Programação Visual em Desenvolvimento e Design de Jogos Digitais

Estudo e Análise da Programação Visual Como Método Alternativo para Lógica em Jogos Digitais

Versão final após defesa

Lucas de Azevedo Lopes Ferreira

Dissertação para obtenção do Grau de Mestre em
Design e Desenvolvimento de Jogos Digitais
(2º ciclo de estudos)

Orientador: Prof. Bruno Miguel Correia da Silva

Março de 2023

Declaração de Integridade

Eu, Lucas de Azevedo Lopes Ferreira, que abaixo assino, estudante com o número de inscrição M11239 de Design e Desenvolvimento de Jogos Digitais da Faculdade de Artes e Letras, declaro ter desenvolvido o presente trabalho e elaborado o presente texto em total consonância com o **Código de Integridades da Universidade da Beira Interior**.

Mais concretamente afirmo não ter incorrido em qualquer das variedades de Fraude Académica, e que aqui declaro conhecer, que em particular atendi à exigida referenciação de frases, extratos, imagens e outras formas de trabalho intelectual, e assumindo assim na íntegra as responsabilidades da autoria.

Universidade da Beira Interior, Covilhã 24/03/2022

Agradecimentos

Um agradecimento enorme à minha família, especialmente aos meus pais, aos meus amigos e a todos que me apoiaram durante esta jornada.
Este trabalho foi realizado e enquadrado no grupo de investigação Network Applications and Services do Instituto de Telecomunicações, em particular no laboratório de investigação SINS.

Resumo

Esta dissertação surge da necessidade de procurar um modo de substituir as várias formas de programação tradicionais para um método alternativo, direcionado especialmente a pessoas que recentemente ingressaram na área de desenvolvimento de jogos e/ou para artistas que, apenas desejam criar programas e jogos usando objetos de 2D e 3D para dar vida e controlo às suas criações.

Neste sentido, esta tese começa por apresentar uma textualização sobre o tema, seguindo por uma parte prática, onde é feita a comparação entre duas linguagens de programação visual. Nesta segunda parte, vão ser apresentados os vários passos que tomei para desenvolver um simples jogo digital completamente funcional em dois programas diferentes, mas mantendo a mesma lógica de código (ou o mais próximo possível) e as mesmas características técnicas.

O principal objetivo desta tese é apresentar um estudo detalhado sobre a programação visual, como uma introdução a qualquer um que tenha interesse nesta área para desenvolver um jogo simples, sem a necessidade de aprender e/ou usar programação tradicional.

Com base nas informações aqui partilhadas, o leitor poderá expandir os seus conhecimentos e ser capaz de construir vários tipos de sistemas que lhe podem ser úteis durante a sua jornada profissional.

Palavras-chave

Programação, programação visual, código, artistas, jogo digital.

Abstract

This thesis arises from the need to search for a way to replace the several forms of traditional programing for an alternative method, aimed especially at people who have recently entered the field of game development and/or artists who just wish to create programs and games, using 2D and 3D objects, to bring life and control to their creations.

In this sense, this thesis begins by presenting a textualization on the subject, followed by a practical part, where a comparison is made between two visual programming languages. In this second part, I will present the many steps I took to develop a simple fully functional videogame in two different programs, but keeping the same code logic (or as close as possible) and the same technical characteristics.

The main goal of this document is to present a detailed study on visual programing, as an introduction to anyone who is interested in this area to develop a simple game without the need to learn and/or use tradicional programming.

Based on the information shared here, the reader will be able to expand their knowledge and build various types of systems that can be useful during their professional journey.

Keywords

Programming, visual programming, code, artists, videogame.

Índice

Lista de Figuras xv
1. Introdução 1
1.1. Objetivos 3
1.2. Organizaçao da tese 3
2. Revisão da literatura 5
2.1 Programação por código 5
2.2. Programação Visual 8
2.3. Diferenças entre código e Programação Visual 10
2.3.1. Vantagens e desvantagens de Programação visual 10
2.3.2. Vantagens e desvantagens de Código tradicional 10
3. Guia para a Programação Visual 12
3.1. História da programação visual 12
3.1.1. Sketchfab (1963) 12
3.1.2. *GRAIL* (1969) 13
3.1.3. *Pygmalion* (1975) 14
3.3.4. *Prograph* (1983) 15
3.3.5. Hypercard (1987) 16
3.2. Linguagens atuais de programação visual 17
3.2.1. Programação baseada em blocos 17
3.2.2. *Scratch* 18
3.2.3. *MIT app inventor* 19
3.2.4. *Google blockly* 20
3.3. Programação baseada em fluxo 21
3.3.1 *RoboFlow* 23
3.3.2. *Unreal Engine – Blueprints* 24
3.3.3. *Unity – Bolt* 25
4. Abstração de lógica em Programação 27
4.1. Simplificação de código 27
4.2. Abstração em Programação Visual 28
5. Implementação 32
5.1. Introdução do projeto e tecnologias usadas 32
5.2. Objetivo do Jogo 34
5.3. Mapa do Jogo 34
5.4. Fluxo de trabalho de concepção de jogos digitais 35
6. Desenvolvimento do jogo em *Blueprints* (*Unreal Engine*) 37
6.1. Criação de projeto e nível 37
6.2. Preparação de *sprites* e do background 38
6.3. Criação do mapa 38

6.4. Preparação da personagem e das suas animações 39
6.5. Preparação da personagem 40
6.6. Controlo de personagem 42
6.7. Movimento 43
6.8. *Nodes* de ativação da animação de movimento 44
6.9. *Nodes* de rotação de personagem 45
6.10. *Nodes* de Salto 48
6.11. Ativação de dano de espigões e condição de derrota 49
6.11.1. Espigões 49
6.11.2. Elemento de UI – *Game Over* 50
6.11.3. Condição de derrota 51
6.12. Ativação de coleção de pontos e condição de vitória 52
6.12.1. Cogumelos 52
6.12.2. Elemento UI – Contador de cogumelos obtidos 53
6.12.3. Elemento UI – Painel de Vitória 55
6.12.4. Condição de vitória 56
6.13. Elemento de UI – Menu principal 57
7. Desenvolvimento do jogo no programa *Bolt* (*Unity*) 60
7.1. Criação de projeto e nível 60
7.2. Preparação do mapa 60
7.3. Colocaçao das peças do conjunto de blocos 61
7.4. Background 63
7.5. Preparação da personagem e das suas animações 64
7.6. Definição da ordem das animações (Animator) 65
7.7. Programação da personagem 67
7.7.1. Movimento 67
7.7.2. *Nodes* de rotação de personagem 69
7.7.3. *Nodes* de ativação da animação de movimento 70
7.7.4. *Nodes* de ativação da animação de *idle* 71
7.7.5. *Nodes* de Salto 71
7.7.6. Deteção de chão (*Raycast*) 72
7.7.7. Ativação da animação de salto 73
7.7.8. *Nodes* de ativação da animação de queda 74
7.8. Elemento de UI – Menu principal 74
7.9. Ativação de dano de espigões e condição de derrota 76
7.9.1. Espigões 76
7.9.2. Elemento de UI – *Game Over* 77
7.9.3. Condição de derrota 78
7.10. Ativação de coleção de pontos e condição de vitória 79

7.10.1. Elemento UI – Painel de Vitória 79
7.10.2. Cogumelos 80
7.10.3. Condição de vitória 82
7.10.4. Elemento UI – Contador de cogumelos obtidos 82
8. *Unity* vs *Unreal Engine* 84
8.1. Preço e público alvo 84
8.2. Diferenças de código e preparação da personagem 85
9. Conclusões finais 87
10. Bibliografia 88

Lista de Figuras

Figura 1 – Programa *Sketchpad* no computador "*The Lincoln TX-2*" (1963). 13
Figura 2 – Programa *GRAIL* (1969).....14
Figura 3 – Demonstração de funcionamento do programa *Pygmalion* (1975).....15
Figura 4 – Operação no programa *Prograph* (1983).....16
Figura 5 – Demonstração do *layout* e organização de código do programa MIT. 20
Figura 6 – Demonstração da mesma lógica de código no Google Blockly e em *JavaScript*. ..20
Figura 7 – Diagrama representando o conceito de FBP. 21
Figura 8 – Representação resumida da funcionalidade de um programa de codificação visual baseada em fluxo.....22
Figura 9 – Ligação entre elementes de *inputs* e *outputs*. O elemento de *input* de cor roxa é um valor *boolean* que decide se o fluxo vindo do elemento *input* azul vai se dirigir para o valor *output* vermelho ou verde. 23
Figura 10 – Representação do *layout* do programa *Roboflow*. 24
Figura 11 – Representação de conecções entre *nodes* no programa *Unreal Engine*. 25
Figura 12 - Representação de conecções entre *nodes* no programa *Unity*. 26
Figura 13 – Código do programa "*Hello, world*" desenvolvido através da linguagem de programação *Assembly*. 27
Figura 14 - Representação do código para a função de *for loop*, com o objetivo de pôr em ação um comportamento com base numa condição. 29
Figura 15 – Coluna do lado esquerdo do programa *Scratch*, apresentando a lista de todos os blocos de controlo. Os elementos de destacados pelo retângulo de cor verde são os blocos de condição. 30
Figura 16 – Procedimento do código para executar a função de loop descrito pelos passos anteriores. 30
Figura 17 – Animação *idle* da personagem. 33
Figura 18 – Animação de movimento da personagem. 33
Figura 19 – Animação de salto da personagem. 33
Figura 20 – Imagem de fundo do futuro jogo. 33
Figura 21 – Blocos utilizados para a construção de nível. 33
Figura 22 – Elemento de obtenção de pontos. 34
Figura 23 – Elemento com o objetivo de danificar a personagem em caso de contacto físico. 34
Figura 24 – Esboço do mapa pretendido. 35
Figura 25 – Menu inicial do programa *Unreal Engine*. 37
Figura 26 – Colocação da imagem de fundo no ambiente. 38
Figura 27 – Janela dedicada ao "Tile Set", para preparar cada bloco para depois ser colocado no nível. 39
Figura 28 – Janela do elemento de flipbook para editar a respetiva animação de personagem. 40
Figura 29 – Representação da lista correta de componentes que formam a personagem. 42
Figura 30 – Representação da colocação correta entre a personagem, o "*spring arm*" e a câmara. 42
Figura 31 – Menu de ações, onde se situa a lista de todos os *nodes* utilizáveis. 43
Figura 32 – *Nodes* de movimento da personagem. 44
Figura 33 – Ligação entre os vários *nodes* que acionam a animação de mobilidade quando é detetada algum movimento lateral da personagem. 45
Figura 34 – Comunicação ao sistema para que a animação seja posta em ação em cada frame. 45
Figura 35 – Alteração feita na ligação dos *nodes* de movimento de personagem, para que o sistema saiba de que lado é que a personagem está virada quando esta se movimenta para a esquerda ou para a direita, usando uma variável costumizada do tipo *boolean*. 46

Figura 36 – Representação dos *nodes* dedicados à rotação da personagem, tomando referência da variável criada anteriormente. 47
Figura 37 – Adição à referência da função de rotação ao "*Event Tick*", para que seja posta em ação em todos os frames. 47
Figura 38 – Adição da animação de salto da personagem aos *nodes* de animação já existentes. 49
Figura 39 – Representação visual dos espigões e a sua caixa de colisão. 50
Figura 40 – Aplicação de dano ao elemento dos espigões em caso de colisão com outro elemento 50
Figura 41 – Representação do painel de derrota e a lista dos seus componentes presentes na coluna do lado esquerdo. 51
Figura 42 – *Nodes* dedicados à função de cada botão. 51
Figura 43 – *Nodes* de condição de derrota, indicando todos os comportamentos no caso de houver algum dano cometido à personagem. 52
Figura 44 – Representação visual dos cogumelos e a sua caixa de colisão. 53
Figura 45 – Representação de *nodes* da obtenção de pontos em caso de colisão entre os cogumelos e a personagem. 53
Figura 46 – Representação do número de cogumelos obtidos e a lista dos seus componentes presentes na coluna do lado esquerdo. 55
Figura 47 – *Nodes* de comunicação ao sistema para que o elemento anterior relativo aos cogumelos obtidos seja criada e representada no ecrã, no momento que o jogo inicia. 55
Figura 48 – Representação do painel de vitória e a lista dos seus componentes presentes na coluna do lado esquerdo. 56
Figura 49 – No caso de colisão entre os cogumelos e a personagem, é retirado um valor ao número total de cogumelos presentes no mapa (Total Cogumelos). 56
Figura 50 – Ligações entre os *nodes* dedicados à condição de vitória. Indicando os comportamentos em caso do número total de cogumelos presentes no mapa for igual a zero. 57
Figura 51 – Representação do menu incial e a lista dos seus componentes presentes na coluna do lado esquerdo. 58
Figura 52 – *Nodes* dedicados à função de cada botão presente no menu principal. 59
Figura 53 – Menu inicial do programa *Unity*. 60
Figura 54 – Menu de preparação da grelha que divide cada bloco, dentro da janela do "*Tile set*". 61
Figura 55 – Colocação de um frame retangular, através do menu que surge com o botão direito do rato dentro da área da coluna do lado esquerdo. 62
Figura 56 – Janela do "*Tile Palette*", onde se constrói o nível através da coleção de blocos. 62
Figura 57 – Blocos de uma parte do mapa e as suas caixas de colisão representadas pelas linhas de cor verde. 63
Figura 58 – Menu da lista de componentes, com o seu motor de busca a ser utilizado para encontrar o elemento de "*Box Collider 2D*". 63
Figura 59 – Representação da posição da imagem de fundo em relação ao nível. 64
Figura 60 – *Timeline* dentro do separador "*Animation*", onde é possivel editar a cronometragem de cada frame das animações da personagem. 65
Figura 61 – Representação do ciclo de animações da personagem. O elemento "*Entry*" vai por em ação a primeira animação que queremos que seja acionada no início do jogo, neste caso a animação idle (*New Animation*). A partir deste, só pode ser ativada a animação de movimento (*run*). E, em qualquer estado da personagem (*Any State*), pode ser posta em ação a animação de salto (*jump / Fall*), que por sua vez vai retornar ao estado *idle*. 66
Figura 62 – Primeira parte do código de movimento. São *nodes* de ligação entre o *input* do jogador com a variável de "speed", resultando no movimento da personagem. 68
Figura 63 – *Nodes* de rotação da personagem. 70
Figura 64 – *Nodes* de ativação de animação. O "*branch*" pertence ao código representado na figura 63. 71

Figura 65 – Indicação ao sistema que a animação idle deve ser acionada logo no inicio do jogo. 71
Figura 66 – *Nodes* de ativação da porpulção da personagem no eixo do Y, resultando no salto. 72
Figura 67 – Ligação de *nodes* que consiste na criação do *Raycast*. 73
Figura 68 – *Nodes* de ativação da animação de salto, usando a cópia do mesmo código anterior para definir a situação em que se pode acionar a respetiva animação. 74
Figura 69 – *Nodes* de ativação da animação de queda, usando como referência a velocidade no eixo de Y. 74
Figura 70 – Colocação de um painel de UI através do menu que surge com o botão direito do rato dentro da área da coluna do lado esquerdo. 75
Figura 71 - – Representação do menu incial e a lista dos seus componentes presentes na coluna do lado esquerdo. 76
Figura 72 – *Nodes* dedicados à função do botão "PLAY". 76
Figura 73 – *Nodes* dedicados à função do botão "EXIT". 76
Figura 74 – Indicação ao sistema que no momento de colisão entre os espigões e a personagem, irá por em ação um comportamento com o nome de "*Death*". 77
Figura 75 – Representação do painel de derrota. 77
Figura 76 - *Nodes* dedicados à função do botão "*Back to menu*". 78
Figura 77 – Elemento referente do painel de derrota como uma variável. 78
Figura 78 – Descrição do comportamento "*Death*" criado anteriormente. 79
Figura 79 – Adição do *node* "*Set time Scale*" aos *nodes* de função do botão "*Back to menu*". 79
Figura 80 – Representaçao do painel de vitória. 80
Figura 81 – Lista de variáveis, presente na coluna do lado direito. A variável "*CoinTotal*" representa o número total de cogumelos obtidos pela personagem, e a variável "*CoinWin*" é o número de cogumelos que é preciso alcansar para que o jogador ganhe o jogo. 81
Figura 82 – *Nodes* de obtenção de pontos a partir da colisão entre a personagem e os cogumelos. Durante esse acontecimento, o valor do elemento do cogumelo (*CoinValue*) é adicionada ao número total de cogumelos obtidos pela personagem (*CoinTotal*). 81
Figura 83 – *Nodes* de condição de vitória. No momento que o número total de cogumelos obtidos for igual a três, o jogador ganha o jogo. 82

1. Introdução

Nos tempos de hoje, a grande maioria das pessoas em todo o mundo têm que admitir que todos nós vivemos num mundo de programação. Não é preciso emergir numa intensa ponderação para chegar á conclusão deste facto.

Se nós gostamos de jogar jogos digitais, se usufruímos das utilidades de programas como o Word, se comunicamos por email, se utilizarmos a *internet* para navegar na *web* através do nosso *smartphone*, que por sua vez também se enquadra neste campo; então sim, podemos afirmar que estas tecnologias se tornaram indispensáveis para o nosso bem-estar.

Segundo um estudo realizado entre 2016 e 2021, o número atual de utilizadores de telemóveis em 2023 é de aproximadamente 6,84 biliões. Colocando esse dado em perspetiva, este número representa cerca de 85% da população mundial. A mesma fonte também concluí que o número global de residências com um computador pessoal é de 971.98 milhões (Statista 2022).

Retirando este tipo de aparelhos, ainda assim damos por nós rodeados na maior parte das vezes por sistemas contruídos por certas linguagens programação. Tomando o exemplo da linguagem "C", que é bastante utilizada para programar os sistemas de despertadores digitais, dos micro-ondas, televisões, da maioria de máquinas de vendas, caixas registadoras, cartões de créditos, etc. (Munoz).

Foi no final da década de quarenta que surgiram as primeiras linguagens. E, a partir daí, o nosso mundo mudou para sempre. Os tais sistemas que hoje tanto achamos indispensáveis, tiveram a sua origem nesse mesmo ponto da história humana, começando com um antigo conceito de utilização de cartões perfurados, idealizado para a indústria textil, no ano de 1800, até os vários sistemas avançados que dão corpo a aplicações complexas, tais como o *Java*, C++ ou o *Visual Basic*, umas das linguagens de alto nível que ainda se mantém no topo da tabela das linguagens de programação.

Uma linguagem de programação de alto nível é uma linguagem de programação com um grau forte de abstração dos detalhes do computador, capaz de espojar os detalhes subjacentes sem perder a sua complexidade (“WayBackMachine”). Deste modo, o utilizador consegue programar um sistema sem a necessidade de se preocupar com as partes mais técnicas, permitindo colocar toda a sua atenção ao seu objetivo principal. Mas, ainda assim, sempre foi um desafio para uma pessoa que não se encontra nesta área, aprender a escrever linhas de código, mesmo utilizando uma linguagem com um grau elevado de abstração.

Falando de um modo mais pessoal, a área de programação sempre foi algo que quis evitar, pois sempre fui intimidado pelas extensas linhas de código que dão corpo às grandes tecnologias. Fui e sempre serei um artista dedicado as artes visuais e ao longo do tempo fui

demonstrando interesse em dar vida as minhas criações. Através da grande paixão que sempre tive por jogos digitais, também surgiu uma vontade de talvez um dia conseguir construir um jogo de raiz, mas é um facto de que a programação não é própriamente atrativa para todos os públicos. Chegamos a esta conclusão pelo simples facto de que nem toda a gente possui certos requisitos importantes para esta área. Não são necessariamente obrigatórios, mas que são muito relevantes para a aprendizagem e domínio de programação por código.

Primeiramente é necessário ter a habilidade de visionar o projeto e transformar a visão em métodos de código para atingir o fim desejado. Para esse objetivo é preciso que o desenvolvedor possua a habilidade de resolução de problemas usando o pensamento lógico e algorítmico de forma independente. É recomendável que a pessoa tenha um verdadeiro interesse em entender o funcionamento do nosso mundo digital, sem ficar facilmente frustrado ao se deparar com as barreiras do ofício. As respostas para os eventuais problemas nesta área, devem ser encontradas pela própria vontade do desenvolvedor, pois são aqueles que entendem em decompor os problemas e procurar soluções em todas as documentações de código, que realmente têm mais sucesso na área (Grant 2022).

Felizmente, este obstáculo que durante muito tempo se manteve firme, tornou-se muito mais quebrável pela descoberta das linguagens de programação visual. Um tipo alternativo de linguagem que usa um método de abstração de uma forma inovadora e, o mais importante, tem sido capaz de motivar todo o tipo de pessoas a aprender a criar lógica de uma forma apelativa, simples e divertida.

Estas linguagens avançaram muito, não só na sua complexidade como também no nível de acessibilidade, para que seja mais fácil para qualquer um aprender a construir um sistema funcional utilizando elementos visuais.

A sua funcionalidade depende do programa a ser utilizado, mas todos seguem o mesmo conceito: Facilitar a comunicação entre humano e máquina, com formas básicas, para que o utilizador consiga focar mais facilmente na lógica do código, sem se preocupar na necessidade de memorizar os vários tipos de termos e expressões que cada linha de código tradicional precisa para conceber um sistema funcional.

O conceito de programação visual não é de todo recente, curiosamente começou a ser desenvolvido pouco tempo depois do surgimento das linguagens de programação mais complexas. Em 1958, por exemplo, surgiu o "*Sketchfab"*, um dos primeiros sistemas de programação visual, executável no computador "*The Lincon TX-2*", no mesmo ano do aparecimento da linguagem de programação por código com o nome de "*ALGOL"*, que marcou o início do desenvolvimento de C++ ("The History of Computing Podcast" 2020).

A partir desse ano, foram feitas mais experiências dentro do campo de programação visual, mas, devido ao facto dos computadores da altura não possuirem uma capacidade gráfica para suportar estas inovações, estas linguagens nunca chegaram a um grande potêncial.

Nos dias de hoje, são ferramentas muito poderosas que permitem qualquer um contruir sistemas com um certo nível de complexidade. Muitos dos seus utilizadores têm preferencia ao uso destas linguagens em contraste do código tradicional, mas por outro lado, tem os seus limites. A programação visual era muito mais promissora nos seus estágios iniciais, porque as aplicações e todo o tipo de sistemas digitais tornaram se demasiado complexos para serem representados visualmente.

1.1. Objetivos

O estudo presente neste documento propõe apresentar uma introdução sobre a programação visual e os alguns dos seus métodos como uma forma alternativa de criar jogos digitais sem entrar em grande detalhe técnico, permitindo uma compreensão clara para que cada leitor tenha conhecimento que não é indispensável dominar os fundamentos de linguagens de programação tradicional para o desenvolvimento de sistemas gamificados completamente funcionais e profissionais.

Para esse efeito, realizei um estudo do estado de arte, uma revisão das linguagens de programação visual e de código, seguindo pelas suas respetivas análises. De seguida irei apresentar um projeto com o objetivo de exibir os vários passos que tomei para a elaboração de um jogo digital em duas linguagens de programação visual diferentes. Por fim, irá ser feita uma análise de resultados e uma comparação direta entre eles, concluindo com defecho sobre o tópico geral.

1.2. Organizaçao da tese

Esta dissertação está estruturada da seguinte forma:

- No capítulo 1 é efetuada uma introdução ao tema da presente dissertação, com o enquadramento do tema e os objetivos propostos;
- No capítulo 2 é feita uma revisão da literatura com a apresentação da base teória relativa à programação por código e o seu progresso ao longo dos anos, seguido a uma explicação sobre as funcionalidades de programação visual e as suas diferênças em relação à programação tradicional;

- O capítulo 3 é dedicado a um guia sobre programação visual, mencionando a história dessa metodologia no desenvolvimento de sistemas e os dois tipos diferentes desses programas;
- No capítulo 4 será apresentada uma explicação sobre a abstração de código e como a linguagem Scratch efetua esse método para ocultar elementos de código não desejados na elaboração de comportamentos em comparação à programação por código;
- No capítulo 5 surge uma explicação sobre a elaboração do projeto desenvolvido que envolve em desenvolver um jogo digital usando duas linguagens de programação visual baseadas em fluxo;
- No capítulo 6 serão listados os passos tomados para construir um jogo através do programa *Unreal Engine*;
- No capítulo 7 serão apresentados os passos tomados para construir o mesmo jogo usando o programa *Unity*, com as mesmas especificações técnicas, ou o mais próximo possível;
- O capítulo 8 dedica-se a uma comparação direta entre os dois programas;
- No capítulo 9 serão apresentadas as conlcusões finais.

2. Revisão da literatura

Neste capítulo serão esclarecidos as bases fundamentais de programação por código e das linguagens de programação visual, bem como uma introdução dos vários sistemas entre cada metodologia de programação referida que foram surgindo ao longo dos anos, desde os primeiros exemplos, até aos mais conhecidos atualmente.

Também será feita uma comparação direta entre eles, incluindo as vantagens e desvantagens de cada um.

Para desenvolver a base teórica, foi feita uma investigação de publicações legítimas sobre os seguintes tópicos: conceitos de programação visual, as várias linguagens de programação visual e os seus diferentes métodos, história da programação visual, surgimento da programação por código, o aparecimento das linguagens de programação tradicional ao longo dos anos e as respetivas funcionalidades, elucidação de definição de código em artigos científicos e sites educativos.

2.1 Programação por código

Todas as linguagens de computador baseadas em texto focam-se inteiramente na execução, ou seja, é tudo uma questão de fazer com que o computador siga todos os passos específicos para gerar uma experiência para os futuros utilizadores. Por outras palavras, a responsabilidade do programador e transformar a demanda humana em procedimentos que são compatíveis com as capacidades do computador.

Esta representação textual de código baseia-se no uso de palavras, números, vários tipos de pontuação e sinais para gerar algum tipo de lógica, da mesma maneira como representamos a maioria das linguagens existentes no nosso planeta.

Programação é uma área bastante desafiante. Embora tenham surgido programas de código cada vez mais avançadas e intuitivas, ainda é necessária muita experiência para desenvolver um programa minimamente complexo.

As primeiras linguagens de programação remontam aos anos cinquenta, mas os seus antecedentes iniciais surgiram ainda antes do aparecimento dos primeiros computadores eletrónicos.

Tudo começou com o uso de objetos com o nome de cartões perfurados usados na intustria textil, no início de 1800. Joseph Marie Jacquard, um inventor e tecelão francês, originou um aparelho mecânico inteiramente automatizado, uma versão avançada do *tear* (aparelho para fins de tecelagem), que tinha como objetivo tecer vários tipos de pano com

padrões complexos sem a colaboração de algum esforço humano, através destes cartões perfurados.

Simplificadamente falando, cada furo no cartão comunicava à respetiva máquina em que local do tecido é que a agulha deve passar. Deste modo, cada cartão controlava um movimento da lançadeira (Barfield, 2021).

