INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Scarlett Game Studio

JOÃO FILIPE DA SILVA ALVES
Outubro de 2016

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Scarlett Game Studio

João Filipe da Silva Alves

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Sistemas Gráficos e Multimédia

Orientador: Filipe de Faria Pacheco

Júri:

Presidente:

[Nome do Presidente, Categoria, Escola]

Vogais:

[Nome do Vogal1, Categoria, Escola]

[Nome do Vogal2, Categoria, Escola] (até 4 vogais)

Porto, outubro 2016

Agradeço a todos que de uma forma ou de outra fizeram parte do processo de idealização e criação desta dissertação, tendo a consciência de que facilitaram o desenvolvimento da mesma.

Resumo

Vivemos num mundo cada vez mais enriquecido por aplicações multimédia, onde existe uma oferta exorbitante de aplicações com elevada qualidade. Os padrões de desenvolvimento são criados com o objetivo de estabelecer uma estrutura homogênea no que toca à criação de software em geral. Do processo criativo à finalização de uma aplicação, podemos também concordar que as ferramentas de desenvolvimento são em grande parte, a essência de qualquer criação, infligindo um impacto significativo sobre a qualidade das mesmas.

Nesta dissertação, vai ser abordado o processo detalhado sobre a pesquisa, desenvolvimento e análise da construção de um ambiente de desenvolvimento de videojogos. A esta aplicação, vamos dar o nome de Scarlett Game Studio, que em poucas palavras, pode ser descrita como uma ferramenta flexível que possui as funcionalidades necessárias para o desenvolvimento de videojogos em duas dimensões.

Palavras-chave: Multimédia, Ferramentas de Desenvolvimento, Videojogos, 2D

Abstract

We live in a world enriched by multimedia applications where there is an enormous offering of high quality applications. The development patterns are created with the goal of establishing a homogeneous path regarding the creation of software in general. From the creative process to the completion of an application, we can also agree that development tools are largely the essence of any creation, inflicting a significant impact on their quality.

In this dissertation, it will be discussed in detail the process about the research, development and analysis on how a video game engine is created. To this application, we will give the name of Scarlett Game Studio, which in a nutshell, can be described as a flexible tool that has all the features needed in a typical scenario of a 2D game development.

Keywords: Multimedia, Development Tools, Videogames, 2D

Índice

Lista de Figuras

Lista de Tabelas

Acrónimos e Símbolos

Lista de Acrónimos

API Do inglês, *Application Programming Interface* (Interface de Programação de Aplicativos) é um conjunto de rotinas e padrões estabelecidos por um software para a utilização das suas funcionalidades por aplicativos que não pretendem envolver-se em detalhes da implementação do software, mas apenas usar seus serviços.

IDE Do inglês*, Integrated Development Environment* (Ambiente Integrado de Desenvolvimento).

ISEP Instituto superior de Engenharia do Porto

MVC *Model-View-Controller*

XML *Extensible Markup Language*, linguagem baseada em marcações.

DOM Do inglês, *Document Object Model*, é uma convenção utilizada para a representação e interação com objetos em documentos HTML, XHTML e XML.

HTTP Do inglês, Hypertext Transfer Protocol, é um protocolo de comunicação de dados da web

SOAP Do inglês, Simple Object Access Protocol, é um protocolo usado para troca de informação estruturada baseada em XML.

Lista de Símbolos

π Pi

1 Introdução

Neste capitulo apresenta-se, de forma sucinta, o enquadramento e contribuição do Scarlett Game Studio e os objetivos pretendidos no desenvolvimento do mesmo. É também abreviadamente descrita a organização do documento com os capítulos envolventes.

1.1 Enquadramento do Documento

Este documento foi realizado no âmbito da disciplina de Projeto de Tese do curso de Mestrado em Engenharia Informática no Instituto Superior de Engenharia do Porto, ramo de Sistemas Gráficos e Multimédia. Esta unidade curricular faz parte do percurso académico do mestrado e é realizada anualmente no segundo ano do curso, tendo como principal objetivo a pesquisa e desenvolvimento científico sobre um tema relacionado com a área de estudo.

1.2 Âmbito de Desenvolvimento

Desde as décadas de 1970 e 1980, altura em que começaram a surgir os primeiros jogos eletrónicos (Kent, 2010), o mercado associado tem vindo a crescer a um passo surpreendente. Atualmente os videojogos são uma das formas mais populares de entretenimento competindo com outras grandes áreas tais como o cinema e música. Ao longo da sua evolução, tecnicamente, os videojogos têm desenvolvido de uma forma bastante progressiva (Kent, 2010). É de certa forma fascinante observar o que conseguimos alcançar nos dias de hoje e pensar que tudo começou pela apresentação de simples figuras geométricas num ecrã eletrónico.

Da mesma forma que os videojogos evoluíram, os ambientes de desenvolvimento que suportam a sua criação, também seguiram o mesmo percurso e cada vez mais proporcionam um conjunto de métodos de trabalho mais alargado e eficiente. A estes ambientes denominamos de motores de jogo (“*game engines” em inglês)* que, na sua essência, são *frameworks* desenhadas para a criação e desenvolvimento especializado de videojogos. A complexidade de um motor de jogo pode diferenciar bastante dependendo das necessidades associadas, no entanto, tipicamente é composta por um motor de renderização (módulo responsável por renderizar objetos 2D ou 3D num *ecrã*), áudio, animação, colisão e *scripting* (Eberly, 2006).

Nesse sentido, o âmbito base de desenvolvimento desta dissertação, debruça sobre o processo completo de criação de um motor de jogos 2D chamado Scarlett Game Studio. Nos seguintes capítulos, irá ser apresentada uma abordagem extensiva sobre o desenvolvimento deste tipo de software, aplicando algumas das melhoras práticas que existem atualmente na sua implementação tendo por base o estado da arte relacionado. O resultado final inclui um ambiente inteiramente preparado com todas as funcionalidades apropriadas para a criação e manutenção de videojogos.

1.3 Motivações e Contributos

De forma a suportar o crescimento tecnológico do âmbito envolvente, existe a necessidade crescente de uma documentação extensiva e atualizada sobre todo o processo empregue no planeamento e construção de um motor de criação de jogos. Grande parte da informação tradicional disponível em formato digital não aborda as tecnologias mais recentes de ponta a ponta o que torna fundamental a criação deste género de conteúdo.

Além da descrição do processo de implementação, o Scarlett Game Studio tem como principal objetivo disponibilizar funcionalidades inéditas (descritas ao detalhe nos próximos capítulos), que certamente irão facilitar e sustentar o desenvolvimento de videojogos em contextos profissionais e pessoais. Com o objetivo de endereçar mais facilmente os desenvolvedores, as tecnologias utilizadas neste desenvolvimento foram cuidadosamente selecionadas de forma a serem facilmente compreendidas e utilizadas.

Por fim, outro aspeto importante de trabalhar sobre tecnologias modernas é a possibilidade de acompanhar, analisar e suportar passo a passo novas capacidades das mesmas. É uma excelente oportunidade de contribuir para o avanço das várias bibliotecas e ferramentas relacionadas a este desenvolvimento que ou por falta de experimentação ou necessidade, não providenciam determinadas funcionalidades ou mecanismos. Durante o desenvolvimento do Scarlett Game Studio foram elaboradas algumas contribuições, por carência, em projetos de código aberto que fazem parte das dependências do mesmo.

