Projeto de tecnologias e programação de sistemas de informação (5425)

****

**Keep Moving**

Relatório Final

**Autores:**

Daniel Pinto

Wilson Ribeiro

Roberto Coutinho

13 de março de 2023

**RESUMOÍNDICE**

[1. INTRODUÇÃO 4](#_Toc129548185)

[2. PLANEAMENTO E METODOLOGIA 5](#_Toc129548186)

[2.1 - Planeamento 5](#_Toc129548187)

[2.2 - Metodologia 8](#_Toc129548188)

[3. IMPLEMENTAÇÃO 10](#_Toc129548189)

[3.1 – Software e recursos utilizados. 10](#_Toc129548190)

[3.2 – O que é o “Keep Moving”? 10](#_Toc129548191)

[3.3 – Características técnicas: 10](#_Toc129548192)

[3.4 – Organização do código fonte e recursos. 11](#_Toc129548193)

[3.5 – Pygame e Split Screen. 11](#_Toc129548194)

[3.5 – Saltar. 14](#_Toc129548195)

[3.6 – Colisões. 16](#_Toc129548196)

[3.7 – Fundo e chão infinitos. 16](#_Toc129548197)

[3.8 – AI 18](#_Toc129548198)

[4. RESULTADOS 22](#_Toc129548199)

[5. CONCLUSÕES 23](#_Toc129548200)

[BIBLIOGRAFIA 24](#_Toc129548201)

[ANEXO 1 – GAME DESIGN DOCUMENT 25](#_Toc129548202)

[ANEXO 2 – CÓDIGO FONTE 26](#_Toc129548203)

# 1. INTRODUÇÃO

O jogo “*Keep Moving*” foi criado em *Python*, utilizando a biblioteca *Pygame*, tendo como objetivo proporcionar aos jogadores uma experiência divertida e desafiadora, enquanto celebram o Ano Chinês do Coelho, ajudando o personagem principal, um coelho, a superar obstáculos. O jogo diferencia-se por, além do modo *single* *player*, comum nos jogos deste género, apresentar um modo de dois jogadores, permitindo que duas pessoas joguem juntas através da divisão da tela.

Neste relatório é descrito o processo de criação do jogo, incluindo o seu planeamento e metodologia, a implementação e os resultados alcançados. Descreve-se também, detalhadamente, as principais vitórias e desafios enfrentados durante o desenvolvimento do jogo.

# 2. PLANEAMENTO E METODOLOGIA

## 2.1 - Planeamento

O planeamento e a metodologia são fundamentais para o sucesso de qualquer projeto. Sabendo disto, na primeira reunião de equipa, após definidas as capacidades e objetivos de cada um dos integrantes para este projeto. Seguidamente foi consensual que o jogo a desenvolver seria definido, procurando um projeto que fosse de encontro aos objetivos propostos para a conclusão deste.

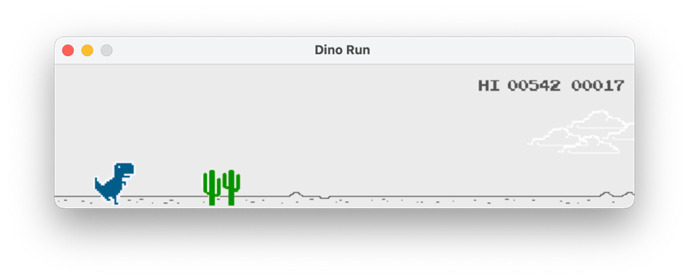
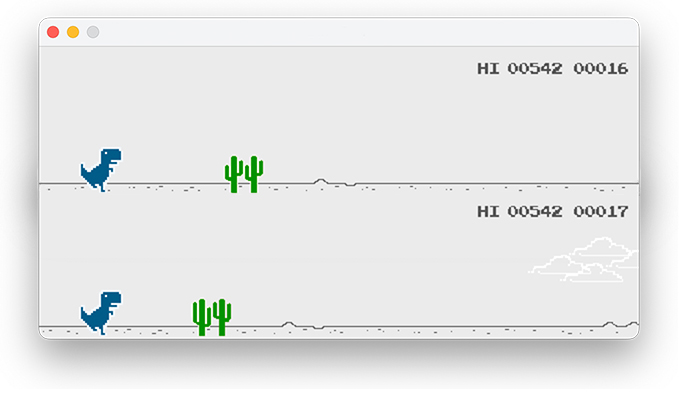
Uma vez que os conhecimentos na linguagem de programação necessários e obrigatórios para o desenvolvimento deste projeto ainda eram escassos ou nulos, e não tendo ainda noção das possibilidades máximas desta linguagem, a equipa partiu em busca de projetos *open source* que, à vista destreinada, parecessem simples e cumprissem os seguintes requisitos obrigatórios:

