar a opção Transition Duration, pois não há necessidade de a transição ter qualquer duração. Isso fará a transição ser instantânea, levando o personagem de um estado para o outro diretamente.

Nesse momento, se apertarmos Play em nosso jogo veremos que o personagem executará uma vez a animação Idle, por ser a animação padrão, e, em seguida, alterará a animação para Walking, visto que há uma transição lá indicada. Veja isso acontecendo! Aperta o Play!

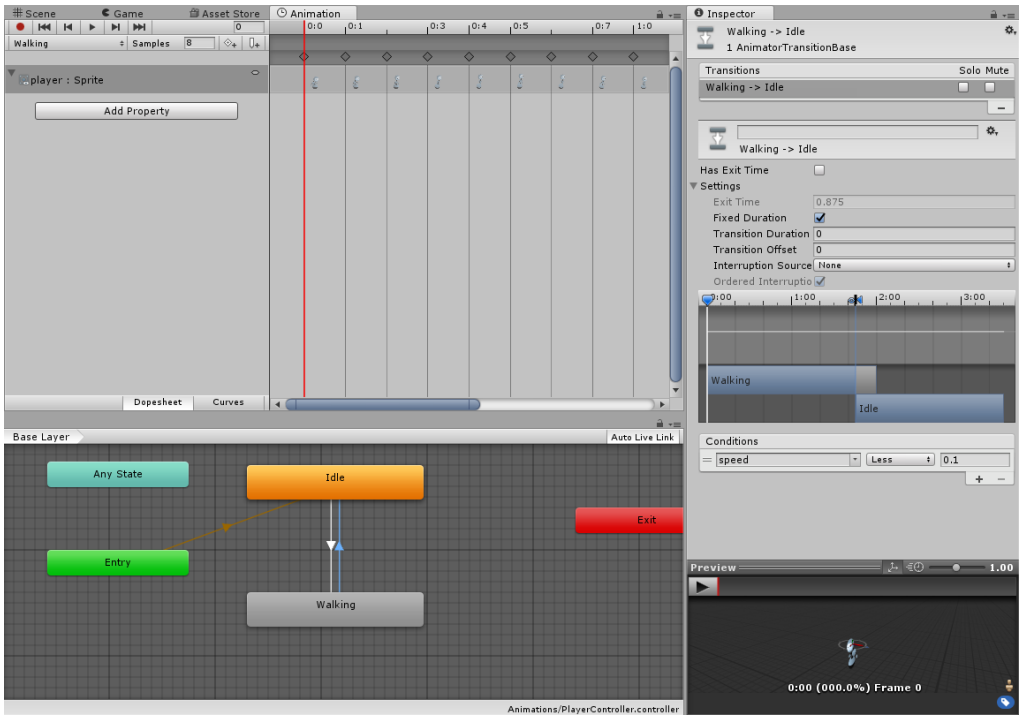
Viu? Pois é. Não é isso que queremos acontecendo em nosso jogo. Precisamos, então, dar uma condição a essa transição para ela acontecer. E essa transição será relacionada ao nosso parâmetro! Veja, na **Figura 5**, que temos uma opção chamada Conditions nas propriedades da transição. Essa opção nos permite criar uma condição para aquela transição acontecer. E adivinha baseado em quê? Exatamente! Nos parâmetros criados previamente. Clique no +, que fica abaixo da lista de condições, e o nosso parâmetro (por ser o único) imediatamente aparecerá dentro da lista de condições! Agora, podemos escolher quando a transição vai acontecer e quando não vai. Massa!

Como já havíamos conversado, a transição acontecerá quando a velocidade for maior que 0.1, correto? Então, devemos definir isso na lista de condições. O parâmetro já está escolhido – *speed*. Agora, selecionamos entre as opções maior (greater) ou menor (less) e o valor que queremos comparar. Para o nosso caso, montaremos a condição “speed greater 0.1”. Com isso, a nossa transição só acontecerá quando o personagem começar a se mover de verdade!

Ok. Isso funcionará assim que tivermos um script para alterar o valor de *speed*. Ou você pode testar isso alterando o valor manualmente, na lista da esquerda! Mas, independentemente de qual for o caso, ainda teremos um problema: não tem como voltar ao estado de Idle!

Esse problema, no entanto, vocês já sabem solucionar. Basta criar uma transição voltando e adicionar a ela uma condição, da mesma maneira como fizemos com a transição de ida, concordam? Não farei aqui o passo a passo, mas lembrem-se que, agora, a volta depende da condição “speed less 0.1” e que a transição deve partir de Walking e não de Idle! Após a configuração, seu Animator Controller deve estar parecido com o visto na **Figura 6**.

Figura 06 - Animator Controller com as transições de ida e volta entre os estados Idle e Walking.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/. Acesso em: 24 de fev de 2017.