1.4 Organização do Documento

Como foi mencionado previamente, o processo de desenvolvimento de motores de jogos é bastante complexo, dessa forma, este documento é distribuído por diversos capítulos que apresentam o estudo e descrição de implementação dos diferentes módulos envolvidos:

- **Capítulo 2** – Motores de Jogos:
  apresentação e descrição geral sobre o conceito de motor de jogo assim como projetos relevantes associados ao tema.

- **Capítulo 3 –** Fundamentos de Desenvolvimento:
  apresentação e descrição sobre conceitos básicos de operações tipicamente realizadas no âmbito de desenvolvimento de videojogos.

- **Capítulo 4 –** Scarlett Game Studio Design:
  apresentação da estrutura do software com a identificação dos diferentes módulos presentes no desenvolvimento.

- **Capítulo 5** – Framework:
  descrição por extenso sobre a Framework implementada no motor de jogo.

- **Capítulo 6** – Serviço Remoto:
  descrição sobre o serviço remoto do motor de jogo (base de dados e API de suporte)

- **Capítulo 7** – Editor:
  descrição por extenso sobre o Editor (interface de utilizador) implementado.

- **Capítulo 8** – Avaliação:
  descrição sobre a apreciação geral do trabalho realizado.

- **Capítulo 9** – Trabalhos relacionados:
  descrição sobre trabalhos e tarefas relacionadas com o desenvolvimento.

- **Capítulo 10** – Conclusões Finais:
  conclusões finais sobre todo o processo elaborado.

2 Motores de Jogos

No ramo de desenvolvimento de videojogos é bastante comum ouvir falar no termo "Motor de Jogo". Esta palavra originou-se no meio da década de 1990 e pode ser superficialmente descrita como um programa de computador que abstrai o desenvolvimento de videojogos (Eberly, 2006). A associação ao nome foi estabelecida em união com alguns lançamentos de jogos em 3D da época, mas hoje em dia possui uma exposição bastante mais detalhada e complexa.

O desenvolvimento de videojogos tem-se tornado cada vez mais num procedimento sofisticado, e nesse sentido a existência dos motores de jogos é fundamental na simplificação do mesmo. Tipicamente, um motor de jogo dispõe de várias funcionalidades abstratas que permitem aos desenvolvedores produzir com agilidade diferentes tipos de jogos. Porém, alguns podem ser orientados a um género em particular, como por exemplo, tiro em primeira pessoa (também conhecido como *First Person Shoter em inglês*).

Na atualidade, existem vários motores de jogos que possuem funcionalidades e especificações que os fazem distinguir dos demais. Um dos critérios mais relevantes é a dimensão espacial disponibilizada pelo motor. Alguns são construídos com a finalidade de desenvolver jogos somente em 3D enquanto outros são mais focados em 2D. Por outro lado, também existem alguns motores híbridos que acabam por permitir ambas as dimensões, embora por vezes acabem por não se evidenciar em nenhuma. Além disso, existem muitas outras diferenças, o que pode tornar a escolha de um motor de jogos numa tarefa complicada quando não se sabe exatamente o que procurar (Lewis & Jacobson, 2002).

Com o aumento progressivo da sofisticação nos videojogos, as bases tecnológicas impostas aos motores de jogos seguem naturalmente o mesmo fluxo. Algumas das capacidades tais como a aptidão de executar videojogos num certo tipo de ambiente (mobile, web, nativo...) acabam por ser implementadas com base nas tendências e necessidades atuais. É compreensível que os desenvolvedores queiram que os seus videojogos estejam disponíveis nas plataformas mais utilizadas e por isso é importante compreender e atender estes requisitos de modo a alcançar uma percentagem elevada de utilizadores satisfeitos.

2.1 Funcionalidades habituais

Apesar de existir uma grande pluralidade de alternativas, existe tipicamente um conjunto de funcionalidades que está presente na maioria dos motores de jogos. Estas funcionalidades normalmente fazem parte do núcleo da arquitetura de um motor e são habituais aos desenvolvedores.

2.1.1 Renderização

Uma das funcionalidades essenciais que um motor de jogo proporciona é a capacidade de renderizar figuras 2D ou objetos 3D num ecrã eletrónico. O processo de renderização é bastante complexo e é normalmente realizado por bibliotecas destinadas a esse efeito tal como o *OpenGL,*

WebGL, Vulcan, DirectX e outros. No entanto, toda a lógica e tratamento dos objetos e figuras é realizado pelo motor de jogo (Eberly, 2006). As bibliotecas disponibilizam uma interface de programação (API) que é utilizada na realização das operações relacionadas com processo de renderização.

Nota: como pode ser observado nos capítulos seguintes, a biblioteca de renderização utilizada neste desenvolvimento é o *WebGL*, e nesse sentido a documentação apresentada irá incidir especialmente sobre essa *framework.*

Tipicamente existe uma sequência de passos necessários para a renderização de objetos a qual chamamos de pipeline de renderização (o processo vai ser explicado com maior detalhe em capítulos posteriores) (Eberly, 2006). A ordem e a execução de cada um dos passos varia consoante a biblioteca de renderização utilizada, mas na maior parte dos casos passa por agregar e injetar a informação sobre os objetos que desejamos apresentar.

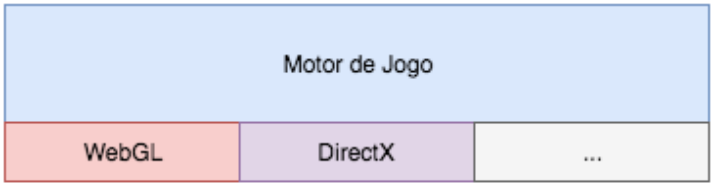

Figura 1 - exemplo de emparelhamento com diferentes bibliotecas de renderização

O motor de jogo deve possuir uma integração eficiente com pelo menos uma das bibliotecas de renderização existentes e facultar a sua funcionalidade de forma transparente ao utilizador. A camada de abstração deve ser executada de forma a que não existam implicações com outras bibliotecas caso seja necessário permutar.

2.1.2 Áudio

Torna-se cada vez mais difícil imaginar um videojogo sem áudio, desde som ambiente a efeitos sonoros, a utilização desta componente é uma particularidade comum em praticamente todos os jogos. Tal como no cenário da renderização, existem algumas bibliotecas (como por exemplo OpenAL) que abstraem, através de uma API, o acesso aos dispositivos de áudio emparelhados.

2.1.3 Controladores de entrada

Existe uma grande variedade de modelos de entrada hoje em dia. Desde a possibilidade em usar um joystick ou até mesmo o "*touch*" de um smartphone, existem várias formas de interagir com videojogos. É de a responsabilidade do motor de jogo possibilitar a sua utilização em diferentes contextos (Gregory, 2009). Dependendo da plataforma onde o jogo é executado, alguns controladores podem não estar disponíveis por não serem compatíveis com o ambiente. Os motores de jogos que permitem exportar os jogos para várias plataformas devem ter essa noção e fornecer meios de teste quando aplicável. Por exemplo, ao desenvolver um videojogo mobile num desktop, devem existir mecanismos que simulem a interação do dispositivo.

