UNIVERSIDADE D
COIMBRA

Rui Pedro Costa Linhares

PUMA GAME PLATFORM
BROWSER/EDITOR

VOLUME 1

Dissertação no âmbito do Mestrado em Engenharia Informática com especialização em Engenharia de Software orientada pelo Professor Doutor Licínio Gomes Roque e apresentada ao Departamento de Engenharia Informática na Faculdade de Ciências e Tecnologia Universidade de Coimbra.

Setembro de 2020

Resumo

A atividade de *gaming* é uma das principais formas de entretenimento, principalmente pelo número de jogadores, e por estar em constante evolução. A produção de videojogos é uma atividade relevante na sociedade contemporânea, cultural e economicamente, enquanto forma de criação. Para alimentar este fenómeno torna-se essencial desenvolver alternativas que facilitem o processo de criação e produção de jogos originais.

Uma *game engine* é uma plataforma de software que permite o desenvolvimento de um videojogo, processo complexo e tecnicamente exigente. A utilização deste tipo de solução, embora acelere o processo em relação a começar do zero, ainda requer frequentemente uma longa adaptação e um longo tempo de produção, tornando difícil e caro criar e ensaiar conceitos inovadores. Com este trabalho procurou-se conceber uma *game engine* minimalista para a prototipagem rápida de jogos, que permitisse a experimentação em "*game design*" de uma forma simplificada.

Buscamos inspiração no fenómeno de *modding*, no qual os jogadores se apropriam dos elementos de um jogo existente, para o transformarem, e estudamos algumas *game engines* mais utilizadas. Com alguns ensaios de criação de jogos clássicos, identificamos um conjunto de conceitos base para propor uma arquitetura que acelere a criação de jogos. Com base neste modelo contribui-se uma linguagem de markup para definição de jogos. Validou-se a contribuição uma prova de conceito que demonstra a codificação de um jogo com a linguagem de markup, passível de ser interpretada de forma automática. A refleção final incidiu sobre a possibilidade desta linguagem oferecer uma alternativa para expandir o acesso ao desenvolvimento de jogos, com novos criadores, pela facilidade de aprendizagem e de utilização.

Palavras-Chave— *Motor de Jogos*, *Gaming*, *Game Design*, *Modding*, *Linguagem de Marcação*.

Abstract

The gaming activity is one of the main forms of entertainment, mainly due to the number of players, and because it is constantly evolving. Video game production is a relevant activity in contemporary society, culturally and economically, as a form of creation. To fuel this phenomenon, it is essential to develop alternatives that facilitate the process of creating and producing original games.

A game engine is a software platform that allows the development of a video game, a complex and technically demanding process. The use of this type of solution, although speeding up the process compared to starting from scratch, still often requires a long adaptation and a long production time, making it difficult and expensive to create and test innovative concepts. With this work we aimed to design a minimalist game engine for the rapid prototyping of games, which would allow experimentation in "game design" in a simplified way.

We looked for inspiration in the modding phenomenon, in which players take over the elements of an existing game, to transform it, and we studied some of the most used game engines. With some case studies on creating classic games, we identified a set of basic concepts to propose an architecture that accelerates the creation of games. Based on this model, a markup language is used to define games. The contribution was validated as a proof of concept that demonstrates the codification of a game using the markup language, which can be interpreted automatically. We ended by reflecting on the possibility of this language offering an alternative to expand access to game development, with new creators, due to the ease of learning and use.

Keywords— *Game Engine*, *Gaming*, *Game Design*, *Modding*, *Markup Language*.

Conteúdo

Glossário

Game Design Solução do problema de criatividade do que os jogos devem ser.

Game Engine Software que fornece um conjunto de funcionalidades ao criadores de jogos para simplificar o processo de desenvolvimento de um jogo.

Gaming Termo desenvolvido para a atividade de jogar jogos eletrónicos.

Mod Abreviatura de *Modding*, "modificação".

Modding Termo usado para designar uma alteração num jogo.

Protótipo Versão inacabada de um jogo para ser testado e/ou modificado.

Sandbox Termo que representa um jogo, na qual os jogadores tem a liberdade de fazerem o que querem dentro do mesmo.

Token Códigos gerados que auxiliam na troca de mensagens.

Siglas

API *Application Programming Interface.*

DEI Depratamento de Engenharia Informárica.

ES Engenharia de Software.

GPU *Graphics Processing Unit.*

GUI *Graphical User Interface.*

HTLM *HyperText Markup Language.*

IA Inteligência Artificial.

JPEG *Joint Photographic Experts Group.*

PNG *Portable Network Graphics.*

PUMA *PUMA Game Plataform.*

UC Universidade de Coimbra.

XML *Extensible Markup Language.*

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Neste capítulo é apresentado o enquadramento do projeto, em que contexto se aplica e relevância que terá para a comunidade envolvida, tal como os principais objetivos a cumprir neste projeto. Também é apresentado todo o estudo dos métodos possíveis para chegar ao resultado final e uma breve apresentação da estrutura do documento.

1.1 Enquadramento

Desde há muito tempo que o setor dos *videojogos* tem vindo sempre a crescer, nomeadamente, tornando-se uma das maiores formas de entretenimento no mundo. Podemos dizer que a actividade de *gaming* já é uma parte bastante importante na vida de muitas pessoas, quer seja a jogar ou a assistir a outras pessoas a jogar (*streaming*) e de forma mais restrita, envolvidas na criação de videojogos.

É nesta última actividade que entra a ideia deste projeto. Focado na criação de jogos, o objectivo é o desenvolvimento de uma *game engine*. Uma *game engine*, basicamente, permite a criação de um jogo do zero de um modo muito mais simples ou com enconomia de esforço, e de certa forma possibilita, hoje em dia, que uma só pessoa consiga criar um jogo com tecnologia actual, o que de outra forma seria muito difícil. De forma abrangente tudo o que seja movimentos, personagens, obstáculos, cenários, efeitos sonoros, texturas, inteligência artificial, física e efeitos 2D/3D, pode ser encontrado numa *game engine* actual, para integrar num jogo funcional, para as diversas plataformas (Consolas, PCs, Smartphones, etc). [22]

O problema das *engines* actuais, é o facto de serem muito generalistas (e pesadas) e complicarem o processo de criação de jogos, pois oferecem demasiadas opções e exigem muitas horas de aprendizagem e produção até se conseguir experimentar as ideias num protótipo. Melhor dizendo, muitas destas *engines* trouxeram uma falsa ilusão de "facilidade" na criação de jogos, já que, uma pessoa leva muito tempo a aprender a utilizar um motor destes para criar o jogo que idealiza.

1.2 Objetivos

O desafio é desenvolver uma *game engine* minimalista, isto é, um *software* simples capaz de facultar aos criadores de jogos um conjunto de ferramentas necessárias para a prototipagem rápida de um jogo. Este objectivo entende-se face ao propósito de acelerar o processo de criação e experimentação de *game design*. Esse *software*, significa, na verdade, um "regredir para simplificar"das *Game Engines* existentes, de modo, a identificar e explorar os mecanismos essenciais que se tem vindo a desfocar com o tempo e, claro, também aproveitar, com essas soluções conceptuais, as melhorias e inovações que trouxeram.

Sendo assim, o desafio divide-se em vários segmentos que representam cada recurso (cenário, personagens e ações, simulação de física, efeitos sonoros, *interface*, etc.) que a *Game Engine* irá ter, de forma que explorando uma composição desses recursos torne o processo de criação de jogos mais simples e rápido. A identificação e modelação destes recursos torna-se uma componente muito importante no projeto, pois uma das missões será averiguar os recursos ou funcionalidades que são utilizadas nas engines atuais ou no *modding* de forma recorrente. Com base nessas referências e na experimentação com prototipagem de jogos ao longo do projeto, identificam-se os recursos mais necessários. Sendo que esses recursos podem ser encontrados em bibliotecas ou *Application Programming Interface* (API)s, também será uma tarefa definir um conjunto de bibliotecas ou APIs a reutilizar, tecnologias que já possuem muitas das ferramentas desejadas.

Em suma, o objetivo centra-se no estudo de um motor de jogo que possa vir a ser utilizado para a construção de jogos 2D adotando uma *interface* de criação que permita ao game designer montar cenários e expressar comportamentos de objetos e personagens, de forma rápida e eficiente. Para tal, pretende-se ainda ensaiar um ambiente de modelação do ambiente do jogo, com o uso de *Petri Nets* para a definição de comportamentos no jogo.

1.3 Relevância

Para falar da importância do desenvolvimento de um novo Game Engine, é preciso falar primeiro da importância que o desenvolvimento de jogos tem para uma das maiores comunidades do mundo.

O mercado de *gaming* foi objeto de um crescimento à escala global, crescimento que tem tendência a aumentar ano após ano. O design de jogos já não trata apenas de criar brinquedos, mas está na base de uma diversidade de áreas, desde o desenvolvimento, à operação de plataforma de jogo e de *streaming*, aos sociais casinos e ao jogo profissionalizado, movimentando biliões de euros. O gaming já atingiu um nível gigantesco que em diversas regiões já considerado um desporto, tanto a nível competitivo como a nível do entretenimento. [22]

Por detrás de um jogo está uma ideia ou uma história capaz cativar o jogador a querer experimentar, saber mais, ambicionar bater recordes, completar a aventura, entre outras coisas que o faça ficar envolvido/comprometido com jogo. Este cenário origina uma constante sede de novidade e uma oportunidade para produção cultural que dificilmente se concretiza tendo em conta a barreira técnica à entrada para novos criadores.

Neste contexto e olhando para as pessoas que gostariam de criar jogos, considerando que muitas delas não são tecnicamente capacitadas em computação, surge a ideia de criar um *software* mais acessível para expressarem as suas ideias e montarem rapidamente um protótipo do seu *game design*.

1.4 Síntese do problema

Este trabalho procura colmatar algumas dificuldades com as *Game Engines* mais populares e deve poder ser usado por qualquer um devido à sua simplicidade. Essas adversidades têm a ver com o facto do processo de criação de um jogo continuar a ser tecnicamente muito complexo, não permitindo desenvolver um protótigo de um *game design* em pouco tempo, sem considerável experiência prévia. Isto é, obriga aos utilizadores a um tempo exagerado de adaptação dos motores de jogo para conseguirem montar um jogo com o conceito e fluidez desejadas para experimentação.

Neste contexto, digamos que existe um foco em dois grupos de criadores, os que sabem programação e os que não sabem, e assim teremos em consideração que os dois tipos de construção de jogos sejam uma possibilidade. Neste duas possibilidades, é preciso ter em conta que quem sabe programação terá mais margem de manobra, isto é, poderá definir ou acrescentar funcionalidades extra, para poderem adicionar variedade ou detalhes ao jogo em si.

Além de acelerar a prototipagem, também é necessário proporcionar aos utilizadores uma boa experiência na criação dos seus jogos. Por isso, além de oferecer simplicidade e rapidez ao desenvolvimento, seria benéfico disponibilizar uma *engine multiplayer* (um ambiente partilhado por dois ou mais utilizadores) e permitindo assim a colaboração na prototipagem. Para que os utilizadores possam criar jogos sozinhos ou em conjunto, usar a plataforma nas suas máquinas de trabalho e lançar o jogo para diversas plataformas, será útil permitir a definição do jogo numa forma partilhável e independente do código que interpreta essa definição, tal como acontece na web, com o html e o browser.

Ainda assim, o foco principal do trabalho está na simplicidade e na velocidade que será possível obter para construir um protótipo de jogo capaz de corresponder às idealizações do *game designer*.

1.5 Estrutura do Documento

O documento está divido nos seguintes capítulos:

No capítulo dois exploramos o estado da arte, identificando conceitos e práticas em game design, analisamos algumas engines e bibliotecas e mapeamos conceitos e recursos de modding em videojogos.

No capítulo três detalhamos objectivos e a forma como os procurámos alcançar.

No capítulo quarto define-se uma primeira proposta de design, definindo conceitos e uma arquitectura de recursos úteis na prototipagem de videojogos.

No capítulo cinco elabora-se uma prova de conceito de forma a testar a viabilidade e utilidade dos conceitos na arquiectura definida.

No capítulo seis desenvolve-se uma linguagem de markup para descrição de jogos com base nos conceitos identificados e desenvolve-se uma prova de conceito para demonstrar a sua viabilidade na definição de um simples jogo de género de aventura gráfica.

No capítulo sete e último, termina-se com algumas conclusões, reflexões e sugestões de trabalho futuro.

Capítulo 2

Estado da Arte

Neste capítulo é apresentado o estado de arte, onde são exploradas as temáticas mais relevantes, relativamente ao trabalho. O primeiro tema é o de Conceitos, cujo foco é contextualizar sobre o projecto de jogos, este tema também se estende a assuntos sobre o mundo dos jogos, além de ser referido o jogo Minecraft que servirá de exemplo para estudar o fenómeno de *modding* ou criação amadora neste trabalho. Os temas seguintes são sobre *game engine*, motores de construção e criação de jogos, sendo que num é apresentado a conceção e no outro algumas das soluções de prototipagem existentes. No fim será feita uma sinopse do problema.

2.1 Conceitos

Jogos eletrónicos, também denominados de *videogames* ou videojogos, são uma das formas mais comuns de entretenimento. Deste modo, este ponto serve para perceber tudo o que envolve a criação de um jogo, desde do seu design ao seu desenvolvimento, passando por todas as fases de concetualização. É abordado o tópico de *gaming* para dar um contexto sobre o mundo dos jogos em geral. Também é importante entender como se procede à criação de jogos, ou seja, ter uma visão do lado do criador. Sobre *Game Design*, é explorado a ideia de como se deve desenhar/projetar um jogo. Sobre *Game Engine*, é descrito o seu propósito. É apresentado o jogo, no tópico de *Minecraft*. Já no *Modding* é retratado um dos temas de criação amadora nos jogos eletrónicos. Estes são os principais conceitos estudados e utilizados ao longo do projeto.

2.1.1 *Gaming*

Gaming é um termo usado para designar a atividade de jogar jogos eletrónicos através de consolas, computadores, smartphones e outros meios tecnológicos e é visto, principalmente, como um passatempo, uma forma de relaxamento das pessoas, quer individualmente, quer coletivamente. Tecnicamente há dois modos para jogar, *singleplayer* e *multiplayer*, onde as pessoas são chamadas de *gamers* [47]. O *gaming* também já é visto como um meio competitivo que tem vindo a afirmar-se cada vez mais.

Podemos dizer que o *gaming*, enquanto atividade de jogo, tem estado em constante evolução, sendo uma das mais populares formas de entretenimento no mundo. Enquanto indústria de *videogames*,

na última década, esse mercado tem crescido significativamente, originando um dos mercados mais poderosos de media [53].

A revolução do mundo tecnológico também afeta diretamente esta área. A actividade de game design tem crescido consideravelmente com o potencial de imersão das novas tecnologias, e isso nota-se pela qualidade apresentada nos jogos recentemente desenvolvidos, levando a arte de projetar e criar um jogo para novos patamares de complexidade. Desde a década de 70 que os videojogos já sofreram tantas mudanças, e evoluíram de tal maneira, que é usado o termo de "casual gaming" para jogos intermitentes (jogos mobile) e "hardcore gaming" para jogos onde os gamers dedicam horas do seu dia a jogar. Como exemplo de jogo de elevada frequência e intensidade podemos considerar o jogo *League of Legends*) (figura 2.1. Nomeado de LOL, em 2014 registou mais de 67 milhões de pessoas a jogar por mês e no seu campeonato mundial contou com um prémio de 2,3 milhões de dólares para a equipa vencedora (fora os outros prémios), além dos mais de 32 milhões de espetadores online [54].

Figura 2.1: League of Legends [54]

2.1.2 Criação de *videogames*

Todo o *videogame* nasce de uma ideia. Este é o primeiro passo na fase de concepção, para a criação de um jogo. A concepção passa pela definição das características de alto nível, isto é, a descrição dos principais aspetos do jogo e das interações do jogador. A partir dessas definições procede-se ao desenvolvimento do protótipo e, consequentemente, do jogo final [22]. A criação de jogos tem duas etapas iniciais críticas, o *design* do jogo e o desenvolver do protótipo do jogo. O primeiro refere-se à experiência de jogar o jogo e como este será, e o segundo ao desenvolvimento concreto do objecto experimentável. [20]

É precisamente na fase de desenvolvimento do jogo que entram os motores para criar jogos. Basicamente, tem a capacidade de conjugar todos os elementos essenciais que tem de estar presentes no jogo. Estes motores permitem aos criadores independentes ou às empresas/estúdios criarem os seus jogos de maneira mais eficiente do que começar do zero.

Com estes programas - *Game Engines* - é possível criar as diversas componentes técnicas de um jogo, como as partes gráficas, a programação de (*scripts*), a definição de inteligência artificial, os sistemas de colisões e os sistemas de comportamentos associados aos personagens e objetos.

2.1.3 *Game Design*

Game Design é a arte que combina a criatividade e solução do problema para a criação de vídeojogos, por outras palavras, "the act of deciding what a game should be."(Schell, 2014) [40]. Esta

é uma área focada na criação de jogos, que abrange desde da conceção até à produção do jogo [17]. Como já referido, a indústria do *gaming* tem estado em constante crescimento, sendo um dos fenómenos da cultura popular da atualidade, assumindo-se como uma das maiores atividades de entretenimento. Dessa forma, a procura pelos profissionais de *Game Design* também tem aumentado, assim como as expectativas de inovação de conceitos e produção de todo o sector.

O que é *Game Design*?

Como Schell menciona, *Game Design* é o ato de decidir o que o jogo deve ser [40]. Não com a simplicidade que faz parecer, pois a criação de um jogo é um processo que implica bastantes decisões. Basicamente o *design* do jogo pode ser um processo mental que envolve tormar decisões e nem é necessário um software para tal, ou sequer saber programar. Aliás, um jogo não é necessariamente tecnológico, abrange todas as categorias de jogos, desde jogos de cartas a jogos físicos, como o clássico jogo da "apanhada". Posto isto, o *Game Design* envolve decidir sobre regras, tempo, ritmo, aparência, recompensas, punições e todas os restantes elementos que incidem e ajudam a dar forma áaventura do jogador, e esse é o papel do *game designer*. [40][31] [20]

Game Designer

O papel de *designer* de jogos é, pode ser entendido como uma especialização, de certa forma, com alguns aspectos comuns ao papel de *designer* de interação, gráfico ou *designer* industrial, ou até ao do arquiteto, na medida em que tem de resolver problemas de criação [31]. Em geral, o *game designer* está envolvido no desenvolvimento do jogo desde o seu início, isto para que, à medida que o processo de desenvolvimento progrida, possa tomar as decisões necessárias com base no que vai sendo possível realizar e face à experiência com os protótipos. É importante realçar que um *game developer* é qualquer um que esteja envolvido na criação de um jogo, e não é correcto confundir os dois papéis de developer e designer. No entanto, de certo modo, qualquer um que tome decisões dentro do que o jogo deve ser, faz dele um *game designer* (*"Designer is a role, not a person"*, Schell). [40][31][17]

Esquema de *Game Design*

Um esquema em *game design* é uma forma de apresentar um ponto de vista de *design* de jogos. Serve de certo modo para organizar o conhecimento. Então, o esquema ajuda a entender o jogo, é um método abstrato que se pode adotar numa criação ou análise de um jogo. No livro *"Rules of Play: Game Design Fundamentals" (Eric Zimmerman e Katie Salen)[31]* é apresentado esse ponto de vista organizados em três esquemas essenciais, **Regras**, **Interação** e **Cultura**:

Regras (organização do projeto)
Esquemas de design para estruturar os princípios de lógicas e matemáticas do jogo.

Interação (experiência humana)
Esquemas de design para representar a experiência do jogador em primeiro plano com o ambiente do jogo, seja por ações do jogo ou contactos com outros jogadores.

Cultura (contextos culturais envolvidos)
Esquemas de design para contextualizar aspetos culturais dentro da composição do jogo projetado.

Estes tipos de esquemas são fundamentos sobre *Game Design* e ajudam a pensar e estudar o design do jogo. Esses esquemas conjugados fazem com que se projete os aspetos pertinentes de um jogo. Não é obrigatório seguir esta metodologia, existem outras igualmente capazes, mas é uma ferramenta para facilitar a conceção de problemas de *design* eficazes e que podem levar a soluções interessantes. [31][40]

2.1.4 *Game Engine*

Uma *Game Engine* (ou motor de jogo) é um software capaz de fornecer aos seus utilizadores um conjunto de funcionalidades essenciais para a criação de jogos complexos de um modo mais eficaz [22]. Para ser mais exato, o motor de jogo consiste numa arquitetura organizada de um conjunto de estruturas de informação (árvore do cenário, avatares, câmaras, etc) e de bibliotecas/APIs selecionadas (renderização gráfica, som, física, etc), para as especificidades de cada jogo [53]. A utilidade da *Game Engine* é a de permitir a tarefa de representar todos os elementos dos jogos de forma eficaz e organizada, desde personagens, cenários, movimentos, áudio, efeitos, etc.

Antes de começar a utilizar as *Game Engine*, os programadores de jogos tinham de programar o jogo todo desde a primeira linha. O jogo era programado desde a criação da tela de jogo até à definição de todas as estruturas de informação, às mecânicas, a física para as colisões, as animações, o áudio, os efeitos especiais, até a própria compilação do jogo para um arquivo de executável e conteúdos, que fosse possível partilhar com os jogadores. A conceção do jogo era assim muito complexa, sendo preciso uma formação avançada e vários anos para a conclusão de um bom jogo. [10]

Deste modo, as *engines* trouxeram um equilíbrio ao nível de recorrer a código e a funcionalidades pré-definidas para a elaboração de um jogo, sendo que umas é necessário programar mais do que outras. Também ajudaram no sentido de não ser preciso um uso extremo dos recursos computacionais. Assim, permite a uma pessoa comum desenvolver um jogo do zero. [22]

2.1.5 Minecraft, um exemplo de sandbox

Minecraft é um dos jogos mais populares dos últimos anos e já alcançou uma legião de fãs no mundo inteiro desde a sua criação. Minecraft é um jogo independente de mundo aberto que permite a construção de infinitos mundos com o uso de blocos, que é o formato do jogo. Neste sentido, os jogadores podem interagir com o mundo do jogo de maneira criativa, envolvendo a destruição e colocação de blocos, e assim construir as suas formas específicas e criativas, inventando várias formas de jogar o jogo a partir da exploração dos seus dois modos (sobrevivência e criativo). [3]

2.1.6 *Modding, uma apropriação de design amador*

Modding é uma expressão que designa o ato de "modificar", alterando aspectos dos *videogames* como conteúdos, cenários, parametros ou até regras [47]. Através de modding o jogador torna-se designer amador e modifica aspetos do jogo original, como a aparência ou o comportamento.