Agora já temos como ir do estado de Idle ao de Walking e voltar. Com isso, nossa animação já pode responder corretamente aos comandos de movimentação lateral de nosso personagem! Para isso acontecer, falta apenas um detalhe. Precisamos que alguém relacione o valor do parâmetro *speed* com a velocidade do personagem, de fato. E eu tenho um ótimo candidato! O PlayerController Script.

2.4 Codificando a Realização das Transições

Agora que já temos as transições feitas e, também, o parâmetro a ser utilizado para as controlar definido, podemos partir para a codificação que de fato fará a mágica acontecer! Para isso, abriremos o nosso PlayerController e faremos algumas pequenas modificações.

A primeira delas é que precisaremos ter acesso ao nosso Animator, para alterar o nosso parâmetro adequadamente. Em seguida, precisaremos, sempre que houver algum comando de movimentação, alterar o valor do nosso parâmetro de *speed*, para a animação do personagem poder se adequar. O código para isso é bem simples e pode ser visto na **Listagem 1**, em destaque.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerController : MonoBehaviour {

    private bool jumping = false;
    private bool grounded = false;
    private bool doubleJump = false;
    private bool doubleJumping = false;
    private bool movingRight = true;

    private Rigidbody2D rigidBody;
    private Animator ani;
    public Transform groundCheck;
    public LayerMask layerMask;

    public float acceleration = 100f;
    public float maxSpeed = 10f;
    public float jumpSpeed = 500f;

    // Use this for initialization
    void Awake () {
        rigidBody = GetComponent<Rigidbody2D> ();
        ani = GetComponent<Animator>();
    }

    // Update is called once per frame
    void Update() {
        grounded = Physics2D.Linecast(transform.position, groundCheck.position, layerMask);

        if (Input.GetButtonDown("Jump")) {
            if (grounded) {
                jumping = true;
                doubleJump = true;
            } else if (doubleJump) {
                doubleJumping = true;
                doubleJump = false;
            }
        }
    }

    //Called in fixed time intervals, frame rate independent
    void FixedUpdate() {
        float moveH = Input.GetAxis ("Horizontal");

        ani.SetFloat("speed", Mathf.Abs(moveH));

        if (moveH < 0 && movingRight) {
            Flip();
        } else if (moveH > 0 && !movingRight) {
```