Embora esta invenção não tinha nenhum fim para a área de matemática, este foi o primeiro sistema binário e o nascimento da programação que conhecemos hoje.

Em 1842, uma matemática e escritora inglesa conhecida como Ada Lovelace, traduziu uma obra de Charles Babbage de 1822 que teorizou o uso de dos cartões perfurados para fazer cálculos avançados. Lovelace apercebeu-se do potencial desta obra e adicionou as suas próprias notas, dizendo que os valores numéricos produzidos por esta máquina seriam capazes de também representar simbolos, notas músicais e todo o tipo de elementos. Estas anotações, publicados em 1843, originaram o primeiro programa de computador e Ada Lovelace tornou-se a primeira programadora de toda a história.

Os anos passaram e foram surgindo cada vez mais inovações usando os mesmos principios dos cartões perfurados. Em 1946, foi elaborada a primeira real linguagem de programação com o nome de *Plankalkül*, criada pelo cientista alemão Konrad Zuse no desenvolvimento do primeiro computador digital completamento automático, o Z3, cujo código dos programas era armazenado em longas fitas perfuradas.

Mais tarde, foi construído em 1951 o primeiro computador comercial, projetado por J. Presper Eckert e John Mauchly, chamado *Univac* (UNIversal Automatic Computer). Ao contrário do seu antecedente, o *Eniac (Electronic Numerical Integrator and Computer*) que funcionava através de cartões perfurados, o *Univac* tomou um passo em frente, usando fitas magnéticas para o armazenamento de dados.

Em 1958, a ACM (*Association for Computing Machinery*) e a *German Society of Applied Mathematics* flutuavam em torno da ideia para uma linguagem universal de programação de computadores. Desde o dia 27 de maio até 2 de junho de 1958 desenvolveram a linguagem *"ALGOL"* que introduziu muitas características inovadoras, tal como o uso de blocos de código, o conceito de usar um par de palavras ou símbolos que começariam e terminariam uma estrofe de código, e funções com escopo aninhado. Também introduziu operadores booleanos e definiram como lidar com caracteres especiais. Assim, conseguiram construir um projeto que deu resultado ao Formalismo de *Backus-Naur*, a primeira padronização de código, permitindo que os algoritmos pudessem ser publicados, compartilhados e facilmente entendidos por outros desenvolvedores que quisessem acompanhar a lógica. Desta forma, *ALGOL* foi o ponto de partida para o desnvolvimento das

linguagens de programação mais relevantes, como o C++ (“The History of Computing Podcast” 2020).

Um ano depois, surgiu a linguagem de programação “*COBOL*” (Common Business Oriented Language), criado pela cientista de computação americana Grace Hopper. Esta linguagem imperativa e orientada a objetos foi desenhada para o sistema de *UNIVAC* e consiste em tomar referência de programas de computadores escritos em linguagens de alto nível e convertê-los noutro tipo de código para que um computador diferente possa entender. Resumidamente, *COBOL*, recebe dados, calcula-os e envia estes dados posteriormente.

É também um sistema capaz de converter inglês para código de computador e a linguagem por trás de muitos processadores de cartão de crédito, caixas eletrónicas, telefones, sinais hospitalares e sistemas de sinais de trânsito (“What is COBOL?” 2022).

Foi em 1972 que foi desenvolvido o que muitos consideram a primeira linguagem de alto nível. Criado por Dennis Ritchie no *Bell Labs* (conhecido por Nokia), “C” é a linguagem de programação que, naquela altura, mais se aproximou à linguagem humana, distanciando-se do código de computador. C foi criado para que um sistema operacional chamado Unix pudesse ser usado em muitos tipos de computadores diferentes.

Esta linguagem tem como ponto forte a sua eficiência, e por não possuir algumas características úteis que se encontram noutras linguagens, como classes e funções aninhadas, algo que pode ser visto como ponto negativo, também permitiu que os sistemas de C fossem escritos mais rapidamente e que o programador permaneça sempre em controle do que o programa está a fazer. C é a linguagem de preferência para o desenvolvimento de sistemas de softwares de base e chegou a influenciar o desenvolvimento de outras linguagens, incluindo *Ruby*, C#, *Java* e *Python* (“C Family Interview” 2000).

No ano de 1979, Bjarne Stroustrup idealizou a teoria de “C com classes”, como uma forma de desenvolver uma versão ainda mais eficiente da linguagem C, juntando a utilização de elementos como classes, funções virtuais e templates. Esta inovação, que começou como uma experiência durante a sua dissertação de universidade, evoluiu até ao ponto de C++ ser colocado no top 10 das linguagens de programação desde 1986. É usado no MS Office, Adobe Photoshop, em muitos motores de jogo atuais e outros softwares de grande performance.

Guido Van Rossum, inspirado pela comédia britânica “*Monthy Python*”, criou a linguagem de programação “*Python*” em 1991. A sua versão mais prematura já possuia muitas características avançadas, tais como classes com herança, funções e manipulação de exceções.Rossum nao desenvolveu e evoluiu todos os componentes do *Python* sozinho. A velocidade com que esta linguagem se espalhou pelo mundo é o resultado do trabalho de milhares programadores, muitas vezes anónimos, avaliadores, utilizadores e entusiastas

(Lestal 2020). Python é, até hoje, uma das linguagens de programação mais populares do mundo, usada por empresas como Google, Yahoo e Spotify.

No mesmo ano, 1991, a Microsoft criou a linguagem "*Visual Basic*", para o sistema operativo windows. É uma linguagem de programação que fornece uma interface gráfica de utilizador (GUI, *graphical user interface*). Esta linguagem permite aos programadores modificar código ao arrastar e soltar objetos e definindo o seu comportamento e aparência.

Visual Basic foi desenvolvido com o objetivo de ser uma ferramenta fácil de aprender e intuitiva para escrever código de uma forma mais rápida. Como resultado, este sistema é muitas vezes nomeado como um sistema de desenvolvimento rápido de aplicações (RAD, rapid application development) e é utilizado para construir protótipos de aplicações, que mais tarde serão escritos numa linguagem mais complexa e eficiente.

Desta forma, este sistema foi projetado para ser uma linguagem de programação completa, que continha recursos tais como processamento de strings e computação e permite aos programadores desenvolver uma interface de utilizador fácil de usar, mesmo para os desenvolvedores de pouca experiência ("TechTarget Contributor" 2019).

Em 1995 foi elaborada a linguagem com o nome de "*Java*", por James Gosling. Foi originalmente projetada para um projeto de televisão interativa, mas era demasiado avançada para a indústria de televisão digital a cabo na época. Gosling criou este projeto com sintaxe no estilo C/C++, que era muito familiar para os programadores de sistemas e aplicações, e evoluiu para uma linguagem de alto nível, entre as principais e mais populares do mundo. Java pode ser encontrado em todos os lugares, desde computadores e smartphones para parquímetros.

No ano 2000, surgiu o C#. Foi desenvolvido pela Microsoft com a esperança de combinar a capacidade de computação da linguagem C++ com a simplicidade do *Visual Basic*, adicionando também algumas semelhanças com o *Java*. C# é usada em quase todos os produtos da Microsoft e é vista como a pincipal linguagem no desenvolvimento de aplicativos de desktop (Lestal 2020).

2.2. Programação Visual

Código pode ser demasiado difícil para aprender, especialmente para alguém que ingressou recentemente nesta área de game design. Programação visual é uma das diversas formas de contornar este problema sem desistir do desejo de criar de raiz um programa digital. É uma linguagem de programação que permite-nos descrever um processo através de ilustração. Tal como uma linguagem de programação típica baseada em texto faz como que um programador pense como um computador, as linguagens de programação visual

permitem ao programador descrever um processo de maneira que faça sentido a um ser humano. Para esse efeito, disponibilizam elementos visuais para serem manipulados pelo utilizador, tais como vários tipos de símbolos e expressões visuais, cada uma com uma lógica implementada (“Kissflow” 2022).

O uso mais comum de programação visual é por pessoas que querem aprender a escrever código de uma maneira mais eficiente e que se especializam noutras áreas, tais como artes visuais e design. A natureza visual desta linguagem acaba por facilitar a entrada de quem dispõem um interesse em programação, ao disponibilizar ferramentas intuitivas para criar um protótipo facilmente e rapidamente.

Grande parte destas linguagens baseiam-se no uso de "blocos e setas", onde os blocos representam-se como entidades que contêm código pré-inserido e que se conectam com outros blocos com setas, linhas ou arcos. Em vez de perder imenso tempo a escrever código à mão, que envolve um conhecimento considerável de linhas de logica e comandos, com o *visual scripting* podemos simplesmente interligar os vários blocos específicos de maneira a obter um projeto complexo e funcional. Esses blocos são chamados "*Nodes*".

Resumidamente, o principal objetivo das linguagens de programação visual é tornar a área de programação mais acessível a novatos e ajudar os programadores em 3 níveis (Repenning 2017):

Sintaxe: Utilizam ícones/blocos, formulários e diagramas com o objetivo de reduzir ou mesmo eliminar potenciais erros sintáticos durante o desenvolvimento de um programa

Semântica: Este tipo de programação tem a capacidade de fornecer mecanismos para divulgar o significado de conceitos base de programação.

Pragmática: As linguagens de programação visual apoiam o estudo dos significados dos programas em situações particulares. Esse nível de suporte permite que os utilizadores coloquem artefactos criados por estes sistemas num determinado estado para explorar como o programa reagirá a esse estado.

2.3. Diferenças entre código e Programação Visual

2.3.1. Vantagens e desvantagens de Programação visual

Vantagens:

- Desenvolvimento de sistemas de forma mais intuitiva;
- Manutenção de código rápida e facilitada;
- Fácil de aprender;
- Ideal para desenvolver programas em conjunto com outras pessoas, possibilitando um eficiente trabalho de equipa;
- Oculta muito detalhes com as quais o desenvolvedor não precisa de se preocupar;
- Possui uma demonstração de código muito mais claro;
- Na maior parte das vezes não possibilita a existência de erros de código;

Desvantagens:

- Falta de flexibilidade;
- O desenvolvedor não vê o que realmente está a acontecer dentro do código;
- É possível aprender a resolver um problema, mas o utilizador não poderá fazê-lo fora do programa específico;
- Não é possível fazer atualizações de desempenho;
- É dependente no motor que está a ser usado;
- Sempre que o programador decide mudar para outro motor, ele terá que aprender como ele funciona a partir do início;
- Se houver um bug no motor, não é possível corrigi-lo. Tem que se esperar por uma atualização do seu sistema;
- A maioria dos programas de programação visual não são gratuitos;
- Possibilidades limitadas.

2.3.2. Vantagens e desvantagens de Código tradicional

Vantagens:

- Permite que o utilizador aprenda realmente como fazer algo;
- Muito mais possibilidades;

- Se o utilizador aprender a codificar, ele poderá construir qualquer coisa com código;
- Praticamente sem limites;
- Se o programador aprender uma linguagem de código, é muito mais fácil para ele entender qualquer outra linguagem moderna;
- O utilizador está em controlo do que está a acontecer na totalidade;
- Dá o verdadeiro conhecimento sobre como escrever algoritmos e programar aplicações;
- A maioria de programas de código tradicional são gratuitos;

Desvantagens:

- É mais difícil aprender métodos avançados;
- A manutenção de código de grandes projetos é um desafio, mesmo para programadores experientes;
- Se o programador se desleixar na escrita de código, é capaz de destruir elementos ou simplesmente fazer com que o programa deixe de funcionar como é o desejado:
- É muitas vezes difícil para outros utilizadores entenderem;
- A demonstração de código não é tão clara como nas linguagens de programação visual;
- Para realmente desenvolver um sistema, é preciso aprender uma quantidade significativa de conceitos complexos de programação;

3. Guia para a Programação Visual

3.1. História da programação visual

Admiravelmente, o conceito de quebrar a barreira comunicativa entre o ser humano e o computador não é de todo algo recente. Esta ideia surgiu durante os primeiros desenvolvimentos das linguagens de código que foram se tornando cada vez mais avançadas à medida que que foi crescendo a necessidade de desenvolver tecnologias mais complexas.

Logo a seguir, serão apresentados uns dos principais sistemas antigos que mais contribuíram para facilitar a conceção de aplicações.

A ideia de construir logica usando representações gráficas já é explorada há muitos anos, não muito depois do surgimento da primeira forma de programação computacional, mais especificamente a partir os anos sessenta.

3.1.1. Sketchfab (1963)

O primeiro exemplo surgiu em 1963, com o nome de *Sketchfab*, criado como projeto de doutoramento de Ivan Sutherland. A sua tese acabou por ser pioneira no conceito de interação homem-computador (*HCI, Human-computer interaction*) e, por isso, é considerada a precursora do atual software de design auxiliado por computador (*CAD, Computer-aided design*) (Heather 2021).

Sketchfab, nomeado como uma descoberta na área de computação gráfica, foi usado por Ivan Sutherland para desenvolver elementos técnicos e criativos de computação gráfica, destacando a importância da interação humano-computador. Este programa foi, por exemplo, usado para desenvolver a programação orientada a objetos e a interface gráfica do utilizador (*GUI, graphical user interface*).

Figura 1 – Programa *Sketchpad* no computador "*The Lincoln TX-2*" (1963).

Para executar o *Sketchfab* foi usado um computador com o nome de "*The Lincoln TX-2*" (1958), no MIT, que continha 64 mil palavras de 36 bits. O computador também possuía uma caneta ótica desenhado para rabiscar a tela.

Com a caneta, o utilizador, para além de poder desenhar diretamente no ecrã, também é possível mover secções da arte e modificá-los. Os ajustes a serem feitos são controlados por um conjunto de botões com funcionalidades como apagar, mover, etc. Ou seja, o que é desenhado no ecrã é o *input* do utilizador que sofre alterações conforme o desejo dessa pessoa, usando os botões na parte lateral. Desta forma, o computador recebe as instruções e desenha pelo utilizador.

3.1.2. *GRAIL* (1969)

Grail (graphical input language) foi um projeto elaborado por vários cientistas de computaçao em 1969 durante uma experiência com o objetivo de facilitar resolução de problemas, fornecendo uma interface útil entre homem e máquina.

O projeto investigou técnicas para a interpretação de gestos de mão livre em tempo real num dispositivo chamado *RAND Tablet* (que foi desenvolvido durante uma investigação sobre o Sketchpad, e consiste em registar o *input* do utilizador, com uma caneta, para ser apresentado no ecrã do computador), métodos de representação de ecrã e as suas aplicações para construir um programa de computador através de um fluxograma.

O sistema permite a construção, edição, compilação, debugging, documentação e execução de programas de computador especificados por fluxogramas para auxiliar o homem na formulação de problemas.

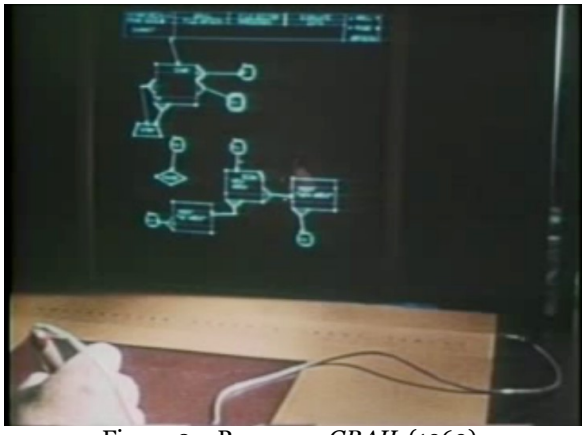

Figura 2 – Programa *GRAIL* (1969).

O utilizador teria que desenhar todas as secções, como está representado na Figura 2, e ilustrar as linhas que conectam esses elementos. A tecnologia de reconhecimento ótico de caracteres (OCR) desempenha o papel de transformar os desenhos e letras em componentes de programação (Instadeq Team 2022).

3.1.3. *Pygmalion* (1975)

David Canfield Smith, ao estudar como as pessoas pensam e comunicam, tentou construir um ambiente de programação que facilitasse a comunicação e que estimulasse a capacidade as pessoas a pensar criativamente.

David, é um informático muito conhecido por inventar os ícones de computador e a técnica de programação por demonstração.

O seu objetivo durante a sua jornada foi tornar computadores mais acessíveis às pessoas. O informático, na mesma tese onde apresentou o conceito de ícones de computadores e de programação por demonstração, introduziu um programa contendo esses elementos, chamado *Pygmalion*.

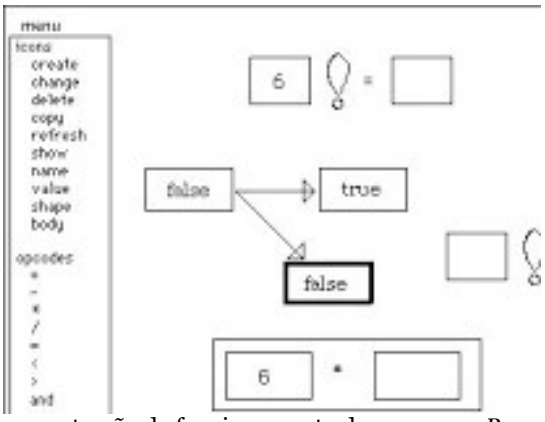

Figura 3 – Demonstração de funcionamento do programa *Pygmalion* (1975)

O criador, na sua tese, ditou que parte da complexidade de programação estava na capacidade do utilizador de manter um modelo mental do estado do seu programa a desenvolver. Então, através de *Pygmalion*, quis tornar possível mostrar esse estado na tela. A ideia principal, resumindo, era visualizar e analisar e modificar cada pedaço de estado que o seu programa pode fazer.

Falando de uma forma mais técnica, *Pygmalion* é uma linguagem de programação e, ao mesmo tempo, um meio para experimentação de ideias. Torna possível uma comunicação mais clara e facilitada entre o homem e computador, através de entidades chamadas de Ícones contendo os conceitos de variáveis, dados, estruturas e funções. Os Ícones são desenhados na tela de exibição.

O coração do sistema é um editor interativo de "lembrança" de ícones, que tanto executa operaçoes como grava no sistema para uma reexecução posterior (Instadeq Team 2022).

3.3.4. *Prograph* (1983)

O desenvolvimento do *Prograph* começou na Universidade *Acadia* como uma investigação geral sobre linguagens de programação baseadas em fluxo de dados, estimulada por um seminário sobre linguagens funcionais conduzido por Michael Levin. Levantou-se a questão: “Já que os diagramas visuais são mais claros que o código, porque não tornar os próprios diagramas executáveis?” Assim, a partir deste ponto, *Prograph* nasceu como uma linguagem visual de fluxo completamente funcional, em 1983. Este projeto foi liderado pelo Dr. Tomasz Pietrzykowski, com a assistência de Stan Matwin e Thomas Muldner (“WayBackMachine” 2005).

Prograph introduziu uma combinação de metodologias orientadas a objetos e um ambiente totalmente visual para programação. Os objetos são representados por hexágonos com dois lados, um contendo os campos de dados e o outro pussui os métodos que operam sobre eles. Ao carregar duas vezes em qualquer um dos lados abriria uma janela mostrando os detalhes desse objeto. Por exemplo, abrir o lado das variáveis mostraria as variáveis de classe na parte superior e as variáveis de instânca abaixo. Clicar duas vezes no lado do método mostra os métodos implementados nesta classe, bem como aqueles herdados da superclasse. Quando um método é selecionado duas vezes, ele abre numa outra janela exibindo a lógica.

No *Prograph*, um método é representado por uma série de ícones, e cada um destes ícones contêm uma ou mais instruções. Dentro de cada método, o fluxo de dados é representado por linhas num gráfico. Os dados fluem na parte superior do diagrama, passando por várias instruções e, eventualmente, retornam à parte inferior ("MACHTECH").

Prograph era uma linguagem de programação comercial exclusiva para o *Macintosh* nos finais dos anos noventa. Com a diminuição de utilizadores do Mac, os criadores da linguagem tentaram converter o *Prograph* para uma versão do *Windows*, mas nunca conseguiram torná-la funcional para esse sistema operativo, o que resultou na estagnação do Mac.

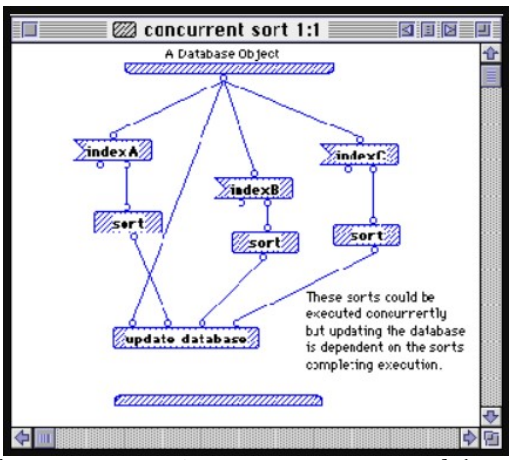

Figura 4 – Operação no programa *Prograph* (1983)

3.3.5. Hypercard (1987)

A Apple, em 1987, lançou uma ferramenta de abstração visual chamada *Hypercard*, para desenvolvimento rápido de aplicações no *Macintosh*. Era uma linguagem de programação baseada no uso de cartas.

Cada carta continha um conjunto de objetos, tal como áreas para definir dados, imagens ou animações e quaisquer outros elementos onde pudessem ser inseridos. Era também possível anexar funções escritas de programação, condicionais (*if, then, else, do…while*, etc) e botões com funcionalidades de pausa, câmara lenta, avanço rápido e assim por diante ("Randocity" 2018).

O *HyperCard* apresentava um *layout* intuitivo de interface gráfica de utilizador e permitiu um desenvolvimento e prototipagem imediatos, sem qualquer codificação. "*Empowerment*" tornou-se numa palavra de ordem quando essa possibilidade foi abraçada pela comunidade Macintosh, assim como a frase "*programming for the rest of us*", isto é, qualquer pessoa, não apenas programadores profissionais. *HyperCard* também possuia uma linguagem de *script* orientada a objetos com o nome de *HyperTalk*, conhecida por ter uma sintaxe semelhante à linguagem casual do inglês, tornando-se muito fácil de usar e ser entendida (Winograd 1996). Esta linguagem também suporta a maioria das estruturas de programação padrão, como "*if-then*" e "*repeate*".

3.2. Linguagens atuais de programação visual

3.2.1. Programação baseada em blocos

Este tipo de linguagem foca-se principalmente em introduzir e ensinar, de uma maneira mais simples e prática, os aspetos principais de programação a crianças e novatos.

A programação visual baseada em blocos consiste em agrupar várias sequências de blocos para formar uma estrutura lexical de um programa. Podem ser blocos de controlo, eventos, variáveis, movimento, etc. Este tipo de linguagem usa um tipo de programação primitiva semelhante a montagem de peças de *puzzle* e um mecanismo de arrastar e soltar que, ao mesmo tempo, dá indicações ao utilizador sobre como e onde cada elemento pode ser utilizado, dando, desta maneira, feedback educativo ao utilizador se as combinações usadas são legítimas (Resnick et al., 2009).

No caso de não ser possível conectar dois blocos, por serem incompatíveis, o próprio programa impede a interação entre eles, ou seja, elimina qualquer hipótese de haver qualquer erro de sintaxe.

Por serem bastante intuitivas e fáceis de aprender, a popularidade destas linguagens continua bastante alta, porém, perdem-se na praticidade, enquanto cada vez mais surge a necessidade de evoluir o nível de complexidade de programas.

Enquanto um software evolui, também a complexidade da sua estrutura interna de código, necessitando assim de intervenções de programação especificas para combater essa complexidade, nomeadamente de "Refatoração de software" - melhoração de qualidade de uma base de código, preservando a sua funcionalidade externa. Isto tornou-se numa parte indispensável no processo de desenvolvimento de um software moderno, e as linguagens de programação visual baseadas em blocos não o possuem e, por isso, apenas mantém a sua popularidade no campo pedagógico (Techapalokul e Tilevich 2015). As subsecções seguintes apresentam os softwares mais populares de programação baseada em blocos.

3.2.2. *Scratch*

Scratch é a maior comunidade de programação para crianças e também uma linguagem de código gratuita com uma simples interface visual. Foi criada em 2007 pelo Media Lab do MIT (Instituto de Tecnologia de Massachusetts). Por não exigir conhecimento prévio de outras linguagens de programação é ideal, principalmente para jovens e noviços programadores, ajudar na aprendizagem de conceitos matemáticos e computacionais. Desta forma é possível criar histórias digitais, jogos e animações (Annette Lamb e Larry Johnson 2011).

Este programa promove o pensamento computacional e as habilidades de resolução de problemas; ensino e aprendizagem de forma criativa; autoexpressão e colaboração e equidade em computação.

A interface de Scratch está dividida em três secções: stage area (área de palco), *block palette* (paleta de blocos) e uma área de codificação, desenhada para colocar e organizar os blocos em *scripts* que podem ser executados pressionando um botão específico. A área de palco apresenta os resultados e todas as miniaturas de *sprites* listadas na área inferior. Com um *sprite* selecionado nessa área, é possível aplicar blocos de comando nele, arrastando-os da palete de bloco para a área de codificação. O separador "Costumes" permite ao utilizador mudar a aparência do *sprite* com um vetor e o editor de *bitmap* para criar vários efeitos, incluindo animações. O programa também contém um separador de som, para colocar qualquer tipo de som ou música a qualquer *sprite*.

Sprites são todas as imagens usadas no desenvolvimento de cada projeto de *Scratch*. Todos os programas criados no Scratch são elaborados por *sprite*s e os *scripts* (instruções) que os controlam (“findout!”).

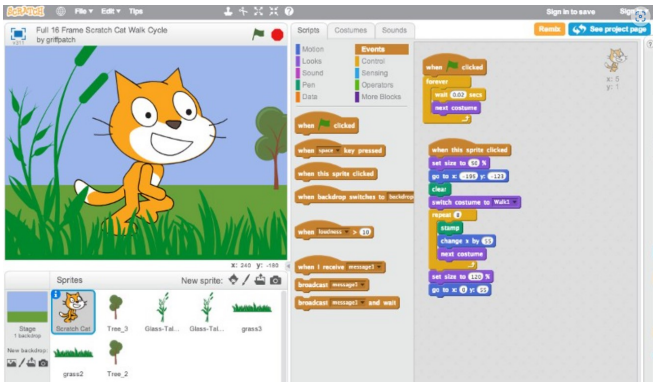

Figura 5 – Capturação de tela do programa *Srcatch*. No canto inferior esquerdo, encontra-se a lista de *sprites*.

Eles podem ser programados para se moverem ao longo do ecrã, para mudar a sua aparência, reagir quando este toca noutros elementos visuais e também para serem controlados pelo utilizador.

Os *sprites* também podem ser programados para comunicar em forma de balões de fala. Cada projeto pode conter um número elevado de *sprites*, e cada *sprite* pode ser controlado por muitos *scripts*.