* O *software* deve de ter um módulo de entrar no jogo.
* O *software* deve de permitir o jogador iniciar um jogo.
* O *software* deve de permitir dois jogadores jogarem um contra o outro.
* O *software* deve de permitir o jogador jogar contra o computador (jogador-computador).
* O *software* deve de permitir o jogador escolher o nível do jogo (jogador-computador).
* O *software* deve de ter três níveis de dificuldade quando o jogo for (jogador-computador).
* O *software* deve de permitir o jogador escolher entre os três níveis de dificuldade, quando o jogo for (jogador-computador), no qual os níveis são: fácil, médio e difícil.
* O *software* deve de permitir o jogador sair no meio da partida contra o computador.
* O *software* deve de iniciar uma nova partida no fim da partida.
* O *software* deve de permitir o usuário encerrar a partida

Rapidamente surgiu a ideia de um “*Endless Runner*” (jogo de corrida interminável). Este tipo de jogo, pela simplicidade que apresenta, parecia ser de fácil adaptação a todos os requisitos à exceção de ser incompatível com *PvP* (Jogador contra Jogador). Com esta ideia já em mente decidimos usar, como inspiração, o jogo *“T-Rex Game”*, também denominado “*Chrome Dino”*, jogo este bastante popular por estar integrado no navegador Google Chrome. Após este passo, a equipa partiu em busca de vários projetos em *Python* que tivessem como objetivo a replica desse mesmo jogo e foi nesse momento que houve o primeiro contacto com a biblioteca *Pygame*. Esta biblioteca veio facilitar a criação do jogo, fornecendo as ferramentas essenciais para criar a interface gráfica do jogo, fundos, objetos, além do som e da música.

Estava planeado o projeto: Um jogo *“Endeless Runner*” ao estilo “*T-Rex Game*” desenvolvido em *Python* com recurso ao *Pygame*. Ainda sem saber bem qual seria o tema final do jogo, a equipa começou a testar vários dos projetos encontrados, a estudar a biblioteca *Pygame* e a definir, mesmo sabendo que seria provisório, o layout do jogo e menus deste (Figura 1).

Figura 1 - Menus, modo single payer e multiplayer



Com esta ideia já assente e maturada, foi decidido que, uma vez que o jogo teria, obrigatoriamente, três níveis de dificuldade, tanto a velocidade como a frequência de objetos seriam o fator diferenciador entre os vários níveis. No entanto, sendo este um jogo que tem, na sua essência, um aumento de velocidade por natureza, a diferenciação entre níveis viria na forma de incrementos maiores ou menores, consoante o nível de dificuldade.

Com isto, e porque desde o primeiro dia a equipa ambicionava um jogo extremamente simples de jogar, foi tomada a decisão de eliminar um dos comandos originais do jogo para simplificar a sua jogabilidade. No jogo original, o jogador pode baixar-se ou saltar para se desviar dos obstáculos. No “*Keep Moving*” foi eliminada a opção de baixar. Na ótica da equipa, a simplicidade de ser possível jogar apenas com uma tecla dá a liberdade ao jogador de focar a sua atenção em correr cada vez mais rápido e mais longe. A possibilidade de outra ação poderia criar uma frustração extra, coisa que, já pela natureza do jogo, existe. Com isto as teclas do jogo ficaram definidas: a tecla “w” que fará com que a personagem do jogador um salte, e a tecla “up” que fará o mesmo para o personagem do jogador dois.

Por fim, o tema do jogo. Este foi uma das decisões mais demoradas. A natureza do jogo não limitava a imaginação e, por isso, a escolha foi arrastada ao limite possível. Mas, por fim essa decisão foi tomada. Aproveitando o facto de que no dia 22 de janeiro de 2023 começou o ano do coelho no Calendário lunar chinês, data esta próxima da entrega do primeiro esboço deste relatório, foi escolhido o animal coelho como personagem principal do jogo, e que o jogo seria decorado com esse tema em homenagem a este acontecimento.

Com este ponto final, a equipa estava finalmente pronta para iniciar o processo de desenvolvimento do jogo.

## 2.2 - Metodologia

A nível de gestão do projeto a equipa optou por criar, gradualmente, pequenas partes do jogo, testando o código constantemente, com as alterações necessárias. Utilizando uma metodologia próxima das diretrizes das metodologias *Agile*, mesmo que, no caso deste projeto, não exista um cliente “real”, os elementos da equipa agiam como tal. O controlo de versões foi feito através do “*GitHub”*, permitindo a colaboração dos vários membros da equipa, mesmo em períodos fora do horário predefino para o desenvolvimento do projeto. Além disso, a equipa realizou testes e correções frequentes a cada implementação, garantindo que o jogo estivesse funcional e livre de *bugs* a cada versão.