2.1.4 Sistemas de colisão

É cada vez mais uma prática comum incluir um ou vários sistemas de colisão em motores de jogos. O objetivo principal é permitir detetar quando um objeto entra em colisão com outro, para que dessa forma seja possível desencadear alguma ação correspondente. Alguns sistemas de colisão chegam a ser simuladores complexos que aplicam leis da física aos objetos virtuais de jogo (Bourg, 2002) de forma a simular o seu comportamento natural (podendo configurar alguns parâmetros tais como rigidez, gravidade, massa...).

Felizmente para quem está a desenvolver um motor de jogo, não existe a necessidade de implementar de base um sistema de colisão complexo. Existem várias bibliotecas disponíveis tais como o *Box2D*, *MatterJS* entre outros, que podem ser integrados no motor sem grandes dificuldades e já incluem sistema de física. Tipicamente o funcionamento de um sistema de física passa por incluir um mundo virtual ao qual podem ser adicionados objetos com determinadas características. O motor de jogo é responsável por fazer coincidir os objetos já pertencentes no modelo dos cenários com os objetos existentes no mundo físico virtual.

2.1.5 Editor

Alguns motores de jogos não possuem uma interface de utilizador e toda a sua utilização é realizada por código usando a API fornecida. Com o intuito de auxiliar e agilizar o desenvolvimento de videojogos, alguns motores de jogos fornecem também um editor que permite criar de forma visual os cenários de jogo. Além de facilitar algumas tarefas em geral, também permite que utilizadores sem conhecimento de programação possam contribuir no desenvolvimento dos jogos. É típico que game designers utilizem estes editores de forma a criar níveis para os seus jogos, sendo que durante o processo não costumam ter a necessidade de programar.

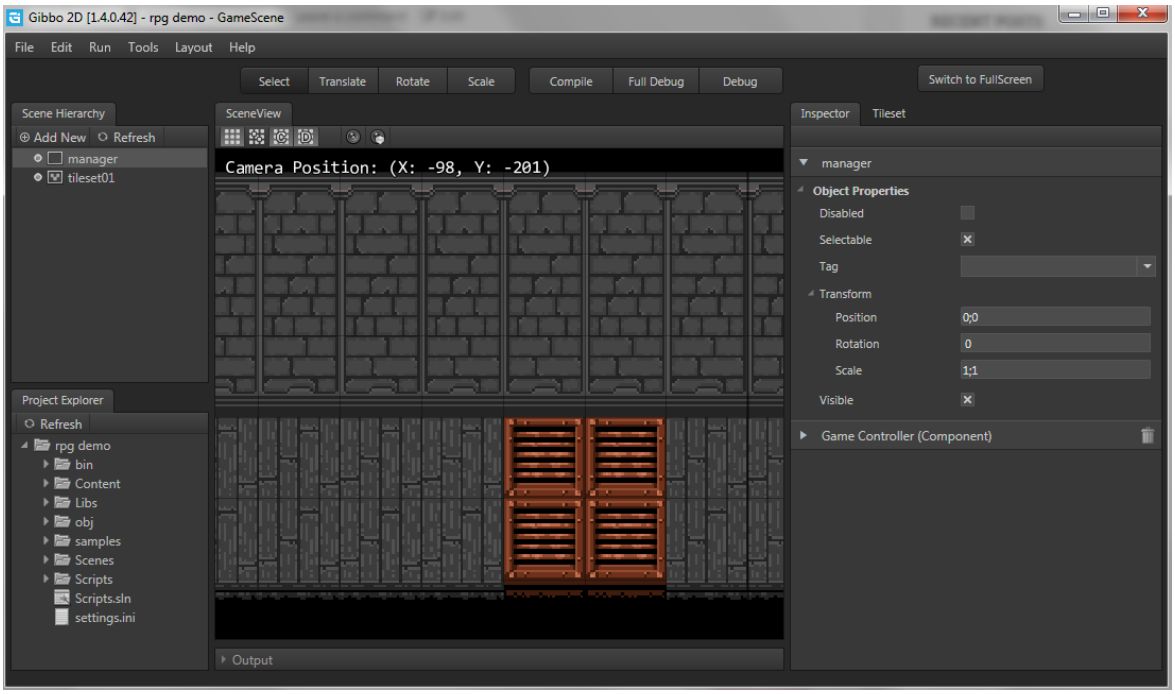

Figura 2 - exemplo de um editor de jogo - (Gibbo2D)

Os editores de motor de jogo são algumas vezes confundidos com editores de níveis. Embora sejam de certa forma parecidos, um editor de jogo permite um leque de funcionalidades mais elevado podendo ser compreendido como uma ferramenta que permite manipular todo o projeto e não apenas editar/criar níveis (Eberly, 2006). Normalmente os editores de níveis são distribuídos com os videojogos para que os jogadores possam criar níveis enquanto que os editores raramente são publicados.

Das funcionalidades mais comuns num editor podemos destacar a possibilidade de executar e depurar uma sessão de jogo, criar/editar cenários, gerir o conteúdo associado (imagens, scripts, ficheiros...), interagir visualmente com os objetos virtuais de jogo e a capacidade de exportar o projeto para diferentes plataformas.

Infelizmente são raros os casos em que os editores dos motores de jogo se propagam em diferentes sistemas operativos, normalmente estão restritos a apenas um devido a limitações nas tecnologias utilizadas.

2.1.6 Scripting

O termo 'scripting' é por base dirigido a linguagens de extensão que são executadas no interior de programas ou até mesmo de outras linguagens de programação. As linguagens de scripting são utilizadas para estender a funcionalidade de uma aplicação, e no caso de um motor de jogo, são aplicadas regularmente de forma a criar comportamentos e ações dinâmicas nos cenários de jogo (Gregory, 2009).

Na lista seguinte são apresentadas algumas das linguagens de scripting que são normalmente utilizadas em motores de jogos:

- Javascript
- Lua
- Python
- Ruby
- ActionScript

A integração de uma linguagem de scripting é por vezes associada a editores externos, como por exemplo o Visual Studio, que já possuem mecanismos e funcionalidades avançadas para o efeito.

Apesar de não ser muito comum, também existem formas de criar estes scripts usando uma interface visual (mais concretamente Visual Scripting). Este último, permite que utilizadores que não saibam programar possam implementar comportamentos e ações mais complexas não estando restritas às oferecidas nativamente pelo editor.