3.2.3. *MIT app inventor*

Originalmente criada pela *Google*, em 2010, e mantida pelo Instituto de Tecnologia de Massachusetts (MIT), é um programa que permite desenvolver aplicações completamente funcionais para *smartphones* e *tablets* (para o sistema operacional Andriod). O *layout* intuitivo torna o seu funcionamento muito facilitado mesmo para crianças, permitindo construir uma aplicação em menos de trinta minutos. Para este efeito, o *MIT App Inventor* oferece várias ferramentas de programação baseadas em blocos (“appinventor” 2021).

É um programa bastante estável com o objetivo de desenvolver apenas aplicações para pesquisas escolares, impossibilitando o desenvolvimento de qualquer aplicativo profissional.

Este programa usa um tipo de interface gráfico muito semelhante a *Scratch*, que consiste em arrastar e largar elementos visuais para criar aplicações.

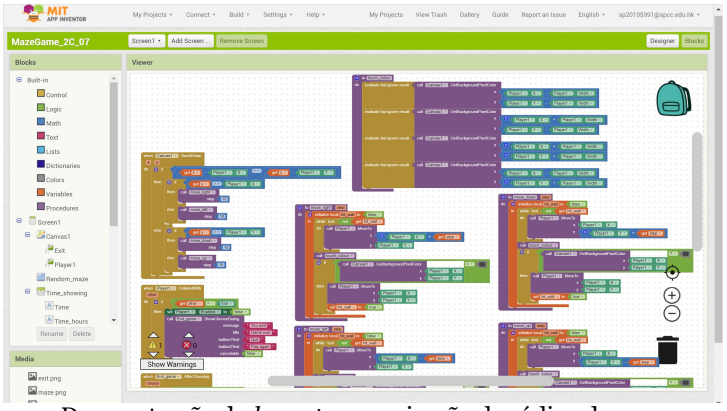

Figura 5 – Demonstração do *layout* e organização de código do programa MIT.

3.2.4. *Google blockly*

Blockly é uma biblioteca desenvolvida em 2012 pela *Google* e pelo MIT, que adiciona um editor de código visual a aplicações moveis e de *web*. O editor *Blockly* usa blocos gráficos capazes de se interligarem a outros elementos para representar conceitos de código como variáveis, expressões de logica, loops e muito mais.

Este programa permite aos utilizadores a aplicação de princípios de programação sem se preocupar com a sintaxe.

Na perspetiva de um utilizador normal, *Blockly* é uma forma intuitiva e visual de construir código. Na perspetiva de um desenvolvedor, *Blockly* é uma interface de utilizador com o objetivo de criar uma linguagem visual que emite código gerado pelo utilizador sintaticamente correto.

Blockly permite exportar blocos para muitas linguagens de programação tais como *JavaScript, Python, PHP, Lua, Dart*, etc (“*Google Developers*”).

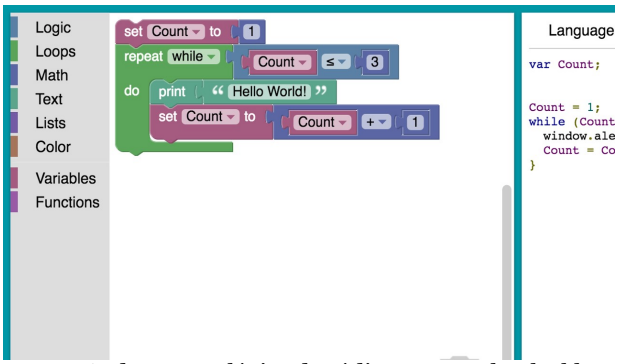

Figura 6 – Demonstração da mesma lógica de código no Google Blockly e em *JavaScript*.

3.3. Programação baseada em fluxo

As linguagens baseadas em fluxo (*FBP - Flow-based programing*) foram desenhadas para transferir dados de um *node* para outro. O resultado dessas interações entre *nodes* define-se como um fluxo, ou seja, uma rede de caixas ("*black box processes*", Morrison) contendo uma série de funções que executam tarefas especificas. Cada função tem um input, *output* ou ambos.

Este tipo de linguagem torna possível criar ligações entre cada um destes componentes conectando o *output* de um elemento ao *input* de outro, criando no final vários tipos de aplicações digitais.

O resultado acaba por ser igual mesmo que o utilizador estivesse a utilizar código tradicional: O user expressa o que ele quer que o computador faça e este apenas segue as instruções e executa-os.

FBP é um paradigma de programação descoberto/inventado por J. Paul Rodker Morrison nos finais dos anos de sessenta. Morrison define-o como uma rede de "caixas pretas" que comunicam um com o outro enviando dados em forma de mensagens chamadas de Pacotes de Informação (PI). Estas mensagens percorrem as vias predefinidas onde as conexões são especificadas externamente para algum tipo de processo.

Estes processos de caixas pretas podem ser reconectados de inúmeras maneiras para formar todo o tipo de aplicações. É uma enorme vantagem, pois dá origem à reutilização de componentes. Assim, em vez de construir tudo a partir do 0, o projeto pode ser contruído

usando componentes pré-existentes, tornando o FBP mais económico (Morrison,1970, 2005, 2012).

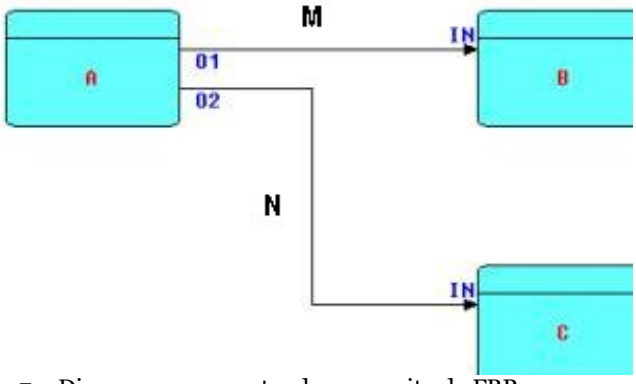

Figura 7 – Diagrama representando o conceito de FBP.

Na Figura 7 está representado um diagrama contendo três processos (A, B e C) e cada um com portos de entrada (*input*) e/ou saída (*output*). Podemos observar que o processo A possui dois portos de saída, O1 e O2, enquanto o processo B tem um porto de entrada e outro de saída, IN1 e O3, e o processo C contém dois processos de entrada, IN2 e IN3. Estes processos estão ligados através dos conectores, M, N e T, com o nome de *buffer* limitado (*bounded buffer*).

A qualquer momento, cada PI é, por um lado, controlado por um único processo ou está a ser transferido de um processo para outro. Quando algum processo recebe um Pacote de Informação, o código desse mesmo processo e usado para manipular o PI.

Esse código afiliado a um processo não é visível no mundo exterior, por essa razão estes processos foram denominados caixas pretas (Morrison 2005).

Na programação visual baseada em fluxo, a ordem dos seus elementos é importante e tem significado.

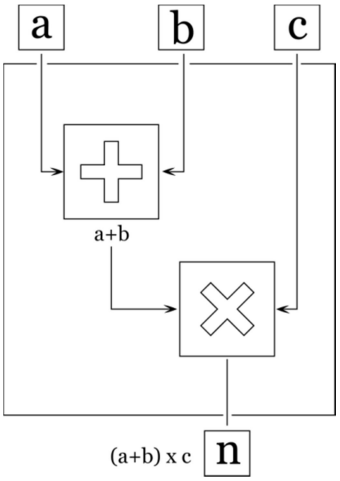

Figura 8 – Representação resumida da funcionalidade de um programa de codificação visual baseada em fluxo

Na Figura 8 está representado um exemplo da funcionalidade de um programa desenvolvido em programação visual baseada em fluxo. Este programa usa três números, a, b, c, e cria a expressão aritmética (a + b) x c. E os restantes elementos em forma quadrada são operadores com as suas respetivas conexões. Chamamos de "operadores" porque funcionam como pequenas unidades operacionais que processem dados. Eles pegam nos dados que chegam dos seus *inputs* e transformam-nos noutros dados para os seus *outputs*.

Até a este ponto, ainda não temos nenhum tipo de controlo do fluxo do programa. Todos os dados seguem o mesmo caminho todas as vezes. O exemplo representado pela Figura 9 mostra uma solução para esse problema.

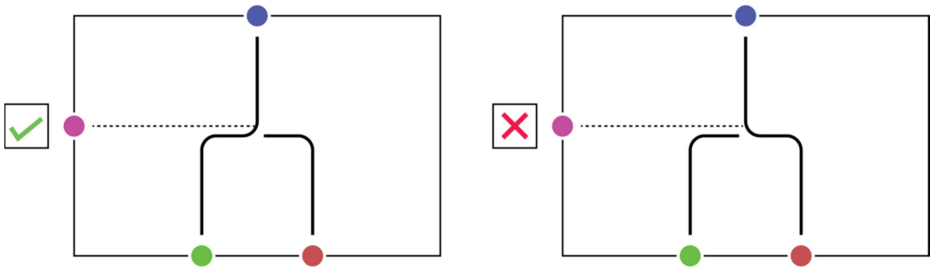

Figura 9 – Ligação entre elementes de *inputs* e *outputs*. O elemento de *input* de cor roxa é um valor *boolean* que decide se o fluxo vindo do elemento *input* azul vai se dirigir para o valor *output* vermelho ou verde.

No exemplo representado na Figura 9 temos dois *inputs* e dois *outputs*. Os *inputs* possuem a cor roxa e azul, e a cor verde e vermelha pertence aos *outputs*.

Para entender o exemplo temos que imaginar esta operação como um interruptor, sendo o *input* roxo um sinaleiro. Sempre que qualquer dado chegue ao azul, logo a seguir terá que ser enviado para verde ou vermelho. O roxo representa o que é conhecido como um valor *boolean* que é verdadeiro ou falso. Dependendo se os dados que chegam em roxo são verdadeiros (imagem da esquerda) ou falsos (imagem da direita), os dados que chegam em azul são enviados para a o verde ou vermelho, respetivamente.

Usando esta abordagem também é possível representar loops, no entanto isso já possui um grau de dificuldade um pouco maior.

3.3.1 *RoboFlow*

Roboflow é uma linguagem de programação visual desenvolvido por Alexandrova (2015). Foi desenhado especificamente a end-users e consiste em programar um robô para qualquer ambiente particular, combinando duas técnicas de programação: Programação visual e programação por demonstração.

A partir do momento que o utilizador demonstra um comportamento, é criado um programa linear padrão. Depois, através do *Roboflow editor*, o utilizador consegue adicionar um sistema de *Branching* e *looping* (Alexandrova et al., 2015).

Branching - Permite realizar alterações sem comprometer o código estável, para evitar qualquer erro na hora de modificar ou inserir alguma funcionalidade num software;

Looping - Podemos descrever o loop num *software* como uma instrução ao sistema para que este repita a mesmo ação até que uma determinada condição seja implantada.

A Figura 10 demonstra o *layout* de *Roboflow*. As caixas representam procedimentos ou funções, cada um contendo *inputs* e *outputs* e os dados viajam de uma caixa para outra através de linhas.

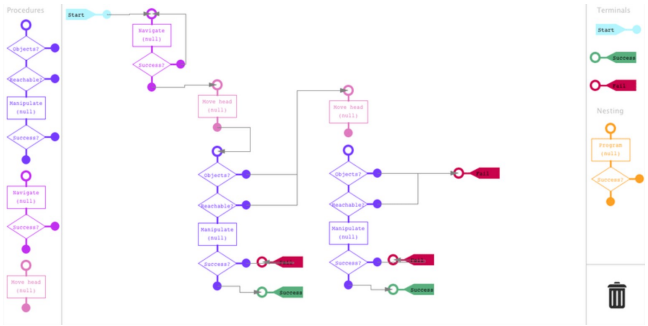

Figura 10 – Representação do *layout* do programa *Roboflow*.

Um estudo foi realizado sobre o funcionamento de *Roboflow* com três grupos de pessoas. Surgiu a conclusão de que os dois grupos, "roboticistas" e "especialistas em linguagem de programação" afirmam que usar uma linguagem de programação baseada em texto é preferível do que usar uma linguagem como o *Roboflow*. No entanto, o último grupo, de "não-programadores" prefere usar *Roboflow* para desenvolver os seus projetos.

3.3.2. *Unreal Engine – Blueprints*

Esta linguagem de programação visual é um sistema dentro do motor de jogo *Unreal Engine,* uma ferramenta gratuita de desenvolvimento de videogame da empresa de desenvolvimento de *software* e jogos digitais *Epic Games*. O motor de jogo oferece várias ferramentas para a elaboração de simulações, edição de vídeos ou som e renderização de animações, num layout intuitivo e atrativo tanto para a comunidade experiente de design de jogos como para iniciantes. Também tornou possível a elaboração de alguns dos jogos mais populares do mercado atual (Moss).

Blueprints é baseado no conceito de usar uma interface baseada em *nodes* para criar vários elementos de jogo e comportamentos.

O sistema é extremamente flexível e poderoso, pois oferece a habilidade aos designers de usar praticamente toda a gama de conceitos e ferramentas disponíveis normalmente para programadores.

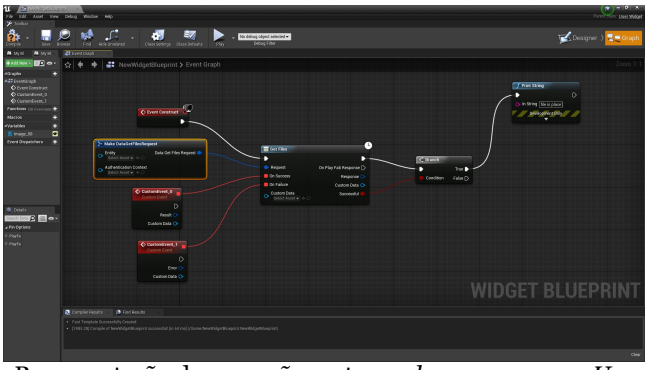

Figura 11 – Representação de conecções entre *nodes* no programa *Unreal Engine*.

Na sua forma básica, *Blueprints* sao adições visualmente programadas no seu jogo. Ao conectar *nodes*, eventos, funções e variáveis com linhas, é possível criar elementos de jogabilidade complexos.

Neste sistema, o primeiro *node* a ser executado é um evento e, logo a seguir, a execução flui através das linhas brancas de execução da esquerda para a direita. É possível visualizar o fluxo de execução enquanto o jogo está a ser testado no editor, o que pode ajudar no *debugging*.

Os dados também fluem através de fios coloridos para corresponder aos tipos de variáveis. Os pinos de *input* são avaliados quando o *node* é executado, rastreando as linhas de dados até que o resultado final seja calculado e fornecido ao node.

Nodes com pinos de execução (*nodes* impuros) armazenam os valores dos seus pinos de *output*, enquanto os *nodes* sem pinos de execução (*nodes* puros) reavaliam os seus *outputs* sempre que um *node* conectado aos seus *outputs* é executado (“*Unreal Engine*”).

3.3.3. *Unity – Bolt*

O *Unity*, criado pela empresa *Unity Technologies*, é outro motor de jogo muito popular entre a comunidade de desenvolvimento independente de jogos. Já foi utilizado para a criação de milhares jogos digitais para computadores, dispositivos móveis e consolas.

A existência do programa *Unity* e de produtos semelhantes, como o *Unreal Engine*, ajudou muitos desenvolvedores a concentrarem-se menos na criação da tecnologia subjacente de um jogo digital e mais nos processos artísticos e criativos, permitindo a utilização de ferramentas poderosas com pouco ou nenhum custo. David Helgason, CEO e co-fundador de *Unity Technologies*, explica que o motor de jogo é "um conjunto de ferramentas usadas para criar jogos num programa que possui uma tecnologia que executa os gráficos, o áudio, forças físicas e ligações de rede".

O que torna o *Unity* tão atraente é o facto de ser um programa muito bem otimizado e simples de usar pela sua estrutura "*user-friendly*", perfeito para um iniciante publicar o seu primeiro título.

Existe também um *marketplace* online de componentes para serem instalados no *Unity*, com o nome de *Asset Store*, onde os utilizadores podem fazer upload das suas próprias criações como também prodem procurar elementos construídos por outras pessoas para ajudar na elaboração de um jogo. Inclui vários tipos de efeitos, coleções de personagens, objetos complexos e *plugins* (Brodkin 2013).

Bolt é um programa presente na *Asset Store*, pronto para ser instalado gratuitamente e vinculado ao *Unity*. Os utilizadores deste motor de jogo, tirando proveito dos sistemas de programação visual de *Bolt*, serão capazes de criar lógica para jogos e aplicações sem a necessidade de escrever código. Essa logica é rapidamente construída através elementos visuais em forma de *nodes*, desenhada de maneira intuitiva tanto para programadores como não programadores. *Bolt* automaticamente traduz expressões complicadas de código para um formato facilmente legível e compreendido por designers e artistas.

Em caso de alguma eventual possibilidade de erro na ligação dos vários *nodes* existentes, o programa consegue analisar e prever falhas antes de executar o projeto, ao salientar a fonte da anomalia sublinhando o elemento que está a ser usado incorretamente (Bowell 2020).

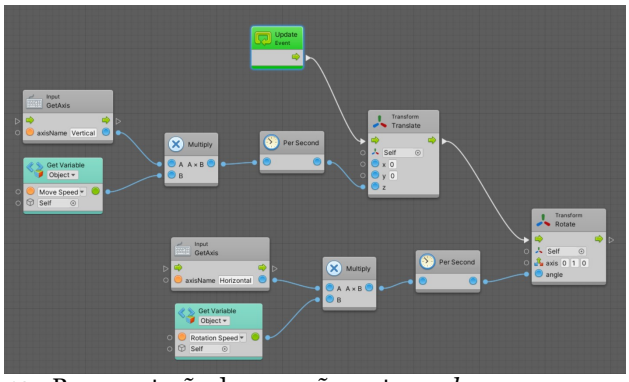

Figura 12 - Representação de conecções entre *nodes* no programa *Unity*.

4. Abstração de lógica em Programação

Os programas avançados de programação visual baseadas em fluxo, como alguns que mencionei anteriormente, são paradigmas de programação que tentam naturalmente usar um método de abstração de lógica e, numa forma óbvia, torna possível a visualização dos elementos de código de um programa. Quanto mais abstração a linguagem tiver do código, mais próximo fica o programador da arquitetura do programa.

4.1. Simplificação de código

A Figura 13 apresenta o código do famoso programa "*hello world*", cujo objetivo é exibir a frase "*Hello, world*" no ecrã. Este simples programa foi construído a partir de uma das primeiras linguagens de programação chamada "*Assembly*" que surgiu em 1947.

```
section  .text
global   _start

_start:

    mov  edx, len
    mov  ecx, msg
    mov  ebx, 1
    mov  eax, 4
    int  0x80

    mov  eax, 1
    int  0x80
```

Figura 13 – Código do programa "*Hello, world*" desenvolvido através da linguagem de programação *Assembly*.

Para os sistemas de *software* atuais, seria impossível escrever código e/ou mantê-lo se continuássemos a usar uma linguagem de programação tão antiga como esta, pois a estrutura do código tornar-se-ia demasiado confusa até para um programador com muitos anos de experiência.

A complexidade e a capacidade dos sistemas de software de hoje em dia só foram alcançáveis graças à "Abstração".

Todos nós usamos a abstração para entender e definir o mundo ao nosso redor (Matschinske 2018).

Quando olhamos para um peixe, não estamos interessados em saber quantas escamas ele possui, da mesma maneira que não nos interessa o número exato de peixes presentes num grande cardume. Abstração é o conceito de esconder detalhes desnecessários.

Este conceito também é aplicável da mesma forma na área de programação. Quando observamos o código na Figura 13, chegamos à conclusão de que, para uma tarefa básica como mostrar duas palavras no ecrã, foram precisas demasiadas linhas de código. Por essa razão, ao longo do tempo foram surgindo linguagens cada vez mais complexas que tentam esconder estes detalhes, oferecendo a habilidade de escrever instruções de uma maneira mais clara e direta.

Este é o outro objetivo das linguagens de programação: esconder detalhes da implementação de código para que o programador possa focar naquilo que realmente importa, que é a lógica do programa a ser desenvolvido e a redução da lacuna entre a linguagem de computador e o pensamento humano.

4.2. Abstração em Programação Visual

Esta subsecção é dedicada a uma exemplificação de abstração de código na programação visual. Para esse efeito, irei mostrar em primeiro lugar o procedimento de uma função de loop em C++, para depois compará-lo á metodologia usada na linguagem de programação *Scratch*. Deste modo, qualquer leitor entenderá que o nível de abstração de código nas linguagens de programação visual é de um nível completamente diferente, capaz de eliminar quaisquer detalhes que possam ser desorientadores, permitindo uma aprendizagem muito intuitiva e/ou simplesmente que o utilizador consiga efeituar uma função com a maior rapidez possivel.

Suponhamos que temos a tarefa de apresentar a frase "*Hello World*" múltiplas vezes por código. Para obter esse resultado é necessário usar um *loop*. Os três tipos de *loops* na linguagem de programação C++ são: *for loop, while loop* e *do-while*. Mas apenas para este exemplo, irei apenas indicar o processo de for *loop*.

For loop é um tipo de *loop* de controlo por entrada, isto é, a condição para a sua execução é testada previamente usando uma variável. Primeiramente, é fundamental iniciar esta variável de *loop* com algum valor e, em seguida, verificar se esta variável é menor ou maior que o valor do contador. Se a declaração for verdadeira, o corpo do *loop* é executado e a variável do *loop* é atualizada. As etapas são repetidas até que a condição de saída chegue ("Geeksforgeeks" 2022).

Expressão de iniciação: Nesta expressão, temos que inicializar o contador de loop para algum valor. Por exemplo: int i=1;

Expressão de teste: Nesta expressão, temos que testar a condição. Se a condição for avaliada como verdadeira, o corpo do loop é executada e iremos atualizar a expressão, caso contrário, sairemos do for loop. Por exemplo: i<=10;

Expressão de Atualização: Depois de executar o corpo do loop, essa expressão aumenta ou diminui a variável do loop em algum valor. Por exemplo: i++.

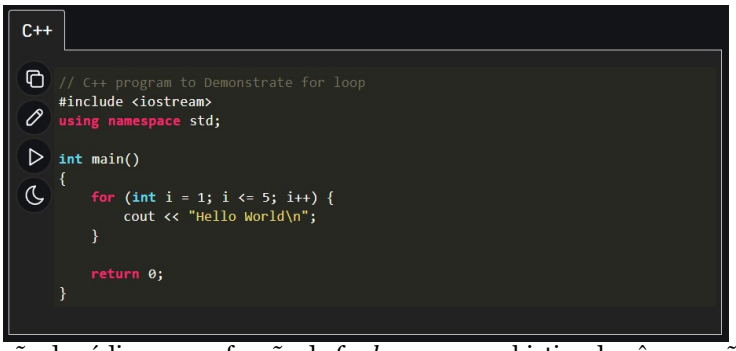

Figura 14 - Representação do código para a função de *for loop*, com o objetivo de pôr em ação um comportamento com base numa condição.

Para efetuar um comportamento parecido ao exemplo anterior no programa *Scratch*, temos que utilizar um bloco de controlo entre os dois tipo de blocos de condição, representados na Figura 14 (*if() then* e *if() then else*). Como pretendemos apenas executar uma função se a condição for confirmada verdadeira, é necessário usar o bloco *if() then*.

O objetivo do seguinte código é mover uma imagem durante um segundo a partir da sua posição inícial para outra posição aleatória no ecrã, quando a tecla “a” do teclado é pressionada pelo utilizador. O procedimento é o seguinte (“Geeksforgeeks” 2021):

Etapa 1: Selecionar o bloco de eventos e escolher o primeiro bloco de controlo que indica o início do programa, arrastando-o para o centro do canvas;

Etapa 2: Carregar no bloco de controlo e arrastá-lo abaixo do elemento criado na primeira etapa;

Etapa 3: Direcionar-se a coleção de blocos de deteção e procurar o bloco "*key space pressed*". Esse elemento terá que ser colocado na condição, que se encontra dentro do bloco de controlo. De seguida, alterar a informação que diz "*space*" para "a";

Etapa 4: Arrastar o bloco "*glide* 1 *sec to random position*" que se encontra dentro da coleção de blocos de movimento. O bloco deve ser posto dentro do bloco *if-then*;

Etapa 5: Por fim, executar o programa.

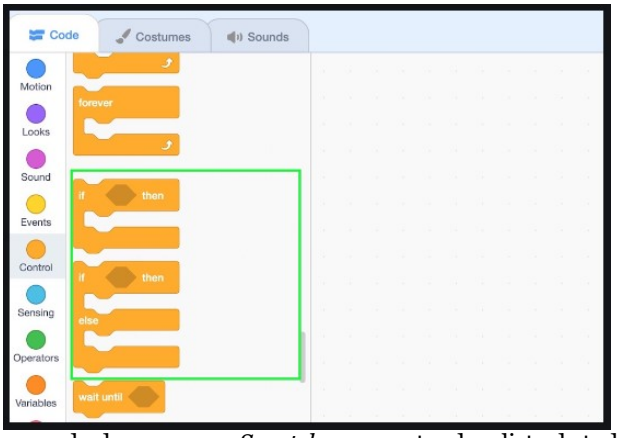

Figura 15 – Coluna do lado esquerdo do programa *Scratch*, apresentando a lista de todos os blocos de controlo. Os elementos de destacados pelo retângulo de cor verde são os blocos de condição.

Figura 16 – Procedimento do código para executar a função de loop descrito pelos passos anteriores.

Como podemos observar, é evidente que o primeiro procedimento usando a linguagem C++, embora tenha menos etapas que o segundo, é muito mais complexo e requer um conhecimento previo consideravelmente superior do uso de todos os elementos que constituem o código. Enquanto que o fluxo de trabalho no programa *Scratch* é mais natural e intuitivo, porque direciona-se logo de imediato ao objetivo da função que o utilizador quer desenvolver, encobrindo o código responsável do funcionamento de cada bloco. O único

ponto negativo no desenvolvimento deste teste em relação ao primeiro procedimento é que a construção de cada comportamento usando os vários tipos de blocos acaba por consumir um pouco mais tempo do que escrever as linhas de código, isto é, se o utilizador tiver um conhecimento claro do funcionamento, da ordem e posicionamento correto dos elementos escritos da linguagem de programação C++.

5. Implementação

Nesta etapa vai ser demonstrada a implementação que pretendo efetuar em duas linguagens de programação visual baseadas em fluxo, no desenvolvimento de um jogo digital. O processo é constituido pela introdução do projeto, onde vai ser esclarecido o objetivo da atividade e a descrição das tecnologias utilizadas, incluindo a informação de cada programa, *Unity* e *Unreal Engine*, e os objetos (*Assets*). Seguido pela meta que tenciono alcançar dentro do jogo, isto é, a condição de vitória, construíndo uma jogabilidade minimamente desafiadora, tanto como a definição da condição de derrota. Por fim, uma demonstração do mapa pretendido, onde a personagem se vai movimentar.