A metodologia *Agile,* tal como foi mencionado*,* foi adaptada ao projeto de implementação do jogo "*Keep moving*" por meio de práticas como *Sprint Planning*, *Daily Stand-up* e *Sprint Review*.

O *Sprint Planning* foi realizado semanalmente para definir as atividades a serem realizadas, incluindo recursos necessários, funcionalidades a serem desenvolvidas e a divisão de tarefas entre os membros da equipa.

O *Daily Stand-up* foi realizado no início de todas as aulas disponíveis para o desenvolvimento do projeto. Este momento foi usado para discutir o progresso do trabalho e identificar quaisquer obstáculos que estivessem a impedir ou dificultar o desenvolvimento dele.

Durante a *Sprint Review*, foram abordadas soluções encontradas, a equipa discutiu os resultados obtidos e discutiu todas as alterações posteriores, agindo neste caso como *stakeholders* do seu próprio projeto.

Este planeamento e metodologia foram cruciais para o sucesso do projeto, garantindo que as etapas do processo de criação do jogo e os objetivos definidos fossem cumpridos, resultando num jogo funcional e divertido de jogar.

# 3. IMPLEMENTAÇÃO

Na Figura x apresenta-se a calendarização...

IMAGEM COM A Calanderização:

Porque expor a implementação deste projeto pode tornar-se confusa, serão expostas inicialmente várias informações relevantes para uma melhor compreensão e posteriormente será descrito o processo de desenvolvimento dos processos principais do jogo. Sempre que possível, os processos serão ilustrados através de diagramas, figuras ou descrições textuais. Todavia, o código pode ser consultado no Anexo 2.

## 3.1 – Software e recursos utilizados.

A implementação do jogo “*Keep Moving*” foi realizada na linguagem de programação *Python* na versão 3.10.6 e com recurso à biblioteca *Pygame* na versão 2.1.2. Foi também usado o “Visual Studio Code” como IDE de eleição entre a equipa.

## 3.2 – O que é o “Keep Moving”?

O jogo “*Keep Moving*” é um jogo 2D que se caracteriza como um jogo “*Endless Runner*”. Isto significa que não é possível vencer o jogo, apenas vencer o adversário, no casos multi-player. Para evoluir no jogo, o personagem tem de saltar obstáculos e continuará assim até que colida com um dos obstáculos, terminando o jogo. No modo “*2 Players*” este jogo continua com o mesmo comportamento sendo que, se um dos jogadores perder, o outro tem a possibilidade de continuar a jogar até perder também. No modo “Player VS CPU”, o ecrã inferior é o CPU sendo que o objetivo aqui é que o jogador chegue mais longe que o CPU.

## 3.3 – Características técnicas

Nesta secção serão apresentadas as características técnicas do jogo em análise:

* Uma janela com resolução de 1200x400 usada como janela principal. É nesta janela que serão expostos os menus do jogo bem como o modo “*Single Player”*;
* Divisão da janela de jogo verticalmente nos casos *Multiplayer*;
* Um coelho para cada jogador que se movimenta apenas verticalmente simulando um salto a cada vez que o jogador prima a tecla correspondente ao seu personagem;
* Uma superfície a simular o chão que se movimenta da direita para a esquerda infinitamente criando a ilusão de que o coelho está a correr;
* Obstáculos que aparecem pelo lado direito do ecrã e que, quando colidem com o personagem do jogo, fazem o jogo terminar;
* Um fundo infinito que se movimenta da direita para a esquerda infinitamente criando a ilusão de que o colhe está a correr;
* Desenvolvimento de uma AI que faça com que o jogo possa jogar sem intervenção humana.

## 3.4 – Organização do código fonte e recursos.

O código fonte deste jogo foi dividido em dois ficheiros. Um deles “***main.py”*** responsável por inicializar o jogo, carregar os recursos e iniciar o *loop* principal do jogo o outro, “***objects.py”*** que contêm as classes referentes aos objetos do jogo. Este ficheiro contém as seguintes classes:

* *Background* - Inicializa, atualiza e projeta o fundo do jogo;
* *Ground* – Inicializa, atualiza e projeta o chão do jogo;
* *Button* - Classe utilizada para todos os botões de seleção no jogo;
* *Rabbit* - Inicializa, atualiza e projeta os coelhos bem como o comportamento deles no jogo;
* *Obstacles* - Inicializa, atualiza e projeta o chão do jogo;

Existe também uma pasta com o nome “**Assests”** que contem todos os recursos adicionais ao código tais como imagens e sons.

## 3.5 – Pygame e Split Screen.

O *Pygame* é uma biblioteca de programação de jogos em *Python*, isto é, um conjunto de códigos pré-escritos e recursos que podem ser usados para ajudar os programadores a desenvolver software mais rapida e eficientemente.