```
52          Flip();
53      }
54
55      if (rigidBody.velocity.x * moveH < maxSpeed) {
56          rigidBody.AddForce (Vector2.right * moveH * acceleration);
57      }
58
59      if (Mathf.Abs (rigidBody.velocity.x) > maxSpeed) {
60          Vector2 vel = new Vector2 (Mathf.Sign (rigidBody.velocity.x) * maxSpeed, rigidBody.velocity.y
61          rigidBody.velocity = vel;
62      }
63
64      if (jumping) {
65          rigidBody.AddForce(new Vector2(0f, jumpSpeed));
66          jumping = false;
67      }
68      if (doubleJumping) {
69          rigidBody.velocity = new Vector2 (rigidBody.velocity.x, 0);
70          rigidBody.AddForce(new Vector2(0f, jumpSpeed));
71          doubleJumping = false;
72      }
73
74  }
75
76  void Flip() {
77      movingRight = !movingRight;
78      transform.localScale = new Vector3((transform.localScale.x * -1), transform.localScale.y, transfo
79  }
80 }
```

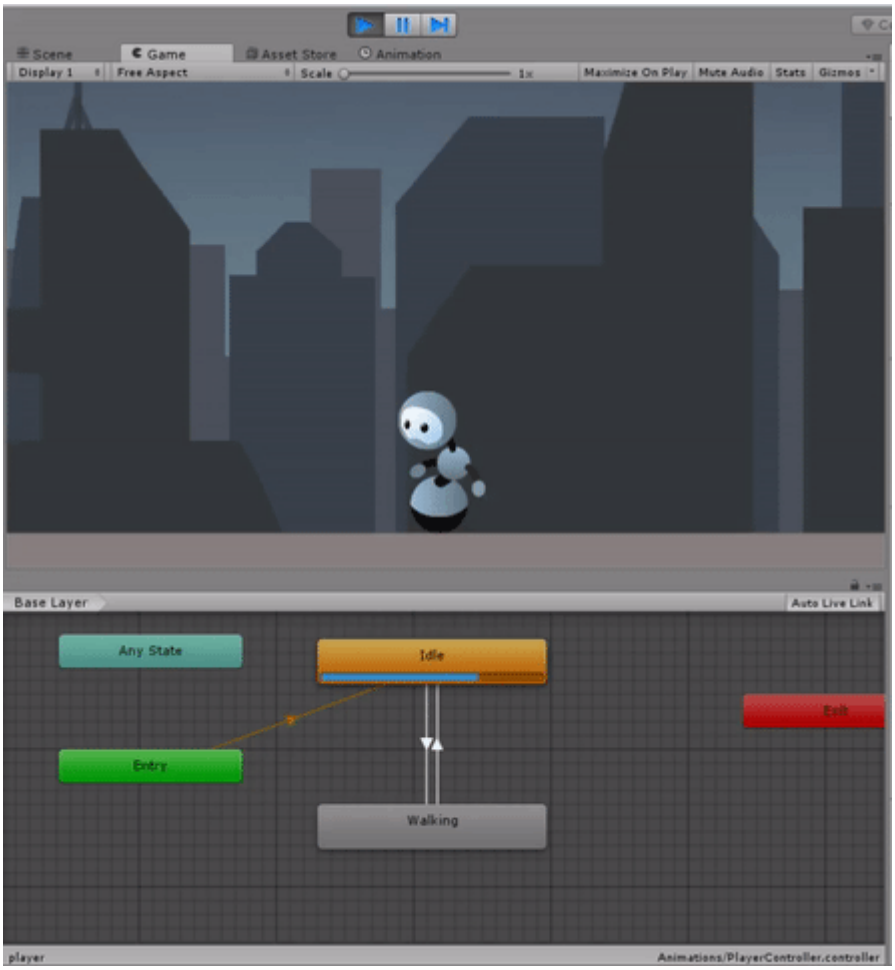
Listagem 1 - Código do PlayerController adicionando referências ao Animator e alterando o valor do parâmetro *speed*.

Veja que foram alteradas apenas três linhas de código a fim de a nossa animação passar a receber as informações adequadamente. Na primeira, criamos uma variável chamada “ani” para receber o nosso animador. Na segunda, utilizamos o método de inicialização para buscar pelo componente Animator, como já havíamos feito antes com o componente RigidBody2D. Por fim, na terceira linha, utilizamos a função matemática de módulo, a qual já havíamos estudado para passar ao parâmetro *speed* a informação de que houve uma movimentação do personagem por parte do jogador, independentemente da direção. A alteração em si foi feita através da função SetFloat, do Animator, a qual recebe o nome do Float a ser alterado e o valor que ele deve ter.

Pronto! Com isso, podemos executar o nosso jogo e vê-lo funcionar corretamente, realizando tanto as transições de ida, quanto as transições de volta. E dá para ver bem que, quando o personagem para de se mexer, este sai apenas

deslizando de acordo com a força de sua inércia e o seu baixo atrito com o solo, definidos há algumas aulas. Ficou muito bom! \o/

Figura 07 - Transição entre estados acontecendo à medida que há um input do usuário.

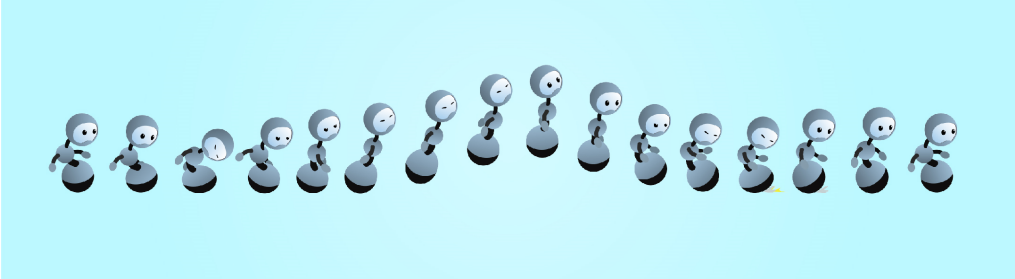


Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/. Acesso em: 24 de fev de 2017.

Agora só temos mais um estado que precisamos cuidar antes de poder seguir adiante com o nosso projeto – o pulo. Estudamos, na aula sobre mecânicas de pulo, como criar um pulo bem elaborado para o nosso personagem e aprendemos algumas variações, as quais não incluímos no projeto. Utilizaremos, agora, uma técnica um pouco mais complexa para animar o pulo corretamente. Essa técnica é conhecida como Blend Tree. Veremos um pouco mais sobre ela a seguir. Aguenta aí! ;)

3. Utilizando Blend Trees

A animação do pulo é uma animação um pouco diferente das demais. Digo isso por dois principais motivos: o primeiro é que o pulo tem uma parte na qual a movimentação não acontece muito, quando o personagem já se impulsionou no chão e está simplesmente no ar; o segundo é porque a animação de pulo, usualmente, também interfere na animação de queda. Se você está em uma parte mais alta do cenário e simplesmente cai, você não deve passar pela animação de pulo, mesmo tendo saído do chão, embora seja esse o caso essencial para haver um pulo. Nesses casos, em que há uma simples queda, a animação deve ser capaz de se adequar para o personagem não parecer estar pulando. Por essas razões, vale a pena construir uma Blend Tree.



Uma Blend Tree, como o nome indica, é uma árvore de estados diversos de animações, as quais podem se misturar entre si, criando uma animação mais fluida e complexa a partir de pequenos estados definidos anteriormente. No caso de 3D, em animações com esqueletos, por exemplo, podemos utilizar as Blend Trees para adequar as transições de maneira mais delicada e chegar a resultados visualmente melhores. Já em 2D, como não há a mistura entre os estados, podemos simplesmente utilizar uma Blend Tree para criar uma faixa diversificada de valores , de acordo com a qual as animações se alteram, a fim de criar uma animação final, capaz de se adequar a situações variadas, como pulos mais altos ou quedas, independentemente do tamanho delas. Ao criarmos pequenas faixas de valores na qual os frames serão alterados, o controle da animação passa a ser de acordo com isso e não depende mais de tempo ou de uma lista de condições.

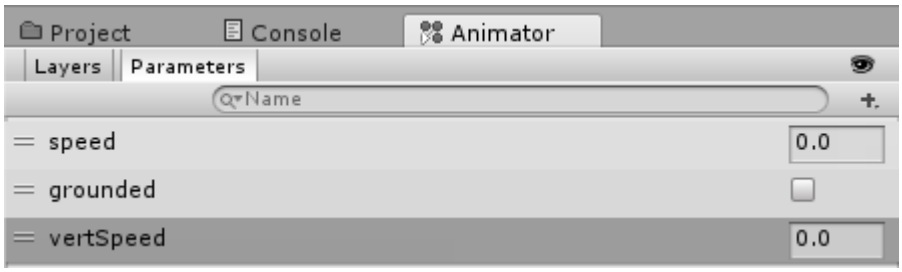
Podemos considerar, também, que a utilização de uma Blend Tree nesses casos economiza espaço em nossa máquina de estados, tornando a compreensão um pouco mais fácil. Como quase tudo realizado até agora, há mais de uma maneira de abordar o comportamento que desejamos reproduzir. Utilizaremos esta, no entanto, para fins de aprendizado! Vamos pôr a mão na massa?