Essas alterações são chamadas de *mods*. A maioria dos *mods* são adições de conteúdo ao jogo, para acrescentar criatividade ao mundo ou aumentar as opções de interação do jogador [3]. Outro género de *modding* muito utilizado é o de aumentar a performance do jogo para que funcione melhor, este serve sobretudo para fazer ajustes na jogabilidade da preferência do jogador. De certo modo, o *modding* aponta para melhorar ou personalizar os jogos depois do seu lançamento, mas visam sobretudo realizar adaptações da preferência de grupos de jogadores.

2.2 *Game Engines*

Antes da existência das *game engines*, os vídeojogos eram desenvolvidos desde o nível mais basico da programação, o que tornava a tarefa de criar um jogo, muito complexa e tecnicamente inacessível à maioria das pessoas. Os jogos eram escritos de raiz, com recurso a programação pura[53]. Uma modificação num sistema de jogo implicava um esforço considerável das várias partes, incluindo *designers* de jogo, artistas gráficos, e programadores, para poderem desenvolver o protótipo pretendido. [10]

Hoje em dia, a maioria dos pesquisadores procura uma plataforma de criação de jogos que facilite o desenvolvimento e a modificação de tais aplicativos de maneira fácil e rápida, sem ser preciso programar "intensivamente"o jogo inteiro. Por esse motivo, em meados dos anos 90 surgiu o conceito de motor de jogo, que separa os mecanismos basicos como a definição de estruturas de informação e conteúdo, renderização gráfica e sonora, etc., do desenvolvimento de conteúdo específico do comportamento do jogo (habitualmente na forma de scripting)[22].

Assim sendo, serão abordados os moldes em que iremos caracterizar as *game engines* e as ferramentas de *software* que dão suporte. Algumas soluções de sucesso estudadas, de forma a entender os padrões de ferramentas que dispõe e recursos que fazem a diferença em relação aos outros.

2.2.1 Bibliotecas e *Application Programming Interface* (API)s

Como referido acima, uma das definições de Game Engine consiste num conjunto de bibliotecas. Estas bibliotecas servem para acelerar o processo de desenvolvimento de um jogo [53]. As APIs tem a mesma função, oferecendo formas normalizadas de invocação de subprogramas que dão suporte ao motor de jogo. Concretamente, as bibliotecas e as APIs são *frameworks* que suportam as necessidades específicas que permitem implementar as mecânicas do jogo. Sendo que as APIs estão mais ligadas ao desempenho dos jogos, como os gráficos, por exemplo, e as bibliotecas frequentes para representar ou incluir recursos dentro dos jogos, como a física (movimentos, colisões, etc.).

Este conceito de motor de jogo enquanto conjunto de bibliotecas surge devido à alta complexidade que um motor de jogo apresenta, dessa maneira permitindo (e obrigando) a “reutilizar” ao máximo essas ferramentas (bibliotecas e APIs) que já dispõem o que é pretendido [18].

De seguida, apresentamos as categorias que são mais aplicadas nos jogos e uma breve descrição das suas principais funcionalidades para os jogos. Entre as categorias com maior relevância, destacaremos o áudio, vídeo, física, modelagem, texturização, renderização, inteligência artificial, programação de eventos, *interface* do utilizador, dispositivos de interação, redes e segurança, com base na Monografia de Toledo e Silva [18].

Audio
Componente que é responsável por controlar o áudio do jogo. Essas componentes têm recursos de qualidade para desempenhar funcionalidades como edição, gravação, compressão, reprodução com base em posicionamento de áudio 3D e mistura conforme os eventos pré-programados. As novas tecnologias de áudio já permitem a reprodução simultânea até 128 camadas de áudio, o que permite uma alta qualidade e detalhe na mistura sonora.

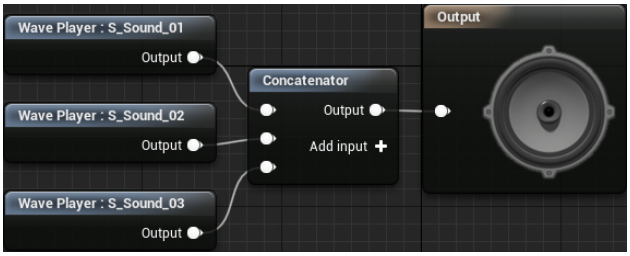


Figura 2.2: Unreal4: editor de áudio
Fonte: https://docs.unrealengine.com/en-US/Engine/Audio/SoundCues/Editor/index.html

Vídeo
Esta componente é responsável pelo processamento de vídeo nos jogos, desde a produção de animações à inserção de vídeos em objetos tridimensionais. Edição, compressão, conversão e reprodução de vídeos nos jogos são funcionalidades comuns nesta categoria.

Animação
Esta categoria está relacionada com as animações realistas nos jogos, como movimentos corporais e faciais dos personagens em tempo real. Estas tecnologias são constituídas por captura de movimentos, deteção de colisões e simuladores de corpos rígidos e flexíveis (ver figura 2.3).

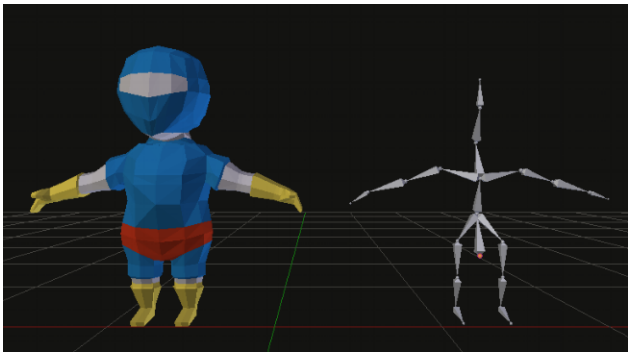


Figura 2.3: Animação 3D
Fonte: https://www.haroldserrano.com/blog/how-3d-animations-work-in-game-engines-an-overview

Modelação
Componente que está relacionada com o desenvolvimento de modelos tridimensionais, esses modelos já podem vir em bibliotecas de objetos como personagens (ver figura 2.4), árvores, nuvens, cidades, mundos, entre outros, disponíveis para facilitar na produção do jogo.

Figura 2.4: Modelação 3D
Fonte: http://www.aldeiarpg.com/t2148-blender-modelagem-e-criacao-de-games-3d

Texturização

Área focada em manipular texturas para a criação de ambientes realistas ou estilizados. As novas tecnologias de texturização facultam a sobreposição de várias camadas de imagens nos objetos de modo a aperfeiçoar os detalhes neles visiveis (ver figura 2.5).

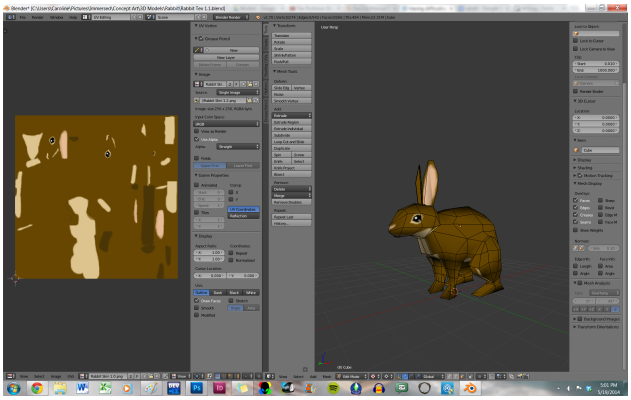


Figura 2.5: Edição de texturas num objeto
Fonte: https://blender.stackexchange.com/questions/10517/textures-appearing-different-in-edit-and-object-mode

Renderização

Renderização é o processo de geração de uma determinada cena 3D em imagem digital (2D) final através de cálculos executados por um *software* de renderização. Durante esse processo, são utilizados alguns algoritmos de modo alcançar imagens realistas capazes de simular efeitos visuais de acordo com a natureza. Esta área de desenvolvimento costuma ser a maior responsável por apresentar altos níveis de qualidade gráfica nos jogos.

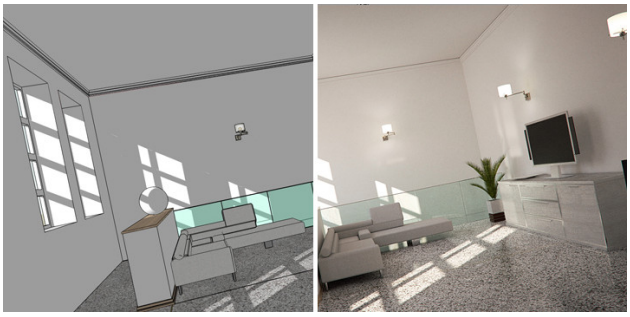


Figura 2.6: Renderização, antes e depois
Fonte: https://www.hometeka.com.br/pro/os-6-melhores-renders-para-sketchup/

Física
Componente que tem como função gerar soluções na deteção de colisões, na coordenação dos movimentos das personagens, nos efeitos de partículas e em simulações, como simulação de projecteis, fluidos e tecidos.

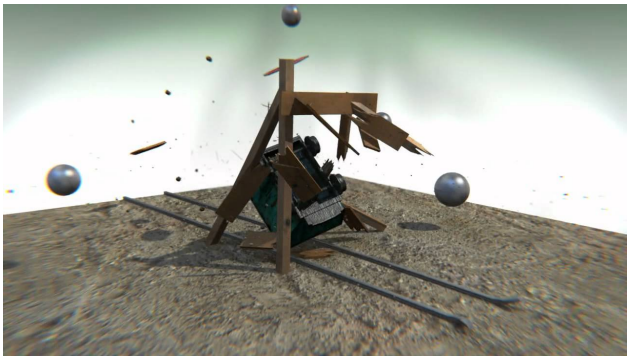


Figura 2.7: Destruição (simulação física)
Fonte: $https://www.youtube.com/watch?v = hS9seZ_G6pU(2:09)$

Inteligência Artificial
Área responsável pela programação da dinâmica dos personagens extras que não são controlados pelo jogador e que precisam de ter certos comportamentos no jogo, normalmente designados personagens não jogáveis (*non-player character* **NPC**). A inteligência artificial já permite a simulação da perceção espacial de personagens, autoprogramação de trajetórias, simulação de comportamentos, entre outras coisas.

Figura 2.8: IA nos jogadores adversários que tentam retirar a bola (Fifa 14)
Fonte: https://www.techtudo.com.br/noticias/noticia/2013/05/fifa-14-tera-jogadores-com-inteligencia-artifical-melhorada-segundo-ea.html

Programação de eventos
Programação de eventos, como o nome indica possibilita meios interativos e dinâmicos, através de programação visual (*interface* com funções pré-definidas de eventos) ou através de código (*scripts*) caso sejam precisos eventos que não estão definidos na programação visual. Essas bibliotecas e APIs podem fornecer bastantes funções própria que proporciona uma maior flexibilidade na programação dos eventos, tornado esse processo muito menos prolongado.

***Interface* do utilizador**
Interface gráfica de utilizador (*Graphical User Interface* **GUI**) permite a interação do utilizador com os dispositivos digitais. Estas *interfaces* são programadas para facilitar essas interações dos utilizadores, em tempo real.

Figura 2.9: GUI de um jogo (menu)
Fonte: https://marketplace.coronalabs.com/graphics/medieval-ages-game-gui

Conexão de Rede
Componente para permitir que os jogos sejam *multiplayer* (vários jogadores em simultâneo no mesmo jogo, figura 2.10), através da internet ou mesmo numa rede local. Esta é importante para garantir a conexão em segurança à internet, a privacidade dos dados fornecidos pelos

jogadores, que a conexão não seja lenta (latência alta ou "ping"alto) e proporcionar servidores de 24 horas capazes de suportar todos os utilizadores em jogos online.

Figura 2.10: Jogo multijogador (Fortnite)
Fonte: https://fnbrmobile.wordpress.com/2018/03/29/download-fortnite-battle-royale-for-android-ios/

Segurança
Esta é uma área que procura garantir os direitos de autor dos desenvolvedores na fase de lançamento do produto. Também procura garantir a seguranças dos utilizadores e a viabilidade dos produtos, como autenticação online para os utilizadores e inibição do uso de "*cheats*" nos jogos online (fazer batota).

2.2.2 Unity

A Unity[1] é uma engine que domina 45% do mercado global. Permite criar jogos 2D e 3D com estilos de gráficos diferenciados, tais como as suas mecânicas [4]. Este possibilita a criação de jogos para as diferentes plataformas.

Esta engine está entre as mais utilizadas para a criação de jogos, de modo que iremos averiguar quais são os fatores para a sua popularidade:

- **Facilidade**
  Para criadores iniciantes esta *engine* pode ser uma porta de entrada devido à facilidade de aprender. Tem uma *interface* intuitiva, com as várias secções devidamente separadas, como as secções de *scripts*, colisões de objetos, *assets* e outras funções. O Unity dispõe de uma série de tutoriais para o utilizador, criados pela própria empresa. Estes tutoriais tanto ajudam a dar os primeiros passos como para utilizadores que não sabem programar.
- **Documentação**
  A Unity[1], como os tutoriais que oferece, também disponibiliza aos seus utilizadores vários fóruns para discussão como documentações que estão disponíveis online. Devido à sua popularidade também existem vários exemplos feitos por utilizadores disponiveis online. Desde tutoriais no YouTube a projetos no GitHub, muitos exemplos podem ser encontrados.
- **Multi-plataformas**
  Esta *engine* fornece a conteúdo para mais de 25 plataformas. Para os criadores, ainda possui versões para os três sistemas operativos apesar de só ter alternativas para PCs.

[1]para mais informações: https://unity.com/

- **Assets**
  A Unity[1] tem a sua própria loja de texturas, gráficos, efeitos sonoros, animações e outros materiais para a construção de um jogo, disponíveis para compra e venda por parte dos utilizadores e muitos desses materiais são gratuitos.

- **Free**
  É possível criar um jogo do zero sem precisar de gastar nada, a Unity tem uma versão gratuita com a maior parte das funcionalidades, somente utilizadores que conseguem ganhar alguma receita com jogos já criados é que optam por comprar a versão paga para ter mais opções da *engine*.

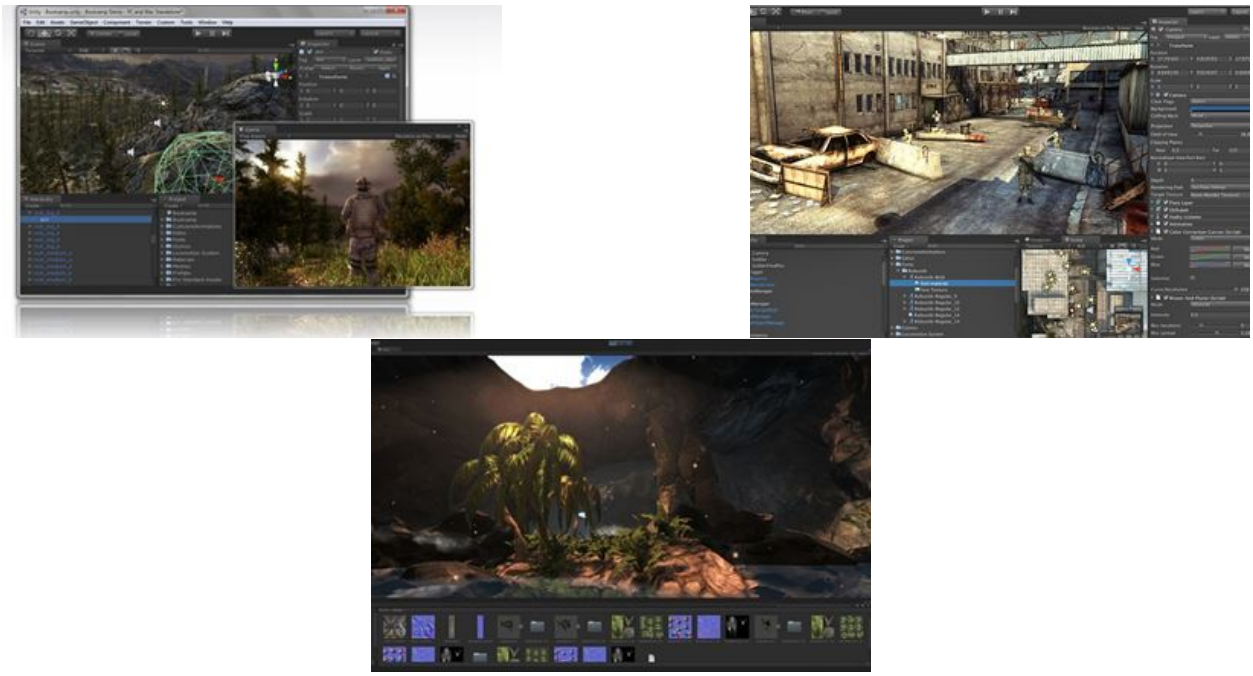


Figura 2.11: Unity
Fonte: https://www.baixaki.com.br/download/unity-engine.htm

2.2.3 UnReal

Esta engine está por trás de jogos famosos como a série do Assassin's Creed, Batman, BioShock, Unreal Tournament e muitos outros jogos de sucesso. A Unreal Engine[2] foi lançada pela Epic Games, uma conceituada produtora/editora de jogos eletrónicos. A Unreal[2] foi pensada principalmente para a criação de jogos de tiro e o primeiro jogo feito foi um jogo de tiro em primeira pessoa (FPS). Acabou por ser utilizada para diversos estilos de jogos e já conta com 4 versões:

Unreal 1
Na primeira versão, o *software* incluía recursos para renderização, deteção de colisões, inteligência artificial, sistema de redes e sistema de arquivo. Também já tinha suporte de uma linguagem para *scripts* e de um sistema cliente-servidor.

[2]para mais informações: https://www.unrealengine.com/

Unreal 2
A segunda versão recebeu um *update* no editor, escreveram de novo o código de renderização, acrescentaram suporte para novas plataformas e para a física de veículos e física de ragdoll.

Unreal 3
Esta versão apresentou uma grande adição, os efeitos gráficos e suporte para outras plataformas.

Unreal 4
Esta é a versão mais recente e trouxe uma série de novidades, como nas outras, proporcionou um maior suporte para as plataformas mais recentes, um sistema de iluminação global (antes usavam iluminação pré-renderizada), um novo sistema de scripts (o "Blueprint), uma melhoria no funcionamento do programa e um desenvolvimento mais rápido por parte dos utilizadores.

Recursos

- **Blueprint**
  Sistema de *scripts*, editor que possibilita a modelação e a montagem de mecânicas nos jogos e desenhar a *interface* do utilizador sem o auxílio de programação.

- **Niagara VFX**
  Programa para gerar efeitos especiais (fogo, chuva, fumo, etc.). Inclui um simulador de partículas e um sistemas de colisões.

- **Matinee**
  Ferramenta de animações que oferece ao utilizador a capacidade de animar os intervenientes ao longo do jogo e assim desenvolver sequências cinematográficas para melhorar a jogabilidade de um jogo.

- **Inteligência Artificial**
  Esta *engine* é capaz de aplicar inteligência artificial nos personagens para que as suas ações nos jogos sejam mais inteligentes em relação ao cenário que estão envolvidos.

- **Hot Reload**
  Função que permite ao utilizador atualizar o código do jogo ao mesmo tempo que o jogo está a correr, isto é, não precisa de voltar a correr o jogo para testar alguma mudança na edição.

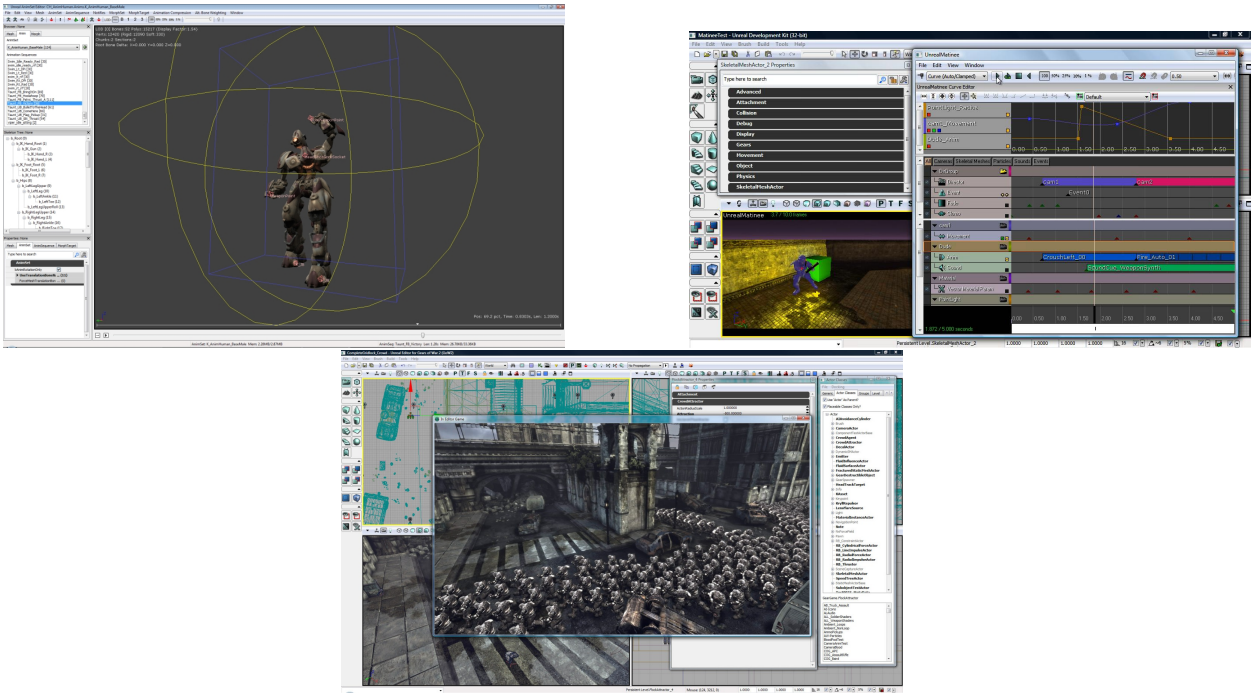


Figura 2.12: Unreal
Fonte: https://www.baixaki.com.br/download/unreal-development-kit.htm

2.2.4 Construct2

Das *Game Engines* analisadas, esta *engine* é a que se aproxima mais aos objetivos deste projeto. A Construct2[3] possibilita a criação de jogos digitais 2D, com o apoio de HTML5. Este engine pode ser usado por criadores que não sabem programar, apenas sendo preciso usar algum raciocínio lógico. Todo o desenvolvimento de um jogo é feito através de um meio visual que usa um conceito de *evento-ações*, isto é, para um determinado evento sucede-se uma ou mais ações. Para este conceito a Construct2[3] já fornece um conjunto de componentes e comportamentos para a construção de um jogo. Esta *engine* é mais propícia a utilizadores que se estão a iniciar como criadores de jogos por ser bastante simples e fácil de aprender, permitindo criação de jogos funcionais em 2D de uma maneira bastante rápida. [34] [22]

Recursos

- ***Quick & Easy***
  Apresenta uma *interface* visual simples de usar e um sistema de *drag-and-drop*, o criador pode arrastar para a sua área de trabalho todos os itens e funções disponíveis no sistema.