Esta biblioteca é uma escolha popular porque esses módulos podem ser usados para lidar com gráficos, som, entradas do utilizador, eventos e outras funcionalidades necessárias para criar jogos ou aplicações. Alguns desses módulos incluem:

* **pygame.display** - Fornece funções para criar e gerenciar janelas e superfícies de exibição.
* **pygame.image** - Fornece funções para carregar, manipular e salvar imagens.
* **pygame.mixer** - Fornece funções para tocar e gerenciar áudio.
* **pygame.event** - Fornece funções para gerenciar eventos, como cliques do rato e teclas.
* **pygame.time** - Fornece funções para lidar com o tempo e atrasos no jogo.

Além disso, o *Pygame* também dispõe uma série de classes e métodos que podem ser usados para criar e manipular objetos de jogos, como *Sprites*, animações e colisões. Com todos estes recursos a escolha foi obvia, no entanto mesmo com toda esta documentação a equipa enfrentou o primeiro problema: *Split Screen*.

Criar uma janela inicial de jogo é algo relativamente simples, no entanto, não existe nenhum modulo, função ou método parametrizado para dividir a janela de jogo em dois ou mais de uma forma simples.

Para conceber uma janela no Pygame precisamos usar o modulo **pygame.display**. Por norma, cria-se uma variável com o nome ***screen*** e passamos o comprimento e largura em pixéis (px) da janela pretendida. Depois usamos o **pygame.display.update()** para que esta seja projetada no ecrã.

# Base para criar um jogo no Pygame

# 1- importar a biblioteca

import pygame

#2 - Definir o tamanho da janela

WIDTH= 1200

HEIGTH= 600

# 3 - Criar o display do jogo

screen = pygame.display.set\_mode((WIDTH,HEIGTH))

"""

4 - loop principal

Definimos a variavel run como true de forma a que o loop corra por defeito.

"""

run = True

while run:

# 5 cria uma forma de sair do loop, neste caso, ao clicar na janela para sair.

for event in pygame.event.get():

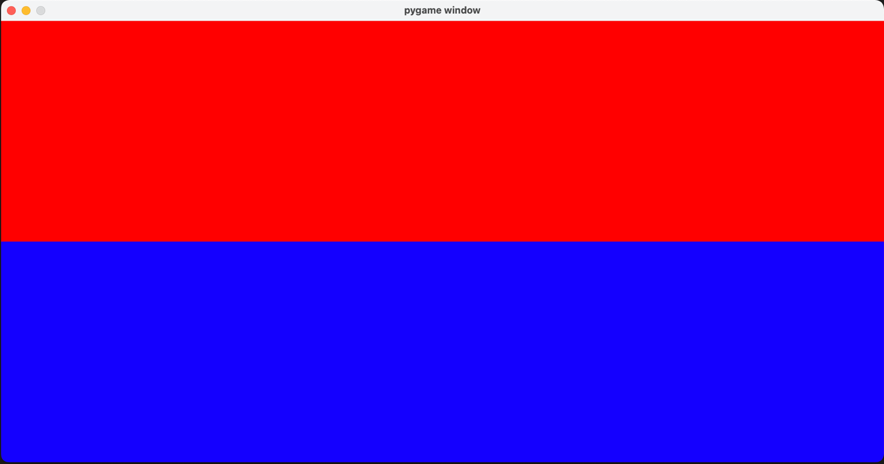
if event.type == pygame.QUIT:

run = False

pygame.display.update()

pygame.quit()

O *output* deste código é apenas uma janela a preto com as dimensões passadas, no entanto, o objetivo era que esta janela fosse dividida em duas superfícies independentes, como se esta fosse dois ecrãs, um por cima do outro. Algo como se exemplifica na seguinte imagem, sendo que o ecrã vermelho seria totalmente independente do azul:



Para isso, a solução foi usar o método **subsurface().** Este método é usado para representar uma parte de uma superfície maior. Assim sendo, criamos dois retângulos a que chamamos **cam\_1** e **cam\_2** (como se fossem duas câmaras) e definimos a posição destas exatamente uma em cima da outra. Após isso podem ser usadas cada uma dessas câmaras no método **subsurface()** e criar uma variável que represente cada parte do ecrã:

cam\_1 = pygame.Rect(0, 0, WIDTH, HEIGHT // 2)

cam\_2 = pygame.Rect(0, HEIGHT // 2, WIDTH, HEIGHT//2)

display\_1 = screen.subsurface(cam\_1)

display\_2 = screen.subsurface(cam\_2)

E com isto, o nosso problema inicial ficou resolvido. Com esta solução é possível usar as variáveis, display\_1 e **display\_2** (neste caso) para controlar as diferentes partes do ecrã independentemente.