3.1 Preparando o Animator Controller

A fim de criarmos a animação do nosso personagem para o pulo e a queda através da Blend Tree, utilizaremos os mesmos princípios iniciais já realizados anteriormente. Precisaremos de parâmetros em nossa animação que serão responsáveis por controlar o momento do pulo, assim como precisaremos alterar o nosso script para esses parâmetros serem atualizados pelo código, a cada novo frame.

Começaremos, então, pelos nossos parâmetros! Dentro da aba Animator, com o nosso Animator Controller selecionado, adicionaremos dois novos parâmetros ao nosso controlador: o booleano grounded e o float vertSpeed. Faremos isso da mesma maneira como fizemos anteriormente. O resultado final pode ser visto na **Figura 8**.

Figura 08 - Novos parâmetros adicionados ao Animator Controller.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/. Acesso em: 24 de fev de 2017.

O parâmetro grounded será o responsável por dizer à animação se estamos ou não no chão. Esse parâmetro será obtido a partir da variável grounded, a qual temos em nosso código e serve justamente para isso. Assim, poderemos saber quando iniciar a animação de pulo. O parâmetro grounded será a base do nosso controlador para saber quando trocar de animação!

O segundo parâmetro, o vertSpeed, indicará a velocidade que estamos no ar. Isso será importante para a posição do nosso personagem poder se adequar em relação à velocidade que ele está. Através desse parâmetro, é possível saber se estamos iniciando o pulo, chegando ao ponto mais alto, começando a descer ou, até mesmo, já chegando no chão. Esse parâmetro será a base da nossa Blend Tree para saber quando trocar de animação internamente!

Agora que já criamos os parâmetros, qual o próximo passo? A resposta já foi dada anteriormente!

Acertou quem pensou em definir os valores a partir de nosso código! Agora que já temos os parâmetros da animação, devemos alterar o nosso código principal a fim de esses parâmetros serem atualizados de acordo com as variáveis que lhe dizem respeito! Utilizaremos os mesmos métodos de setBool e setFloat no nosso Animator. As alterações podem ser vistas em destaque, na **Listagem 2**.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerController : MonoBehaviour {

    private bool jumping = false;
    private bool grounded = false;
    private bool doubleJump = false;
    private bool doubleJumping = false;
    private bool movingRight = true;

    private Rigidbody2D rigidBody;
    private Animator ani;
    public Transform groundCheck;
    public LayerMask layerMask;

    public float acceleration = 100f;
    public float maxSpeed = 10f;
    public float jumpSpeed = 500f;

    // Use this for initialization
    void Awake () {
        rigidBody = GetComponent<Rigidbody2D> ();
        ani = GetComponent<Animator>();
    }

    // Update is called once per frame
    void Update() {

        if (Input.GetButtonDown("Jump")) {
            if (grounded) {
                jumping = true;
                doubleJump = true;
            } else if (doubleJump) {
                doubleJumping = true;
                doubleJump = false;
            }
        }

    }

    //Called in fixed time intervals, frame rate independent
    void FixedUpdate() {
        float moveH = Input.GetAxis ("Horizontal");

        ani.SetFloat("speed", Mathf.Abs(moveH));

        grounded = Physics2D.Linecast(transform.position, groundCheck.position, layerMask);
        ani.SetBool("grounded", grounded);
        ani.SetFloat("vertSpeed", rigidBody.velocity.y);
```

```
52
53      if (moveH < 0 && movingRight) {
54          Flip();
55      } else if (moveH > 0 && !movingRight) {
56          Flip();
57      }
58
59      if (rigidBody.velocity.x * moveH < maxSpeed) {
60          rigidBody.AddForce (Vector2.right * moveH * acceleration);
61      }
62
63      if (Mathf.Abs (rigidBody.velocity.x) > maxSpeed) {
64          Vector2 vel = new Vector2 (Mathf.Sign (rigidBody.velocity.x) * maxSpeed, rigidBody.velocity.y
65          rigidBody.velocity = vel;
66      }
67
68      if (jumping) {
69          rigidBody.AddForce(new Vector2(0f, jumpSpeed));
70          jumping = false;
71      }
72      if (doubleJumping) {
73          rigidBody.velocity = new Vector2 (rigidBody.velocity.x, 0);
74          rigidBody.AddForce(new Vector2(0f, jumpSpeed));
75          doubleJumping = false;
76      }
77
78  }
79
80  void Flip() {
81      movingRight = !movingRight;
82      transform.localScale = new Vector3((transform.localScale.x * -1), transform.localScale.y, transfo
83  }
84 }
```