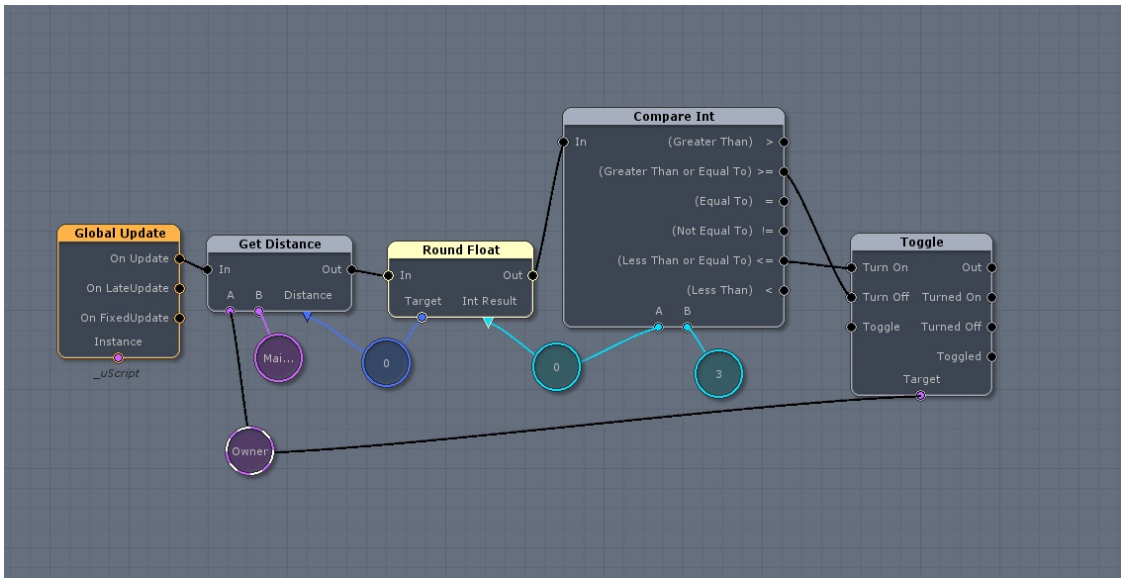

Figura 3 - exemplo de visual scripting (Unreal Engine Visual Scripting)

Os scripts são normalmente compilados em tempo de execução o que permite atualizar o seu comportamento assim que for desejado. Em ambientes de desenvolvimento é uma característica muito importante pois tornar-se-ia complicado reiniciar todo o motor de jogo sempre que um script fosse editado.

A tendência atual que existe sobre scripting recai principalmente sobre projetar módulos que podem ser atribuídos a objetos virtuais no jogo. Estes módulos podem ter ou não propriedades que idealmente possam ser modificadas no editor. Desta forma é possível criar cenários com comportamentos semelhantes que podem ser replicados em diferentes objetos sem grandes dificuldades.

2.2 Soluções relacionadas

Na atualidade existem bastantes soluções disponíveis no que toca ao desenvolvimento de videojogos. Com o objetivo de identificar as principais características dos motores de jogos existentes, foi elaborada uma pesquisa que envolve as tecnologias mais utilizadas e/ou que possuem funcionalidades relevantes ao âmbito deste documento.

2.2.1 Unity3D

O *Unity3D* é um motor de jogo consideravelmente flexível que é utilizado na criação de jogos 2D e 3D que podem ser exportados para uma grande variedade de plataformas. Uma das grandes vantagens que possui é a sua comunidade crescente que tem dado suporte ao seu desenvolvimento (Creighton, 2010).

Em geral podemos identificar que é um motor de jogo otimizado à agilidade de produção oferecendo um fluxo de trabalho direcionado para esse propósito. Das suas várias funcionalidades, podemos destacar os seguintes aspetos principais:

- Editor de cenários
- Componente scripting
- Sistema de física
- Capacidade de exportação para múltiplas plataformas
- Possibilidade de criação de jogos em 2D e 3D
- Ferramentas de animação
- Documentação apropriada

Uma vez que fornece um leque de opções elevado acaba por tornar o ambiente bastante complexo e a documentação vasta. Seria interessante se a interface de utilizador pudesse "vestir" uma variante mais básica que incluísse apenas as funcionalidades essenciais. Desta forma o utilizador poderia ir desbravando pouco a pouco todas as possibilidades existentes.

Atualmente os jogos criados no Unity3D podem ser exportados para mais de 20 plataformas distintas incluindo as seguintes:

- Windows
- Linux
- MacOS
- ChromeOS
- Playstation
- Android
- iOS
- WiiU
- Xbox

Os desenvolvedores de jogos podem escolher entre Javascript e C# na criação de scripts . Os módulos criados são depois atribuídos a objetos de jogo em forma de componentes. É possível expor propriedades públicas nesses componentes que podem ser alteradas no editor conforme desejado. Não existe muita documentação sobre o fluxo interno do motor de jogo pois está em código fechado, toda a extensão feita pelos desenvolvedores é feita pelos componentes (Creighton, 2010).

Atualmente existem algumas características menos boas no Unity3D que incluem, por exemplo, o sistema de *networking* integrado e o sistema gráfico de utilizador (GUI) nos jogos que continuam com bastantes limitações. Existem também alguns problemas relatados do editor em alguns ecrãs de elevada resolução (por ex. ecrãs retina) em que apresenta de forma distorcida o conteúdo do mesmo. É importante notar também que o editor está disponível em Windows e MacOS deixando de parte ambientes Linux.

O Unity3D possui duas licenças, uma versão de comunidade e uma versão profissional. A licença de comunidade está disponível a qualquer utilizador pessoal e entidades legais que não possuam uma receita anual bruta superior a US$100,000. A licença profissional possui todas as funcionalidades da versão gratuita e inclui funcionalidades extra como análise de performance, "cloud-sync" e uma maior customização do editor.

2.2.2 Phaser.io

Phaser.io é um motor de jogo 2D construído em *Javascript*. Este não possui um editor e toda a sua funcionalidade é disponibilizada por uma API de programação. A biblioteca de renderização utilizada é o *WebGL* o que permite executar os videojogos em qualquer ambiente web. Atualmente possui várias características estimulantes sendo que se destacam as seguintes:

- Suporte *multi-touch*
- Filtros de renderização
- Suporte a *spritesheets*
- Renderização de texto em tempo de execução
- Facilidade de aprendizagem
- Documentação apropriada

Este motor é conhecido por não causar grande fadiga no que toca a decisão de implementação em videojogos simples e também por ser fácil de aprender. Em poucas horas é possível estudar grande parte das funcionalidades podendo mesmo começar a criação de um jogo sem grandes dificuldades.

Como no *Unity3D*, a comunidade do *Phaser.io* é também um ponto forte do motor. Nos últimos anos tem existido um envolvimento bastante interessante da comunidade que além de contribuírem com sugestões, podem contribuir com implementações no próprio motor pois está disponível em código aberto.

Não existe nenhuma característica em especial no que toca ao licenciamento, a sua utilização pode ser utilizada livremente sem qualquer custo.

2.2.3 Gibbo2D

O *Gibbo2D* tal como o Unity3D é um motor de jogo "tudo-em-um" com editor que disponibiliza um conjunto variado de funcionalidades. Este motor foi desenvolvido pelo o autor deste documento em 2013 com o objetivo de criar um ambiente exclusivo a criação de jogos 2D com integração a APIs de terceiros.

O editor inclui várias ferramentas que permitem um desenvolvimento ágil de videojogos tendo como principal limitação as plataformas de exportação que apenas incluem ambientes Windows e Linux (também é possível a exportação para Android, mas com um custo associado a uma biblioteca de terceiros necessária para a execução do mesmo).