## 3.5 – Saltar.

O processo de salto não foi um problema, existem imensos tutoriais e projetos *open source* que podem ser consultados. Optou-se por escolher um que fazia sentido para a equipa e que oferecesse o comportamento desejado ao longo de todo o jogo.

Assim sendo, o processo de salto é implementado dentro da classe **Rabbit**, mais especificamente na função **update()**. Quando o jogador pressiona a tecla "w", a variável “**jump”** passa o seu estado a **True**. Esta mudança do estado da variável altera a posição vertical do personagem, que então salta. A altura do salto é determinada pelo valor da variável "**self.vel**", que é decrementada gradualmente enquanto o personagem sobe e, em seguida, incrementada gradualmente enquanto o personagem cai.

A seguinte tabela explica melhor este comportamento:

|  |  |  |  |
| --- | --- | --- | --- |
| self.vel | posição da personagem | jump | is.jumping |
| 15 | 302 | False | False |
| 15 | 302 | False | False |
| -14 | **288** | **True** | **True** |
| -13 | 275 | False | True |
| -12 | 263 | False | True |
| -11 | 252 | False | True |
| -10 | 242 | False | True |
| -9 | 233 | False | True |
| -8 | 225 | False | True |
| -7 | 218 | False | True |
| -6 | 212 | False | True |
| -5 | 207 | False | True |
| -4 | 203 | False | True |
| -3 | 200 | False | True |
| -2 | 198 | False | True |
| -1 | 197 | False | True |
| 0 | 197 | False | True |
| 1 | 198 | False | True |
| 2 | 200 | False | True |
| 3 | 203 | False | True |
| 4 | 207 | False | True |
| 5 | 212 | False | True |
| 6 | 218 | False | True |
| 7 | 225 | False | True |
| 8 | 233 | False | True |
| 9 | 242 | False | True |
| 10 | 252 | False | True |
| 11 | 263 | False | True |
| 12 | 275 | False | True |
| 13 | 288 | False | True |
| 14 | 302 | False | False |
| 15 | 302 | False | False |
| 15 | 302 | False | False |

A segunda coluna da tabela demonstra a posição vertical do coelho. Esta posição é alterada quando o valor da posição anterior é somado ao valor da variável **“vel”**. Embora pareça que os dados estão invertidos, as coordenadas da posição dos objetos são passadas da esquerda para a direita e de cima para baixo, por exemplo: o canto superior esquerdo tem as coordenadas (0,0) enquanto o canto inferior direito tem as coordenadas (1200,400). Sendo assim, quando menor for o valor da coluna “posição do personagem”, mais alto o personagem é representado no ecrã. Além disso, a função "**draw()**" é chamada a cada *frame* do jogo, o que garante que a personagem seja “renderizada” com as mudanças de posição e altura. A função também verifica se o personagem está no chão ou no ar, e muda a imagem do personagem para representar a sua posição.

## 3.6 – Colisões.

As colisões são a base do “*Keep Moving*”. Esta é a única forma de perder o jogo assim sendo mereceu bastante atenção por parte da equipa. No entanto, é aqui que entra a grande vantagem de utilizar uma biblioteca como o *Pygame*. O *Pygame* já inclui uma função dedicada a este problema, que é a função **pygame.sprite.collide\_mask()**. Esta função é usada para detetar colisões entre *Sprites* com base nas suas máscaras de bits. Ela retorna **True** se houver uma colisão entre as máscaras de dois *Sprites* e **False** caso contrário. A máscara de bits é uma imagem a preto e branco que representa a forma de um *Sprite*.

Ao usar a função **pygame.sprite.collide\_mask()**, o *Pygame* compara as máscaras de bits de dois *Sprites* para determinar se há colisão. Isso permite que as colisões sejam detetadas com maior precisão do que usando o método mais comum que é um retângulo delimitador.

A equipa estava ciente de que o uso da máscara de bits tende a ser computacionalmente mais intensivo, no entanto no jogo em causa, não existe uma quantidade significativa de *Sprites* em cena que seja relevante para esta condição.

## 3.7 – Fundo e chão infinitos.

Tendo o jogo “Keep Moving” como princípio o conceito de jogo sem fim (“Endless Runner”), era necessário que o cenário do jogo também fosse virtualmente sem fim, ou que pelo menos criasse esse efeito.

O fundo do jogo e o chão são os elementos que criam a ilusão de movimento e velocidade do jogo, assim sendo, este ponto mereceu bastante atenção por parte da equipa. Para que esta ilusão fosse criada, o chão e o fundo do jogo precisavam de movimentar-se da direita para a esquerda. Este movimento é criado, atualizando a posição das imagens subtraindo a velocidade desejada (em *frames*) à posição no eixo do x.