Listagem 2 - Atualização de parâmetros de animação.

Aproveitamos, ainda, para alterar um pouco o nosso código, movendo o teste de grounded, que anteriormente estava no método Update, para o método FixedUpdate. Fizemos isso por duas razões: o teste de grounded utiliza o motor de física, o qual sempre é atualizado de acordo com o FixedUpdate, e as animações também se baseiam no FixedUpdate, de acordo com o que foi previamente definido, para serem atualizadas. Além disso, configuramos a alteração dos dois parâmetros necessários.

Perceba agora que, ao apertarmos espaço, o valor de grounded que se inicia em true se altera para false, enquanto o vertSpeed parte de uma velocidade por volta de 8, até chegar a 0 e retornar ao chão com -8. Com isso, estamos prontos para montar

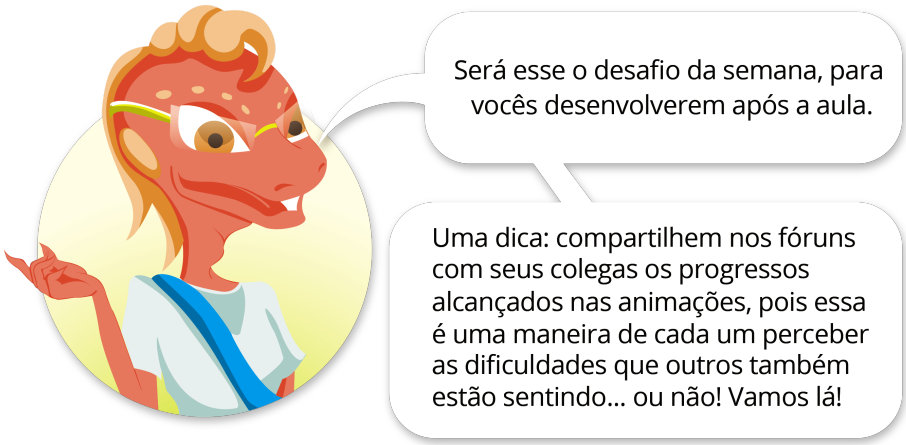
a nossa Blend Tree e configurá-la para se alterar de acordo com o valor de velocidade vertical!

Vamos adiante!

3.2 Criando as Animações Necessárias

Em razão de já termos o controle adequado no nosso controlador, o próximo passo de nosso desenvolvimento é criar cada uma das animações que fazem parte de nossa Blend Tree. Para isso, devemos selecionar a nossa aba de Animation e, em seguida, selecionar o jogador. Com isso, voltaremos à visão das animações já criadas que temos. Do mesmo modo como fizemos para adicionar a animação de Walking, criaremos um novo clipe para o primeiro sprite da animação de pulo, outro para o segundo, outro para o terceiro, e assim por diante! Na verdade, precisaremos criar X novas animações, a fim de comportar todos os pedacinhos de nosso pulo! Chamaremos as animações de Jumping1 até Jumping12 e colocaremos em cada uma delas apenas um sprite, na ordem em que foram cortados. Não esqueça, também, de movê-las para a pasta Animations após finalizar a criação de todas!