5.1. Introdução do projeto e tecnologias usadas

Esta fase da presente dissertação consiste numa demonstração prática de técnicas de programação visual e, ao mesmo tempo, numa comparação direta entre duas linguagens diferentes. Para este efeito, foi decidido desenvolver um jogo do tipo *sidescroller* 2D, contendo todos os requisitos mínimos para ser totalmente funcional e com uma condição de início e fim.

O projeto foi feito através de dois motores de jogo completamente diferentes, cada um contendo uma linguagem própria de programação visual, tanto como uma linguagem de código tradicional, com as suas características especificas.

O primeiro motor de jogo usado é o *Unreal Engine* 5 com o sistema de programação visual chamado *Blueprints*. O segundo motor, por sua vez, é o *Unity* (versão 2020.2.2f1) que possui um sistema do mesmo tipo com o nome de *Bolt*.

Independentemente da linguagem de programação de cada motor, estes dois programas são conhecidos como um dos principais líderes neste campo, pois fornecem as melhores ferramentas para o desenvolvimento de jogos digitais e têm a vantagem de serem gratuitos. A minha área de trabalho é vinculada com as artes visuais, mais especificamente a ilustração e modelação 3D, mas para este jogo não construi nenhum elemento visual. Em vez disso, utilizei uma coleção de *assets*, chamada "*Sunny Land*", que está disponível para *download* gratuito para qualquer tipo de uso. Desta forma, foi possível manter o foco exclusivamente na parte técnica do projeto, preservando a minha atenção no tema principal desta dissertação.

A coleção é composta por vários elementos visuais, mas para manter a simplicidade do jogo, apenas utilizei alguns dos mais importantes:

- Uma personagem e as suas animações de movimento, salto, queda e *idle*;
- Uma imagem de fundo;
- Um elemento colecionável para a obtenção de pontos (cogumelos);
- Um elemento que apresenta perigo à personagem (espigões);
- Vários blocos para a construção do nível.

Figura 17 – Animação *idle* da personagem.

Figura 18 – Animação de movimento da personagem.

Figura 19 – Animação de salto da personagem.

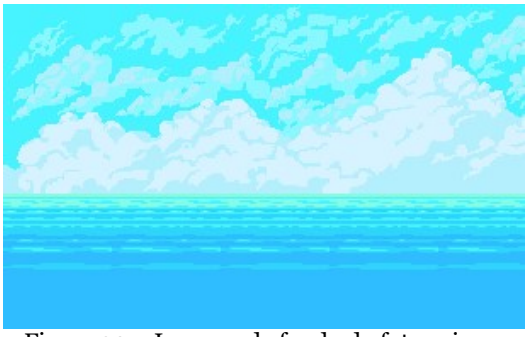

Figura 20 – Imagem de fundo do futuro jogo.

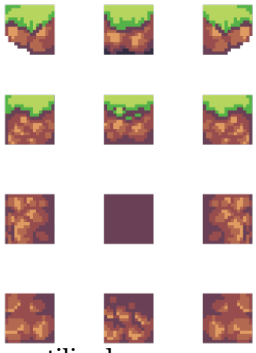

Figura 21 – Blocos utilizados para a construção de nível.

Figura 22 – Elemento de obtenção de pontos.

Figura 23 – Elemento com o objetivo de danificar a personagem em caso de contacto físico.

5.2. Objetivo do Jogo

O jogo consiste em controlar uma personagem num ambiente limitado com o objetivo de colecionar três cogumelos sem cair na área onde se encontram os espigões. Para torná-lo completamente viável para o tipo de jogo que pretendo enquadrar (sidescroller 2D/plataforma), este necessita de ter certas características. Primeiramente, o mapa precisa de ser maior que o ecrã e os objetos que nele se encontram têm que aparecer à medida que a personagem explora o ambiente; em segundo lugar, todos os elementos-chave do jogo (personagem, cogumelos, espigões e todos os blocos que formam o nível) precisam de ter caixas de colisão para desencadear certos comportamentos quando estes se colidem entre si. A terceira característica é a existência de algum tipo de user interface e de algum elemento visual dedicado à pontuação do jogador.

5.3. Mapa do Jogo

Na construção do ambiente, decidi manter a simplicidade ao idealizar um mapa de curta distância e de baixa complexidade. Ficou com as dimensões de 28 x 9, tendo em conta que cada bloco tem o valor de 1

O mapa é composto pelo solo principal, duas colunas em cada ponto extremo, com o objetivo de limitar o jogador a navegar apenas dentro da area interior, e três plataformas aéreas.

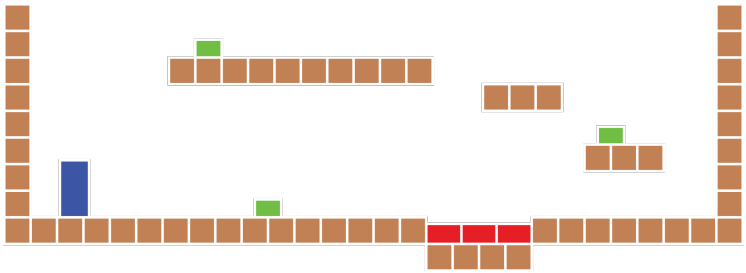

Figura 24 – Esboço do mapa pretendido.

Na Figura 24 está demonstrado um esboço do mapa final que mais tarde seria desenvolvido em cada motor de jogo. A cor azul representa a área inicial onde o jogador é colocado antes de poder avançar, os elementos de cor verde correspondem aos cogumelos que oferecem um ponto cada vez que o jogador colide com eles, a zona vermelha por sua vez representa a area onde iram ser colocados os espigões.

5.4. Fluxo de trabalho de concepção de jogos digitais

Na construção de um jogo, a ordem de preparação dos seus componentes pode variar, pois não existe um *workflow* que seja considerado como a forma perfeita de construir um jogo digital. Porém, durante o mestrado de Design e Desenvolvimento de Jogos Digitais e através de todo o conhecimento que tenho adquirido ao longo dos anos dentro da área de design de jogos, fui chegando à conclusão que poderá existir uma ordem que seja recomendável para um fluxo de trabalho fluido no desenvolvimento de um jogo simples como este.

A parte crucial do jogo é a própria personagem principal e o seu movimento associado ao input do jogador. Por isso, esse seria o primeiro elemento a ser programado. Mas antes de testar o movimento da personagem, teríamos que construir uma base com apenas alguns blocos, ou, se preferível, o mapa na sua totalidade com as respetivas caixas de colisão.

Desta forma já se torna possível experienciar e aperfeiçoar o código de todas as ações da personagem para que também se ajustem ao nível construído.

A segunda parte dedica-se à ligação das animações para cada movimento, seguido pela preparação dos elementos que irão reagir com a personagem, neste caso os cogumelos e os espigões.

Com todos os componentes chave concebidos, resta a elaboração dos elementos de interface do utilizador para que o jogo possa fornecer dados importantes sobre o seu estado e progresso ao jogador. Estes representam se como elementos de texto, imagens e formas

básicas ou complexas, formando neste caso os menus que aparecem no momento que o jogador perde ou vence o jogo e um método de exibir o número de cogumelos obtidos no nível.

6. Desenvolvimento do jogo em *Blueprints* (*Unreal Engine*)

6.1. Criação de projeto e nível

No menu inicial, depois de abrir o programa *Unreal Engine*, podemos ver várias opções em cada separador do lado esquerdo.

Para construir o jogo pretendido, é necessário criar um novo projeto vazio da seguinte forma:

1. Selecionar o segundo separador (*Games*) e a opção "*Blank*";
2. Optar pela opção "*Blueprints*" para que a programação do jogo seja feita através do sistema de programação visual;
3. Colocar um nome do projeto à escolha e carregar no botão "*Create*".

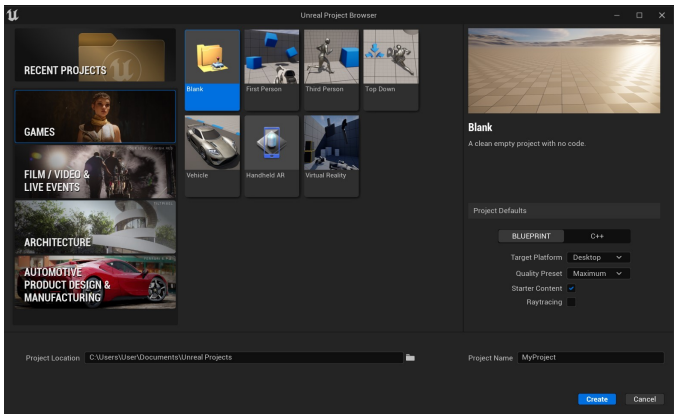

Figura 25 – Menu inicial do programa *Unreal Engine*.

O *Unreal Engine* automaticamente apresenta-nos um ambiente com alguns elementos pré-feitos, mas estes não vão ter qualquer uso para o jogo que queremos desenvolver. O passo seguinte é criar um novo nível, sem nenhum objeto ou qualquer elemento.

1. Carregar na opção "*New Level*" (*File \ New Level*), e no seguinte menu, escolher "*Empty Level*";
2. No content *browser*, a coluna da zona inferior, criar uma nova pasta dedicado a este nível, dentro da pasta pré-existente com o nome de "*content*";

3. Arrastar a pasta com o nome de PNG presente no "*Asset Pack*" de "*Sunny Land*" (*Sunny-land-assets-files* \ PNG), pois é a única pasta que precisamos para este projeto.

6.2. Preparação de *sprites* e do background

Após colocar a pasta PNG, o programa automaticamente mantém todos o seu conteúdo selecionado, ou seja as texturas presente na pasta. Estas texturas, antes de serem utilizadas, precisam de ser compressas de modo a serem apresentadas devidamente para o estilo "*Pixel Art*".

1. Sem desselecionar os elementos, carregar no botão direito do rato e aplicar a opção "*Apply Paper 2D Texture Settings*" (*Sprite Actions* \ *Apply Paper 2D Texture Settings*);
2. Escolher a imagem de "*background*" e selecionar a opção "create *sprite*". De seguida, arrastar o *sprite* criado para a cena principal e mudar o seu valor de X,Y,Z para 0 (mais tarde, durante o desenvolvimento do jogo, estes valores podem sofrer alterações).

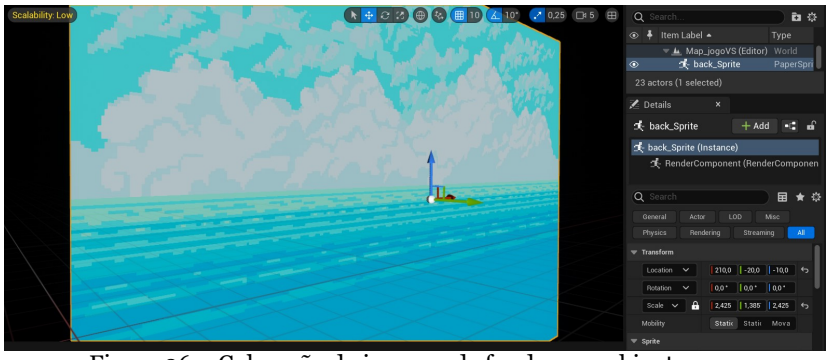

Figura 26 – Colocação da imagem de fundo no ambiente.

6.3. Criação do mapa

Nesta etapa vai ser efetuada a criação do mapa.

Selecionar a imagem com o nome de "*Tileset*" da pasta PNG (*Content* \ *PNG* \ *environment* \ *layers*);

Escolher a opção "*Create Tile Set*" (*Sprite actions* \ *Create tile set*);

1. Abrir o novo elemento criado.

A nova janela que se abriu com o passo anterior serve para listar os vários blocos que depois constituem o ambiente do nível, como também para fazer algum tipo de alteração técnica. Podemos observar logo de seguida que a caixa de seleção é maior que cada bloco do "*Tile set*". Para ajustar, basta mudar o tamanho das telhas no painel do lado direito. Neste caso, é necessário mudar para 16 x 16.

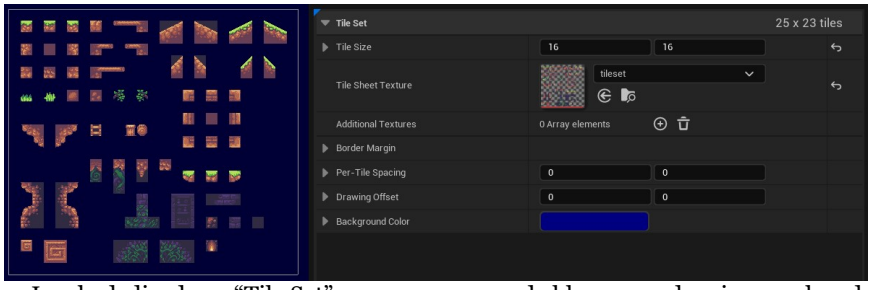

Figura 27 – Janela dedicada ao "Tile Set", para preparar cada bloco para depois ser colocado no nível.

2. Adicionar caixa de colisão para todos os blocos que irão servir como plataforma no nível;
3. Após guardar as configurações anteriores, selecionar o mesmo "*Tile Set*" novamente, na janela principal, e escolher a opção para criar o mapa ("*Create Tile Map*");

O "*Tile Map*" permite abrir uma janela onde já é possível construir cada telha que irá constituir o mapa na sua totalidade, colocando bloco por bloco os elementos que essa telha contém.

4. Depois da telha estar completa, arrastar este novo elemento para a cena principal. (Nota: para não haver algum clipping entre o "*background*" e os outros blocos, é recomendável mover o *sprite* do "*background*" ligeiramente para trás, no eixo de Y.

6.4. Preparação da personagem e das suas animações

Nesta etapa irão ser desenvolvidas as animações da personagem que são compostas por frames separados. Estas animações têm o nome de "*Flipbooks*".

1. Selecionar as imagens das animações da personagem, começando com o a animação idle (*Content \ PNG \ sprites \ player \ idle*). Em cada pasta estão todos os frames de cada animação;

2. Com as imagens da animação Idle selecionadas, escolher a opção "*Create Sprite" (Sprite actions \ Create Sprite*);
3. Selecionar a opção "*Create Flipbook*" com os novos *sprites* que foram criados no passo anterior;
4. Ao abrir o *flipbook*, uma nova janela aparece onde podemos editar a animação. Aqui, foi alterado o valor de frames por segundo pelo valor de 60, e cada *keyframe* para 6;
5. Repetir os passos anteriores para todas as animações que pretendemos colocar na personagem, nomeadamente as animações de movimento, de salto e de queda.

Nota: A animação de salto encontra-se na respetiva pasta com dois frames. O primeiro frame representa a propulsão da personagem e o segundo representa a queda da personagem até chegar à plataforma. Por isso, é necessário separar os dois frames em duas animações diferentes apenas o único frame adequado.

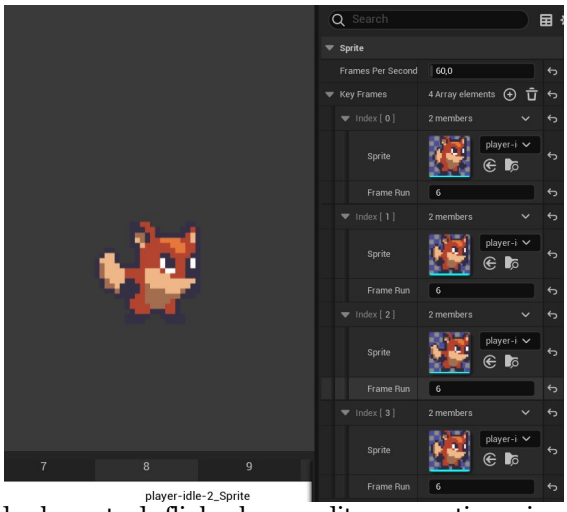

Figura 28 – Janela do elemento de flipbook para editar a respetiva animação de personagem.

6.5. Preparação da personagem

Já possuímos as animações, agora resta fazer os "*blueprints*" da personagem que irá conter todo o código.

O sistema de *visual scripting Blueprint*, como já foi dito anteriormente, é um sistema de *script* baseado no conceito de usar uma interface com *nodes* para criar elementos de "*gameplay*" a partir do *Unreal Editor*.

Uma Classe *Blueprint*, geralmente abreviada como *Blueprint*, é um recurso que permite aos programadores adicionar funcionalidades às classes de jogabilidade existentes.

Os *Blueprints* são criados dentro do *Unreal Editor* visualmente e definem essencialmente uma nova classe ou um tipo de Ator (Ator, no *Unreal Engine*, refere-se a qualquer objeto que pode ser colocado num nível, tal como os *sprites*).

Existem vários tipos de *Blueprints*, cada um o seu próprio uso específico, desde a criação de novos tipos até à realização de eventos personalizados de nível para definir interfaces ou macros a serem usados por outros *Blueprints*.

1. Na pasta "*Content*", criar uma nova pasta dedicado aos *blueprints*. Dentro da mesma, selecionar a opção "*Blueprint Class*" e escolher um nome para este elemento. No meu caso optei por nomeá-lo de "*playerblueprint*";
2. Na nova janela, escolher a opção "*PaperCharacter*";

Ao abrir o novo item, podemos observar, na coluna do lado esquerdo, que este *Blueprint* básico já contém uma cápsula de colisão, um elemento vazio dedicado ao *sprite* e um componente de movimento da personagem.

3. No separador de *Sprite*, dentro da coluna do lado direito, colocar o *flipbook* da animação *idle*;
4. Editar a cápsula de colisão, para que esta se apresente à volta da personagem corretamente;
5. Dentro da lista componentes, na coluna da esquerda, criar um novo elemento chamado "*Spring Arm*";
6. Com o *Spring Arm* selecionado, usar o mesmo método para criar o elemento de câmera. Ajustar este objeto para que se situe de frente à personagem.

Deste modo, temos a personagem principal pronta para ser programada, já com a própria caixa de colisão e uma câmera virtual vinculada, cujo seu objetivo é definir a perspectiva do jogador. Todos estes elementos formam um único actor, que se mantém disponível para ser arrastado para o mapa.

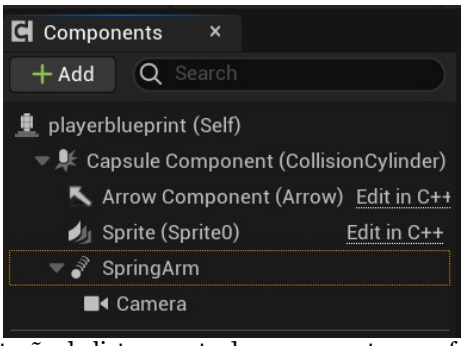

Figura 29 – Representação da lista correta de componentes que formam a personagem.

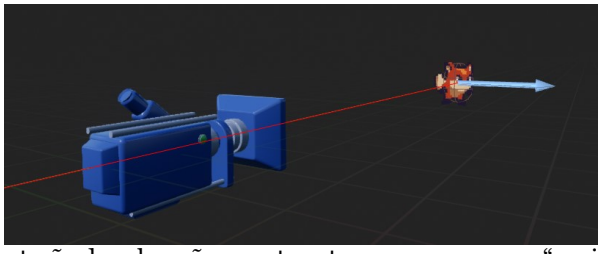

Figura 30 – Representação da colocação correta entre a personagem, o "*spring arm*" e a câmara.

O passo seguinte é a criação de controlos personalizados, mais especificamente, criar um *input* que comunique ao sistema quais são as teclas dedicadas ao salto da personagem e do movimento da mesma dentro do eixo do X. Estes *inputs* irão servir mais tarde como *nodes* dentro do blueprint da personagem.

1. Abrir "*Project Settings*" (*Edit \ Project Settings*) e selecionar a opção de *Input* na lista do lado esquerdo;
2. Criar um "*Axis Mapping*" com o nome de "*Moveright*" (pode ser qualquer nome) e associá-lo aos botões desejados para andar mover a personagem no eixo de X. Um valor de escala positivo irá comunicar ao sistema para mover a personagem apara o lado direito, enquanto um valor negativo dá o efeito contrário;
3. Criar um "*Action Mapping*" para saltar, no meu caso foi escolhido a barra de espaço.

6.6. Controlo de personagem

Para definir os todos os comportamentos da personagem, é necessário abrir o *Event Graph* no *Blueprint da personagem*. Nesta área vão ser colocadas todas as ações dedicadas ao código principal de controlo da personagem e das ativações das respetivas animações,

através da disposição dos diferentes *nodes*. Para um melhor entendimento da forma como os *nodes* se interligam, serão apresentadas imagens contendo o código completo.

6.7. Movimento

Nesta etapa vai ser implementado os controlos para o movimento da personagem.

1. Através do menu de ações, colocar o *node* de *input* do *Axis Mapping* que acabámos de criar nos *Project Settings*;

Para abrir o menu de ações que contém todos os *nodes* existentes, deve-se carregar no botão direito do rato numa zona vazia do *Event Graph*.

A outra forma seria arrastar o rato após carregar no lado esquerdo do rato num conector de *output* de um *node* já existente para a zona vazia. Desta forma o menu abre, apresentando apenas as opções compatíveis com esse *output*.

2. Conectar o *node* do *Axis Mapping* ao *node Add Movement Input* e alterar apenas o valor de X para 1 que a personagem se mova no eixo de X (mais tarde, durante o desenvolvimento do jogo, estes valores podem sofrer alterações).

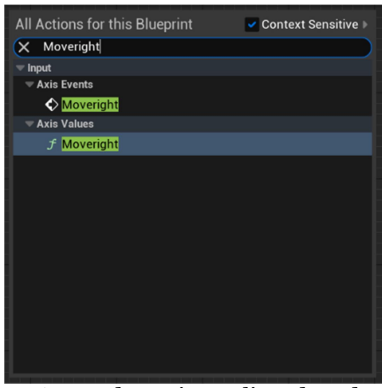

Figura 31 – Menu de ações, onde se situa a lista de todos os *nodes* utilizáveis.

Figura 32 – *Nodes* de movimento da personagem.

6.8. *Nodes* de ativação da animação de movimento

Com o código construído nos seguintes passos, o sistema vai detetar constantemente se a personagem está em movimento, e de acordo com a conclusão uma das animações vai se colocar em ação. Se a personagem não estiver em movimento (falso) a animação de *idle* aciona, em caso contrário (verdadeiro) será a animação de correr.

1. Criar um "*Costum event*" e dar-lhe o nome de "Animação" (pode ser qualquer nome). Esse tipo de *nodes*, geralmente de cor vermelha, executam uma ação customizada.
2. Colocar a referência do componente de movimento da personagem;
3. Ligar a referência ao *node* de "*Velocity*" (*Get Velocity*);
4. Tirar a informação de "*Vector Lenth*" do elemento anterior;
5. Verificar se o "*Vector Lenth*" é maior que 0 com o *node* "*Greater*" (>);
6. Ligar o *node* a outro com o nome de "*select*" no *conector index*;
7. Obter a referência de *Sprite* e ligá-lo ao *node* "*set flipbook*";
8. O *node* "*set flipbook*" deve ser conectado ao *node* "*select*";
9. Dentro do *node* "*select*", colocar a animação de idle na opção "*False*" e a animação de movimento na opção "*True*".
10. Este *script* precisa de ser executado constantemente em todos os frames. Para esse efeito, é necessário colocar uma referência ao *script* e conectá-lo ao *node* com o nome de "*Event Tick*". Deste modo, o sistema nunca vai deixar de operar as animações e de verificar se a personagem está a deslocar no mapa ou não.

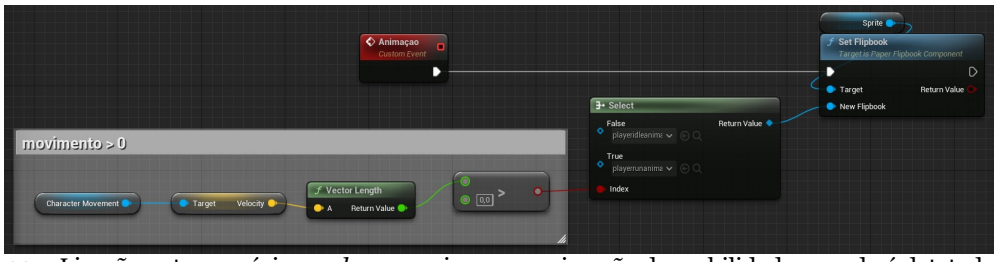

Figura 33 – Ligação entre os vários *nodes* que acionam a animação de mobilidade quando é detetada algum movimento lateral da personagem.

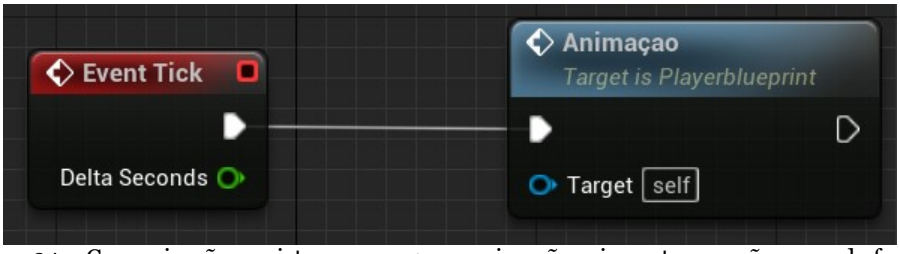

Figura 34 – Comunicação ao sistema para que a animação seja posta em ação em cada frame.

6.9. *Nodes* de rotação de personagem

A este ponto, temos o movimento da personagem bem definido, tal como a sua animação. A próxima etapa é codificar a personagem para que esta mude de direção sempre quando optamos por seguir na direção oposta.

Antes de codificar a rotação da personagem, é necessário fazer umas pequenas alterações ao costum event do movimento.

1. Na coluna do lado direito, criar uma variável do tipo *boolean*. Esta variável tem como objetivo comunicar ao sistema se a personagem está virada para a direira ou não. No meu caso, coloquei o nome de "estaadireita";
2. Ligar o conector de "*Axis Value*" do *node* "*Moveright*" para o *node* "*Greater*" (>) e deixar o valor de 0;
3. Unir o conector "*Return Value*" do *node* anterior para o *node* de "*Branch*". Esse *node* irá colocar em funcionamento diferentes comportamentos se a condição anterior for verdadeira ou falsa;
4. Conectar a referência da variável que foi criada anteriormente para a opção "*True*" no *node* do "*Branch*". De seguida marcar a caixa do valor *boolean* para que esta seja verdadeira.
5. Na opção "*False*", conectar outro *node* de "*Branch*";

6. Ligar o conector de "*Axis Value*" do *node* "*Moveright*" para o *node* "*Less*" (<) e deixar o valor de 0;
7. Associar o conector "Return Value" do *node* "*Less*" à condição do último *node* "*Branch*" que foi criado
8. Conectar a referência da variável para a opção "*True*" no "*Branch*" anterior e deixar a sua caixa vazia do valor *boolean* para que esta seja falsa.