Numa abordagem superficial ao problema poderíamos pensar em apenas uma imagem grande o suficiente que para que, durante o jogo, ela nunca terminasse, no entanto não seria infinito e não foi a solução adotada. A equipa sabia que a solução seria a escolha de uma imagem que, quando impressa lado a lado com uma imagem igual, criasse a ilusão de continuidade. No entanto, numa primeira implementação, devido aos incrementos de velocidade, esta solução mostrou-se fraca e pouco consistente. Desta a equipa foi em busca da solução seguinte e final que se resumiu em:

1. Criar uma variável que verificasse a largura da imagem;
2. Dividir a largura do ecrã por essa variável e somar 1. Esta soma de mais uma unidade garante que não existe nenhum momento em que o ecrã fique sem fundo.
3. Imprimir as imagens de forma sequencial, usando a sua posição horizontal e criando a ilusão de movimento subtraindo uma variável (variável **scroll** - que é incrementada a cada *frame*) a essa posição;
4. Garantir que, no momento em que a primeira imagem está fora dos limites do ecrã, a variável **scroll** é inicializada novamente.

A imagem abaixo (Figura x), ilustra o comportamento descrito anteriormente. Ao usar este método, garantimos que o jogo apenas carrega a quantidade mínima necessária de imagens para que o fundo e chão criem a ilusão pretendida.

Uma imagem com ecrã, escuro, plano

Descrição gerada automaticamente

## 3.8 – Artificial intelligence (AI)

Neste projeto foram introduzidas três diferentes instâncias da *Artificial intelligence* (AI), uma para cada uma das dificuldades definidas, nomeadamente Fácil, Médio e Difícil. É importante saber qual as variáveis associadas à AI, para melhor compreender como funciona.

            dx = obstacles2.rect.x - rabbit\_2.rect.x

            fudge = (int(75+(counter\_player\_2/20)))

            if dx <= (fudge):

jump2 = True

O código acima atribui a “dx” o resultado da subtração do valor da posição do obstáculo (“obstacles2.rect.x”) no eixo horizontal (0 sendo o limite esquerdo da janela do jogo) e a posição do coelho no mesmo eixo (valor fixo de 50).

De seguida vemos uma das atribuições de “fudge”, onde utilizamos o valor para limitar a distância máxima onde o obstáculo terá de estar para o coelho saltar com as linhas de código: “if dx <= (fudge): jump2 = True”. Na atribuição acima mencionada, o 75 indica a distância base para o salto (tendo em conta que a variável “counter\_player\_2” começa com o valor de 0).

Segue um exemplo de valores de “dx” em uma instância do jogo enquanto temos um obstáculo a 750px do limite esquerdo da janela:

            dx = obstacles2.rect.x - rabbit\_2.rect.x

dx = 750 - 50

dx = 700

A variável “dx” será sempre o valor da distância em pixéis entre o coelho e os obstáculos. Agora precisamos de algo que faça com que o coelho salte quando o obstáculo está na posição certa, e para isso serve “fudge”, que é o int (número inteiro) da soma de um valor inteiro com o valor da variável “counter\_player\_2” dividido por um valor inteiro.

            fudge = (int(75+(counter\_player\_2/20)))

A variável “counter\_player\_2” é utilizada para calcular a pontuação do Jogador 2, logo, com o decorrer do jogo assume o valor de 5x a pontuação atual, após ser dividido para assumir valores menores, a variável de “fudge” assume esse valor dividido mais 75, que representa o valor inicial de “fudge”. É importante fazer este cálculo para atribuir um valor dinâmico que seja incrementalmente maior para que conforme a velocidade dos objetos aumente, a distância em que o coelho salta aumente também.

            if dx <= (fudge):

            jump2 = True

Finalmente, se a afirmação “dx <= (fudge)” for verdadeira, o coelho salta “jump2 = True” e com este método, e com alguns ajustes na variável “fudge” conseguimos uma AI que tinha potencial para nunca perder, agora restava ajustar as AI’s às dificuldades e as fazer perder na pontuação correspondente.

A ideia que surgiu de imediato foi, em vez de atribuir um valor muito baixo a “fudge” e fazer com que fosse um valor aleatório e não fixo, para a AI perder dentro de intervalos de pontuação, e não em uma pontuação exata.

for obstacles2 in obstacles2\_group:

    match difficulty:

        case 1:  # Difficulty: EASY

            if AI:

                dx = obstacles2.rect.x - rabbit\_2.rect.x

                fudge = (int(70+(counter\_player\_2/80)))

                if score\_player\_2 >= 415: # / if fudge >= 96:

                  fudge = int(random.uniform(55, 100))

                    if score\_player\_2 >= 500: #TO LOSE (505 MAX SCORE)

                        fudge = 50

                if dx <= (fudge):

                    jump2 = True

Nesta AI apenas temos 2 condições para além das que já vimos acima, uma que indica que quando a pontuação do jogador 2 for acima do 415, “fudge” (atualmente = 96) passa a ser um numero aleatório entre 55 e 100, como o valor de “random.uniform(55, 100)” é um *float* (número com casas decimais) é necessário preceder com “int”.