- ***Powerful Event System***
  Também tem um sistema capaz de criar eventos e recursos para adicionar comportamentos a objetos.

[3]para mais informações: https://www.scirra.com/construct2

- ***Flexible Behaviors***
  Dispõe de um conjunto de funções pré-inseridas para atribuir comportamentos aos objetos.
- ***Instant Preview***
  Permite visualizar o jogo instantaneamente. Não é preciso compilar o jogo para ser testado.
- ***Stunning Visual Effects***
  Tem um editor de imagens integrado que fornece ajustes ágeis de gráficos, com ajuda dos modos de efeitos, sistemas de partículas e misturas.
- ***Multiplatform Export***
  Ampla gama de plataformas para exportar os jogos num único projeto.
- ***Easy Extensibility***
  Usa *JavaScript* como linguagem de suporte para a aplicação de *plug-ins* e comportamentos.

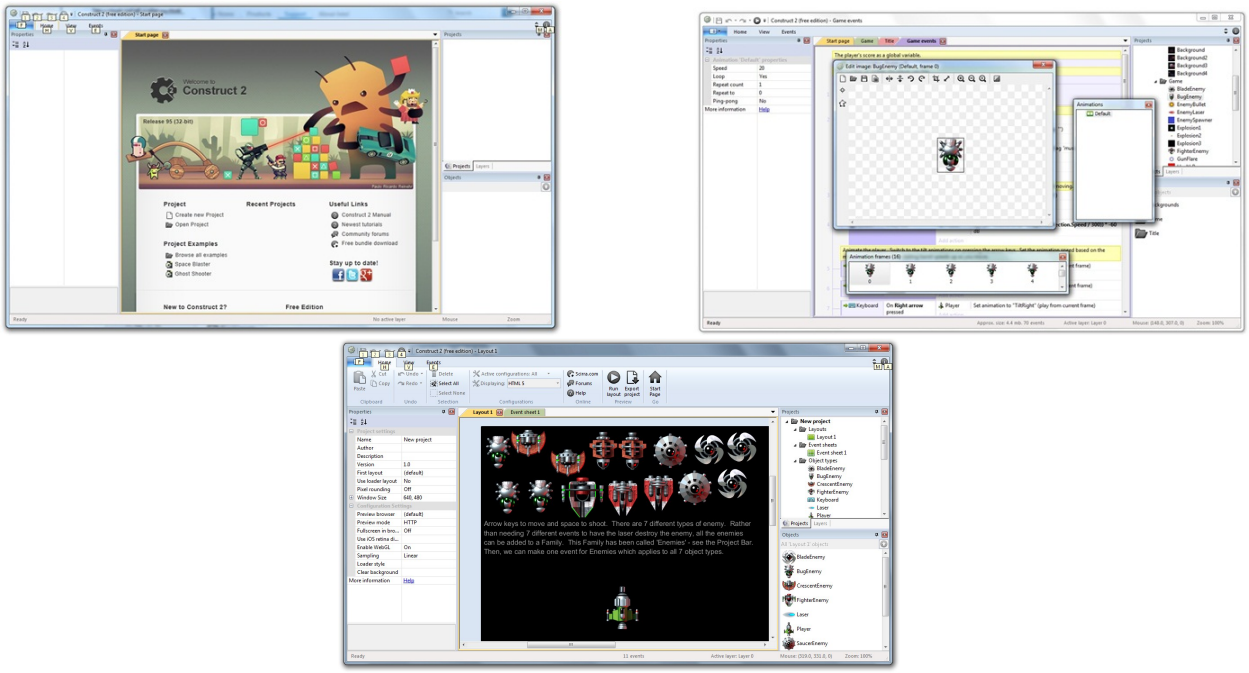


Figura 2.13: Constrcut2
Fonte: https://www.baixaki.com.br/download/construct-2.htm

2.2.5 PICO-8

Muito parecido com os jogos de consolas dos anos 80, o PICO-8 é consola virtual para criar, jogar e partilhar pequenos jogos e animações [33]. Criado pela Lexaloffle Games[4] com o intuito de promover grande parte da criatividade que caracteriza os jogos de antigamente (ver figura 2.14, esta consola virtual foi pensada para impor limitações no desenvolvimento dos jogos, de modo, a que os desenvolvedores precisam de elevar a sua imaginação e encarar esse obstáculo para criarem os seus próprios jogos [6].

[4]Para mais informações: https://www.lexaloffle.com/

Specifications	
Display	128x128 16 colours
Cartridge Size	32k
Sound	4 channel chip blerps
Code	Lua
Sprites	256 8x8 sprites
Map	128x32 cels

A plataforma funciona como um *Game Engine*, onde os *game designers* recebem as ferramentas necessárias para desenvolverem os seus jogos, mas tem que se adequar às limitações do PICO-8 (figura ao lado), uma tela de 128 x 128 pixels, palete de 16 cores, 4 canais para efeitos sonoros e ainda uma capacidade máxima de armazenamento de 32 kB [33][6]. Estas limitações foram propositadamente colocadas para incentivar a fazer projetos pequenos, mas que sejam muito expressivos promovendo identidade pessoal de cada um e, principalmente, para serem divertidos de desenvolver [33].

Recursos

- ***Creative Tools***
  O PICO-8 dispõe de um ambiente de desenvolvimento muito recetivo, fornecendo ferramentas para a edição de código (em linguagem Lua), música e som, *sprites* (objetos animados) e mapas pré-construídos.

- ***Shareable Cartridges***
  PICO-8 *cartridges* (ou cartuchos) podem ser guardados em formato PNG e facilmente compartilhados com outros utilizadores, através da exportação para aplicativos HTML5, Windows, Mac e Linux, ou então através da Web.

- ***Explore the Cartverse***
  O PICO-8 apresenta uma funcionalidade de exploração chamada SPLORE, uma coleção *online* de *carts* para pesquisar.

- ***Community Resources***
  O PICO-8 já conta com uma comunidade agradável, possibilitando partilhas de conhecimentos, ideias, ferramentas e tutoriais.

- ***Just Add Hardware***
  O PICO-8 é um programa que ocupa pouco espaço e funciona em Windows, Mac, Linux e Raspberry Pi (computador de baixo custo) com 700MHz de CPU.

Legends of Pizza Night | Dank Tomb | Combo Pool | Hyperspace

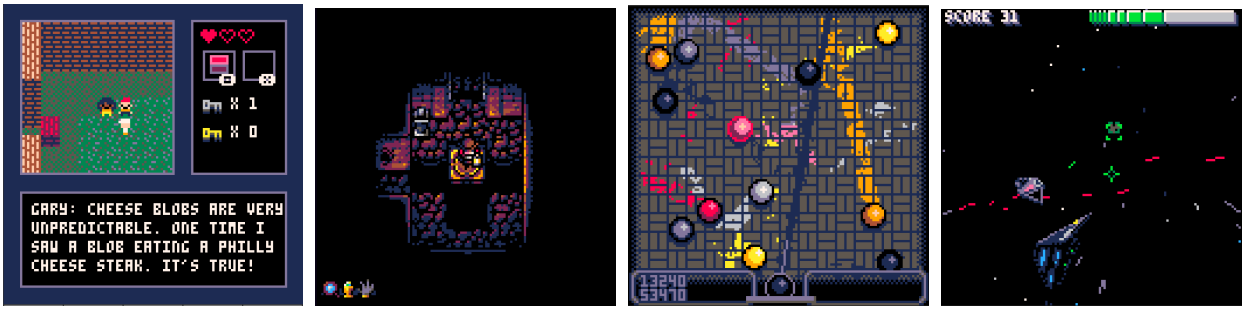


Figura 2.14: Jogos feitos em PICO-8 [23]

2.2.6 Tabela Comparativa

Nesta secção faremoe uma síntese comparativa das *game engines* enunciadas. São apresentadas duas tabelas, 2.1 e 2.2, para comparar as principais características de cada engine, tal como os recursos que disponibilizam para editar e criar jogos. Também é feita uma revisão individual dos **PROS** e **CONTRAS** que cada *engine* expõe, nas posteriores tabelas, 1 (Unity3D) [13], 2 (Unreal4) [14], 3 (Construct 2) [11], 4 (Pico-8) [12], com base em *reviews* de utilizadores reais (ver em AppendixA 7.3).

Nome	**Scripting (Linguagem de programação)**	**plataforma**	**2D/3D**
unity	C#, Cg, HLSL	cross-platafrom	2D,3D
UNREAL ENGINE	GLSL, Cg, HLSL, UnrealScript, C++, Blueprints	cross-platafrom	3D
Construct 2	JavaScript, Event System	Windows, OS X, Wii U, HTML5 internet browsers	2D
PICO-8	Lua	HTML5, Windows, Mac, Linux	2D

Tabela 2.1: Comparações de *Game Engines* [49][24][35][30]

Nome	Recursos de edição	Recursos de criação
unity	2D and 3D mode settings, Preferences, Presets, Shortcuts Manager, Build Settings, Project Settings, Visual Studio, C# integration, RenderDoc Integration, Xcode Frame, Debugger integration, Editor Analytics, Check For Updates, IME in Unity, Special folder names, Reusing Assets between Projects, Version Control, Troubleshooting The Editor	Scenes, GameObjects, Prefabs, Input, Transforms, Constraints, Rotation and Orientation in Unity, Lights, Cameras, Adding Random Gameplay Elements, Cross-Platform Considerations, Publishing Builds, Troubleshooting
UNREAL ENGINE	Level Editor, Material Editor, Blueprint Editor, Behavior Tree Editor, Persona Editor, Cascade Editor, Niagara Editor, UMG UI Editor, Matinee Editor, Sound Cue Editor, Paper2D Sprite Editor, Paper2D Flipbook Editor, Physics Asset Tool Editor, Static Mesh Editor, Media Player Editor, Font Editor	Projects, Objects, Classes, Actors, Components, Pawns, Characters, PlayerController, AIController, Brushes, Levels, World, GameStates, PlayerStates
Construct 2	File menu, Ribbon, Properties Bar, Object Bar, Layers Bar, Z Order Bar, Tilemap Bar, Bookmarks Bar, Image and Animations editor, Layout View, Event Sheet View, Dialogs, Keyboard Shortcuts, Debugger	Projects, Layouts ,Layers, Objects (Plugins, Object types, Instances, Instance variables, Behaviors, Effects, Families, Containers), Events (How events work, Event sheets, Conditions, Actions, Expressions, Sub-events, Groups, Comments, Includes, Variables, Breakpoints), Files, Sounds & Music
PICO-8	Code Editor, Sprite Editor, Map Editor, SFX Editor, Music Editor	File System, Loading and Saving, Using an External Text Editor, Backups, Configuration, Controller Setup, Screenshots, Videos and Cartridge Labels, Sharing Cartridges, Exporters / Importers, Splore

Tabela 2.2: Comparações de *Game Engines* (cont.) [49][24][35][30]

2.2.7 Exemplos de Bibliotecas e APIs

Depois de uma breve análise das categorias mais aplicadas nos jogos mais acima, é demonstrado algumas das bibliotecas e APIs existentes para jogos. Estes são exemplos que são integráveis nas *Game Engines* e foram retirados com ajuda deste site[5] [16].

Renderização

Vulkan (Khronos Group), normalização do acesso de baixo nível ao hardware da GPU [32]

DirectX 12 (Microsoft), API que fornece acesso de baixo nível ao hardware da GPU em Windows [36]

Metal (Apple), API que permite acesso de baixo nível ao hardware da GPU em sistemas da Apple [21]

OpenGL, API gráfica 3D para dar acesso à acelaração de hardware [38]

Audio

Audiere, API de áudio de alto nível [5]

FMOD, API de áudio adaptável [46]

OpenAL, API para manipular áudio multicanal tridimensional [37]

Video

GStreamer, biblioteca de manipulação de video/áudio streaming [26]

JavaMedia, framework em Java que permite manipular conteúdo multimédia (áudio/video/imagem) [39]

Física

Havok, tecnologia de deteção de colisões e simulação de física [29]

Bullet, motor de física para simular deteção de colisões e dinâmica de corpos rígidos e corpos flexiveis [7]

PhysX, motor de calculos de simulação física para jogos [25]

Chipmunk2D, biblioteca de física de corpo rígido 2D simples [9]

[5]para mais informações: http://desenvolvimentodejogos.wikidot.com/bibliotecas-e-apis

Imagem

Libjpeg, biblioteca para manipulação de imagens em formato JPEG [44]

Libpng, biblioteca para manipulação de imagens em formato PNG [1]

DevIL, biblioteca de imganes com vários recursos de carregamento de imagens [43]

GUI (Interface do utilizador)

CEGUI, biblioteca que oferece janelas e *widgets*(elementos de interação) para fazer GUIs [45]

SFML, biblioteca multiplataforma que fornece uma interface simples para vários componentes multimédia [41]

GTK, *toolkit* (conjunto de ferramentas) multiplataforma para criar GUIs [27]

WxWidgets, *toolkit* para criar GUIs multiplataforma [55]

Apreciação,
estas bibliotecas têm a possibilidades de dar suporte ao motor de jogo deste trabalho. Há ainda outras tantas não mencionadas que também poderiam ser alternativas viáveis, embora menos conhecidas ou documentadas. Fundamentalmente a escolha deverá incidir nas bibliotecas e APIs que possibilitam a sua utilização em múltiplas plataformas, de modo a que o motor de jogo seja também para multiplataformas (*cross-platform software*).

Vulkan

Vulkan é uma API recente. Esta é uma API gráfica 3D e de computação feita para múltiplas plataformas e que tem um *overhead* baixo. A Vulkan procura, sobretudo, oferecer ao desenvolvedor um controlo de baixo nível de hardware, além de fornecer alta eficiência, performance e recursos da GPU, que são utilizadas em vários dispositivos tecnológicos. [32]

Fazendo uma comparação com OpenGL, outra API de computação gráfica muito requisitada, a Vulkan possibilita que o hardware tenha mais tempo para fornecer os recursos e para proporcionar um melhor desempenho e qualidade de imagem, devido à redução considerável do "overhead" que a API da Vulkan exibe. [8]

Unreal Engine 4 e *Unity3D* são dois exemplos de engines suportadas pela Vulkan.

2.3 Outros Temas

Nesta secção mostra-se o que foi explorado além do tema principal. Estes são temas que podem de alguma forma ajudar a construir a *Game Engine* proposta. Sobre o Minecraft fez-se uma análise para compreender os mecanismos do jogo e como se tornou num dos jogos mais populares nos últimos anos. No *Modding* estudou-se a sua finalidade e alguns *mods* existentes no Minecraft. Estes dois temas juntos ajudam a dar uma perceção de simplicidade no meio dos jogos.

2.3.1 Minecraft

Visto que se verifica um movimento significativo de *modding* no Minecraft, este aponta para o potencial de um *sandbox* servir de base para ensaios de *game design*. Sendo assim, é importante estudar toda a estrutura do jogo e as motivações dos jogadores para entender na visão de construção de jogos a simplicidade que o jogo apresenta na criação de mundos. As infinitas possibilidades de mundos gerados torna o jogo numa fonte de criatividade para o jogador pelos recursos ilimitados que dispõe para a construção do qualquer coisa que imaginam. Muitas das funcionalidades que o jogo tem pode servir de base para a estrutura e a realização deste projeto, muito pela simplicidade que apresenta e pela cativação que obteve da comunidade do *gaming*.

2.3.2 Game Design do Minecraft

O jogo do Minecraft envolve os jogadores a construírem mundos com vários tipos de blocos [2]. Existe muito da criatividade dos jogadores, que além de construir podem fazer outras atividades como explorar, apanhar recursos e batalhar. [3]

É um tipo jogo muito diferenciado dos demais. Tem uma aparência visual que faz lembrar o LEGO[6] (ver figuras 2.15 e 2.16), por apresentar mundo composto por blocos. Digamos que parece que tem um conjunto de gráficos antigos. Conta com uma *interface* muito simples que é uma característica de todo o jogo. [52]

No início do jogo, o jogador está completamente vulnerável e enfraquecido e à medida que avançar, o jogador obtém recursos, aprende a construir, e a "conquistar o mundo!" [2]

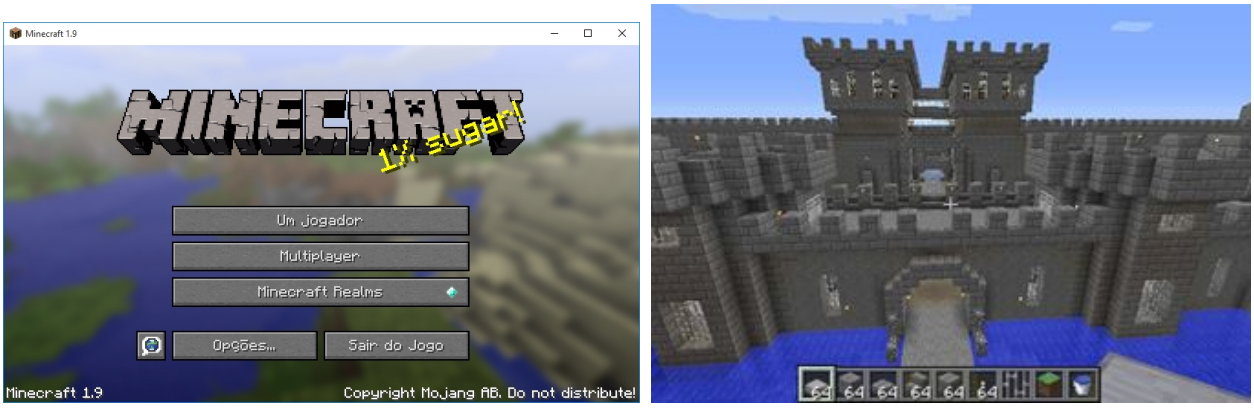


Figura 2.15: Minecraft
Fonte: https://www.baixaki.com.br/download/minecraft.htm

2.3.3 Modding

Os jogadores ou fãs de um jogo são os principais criadores desses *mods*. Estes são os que mais idealizam, imaginam e desejam acrescentar melhorias aos seus jogos preferidos. Assim eles podem

[6]Para mais informações: https://www.lego.com/

Figura 2.16: Lego Minecraft World
Fonte: https://www.lego.com/en-us/product/the-wool-farm-21153

expressar as suas ideias através deste mecanismo. Estes desenvolvedores de *mods* desempenham uma parte importante para os outros jogadores que desfrutam do jogo e dos *mods* disponíveis, desde da evolução proporcionam ao jogo até a correção de bugs (falhas encontrados dentro do jogo).

Porque o *modding* é importante para uma *game engine*, devem se questionar? É muito simples, o fenómeno de *modding* em Minecraft é inspirador porque mostra o potencial de usar uma *sandbox* como base para ensaios de *game design*. Assim, o *Modding* ajuda a perceber o que os jogadores gostam de ter e acrescentar num jogo. Portanto, percebendo as modificações comuns, ajuda também a atingir as funcionalidades que um *game engine* deve ter, de modo a proporcionar aos seus utilizadores as ferramentas necessárias para construírem os seus jogos.

***Modding* no Minecraft**

Na sua maioria, o *modding* no Minecraft, é caracterizado por adicionar conteúdo ao jogo. Esse conteúdo é representado por alterar o jogo nos aspetos visuais ou nas possibilidades extra de interações que o jogador pode fazer [3]. Estes são *mods* ao nível de modificações no jogo, realizadas por fãs. A finalidade deste mods é oferecer uma experiência completa do jogo [50].

A seguir é mostrado alguns exemplos de *mods* no minecraft, a lista é retirada do blog, "Conheça alguns dos melhores Mods para Minecraft"[50].

Pokémobs Mod

Como muitos outros *mods* que se baseiam em universos de jogos, filmes, séries, entre outros, este mod recria o universo do pokémon no Minecraft, como no jogo, o mod permite capturar e batalhar entre pokémons.

Figura 2.17: Pokémobs Mod *(Imagem: Reprodução/FANDOM Powered by Wikia)* [50]

SteamCraft Mod

Mod que contam com múltiplos itens extra adicionados, muitos desses itens tem de ser conquistados pelos jogadores, como armas.

Solar System Mod

Este *mod* conta com uma alteração do cenário visual, um cenário alusivo ao sistema solar, que permite ao jogador viajar entre planetas.

ChocoCraft Mod

Este mod é um mundo sobre a série Final Fantasy, conta com a presença dos Chocobos (aves conhecidas da série) e permite plantar Gysahl Greens.

Figura 2.18: ChocoCraft Mod *(Imagem: Reprodução/9Minecraft)[50]*

Tornadoes Mod

Possibilita aos jogadores ativar tornados no jogo, onde esses tornados devastam tudo por onde passam, assim os jogadores precisam de fazer construções resistentes e conquistar armaduras.

Portal Gun Mod

Mod que fornece aos jogadores a possibilidade de abrir portais no jogo ao disparar armas, baseado no universo da série Portal.

Zombieland Mod

Mod de um mundo com zombies (mortos-vivos), o jogador tem de sobreviver a um Apocalipse zombie e combater contra eles.

Apreciação,
Esses são exemplos, onde os fãs produzem ambiente diversos no jogo. Essas produções podem ser baseadas em outras histórias que enquadram bem no jogo. Temos exemplos de adição de personagens com características específicas (como poderes), items para acrescentar opções ao jogo, modos de interação entre jogadores, habilidades extras, eventos inseridos no jogo (como chuva), entre muitas outras possibilidades de alterações do jogo. Basicamente, estes *mods* tentam simular algo mais complexo que os fãs desejam, simulações como uma mapa todo detalhado ou um modo aventura repleto de desafios. É mesmo nestas infinidades de opções, que o *modding* se assemelha a um motor jogo.

Capítulo 3

Objetivos e Metodologia

Neste capítulo são apresentados os objetivos, a metodologia e o planeamento adotado para a realização deste trabalho. Mostra os objetivos pré-definidos, a metodologia a aplicar para cumprir esses objetivos e o planeamento do trabalho.