Note que não utilizamos todos os sprites! Os três últimos, na verdade, correspondem à chegada do personagem ao chão. Precisaríamos, então, de outro estado de animação para isso. Na verdade, isso é algo que vocês já conseguiriam fazer sem problemas, utilizando os três últimos sprites!



Uma vez que tenhamos as doze animações criadas, contendo um sprite em cada uma delas, podemos passar para a criação da Blend Tree em si, utilizando cada uma dessas animações de um sprite a fim de fazer uma movimentação de pulo adequada ao personagem.

3.3 Criando a Blend Tree

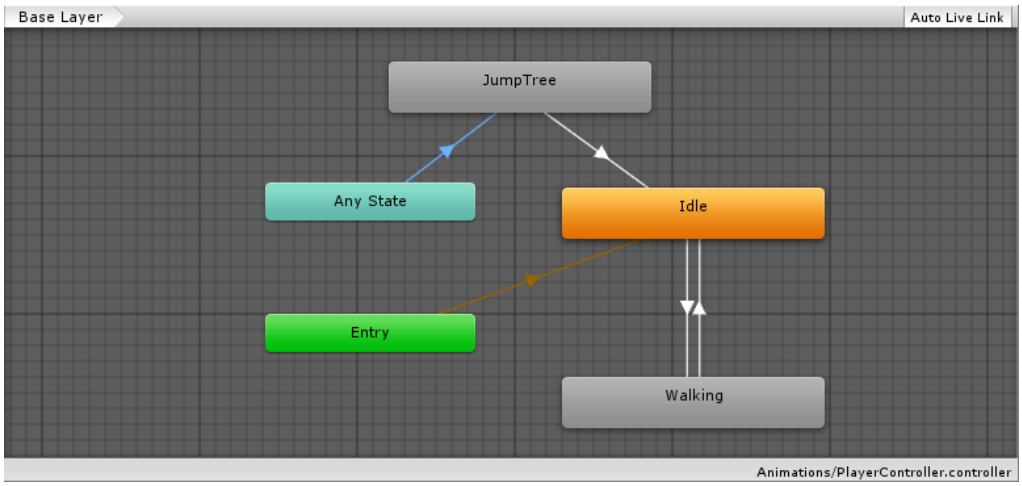
Agora que já estamos com tudo pronto, vamos à parte final da nossa animação de pulo. Para criar uma nova Blend Tree, vá até o Animator Controller de seu personagem, na aba Animator e, então, clique com o botão direito em algum lugar vazio de sua máquina de estados e escolha a opção Create -> From New Blend Tree. Feito isso, um novo estado aparecerá em seu controlador. Clique nele para selecioná-lo e, depois, na aba Inspector, altere o seu nome para JumpTree.

Precisaremos, antes mesmo de configurar a árvore, adicionar transições para ser possível chegarmos a esse estado e, então, sairmos dele. Já vimos o caminho para isso! Precisamos apenas decidir de onde entraremos nesse estado. E aí tem um ponto que também já passamos nessas aulas: o estado Any State, lembram?

O estado Any State indica que, a partir de qualquer estado, poderemos fazer uma transição para o estado escolhido. É exatamente isso que queremos, concorda? Como poderemos pular tanto a partir do estado de descanso quanto do estado de caminhada, é válido utilizarmos o Any State para ser a base de nossa árvore. Cria-se uma transição do estado Any State para a nossa árvore, utilizando o botão direito em Any State -> Make Transition.

Feita a transição, não esqueçam de remover a duração (Transition Duration) dela em Settings, como vimos anteriormente. Em seguida, deve-se fazer uma transição da árvore para o estado de Idle, para o qual voltaremos ao cair de um pulo, mesmo que por um só instante, antes de voltar a correr. Façam também essa transição e não esqueçam de remover a duração! Seu Animator Controller, após essas modificações, deve ficar parecido com o visto na **Figura 9**.

Figura 09 - Animator Controller com a nova Blend Tree de pulo.