                if score\_player\_2 >= 415:

                  fudge = int(random.uniform(55, 100))

Essa condição faz com que após “score\_player\_2 >= 415”, existe uma chance de o coelho falhar o salto, mas não é certo que aconteça, até a próxima condição que é:

                    if score\_player\_2 >= 500:

                        fudge = 50

Esta condição nem sempre será verificada porque a probabilidade dita que por esta altura o coelho já terá perdido, mas há uma pequena possibilidade de não perder porque o programa seleciona os obstáculos de uma forma aleatória mas não são todos do mesmo tamanho, logo se acontecer de serem gerados os mais pequenos em sucessão enquanto o valor de “random.uniform(55, 100)” está muito próximo de 100 na altura correta, o coelho continuará, mas esta condição resolve esse problema, atribuímos o valor de 50 a “fudge” que faz o coelho saltar muito próximo do obstáculo seguinte, mesmo que seja o obstáculo mais pequeno.

Não existe nenhuma alteração na lógica na dificuldade média, como pode ser confirmado a seguir:

case 2: # Difficulty: MEDIUM

    if AI:

        dx = obstacles2.rect.x - rabbit\_2.rect.x

        fudge = (int(60+(counter\_player\_2/10)))

        if score\_player\_2 >= 670: #670 SCORE

            fudge = int(random.uniform(110, 130))

            if score\_player\_2 >= 760: #TO LOSE (754 MAX SCORE)

                fudge = 100

        if dx <= (fudge):

            jump2 = True

A dificuldade difícil tem uma grande diferença, só tem uma condição, eliminando a segunda que faz com que o coelho perca tendo em conta termos eliminado a chance de não o fazer da seguinte forma:

case 3: # Difficulty: HARD

if AI:

dx = obstacles2.rect.x - rabbit\_2.rect.x

fudge = (int(75+(counter\_player\_2/20)))

if score\_player\_2 >= 820: #fudge 280

fudge = int(random.uniform(130, 175)) #1108 MAX SCORE (LUCKY)

jump2 = True

Para uma melhor experiência com resultados mais variados após os 820 de pontuação a AI tem sempre uma chance de perder. Não foi necessária uma condição adicional porque após testes verificamos que se “fudge = 175” mesmo com apenas os obstáculos mais pequenos, a pontuação máxima é de 1108 pontos (Teste efetuado +1000 vezes).

A dificuldade de encontrar um número que atinga uma certa pontuação nas outras dificuldades está centrada na velocidade menor, a variável “fudge” com um valor fixo não é muito consistente para perder garantidamente em velocidades baixas, tendo em conta não ser necessário um número tão grande, e para evitar qualquer tipo de desvio do nosso objetivo adicionamos a condição que obriga o fim da AI.

Foram implementadas outras funcionalidades complementares, tais como:

* Animações dos personagens;
* Menus;
* Ecrã inicial;
* Modo de “*Single Player*”;
* Contagem de pontos e exibição da pontuação atual;
* Efeitos sonoros e música de fundo.