Este motor fornece um sistema de scripting inspirado ao que existe disponível no Unity3D. É possível programar blocos de código em C# que podem ser então associados a objetos de jogo. Tal como no Unity3D, as propriedades podem ser alteradas no editor sem existir a necessidade de recompilar ou reiniciar o motor de jogo. Ao contrário do Unity3D que necessita de um editor externo para a criação e edição de scripts, o Gibbo2D oferece a possibilidade de utilizar um editor de programação interno para o efeito. Em suma, existem algumas ferramentas que se destacam no que toca ao desenvolvimento de videojogos em 2D:

- Mapeamento 2D especializado
- Tilesets
- Animação de sprites
- Sistema de partículas
- Sistema de física
- Documentação apropriada

O *Gibbo2D* tal como o Phaser.io está disponível em código aberto (desde 2014) e qualquer pessoa pode contribuir para o seu desenvolvimento. Não existem restrições sobre licenciamento, qualquer utilizador pode desenvolver sem qualquer custo associado.

3 Conceitos científicos

Neste capítulo vamos abordar os conceitos científicos de base associados ao desenvolvimento de videojogos. É assumido que o leitor é familiarizado com conhecimentos de álgebra, matrizes e trigonometria. De forma a que a seguinte documentação não seja demasiado extensa, não será abordado com detalhe extremo os temas abordados. Será apresentado, no entanto as noções básicas necessárias para um desenvolvimento apropriado e audaz.

3.1 Sistema de Coordenadas

É bastante comum a utilização de tuplos em desenvolvimento de videojogos. Tipicamente são utilizados tuplos de três e dois elementos para apresentar coordenadas 3D e 2D respetivamente no mundo virtual (Eberly, 2006). A sua representação é normalmente da seguinte forma:

- 2D: (x, y)
- 3D: (x, y, z)

Estes componentes são normalmente identificados como coordenadas cartesianas de um ponto. Este sistema de coordenadas é facilmente compreendido, mas possui alguns aspetos interessantes associados, não podemos esquecer que apesar de existirem coordenadas em 3D no mundo virtual, o resultado final de renderização será sempre a duas dimensões. Para obter esse efeito são aplicadas várias transformações matemáticas com o objetivo de simular o efeito de profundidade (Dunn & Parberry, 2015).

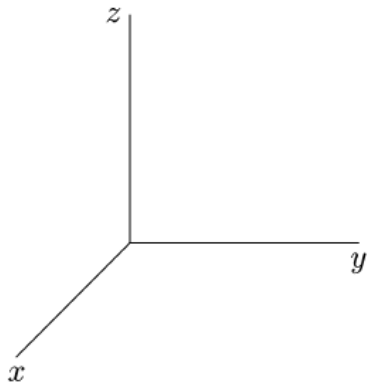

Figura 4 - exemplo de um eixo com três dimensões

No mundo virtual, existem vários domínios de coordenadas, os mais importantes são o espaço de modelo (o espaço que cada objeto ocupa no cenário), o espaço do mundo (espaço do cenário de jogo que inclui todos os objetos) e o espaço de câmara (também chamado de projeção virtual que inclui o espaço do que vai ser apresentado no cenário).

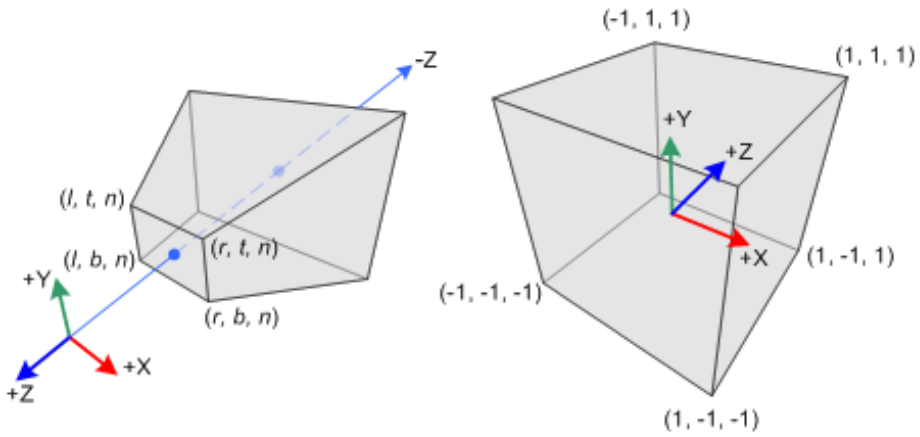

Figura 5 - projeção de câmara (esquerda) e um modelo de objeto (direita) - (OpenGL Projection Matrix)

Um aspeto interessante sobre a dimensão e modelo de um objeto é que além do seu espaço, tipicamente é definido um ponto de origem sobre qual todas as operações realizadas vão ser aplicadas. Por exemplo, ao efetuar a rotação sobre um objeto, o seu resultado é sempre sobre a sua origem. A posição não necessita de ser o centro do objeto, a origem pode ser alterada conforme necessário.

Toda esta explicação é compreendida de forma semelhante em cenários 2D. Em 3D existe uma maior complexidade associada à adição de uma terceira coordenada. Além dos objetos possuírem construções mais detalhadas, as câmaras (visão de jogo) também são mais complexas pois incluem a noção de profundidade.

Tal como em programação, podemos associar matematicamente tuplos de coordenadas a identificadores com o objetivo de simplificar a escrita de funções:

Exemplos:

- $\mathrm{P} = (\mathrm{x}, \mathrm{y}, \mathrm{z})$
- $\mathrm{B} = (10, 30)$

Um sistema de coordenadas com três dimensões pode ser apresentado sucintamente com o seguinte formato:

$$\{O; V, A, D\}$$

Os identificadores em cima, representam a origem e os vetores de direção de um sistema de coordenadas 3D:

- ***O*** = ponto de origem
- ***V*** = direção de visualização
- ***A*** = direção acima (a direção do topo do sistema)
- ***D*** = direção do sistema (a direção para onde o sistema está volvido)

Qualquer ponto referente a este sistema pode ser representado da seguinte forma:

$$X = O + vV + aA + dD$$

Os identificadores ***v***, ***a*** e ***d*** são escalares que medem a relação do movimento com a direção necessária para chegar ao ponto ***X***. O tuplo (v, a, d) representa as coordenadas do ponto ***X*** relativamente as coordenadas do sistema. Estamos então a envolver sobre um ponto relativo, o que significa que a sua posição é somada ao sistema em que pertence.

Em duas dimensões, um sistema de coordenadas pode ser descrito de forma ainda mais sucinta:

$$\{O; X, Y\}$$

Os identificadores em cima, representam a origem e os vetores de direção correspondentes a cada eixo existente no sistema:

- ***O*** = ponto de origem
- ***X*** = direção no eixo X
- ***Y*** = direção no eixo Y

Semelhantemente, a representação de qualquer ponto relativo ao eixo pode ser representada da seguinte forma e a sua explicação recai sobre a que já foi explicada no caso a três dimensões:

$$M = O + xX + yY$$

3.2 Pontos e Vetores

Em primeira instância, é importante saber que um ponto não é um vetor nem um vetor é um ponto. Apesar de poderem ser representados com o mesmo formato, existe uma grande diferença que distingue ambos. Os pontos podem ser descritos de forma breve como coordenadas num espaço e os vetores como deslocações no espaço (Dunn & Parberry, 2015). Em analogia, podemos equiparar o conceito de tempo. Mais concretamente, podemos descrever pontos como o tempo num determinado instante e os vetores como durações de tempo. É importante saber também que as diferenças de deslocação podem ser negativas, ou seja, os valores de deslocação não precisam de ser necessariamente positivos.