Resumindo, se o valor do eixo X for maior que 0 ("*True*") a variável *boolean* "estaadireita" será verdadeira. Se for falso, é preciso adicionar mais um "*Branch*". Nesse "*Branch*", se a condição do valor do eixo X de ser menos que 0 for verdadeira, a variável será falsa.

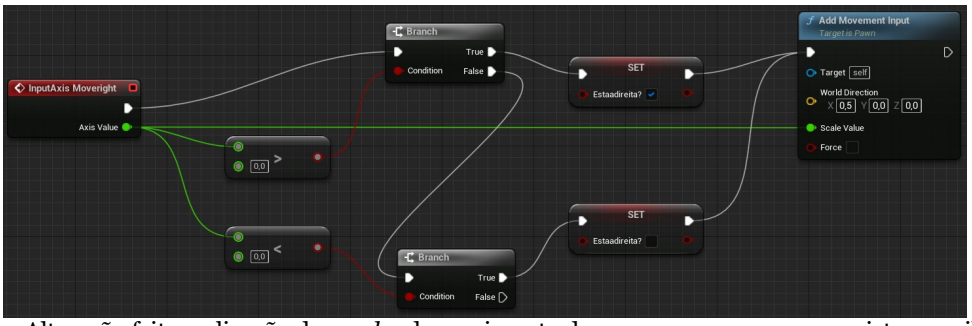

Figura 35 – Alteração feita na ligação dos *nodes* de movimento de personagem, para que o sistema saiba de que lado é que a personagem está virada quando esta se movimenta para a esquerda ou para a direita, usando uma variável costumizada do tipo *boolean*.

Agora que o sistema já tem os dados que informa em que situação o valor de *boolean* é verdadeiro ou falso, chegou a altura de codificar o comportamento que realmente produz a rotação da personagem no eixo de Z.

9. Criar um "*Costum event*". No meu caso, coloquei o nome de "Rotação" (pode ser qualquer nome);
10. Colocar a referência do componente de movimento da personagem;
11. Ligar a referência ao *node* de "*Velocity*" (*Get Velocity*);
12. Verificar a velocidade no eixo do X e ligar esse conector ao *input* do *node* "*Compare Float*". Deixar o seu valor *float* como 0;

O *node* "*Compare Float*" vai comparar um valor anterior conectado ao seu *input* a outro valor escolhido e apresentar três *outputs* diferentes cada um com uma condição: Se o valor anterior for maior, igual ou menor ao segundo número.

13. No caso da velocidade da personagem no eixo de X for maior que 0, conectar o *output* de ">" ao *node* "*Set Control Rotation*", deixando todos os seus valores X, Y e Z como 0. Esse *node*, por sua vez, tem como referência a personagem, através do *node* "*Get Controller*".
14. Repetir o passo anterior, mas com o *output* de "<". Desta vez, é necessário alterar o valor do Z para 180;

No caso de a personagem parar de se movimentar, ou seja, se a sua velocidade for igual a 0, o sistema tem que saber de que lado é que a personagem se deve manter, sem sofrer alguma alteração. Por isso é que criamos o valor *boolean* anterior.

15. Conectar a referência do valor *boolean* "estaadireita" ("*Get* estaadireita") à condição de de um *node* "Branch";
16. O "Branch", por sua vez, será conectado ao *output* de "=" do *node* "*Compare Float*"
17. Se a condição do "*Branch*" for verdadeira, deve-se ligar o conector de "*True*" ao primeiro "*Set Control Rotation*" criado. Se for falsa, ligar o conector de "*False*" para o "*Set Control Rotation*" que tem o valor de Z de 180;
18. Adicionar uma referência ao *script* de rotação e conectá-lo ao *node* "*Event Tick*".

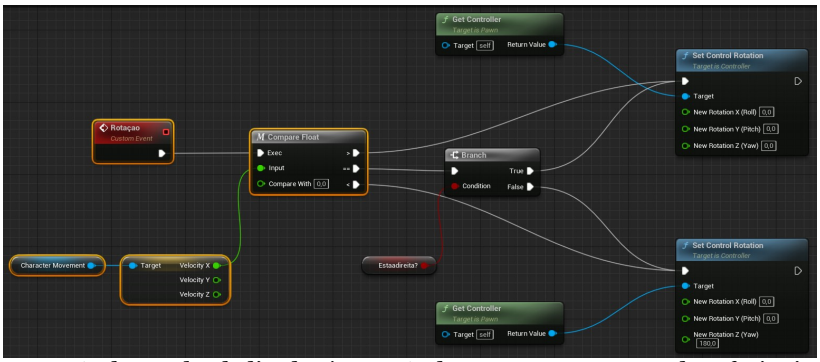

Figura 36 – Representação dos *nodes* dedicados à rotação da personagem, tomando referência da variável criada anteriormente.

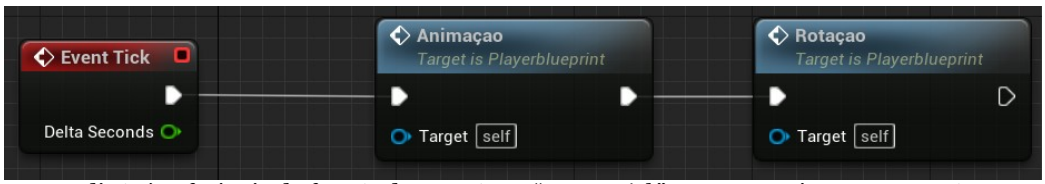

Figura 37 – Adição à referência da função de rotação ao "*Event Tick*", para que seja posta em ação em todos os frames.

6.10. *Nodes* de Salto

Nesta etapa vai ser programado o movimento de salto da personagem.

1. Colocar o *node* de "*InputAction*" com o nome de "*Jump*" que foi criado nos "*Project Settings*";
2. Ligar o conector "*Pressed*" ao *node* de função que também possui o nome "*Jump*";
3. Unir um *node* do mesmo tipo com o *node* de "*Stop Jumping*" ao segundo conector;
4. Retornar para o código dedicado às animações movimento e *idle* da personagem. Procurar o *node* de função "*is Falling*", que já possui a referência do movimento da personagem;
5. Conectar o *node* do "*Costum event*" inicial a um "*Branch*";
6. Associar a condição do "*Branch*" ao *node* "*Is Falling*";
7. Ligar a opção "*False*" ao *node* "*Set Flipbook*" já existente;
8. Conectar a opção "*True*" a outro *node* "*Set Flipbook*" e, tal como o anterior, associá-lo a um *node* "*Select*";
9. Introduzir o *node* "*Get Velocity*" e associar o seu valor de Z ao *node* "*Less*" (<).
10. O *node* "*Less*", mantendo o seu valor de 0, deve ser conectado ao conector Index do *node* "*Select*" que foi criado no oitavo passo;
11. Dentro do *node* "*Select*" mais recente, colocar a animação de salto ("*jump*") na opção "*False*" e a animação de queda ("*falling*") na opção "*True*".

Seguindo estes passos, o sistema irá agora detetar em primeiro lugar, através do *node* "*Is Falling*" se a personagem está a cair dentro do seu ambiente ou apenas se esta não está em contacto com o chão. Se estas condições forem verdadeiras e se a velocidade da personagem no eixo de Z for menor que 0, ou seja, que tenha um valor negativo, a animação de queda vai acionar. No caso contrário a personagem irá se manter na animação de salto após esta ação ter sido ordenada.

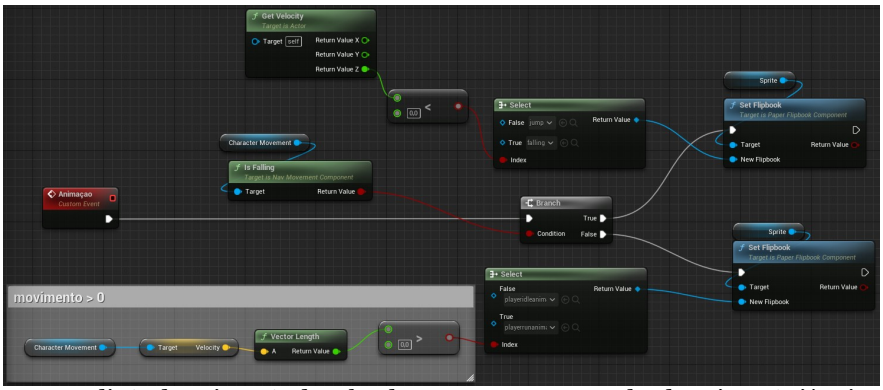

Figura 38 – Adição da animação de salto da personagem aos *nodes* de animação já existentes.

6.11. Ativação de dano de espigões e condição de derrota

Este passo é dedicado à ativação dos comportamentos de colisão entre a personagem e da condição de derrota.

6.11.1. Espigões

1. Na pasta "*Content*", criar um *Blueprint* de class "*Actor*";
2. Abrir o objeto criado e, na coluna do lado esquerdo dentro deste elemento, adicionar um componente "*Paper Sprite*";
3. Escolher o *sprite* dos espigões;
4. Adicionar uma cápsula de colisão e ajustá-lo;
5. Mudar para o separador do *Event Graph* deste *Blueprint*;
6. Colocar o *node* "*Event ActorBeginOverlap*" e ligá-lo ao *node* "*Apply Damage*";
7. Mudar o seu *Base Damage* para 1;
8. Abrir o *Event Graph* do *Blueprint* da personagem.

Aqui o objetivo foi preparar o elemento de *sprite* para que sempre que o sistema deteta um encontro com outro actor com uma caixa de colisão, o espigão irá aplicar dano com o valor de 1.

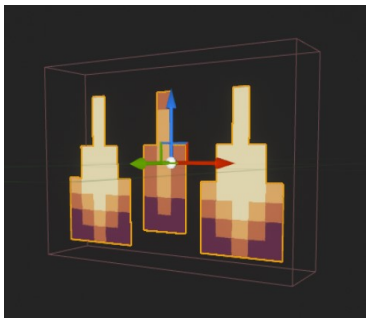

Figura 39 – Representação visual dos espigões e a sua caixa de colisão.

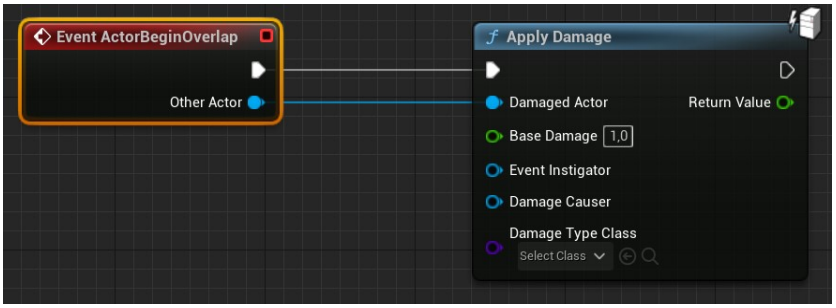

Figura 40 – Aplicação de dano ao elemento dos espigões em caso de colisão com outro elemento

6.11.2. Elemento de UI – *Game Over*

Nesta etapa será desenvolvido um dos elementos de interface de utilizador (UI). No *Unreal Engine*, estes objetos têm o nome de "*Widgets*" e são essencialmente elementos de ecrã como uma forma de comunicar de forma clara e legível informação do jogo para o jogador.

Os seguintes passos descrevem a criação de um elemento de UI para informar ao jogador que a personagem foi derrotada.

1. Na pasta "*Content*", criar um *Blueprint* de class "*Widget*" (*User Interface / Widget Blueprint*);
2. Criar um painel de derrota. No meu caso adicionei uma caixa de texto contendo a frase "*Game Over*" e dois botões: um para voltar para o menu ("*Back to menu*") e outro para sair do jogo ("*Exit*"). Estes elementos são criados na coluna do lado esquerdo;
3. Em cada botão, selecionar a opção "*On Clicked*" presente na coluna do lado direito, no separador "*Events*". Deste modo vão ser criados *nodes* correspondentes a cada botão,

no "*Event Graph*", para definir os comportamentos sempre que estes são pressionados;

4. Conectar o *node* do botão de sair de jogo ao *node* "*Quit Game*";
5. Conectar o segundo *node* para outro de "*Open Level*" (*by Name*) e colocar o nome do nível presente. Assim sempre que este botão é selecionado o jogo vai começar de novo.

Figura 41 – Representação do painel de derrota e a lista dos seus componentes presentes na coluna do lado esquerdo.

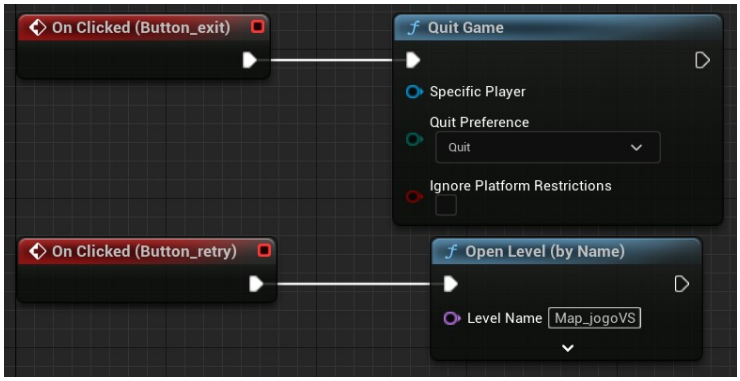

Figura 42 – *Nodes* dedicados à função de cada botão.

6.11.3. Condição de derrota

Neste ponto, resta criar o comportamento referente à condição de derrota. Aqui comunicamos ao sistema que no caso de qualquer dano causado à personagem, o jogo será colocado em pausa e o *widget* "*Game Over*" irá ser adicionado ao ecrã. Nesse momento o jogador terá acesso ao controlo do rato para poder selecionar os botões apresentados.

1. Abrir o "*Event Graph*" do *Blueprint* da personagem;
2. Adicionar o *node* "*Event AnyDamage*" e ligá-lo ao *node* "*Create Gameover Widget*", que por sua vez será conectado ao *node* "**Add to Viewport**".

3. Conectar o último *node* ao *node* "*Set Game Paused*" e associá-lo ao *node* "*Set Show Mouse Cursor*" com a referência do controlo do jogador ("*Get Player Controller*").

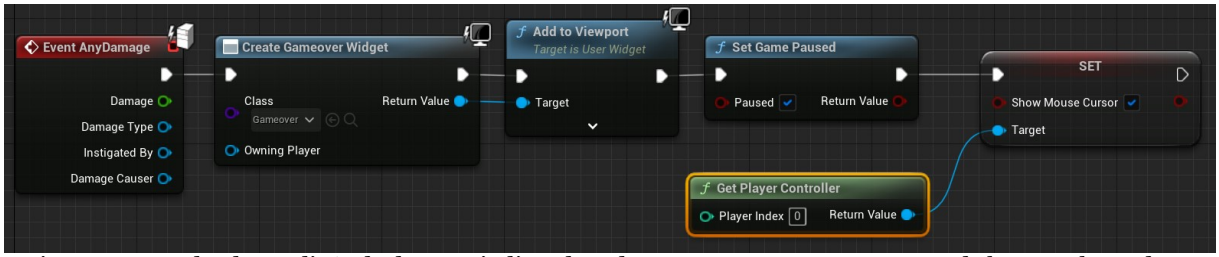

Figura 43 – *Nodes* de condição de derrota, indicando todos os comportamentos no caso de houver algum dano cometido à personagem.

6.12. Ativação de coleção de pontos e condição de vitória

6.12.1. Cogumelos

1. Criar duas variáveis do tipo *Integer* (valor racional) dentro do *blueprint* da personagem, na coluna do lado esquerdo. A primeira variável, neste caso com o nome de "Cogumelos", representa o número atual de cogumelos, com o valor de 0 ("*Default Value*"). A segunda variável, com o nome de "*Total Cogumelos*", possui a informação do número total de cogumelos presentes no nível, por isso devemos mudar o seu valor para 3.
2. Na pasta "*Content*", criar um *Blueprint* de class "*Actor*";
3. Abrir o objeto criado e, na coluna do lado esquerdo dentro deste elemento, adicionar um componente "*Paper Sprite*";
4. Escolher o *sprite* relativo aos cogumelos:
5. Adicionar uma cápsula de colisão e ajustá-lo;
6. Mudar para o separador do "*Event Graph*" deste *Blueprint*;
7. Selecionar a opção "*On Component Begin Overlap*" presente na coluna do lado direito, no separador "*Events*". Deste modo vai ser criado automaticamente um *node* com esse nome no "*Event Graph*" e a partir dele podemos definir um comportamento sempre que outro actor colide com os cogumelos;
8. Conectá-lo com o *node* "*Cast to playerblueprint*" (Nome do blueprint da personagem)";

9. No *node* criado anteriormente, através do seu *output* "*As playerblueprint*", conectar a referência da variável "Cogumelos" ("*Get* Cogumelos"), ligá-la ao *node* "*Add*" e alterar o seu valor para 1;
10. Associar outra referência da variável "Cogumelos" ("*Set* Cogumelos") ao "*Cast to playerblueprint*";
11. Conectar o *node* "*Destroy Actor*" ao "*Set* Cogumelos".

Seguindo estes passos, o sistema vai apagar o elemento referente aos cogumelos sempre que a personagem colide com eles e, ao mesmo tempo, definir um novo número de cogumelos obtidos.

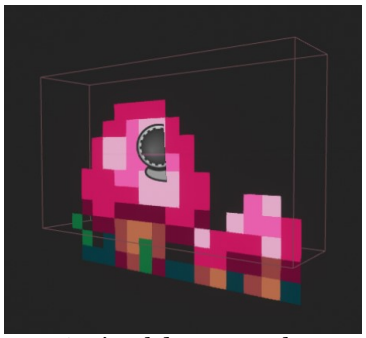

Figura 44 – Representação visual dos cogumelos e a sua caixa de colisão.

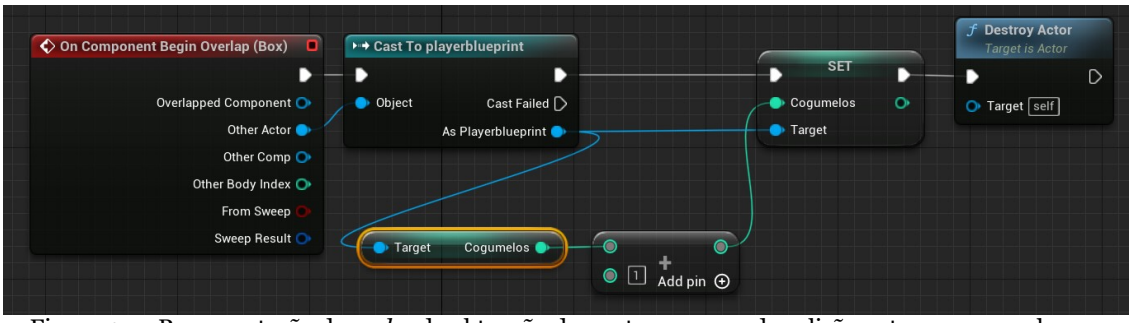

Figura 45 – Representação de *nodes* da obtenção de pontos em caso de colisão entre os cogumelos e a personagem.

6.12.2. Elemento UI – Contador de cogumelos obtidos

Conseguimos definir o comportamento referente à mudança de valor dos cogumelos que são obtidos pela personagem, mas ainda não existe uma forma de o apresentar visualmente. Os próximos passos demonstram como podemos criar um elemento de UI ("*widget*") com um número associado a essa informação.

1. Na pasta "*Content*", criar um *Blueprint* de class "*Widget*" (*User Interface / Widget Blueprint*). No meu caso, a esse elemento foi dado o nome de "*Counter*";
2. Através das opções da coluna do lado esquerdo, colocar no canvas uma caixa de texto contendo o número 0;
3. Introduzir o *sprite* referente aos cogumelos ao lado do elemento de texto anterior, simplesmente para comunicar ao jogador que o número se refere ao objeto que a personagem terá que apanhar;
4. No "*Event Graph*" do *blueprint* presente;
5. Colocar o *node* "*Event Construct*" ao "*Cast playerblueprint*" e ligar este à referencia da personagem ("*Get Player Character*");
6. Arrastar com o rato a partir do *output* de cor azul do *node* "*Cast playerblueprint*" para uma zona vazia. Ao aparecer o menu de ações, escolher a opção "*Promote to variable*", criando assim uma variável. No meu caso foi-lhe dado o nome de "*Player Ref*";

O objetivo do quinto e sexto passo foi criar um elemento que representa uma referência a qualquer variável situada dentro do blueprint da personagem.

7. Voltando para o modo Designer, onde foram criados os elementos de UI anteriores, selecionar a caixa de texto com o número 0;
8. Na coluna do lado direito, no separador "*Content*", Alterar a opção de "*Binding*" para a opção contendo o nome da recente variável ("*Player Ref*") e, de seguida, optar pela referência dos "Cogumelos".

Finalizando, para adicionar os elementos de UI no ecrã, é necessário voltar para o "*Event Graph*" do *blueprint* da personagem e adicionar um código para esse propósito.

9. Introduzir o *node* "*Event BeginPlay*";
10. Conectá-lo ao *node* "*Create Widget*" e alterar o seu objeto de referência para o nome do *widget* referente ao contador de cogumelos ("*Counter*");
11. Ligá-lo ao *node* "*Add to Viewport*".

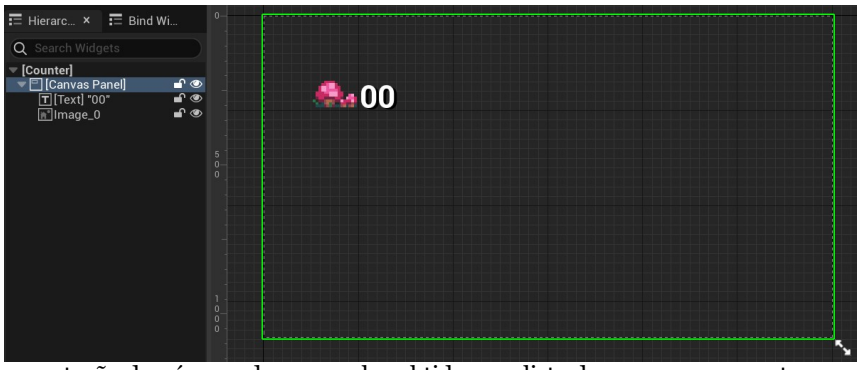

Figura 46 – Representação do número de cogumelos obtidos e a lista dos seus componentes presentes na coluna do lado esquerdo.

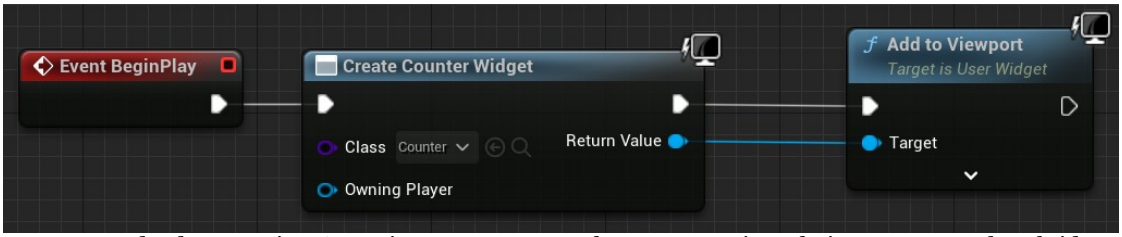

Figura 47 – *Nodes* de comunicação ao sistema para que o elemento anterior relativo aos cogumelos obtidos seja criada e representada no ecrã, no momento que o jogo inicia.

6.12.3. Elemento UI – Painel de Vitória

1. Na pasta "*Content*", criar um *Blueprint* de class "Widget" (*User Interface / Widget Blueprint*);
2. Criar um painel de vitória de jogo, da mesma forma como o painel de derrota.

Para manter a simplicidade do jogo, foi decidido usar os mesmos botões com as mesmas funcionalidades ("*Back to Menu*" e "*Exit*"), com a única alteração da mensagem principal ser diferente, apenas com o objetivo de felicitar o jogador de ter completado o nível.

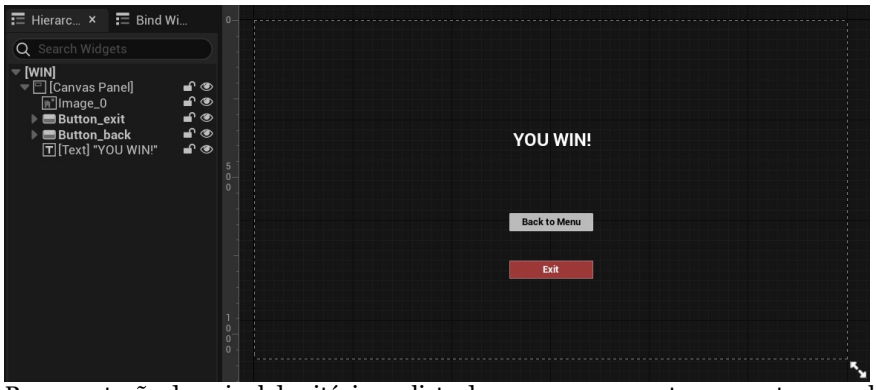

Figura 48 – Representação do painel de vitória e a lista dos seus componentes presentes na coluna do lado esquerdo.

6.12.4. Condição de vitória

1. Voltar para o "*Event Graph*" do *Blueprint* dedicado aos cogumelos:
2. Associar o último *node*, "*Destroy Actor*", à referência da variável "Total Cogumelos" ("*Set Total* Cogumelos");
3. Colocar a mesma referência, mas com o *node* "*Get Total* Cogumelos" e conectá-lo com o *node* "*Cast to playerblueprint*" através do *output* de cor azul;
4. Ligar o *node* "*Get Total* Cogumelos" ao *node* "*Subtract*" e alterar o seu valor para 1;
5. O *node* "*Subtract*" terá que ser conectado com o "*Get Total* Cogumelos";

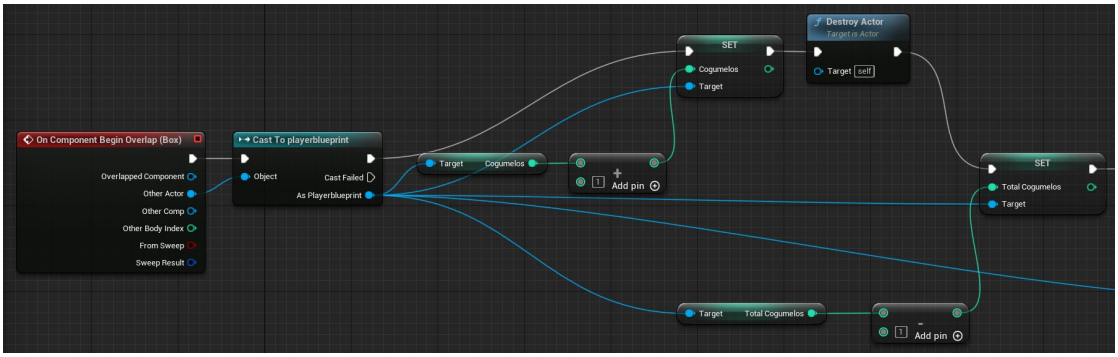

Figura 49 – No caso de colisão entre os cogumelos e a personagem, é retirado um valor ao número total de cogumelos presentes no mapa (Total Cogumelos).