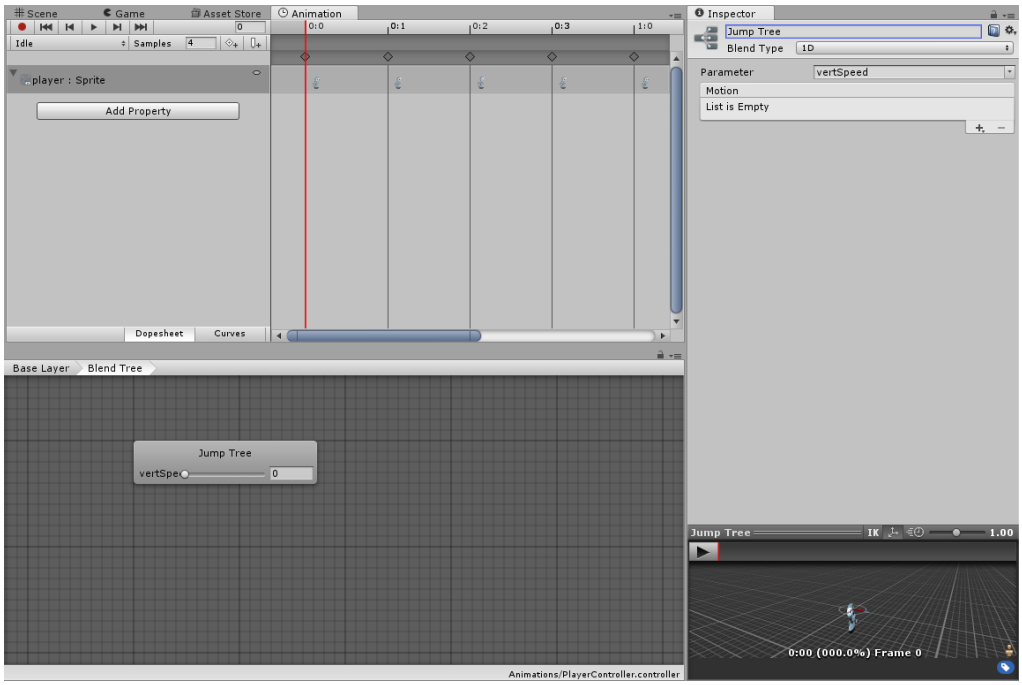


Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/. Acesso em: 24 de fev de 2017.

Pronto! Agora que temos a nossa árvore, além das transições de entrada e saída, podemos configurar a Blend Tree em si, a partir das animações criadas com os doze sprites iniciais do nosso pulo.

Para acessar o menu de configuração da Blend Tree, basta clicarmos duas vezes nela, a partir do Animator. Feito isso, uma nova interface se abrirá dentro dessa própria aba, mostrando a palavra speed e um valor 0, com um slider. Ao clicar nessa caixinha, o Inspector mostrará algumas propriedades da Blend Tree. Configure-a para ser 1D, o que significa que ela será baseada apenas em um valor, e escolha, logo abaixo, o parâmetro como vertSpeed, no lugar de speed. Com isso, suas configurações devem ficar como vistas na Figura 10.

Figura 10 - Configurando o Blend Type e o Parameter da Blend Tree.

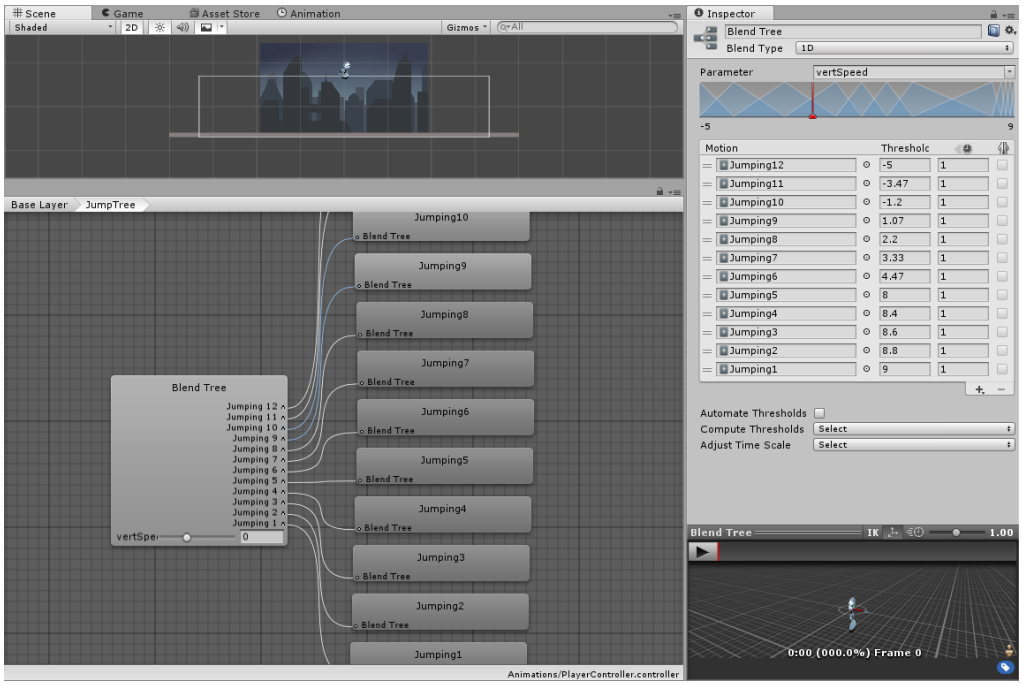


Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/. Acesso em: 24 de fev de 2017.