Sabendo a distinção entre ambos, conseguimos deduzir os seguintes princípios:

- A soma entre um ponto e um vetor é um ponto.
- A diferença entre dois pontos é um vetor de deslocação.
- Ao adicionar dois vetores obtemos uma deslocação composta.

Em matemática, um vetor normalmente possui um identificador com uma seta em cima (Dunn & Parberry, 2015). O símbolo tem como objetivo indicar que o tuplo representa uma deslocação:

$$\overrightarrow{V} = (\mathbf{10}, \mathbf{20})$$

3.3 Matrizes

Uma matriz pode ser descrita como um conjunto de números (positivos ou negativos) ou expressões distribuídas em várias linhas e colunas (Dunn & Parberry, 2015). Os itens individuais de uma matriz são tipicamente chamados de elementos ou entradas.

A ordem (***o***) de uma matriz informa o seu tamanho e indica a quantidade de linhas (***m***) e colunas (***n***) nela contida:

$$\boldsymbol{o = m * n}$$

Quando uma matriz possui o mesmo número de linhas e colunas chamamos de matriz quadrada (Dunn & Parberry, 2015). Dessa forma, podemos dizer que ***(m)*** tem a mesma quantidade de elementos que ***(n)***.

Simbolicamente, uma matriz 3x3 possui normalmente a seguinte forma:

$$\begin{bmatrix} \mathbf{-30} & \mathbf{2} & \mathbf{2} \\ \mathbf{5} & \mathbf{84} & \mathbf{-4} \\ \mathbf{4} & \mathbf{44} & \mathbf{60} \end{bmatrix}$$

As matrizes podem também conter apenas uma coluna ou uma linha, a estas chamamos de matriz de coluna ou matriz de linha respetivamente. As seguintes matrizes representam simbolicamente uma ordem de 1x3 (linha) e 3x1 (coluna) respetivamente:

$$\begin{bmatrix} \mathbf{1} & \mathbf{2} & \mathbf{3} \end{bmatrix} \qquad \begin{bmatrix} \mathbf{3} \\ \mathbf{2} \\ \mathbf{1} \end{bmatrix}$$

Em videojogos as matrizes são utilizadas para representar transformações tais como deslocação, rotação e dimensão (Eberly, 2006). A razão prática pela qual as matrizes são utilizadas para esse efeito é a rapidez de cálculo nos dispositivos atuais. Como é expectável, a execução de um jogo é um processo bastante intensivo e qualquer otimização aplicável é fundamental. Outra grande vantagem que existe na utilização de matrizes é que as transformações (descritas em maior detalhe no subcapítulo seguinte) podem ser combinadas as vezes que forem necessárias desde que a ordem de operação seja respeitada (Dunn & Parberry, 2015). Ou seja, podemos representar o resultado de uma deslocação, rotação e dimensão com apenas uma matriz após a execução de todas as operações.

A operação usada para combinar matrizes com o objetivo de representar as transformações de um objeto é a multiplicação. Supondo que o identificador ***D*** corresponde ao valor de deslocação e o identificador ***R*** corresponde ao valor de rotação a aplicar num objeto, podemos dizer que o identificador ***G*** representa a combinação de ambas as operações:

$$\boldsymbol{G = D * R}$$

O resultado desta combinação pode ser então aplicado à matriz de visualização de um objeto. A matriz de visualização de um objeto representa o valor original da transformação de um objeto no espaço (sem nenhuma transformação aplicada).

A ordem pela qual são efetuadas as multiplicações tem impacto no resultado e por isso deve-se efetuar uma combinação válida. Tipicamente a ordem aplicada quando se pretende executar as transformações (deslocação, rotação e dimensão) é a seguinte:

$$\boldsymbol{L = Deslocação * Rotação * Dimensão}$$

Note que em matrizes, a operação de multiplicação é feita pela ordem inversa (Dunn & Parberry, 2015), ou seja, primeiro é efetuado o cálculo da dimensão, depois o da rotação e por fim a deslocação.

3.4 Transformações algébricas

As transformações algébricas possuem uma elevada relevância no desenvolvimento de videojogos sendo usadas para deslocar coordenadas no espaço virtual. Note-se que como o objetivo deste documento não é sobre instrução em álgebra linear, apenas iremos abordar as transformações mais empregadas.

A utilização de transformações lineares permite aplicar alterações em objetos tais como rotação, deslocação e dimensão (Eberly, 2006). No desenvolvimento de videojogos existem bibliotecas especificas responsáveis por estes cálculos, mas é importante saber os fundamentos associados pois a sua aplicação é constante.

As transformações lineares são normalmente aplicadas sobre vetores de deslocação e possuem a seguinte forma simbólica:

$$\boldsymbol{B = F(A)}$$

A equação afirma que para uma função de transformação linear ***F(x)*** com o vetor de entrada ***A*** o vetor de saída será ***B***. Cada um destes elementos (a função e os dois vetores) pode ser representado com uma matriz. O vetor ***B*** com uma matriz de ordem 1x3, o vetor ***A*** também com uma matriz de ordem 1x3 e a transformação linear ***F*** com uma matriz de ordem 3x3 (matriz de transformação).

Em extensão podemos representar a mesma equação da seguinte forma:

$$\begin{bmatrix} \boldsymbol{B_0} \\ \boldsymbol{B_1} \\ \boldsymbol{B_2} \end{bmatrix} = \begin{bmatrix} \boldsymbol{F_{00}} & \boldsymbol{F_{01}} & \boldsymbol{F_{02}} \\ \boldsymbol{F_{10}} & \boldsymbol{F_{11}} & \boldsymbol{F_{12}} \\ \boldsymbol{F_{20}} & \boldsymbol{F_{21}} & \boldsymbol{F_{22}} \end{bmatrix} \begin{bmatrix} \boldsymbol{A_0} \\ \boldsymbol{A_1} \\ \boldsymbol{A_2} \end{bmatrix}$$

Note-se que para adicionar a operação de rotação ou dimensão, outros vetores e respetivos valores devem ser adicionados aos elementos de entrada.

A operação de rotação por definição corresponde ao movimento circular de um objeto sobre um ponto (origem). Como foi referido anteriormente, o ponto de origem não precisa de ser

necessariamente o centro do objeto virtual, mas incide sempre sobre quaisquer dois eixos (plano XY, XZ ou YZ).