A partir deste momento, sempre que a personagem colidir com um cogumelo presente no nível, mais especificamente quando cada um desses elementos é destruído com esta interaçao, o sistema vai retirar um valor da variável "Total Cogumelos" e oficializar um novo valor.

Nos seguintes passos, vamos definir que na altura em que o valor da variável "Total Cogumelos" for 0, o jogo suspende, ao mesmo tempo que o widget do painel de vitória

aparece, oferecendo também ao jogador controlo do rato para selecionar uma das opções apresentadas no ecrã.

6. Colocar a referência da variável "*Get Total* Cogumelos";
7. Ligá-lo ao *node* "*Equal*" e alterar o seu valor para 0;
8. Associar o *node* "*Equal*" à condição de um "*Branch*";
9. Conectar o *node* "*Set Total* Cogumelos" anterior para o "*Branch*";
10. Conectar o "*Branch*" ao *node* "*Create Gameover Widget*", que por sua vez será conectado ao *node* "*Add to Viewport*";
11. Conectar o último *node* ao *node* "*Set Game Paused*" e associá-lo ao *node* "*Set Input Mode UI Only*".
12. O "*Set Input Mode UI Only*" terá que ser ligado ao *node* "*Set Show Mouse Cursor*".
13. Colocar a referência do controlo do jogador ("*Get Player Controller*") aos dois últimos *nodes*.

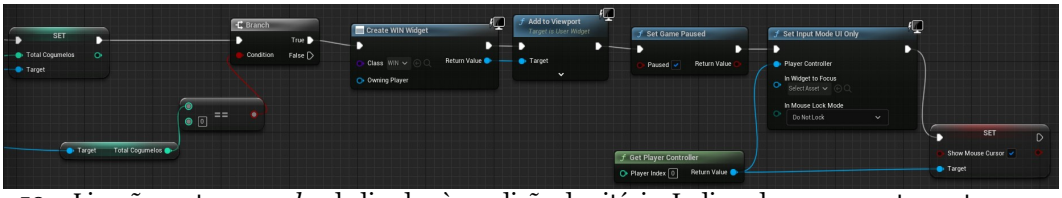

Figura 50 – Ligações entre os *nodes* dedicados à condição de vitória. Indicando os comportamentos em caso do número total de cogumelos presentes no mapa for igual a zero.

6.13. Elemento de UI – Menu principal

Curiosamente, o último elemento do jogo a ser criado vai ser a primeira coisa que o jogador vai observar e interagir. Para construir o menu principal do jogo, temos que ter em conta que este é o ponto inicial da aplicação que contém o gatilho para o começo do nível. Por isso, temos que comunicar ao sistema que o jogo em si deve ser colocado suspensão logo ao início, esperando este comece a funcionar no momento que o jogador carrega no botão para começar a jogar.

1. Na pasta "*Content*", criar um *Blueprint* de class "*Widget*" (*User Interface / Widget Blueprint*);
2. Criar um painel dedicado ao menu principal, usando os mesmos métodos apresentados anteriormente;

3. Dentro do seu "*Event Graph*", codificar o botão "*Exit*" para sair do jogo;
4. No caso do botão "*Play*", conectar o seu respetivo *node* ("*On Clicked (Button_PLAY)*") ao *node* "*Remove from Parent*";
5. Ligar o "*Remove from Parent*" ao *node* "*Set Game Paused*" e deixar a caixa com a legenda "*Paused*" sem marca;
6. Conectar o último *node* ao "*Set Input Mode Game Only*" e ligá-lo ao *node* "*Set Show Mouse Cursor*" com a referência do controlo do jogador ("*Get Player Controller*"). A caixa com a legenda "*Show Mouse Cursor*" deve ser desativada;
7. Voltar para o "*Event Graph*" do *Blueprint* da personagem, para ajustar o código a partir do *node* "*Event BeginPlay*";
8. Antes do "*Add to Viewport*", adicionar o *node* "*Create Menu Widget*" (este nome vai depender da nomeação feita ao *node* dedicado ao menu principal);
9. Ligar o "*Add to Viewport*" ao *node* "*Set Game Paused*" e ativar a caixa com a legenda "*Paused*";
10. Conectar o último *node* ao "*Set Input Mode UI Only*" e ligá-lo ao *node* "*Set Show Mouse Cursor*" com a referência do controlo do jogador ("*Get Player Controller*"). A caixa com a legenda "*Show Mouse Cursor*" deve ser ativada;

Desta forma, na primeira instância do jogo, o nível será posto em pausa, apresentando apenas o elemento de UI do menu principal. O botão "*PLAY*" foi codificado para que sempre que o jogador seleciona a opção, o próprio widget e o cursor do rato vão desaparecer e o nível irá começar automaticamente.

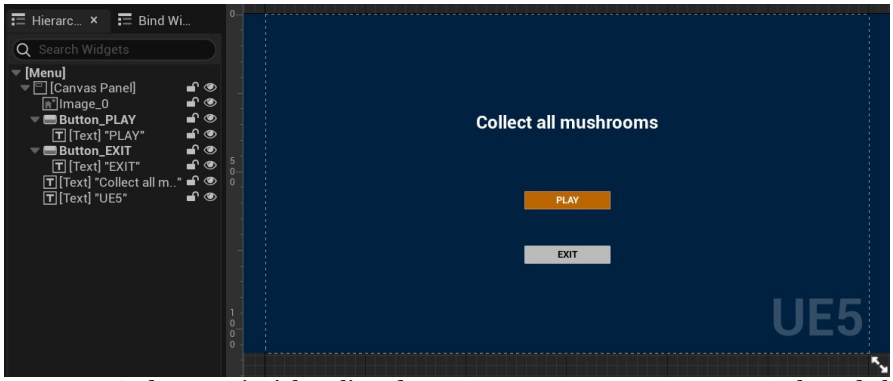

Figura 51 – Representação do menu incial e a lista dos seus componentes presentes na coluna do lado esquerdo.

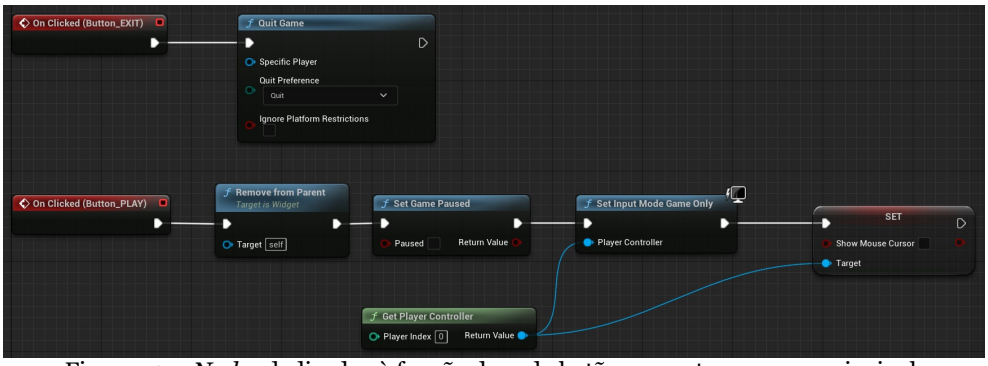

Figura 52 – *Nodes* dedicados à função de cada botão presente no menu principal.

E assim, seguindo todos os passos anteriores, se conclui a construção por completo deste simples jogo feito com o programa *Unreal Engine*.

7. Desenvolvimento do jogo no programa *Bolt* (*Unity*)

7.1. Criação de projeto e nível

1. Abrir o programa *Unity*;
2. Carregar no botão "*New*" e escolher a opção "2D";
3. Criar o projeto depois de colocar o nome;
4. Mudar a persectiva do ambiente para 2D, no segundo separador dentro da área de "*Scene*".

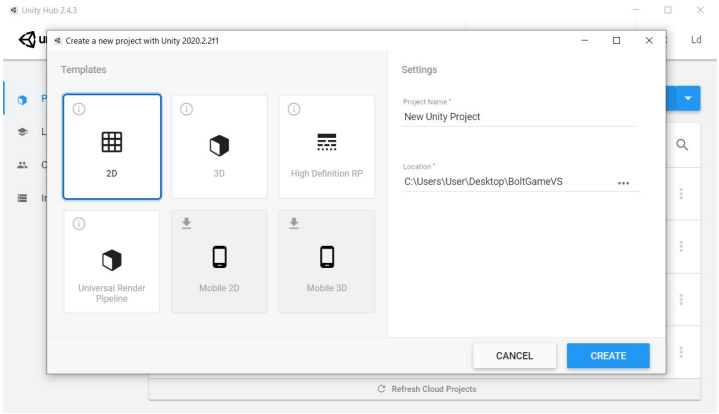

Figura 53 – Menu inicial do programa *Unity*.

7.2. Preparação do mapa

O "*Tile set*", tal como referido anteriormente, é a coleção dos blocos para a construção do mapa de cada nível. É necessário recortar este elemento para que cada bloco possa ser retirado e colocado individualmente como se fossem peças de *puzzle*.

1. Arrastar a pasta relativo ao "*Asset Pack*" de "*Sunny Land*" descarregado inicialmente para a pasta "*Assets*" dentro do *Unity*. No meu caso, foi dado o nome de "Sunny 2D" a essa pasta;
2. No separador "*Inspector*", alterar o número de pixeis por unidade para 16 ao elemento referente à coleção de blocos que irão ser usados para definir o nível. Este

elemento está presente na pasta chamada "*environment*" (*Assets \ Sunny 2D \ Sunny-Land-assets-files* \ PNG \ *environment*);

3. Mudar a opção de "*Sprite Mode*" para "*Multiple*";
4. Abrir o "*Sprite Editor*" do "*Tile set*";
5. Na nova janela, abrir o separador "*Slice*";
6. Alterar o *Type* para "*Grid By Cell Size*" e mudar os valores de X e Y para 16. No fim, selecionar a opção para executar.

Depois de executar podemos observar que cada elemento do "*Tile set*" está dividido devidamente, tornando assim possível retirar cada bloco.

Nota importante: No caso dos elementos visuais do "*Asset Pack*" parecerem embaçados no "*Unity Editor*", é necessário alterar a opção de "*Filter Mode*" para "*Point (no filter)*" com esses elementos selecionados, para que os *sprites* sejam compressos para uma boa e correta visualização de elementos de estilo "*Pixel Art*".

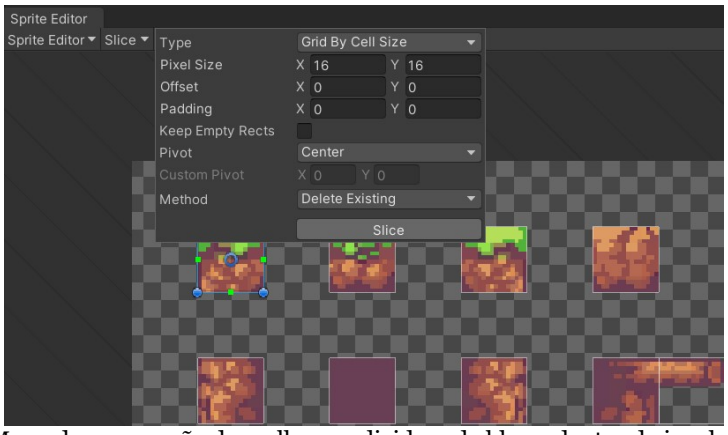

Figura 54 – Menu de preparação da grelha que divide cada bloco, dentro da janela do "*Tile set*".

7.3. Colocaçao das peças do conjunto de blocos

Com as peças do "*Tile set*" já separadas, só nos resta definir um *frame*, como uma espécie de *guideline* para colocar os blocos.

1. Na coluna do lado esquerdo do *Unity Editor*, colocar um "*Tilemap*" retangular (*2D Object / Tilemap / Rectangular*);
2. No separador "*Windows*" Abrir o "*Tile Palette*" para criar as telhas que irão constituir o nível (*Windows / 2D / Tile Palette*). Ao seguir este passo, surge uma nova janela para finalmente contruir os blocos individualmente;

3. Criar uma pasta onde irão ser guardados os blocos. No meu caso guardei a pasta com o nome "*Tiles UNITY*" dentro da pasta "*environment*" (*Assets / Sunny 2D / Sunny-Land-assets-files* / PNG / *environment / Tiles UNITY*);
4. Arrastar o "*Tile set*" anterior para a zona vazia da janela de "*Tile Palette*";

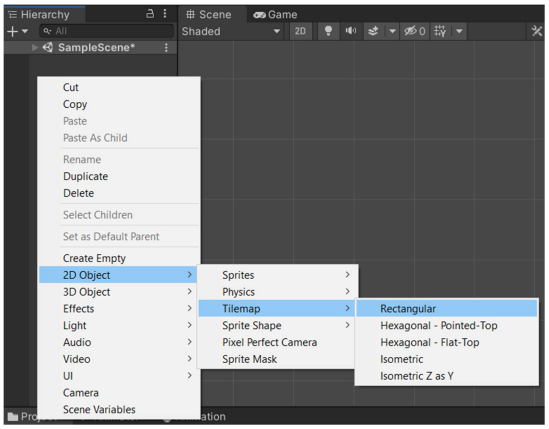

Figura 55 – Colocação de um frame retangular, através do menu que surge com o botão direito do rato dentro da área da coluna do lado esquerdo.

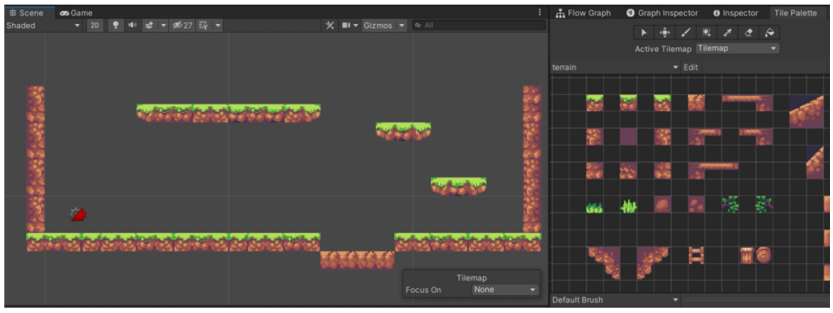

Figura 56 – Janela do "*Tile Palette*", onde se constrói o nível através da coleção de blocos.

Após seguir estes passos, já é possível construir o mapa. O "*Tile Palette*" oferece um sistema intuitivo que nos permite selecionar cada bloco como se fossem pinceis para serem introduzidos no canvas dentro da grelha que foi criada anteriormente.
A próxima etapa será a disposição de caixas de colisão. Tal como muitos outros elementos dentro do *Unity*, as caixas de colisão são componentes separados que se introduzem em cada objeto.

Como o mapa em si apresenta elementos separados, é necessário adicionar várias caixas de colisão em cada lugar onde nós queremos que a personagem embate.

5. Selecionar o objeto referente ao mapa e, na coluna do lado direiro, dentro do separador "*Inspect*", carregar na opção para adicionar um componente;

6. Procurar o elemento "*Box Collider 2D*". Para cada plataforma e parede é necessário colocar uma caixa de colisão;
7. Para cada bloco, deve-se editar a sua caixa de colisão para que esta se ajuste à dimensão do respetivo bloco. Para tal, selecionar a opção "*Edit Collider*".

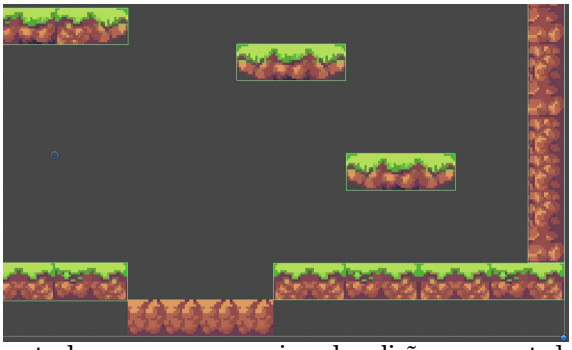

Figura 57 – Blocos de uma parte do mapa e as suas caixas de colisão representadas pelas linhas de cor verde.

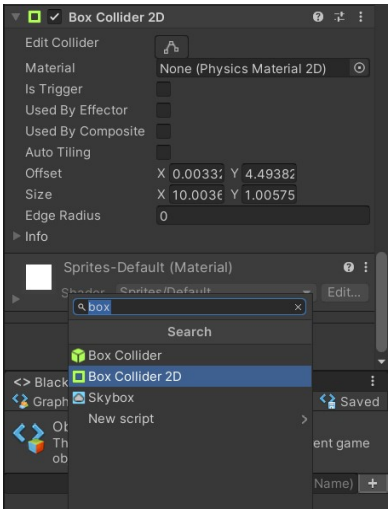

Figura 58 – Menu da lista de componentes, com o seu motor de busca a ser utilizado para encontrar o elemento de "*Box Collider 2D*".

7.4. Background

1. Procurar o elemento visual de imagem de fundo localizado na pasta "*layers*" (*SUNNY 2D \ Sunny-land-assets-files \ PNG \ environment \ layers) e arrastá-lo para a coluna do lado esquerdo ("Hierarchy")*.

Desta forma o elemento vai aparecer no canvas e a partir daí cabe ao desenvolvedor redimensionar a imagem e colocá-la onde parecer mais correto.

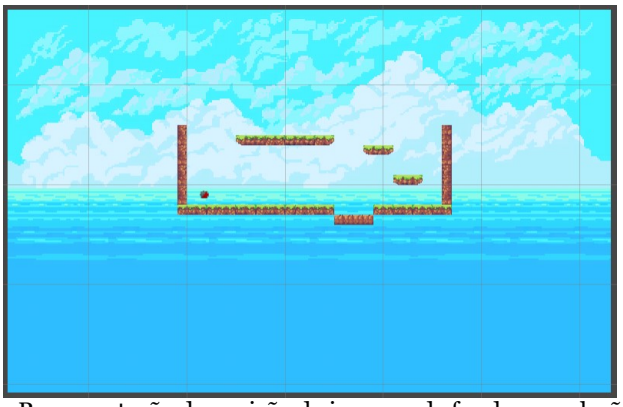

Figura 59 – Representação da posição da imagem de fundo em relação ao nível.

7.5. Preparação da personagem e das suas animações

1. Colocar um dos *sprites* relativos à animaçao idle da personagem no ambiente. Estão situados na pasta chamada "*idle*" (*SUNNY 2D \ Sunny-land-assets-files \ PNG \ sprites \ player \ idle*);
2. Com o *sprite* selecionado, adicionar um componente "*Rigidbody 2D*" e congelar o seu movimento no eixo de Z;
3. Adicionar uma caixa de colisão e ajustá-lo à medida, à volta da personagem;
4. Adicionar, da mesma forma, o componente "*Animator*", para gerir as suas animações;
5. Ir para o separador "*Animation*";
6. Dentro desse separador, que por enquanto se encontra vazio, selecionar a opção "*Create*" e guardar o elemento na mesma pasta de idle;
7. Através da coluna do lado esquerdo dedicado às pastas do projeto, arrastar todos os frames presentes na pasta de animação de idle para a janela de "*Animation*";
8. Ajustar os frames para uma animação suave. No meu caso, alterei a posição dos frames para que a animação tenha 25 frames no total;
9. Repetir os mesmos passos para cada animação que vai ser utilzada.

Nota: Tal como no programa *Unreal Engine*, a animação de salto deve ser separado em duas animações diferentes apenas o respetivo frame.

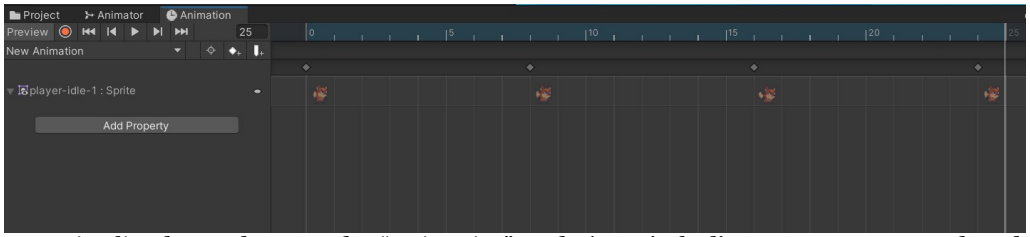

Figura 60 – *Timeline* dentro do separador "*Animation*", onde é possivel editar a cronometragem de cada frame das animações da personagem.

Por fim, resta vincular uma câmara à personagem que irá representar a perspectiva do jogador. Para isso, é necessário descarregar um programa que não se apresenta disponível no *Unity* por padrão.

1. Abrir o "*Package Manager*" através do separador *Windows* (*Windows \ Package Manager*);
2. Procurar, e fazer o download do pacote com o nome de "*Cinemachine*";
3. Após o download, irá aparecer um novo separador contendo o mesmo nome. Para criar uma câmara compatível com este projeto, deve-se selecionar a opção "*Create 2D Camera*" (*Cinemachine \ Create 2D Camera*);
4. Selecionar o elemento "*CM vcam1*" presente na coluna do lado esquerdo para visualizar as opções na coluna contrária do lado direito;
5. Definir a personagem como o objeto que queremos que a câmara acompanhe, na opção "*Follow*";
6. Na mesma coluna, podemos alterar o valor "*Orthographic Size*" para definir o zoom da câmara.

7.6. Definição da ordem das animações (Animator)

Este é um passo importante para comunicar ao sistema o ciclo de transiçao das animações e as condições que cada um terá para estas serem acionadas.

1. Dirigir-se ao separador "*Animator*";
2. Criar uma caixa para cada animação;
3. Selecionando cada uma, definir a respetiva animação na opção "*Motion*" presente no separador "*Inspector*";

A Figura 61, mostra o ciclo de animações que foi desenvolvido no meu caso.

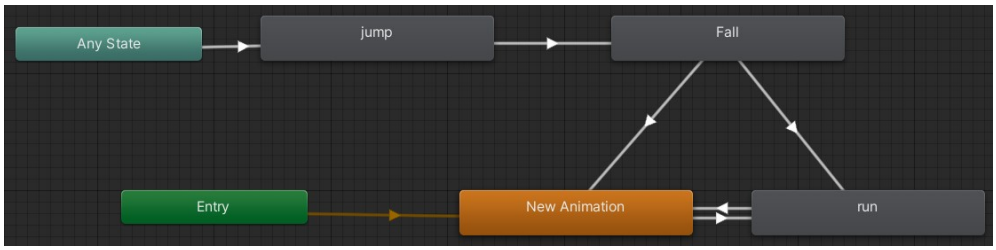

Figura 61 – Representação do ciclo de animações da personagem. O elemento "*Entry*" vai por em ação a primeira animação que queremos que seja acionada no início do jogo, neste caso a animação idle (*New Animation*). A partir deste, só pode ser ativada a animação de movimento (*run*). E, em qualquer estado da personagem (*Any State*), pode ser posta em ação a animação de salto (*jump / Fall*), que por sua vez vai retornar ao estado *idle*.

4. Dentro dos parâmetros ("*Parameters*"), no "*Animator*", colocar dois valores. O primeiro com o nome "*Speed*" do tipo *Float* com o seu valor de 0. O segundo é do tipo *Boolean*, com a caixa marcada, com o nome de "*is grounded*".

Vamos usar este último parâmetro para definir o estado de cada animação, mais especificamente, comunicar em que animação a personagem se encontra no chão do mapa. Futuramente, vai ser codificado como o sistema vai detetar este facto.

Para cada transição que é apresentada visualmente pelas setas brancas, é necessário impor condições diferentes na coluna do lado direito, no "*Inspector*".

As condições de transição são as seguintes:

1. "*Any state*" para "*jump*"

- *is grounded*: "*False*".

2. "*jump*" para "*Fall*"

- Não possui condição, pois é feita naturalmente.

3. "*Fall*" para "*New Animation*" (*idle*)

- *speed*: (*Less*) 0.01;
- *is grounded*: "*True*".

4. "*Fall*" para "*run*"

- *speed*: (*Greater*) 0.01;

- *is grounded*: "*True*".

5. "*New Animation*" (*idle*) para "*run*"

- *speed*: (*Greater*) 0.01.

6. "*run*" para "*New Animation*" (*idle*)

- *speed*: (*Less*) 0.01.

7.7. Programação da personagem

7.7.1. Movimento

Da mesma forma como o *Unreal Engine* possui os objetos *Blueprint* que contêm os códigos dentro do "*Event Graph*", o *Unity* também dispõe elementos parecidos, porém numa abordagem diferente.

Cada objeto dentro do *Unreal Editor* tem a possibilidade de lhe ser adicionado um componente com o nome de "*State Machine*" e cada um desses elementos retêm um *Macro*, cuja sua função é oferecer as ferramentas para desenvolver código e conservá-los dentro do "*Flow Graph*".

1. Criar uma pasta dedicada a todos os códigos que irão ser implementados. No meu caso, decidi guardar dentro da pasta "*player*" ("*SUNNY 2D \ Sunny-land-assets-files \ PNG \ sprites \ player \ Macros*);
2. Com a personagem selecionada, adicionar o componente "*State Machine*";

Nesse componente, sempre que for criado um novo "*State Machine*", deve-se guardá-lo dentro da pasta "*Macros*". Sempre que um novo *Macro* for criado é recomendável guardá-lo dentro da pasta "*Macros*".

3. Dentro da personagem, criar duas variáveis, uma para definir a velocidade do movimento e outra para a força do salto. Neste projeto, dei-lhes o nome de "*speed*" e "*jump*", com os valores de 8 e 17, respetivamente;
4. Com a personagem selecionada, ir para o componente "*State Machine*" e criar um novo *Macro*. No meu caso coloquei o nome de "*Player Controller*";

Neste caso, deve-se começar o desenvolvimento de código a partir do *node* de cor verde "*Update*". Com o clique do botão direito do rato o menu de ações aparece, tal como no *Unreal Engine*.

5. Conectar o *node* "*Update*" ao *node* "*Set Variable*" e escrever "*Movimento*" no campo vazio. Desta forma, criou-se uma variável com este nome;
6. Ligar o conector de *input* de "*Set Variable*" ao *output node* "*Multiply*";
7. Colocar o *node* "*Input: Get Axis (Axis Name)*" e escrever "*Horizontal*" no seu campo vazio. Associar este *node* ao *input* A de "*Multiply*";
8. O *input* B por sua vez, tem que ser associado com a variável "*speed*" ("*Get Variable*");

Com os passos anteriores, definimos uma variável de movimento da personagem baseado no *input*. O *node* de *Input* já está definido logo automaticamente com os controlos do jogador horizontalmente, com os botões "A" e seta esquerda do teclado para ir para a esquerda, e os botões "D" e seta direita do teclado para se deslocar para a direita.
A próxima etapa é a segunda parte do controlo do movimento, onde referênciamos a variável de movimento e definimos a velocidade do mesmo.