3.1 Objetivos

O objetivo principal centra-se no projecto de uma *Game Engine* minimalista, uma plataforma para criar jogos de forma rápida e simples. A intenção é projetar uma *engine* o mais simples possível, capaz de prototipar um jogo muito rapidamente. De certa maneira, há um plano de retroceder no tempo das *Game Engines* atuais, de forma, a retirar dividendos que se foram perdendo no tempo com o acrescentar de variantes e complexidade. Pretende-se um recomeço das *Game Engines*, explorando um caminho diferente, mas mais favorável, no contexto da experimentação em *game design*. Para tal, esta *engine* será inspirada no movimento de *modding* e numa análise crítica das *engines* estudadas. Com isto em mente, definem-se ainda outros objetivos secundários para ajudar a chegar à plataforma desejada.

3.1.1 Editor Simples

Apresentar um canvas simples e fácil de aprender, é uma prioridade. Ter uma *interface* visual muito simplificada ajuda ao utilizador a ter uma experiência acelerada no momento de ensaiar os seus jogos. De modo a construir um editor de jogo simples, são apontados sub-objetivos para ajudar:

Jogos 2D
A construção de jogos em 3D é um processo muito mais complexo e sobretudo mais demorado, do que em 2D. Como a ideia é ter um *software* o mais simples possível, a prioridade será só poder realizar jogos em 2D. Os jogos em 3D iam exigir muitas competências e horas para construir um protótipo simples, tornava a experimentação de extremamente complexa, sendo difícil criar ou modificar demos de jogos num curto espaço de tempo. O sistema 3D pode ser um possível implementação para uma próxima versão.

Poucas Skills de Programação

Ter uma plataforma que funcione à volta da lógica (com base nas redes de Petri), sem ser necessário saber programar de todo. Assim, proporcionar a todo o tipo de utilizadores a possiblididade de criar os seus próprios jogos. Sendo que, a utilização de *scripts*(programar) pode ser pensada para uma eventual versão futura da plataforma.

Ferramentas
A plataforma, dispor de um conjunto de ferramentas e recursos para o utilizador adicionar ao seu jogo, como objetos e comportamentos.

Sistema Drag-and-drop
Sistema *drag-and-drop* para o utilizador poder arrastar items, recursos ou funções disponíveis para o seu ambiente de trabalho e assim tornar a construção de jogos mais ágil.

3.1.2 Rapidez

Além de proporcionar um editor de jogos simples e partilhado, é fulcral esse editor ter a rapidez necessária, para a plataforma conter uma usabilidade coincidente com o que o utilizador pretende e, assim, oferecer a fluidez essencial para não prolongar o desenvolvimento de um jogo.

3.1.3 *Playfull Design*

Adotar um *Playfull Design* para proporcionar uma atividade lúcida ao utilizador, de modo ajudar na demonstração, envolvência e realização da plataforma fornecida. E assim criar uma melhor experiência geral do consumidor.

3.1.4 Funcionalidade de baixo nível

Sendo que o desafio recaí num editor de jogos mais simples, apresentar funcionalidades de baixo nível acaba por ser uma condição necessária. Pois, apostar em ferramentas mais básicas e essenciais na construção de jogos ajuda a tornar esse editor menos complexo e fácil de compreensão.

3.1.5 Petri Net

Para apresentar um canvas simples capaz de modelar comportamentos num jogo, é um objetivo explorar as redes de Petri como uma linguagem de modelação, um sistema distribuído para definir os comportamentos conforme determinados eventos.

3.1.6 Engines Estudadas

As *Game Engines* estudadas vão servir como modelo deste projeto. Algumas ideias vão ser utilizadas com base nelas. A intenção é reaproveitar algumas implementações dos seus softwares. A ideia é espremer dessas *engines*, funcionalidades que torna uma engine nova mais simples. Deste modo, a Construct2 vai ter um papel mais importante, já que, a sua conceção é próxima à ideia do projeto. Não alvorando as outras três.

3.1.7 Público Alvo

Esta plataforma é voltada para um público com pouca ou sem experiência no ramo de criação de jogos. Os potenciais clientes são os criadores independentes sem orçamento, isto é, criadores que desejam criar demonstrações de jogos e experimentar ideias. Também é propicia para pessoas que surgem uma ideia de um jogo simples para mobile e querem criá-lo, mas não tem nenhuma prática. Em geral, a plataforma procura um público que pretende criar jogos modestos ou experimentar jogos, na qual o plano passar por concretizar esses jogos noutra *engine* mais potente.

3.2 Metodologia *Design Science Research*

Design Science Research é uma metodologia usada para a criação de novos artefactos, com base nas pesquisas realizadas e conhecimentos adquiridos, que se propõe a resolver problemas. Esta metodologia é uma forma mais sistemática para projetar e/ou melhorar artefatos.

Design Science Research dispõe de 5 etapas (figura 3.1) para a produção dos respetivos artefactos, **Awareness of Problem** (relevância do problema), **Suggestion** (sugestão), **Development** (implementação), **Evaluation** (avaliação), e **Conclusion** (conclusão). [51]

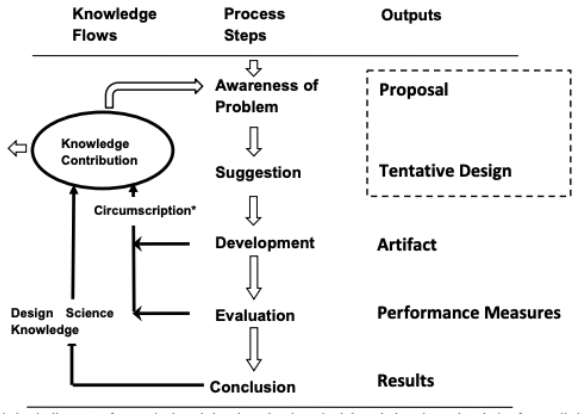


Figura 3.1: *Design Science Research Process Model (DSR Cycle)* [51]

Awareness of Problem, esta primeira etapa serve para pesquisar sobre problema em questão, envolve identificar e definir o problema, para formalizar o assunto pesquisado.

Suggestion, esta é uma etapa de sugerir e apelar à criatividade de novos recursos previstos. Consiste numa proposta de conceito para a solução do problema, isto é, o desenho da arquitetura.

Development, depois da tentativa de uma proposta de conceito, esta etapa é de desenvolvimento e implementação do protótipo como prova de conceito. Esta etapa vai ser representada por um processo iterativo (modelo incremental) [28], onde vai ser adotado uma prototipagem evolutiva, com incrementos/iterações de mês a mês, para implementar a prova de conceito.

Evaluation, esta etapa é de avaliação dos protótipos desenvolvidos. Avaliar o artefacto produzido conforme os critérios delineados e a experiência de utilizadores reais. Esses critérios

vão ser para ensaiar o uso do protótipo, tal com rever o seu design, para melhorar progressivamente o protótipo.

Conclusion, esta etapa final representa o terminar do ciclo de pesquisa e do projeto. Isto significa a conclusão do relatório de progresso e, consequentemente, da dissertação.

3.3 Plano de Trabalho

Nesta secção é a apresentado os aspetos relevantes para o plano de atividades deste projeto. Para o 1.º semestre é descrito as atividades realizadas divididas em tópicos. No segundo semestre, já com a metodologia de desenvolvimento escolhida, é feita uma expectativa de planos de trabalhos de acordo com essa metodologia.

3.3.1 Semestre 1

Neste 1.º semestre temos a elaboração do plano de trabalho (diagrama 3.2), onde é marcado por três etapas. A primeira etapa representa a definição do Estado de Arte. A segunda consiste na definição dos objetivos e escolha da metodologia a adotar, depois da análise aos objetivos e do estudo sobre o tema principal. O último ponto foca-se na avaliação intermédia.

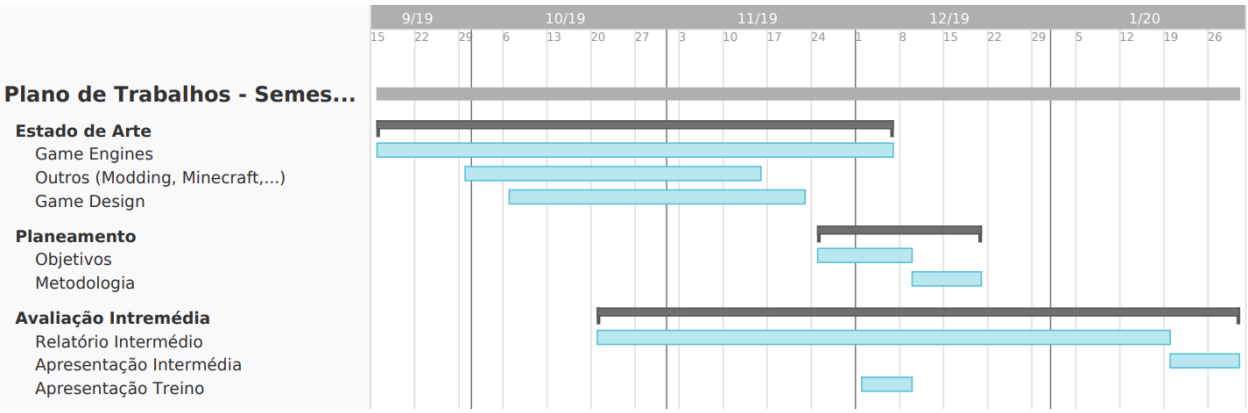


Figura 3.2: Diagrama de Gantt: Planos de Trabalho - Semestre 1

3.3.2 Semestre 2

Neste semestre o plano de trabalho é divido em 4 etapas principais. Segundo a metodologia adotada (*Design Science Research*), a experiência das tarefas realizadas no 1.º semestre, e também pela adoção do método de desenvolvimento iterativo, é feita a expectativa de duração das atividades para o 2.º semestre.

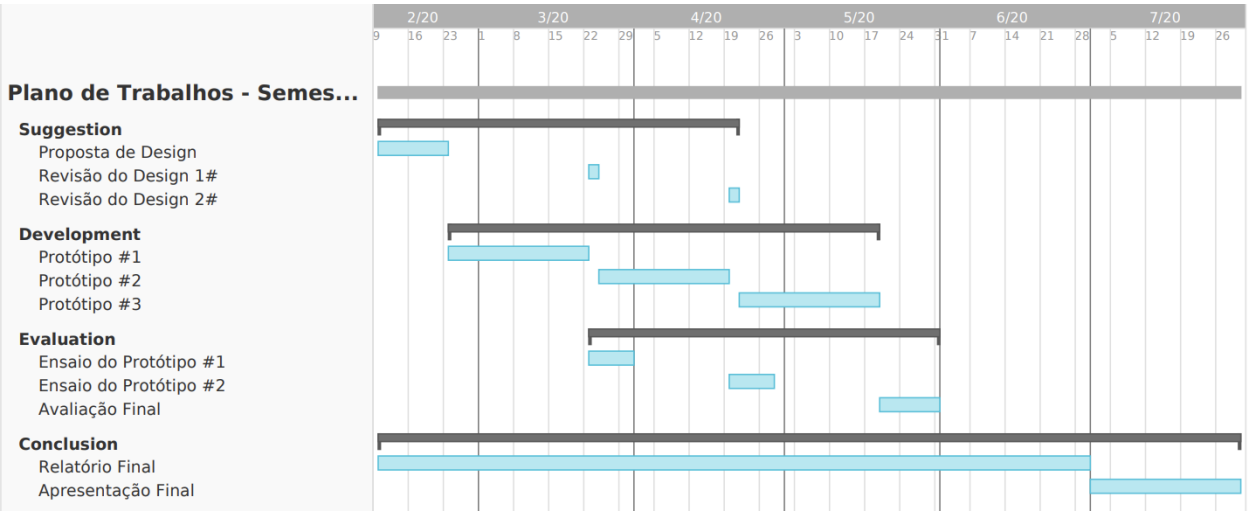


Figura 3.3: Diagrama de Gantt: Planos de Trabalho - Semestre 2 (Expectativas)

Pelo diagrama 3.3, é possível ver as quatro fases destacadas,

Primeira fase (Suggestion), fase de sugestão, onde é feita a proposta de Design com base na pesquisa realizada e nos objetivos delineados.

Segunda fase (Development), é a fase de desenvolvimento, representada pelos protótipos a produzir de mês a mês conforme o método iterativo, é de notar que depois dos dois primeiros protótipos vai haver uma revisão do design e, caso seja necessário, alterações de ajustes.

Terceira fase (Evaluation), são as etapas de avaliação que também será seguida de cada fase de desenvolvimento, onde é feita a avaliação do protótipo, na qual se verifica ou não o que foi traçado para o protótipo.

Quarta fase (Conclusion), temos a conclusão que coincide com o desenvolvimento da dissertação e que irá significar as conclusões finais, além de todo o processo realizado. A conclusão também conta com a apresentação final deste projeto.

3.3.3 Gestão de Riscos

Num projeto de desenvolvimento de um software tem sempre uma componente incerteza, que é colmatada com uma gestão de riscos. A gestão de riscos envolve uma análise de possíveis riscos que podem comprometer o produto final. É considerado um conceito de engenharia de software (ES) e é crucial para uma boa gestão de todo o projeto. [42]

A gestão de riscos ajusta-se em mitigar os riscos, ou seja, contê-los para não causar estragos no projeto. Essa gestão implica identificar os riscos, avaliá-los conforme a probabilidade de ocorrência e o impacto que podem ter no projeto, e consequentemente, um plano de mitigação. [42]

Riscos identificados:

Risco_01. Pouca experiência na metodologia adotada, pode causar más estimativas no planeamento.

Risco_02. Necessidade de haver mais incrementos que o previsto (três incrementações delineadas).

Risco_ID	Probabilidade	Impacto	Plano de mitigação
_01	Baixa	Alta	Ajuda por parte do orientador a definir o planeamento
_02	Média	Alta	Diminuir o tempo de incrementações (menos de mês a mês)

Baixa	($\leq 33.33\%$)
Média	($> 33.33\%$) $\wedge$ ($\leq 66.66\%$)
Alta	($> 66.66\%$)

Tabela 3.1: Riscos

Capítulo 4

Proposta de design

Neste capítulo é esboçada a interação com a plataforma e desenvolvida a proposta de arquitetura do projeto PUMA.

4.1 Conceitos do projeto

Esta secção os elementos básicos do projeto. Estes elementos introduzem os conceitos necessários para construir os jogos. São conceitos que precisam de ser entendidos antes de abordar tópicos mais específicos.

Layouts

Os *layouts* é a representação de todo o cenário exibido na tela do jogo. Servem como plano principal para a construção do jogo incluindo, os menus, os níveis, as caixas de textos, a organização dos objetos e os outros *layouts* de jogo. Os *layouts* também se baseiam em várias camadas (*layers*), de maneira a proporcionar uma melhor organização dos objetos. Assim é possível organizar os objetos nas camadas de primeiro plano e de segundo plano.

Objetos

Os objetos são, por norma, a componente que mais representa um jogo. Estes podem ter diferentes tipos no qual o objeto se baseia. Cada objeto tem a sua própria folha de eventos. São os objetos que vão preencher o cenário e a personagens do jogo. Os objetos podem se agrupar em objetos do mesmo tipo, de modo a organizá-los e assim poder ter as mesmas ações no jogo. Estes objetos estão divididos em duas categorias:

Objetos atores
Objetos que o utilizador pode controlar. Normalmente, estão associadas às personagens principais, onde o jogador tem controlo total mas também podem ser objetos que tem certas ações conforme a interação do jogador.

Objetos cénicos

Objetos que o utilizador pode interagir indiretamente ou apenas estão a decorar o cenário do jogo. Estes objetos são estáticos ou que reagem a determinadas ações do jogo sem serem controladas. São estes que trazem dinâmica ao jogo.

Folha de eventos

Na *PUMA Game Plataform* (PUMA) os eventos são uma forma de contornar a programação ou scripts para construir o jogo. A lista de todos os eventos definem a lógica do jogo. Com base nas redes Petri Net (PN) é definido o evento. Cada layout tem a sua própria folha de evento e estas podem ser reutilizadas e incluídas noutras folhas de evento. Cada elemento do jogo também terá a sua própria folha de evento simples já pré-definida.

Efeitos sonoros

Aqui são ficheiros de áudio que podem ser usados para músicas e efeitos sonoros no jogo. Desta forma, os efeitos sonoros são para eventos mais curtos como no momento de colisões ou explosões, já as músicas são para eventos mais longos como na "intro" do jogo ou mesmo a música do jogo a rodar. É importante ter uma pasta para guardar os ficheiros de áudio de forma a organizar o conteúdo. Será possível adicionar sons através de ficheiros externos e através de uma funcionalidade de criar o próprio som com o auxílio do microfone do dispositivo, além dos sons já pré-existentes.

Unidades essenciais

A PUMA vai necessitar de inserir alguns valores em unidades comuns, como tamanhos, velocidades ou tempos, que são praticamente essenciais nos diversos tipos de jogos. As unidades presentes são descritas em baixo com as suas medições.

Position, para definir as posições de cada elemento. Está em pixels de modo a definir as variáveis do eixo do X e do eixo do Y, conforme o sistema de coordenadas.

Angle, para definir o ângulo de cada elemento. Está em graus de modo a definir a orientação do elemento, incrementando no sentido horário.

Size, para definir os tamanhos dos elementos. Está em pixels de modo definir o quanto se transforma do tamanho original.

Speed, para definir a velocidade de cada elemento conforme cada frame. Está em pixels por frame de modo a definir quantos pixels o elemento se move por frame.

Acceleration, para definir a velocidade que o elemento ganha conforme a cada frame. Está em pixels por frame.

Time, para definir os tempos de ações do elemento. Está em número frames de modo a definir a unidade de tempo.

4.2 Estrututa do projeto

Esta secção descreve a estrutura do projeto, ou seja, as características das componentes do *software*. São os princípios que sustentam a arquitetura deste projeto. Apresentam os elementos básicos da plataforma na qual visa a fornecer funcionalidades para a construção de jogos. É detalhado cada componente da estrutura e as suas propriedades.

4.2.1 Projects (Projetos)

Um projeto na plataforma PUMA é a criação de um jogo completo. Os projetos contêm todos os elementos essenciais para compor o jogo. O ecrã principal mostra a visão geral do projeto. Os elementos do jogo podem adicionados, alterados, removidos e organizados dentro do projeto. No **Project Bar** é possível ver um resumo através de uma árvore com os elementos presentes no jogo.

Os projetos podem ser guardados, fechados e posteriormente abertos. Com a conclusão do projeto é possível exportar para publicar ou o partilhar jogo.

Propriedades

ID, um ID único que identifica o projeto.

Name, nome ou titulo do projeto.

Description, breve descrição do projeto. Um pequeno resumo para demonstrar a ideia do projeto.

Author, nome do autor ou da companhia que está a desenvolver o projeto.

Website, link para publicar o projeto ou para aceder à página web do autor/companhia.

Inicial Layout, primeiro layout aparecer do projeto.

Window size, tamanho da janela de exibição do jogo.

4.2.2 Layouts (Cenas)

O *layout* é a representação dos cenários do jogo, serve tanto para fazer menus, níveis e outras tipos de ecrãs de jogo. Os *layouts* podem ser adicionado e removidos do projeto. Os *layouts* tem a sua folha de eventos agregada. Cada *layout* contêm um conjunto de *layers* onde os objetos pertencem a uma deles e não ao *layout* em si. O nome do *layout* serve como referência para o projeto. A edição dos *layouts* é feita na **Layout View**.

Propriedades

Name, nome do layout.

Layout size, tamanho da janela do layout.

Event sheet, folha de eventos associada a esse layout para definir como ele funciona.

Selected layer, layer selecionado para adicionar ou alterar objetos.

Scrolling, permitir *scrolling* ilimitado, para além das bordas do *layout*.

Margins, tamanho das margens do layout. As margens são a área de volta do layout, área que está de fora do jogo.

4.2.3 Layers (Planos)

Os *layers* são as camadas dentro do layout, estas servem para definir planos de perspetiva diferentes. São usadas para agrupar diferentes objetos e posicioná-los uns à frente dos outros, isto é, uns estão no primeiro plano e outros no plano de fundo. Os *layers* podem ser adicionados, removidos e alterados e podem ser tratadas de maneiras diferentes dentro do mesmo layout de modo a possibilitar efeitos de *parallax*, isto é, efeitos visuais do jogo.

Os *layers* vem com uma organização por *default* comum:
HUD (*heads-up display* - tela de alerta)
Foreground (primeiro plano)
Middleground (plano do meio)
Backgourd (plano de fundo)

Propriedades

Name, nome do *layer*.

Background color, cor do plano de fundo do *layer*.

Transparent, ignora a cor do plano de fundo e torna-a tranparente.

Opacity, opacidade do *layer* de 0 (invisível) a 100 (opaco).

Scale, alterar a taxa de escala do *layer*.

Parallax, alterar a taxa de *parallax* do *layer* nas direções horizontal e vertical.

Visibility, se o *layer* está ou não visivel na tela de edição do *layout*.

Locked, bloquer ou desbloquear o *layer* para a exebição do *layout*. Os objetos não podem ser selecionados se o *layer* estiver bloqueado.

4.2.4 Objects (Objetos)

Os objetos são a parte mais essencial de um projeto. São os elementos que mais representam um jogo, tanto os objetos principais como os figurantes e ainda os objetos ocultos do jogo. Sendo assim, existem diferentes tipos de objetos, logo, ao adicionar um objeto, esse já tem um tipo definido. Aos objetos podem ser atribuídos comportamentos que estes podem ter no jogo e também acrescentar variáveis que não estão definidas para ajudar a compor o objeto (como adicionar uma barra de vida à personagem).

Propriedades

Name, nome de referência do objeto.

Type, tipo de objeto, define uma 'class' do objeto.

Layer, o *layer* onde o objeto está inserido.

Position, posição do objeto, coordenadas do X e Y.

Size, tamanho do objeto, altura e largura.

Angle, ângulo do objeto para onde está orientado.

Opacity, opacidade do objeto, de 0 (invisível) a 100 (opaco).

Variables, adicionar variáveis extra para o objeto.

Behaviors, adicionar comportamentos ao objeto.

4.2.5 Events (Eventos)

Os eventos são um recurso da PUMA para contornar o uso de *scripts* na criação do jogo. Eles são projetados para o utilizador definir toda a lógica do jogo, de um modo simples e intuitivo. Essa definição tem uma maneira particular de executar e esta secção descreve o processo de funcionamento dos eventos.

Os eventos basicamente consistem em ações que devem ser executadas caso certas condições se verificarem e caso haja mais sub eventos que cumprem as condições, mais ações são efetuadas e por aí em diante. Os eventos são editados na Event View. As seguintes partes constituem os eventos.

Propriedades

Evente sheet, são as folhas de eventos que servem para definir o evento. Estas folhas podem ser atribuídas a um layout, incluídas noutras folhas de eventos ou em objetos específicos.

Conditions, são as condições que verificam se determinados critérios estão de acordo. Permite filtrar os objetos que verificam as condições.