Feito isso, a nossa Blend Tree já estará configurada corretamente. O próximo passo é definir quais serão as animações necessárias e o valor a partir do qual cada uma delas deverá assumir o controle do sprite do personagem. Para isso, no menu Motion, visto na **Figura 10**, adicionaremos doze novos Motion Fields. É bem simples! Só clicar no + e escolher a opção Add Motion Field, doze vezes!

O passo seguinte é escolher qual animação será utilizada em cada um desses Motion Fields. Para fazer esse passo, clique na bolinha desenhada em cada linha, podendo tal bolinha ser vista na **Figura 11**, e escolha uma animação dentre as que acabamos de criar. Ao terminar, você deve ter uma lista de Motion Fields com as animações Jumping1 a Jumping12. Feito isso, desmarque o campo Automate Tresholds a fim de podermos configurar manualmente o valor de vertSpeed para que cada animação assuma. Por fim, configure os valores de acordo com os exibidos na **Figura 11**!

Figura 11 - Blend Tree com todas as Motions configuradas.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/. Acesso em: 24 de fev de 2017.

"E de onde vêm esses valores?", você me pergunta. Tentativa e erro! A melhor maneira de configurar esses thresholds, a partir do vertSpeed, é através de tentativa e erro! Estou colocando os valores para vocês a partir do que eu percebi do meu pulo. Com a força que escolhi e o peso do meu personagem, a velocidade vertical de saída dele é em torno de 9. Assim, faço os primeiros passos acontecerem logo a partir desse valor. Começando com 9, 8.8 em seguida, 8.6, até chegarmos a 8. Essa animação representa-o fazendo um esforço para subir e pode ser mantida por mais tempo, até ele de fato começar a parar de subir. Aí vamos utilizando as próximas animações até chegar ao valor -5, que corresponde à animação dele esperando o fim da queda.

Pode ser que você precise adequar esses valores, à medida que o seu pulo seja mais rápido ou devagar. Preste bem atenção!

E não esqueça, também, de alterar o mínimo e o máximo dos valores, caso precise alterar os thresholds!

Pronto! Com isso, nosso personagem já será capaz de responder ao pulo, simples ou duplo, e a qualquer queda encontrada durante o cenário! Dá uma testada lá no Play para ver como ficou! A fim de apreciar melhor a animação de queda, modifique a posição inicial do personagem para ele começar o jogo já caindo, como visto na **Figura 11**. Vejamos o resultado final do que desenvolvemos juntos na **Figura 12**!

Figura 12 - Jogo com as três animações desenvolvidas

Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/. Acesso em: 24 de fev de 2017.

Assim chegamos ao final da nossa aula! Essa parte de animação é tão interessante e complexa que nos tomou duas aulas inteiras e ainda ficamos sem tempo para cobrir todos os aspectos envolvidos. O que podemos garantir, no entanto, é realmente termos nos capacitado a utilizar diversos tipos de animação, as quais são o bastante para desenvolvermos os nossos jogos ou, no mínimo, servem de base para podermos aprender novos estilos não abordados aqui. Espero que todos tenham aprendido tanto quanto eu aprendi desenvolvendo esta aula. Nos veremos em breve, na nossa próxima aula! Até! o/

Leitura Complementar

Manual oficial do Unity sobre animação

<https://docs.unity3d.com/Manual/AnimationSection.html>

Blend Trees

<https://docs.unity3d.com/Manual/class-BlendTree.html>

Resumo

Nesta aula, conhecemos as transições entre animações, a partir de um Animator Controller que possui múltiplos estados no Unity. Vimos como os parâmetros podem ser utilizados para isso funcionar e como utilizamos variáveis para atualizar esses valores durante a execução do nosso jogo.

Em seguida, conhecemos a Blend Tree, componente capaz de intercalar animações e utilizar um valor de referência para alternar entre elas. Isso se mostrou útil, por exemplo, para fazermos uma única animação capaz de lidar com o pulo e a queda do personagem, sem a necessidade de reconstrução ou duplicação.

Aos que precisarem, o projeto que desenvolvemos hoje pode ser encontrado aqui! Aproveitem bem e até a próxima!

Autoavaliação

1. Para que servem as transições em animações no Unity?
2. Para que são utilizados os parâmetros em um Animator Controller no Unity?
3. Como funciona uma máquina de estados de uma animação?
4. O que é uma Blend Tree? Como ela pode ajudar em animações 2D?

Referências

Documentação oficial do Unity. Disponível em: <https://docs.unity3d.com/Manual/index.html>.

Tutoriais oficiais do Unity. Disponível em: <https://unity3d.com/pt/learn/tutorials>.

RABIN, Steve. **Introdução ao Desenvolvimento de Games**, Vol 2. CENGAGE.

RABIN, Steve. **Introdução ao Desenvolvimento de Games**, Vol 3. CENGAGE.