Dependendo do plano utilizado na rotação, existem diferentes formas de aplicar a transformação. Em duas dimensões o mais comum é a utilização do plano XY que pode ser representado da seguinte forma:

$$\textbf{\textit{Rotação}} = \begin{bmatrix} \boldsymbol{cos\,\theta} & \boldsymbol{-sin\,\theta} & \mathbf{0} \\ \boldsymbol{sin\,\theta} & \boldsymbol{cos\,\theta} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} \end{bmatrix}$$

Nesse sentido, se pretendermos rodar um ponto ***A*** no plano XY em 45º (em radianos: $\frac{\pi}{4}$) seria necessária a seguinte operação:

$$\begin{bmatrix} \boldsymbol{B_0} \\ \boldsymbol{B_1} \\ \boldsymbol{B_2} \end{bmatrix} = \begin{bmatrix} \boldsymbol{cos(\pi/4)} & \boldsymbol{-sin(\pi/4)} & \mathbf{0} \\ \boldsymbol{sin(\pi/4)} & \boldsymbol{cos(\pi/4)} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} \end{bmatrix} \begin{bmatrix} \boldsymbol{A_0} \\ \boldsymbol{A_1} \\ \boldsymbol{A_2} \end{bmatrix}$$

No caso das transformações de dimensão de um objeto, o processo é relativamente mais simples. A operação requer um tuplo com o mesmo número de elementos da dimensão aplicada. Cada elemento deve representar o valor de dimensão representativo ao eixo. No caso de duas dimensões, o elemento de entrada apenas deve incluir os valores relativos ao eixo X e Y. Em três dimensões deve-se incluir também o eixo de profundidade.

Simbolicamente a matriz de transformação de dimensão pode ser apresentada da seguinte forma, sendo que ***D*** corresponde ao tuplo de entrada:

$$\textbf{\textit{Dimensão}} = \begin{bmatrix} \boldsymbol{D_0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{D_1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \boldsymbol{D_2} \end{bmatrix}$$

Nesse sentido, se pretendermos triplicar o tamanho no eixo X mantendo o valor original nos restantes eixos, ficaríamos com a seguinte operação:

$$\boldsymbol{D} = (\mathbf{3}, \mathbf{1}, \mathbf{1}) \qquad \begin{bmatrix} \boldsymbol{B_0} \\ \boldsymbol{B_1} \\ \boldsymbol{B_2} \end{bmatrix} = \begin{bmatrix} \boldsymbol{D_0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{D_1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \boldsymbol{D_2} \end{bmatrix} \begin{bmatrix} \boldsymbol{A_0} \\ \boldsymbol{A_1} \\ \boldsymbol{A_2} \end{bmatrix}$$

3.5 Rasterização

A rasterização é uma das técnicas mais comuns quando se pretende apresentar visualmente imagens de cenários virtuais em tempo de execução. Um dos objetivos principais envolvidos no processo é solucionar o problema que existe em identificar os objetos que devem ser apresentados com base na câmara de visualização aplicada.

Os elementos que a maioria dos motores de renderização utilizam são segmentos de linhas e triângulos, embora por vezes (raramente) também sejam utilizados círculos e elipses. A explicação sobre a execução completa deste processo é bastante extensa, neste subcapítulo apenas vai ser abordado de forma superficial as características articuladas.

3.5.1 Segmentos de linha

Um segmento de linha representa a ligação entre dois pontos. Sabendo que estamos sempre a operar num resultado que vai ser apresentado em pixéis, as decisões de apresentação são calculadas de forma a obter o segmento mais equilibrado possível. O algoritmo de decisão processa a quantidade de pixéis preenchidos com base no grau de magnitude de declive da linha.

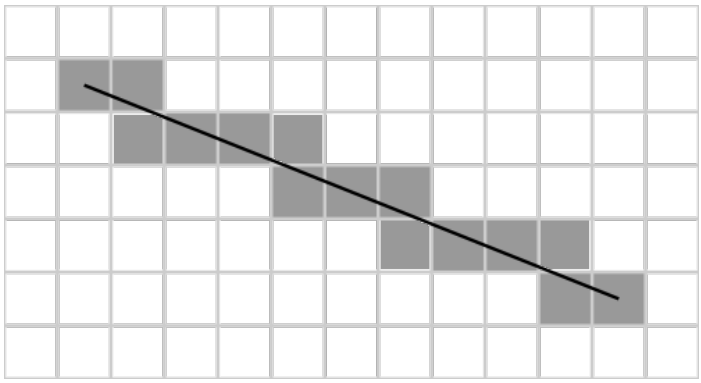

Figura 6 - exemplo de renderização de um segmento de linha

3.5.2 Triângulos

Os triângulos são a forma mais eficiente e comum de simular a representação de objetos virtuais num ecrã a duas dimensões. A sua renderização requer três pontos que são utilizados para definir os cantos do triângulo. Em primeira instância, são utilizados segmentos de linha para unir os três vértices do triângulo. De seguida, o interior do triângulo é preenchido com uma cor:

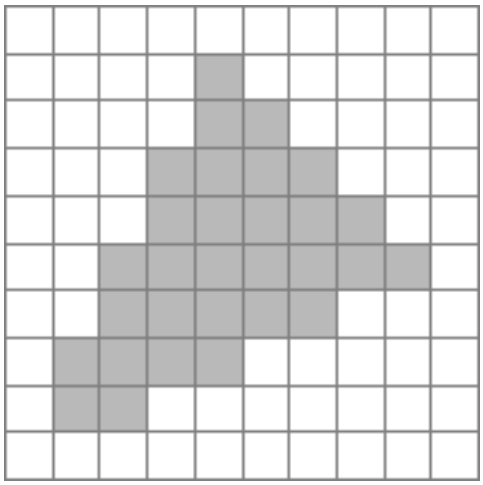

Figura 7 - exemplo de renderização de um triângulo

3.5.3 Anti-aliasing

O *anti-aliasing* é uma técnica utilizada para minimizar o efeito de serrilha (*aliased*) que é observável ao desenhar uma reta com declive (Hill & Kelley, 2007). Uma vez que a divisão unitária

de um ecrã é em pixéis, uma reta é na verdade um conjunto de quadrados o que resulta num efeito “serrado” ao longo da linha apresentada.

Figura 8 - exemplo de uma apresentação com *aliasing* (esquerda) e *anti-aliasing* (direita)

Alguns exemplos de técnicas relevantes de *anti-aliasing* existentes na atualidade são: *MSAA* (**M**ulti**S**ample **A**nti-**A**liasing), *FXAA* (**F**ast Appro**x**imate **A**nti-**A**liasing), *SSAA* (**S**uper **S**ampling **A**nti-**A**liasing) e *TXAA* (**T**emporal **A**nti-**A**liasing).

4 Scarlett Game Studio Design

Planear a estrutura de um motor de jogo é uma tarefa bastante extensa, é preciso ter em consideração não apenas os requisitos funcionais, mas também todas as dependências e limitações associadas às tecnologias que vão ser utilizadas.

Em primeira instância, vamos analisar os requisitos e módulos a implementar no Scarlett Game Studio. De seguida vamos descrever a API de interação que os desenvolvedores irão ter acesso de forma a construírem os seus jogos. Por fim vamos efetuar uma análise com o objetivo de validar todos os componentes envolvidos no desenvolvimento.