Actions, são as ações ou os acontecimentos que ocorrem no jogo.

Expressions, são as expressões para calcular novos valores das variáveis dos objetos.

Sub events, são os sub eventos que procedem a outros eventos. Permite fazer novas condições e, assim, filtrar mais os objetos, e executar novas ações.

Event variables, são as variáveis dos eventos que serve para adicionar e guardar variáveis globais (todos os layouts) ou locais (para uma folha de evento). Essas variáveis ajudam a guardar números, textos ou mensagens (tokens).

Notes, são as notas que servem para o utilizador descrever os eventos ou deixar alguns comentários. Ajuda ao utilizador a organizar-se melhor e a ter um enquadramento do que tem feito.

Mais abaixo é explicado como os eventos funcionam, com mais detalhes e exemplos.

4.2.6 Sound Efects

Os efeitos sonoros são ficheiros de áudio. Na PUMA esses ficheiros podem ser usados para músicas, sons dos objetos, falas, entre outras possibilidades. Esta função permite dinamizar os jogos com várias hipóteses de manipulação de sons. Os ficheiros de áudio podem ser importados para os projetos (ficheiros externos), gravados na própria plataforma (com o recurso do microfone do dispositivo) ou usar sons presentes na biblioteca da PUMA.

4.2.7 External Files

Ficheiros externos podem ser importados para dentro do projeto. Esta funcionalidade é útil para adicionar arquivos que podem ser precisos no projeto que a plataforma não forneça. Esses podem ser vídeos, imagens, sons, músicas, documentos ou outros ficheiros comuns que permite ao utilizador fazer *upload* e acrescentar ao seu projeto. Os ficheiros podem ser adicionados através da opção **Import Files**.

4.3 Design da interface

Nesta secção, é apresentado um esboço inicial do projeto. Esta visão inclui a página inicial, a interface e os principais conceitos do projeto.

4.3.1 Página Inicial

Esta página é um ponto de partida útil para cada vez que se inicia a plataforma. É uma página simples com alguns links úteis, como Criar um novo projeto ou abrir um dos projetos já criados.

Assim, ao abrir o software, em vez de carregar imediatamente o último projeto, abre esta página e dá a opção de abrir um projeto guardado ou de criar um novo.

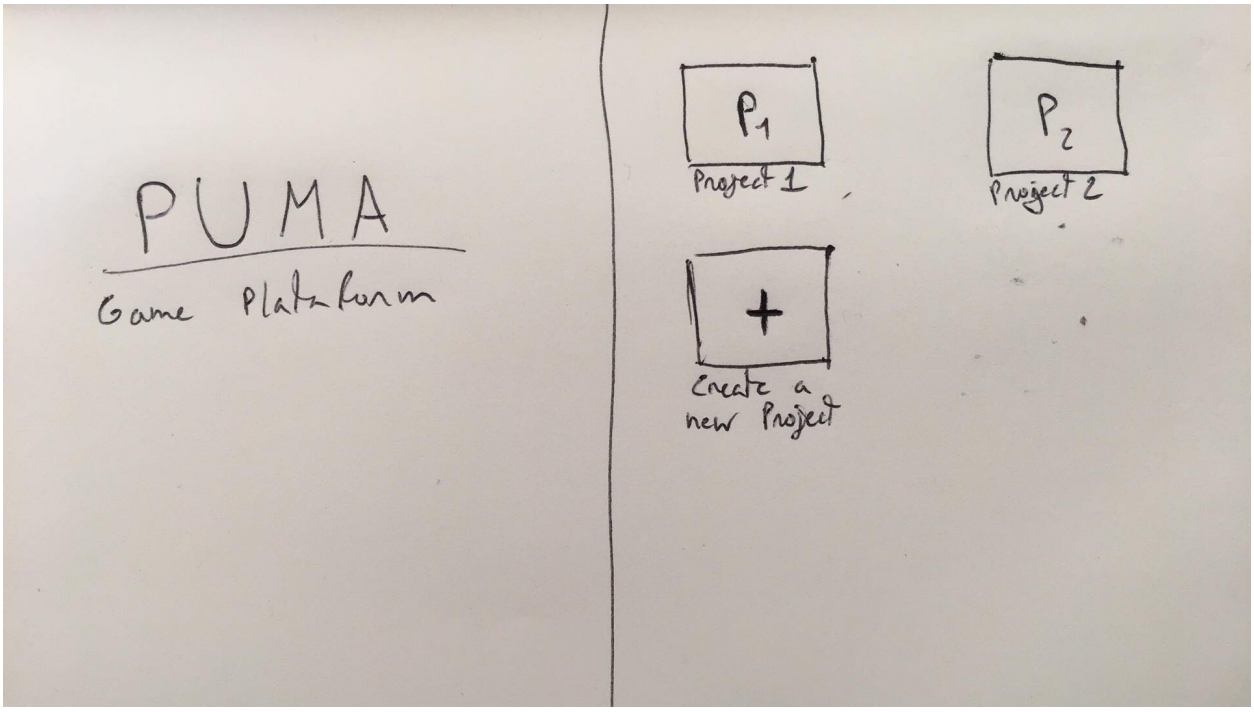


Figura 4.1: Mockup da Página Inicial
Opção de clicar para criar um novo projeto ou abrir um guardado.

4.3.2 Esboços da Interface

A interface do utilizador é destacada pelas seguintes partes. Em baixo é fornecido uma visão geral dessas partes.

1. View Tabs

O View Tabs permitirá trocar entre os diferentes layouts views, de modo a definir as várias lógicas e cenários no jogo. É uma barra que se encontra no topo da interface que contém os layouts do jogo em cada separador. Podendo ser Layouts Views ou Event Views, para definir a lógica conforme os eventos determinados.

2. Layout View

O Layout View é a janela principal para edição e construção do jogo, isto é, o design visual para os seus elementos. Pode ser chamado de palco que vai ser organizado por objetos. Assim permite a configuração de níveis, menus ou telas do jogo.

3. Project Bar

O Project Bar é a visão geral de todo o projeto. Tem como função facultar ao utilizador que consulte a estrutura do projeto, de modo a obter um esquema de todos os elementos que compõem o projeto

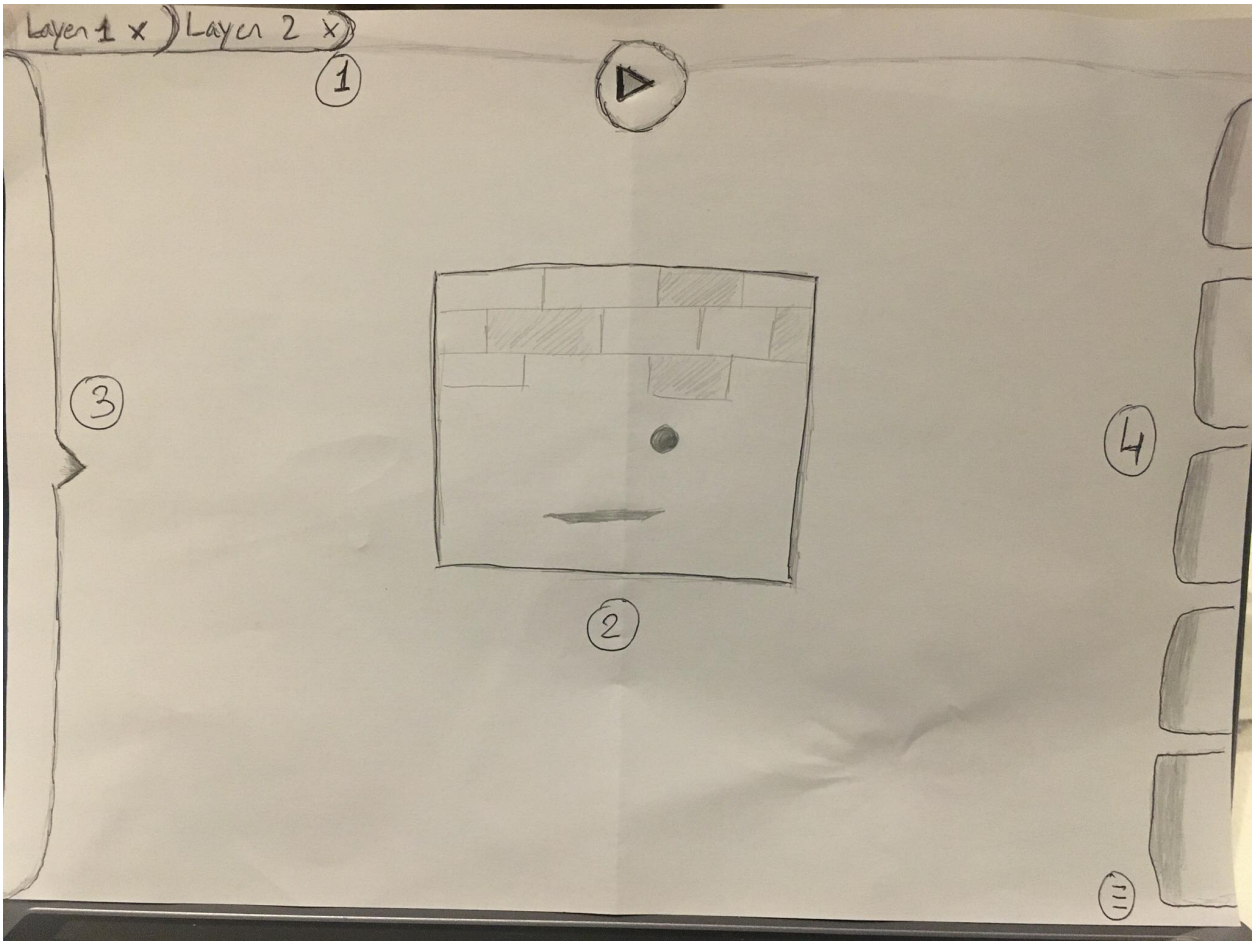


Figura 4.2: Mockup 1 da Interface
Legenda: (1)View Tabs, (2)Layout View, (3)Project Bar, (4)Object Bar.

e assim aceder aos detalhes de cada elemento. Esta aba, normalmente encontra-se fechada, abre-se para verificar objetos difíceis de aceder ou para agrupar a outros cenários. No fundo, é a árvore de cena do projeto.

4. Object Bar

O Object Bar contém a lista de objetos que se pode adicionar ao jogo. É a biblioteca de objetos importados para o projeto utilizando um sistema de drag-and-drop, onde se pode arrastar o objeto para colocar no layout de exibição. Esta aba também se encontra fechada para haver mais espaço para o Layout View, abre-se para adicionar novos objetos ao projeto.

5. Event View

O Event View é a janela para definir os eventos com base nas Redes de Petri NET (PNs). Possibilita novas ações, ou mesmo inibie, conforme as ações que o ator tem no jogo. As redes de definição do jogo utilizam tokens com base na transmissão de mensagens entre os diversos eventos.

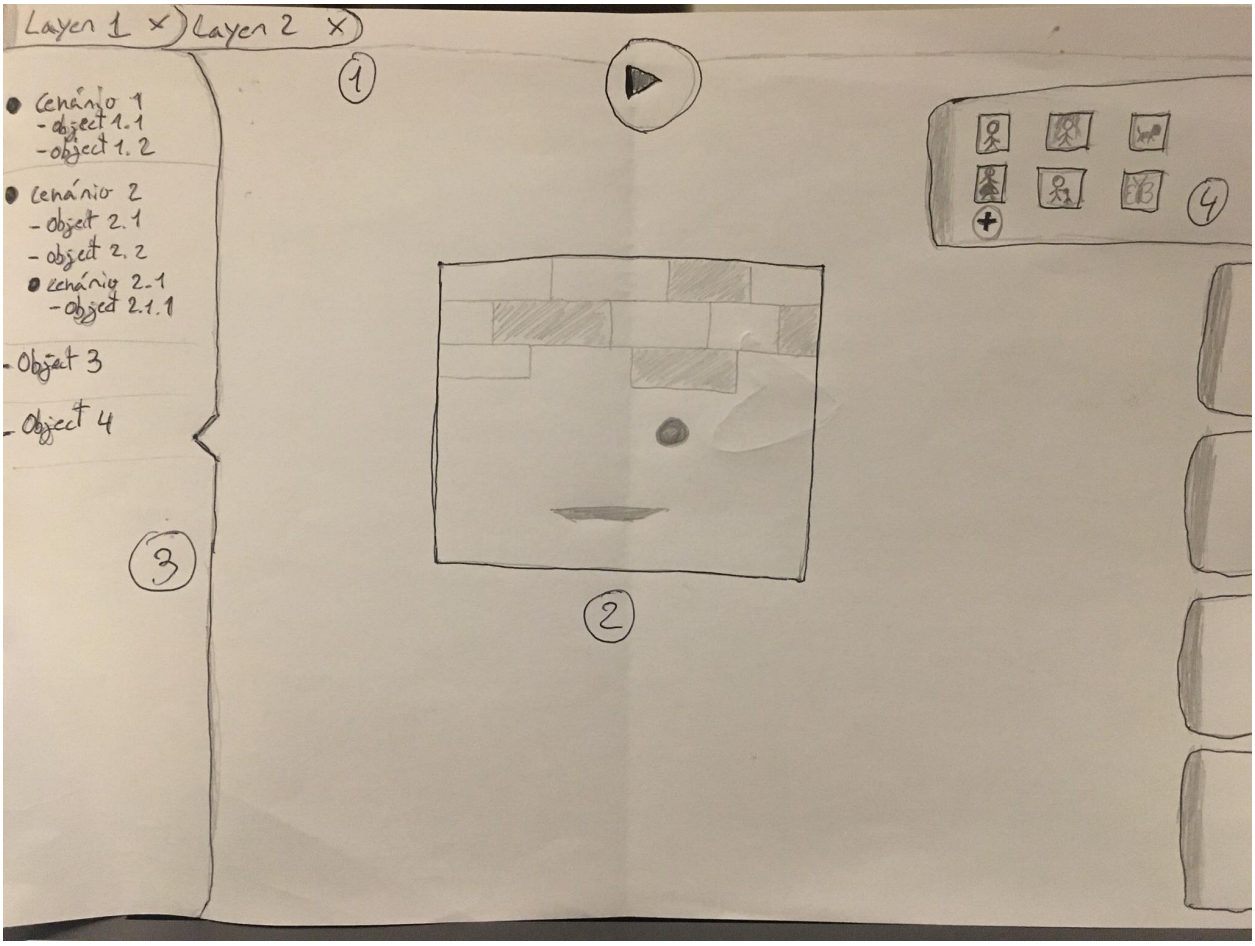


Figura 4.3: Mockup 2 da Interface
Interface com as abas de (3)Project Bar e (4)Object Bar abertas.

4.4 Definição de Comportamentos

Esta secção descreve a maneira de como os eventos funcionam. Esta é a ferramenta principal para criar jogos melhores e mais complexos.

As redes de Petri Net serão o mecanismo usado para definir os eventos que ocorrem no jogo. De modo geral, os eventos vão funcionar da seguinte maneira, é filtrado os objetos que serão alvo do evento através das condições e as ações são executadas nesses objetos já filtrados. Por outras palavras, se as condições do evento forem todas verdadeiras, as ações são executadas nos objetos que verificam essas condições.

Abaixo desenvolvem-se os vários tópicos que caracterizam o mecanismo de processamento dos eventos.

4.4.1 Petri Nets

A forma de modelar o processamento dos eventos será baseado em Redes de Petri. A ideia é aproveitar das Petri Nets a forma simples de montar uma rede de eventos sem entrar em muita complexidade. Mesmo na Petri Nets o objetivo não é aprofundar nessa matéria. A simplicidade vai passar por montar pequenas redes fáceis de perceber, de maneira que no final, com as redes todas definidas, o jogo tenha o seu conjunto de eventos completo e funcional.

As Petri Nets são compostas por dois tipos de componentes: a Transição e o Lugar. A Transição corresponde a uma ação realizada no sistema e o Lugar está associado a uma variável de estado do sistema. A relação entre Lugares e Transições permite realizar determinadas ações. A realização dessas ações é encadeada por pré-condições das variáveis de estado. Depois de uma ação, as informações de certos Lugares são alteradas. Nas redes de Preti as Transições são representados por barras ou traços opados e os Lugares por círculos. [19]

Na PUMA as Transições vão ser representadas pelas Ações e os Lugares pelas Condições. Na componente Condição, os objetos são selecionados através da definição de condições. Verificando essas condições, esses objetos executam as ações definidas, na componente Ação. Como nas redes de Petri, os Lugares precisam de satisfazer certas condições para ativar as Transições, neste projeto significam a mesma coisa. Depois de executada uma Ação há a possibilidade de o evento acabar ou transitar para outra Condição e desbloquear outro processo.

4.4.2 Events Sheet

A folha de eventos é a folha de edição e organização da lista de eventos. Todo processo de construção da lógica de definição e processamento de eventos do jogo será "escrita" nestas folhas.

Cada *layout* terá a sua folha de eventos de modo a definir como esse *layout* funciona, quais as acções que aí serão possíveis e como os objectos aí representados interagem. Também será possível adicionar novas folhas de eventos

4.4.3 Conditions

Como já referido, as condições são adicionadas para filtrar progressivamente os objetos de jogo que irão executar as ações desse evento. Um conjunto de condições são vistas como restrições que os objetos têm de cumprir no total, na lógica é um ou mais 'AND'. Assim, os objetos que cumprem as restrições executam as ações, caso não haja objetos que cumprem as ações não são executadas. Também é possível criar condições com a lógica 'OR' onde só é necessário uma ser verdadeira.

As condições podem selecionar um objeto ou um grupo, mas, em geral, são para verificar se são verdadeiras ou falsas.

Um exemplo de um evento com duas condições:
1º - Todos os carros da tela de exibição são selecionados.
2º - Os carros selecionados são reduzidos para os que estão com velocidade a mais de 10 unidades.

4.4.4 Actions

As ações são fazem alguma coisa acontecer no jogo. Essas ações apenas afetam os objetos que verificam as condições definidas. Uma ação num evento pode ser destruir, criar ou deslocar um objeto.

Dentro de um evento é possível adicionar ações, de um lado tem as condições desse evento e do outro as ações. Um evento pode ter várias condicionantes como várias ações.

Seguindo o exempo acima: **3º** - Os carros selecionados reduzem em 5 unidades de velocidade.

Assim, os carros que vão a mais de 10 de velocidade são reduzidos para uma velocidade de menos 5. Os que vão menos de 10 de velocidade não acontece nada.

4.4.5 Expressions

Nos eventos, as expressões são usadas para calcular os valores das variáveis ou obter informação dos objetos. As expressões podem ser inseridas tanto nos parâmetros das condições como das ações. Nas condições serve mais para calcular valores e compara-los, já nas ações serve mais para atribuir esses valores calculados.

As expressões podem ir de um simples número a um cálculo mais complexo. Esses valores podem ser retirados de variáveis do jogo ou de funções.

Para as expressões são usados:

Numbers
São dígitos numéricos, podem ser do tipo int (inteiros) ou float (decimais).

String
Sequência de caracteres.

Calculation operators
Operadores de cálculo:
$+$ (adição)
$-$ (subtração)
$*$ (multiplicação)
/ (divisão)
% (módulo)
$\wedge$ (expoente)

Comparison operators
Operadore para comprar valores: $=, !=, <, <=, >, >=$. Se a comparação for verdadeira retorna true, caso contrário retorna false.

Object attributes
Os objetos têm os suas próprias variáveis, ou seja, atributos para guardar informações sobre eles. Esses atributos servem com GET e SET do objeto, onde é possível obter e atribuir informações aos objetos. Essas variáveis são escritas desse modo Object.X (neste caso, a coordenado X do objeto).

4.4.6 Event variables

As variáveis de evento são para guardar informações dos eventos, valores ou textos. Podem ser variáveis globais do projeto ou locais para folha de eventos particulares.

Há uma secção propositada para adicionar as variáveis nos eventos. As variáveis são modificadas ao longo do recorrer das ações. Essas variáveis tem um nome e podem ser acedidas através do nome único de cada uma.

Variáveis globais
As variáveis globais guardam informações entre todos os layouts do jogo. Podem ser acedidas por qualquer *layout*, tanto para retirar informação como para alterar, mesmo que tenha sido criada noutro *layout*.

Variáveis locais
As variáveis locais são usadas apenas no próprio *layout* ou folha de eventos extra. Só esse *layout* é que podem interagir com essas variáveis. Este tipo serve para guardar informação apenas referente a esse *layout* ou como variáveis temporárias. Também ajuda a não encher o projeto cheio de variáveis globais, tornando o projeto mais simplificado e fácil de entender.

Também é possível marcar as variáveis como constantes, de modo a não sofrerem alterações no seu valor. Elas são estão disponíveis para leitura e ajuda a relembrar ao utilizador o fim delas.

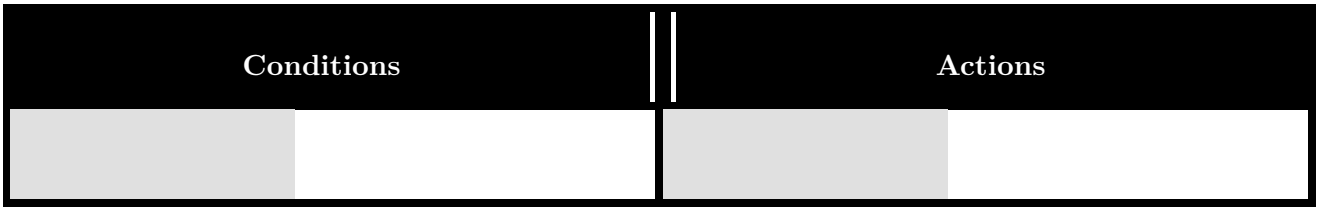
4.4.7 Sub-events

Os sub-eventos ocorrem logo a seguir a um evento. Basicamente, existe o evento pai e o evento filho, o filho é o sub evento que só é realizado caso o evento pai tenha sido executado as suas ações, ou seja, o pai tem de verificar as suas condições, realizar as ações do evento e só a seguir é que entra o evento filho (sub evento). Depois o sub evento funciona como um evento normal tem as suas condições e ações. Desta forma os sub eventos também podem ter outros sub eventos. Este mecanismo torna o processo de criar jogos mais flexível.

Os sub eventos podem ser conjugados de várias formas, como ter dois eventos pais para o mesmo filho ou um pai para três filhos. São possibilidades que fazem com que o jogo tenha rumos, caminhos ou histórias diferentes, conforme os eventos que vão ocorrer.

4.4.8 Como criar um evento?

Um evento é composto por duas secções: das condições e das ações. Pela tabela abaixo é possível ter uma ideia de como é a composição visual, onde um lado do bloco serve para compor, as condições e do outro, as ações.