9. Ligar o "*Set Variable*" ao *node* "*Rigidbody 2D*";
10. Conectar o *input* ao *node* "*Vector 2*";
11. Colocar a referência de "*Movimento*" ("*Get Variable*") ao *input* X do "Vector 2 Create". O *input* Y deve ser conectado com o *node* "*Vector 2 Get Y*";
12. Ligar o *node* "*Rigidbody 2D Get Velocity*" ao "*Vector 2 Get Y*".

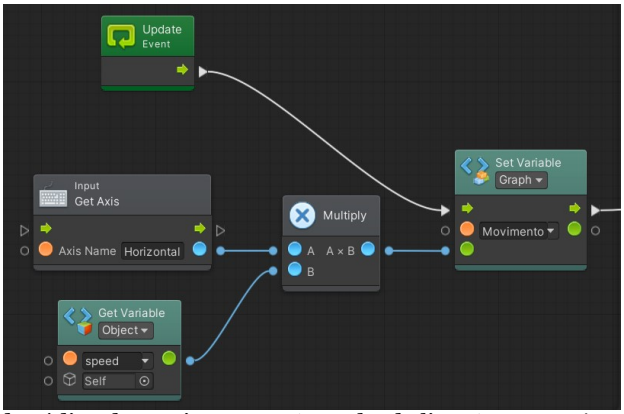

Figura 62 – Primeira parte do código de movimento. São *nodes* de ligação entre o *input* do jogador com a variável de "speed", resultando no movimento da personagem.

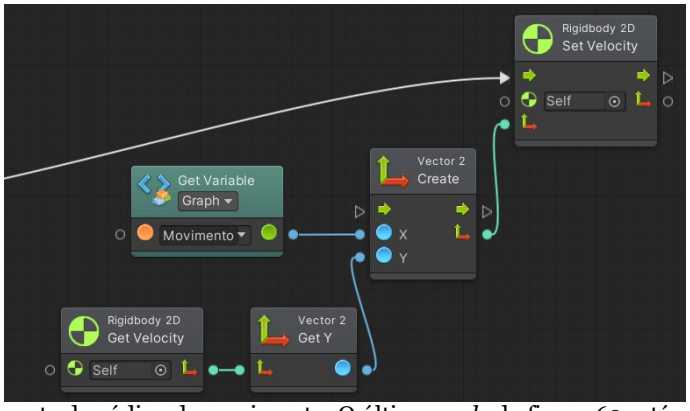

Figura 63 – Segunda parte do código de movimento. O último *node* da figura 62 está conectado com o *node* "*Rigidbody 2D*". Usando a referência da variável de movimento da personagem, estes *nodes* têm o objetivo de indicar que a personagem se move no eixo de X e designar a velocidade no eixo do Y, que representa a gravidade.

7.7.2. *Nodes* de rotação de personagem

1. Introduzir a referência à variável "Movimento" e ligá-lo ao *node* "*Comparison*", deixando o seu valor como 0;
2. Na opção "A<B" de "*Comparison*" colocar um *node* "*Select*", que possui a opção "*True*" e "*False*";
3. Ligar o *input* "*True*" a um valor *Float* de -1. No "*False*", ligar um valor *Float* de 1;
4. Adicionar o *node* "*Transform: Set Local Scale*" e colocar o *node* "*Create Vector* 3 (X, Y, Z)" no seu último *input*;
5. Alterar apenas o valores de Y e Z para 1;
6. Ligar o "*Select*" ao *node* anterior "*Create Vector 3*";
7. Conectar último *node* de "*Rigidbody 2D*" ao "*Transform Set Local Scale*";

Desta forma, a personagem faz a rotação no momento em que o *input* de movimento é ativada, porém, no momento que a personagem pára, ele volta a virar para o lado predefinido onde ele estava virado no inicio do nível. Para reverter a situação, temos que codificar uma forma de comunicar ao sistema em que situação é que a personagem pára de se deslocar e, logo a seguir, deixar a personagem na escala onde este se encontrava antes de travar.

8. Ligar a opção "A=B" ao *input* do *node* "*Branch*", tal como no programa *Unreal Engine*;
9. O seu *output* de "*False*" é conectado com o "*Transform Set Local Scale*" existente, e o "*Rigidbody 2D Set Velocity*" anterior tem que ser conectado com o "*Branch*";

Assim, com este único passo, quando a condição de o movimento for igual a 0 (A=B), a personagem vai rodar premanentemente para o outro lado.

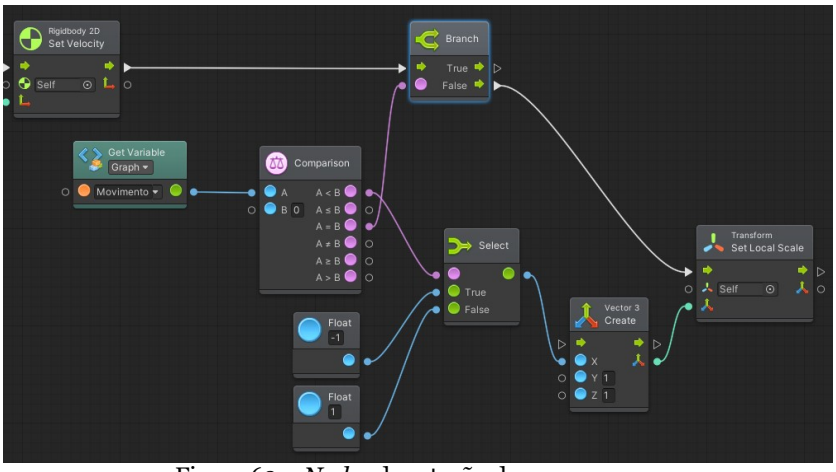

Figura 63 – *Nodes* de rotação da personagem.

7.7.3. *Nodes* de ativação da animação de movimento

1. Colocar a referência "Movimento" ("*Get Variable*");
2. Ligá-lo ao *node* "*Absolute*" e conhectar este elemento ao *node* "*Animator: Set Float (Name, Value)*";
3. Nomear o *float* presente no Animator, "*speed*";
4. Conectar o "*Branch*" anterior e o *node* "*Transform Set Local Scale*", também já existente, ao "*Animator: Set Float (Name, Value)*".

Relembro que quando a personagem se desloca para a direita, o valor do seu movimento é positivo, e é negativo se ela se move para a esquerda.

Através do *node* "*Absolute*", qualquer que seja o valor do movimento da personagem, esse valor é transformado num valor absoluto, ou seja, será sempre positivo. Esse *node* foi usado porque apenas interessa saber se a personagem se está a movimentar ou não, independentemente se é negativo ou positivo. Desta forma, o sistema vai detetar apenas se o movimento é maior que 0; se for, então a animação de movimento irá acionar.

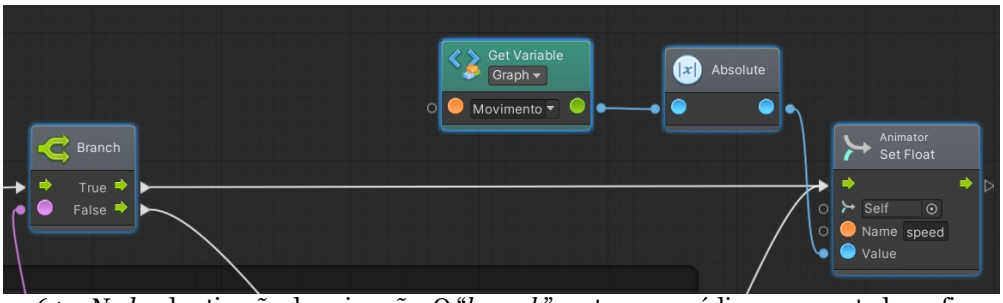

Figura 64 – *Nodes* de ativação de animação. O "*branch*" pertence ao código representado na figura 63.

7.7.4. *Nodes* de ativação da animação de *idle*

1. Introduzir o *node* "*Start*";
2. Conectá-lo com o *node* "*Animator Play*" e mudar o *State name* para "*New Animation*", o nome da animação *idle*.

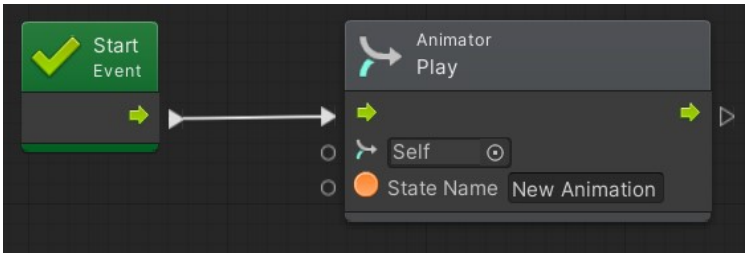

Figura 65 – Indicação ao sistema que a animação idle deve ser acionada logo no inicio do jogo.

7.7.5. *Nodes* de Salto

O próximo passo dedica-se ao controlo de salto e também a desenvolver um código para detetar quando é que a personagem está a tocar no chão, eliminando a habilidade de saltar enquanto este se mantém no ar.

1. Introduzir o *node* "*On Button Input*" e escolher um botão de teclado para executar o salto. No meu caso, foi escolhido a barra de espaço;
2. Ligá-lo a um "*Branch*". A opção "*True*" será conectada com o *node* "*Rigidbody 2D add Force*";
3. Obter a referência da variável "jump" ("*Get Variable*") e conectá-lo ao *input* Y do *node* "*Vector 2 Create*";

4. O "*Vector 2 Create*" por sua vez, tem que ser associado ao segundo *input* do "*Rigidbody 2D add Force*" criado anteriormente.

Assim, com a referência de "*jump*", podemos alterar o seu valor no "*Inspector*", fora do "*Flow Graph*".

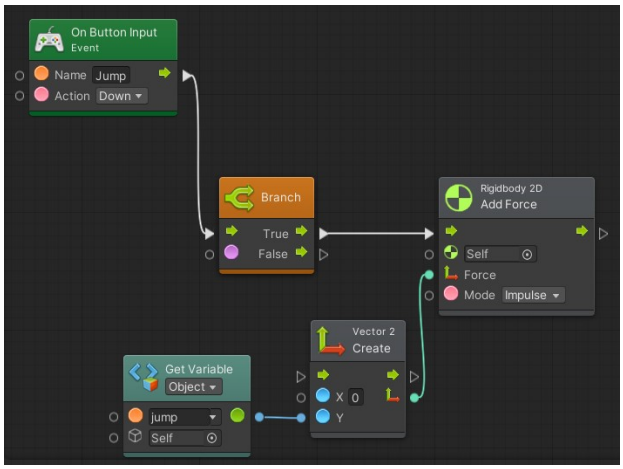

Figura 66 – *Nodes* de ativação da porpulção da personagem no eixo do Y, resultando no salto.

7.7.6. Deteção de chão (*Raycast*)

Existem muitas maneiras de detetar o chão de baixo da personagem, mas esta foi a forma mais simples, direta e mais compatível com o código da personagem já existente que foi possível encontrar no momento.

Consiste na criação de uma linha desde o centro da personagem para baixo. O objetivo é ativar a funcionalidade de salto apenas se esta linha estiver em contacto com uma plataforma.

1. Colocar no canvas o *node* "*Physics 2D: Circle Cast (Origin, Radius, Direction, Distance, Layer, Mask)*". Ligar o seu *input* "*Origin*" ao *node* "*Transform: Get Position*";
2. Alterar os valores de "*Radius*", "*Direction*" e "*Distance*" para 0.4, 0/-1 e 1.1, respetivamente;
3. Colocar o *node* "*Layer Mask*", alterar a opção para o nome do layer onde se encontram as plataformas. De seguida, conectá-lo ao ultimo *input* do "*Circle Cast*";
4. Ligar o "*Circle Cast*" ao *node* "*Expose Raycast 2D*";
5. Conectar o seu *output* de "*Collider*" para o *input* A do *node* "*Not Equal*";

6. O *input* B será conectado com o *node* "*Null*";
7. Ligar o *output* A ≠ B ao *input* referente à condição do "*Branch*" criado anteriormente.

O *node* "*Expose Raycast 2D*" é o elemento que define o que é que a linha deteta. Com este código, foi definido que no caso da linha detetar uma colisão dentro do layer das plataformas, ou seja se esse comportamento for não for nulo, a personagem poderá saltar.

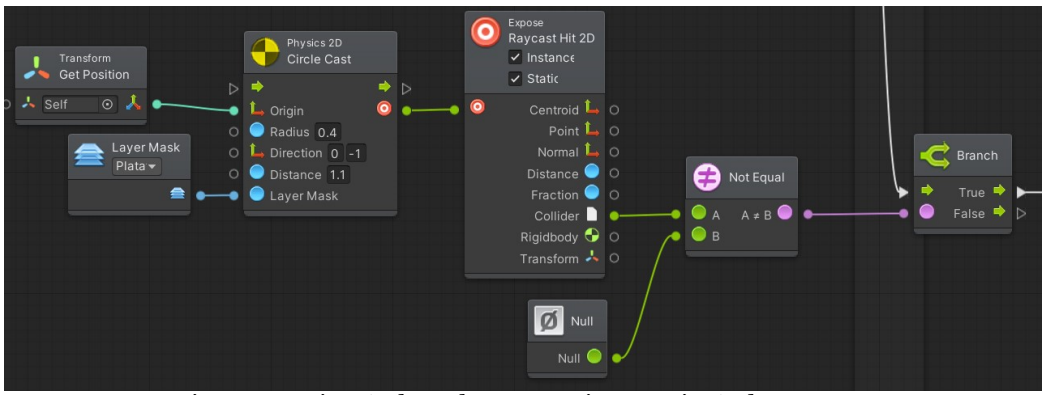

Figura 67 – Ligação de *nodes* que consiste na criação do *Raycast*.

7.7.7. Ativação da animação de salto

Para terminar a etapa do controlo de salto, resta comunicar ao sistema em que situação o parâmetro "*is grounded*" é verdadeiro, com o objetivo de ativar a animação de salto presente no ciclo de animações no "*Animator*". Afortunadamente, já temos isso definido, porque acabámos por desenvolver, nos passos anteriores, um código com esse mesmo propósito. Assim, só é necessário colocar uma referência do "*Animator*".

1. Copiar o código completo do passo anterior, sem contar com o "*Branch*";
2. Ligar o *output* A ≠ B ao *input* de "*Value*" do *node* "*Animator Set Bool*", colocando o nome "*is grounded*";
3. Conectar o *node* existente "*Animator Set Float*" ao "*Animator Set Bool*" anterior.

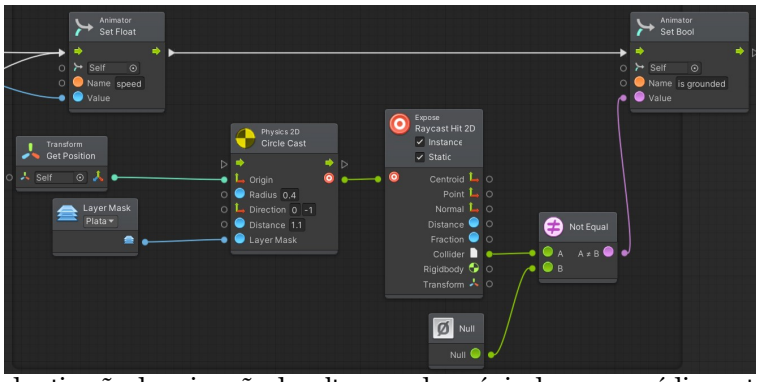

Figura 68 – *Nodes* de ativação da animação de salto, usando a cópia do mesmo código anterior para definir a situação em que se pode acionar a respetiva animação.

7.7.8. *Nodes* de ativação da animação de queda

1. Colocar o *node* "*Update*" e conectá-lo a um "*Branch*". No seu *output* de "*True*", Ligar o *node* "*Animator Play*". Mudar o "*State Name*" para "*Fall*" (Nome da animação de queda);
2. No *input* do "*Branch*", a sua condição será ligada ao *node* "*Less*";
3. O *input* A tem que ser associado ao *node* "*Vector 2 Get Y*";
4. Conectar o *node* "*Rigidbody 2D Get Velocity*" no "*Vector 2 Get Y*" anterior;
5. Associar o valor do tipo *Float* de 0 ao *input* B do *node* "*Less*".

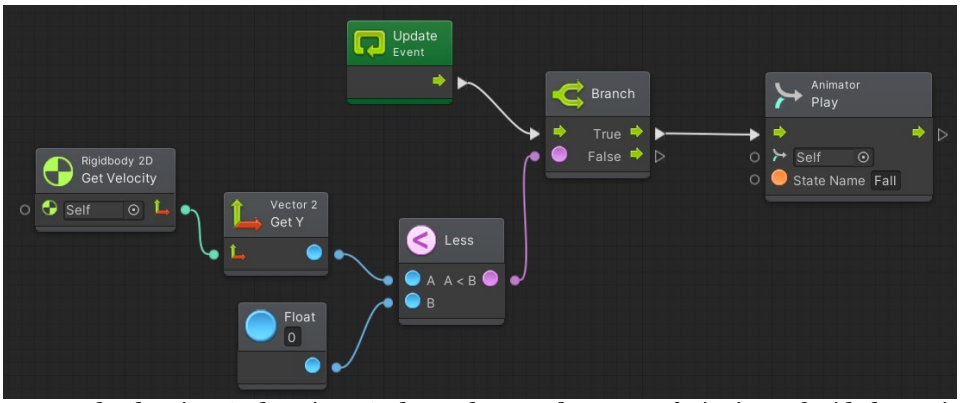

Figura 69 – *Nodes* de ativação da animação de queda, usando como referência a velocidade no eixo de Y.

7.8. Elemento de UI – Menu principal

A melhor forma para gerar um menu principal é criar uma nova cena, separada do nível e adicionar um código para transferir o jogador para o nível no momento que este seleciona o botão de "*Play*".

1. Criar uma nova cena dentro da pasta "*Scenes*" (*Assets \ Scenes*), com o nome de "*Menu*";
2. Abrir a cena anteriormente criada;
3. Na coluna do lado esquerdo do *Unity Editor*, Colocar o elemento "*Canvas*" (*UI \ Canvas*);
4. Com o canvas selecionado, alterar a opção "*UI Scale Mode*", na coluna do lado esquerdo, no "*Inspector*", para "*Scale With Screen Size*" e mudar para uma resolução desejada;

Nota: O contorno do elemento de Canvas deverá corresponder à resolução do ecrã do jogador.

5. Usando o mesmo método para criar o Canvas, colocar um "*Panel*" (*UI \ Panel*), uma caixa de texto (*UI \ Text – TextMeshPro*) e dois botões (*UI \ Button – TextMeshPro*), da mesma forma como criamos anteriormente no programa *Unreal Engine*;
6. Para cada botão, adicionar um "*Flow Machine*" e dentro dele, um novo macro para definir as funcionalidades;
7. Dirigir-se para o "*Flow Graph*" do botão "*PLAY*" e introduzir o *node* "*On Button Click*";
8. Ligá-lo ao *node* "*Scene Manager Load Scene*" e colocar o nome da cena onde se situa o nível. No meu caso é "*SampleScene*";
9. Dirigir-se para o "*Flow Graph*" do botão "*EXIT*" e colocar o *node* "*On Button Click*";
10. Conectá-lo ao *node* "*Application Quit*".

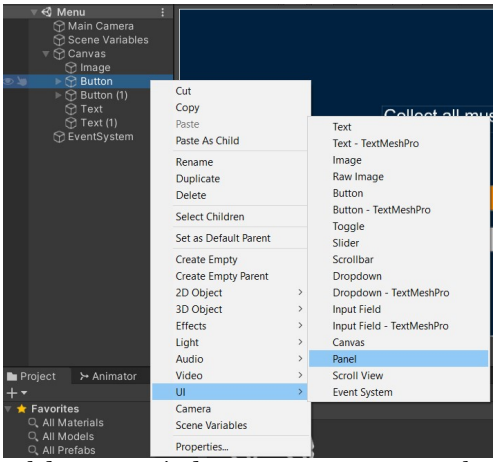

Figura 70 – Colocação de um painel de UI através do menu que surge com o botão direito do rato dentro da área da coluna do lado esquerdo.

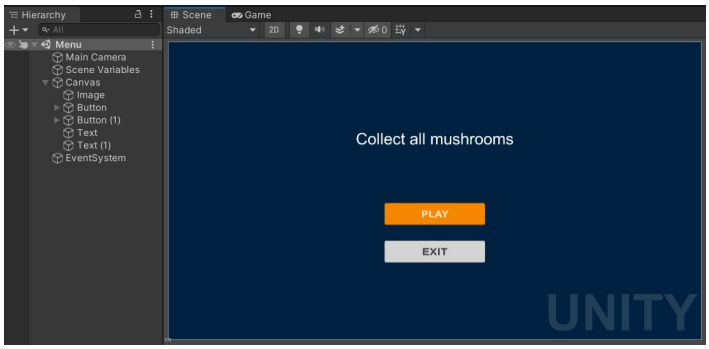

Figura 71 - – Representação do menu incial e a lista dos seus componentes presentes na coluna do lado esquerdo.

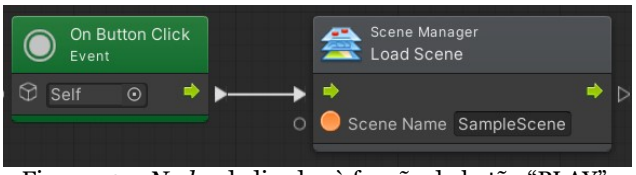

Figura 72 – *Nodes* dedicados à função do botão "PLAY".

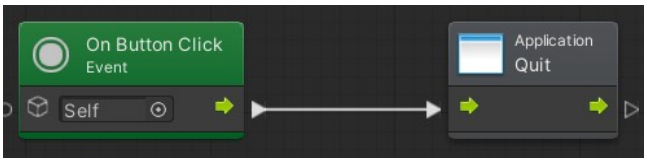

Figura 73 – *Nodes* dedicados à função do botão "EXIT".

7.9. Ativação de dano de espigões e condição de derrota

7.9.1. Espigões

O primeiro passo nesta etapa é definir um evento costumizado onde vai ser programado o comportamento de colisão entre a personagem e os espigões. De seguida, dentro do "*Flow Graph*" da personagem vamos realmente codificar o que acontece quando estes dois elementos colidem.

1. No *Unity Editor*, arrastar os *sprites* dos espigões para o ambiente;
2. Adicionar o componente de "*Box Collider*" e ajustar a caixa de colisão;
3. Adicionar o componente de "*Flow Machine*" e criar um novo macro;
4. Dentro do "*Flow Machine*", colocar o *node* "*On Collision Enter 2D*" e ligá-lo a um "*Branch*";
5. No *output* de "*True*", conectar esta opção pelo *node* "*Costume Event Trigger*" e colocar o nome de "*Death*";

6. O *output* "*Collider*" de "*On Collision Enter 2D*" terá que ser conectado com o último *input* do *node* "*Costume Event Trigger*" e também com um *node* "*Game Object Compare Tag*" com o Tag nomeado de *Player*.

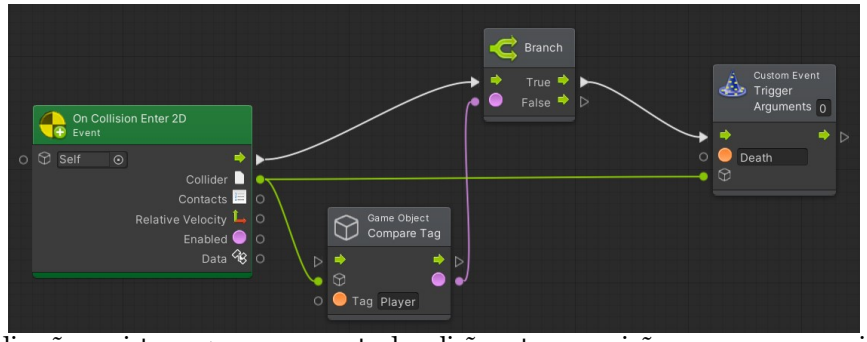

Figura 74 – Indicação ao sistema que no momento de colisão entre os espigões e a personagem, irá por em ação um comportamento com o nome de "*Death*".

7.9.2. Elemento de UI – *Game Over*

1. Criar um painel dedicado à derrota do jogador, usando os mesmos métodos apresentados anteriormente;

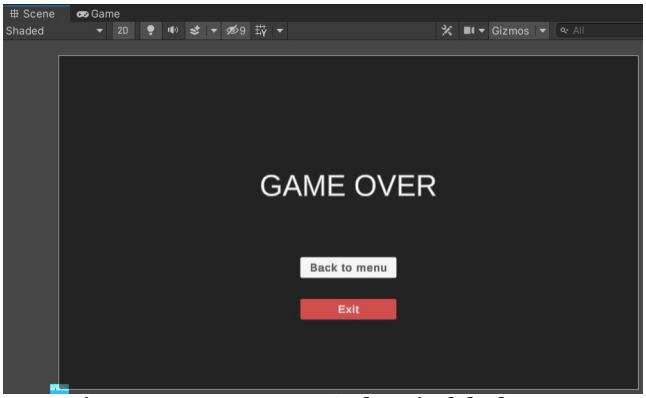

Figura 75 – Representação do painel de derrota.

2. Para cada botão, adicionar um "*Flow Machine*" e dentro dele, um novo macro para definir as funcionalidades;
3. Dirigir-se para o "*Flow Graph*" do botão "*Back to menu*" e introduzir o *node* "*On Button Click*";

4. Ligá-lo ao *node* "*Scene Manager Load Scene*" e colocar o nome de cena do menu principal ("*Menu*"). O botão "*Exit*" terá a mesma funcionalidade do outro botão com o mesmo nome, para sair do jogo;
5. Voltar para o *Unity Editor* e, na coluna do lado esquerdo, criar um elemento vazio "*Empty Object*";
6. Selecionar o "*Empty Object*" e, agora na coluna do lado direito, adicionar o componente de "*Variables*";

Todas as variáveis contidas neste objeto vão pretencer à lista de "*Scene Variables*".

7. Adicionar uma variável do tipo "*Game object*" com o nome de "*Death menu*". De seguida, arrastar o elemento referente ao painel de derrota para a caixa da variável "*Value*".

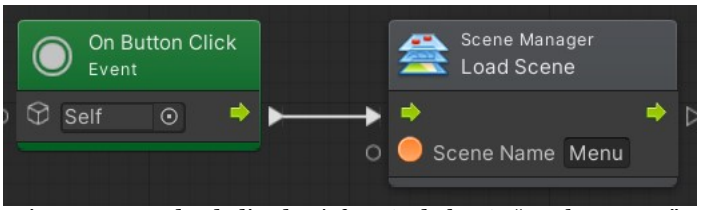

Figura 76 - *Nodes* dedicados à função do botão "*Back to menu*".