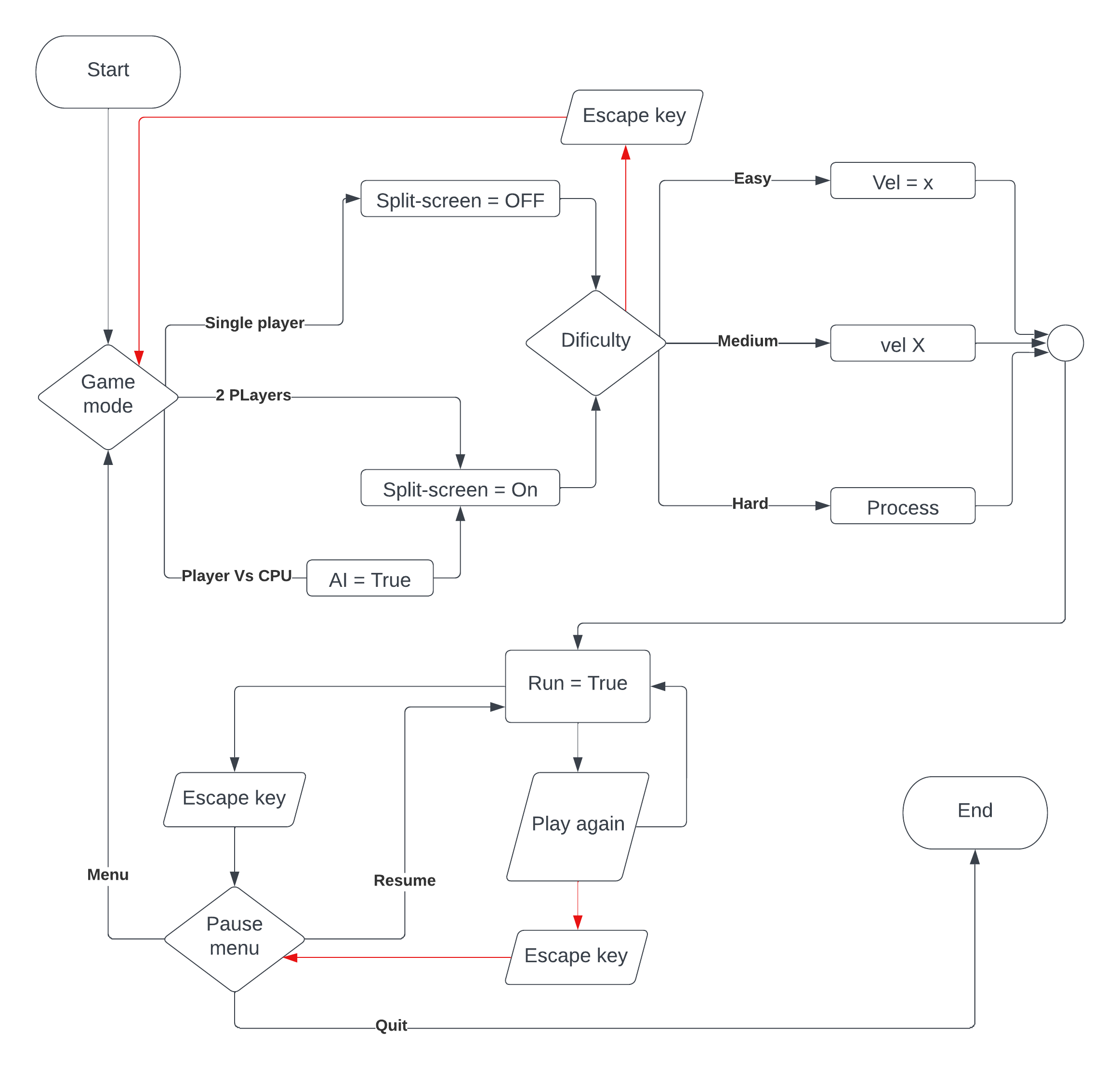
No entanto, e tendo em atenção a extensão deste relatório, foram destacados os maiores desafios enfrentados pelo grupo.

Todas as outras implementações podem ser consultadas no código fonte no Anexo 2.

A equipa fez questão de comentar e organizar todo o código da melhor forma possível para que fosse de fácil compreensão para os interessados.

# 4. RESULTADOS

Os resultados obtidos neste projeto superaram as espectativas iniciais e todos os requisitos iniciais foram cumpridos. O jogo “Keep moving”



- Imagens do jogo

- diagrama funcional do jogo

- descrição das diferenças entre os níveis de dificuldade

# 5. CONCLUSÕES

A equipa acredita que os recursos utilizados na implementação do jogo foram adequados para o desenvolvimento de um jogo que se destaca pela sua simplicidade sem perder o seu lado lúdico e desafiador. A escolha da linguagem de programação *Python* e da biblioteca *Pygame* foram acertadas, pois essas ferramentas foram suficientes para implementar todas as funcionalidades necessárias para o jogo.

Os resultados obtidos com a implementação do jogo "*Keep Moving*" foram positivos. O jogo foi testado por diferentes jogadores, com diferentes níveis de habilidade, e todos eles demonstraram interesse e satisfação no que diz respeito à jogabilidade e estética do jogo. O feedback dos jogadores, foram observados através de comentários positivos sobre a jogabilidade para um ou dois jogadores, a animação dos personagens, e a música de fundo.

Com base nos resultados obtidos, podemos concluir que o jogo "*Keep Moving*" foi implementado com sucesso, atendeu ao desafio proposto e cumpriu e os objetivos estabelecidos na fase do projeto.

# BIBLIOGRAFIA

[Relacionar, em tópicos e em ordem alfabética, as principais fontes de consulta empregadas para estudo e desenvolvimento do projeto. A bibliografia deve ser apresentada na forma de lista numerada, contendo as seguintes informações mínimas por item, na seguinte ordem: nomes dos autores (pessoa física ou instituição), título da fonte (livros, publicações, imagens, documentos em meio eletrônico, etc) entre aspas, data da publicação ou, no caso de meio eletrônico, data de acesso e endereço (*link).*]

GOOOOOOOGLE!!!!!

CHATGPT!!!!

ROUUUUUUBAR CODIGO!!!

YEEAAHH!!

Anda môzinho.. está quase

# ANEXO 1 – GAME DESIGN DOCUMENT

# ANEXO 2 – CÓDIGO FONTE