Essas duas secções permitem adicionar uma condição ou ação no evento. O processo de adicionar

condições e ações é praticamente o mesmo.

A composição de um evento centra-se nesses 3 pontos:

1. Selecionar um objeto para a condição ou a ação (na parte que está a cinzento na tabela).
2. Selecionar a condição ou a ação para esse objeto (na parte que está a branco na tabela).
3. Adicionar nova condição ou ação, caso seja necessário.

Selecionar objetos

A seleção dos objetos, que vão ser alvos das condições ou das ações, é feita através de lista onde é listado todos os tipos objetos dentro do projeto. Os tipos de objetos estão associados à matéria toda que envolve o jogo, como o próprio Sistema do jogo, o Teclado, o Áudio, e todos os objetos adicionados. Para exemplificar, o Sistema pode englobar funcionalidades como escolher aleatoriamente objetos, enquanto o Teclado engloba receber inputs de interação por parte do jogador.

Selecionar condições ou ações

A seleção de condições ou ações funciona da mesma maneira, há uma lista com todas as opções disponíveis, de modo a montar a condição/ação como desejada.

Adicionar novas condições ou ações

Em certos casos que é necessário várias condições ou ações para compor o evento, há a opção de adicionar novas condições/ações e proceder da mesma maneira, seguindo os passos anteriores.

Condições e Ações

Na composição das condições e das ações a PUMA oferece já um conjunto de funções e possibilidades para montar o evento. Essas funções podem precisar de adicionar ou de aceder parâmetros, como variáveis de objetos ou do próprio jogo. As funções são mecanismos da criação de jogos como uma função para criar, destruir ou calcular a colisão de objetos, por exemplo.

4.5 Resumo da Concepção

O desenvolvimento da proposta de design passou por algumas etapas preliminares, tais como, a programação de uma versão para esses quatro jogos mais básicos, Space Invaders, Pac Man Super Mario e Lemmings, o estudo aprimorado dos manuais das Game Engines estudas e dos moddings mais utilizados no Minecraft. Essa etapa ajudou a montar a arquitetura com base nas ideias determinadas. A construção desses jogos ajudou a perceber alguns elementos vitais, que são essenciais em maior parte de jogos e aliado ao conteúdo dos manuais dos software e dos moddings, foi elaborado o design.

O design centra-se em componentes muito básicas na criação de jogos. Os layouts e objetos são os dois pontos primitivos, pois um jogo começa por representar um ou mais cenários com as suas peças e/ou atores. Depois, tem os layers para facultar planos diferentes e dar várias perspetivas ao palco. Por fim, os eventos vêm dinamizar toda enredo montado, com movimento de peças, alterações de cenários, interações do utilizador, entre outras alternativas.

Capítulo 5

Prova de Conceito

Neste capitulo é elaborada uma prova de conceito. No que consiste a prova, o que procura assegurar e como a vai consagrar. Na consagração é realizado um teste com um exemplo prático.

5.1 Qual o conceito a provar?

Num ambiente de desenvolvimento de software, por vezes, torna-se arriscado tentar transformar a teoria em prática. Uma potencial ideia de um projeto pode não ser assim tão extraordinária quando é posta em prática. Sendo assim, a aplicação de uma prova de conceito é uma forma de obter uma noção de como um projeto será executado na prática. [15]

Uma prova de conceito procura evidenciar, em forma de documentação, que um software vai ter o sucesso esperado. Neste caso, procura provar que a proposta de design desenvolvida tem a competência para os objetivos traçados. Principalmente, para provar o mecanismo de definição de eventos no jogo, já que é a segmento fulcral para diferenciação dos softwares existentes.

Trata-se de um método para testar e, sobretudo, para avaliar a ideia do design para este software. Avaliar no sentido em que o conceito é executável e válido. A prova consiste numa pseudo implementação, também consiste numa definição das melhores práticas a aplicar no software.

No fim de contas, este mecanismo permite determinar se o rumo do projeto é o mais correto e se as decisões tomadas são as mais acertadas.

5.2 Como aplicar a Prova de Conceito?

Como forma de testar o design desenvolvido, a Prova de Conceito irá incidir na construção de um jogo com base nos conceitos identificados na proposta de arquitectura. Todo o processo envolvido no desenvolvimento do jogo, a começar pela definição do jogo através da interface e das práticas necessárias para a sua construção, as componentes da arquitetura associadas a essas definições, e por fim, no código fonte a gerar para o jogo.

5.3 Primeiro Teste - Prototipagem do Space Invaders

Esta secção é a composição da prova de conceito. O ensaio consiste na simulação da construção de um jogo simples com base nos elementos da arquitectura, fazendo a prototipagem de um jogo classico. Tomamos como primeiro exemplo o *Space Invaders*. Começamos com uma breve descrição do jogo utilizado. De seguida, definem-se os procedimentos dentro da plataforma para a sua construção. Por último, a composição do código resultante dos conceitos aplicados.

5.3.1 Space Invaders

Space Invaders é um jogo de arcade lançado de 1978. É um *classic game* e um dos primeiros *shooters*. O objetivo do jogo é destruir um conjunto de naves invasoras (Invaders) com uma nave canhão (Shooter) que dispara lasers de forma a ganhar o máximo de pontos e impedir que cheguem á base. [48]

Figura 5.1: Space Invaders

Os controlos do Space Invaders são bastantes simples. Usa as teclas direcionais para movimentar a nave principal (Shooter) para direita e esquerda e com a barra de espaço para disparar.

Na parte superior, as naves aliens (Invaders) estão organizadas em linhas. O jogador tem que evitar que os aliens atinjam a sua nave. A nave principal tem munições infinitas e os aliens disparam aleatoriamente, tendo o jogador que desviar para não perder uma vida. Os aliens têm pontuações diferentes: as duas primeiras filas dão, 10 pontos, a terceira e quarta, 30 pontos, e a quinta, 40 pontos.

5.3.2 Definição do Jogo

Nesta parte é demonstrada a forma de definir um jogo seguindo os pergaminhos do design proposto.

Project

Para iniciar a construção deste jogo, primeiro é preciso criar um novo projeto. Atribuir um nome ao projeto. Definir o tamanho da janela do jogo como 600x450. Posteriormente, definir qual é primeiro layout do jogo (esse layout é automaticamente gerado), layout inicial.

Layouts

Para este exemplo de jogo é preciso 3 layouts, um como layout de início e primeiro aparecer, outro como layout da jogabilidade e por último o layout do fim do jogo, quando se ganha ou perde.

Estes serão os três layouts do jogo:

1. Layout inicial (na figura 5.2)
2. Layout principal (na figura 5.3)
3. Layout final (na figura 5.4)

Layout Inicial

Este layout apenas server para iniciar o jogo. Tem um texto no meio da tela a dizer "Click to start". É apenas uma página para iniciar o jogo quando o jogador clicar no texto. Como só tem um layer que corresponder ao próprio layout, não é necessário adicionar layers neste layout. Sendo assim, somente uma caixa de texto é adicionado ao layout.

Figura 5.2: Layout Inicial

Como tal, este Layout apresenta apenas um evento para a transitar para o Layout Principal no momento que o jogador clicar no rato e, assim, começar a jogar.

Mouse	On click	**System**	Go to layout **Layout Principal**

Layout Principal

No layout principal é onde o jogo se vai desenrolar. É aqui que os objetos e os eventos do jogo são adicionados. Este layout tem dois layers, um para mostrar o score (zona a amarelo na figura abaixo) e o número de vidas restantes e outro para a camada do jogo (zona a vermelho na figura abaixo).

Figura 5.3: Layers do Layout Principal

Layer de informação

Este é um layer HUD (heads-up display - tela de alerta). Esta camada aparece em primeiro lugar e é utilizada para fornecer as informações de estado ao jogador, a informação dos pontos que tem no jogo (score) e do número de vidas. Para montar este layer é preciso adicionar uma caixa de texto para o score e outra para o número de vidas. Para representar as vidas também podem ser usados imagens ou objetos (como corações).

Layer principal

Layer que contém todos os elementos do jogo. Todos os objetos que compõem o jogo são arrastados ou inseridos neste layer. Este layer está no plano abaixo do layer de informação para as imagens do jogo não sobrepor às informações.

- **Objects**

Neste jogo são necessários 3 tipos de objetos, estes são:

Shooter, a nave que é controlada pelo jogador que tenta destruir todos os invasores (invaders).

Invaders, os invasores ou os aliens que tentam destruir a nave principal (shooter).

Bullet, as balas que aparecem quando o shooter ou os invaders disparam.

- **Events**

Os eventos essenciais para este jogo estão representados neste layout. Definem toda a lógica e comportamentos do jogo.

A lista de eventos:

Keyboard	On **RIGHT** pressed	**Shooter**	Move **6** to right

Se o jogador estiver a pressionar a tecla RIGHT do teclado o Shooter move-se 6 pixeis para a direita.

Keyboard	On **LEFT** pressed	**Shooter**	Move **6** to left

Se o jogador estiver a pressionar a tecla LEFT do teclado o Shooter move-se 6 pixeis para a esquerda.

Keyboard	On **SPACE** pressed	**Shooter**	Creat Object **Bullet**(**Shooter.X**, **Shooter.Y** + **Shooter.width** /2)
Shooter	Is not **Bullet**		

Se o jogador pressionar tecla SPACE do teclado e se não exitir uma bala criada pelo Shooter, o Shooter dispara uma bala, ou seja, cria uma Bullet no centro da sua posição.

Shooter Bullet	On colision with **Invader**	**Invader**	Destroy
		Bullet	Destroy
		Shooter	Add **Invader Points** to **Score**

Se a Bullet disparada pelo Shooter colidir com um dos Invaders, a Bullet e o Invader são destruidos e os pontos desse Invader são somados ao score do jogador.

Invader Bullet	On colision with **Shooter**	**Shooter**	Subtract **1** from **Lives**
		Bullet	Destroy

Se a Bullet disparada por um dos Invaders colidir com a Shooter, a Bullet é destruida e o Shooter perde uma vida no jogo.

Invader		**Invader**	Move **4** to **Direction**

Os Invaders move 4 pixels de acordo com a sua variável direction, para RIGHT ou LEFT (a cada frame).

System	If **Invader.X** upper **(Width - 20)**	**Invaders**	Set **RIGTH** to **Direction** Move **15** to down

Se um dos Invaders estiver posicionado na variável X acima de width - 20 pixeis, todos os Invaders movem 15 pixel para DOWN e muda a variavél direction dos Invaders para RIGHT.

System	If **Invader.X** under **20**	**Invaders**	Set **LEFT** to **Direction** Move **15** to down

Se um dos Invaders estiver posicionado na variável X abaixo da 20 pixeis, todos os Invaders movem 15 pixel para DOWN e muda a variavél direction dos Invaders para LEFT.

Invader	Select randomly 1 **Invader**	**Invader**	Creat Object **Bullet**(X, Y + Width /2)

Seleciona um Invader aleatoriamente para disparar uma Bullet.

System	If **Lives** equal **0**	**System**	Go to layout **Layout Final**

No momento que o jogador perder as 3 vidas, o jogo termina e muda para o Layout Final.

Layout Final

O final do jogo acontece quando o jogador destrói todos os Invaders ou perde as três vidas. Neste layout serve apenas para mostrar o fim do jogo (Game Over).

Figura 5.4: Layout final

5.3.3 Componentes de código

Neste segmento, é explorada a componente do código que seria originada depois da definição do jogo na plataforma final.

Este exemplo do código é baseado na linguagem *Processing* por ser suficiente para este propósito, fácil de compreender e frequentemente utilizada para composição de exercícios de design. Outra

linguagem (e biblioteca) poderia ser usada como padrão para a geração do código, pois o processo e a estrutura de criação de jogos ou animações poderia ser similar.

O código gerado consiste em duas partes fundamentais: **Setup** e **Draw**. O Setup é executado apenas uma vez, serve como preparação do cenário. Já o Draw está em constante execução e constitui um loop de atualizações do jogo.

```
public void setup() {
  // setup codes goes here
}
public void draw() {
  // draw codes goes here
}
```

Setup

Quando o programa (jogo) é iniciado, este bloco é executado em primeiro uma única vez. Neste bloco, são compostas as configurações iniciais como inicialização de variáveis ou definição do tamanho da tela de jogo.

Draw

É neste bloco onde o todo o cenário é desenhado. Este bloco que a lógica do jogo atua. Desde o início do jogo até ao fim está num loop constante, onde a cada iteração é um novo frame. De frame em frame os objetos do jogo sofrem alterações e são desenhados conforme esses eventos.

Game Screen

Os cenários de jogo representam os *Layouts*. Neste sentido, cada *gameScreen* é um *Layout* e apenas um é chamado pelo bloco Draw para desenhá-lo. É definido um método diferente para cada cenário a ser ilustrado. No Draw é verificado qual o *gameScreen* a ser chamado. No exemplo em baixo, tem 3 Layouts (initScreen(), gameScreen() e gameOverScreen() e uma variável (gameScreen) para verificar qual deles vai ser exibido.

```
int gameScreen = 0;

/********* DRAW BLOCK *********/

void draw() {
  // Display the contents of the current screen
  if (gameScreen == 0) {
    initScreen();
  } else if (gameScreen == 1) {
    gameScreen();
  } else if (gameScreen == 2) {
    gameOverScreen();
  }
}


/********* SCREEN CONTENTS *********/

void initScreen() {
  // codes of initial screen
}
void gameScreen() {
  // codes of game screen
}
void gameOverScreen() {
  // codes for game over screen
}
```

Draw Game Screen

Cada Layout tem os seus Layers associados. Ao executar a função de cada game screen (do Layout) os Layers são constituídos dentro da função por ordem descrescente, isto é, o Layer que está no topo do plano é o último a ser desenhado e o Layer que está na última camada (fundo do plano) é o primeiro e os Layers do meio são desenhados por essa lógica. No que lhe concerne, os Layers tem os seus objetos associados, ou seja, é na sua extensão onde esses objetos são desenhados.

Class

Os Objects são representados em classes. Nessas classes estão as variáveis e os métodos que ajudam no processo do Draw. Cada um dos objetos vai ser desenhado conforme as suas variáveis do momento.

Inputs

Métodos de interação por parte do utilizador, como interações com o teclado ou com o rato. Esses métodos são executados no momento da interação e realizam a sua função para cada um desses

eventos. Alguns dos métodos é keyPressed ou mousePressed, que verifica qual a tecla que foi pressionada ou se o rato foi clicado.

```
/********* INPUTS *********/

void mousePressed() {
  // These codes will be executed once, when mouse
  // is clicked. Note that mouseButton variable is
  // also be used here.
}


void keyPressed() {
  // These codes will be executed once, when a key
  // is pressed. Note that key and keyCode variables
  // are also usable here.
}
```

Space Invaders

Neste segmento, com o exemplo de jogo, Space Invaders, é mostrado a componente de código resultante com base na definição desse jogo. Esta amostra passa por transcrever o resultado do produto composto na plataforma PUMA, ou seja, o artefacto final após a exportação do jogo.

A demonstração é feita pelas seguintes etapas de construção.

1. **Inicialização**

O primeiro passo é a inicialização que corresponde ao Setup. Neste parte é definido o tamanho da tela e as classes de objetos. Também é inicializado todos os objetos e variáveis envolventes no jogo.

No jogo há 4 tipos de objetos o Shooter, Shooter Bullet, Invaders e Invader Bullets, sendo assim, é necessária a composição das classes para cada objeto.

Class Shooter
x y width height bullet

Class Shooter_bullet
x y width height

Class Invader
x y width height bullet

Class Invader_bullet
x y width height

A configuração do tamanho da tela é de 600x400 como definido na plataforma. São inicializados os objetos. Para os Invaders e Invader Bullets é preciso definir como uma lista por serem vários elementos, já no Shooter e Shooter Bullet só é precisa uma única instância de cada. A variável gameScreen é inicializada para gerir os ecrãs de jogo e as variáveis, score, lives e direction foram adicionadas ao montar o jogo na plataforma (são variáveis extra para os eventos).

```
size(600, 450)

initialize shooter as a Shooter
initialize shooter_bullet as Null
initialize Invaders as a list of Invader
initialize Invader_bullets as an empty list of Invader_bullet

gameScreen = 0
score = 0
lives = 3
direction = RIGHT
```

2. Telas de Jogo

Neste passo é configurado o manuseamento das telas de jogo (Layouts). Na variável para verificar qual é tela a ser exibida é atribuído o identificador do Layout Inicial. No Draw são estabelecidas as verificações (if's) para decidir qual das telas irá ser desenhada. Este jogo tem 3 verificações para o Layout Inicial, Principal e Final. Cada Layout tem a sua função que é chamada após a verificação da tela de jogo atual, essa função corresponde à composição do cenário conforme esse Layout.

Estas são as funções correspondentes a cada Layout:

```
function draw:
  if gameScreen == 0 than
    call layout_inicial()

  if gameScreen == 1 than
    call layout_principal()

  if gameScreen == 2 than
    call layout_final()
end draw

function layout_inicial:
  // codes of layout final

end layout_inicial

function layout_principal:
  // codes of layout final

end layout_principal

function layout_final:
  // codes of layout final

end layout_final
```

Os próximos passos são as chamadas de cada função destes Layouts.

3. Layout Inicial

No início do jogo a variável aponta para o Layout Inicial, ao executar a função Draw, esta, por sua vez, executa a função layout_inicial(). Nessa função vai correr os métodos estabelecidos no seu Layer associado e os eventos determinados no Layout.

```
function layout_inicial:
  /** Layer 1 contents ***/

  /** Events Contents ***/

end layout_inicial
```

Nos métodos do Layer, é definido o backgroud como preto e inserido o texto, “Click to start”.

```
/ * ** Layer 1 contents * * */

backgroud(0)

draw Text "Click to start"
```

Neste Layout apenas tem um evento que quando o rato é pressionado passa para o Layout Principal. Para tal, é definido uma condição que verifique essa ação por parte do utilizador e em caso de cumprir muda a variável.

Events Contents:

Event 1

```
if mousePressed than
  gameScreen = 1
```

4. Layout Principal

No momento que o jogador clica para jogar, este Layout passa a ser exibido pelo jogo. Na chamada deste Layout por parte do Draw, a função executa os métodos delineados pelo Layer Principal, Layer de Infomação e lista de eventos. O Layer Principal é executado primeiro, de modo, essa camada ficar por baixo do Layer de Informação.

```
function layout_principal:
  / * ** Layer Principal contents * * */

  / * ** Layer de Informação contents * * */

  / * ** Events Contents * * */

end layout_inicial
```

Nos métodos do Layer Principal, é definido o background e ilustrados todos os objetos presentes no momento, o Shooter, os Invaders e as Bullets. Se alguns dos Invaders forem destruídos essas instâncias serão apagadas para não serem redesenhadas. No caso das Bullets, só aparecem se um dos aliens ou o Shooter dispararem, e a instância também é apagada no momento da colisão ou por ultrapassar os limites da tela.

```
/ * ** Layer Principal contents * * */

backgroud(0)

draw shooter

draw shooter_bullet

for each invader in list Invaders
  draw invader
end of loop

for each invader_bullet in list Invader_bullets
  draw invader_bullet
end of loop
```

Nos métodos do Layer de Informação, é produzido o texto para mostrar o score e o número de vidas restantes.

```
/ * ** Layer de Informação contents * * */

backgroud(0)

draw Text "Score: "+ Score

draw Text "Lives: "+ Lives
```

Na lista de eventos deste Layout são transcritos todos os eventos definidos pela plataforma. Cada condição e ação de um evento corresponde a um método pré-definido.

Events Contents:

Event 1

```
if keyPressed and keyCode == RIGHT than
  Shooter.X + = 6
end of if
```

Event 2

```
if keyPressed and keyCode == LEFT than
  Shooter.X - = 6
end of if
```

Event 3

```
if keyPressed and keyCode == SPACE and not Shooter.bullet than
  bullet = new Bullet(Shooter.X,Shooter.Y+Shooter.width/2)
  add bullet to list Bullets
end of if
```

Event 4

```
for each invader in list Invaders
  if invader colides with bullet than
    remove invader from list Invaders
    shooter.bullet = null
    shooter.score + = invader.points
  end of if
end of loop
```

Event 5

```
for each invader in list Invaders
  if invader.bullet colides with shooter than
    remove invader from list Invaders
    lives - = 1
    invader.bullet = null
  end of if
end of loop
```

Event 6

```
for each invader in list Invaders
  if Direction == RIGHT than
    invader.x + = 4
  else
    invader.y + = 4
  end of if
end of loop
```

Event 7

```
for each invader in list Invaders
  if invader.x > (width -20) than
    for each invader2 in list Invaders
      invader2.y + = 15
    end of loop
    direction = RIGHT
    break loop
  end of if
end of loop
```

Event 8

```
for each invader in list Invaders
  if invader.x < 20 than
    for each invader2 in list Invaders
      invader2.y + = 15
    end of loop
    direction = LEFT
    break loop
  end of if
end of loop
```

Event 9

```
invader = get rendomly 1 element from list Invaders
invader.bullet = new Bullet(invader.x, invader.y + invader.width/2)
```

Event 10

```
if lives == 0 than
  gameScreen = 2
```

5. Layout Final

No fim do jogo é este o Layout a ser exibido, o cenário é praticamente idêntico ao do início. Sendo assim, a função Draw executa a função layout_final e, neste caso, a função só vai correr os métodos do Layer já que este Layout não apresenta uma lista de eventos.

```
function layout_inicial:
  /** Layer 1 contents ***/

end layout_inicial
```

No Layer do ecrã de fim de jogo, contém apenas a definição do background da tela e a caixa de texto para “Game Over”.

```
backgroud(0)

draw Text "Game Over"
```

5.4 Resumo do Ensaio

Com base neste ensaio de prova de conceito pudemos verificar a exequibilidade da definição e construção de um jogo classico relativamente simples com base nos conceitos definidos na proposta conceptual. Foi ainda possível verificar a estrutura de código a ser gerada com base na definição do jogo exemplo. Este exemplo não será exaustivo mas permitiu realizar um primeiro teste que nos permite dar mais um passo com confiança no sentido de afinar a definição da plataforma PUMA.

Capítulo 6

Desenvolvimento

Neste capitulo é feita uma apresentação da plataforma concebida. São evidenciados o propósito, estrutura, fundamentação e alguns exemplos demonstrativos de como a engine deve funcionar como suporte para a definição e prototipagem de um game design. Apresenta-se uma linguagem de markup para a definição rápida de cenários de jogo e faz-se novo ensaio de viabilidade com base num jogo do género de aventura gráfica.