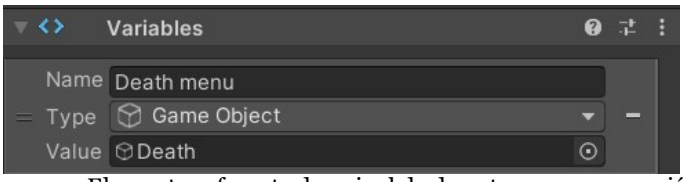

Figura 77 – Elemento referente do painel de derrota como uma variável.

7.9.3. Condição de derrota

1. Ir para o "*Flow Graph*" da personagem e adicionar a referência do comportamento que foi nomeado de "*Death*" ("*Costum Event*", com o nome "*Death*" escrito na zona inferior);
2. Ligá-lo ao *node* "*Game Object Set Active*" com a referência da variável "*Death menu*" ("*Get Variable*");
3. Conectar a referência da variável anterior ao *node* "*Game Object Get Active Self*" e ligar este ao *node* "*Negate*";

4. O *node* "*Negate*" vai se conectar com o *input* "*Value*" do *node* "*Game Object Set Active*";
5. Conectar o "*Game Object Set Active*" ao *node* "*Time Set Time Scale*" e alterar o seu valor para 0, para suspender o jogo;
6. Voltar para o "*Flow Graph*" do botão "*Back to menu*" do painel de derrota, para adicionar o último *node* ao "*Time Set Time Scale*" e alterar o seu valor para 1. Assim, quando se volta para o menu principal, o jogo deixa de estar suspenso.

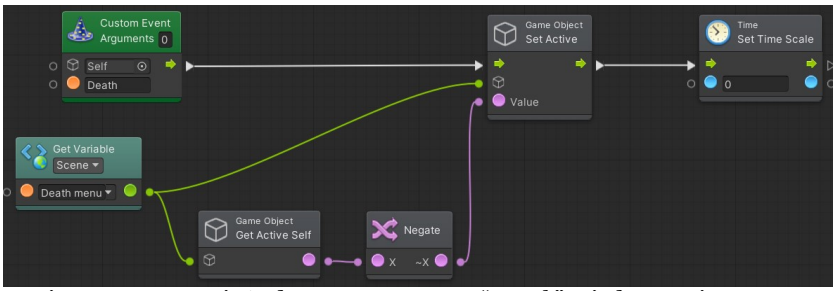

Figura 78 – Descrição do comportamento "*Death*" criado anteriormente.

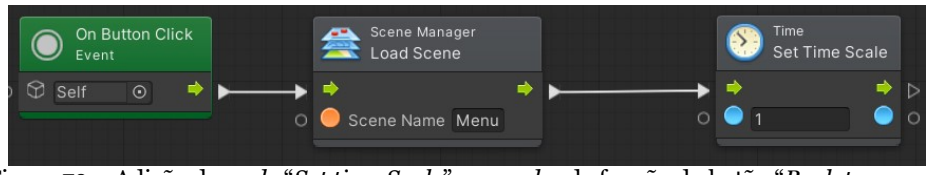

Figura 79 – Adição do *node* "*Set time Scale*" aos *nodes* de função do botão "*Back to menu*".

7.10. Ativação de coleção de pontos e condição de vitória

7.10.1. Elemento UI – Painel de Vitória

1. Criar um painel dedicado à vitória do jogador, usando os mesmos métodos apresentados anteriormente. Este painel será igual ao painel de derrota, com as mesmas funções nos respetivos botões.

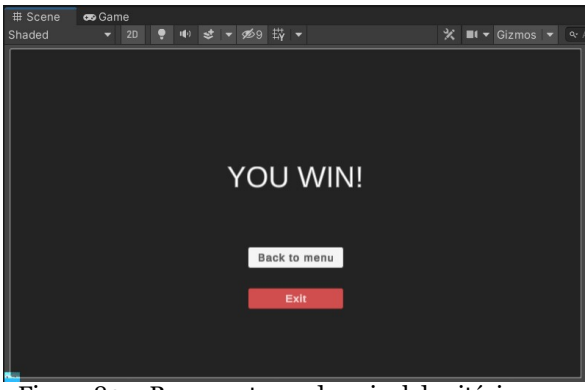

Figura 80 – Representaçao do painel de vitória.

7.10.2. Cogumelos

Para definir a funcionalidade de coleção de pontos, é necessário criar três variáveis diferentes. Uma delas estará presente no próprio elemento de *sprite* dos cogumelos, adicionada dentro de uma componente "*Variables*", cuja sua função é definir o valor de cada cogumelo, neste caso será 1. As restantes serão variáveis separadas que definem o número total de cogumelos que a personagem apanhou ao longo do nível, e o número requisito para alcançar a vitória.

A seguir, vai ser definido no código que no momento em que o número dos dois últimos valores for igual, irá aparecer o elemento UI referente ao final do nivel onde o jogador será felicitado.

1. No *Unity Editor*, arrastar os *sprites* dos cogumelos para o ambiente;
2. Adicionar o componente de "*Box Collider*" e ajustar a caixa de colisão;
3. Adicionar o componente de "*Variables*" e dentro desse elemento, criar uma variável do tipo *Interger* com o valor de 1. No meu caso foi-lhe dado o nome de "*CoinValue*";
4. Selecionar o "Empty Object" da coluna do lado esquerdo e adicionar mais duas variáveis, do tipo *Float*. A primeira terá o nome de "*CoinTotal*" e a segunda de "*CoinWin*". Os seus valores devem ser de 0 e 3, respetivamente;
5. No mesmo objeto, adicionar uma variável do tipo "*Game object*" com o nome de "*Win menu*". De seguida, arrastar o elemento referente ao painel de vitória para a caixa da variável "*Value*";
6. Selecionar o elemento do cogumelo e adicionar o componente de "*Flow Machine*" e criar um novo macro;
7. Dentro do "*Flow Machine*", colocar o *node* "*On Collision Enter 2D*" e ligá-lo à referência da personagem ("*Game Object Compare Tag*"), com o Tag "*Player*";

8. Ligar o "*Game Object Compare Tag*" a um "*Branch*", sendo os *nodes* anteriores a sua condição;
9. O *output* "*True*" deve ser conectado à referência da variável "*CoinTotal*" ("*Set Variable*");
10. Conectar o seu *input* ao *node* "*Add*", sendo o A e B conectados às referências das variáveis "*CoinTotal*" ("*Get Variable*") e "*CoinValue*" ("*Get Variable*"), respetivamente;
11. Ligar a referência da variável "*CoinTotal*" ("*Set Variable*") ao *node* "*Game Object Destroy*", com a referência "*Self*".

Deste modo, sempre que a personagem colidir com os *sprites* dos cogumelos, o valor da variável "*CoinValue*" vai ser somada com a variável "*CoinTotal*", oficializando um novo valor ao número de cogumelos obtidos.

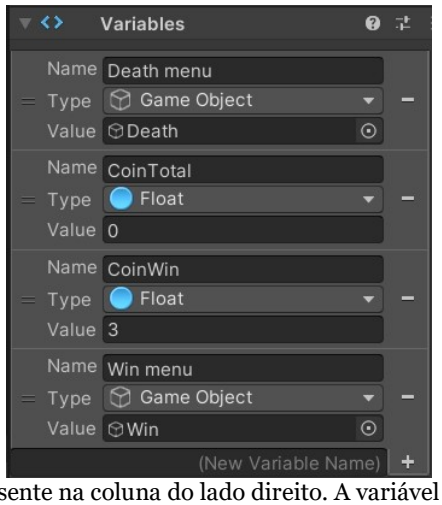

Figura 81 – Lista de variáveis, presente na coluna do lado direito. A variável "*CoinTotal*" representa o número total de cogumelos obtidos pela personagem, e a variável "*CoinWin*" é o número de cogumelos que é preciso alcansar para que o jogador ganhe o jogo.

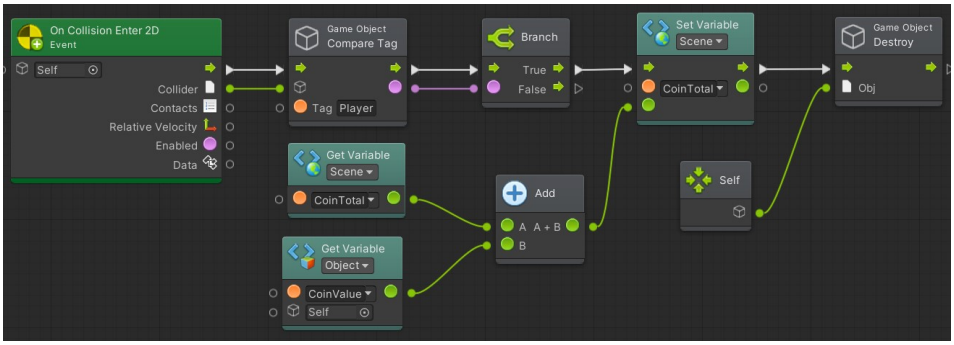

Figura 82 – *Nodes* de obtenção de pontos a partir da colisão entre a personagem e os cogumelos. Durante esse acontecimento, o valor do elemento do cogumelo (*CoinValue*) é adicionada ao número total de cogumelos obtidos pela personagem (*CoinTotal*).

7.10.3. Condição de vitória

1. Dirigir-se para o "*Flow Graph*" do macro principal da personagem e colocar o *node* "*Fixed Update*";
2. Ligá-lo a um "*Branch*". O *input* deste *node* deve ser conectado com o *node* "*Equal*", sendo o A e B conectados às referências das variáveis "*CoinTotal*" ("*Get Variable*") e "*CoinWin*" ("*Get Variable*"), respetivamente;
3. Conectar o "*Branch*" ao *node* "*Game Object Set Active*" com a referência da variável "*Win menu*" ("*Get Variable*");
4. Conectar a referência da variável anterior ao *node* "*Game Object Get Active Self*" e ligar este ao *node* "*Negate*";
5. O *node* "*Negate*" vai se conectar com o *input* "*Value*" do *node* "*Game Object Set Active*";
6. Ligar o "*Game Object Set Active*" ao *node* "*Time Set Time Scale*".

Com os passos anteriores, ordenámos a suspensão do jogo e a aparição do painel de vitória, no momento em que o valor da variável "*CoinTotal*" for igual ao valor da variável "*CoinWin*".

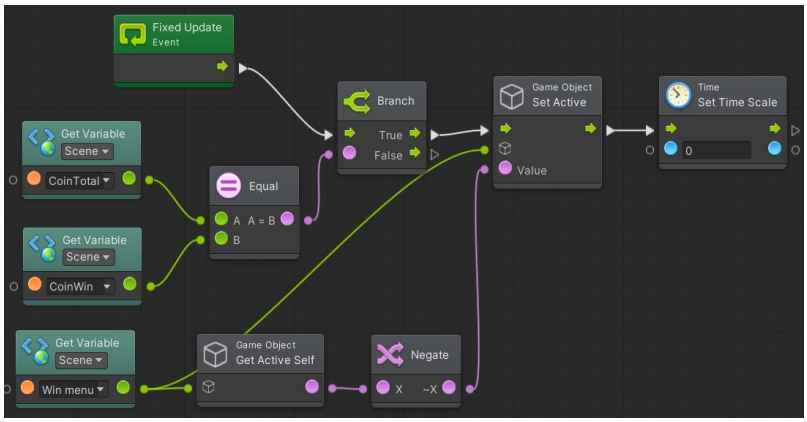

Figura 83 – *Nodes* de condição de vitória. No momento que o número total de cogumelos obtidos for igual a três, o jogador ganha o jogo.

7.10.4. Elemento UI – Contador de cogumelos obtidos

1. Criar um novo elemento de "Canvas" no *Unity Editor*, na coluna do lado esquerdo;
2. Dentro dele, colocar uma imagem referente ao *sprite* dos cogumelos;

3. Colocar um elemento de texto, com o número 0, indicando o número de cogumelos obtidos durante o nível;
4. Criar um "*Empty Object*" separado. Este elemento vai ter como objetivo controlar a contagem de cogumelos. No meu caso, este objeto foi nomeado "*CoinUI*";
5. Com o elemento anterior selecionado, adcionar um componente de "*Variables*", na coluna do lado direito;
6. No mesmo objeto, adicionar uma variável do tipo "*Text*" com o nome de "*Coin Text*". De seguida, arrastar o elemento referente ao texto criado anteriormente dentro do Canvas, para a caixa da variável "*Value*";
7. Selecionar o elemento "*Coin UI*", adicionar o componente de "*Flow Machine*" e criar um novo macro;
8. Dentro do "*Flow Machine*", colocar o *node* "*Update*";
9. Ligá-lo ao *node* "*Text Set Text*" e conectar o primeiro *input* à referência do objeto "*CoinText*" ("*Get Variable*");
10. Introduzir a referência da variável "*CoinTotal*" ("*Get Variable*") e conectá-lo com o *node* "*Inter To String*", que por sua vez deverá ser ligado ao segundo *input* do *node* "*Text Set Text*".

Assim, com estes passos, comunicamos ao sistema para apresentar o valor total de cogumelos obtidos na caixa de texto que criamos dentro do canvas.

Desta forma, após esta etapa, o projeto está completo, com todas as funcionalidades definidas para a jogabilidade deste jogo digital em ambos os programas, *Unity* e *Unreal Engine*. Durante o desenvolvimento do projeto, notei diferenças claras, tanto no design e organização de ferramentas de cada programa, como na construção de lógica e resolução de problemas.

O próximo capitulo dedica-se numa comparação direta entre o *Unity* e *Unreal Engine*, com base na minha própria experiência e pesquisa.

8. *Unity* vs *Unreal Engine*

8.1. Preço e público alvo

Antes de prosseguir a uma comparação mais técnica, irei constatar a dualidade mais obvia e supreficial entre *Unity* e *Unreal Engine*.

Os dois programas embora sejam gratuitos, oferecem umas politicas de preço alternativas. O *Unity* contém tem um pacote básico de utilização e disponibiliza a oportunidade ao desenvolvedor para comprar os restantes pacotes com preços variados, cada um com certas vantagens dependendo do preço. O *Unreal Engine*, por outro lado, é completamente gratuito, mas se por ventura o jogo codificado por este programa gerar um lucro a partir de 1 milhão de dolares, os seus criadores terão que pagar um valor de 5% do valor ganho pelo jogo em questão. Por esta razão, é evidente, pelo menos em termos de uma questão financeira, que o programa *Unreal Engine* vale mais a pena, pois oferece logo todas as ferramentas na sua totalidade desde o príncipio, sem a necessidade do desenvolvedor ter que pagar pela posse de mais vantagens durante a construção do seu jogo.

Primeiramente, durante a aprendizagem destes dois programas, tomei nota quase de imediato que foi mais fácil aprender as funcionalidades de *Unity* em contraste com o seu rival. Para além de apresentar uma interface mais intuitiva, o que também me levou a chegar a esta conclusão foi a existência de uma vasto número de tutoriais e cursos online, e da existência de uma maior comunidade de desenvolvedores dedicada a espalhar as suas ideias e conhecimentos para ajudar as pessoas que ingressaram nesta área recentemente. Por isso, o *Unity* sempre será uma escolha mais popular entre a comunidade de desenvolvimento independente de jogos.

Na minha opiniao, o *Unreal Engine* também tem uma interface não muito complexa para ser entendida, mas é um facto que, embora possua uma certa claridade na sua apresentação, chega a um ponto de parecer um programa transbordado de ferramentas, por oferecer uma vasto número de utensílios sem uma organização adequada para alguém que está apenas a aprender a usar o programa.

Com estes pontos, conseguimos chegar à conclusão que o *Unity* é mais apropriado para alguém novo na área de design de jogos. Porém, o Unreal é uma melhor opção para desenvolvedores experientes neste campo e oferece uma qualidade de ferramentas muito mais avançada que permitem que o jogo chegue mais facilmente a uma classificação AAA.

8.2. Diferenças de código e preparação da personagem

Notei que a abordagem não foi muito diferente na maior parte da construção de lógica do código nas duas plataformas. Porém, existem algumas particularidades que me motivaram a ter alguma preferência para um destes programas, tanto no código como noutros aspetos na preparação do jogo.

A primeira grande diferença surge na preparação da personagem. O *Unreal Engine* foi o primeiro programa que foi usado entre os dois. Durante a preparação das animações da personagem, achei intuitiva e fluida a construção de lógica para a implementação das animações à personagem. Pois apenas consiste em associar diretamente o "*flipbook*" correto para cada comportamento.

Mais tarde, quando cheguei ao momento de implementar as animações no *Unity*, notei que era necessário seguir mais alguns passos. Primeiramente, é preciso definir o ciclo correto das animações e as condições numa janela específica à parte do *Flow Graph*, para definir os requisitos que cada animação necessita para ser acionada. De seguida, as condições, que se apresentam como variáveis, necessitam de ser referenciadas dentro do código em cada um do seu respetivo comportamento de personagem, para que o sistema saiba que animação terá que ser acionada com base se as condições seguem os requisitos ou não. Após usar estas abordagens nos dois programas para o mesmo propósito, achei que a execução das animações no *Unreal* foi o método que fez mais sentido para mim, porque fui capaz de completar este desafio de uma forma mais direta, em poucos passos.

A segunda diferença que marcou a minha preferência está associada ao código e a constituição dos próprios *nodes* usados para a construção de lógica relacionada com o movimento da personagem. Para dar um exemplo, tomo referência à parte do código dedicada ao controlo de salto da personagem.

Quando o movimento da personagem foi testado no *Unity*, reparei que o controlo de salto poderia ser executado múltiplas vezes sem algum limite. Este comportamento apresentou ser um pouco invulgar, pois não passei pelo mesmo no programa *Unreal Engine*.

Após alguma averiguação dentro do programa *Unreal Engine*, tomei conhecimento que o *node* "*Jump*" é mais complexo do que aparenta. Quando é associada ao controlo da personagem, são apresentadas algumas definições customizáveis ao desenvolvedor, e dentro delas existe a opção de definir o número limite de saltos executáveis.

O *Unity* por sua vez, não possui um *node* tão complexo para esta funcionalidade. O que foi preciso fazer para que a personagem não tenha a possibilidade de saltar mais do que uma vez, foi a construção de uma linha invisível chamada "*Raycast*" para detetar se a

personagem está assente no chão ou não. Se o caso apresentar ser falso, o desenvolvedor terá que associar este facto para desativar o controlo de salto durante esse acontecimento.

Depois desta ocorrência e após uma comparação cuidada, concluí que o *workflow* de implementação de código é mais fluido no *Unreal Engine*, pelo uso um pouco mais reduzido de *nodes* em comparação ao *Unity*, pelo menos para construir a jogabilidade e os comportamentos para um jogo deste género. Também tomei nota que o sistema de *nodes* de *Unreal Engine* é mais poderoso, pois oferece as ferramentas para uma resolução de problemas mais eficiente, em contraste com o *Unity*. Os *nodes* em si também possuem nomes mais diretos à sua função, o que torna mais fácil perceber o papel destes elementos.

Irão sempre existir diferenças entre estes dois programas no resto da implementação de lógica dos comportamentos da personagem e nos acontecimentos dentro do nível, pelo simples facto de que são programas separados, com um design diferente, compostos por criadores diferentes. Por outro lado, partilham o mesmo raciocínio na resolução de problemas, na maior parte das vezes, apenas por *nodes* diversos. Mas, na minha perspetiva, como alguém com o mínimo conhecimento de programação de jogos, achei o sistema de programação visual do *Unreal Engine* mais intuitivo e eficaz no desenvolvimento de lógica.

9. Conclusões finais

Unreal Engine e *Unity* estão entre os líderes no que toca a sistemas de desenvolvimento de jogos digitais, mas tudo se resume ao próprio desenvolvedor. A escolha do programa correto depende muito do tipo de jogo que queremos criar, do público-alvo e da linguagem de programação de preferência.

Se o desenvolvedor é um iniciante que deseja publicar o seu primeiro título, o *Unity* pode ser uma opção atraente, pela sua estrutura "user-friendly". No entanto, se deseja ter acesso a uma ampla base de conhecimento e usar ferramentas mais poderosas, confiante que não se irá sentir esmagado pela disponibilização de uma vasta variedade delas, o *Unreal Engine* é mais adequado.

Mas a forma mais eficaz de testar uma solução é definitivamente experimentar cada um destes programas e chegar à sua própria conclusão. Como ambas as plataformas oferecem uma versão gratuita, qualquer um pode baixá-las da internet e testá-las.

Embora a programação visual seja uma opção mais facilitada para desenvolver código, este método acaba por ser um pouco mais lento, no desenvolvimento de projetos mais complexos e de maior dimensão. É fácil de imaginar a confusão que uma quantidade enorme de *nodes* e linhas podem gerar na tela. Se o jogo que foi desenvolvido anteriormente é capaz de desorientar só na parte de código do movimento de uma personagem simples, então como será que deve ser a organização do código de um jogo AAA? Provavelmente seria bastante intimidador. Já para não do facto de que é necessário usar um computador com suficientes recursos de memória para representar centenas elementos de uma só vez.

Definitivamente é um método mais simples, mas manipular e organizar os vários blocos numa tela acaba por consumir mais tempo do que escrever uma linha de código. Porém, para um iniciante, o uso de programação visual é muito mais vantajoso do que aprender a escrever código, nem que seja apenas um ponto de partida educativo, para mais logo transigir para os métodos tradicionais.

10. Bibliografia

Alexandrova, S.,Tatlock, Z. e Cakmak, M. (2015). RoboFlow: A Flow-based Visual Programming Language for Mobile Manipulation Tasks. Disponível em https://homes.cs.washington.edu/~mcakmak/pdfs/2015/alexandrova2015icra.pdf

Barfield, R. (2021, Março 30). Computer Programing a Brief History. Consultado em 2022, Dezembro 17 em https://www.bricsys.com/blog/computer-programing-a-brief-history

Bowell, A. (2020, Julho 22). Bolt visual scripting is now included in all Unity plans. Consultado em 2022, Setembro 19 em https://blog.unity.com/news/bolt-visual-scripting-is-now-included-in-all-unity-plans

Brodkin, J. (2013, Junho 3). How Unity3D Became a Game-Development Beast. Consultado em 2022, Março 12 em https://www.dice.com/career-advice/how-unity3d-become-a-game-development-beast

C Family Interview. (2022, Dezembro 6). The C Family of Languages: Interview with Dennis Ritchie, Bjarne Stroustrup, and James Gosling. Consultado em 2022, Dezembro 18 em http://www.gotw.ca/publications/c_family_interview.htm

Grant, A. (2022, Março 14). 8 Signs You Aren't Meant to Be a Programmer. Consultado em 2022, Março 11 em https://www.makeuseof.com/signs-you-arent-meant-to-be-a-programmer/

Heather. (2021, Janeiro 3). The History of the Sketchpad Computer Program – A Complete Guide. Consultado em 2022, Junho 15 em https://history-computer.com/sketchpad-guide/

Instadeq Team. (2022, Março 7). No-code History: GRAphical Input Language - GRAIL (1969). Disponível em https://instadeq.com/blog/posts/no-code-history-graphical-input-language-grail-1969/

Instadeq Team. (2022, Fevereiro 21).No-code History: Pygmalion (1975). Disponível em https://instadeq.com/blog/posts/no-code-history-pygmalion-1975/#introduction

Lestal, J. (2020, Agosto 5). History of programming languages. Consultado em 2022, Dezembro 17 em https://devskiller.com/history-of-programming-languages/

MACTECH. (s.d). Prograph CPX - A Tutorial. Consultado em 2022, Dezembro 18 em http://preserve.mactech.com/articles/mactech/Vol.10/10.11/PrographCPXTutorial/index.html

Matschinske, J. (2018, Maio 27). What the Hell Is Flow-Based Programming? Consultado em 2022, Setembro 27 em https://medium.com/bitspark/what-the-hell-is-flow-based-programming-d9e88a6a7265

Morrison, J. (2005, Abril). Patterns in Flow-Based Programming. Consultado em 2022, Setembro 19 em http://www.jpaulmorrison.com/fbp/morrison_2005.htm

Moss, E. (s.d). What Is Unreal Engine? Consultado em 2022, Março 11 em https://www.bairesdev.com/blog/what-is-unreal-engine/

Repenning, A. (2017, Julho). Moving Beyond Syntax: Lessons from 20 Years of Blocks Programing in AgentSheets. Disponível em http://ksiresearchorg.ipage.com/vlss/journal/VLSS2017/vlss17paper_10.pdf

Resnick, M., Maloney, J., Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J. e Silverman, B. (2009, Novembro). Scratch: Programming for All, 60-67.

Techapalokul e Tilevich. (2015). Programming Environments for Blocks Need First-Class Software Refactoring Support. Software Innovations Lab. Disponível em https://research.cs.vt.edu/quality4blocks/preprints/Techapalokul2015Programming.pdf

TechTarget Contributor. (2019 Julho). Visual Basic (VB). Consultado em 2022, Outubro 23 em https://devskiller.com/history-of-programming-languages/

WayBackMachine. (2005, Janeiro 3). Consultado em 2022, Março 14 em https://web.archive.org/web/20060913000000/http://en.wikipedia.org:80/wiki/Prograph

WayBackMachine. (s.d). RD GLOSSARY. Consultado em 2022, Dezembro 17 em https://web.archive.org/web/20070826224349/http://www.ittc.ku.edu/hybridthreads/glossary/index.php

Winograd, T. (1996). Bringing Design to Software. Disponível em https://hci.stanford.edu/publications/bds/10p-prototype.html

[s.n]. (2020, Maio 26). The History of Computing Podcast. Consultado em 2022, Dezembro 18 em https://thehistoryofcomputing.net/algol#

[s.n]. (2022, Fevereiro 30). What is COBOL? Consultado em 2022, Dezembro 18 em https://www.microfocus.com/en-us/what-is/cobol

[s.n]. (2022, Julho 6). What Is Visual Programming and How Does It Work? Consultado em 2022, Junho 10 em https://kissflow.com/low-code/visual-programming-overview/

[s.n]. (s.d). About Us. Consultado em 2022, Setembro 10 em https://appinventor.mit.edu/explore/about-us.html

[s.n]. (s.d). Introdução ao Blockly. Consultado em 2022, Setembro 10 em https://developers.google.com/blockly/guides/overview

[s.n]. (s.d). Introduction to Blueprints. Consultado em 2022, Setembro 19 em https://docs.unrealengine.com/5.0/en-US/introduction-to-blueprints-visual-scripting-in-unreal-engine/

[s.n]. (2022, Novembro 16). C++ Loops. Consultado em 2022, Dezembro 18 em https://www.geeksforgeeks.org/cpp-loops/

[s.n]. (2021, Agosto 7). What is Conditional Programming in Scratch? Consultado em 2022, Dezembro 18 em https://www.geeksforgeeks.org/what-is-conditional-programming-in-scratch/

[s.n]. (2022, Fevereiro). Number of smartphone subscriptions worldwide from 2016 to 2021, with forecasts from 2022 to 2027. Consultado em 2023, Março 7 em https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/