6.1 A Definição de uma Linguagem para a PUMA

No sentido de concretizar a utilização da arquitectura proposta e estudada no capítulo anterior propõe-se aqui a definição de uma linguagem de marcação para a definição de cenários de jogo. A linguagem PUMA é uma Markup (linguagem de marcação) desenvolvida com o intuito de prototipagem rápida. A linguagem define um conjunto de elementos comuns da estrutura e envolvente do conteúdo do jogo. Tem como principal função permitir descrever a representação, a aparência e os comportamentos de um jogo na plataforma PUMA.

A essência desta linguagem é ser um método de montar jogos de baixo nível de complexidade de forma bastante rápida, assumindo um conjunto de elementos funcionais frequentemente utilizados. É assim composta por elementos fundamentais para aplicar no documento de definição do jogo.

A linguagem PUMA procura ser acessível e fácil de aprender. Esta pode ser uma maneira invulgar de montar novos jogos, mas pensamos que tem potencial para inicar um caminho/rumo alternativo dos sistemas que permitem desenvolver videojogos, sobretudo numa fase de teste de conceitos onde a velocidade e facilidade de efectuar alterações é essencial. Adotando esta linguagem de marcação pretende-se proporcionar uma abertura no acesso à definição e ensaio de videojogos a não-programadores.

6.1.1 Pré-requisitos

Assume-se aqui que o utilizador desta linguagem terá um conhecimento essencial dos elementos utilizados na framework de definição de um videojogo. Não deverá ser necessário um conhecimento preliminar da linguagem, apenas alguma familiaridade básica com linguagem de marcação, como

as utilizadas para a paginação web (ex. HTML). Na construção de jogos será necessário entender como a arquitetura base da PUMA ajuda no processo de desenvolver a definição do jogo, quais os conceitos e como podem ser explorados, além de que deverá ser preciso saber como criar e organizar os ficheiros num projeto de jogo.

6.1.2 Conceitos Fundamentais

Um dos princípios da PUMA é proporcionar uma semântica adequada para que um jogo possa ser construido de modo correto, conforme o design proposto. Dessa forma, é conveniente saber estruturar um documento destes ou, por outros termos, saber adicionar layouts, layers, objetos e eventos. Para tal, será preciso entender como marcar um documento na criação de um jogo para lhe dar a estrutura e o significado desejado.

A linguagem aqui proposta contém uma coleção de elementos base, essenciais na construção de um jogo. Qualquer outra marca ou elemento deverá ser um complemento dos elementos pré-definidos. Cada elemento define o seu próprio nível de bloco para determinar a sua área no jogo, tais como, a área de documento, cenário, camada, objeto ou evento.

Bloco de jogo: <puma>

Denominada de grande área, contém todo o conteúdo presente num jogo. Marca o início e o fim do documento, estabelece o tamanho da tela de jogo e, caso necessário, acrescenta variáveis globais.

Bloco de layout: <layout>

Grandes divisões que correspondem a cada cenário do jogo. Um jogo pode ser composto por um ou vários cenários, onde cada cenário é montado individualmente. Um cenário tem de ter os seus próprios layers, objetos e eventos. Na marcação de um layout é possível adicionar variáveis extra (variáveis locais).

Bloco de layer: <layer>

Área para marcar um plano dentro de um cenário (layout). Num layout podem haver vários cenários ou, simplesmente, não ter nenhum. Dentro de cada layer são definidos os objetos que pertencem a esse plano ou uma imagem correspondente. Este bloco serve para colocar uma ordem em que os vários layers são desenhados para fornecer sensações de profundidade, como por exemplo.

Bloco de objeto: <object>

Secção para criar um objeto do jogo. Um objeto é criado particularmente, contendo a sua própria classe. Dentro do bloco do objeto podem ser colocados eventos que estão diretamente ligados a esse objeto. Na marcação de um objeto é também possível adicionar variáveis extra, além das variáveis/atributos já pré-definidos.

Bloco de evento: <event>, <transition>

Este bloco está dividido em dois fragmentos distintos, um para marcar a condição e outro para a ação. Um evento pode ser definido por uma ou várias condições que dão origem a uma ou mais ações. Os fragmentos das condições (<event>) geram *tokens* e podem aceder às variáveis do jogo, já os fragmentos das ações (<transition>) recebem *tokens* e podem alterar variáveis do jogo.

Bloco de ação: <transition>

Este é um bloco especial, exclusivamente para as ações do jogo. Este bloco serve para definir as ações, essa definição é dentro do fragmento das ações (<transition>) num bloco de evento. A definição das ações baseia-se na alteração de variáveis de jogo, seguindo a terminologia habitual da programação, e na utilização de um texto padrão para ações do jogo mais específicas, como destruir/criar um objeto ou alterar de layout.

Pasta do jogo

Um projeto PUMA é composto essencialmente por um documento onde é estruturado todo o jogo. Caso o projeto necessita de ficheiros externos, como imagens ou sons, de forma a facilitar o acesso a esses ficheiros, é indicado criar uma pasta com todos os arquivos utilizados no jogo incluindo o documento central.

6.1.3 Elementos de Markup

Nesta secção apresentam-se os elementos básicos da linguagem PUMA, aqui estabelecidos como tags. Estes estão agrupados por função de modo a facilitar a construção de um documento de definição do jogo. Cada elemento tem o papel de marcar um elemento funcional no documento.

Element	Description
<puma>	O <puma> representa o elemento raiz (primeiro elemento) de um documento PUMA. Todos os outros elementos devem proceder depois desse elemento.
<layout>	O <layout> representa um Layout (cenário) de jogo.
<layer>	O <layer> representa um Layer (plano) dentro de um Layout.
<object>	O <object> representa um Object (objeto) dentro de um Layer.
<event>	O <event> representa uma condição de um evento do jogo.
<transition>	O <transition> representa uma ação de um evento do jogo.

Tabela 6.1: Elementos PUMA

6.1.4 Atributos e onde podem ser usados

Os elementos têm atributos de forma adicionar informações e valores na sua definição. Esses atributos permitem configurar e ajustar os diversos elementos onde podem ocorrer, conforme os critérios propostos pelo criador, de forma a estabelecer os detalhes da estruturação e dos comportamentos dos elementos que compõem o jogo.

Attribute name	Elements	Description
name	<layout>, <object>	Nome do elemento, para identificar os vários Layouts e Objetos.
background	<layout>	Define o plano de fundo de um Layout.

width	<puma>, <object>	Define a largura da tela de um jogo PUMA ou define a largura de um Objeto.
height	<puma>, <object>	Define a altura da tela de um jogo PUMA ou define a altura de um Objeto.
x	<object>	Define a coordenada X de um Objeto.
y	<object>	Define a coordenada Y de um Objeto.
img	<layer>, <object>	Define a imagem (ou representação) de um Layer ou de um Objeto.
type	<event>	Define qual o tipo de condição de um Evento.
outputs	<event>, <transition>	Tokens que são gerados caso uma condição de um Evento se verifique.
inputs	<transition>	Tokens necessários para realizar uma ação de um Evento.
*	<puma>, <layout>, <object>	Variáveis adicionais. O nome do atributo corresponde ao nome da variável e o valor do atributo ao valor da variável.

Tabela 6.2: Atributos na linguagem PUMA e em que elementos podem ser usados.

6.1.5 Marcações (tags)

Os elementos da linguagem marcam um nível de bloco. Abaixo será explicada a função de cada marcador e os seus atributos essenciais. Noutra secção serão descritas algumas situações especiais.

<puma>

Este marcador sinaliza o início de um documento PUMA. Sendo o primeiro elemento a aparecer, obrigatoriamente define a raíz e indica que todo o conteúdo que vem a seguir, deve ser processado como um conjunto de códigos PUMA. Todos os elementos posteriores são descendentes deste.

Este elemento inclui os seguintes atributos.

width, largura da tela de jogo. (obrigatório)

height, altura da tela de jogo. (obrigatório)

Exemplo,

```
<puma width="500" height="500">
```

<layout>

Inicializa um novo layout do jogo. É necessário pelo menos um layout no desenvolvimento de um jogo. Todos os elementos posteriores, à exceção do <puma> e do próprio <layout>, são seus descendentes, isto significa que podem ser adicionados layers, objetos e eventos. Cada layout tem um nome como identificador, de forma a alterar o layout a ser exibido. O primeiro layout do documento é o primeiro a ser exibido. Cada layout tem os seus próprios layers, objetos e eventos.

Este elemento inclui os seguintes atributos.

name, nome de referência do layout. (obrigatório)

backgroud, referência da imagem ou cor para o plano de fundo.

Exemplo:

```
<Layout name="screen" background="screen.png" >
```

<layer>

Inicializa um novo layer, entendido como um suporte ou camada de conteúdo, dentro de um layout. Dentro de um layer podem ser adicionado vários objetos. A ordem em que cada layer aparece no documento corresponde à ordem em que cada layer deverá ser desenhado. Seguindo esta ordem o primeiro fica no plano mais a baixo e o último no plano mais a cima (i.e., a última sobreposição).

Este elemento inclui os seguintes atributos.

img, referência da imagem para o plano.

Exemplo,

```
<Layer img="mid_layer.png"> <\Layer>
```

<object>

Define um novo objeto do jogo. Dentro de um objeto podem ser adicionados eventos que lhe estarão diretamente associados. Se o mesmo objeto estiver em dois layouts será preciso criar o mesmo objeto duas vezes, originando duas instâncias, uma em cada layout, e através da definição de eventos atribuir ou modificar variáveis conforme o decorrer do jogo, conforme seja necessário. Cada objeto tem o seu próprio nome, algumas variáveis obrigatórias e possíveis variáveis adicionais, tudo isto definido e inicializado através dos seus atributos.

Este elemento inclui os seguintes atributos.

x, coordenada x do objeto. (obrigatório)

y, coordenada y do objeto. (obrigatório)

width, largura do objeto.

height, altura do objeto.

img, referência da imagem/media para o objeto. (obrigatório)

Exemplo,

```
<Object name="mario" x="100" y="20"  img="mario.png" points="0">
```

<event>

Um evento é composto por uma secção de condições e outra de ações. Este elemento inicializa um evento (ou mensagem) correspondente uma secção de condição. Cada evento define qual o tipo de condição e a condição a verificar e o, ou os, tokens gerados. Este elemento não necessita de marcação para fechar.

O elemento evento inclui os seguintes atributos.

type, qual é o tipo de condição para o evento. (obrigatório)

exp, expressão de condição a validar.

outputs, tokens gerados quando a condição é verificada. (obrigatório)

Exemplo,

```
<Event type="Expression" exp=" Lives < 0 "  outputs="T01">
<Event type= "Mouse Pressed" outputs="T02">
```

<transition>

Dentro de um evento este elemento corresponde à secção de ação. É definido pelos tokens que precisa de receber. Dentro da marcação deste elemento estão as ações correspondentes do evento. No corpo (texto) até à marca de fecho da transição podem ser incluidas codificações dos efeitos pretendidos dessa ação, como por exemplo através de expressões a executar. Aqui pode ser realizada a ponte para a chamadas de funções prédefinidas na implementação dos objectos do jogo, ou em bibliotecas a serem incluídas.

Este elemento inclui os seguintes atributos.

inputs, tokens que o elemento necessita receber para ser ativado. (obrigatório)

outputs, tokens gerados quando a condição é verificada. (obrigatório)

Exemplo,

```
<Transition inputs="T01"> { Lives -= 1 } <\Transition>
<Transition inputs="T02, T03" inputs="T04" > { Go to Layout screen2 } <\Transition>
```

Marcadores especiais

Alguns marcadores têm particularidades para facilitar a construção do documento. Em baixo estão três situações que variam um pouco da estrutura base, mas tem o propósito de colmatar possíveis dificuldades e torná-las mais simples.

Variáveis adicionais
Num documento PUMA podem ser adicionado variáveis extra ao jogo. Essas variáveis podem ser adicionadas nos marcadores <puma>, como variáveis globais ou partilhadas do jogo, no <layout>, variáveis locais do layout, e no <object>, variáveis específicas do objeto. As variáveis são colocadas nos atributos de cada um desses elementos, onde o nome e o valor da variável é o nome e o valor do atributo assim definido. Estas variáveis adicionais permitem acrescentar informação específica a gerir no âmbito do jogo, e são por isso essenciais para o game design adquirir especificidade.

Objetos padrão
Alguns objetos padrão podem ser inseridos no documento com uma tag própria. Assim, objetos típicos, de modo a facilitar o construtor, possuem um elemento individual, onde os seus atributos também variam conforme o seu género. Alguns exemplos de objetos padrão podem ser:

- **Figuras**: <circle> , <square> , <triangle>
- **Textos**: <text>
- **Items**: <item>
- **Botões**: <button>

Tipos de eventos
Nas condições dos eventos é necessário definir qual o tipo da condição de modo a proporcionar ao criador várias possibilidades. A escolha do tipo é através do atributo "type". Por cada tipo de condição escolhida haverá outras categorias de atributos coerentes com essa escolha para compor uma expressão ou condição integralmente. Alguns exemplos de tipos de condições e os seus atributos podem ser:

- **Expressões**: <event type="expression" cond="x == 0" ... >
- **Teclado**: <event type="keyboard pressed" key="space" ... >
- **Rato**: <event type="mouse pressed" ... >
- **Colisões**: <event type="on colision with" targer="object_y" ... >

6.1.6 Inicialização com a PUMA

Nesta secção, é demonstrado como inicializar um jogo cobrindo os fundamentos acima expostos. Através de um exemplo concreto, será descrito o processo de montagem de um documento PUMA. Esse documento traduz um cenário de jogo simples no qual se ilustra e demonstra a utilização dos elementos da linguagem.

A estrutura base de um documento PUMA será semelhante ao exemplo apresentado,

```
<Puma width="1208" height="801">

    <Layout  name="screen1" background="black" >
        <Layer>
            <Item text= "Click to start" x="1208/2" y="801/2" color="white">
                <Event type= "Mouse Pressed" outputs="L101">
            <\Item>
        <\Layer>

        <Transition inputs="L101"> Go to Layout screen2 <\Transition>
    <\Layout>

    <Layout name="screen2" background="assets/screen2.png" >
        <Layer>
            <Object name="Guybrush" x="50" y="250" img="assets/guybrush">
                <Event type="Expression" exp=" x > 600 "  outputs="L203">

                <Transition inputs="L201"> { x += 10 } <\Transition>
                <Transition inputs="L202"> { x -= 10 } <\Transition>
            <\Object>
        <\Layer>

        <Event type= "Keyboard Pressed" key="RIGHT" outputs="L201">
        <Event type= "Keyboard Pressed" key="LEFT" outputs="L202">

        <Transition inputs="L203"> Go to Layout screen3 <\Transition>
    <\Layout>
```

```
<Layout name=”screen3” background=”assets/screen3“ >
        <Layer>
                <Object name=”Guybrush” x=”50” y=”250” img=“assets/guybrush”>
                        <Event type=”Expression” exp=” x < 20 “  outputs=”L304”>
                        <Event type=”Expression” exp=” x > (pirate.x - 20) && x < (pirate.x + 20) ”  outputs=”L305”>

                        <Transition inputs=”L301”> { x += 10 } <\Transition>
                        <Transition inputs=”L302”> { x -= 10 } <\Transition>
                <\Object>

                <Object name=”Pirate” x=”width/2” y=”400” img=“assets/pirate”>

                <\Object>
        <\Layer>

        <Event type= ”Keyboard Pressed” key=”RIGHT” outputs=”L301”>
        <Event type= ”Keyboard Pressed” key=”LEFT” outputs=”L302”>
        <Event type= ”Keyboard Pressed” key=”K” outputs=”L303”>

        <Transition inputs=”L304”> Go to Layout screen2 <\Transition>
        <Transition inputs=”L304”>  { x = 802 - 20 } <\Transition>
        <Transition inputs=”L303, L305”>
                <Text text=”fala 1” size=”20”  x=”1208/2” y=”600” color=”white”>
                <Text text=”fala 2” size=”20”  x=”1208/2” y=”600 + 20” color=”green”>
                <Text text=”fala 3” size=”20”  x=”1208/2” y=”600 + 40” color=”white”>
                <Text text=”fala 4” size=”20”  x=”1208/2” y=”600 + 60” color=”green”>
        <\Transition>
<\Layout>

<\Puma>
```

O documento começa por definir o início do jogo através do marcador, <puma>, e em seguida, são definidos 3 layouts.

Interpretação do Layout 1

O primeiro layout é um ecrã para o inicializar o jogo, na medida, em que o jogador que clica para começar quando estiver pronto.

A este Layout 1 é dado o nome de “screen1” e é definida a cor do background (plano de fundo) como preta.

Dentro do marcador, <layout>, está definido um layer que contém um objeto (correspondente a um item) e um evento. Nesse objeto é atribuído o texto, as coordenadas e a cor do item.

Dentro do objeto está marcada a condição do evento, <event>, que é do tipo “mouse pressed”, isto corresponde que quando o jogodar clicar no item é gerado o token “L101”.

No Layout também está definido a ação do evento, <transition>, na qual necessita de receber o token “L101” para ser ativada, a ação responde à troca do Layout 1 para o Layout 2.

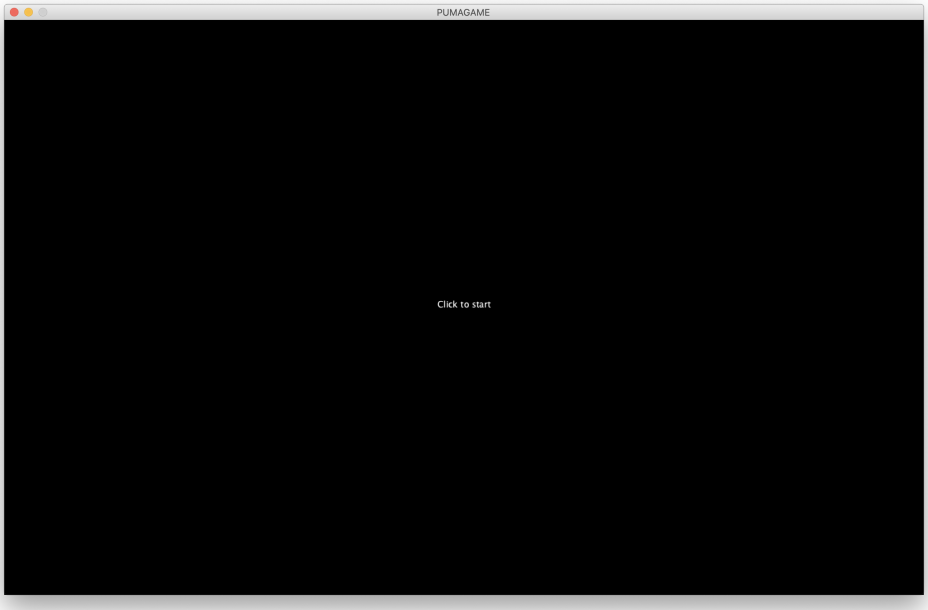


Figura 6.1: PUMA exemplo: Layout 1

Interpretação do Layout 2

Este Layout 2 é o primeiro do jogo. Esta cena apenas contém uma personagem principal que se move para os lados, por cima de um background. Também contém três eventos, dois para reconhecer o teclado para mover a personagem e outro para trocar para o Layout 3 se a personagem chegar ao limite da borda de jogo.

A este Layout 2 é dado o nome de "screen2" e é definido o background (plano de fundo) com a referência para a imagem pretendida, "assets/screen2.png".

No layer deste layout está presente um objeto. Esse objeto tem o nome de "guybrush", as coordenadas do X e Y e a sua imagem de referência, como atributos.

Dentro do objeto tem 1 <event>, para verificar se a personagem ultrapassa os 600 na coordenada do X, e 3 <transition>, que são ações para alterar a sua coordenada X, dependendo do evento transacionado, dois são para mover a personagem para a direita ou esquerda e o outro é para atribuir à coordenada X o valor de (802 - 20) quando a tela de jogo é trocada do layout 3 para o layout 2.

No restante do layout estão definidos 2 <event>, para receber a informação que as teclas LEFT e RIGHT foram pressionadas (para a personagem andar para a direita ou esquerda), mais 1 <transition>, que acionada, troca a tela de jogo para o Layout 3.

Figura 6.2: PUMA exemplo: Layout 2

Interpretação do Layout 3

Este é o terceiro layout, sendo o segundo cenário do jogo. Nesta cena tem um background diferente e tem duas personagens, a principal e o pirata. A personagem principal pode na mesma andar para os lados e falar para o pirata se estiver perto dele. Os eventos deste layout são para o personagem se mover para os lados, clicar numa tecla para falar com o pirata e outra para trocar para o Layout 2 caso o personagem volte para trás.

A este Layout 3 é dado o nome de "screen3" e é definido o background (plano de fundo) com a referência para a imagem pretendida, "assets/screen3.png".

No layer estão presente 2 objetos, um para a personagem principal e outro para o pirata.

Para a personagem principal, o objeto tem o nome de "guybrush", as coordenadas do X e Y e a sua imagem de referência, como atributos.

Dentro desse objeto tem 2 <event>, um para verificar se a coordenada X da personagem está abaixo de 20 e outro para se a personagem está perto do pirata. Também tem 2 <transition> para a personagem se mover para a direita ou esquerda conforme o evento correspondente.

Para o pirata, o objeto tem o nome de "pirate", as coordenadas do X e Y e a sua imagem de referência como atributos.

Nos eventos do layout, há 3 <event>, todos para verificar se as teclas RIGHT, LEFT e 'K' foram pressionadas.

Ainda no Layout 3, tem 2 <transition>, um para trocar para o Layout 2 caso a personagem principal tenha o X abaixo de 20 e outro para apresentar as falas caso a personagem esteja perto do pirata e a tecla 'K' tenha sido pressionada pelo jogador.

Figura 6.3: PUMA exemplo: Layout 3

Para as falas, são adicionados 4 objetos do tipo texto (text) dentro da marcação <transition> porque está associada a esse evento para as falas aparecerem. Nos objetos de texto tem o seu nome que corresponde também ao seu texto, o tamanho, as coordenadas e cor, como atributos.

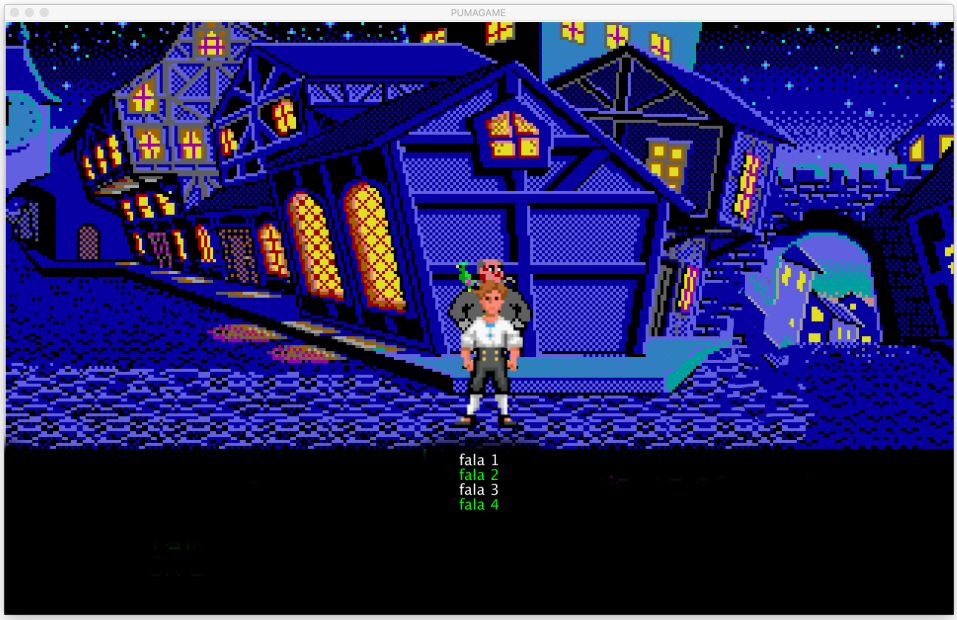


Figura 6.4: PUMA exemplo: Layout 3 com falas sobrepostas

6.1.7 Tradução em Código

Todo o documento PUMA deverá poder ser traduzido numa componente em código, onde o jogo é, na prática, implementado. Essa tradução do documento resulta na programação do mesmo. O código aqui gerado segue a mesma filosofia e fundamentação que foi descrita no ponto **5.3.3** e visa apresentar uma Prova de Conceito de que a proposta é realizável, mesmo não existindo ainda uma implementação alhgoritmica dessa tradução. Com ajuda do exemplo utilizado acima é demonstrado o código que deve resultar desse documento.

1. Inicialização

A inicialização que corresponde ao Setup. Neste parte é definido o tamanho da tela e é definido e inicializado as classes dos objetos, "guybrush" e "pirate".

Class Guybrush
x y img

Class Pirate
x y img

size(1208, 801)

initialize guybrush **as** Guybrush
initialize pirate **as** Pirate

2. Telas de Jogo

Neste exemplo são usados três layouts. O layout de exibição é determinado pela variável "gameScreen" e cada layout tem a sua função individual de desenho.

```
function draw:
  if gameScreen == 0 than
    call screen1()

  if gameScreen == 1 than
    call screen2()

  if gameScreen == 2 than
    call screen3()
end draw

function screen1:
  // codes of screen1

end screen1

function screen2:
  // codes of screen2

end screen2

function screen3:
  // codes of screen3

end screen3
```

3. Screen1

No início do jogo a variável "gameScreen"aponta para o primeiro layout (screen1).

```
function screen1:
  /** Screen1 contents ***/

  /** Events Contents ***/

end screen1
```

No layout, é definido o backgroud como preto e é inserido o item com o texto “Click to start”.

```
backgroud(0)

draw Item ("Click to start")
```

Com apenas um evento é definido uma condição que verifique se o utilizador clica no item e a ação que muda a variável "gameScreen"para o Screen2.

Event 1

```
if mousePressed than
  gameScreen = 1
```

4. Screen2

Na função do Screen2, é definido o background com a imagem de referência e é desenhado a personagem do jogo, o objeto "guybrush".

```
backgroud("assets/screen2.png")

draw guybrush
```

Para os eventos deste Layout estão definidos os seguintes pares de condições-ações:

Event 1

```
if keyPressed and keyCode == RIGHT than
  guybrush.x + = 10
end of if
```

Event 2

```
if keyPressed and keyCode == LEFT than
  guybrush.x − = 10
end of if
```

Event 3

```
if guybrush.x > 600 than
  gameScreen = 2
end of if
```

5. Screen3

Na função do Screen3, é definido o background com a imagem de referência e é desenhado os objetos “guybrush” e "pirate".

```
backgroud("assets/screen2.png")

draw guybrush

draw pirate
```

Para os eventos deste Layout estão definidos os seguintes pares de condições-ações:

Event 1

```
if keyPressed and keyCode == RIGHT than
  guybrush.x + = 10
end of if
```

Event 2

```
if keyPressed and keyCode == LEFT than
  guybrush.x − = 10
end of if
```

Event 3

```
if guybrush.x < 20 than
  gameScreen = 1
  guybrush.x = 802 - 20
end of if
```

Event 4

```
if guybrush.x > (pirate.x - 20) and guybrush.x < (pirate.x + 20) and keyPressed and
keyCode == 'k' than
  draw Text "fala 1"
  draw Text "fala 2"
  draw Text "fala 3"
  draw Text "fala 4"
end of if
```

6.2 Resumo

Apresenta-se aqui uma linguagem de markup para prototipagem rápida de jogos, definem-se os seus elementos e atributos, e elistra-se o seu uso com um exemplo de cenário de jogo. Ilustra-se ainda uma tradução em código do modelo apresentado, verificando a sua viabilidade.

A linguagem PUMA procura ser acessível e fácil de aprender. Esta pode ser uma maneira invulgar de montar novos jogos, mas pensamos que tem potencial para inicar um caminho/rumo alternativo dos sistemas que permitem desenvolver videojogos, sobretudo numa fase de teste de conceitos onde a velocidade e facilidade de efectuar alterações é essencial. Adotando esta linguagem de marcação é expectável uma abertura e crescimento no acesso à definição e ensaio de videojogos, à imagem do que aconteceu com as linguagens de markup na web, que muitos não-programadores acabaram por aprender.

Consideramos esta linguagem como uma versão alpha, isto é, uma primeira fase ainda tentativa, no seu desenvolvimento, e por isso está sujeita a várias melhorias. Nesse perspetiva, dado que a linguagem é aberta a sofrer alterações no sentido de a tornar mais universal e adequada ao propósito, também é expectável que uma versão beta possa vir a ser uma aplicação concretizada na forma de um editor de jogos (o que não foi ainda possível concretizar no âmbito deste estágio). Estas são as duas visões futuras com valor para a plataforma PUMA. .

Capítulo 7

Conclusões

Neste capitulo é formada uma reflexão ampla do trabalho consumado, encerrando assim a matéria com algumas conclusões, dividendos e sugestões futuras.

7.1 Reflexão Sobre o Percurso

Nesta proposta, inicialmente, foi idealizado a elaboração de uma plataforma para criação de jogos ("*Game Engine*") distinta, focada em ensaiar o "*game design*" de jogos experimentais, de forma rápida e simplificada. Um tema, sobretudo, inspirado na forma como o modding marcou diversos jogos pela oportunidade de experimentar diversas modificações.

Dessa forma, fez-se um levantamento de áreas de estudo a respeito do tema com intenção de sistematizar as plataformas que já existem e de elaborar uma nova plataforma, diferenciada pelo seu "minimalismo", indo ao encontro do que foi proposto. Através dessa pesquisa e, consequente, concretização do problema, levaram à definição dos pontos fulcrais e objetivos para a concretização da plataforma desejada.

Após esse estudo e traçados esses planos, sucedeu-se a proposta de design. Esta proposta mostrou ser uma etapa mais complicada do que o esperado. Atingir a simplicidade requereu esforço e repetida ponderação do essencial e do assessório. Percebendo o grau de complexidade que foi desenvolver uma proposta de design para este caso, dedicou-se mais tempo a esta fase de modo a resultar numa arquitetura adequada ao trabalho e propósito desta dissertação.

Para garantir que o design proposto que seria praticável foram realizadas provas de conceito. A primeira dessas provas consistiu em demonstrar que o software a desenvolver teria de facto todas as condições de ser realizado conforme o delineado, antes de partir para o seu desenvolvimento. Assim, a prova passou por uma representação da construção de um jogo utilizando os conceitos definidos na arquitetura e no código que deve ser gerado para essa construção.

Posteriormente, seguiu-se o desenvolvimento onde surgiu a ideia da plataforma PUMA ser estabelecida como uma Linguagem de Marcação ("*Markup Language*"). Essa ideia mostrou ter muito valor, pois é um caminho alternativo às soluções mais abituais na prototipagem de jogos, mas que vai ao encontro do que se procurou com este trabalho. Uma Linguagem de Marcação para o desenvolvimento de jogos é uma possibilidade muito pouco explorada, mas que pode de fato tornar

o desenvolvimento de protótipos rápidos bem mais simples e eficiente, evitando o moroso e pesado processo de aprendizagem e "colagem"de bibliotecas de código.

Sendo assim, confiando no potencial dessa linguagem, desenvolveu-se e documentou-se a mesma, definindo, assim, a sua estruturação e os seus fundamentos. Desta forma, foi concebido um sistema de anotações em texto com base em marcadores, género do XML. Também, foi realizada uma nova prova, de forma, a exemplificar o uso da linguagem para montar um jogo em concreto, seguindo moldes similares à anteriormente reportada. O trabalho consumado, com fundamento nas provas efetuadas, tem capacidade para ser explorado, e o potencial de ser estendido pela introdução de novos conceitos. Por isso, é ainda possível aproveitar o trabalho para outros fins ou mesmo melhorá-lo e investir em novas variantes.

7.2 Resultados Alcançados

Em retrospectiva, um dos pontos mais importantes a comprovar neste projeto era a possibilidade de haver outras formas de construir jogos, mais simples e rápidas, comparativamente ao uso da Game Engines existentes. É possível afirmar que, com base nesta pesquisa, estudo e trabalho realizado, essa possibilidade existe, ou seja, há outros caminhos para novos motores de jogos onde o criador não tenha de gastar demasiado tempo a aprender a utilizar o software, a aprender a construir um jogo ou, mesmo que já tenha esse conhecimento, perder demasiado tempo a colocar as suas ideias no software. Sendo que, é plausível desenvolver novo software com design mais simples, e que mais pessoas conseguirão aprender a utilizar, e por consequentemente, possam expressar e desenvolver as suas ideias de jogos. Ou seja, abre-se assim caminho a mais novas ideias, e novos talentos focados nas componentes artísticas do game design.

7.3 Projeções Futuras

Este é um projeto bastante complexo, e está sempre sujeito a melhorias e novas soluções. Dentro do que foi apresentado nesta dissertação, há duas projeções muito convincentes, uma é a progressão e consolidação da linguagem de marcação como uma Game Engine, e a outra é a transição dessa linguagem para um editor gráfico.

No que toca à linguagem de marcação existem muitas maneiras de evoluir como um motor de jogos, desde logo, novos marcadores como novas categorias de objetos e novas possibilidades de marcar eventos, a reformulação da estrutura de montar um jogo para um modelo mais leve, ter um editor onde de um lado se edita a definição no documento e do outro se experimenta o executável do jogo em tempo real, são algumas das hipóteses de desenvolvimento. A potencialidade de montar jogos através de uma linguagem de marcação pode trazer muito sucesso e experimentação ao ramo, uma vez que cada vez mais pessoas estão familiarizadas com este tipo de tecnologia.

Em relação ao editor gráfico, esta foi uma possibilidade abordada no inicio deste projecto, mas foi bloqueada por não existir nessa altura uma ideia clara sobre os elementos dessa edição. Olhado novamente para o problema, torna-se agora evidente a possibilidade de ajustar a linguagem de marcação a uma engine em concreto, o que se poderia traduzir num editor de jogos, mais simples e intuitivo.

Bibliografia

[1] Libpng. http://www.libpng.org/pub/png/libpng.html. Acesso: Dezembro de 2019.

[2] Minecraft. https://www.minecraft.net/. Acesso: Outubro de 2019.

[3] Minecraft Wiki. https://minecraft.gamepedia.com/. Acesso: Outubro de 2019.

[4] Unit. https://unity3d.com/. Acesso: Outubro de 2019.

[5] Audiere. . http://audiere.sourceforge.net/. Acesso: Dezembro de 2019.

[6] Meio bit. PICO-8, o incrível console retrô que só existe na internet. https://meiobit.com/325111/pico-8-o-incrivel-console-retro-que-so-existe-na-internet/. Acesso: Dezembro de 2019.

[7] Bullet. Bullet Physics. https://pybullet.org/wordpress/. Acesso: Dezembro de 2019.

[8] AMD by rhallock. Radeon GPUs are ready for the Vulkan graphics API. https://community.amd.com/community/gaming/blog/2016/02/16/radeon-gpus-are-ready-for-the-vulkan-graphics-api. Acesso: Dezembro de 2019.

[9] Chipmunk. Chipmunk2D. http://chipmunk-physics.net/. Acesso: Dezembro de 2019.

[10] Stelios Christopoulou, Eleftheria Xinogalos. Overview and comparative analysis of game engines for desktop and mobile devices. *International Journal of Serious Games*, 2017.

[11] Slant community. Construct 2 Review. https://www.slant.co/options/1058/~construct-2-review. Acesso: Janeiro de 2020.

[12] Slant community. Pico-8 Review. https://www.slant.co/options/9018/~pico-8-review. Acesso: Janeiro de 2020.

[13] Slant community. Unity3d Review. https://www.slant.co/options/1047/~unity3d-review. Acesso: Janeiro de 2020.

[14] Slant community. Unreal Engine 4 Review. https://www.slant.co/options/5128/~unreal-engine-4-review. Acesso: Janeiro de 2020.

[15] Cícero Bezerra da Silva. Prova de Conceito (Poc): o que é e como construir? https://blog.nectarcrm.com.br/o-que-e-prova-de-conceito/#como-organizar-uma-prova-de-conceito. Acesso: Maio de 2020.

[16] Desenvolvimento de Jogos. Bibliotecas E Apis. http://desenvolvimentodejogos.wikidot.com/bibliotecas-e-apis. Acesso: Outubro de 2019.

[17] Produção de Jogos. Game Design. https://producaodejogos.com/game-designer/#o-que-e-game-designer. Acesso: Outubro de 2019.

[18] Francisco de Mello Toledo e Silva. O USO DA TECNOLOGIA DE GAME ENGINES NO PROCESSO DE PROJETO DE UNIDADES PRODUTIVAS, 2013. Monografia do Curso de Graduação em Engenharia de Produção da Universidade Federal de São Carlos(UfSCar).

[19] Sílvia Inês Dallavalle de Pádua; Andrea Ribari Yoshizawa da Silva; Arthur José Vieira Porto; Ricardo Yassushi Inamasu. O potencial das redes de Petri em modelagem e análise de processos de negócio. *Gestão&Produção*, v.11(n.1):p.109–119, 2004.

[20] Game Designing. Learn to Design Video Games. https://www.gamedesigning.org/. Acesso: Outubro de 2019.

[21] Apple Developer. Metal. https://developer.apple.com/metal/. Acesso: Dezembro de 2019.

[22] Raphael Dias. Produção de jogos. https://producaodejogos.com/. Acesso: Dezembro de 2019.

[23] Renan Felipe dos Santos. Meus jogos favoritos para PICO-8. https://renansantostradutor.wordpress.com/2018/02/24/meus-jogos-favoritos-para-pico-8/. Acesso: Dezembro de 2019.

[24] Unreal Engine. Unreal Engine 4 Documentation. https://docs.unrealengine.com/en-US/index.html. Acesso: Dezembro de 2019.

[25] Nvidia GameWorks. GameWorks Pphysx Overview. https://developer.nvidia.com/gameworks-physx-overview. Acesso: Dezembro de 2019.

[26] GStreamer. . https://gstreamer.freedesktop.org/. Acesso: Dezembro de 2019.

[27] GTK. The GTK Project. https://www.gtk.org/. Acesso: Dezembro de 2019.

[28] Guru99. Incremental Model in SDLC: Use, Advantage Disadvantage. https://www.guru99.com/what-is-incremental-model-in-sdlc-advantages-disadvantages.html. Acesso: Dezembro de 2019.

[29] Havok. Havok Physics. https://www.havok.com/products/havok-physics/. Acesso: Dezembro de 2019.

[30] Lexaloffle Games LLP Joseph White. PICO-8 Manual. https://www.lexaloffle.com/pico8_manual.txt. Acesso: Dezembro de 2019.

[31] Eric Zimmerman Katie Salen. *Rules of Play: Game Design Fundamentals*. MIT Press, 2003.

[32] Khronos. Vulkan. https://www.khronos.org/vulkan/. Acesso: Dezembro de 2019.

[33] Lexaloffle Games LLP. PICO-8. https://www.lexaloffle.com/pico-8.php. Acesso: Dezembro de 2019.

[34] Scirra Ltda. Construct2. https://www.scirra.com/construct2. Acesso: Dezembro de 2019.

[35] Scirra Ltda. Official Construct 2 Manual. https://www.scirra.com/manual/1/construct-2. Acesso: Dezembro de 2019.

[36] Nvidia. Directx 12. https://blogs.nvidia.com/blog/2014/03/20/directx-12/. Acesso: Dezembro de 2019.

[37] OpenAL. . https://www.openal.org/. Acesso: Dezembro de 2019.

[38] OpenGL. . https://www.opengl.org/documentation. Acesso: Dezembro de 2019.

[39] Oracle. Java Media Framework API (JMF). https://www.oracle.com/technetwork/java/javase/tech/index-jsp-140239.html. Acesso: Dezembro de 2019.

[40] Jesse Schell. *The Art of Game Design: A Book of Lenses*. AK Peters/CRC Press, 2014.

[41] SFML. Simple and Fast Multimedia Library. https://www.sfml-dev.org/. Acesso: Dezembro de 2019.

[42] CAST Software. Risk Management in Software Development and Software Engineering Projects. https://www.castsoftware.com/research-labs/risk-management-in-software-development-and-software-engineering-projects. Acesso: Janeiro de 2020.

[43] SourceForge. DevIL. http://openil.sourceforge.net/. Acesso: Dezembro de 2019.

[44] SourceForge. Libjpeg. http://libjpeg.sourceforge.net/. Acesso: Dezembro de 2019.

[45] Crazy Eddie's GUI System. CEGUI. http://cegui.org.uk/content/getting-started. Acesso: Dezembro de 2019.

[46] Firelight Technologies. FMOD. https://www.fmod.com/. Acesso: Dezembro de 2019.

[47] Techopedia. Technology Dictionary. https://www.techopedia.com/dictionary. Acesso: Novembro de 2019.

[48] TECHTUDO. Space Invaders. https://www.techtudo.com.br/tudo-sobre/space-invaders.html. Acesso: Maio de 2020.

[49] Unity. Unity User Manual (2019.2). https://docs.unity3d.com/Manual/index.html. Acesso: Dezembro de 2019.

[50] Ariane Velasco. Conheça alguns dos melhores Mods para Minecraft. https://canaltech.com.br/games/conheca-alguns-dos-melhores-mods-para-minecraft/. Acesso: Dezembro de 2019.

[51] Bill Kuechler Vijay Vaishnavi and Stacie Petter. "design science research in information systems". *Annalen der Physik*, 2004. Acesso: Outubro de 2019.

[52] "Wikipedia. Minecraft. https://pt.wikipedia.org/wiki/Minecraft. Acesso: Outubro de 2019.

[53] Wikipedia. Motor de jogo. https://pt.wikipedia.org/wiki/Motor_de_jogo. Acesso: Outubro de 2019.

[54] Wikipédia. League of Legends. https://pt.wikipedia.org/wiki/League_of_Legends. Acesso: Janeiro de 2020.

[55] wxWidgets. Cross-Platform GUI Library. https://www.wxwidgets.org/. Acesso: Dezembro de 2019.

Appendices

Appendix A

Unity3D	
PROS	**CONTRAS**
1. Curva de aprendizagem fácil	1. Muito consumo de memória, cria erros e problemas
2. Fornece uma enorme lista de assets através da Asset Store	2. Ficheiros muito pesados
3. Permite prototipagem rápida	3. Ambiente de construção desadequado para a performance (muitas draw calls)
4. Interface intuitiva	4. Não é prático para não-programadores
5. Para mais de 20 plataformas	5. Curva de aprendizagem alta para quem tem dificuldades de programação
6. Sistema de animações e standard shaders	6. Muitos recursos desperdiçados, que ocupam imenso espaço
7. Boas gráficas para efeitos áudios e visuais	7. Adiciona muitos recursos sem corrigir os anteriores
8. Permite criar formulários e ferramentas personalizadas	8. Propício a um desempenho baixo e com bugs
9. Bem estruturado	
10. Debugging	

Tabela 1: Unity3D: Pros e Contras

Unreal Engine 4	
PROS	**CONTRAS**
1. Sistema de scripting visual para não programadores 2. Permite prototipagem rápida 3. Iluminação global dinâmica (voxel cone tracing), diminui a potência computacional necessária 4. Editor e exportação para multiplataformas. 5. Sistema poderoso de efietos visuais e efeitos de iluminação 6. Compilação e interação rápida 7. Gráficos realistas 8. Conjunto de recursos técnicos para todos os aspetos do desenvolvimento de jogos 9. Não é necessária experiência de programação	1. Tamanho dos projetos é elevado 2. Performance lenta 3. Dificuldades para criadores iniciantes, curva de aprendizagem acentuada 4. Processo de construção extremamente longo 5. Desempenho fraco em dispositivos moveis. 6. Design mal estruturado 7. Software muito pesado, propício a crashes

Tabela 2: Unreal Engine 4: Pros e Contras

Construct 2	
PROS	**CONTRAS**
1. Aprendizagem de desenvolvimento rápida 2. Sistema de física integrado 3. Editor de imagens e animações integrado 4. Necessário pouco conhecimento de programação 5. Fácil de criar partículas e animações 6. Exportação para todas as principais plataformas 7. Sistema de comportamentos integrado 8. Ecossistema de plug-in ativo 9. Sistema de eventos diversificado 10. Suporte para câmara, microfone e reconhecimento de fala 11. Visualização do jogo instantânea 12. Interface parecida com a do Microsoft Office 13. Permite criar jogos multiplayer 14. Simplicidade	1. Editor só para Windows 2. HTML5 depende muito do desempenho do browser 3. Não há garantia de desempenho e qualidade para todas as plataformas de exportação 4. Exportar nativamente só para HTML5 5. Desempenho baixo para dispositivos móveis 6. Não exporta para código nativo em dispositivos móveis

Tabela 3: Construct2: Pros e Contras

Pico-8	
PROS	**CONTRAS**
1. Possibilidade de escrever código num editor externo 2. Linguagem de programação muito simples (Lua) 3. Criação de jogos simples para ciradores iniciantes 4. Editores de som mais fáceis de usar 5. Compartilhar o jogo num ficheiro PNG com outros utilizadores do pico-8 6. Quase todos os jogos feitos em pico-8 tem o código disponível para consulta 7. Fácil de montar um jogo	1. Falta de cálculo do delta time 2. Nenhuma biblioteca de colisões 3. Restrições de memória 4. Editor de código

Tabela 4: Pico-8: Pros e Contras