4.1 Objetivos de desenvolvimento

No âmbito desta dissertação, vai ser apresentada a documentação de desenvolvimento de um motor de jogos 2D chamado Scarlett Game Studio.

Pretende-se que o motor de jogo contenha as seguintes características principais:

- Capacidade de criar e gerir projetos de videojogos em duas dimensões (2D), usando funcionalidades como: scripting, exportação de projetos, editor visual de mapas, editor de scripts.
- Capacidade de execução e exportação em diferentes sistemas operativos, nomeadamente: Windows, Mac OS e Linux.

- Possuir uma interface de utilizador que respeita características de usabilidade propicias a um desenvolvimento prático, tais como: clareza, customização, concisão, familiarização (semelhanças com aplicações já existentes) e responsividade.

- Possibilidade de integração com outros sistemas (por ex. permitir usar um editor de programação externo na construção de scripts).

- Possibilidade de guardar os projetos remotamente para que possam ser consultados em qualquer lugar.

4.2 Infraestrutura

O Scarlett Game Studio tem como objetivo ser uma solução extensa e completa no desenvolvimento de videojogos. Nesse sentido e pela motivação de criar uma aplicação modular, a arquitetura deste desenvolvimento vai ser dividida por vários módulos principais distintos.

No lado de servidor será desenvolvido um serviço responsável por executar os pedidos realizados por parte do editor. Toda a informação deverá ser guardada numa base de dados que poderá estar ou não alocada em múltiplos sistemas. Além disso, estará disponível um sistema de ficheiros que

guarda todos arquivos dos projetos criados no motor caso o utilizador assim o pretenda.

No lado de cliente será desenvolvida uma framework que contém todas as classes de negócio do motor de jogo. A framework poderá ser utilizada individualmente o que implica o desenvolvimento de uma API. As funcionalidades expostas poderão ser utilizadas também por um editor com interface de utilizador, sendo um dos módulos a implementar.

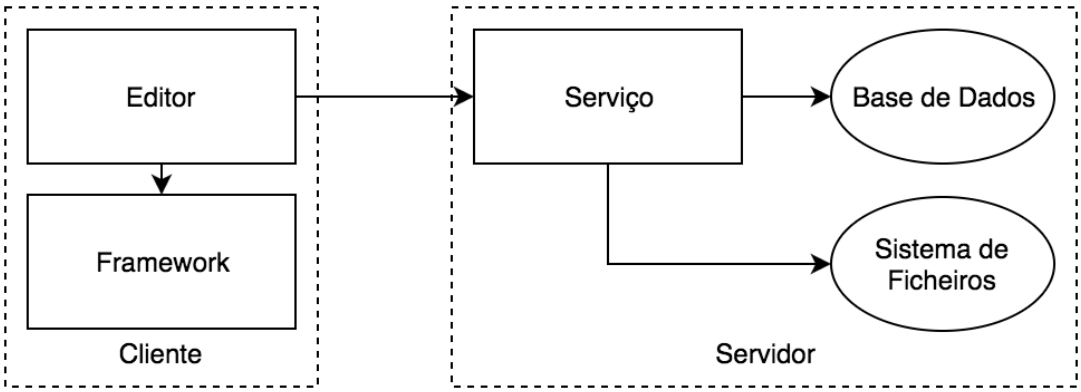

Figura 9 - Infraestrutura do Scarlett Game Studio

Nos capítulos seguintes irá ser apresentada a documentação extensiva sobre cada um dos módulos expostos na Figura 9.

4.3 Requisitos funcionais

A seguinte lista apresenta os requisitos funcionais e contém as funcionalidades principais do editor Scarlett Game Studio. A lista foi construída não só com base sobre o que existe atualmente em sistemas relacionados, mas também com o objetivo de trazer novos mecanismos de desenvolvimento.

- **Criar projeto**

  Esta funcionalidade permite criar novos projetos sem conteúdo. Aquando esta ação está a ser executada, deverá ser criada automaticamente a estrutura do projeto na pasta selecionada.

- **Guardar projeto**

  Esta funcionalidade permite guardar o estado num determinado instante de desenvolvimento de um projeto.

- **Carregar projeto**

  Esta funcionalidade permite carregar projetos guardados anteriormente. Sendo assim, é possível criar um jogo em várias sessões de desenvolvimento.

- **Exportar projeto**

Esta funcionalidade permite exportar jogos desenvolvidos no Scarlett Game Studio para uma plataforma desejada (Web, nativo...).

- **Visualizar conteúdo do projeto**

Esta funcionalidade permite visualizar o conteúdo associado ao projeto. Esta visualização deve ser apresentada de forma hierárquica num painel demonstrativo com os ficheiros do projeto.

- **Importar conteúdo**

Esta funcionalidade permite adicionar conteúdo externo ao jogo. O conteúdo adicionado pode ser variado e inclui imagens, scripts, áudio, etc.

- **Remover conteúdo**

Esta funcionalidade permite remover conteúdo associado ao jogo.

- **Mudar nome de ficheiros pertencentes ao conteúdo de jogo**

Esta funcionalidade permite alterar o nome de ficheiros e pastas associados ao jogo.

- **Compilar scripts**

Esta funcionalidade permite atualizar e compilar os scripts adicionados ao jogo em tempo de execução.

- **Adicionar cenários de jogo**

Esta funcionalidade permite adicionar novos cenários ao projeto. Cada cenário deverá ter a capacidade de guardar e gerir os objetos virtuais de jogo.

- **Adicionar objetos de jogo**

Esta funcionalidade permite adicionar novos objetos a um cenário de jogo. Deverão existir Templates para cada tipo de objeto com a intenção de separar a lógica associada a cada um (por ex. objetos de texto, imagens, áudio...). É importante que seja possível adicionar de forma hierárquica os objetos de jogo de forma a existirem objetos com maior complexidade.

- **Remover objetos de jogo**

Esta funcionalidade permite remover objetos que estejam associados a um cenário de jogo. O cenário em questão deve estar aberto para que seja possível remover o objeto.

- **Selecionar objetos de jogo**

Esta funcionalidade permite selecionar objetos que estejam associados ao cenário de jogo selecionado. Deve ser possível fazer a seleção através de um painel com a hierarquia dos objetos tal como pelo ecrã de visualização do cenário.

- **Clonar objetos de jogo**

Esta funcionalidade permite clonar objetos que estejam associados ao cenário de jogo selecionado. Deve ser possível fazer o clone de objetos através de um painel com a hierarquia dos objetos tal como pelo ecrã de visualização do cenário.

- **Editar as propriedades dos objetos de jogo**

O Editor deverá fornecer uma ferramenta que permita editar as propriedades dos objetos associados ao cenário selecionado.

- **Ferramentas de transformação de objetos de jogo**

O Editor deverá fornecer uma ferramenta que permita efetuar operações de transformação (deslocação, rotação e dimensão) sobre os objetos de jogo no cenário selecionado.

- **Movimentar câmara de jogo**

Esta funcionalidade permite deslocar e aplicar zoom à câmara de jogo. Idealmente deverá ser possível utilizar tanto o teclado como o rato.

- **Executar jogo**

Esta funcionalidade permite executar o jogo no cenário selecionado com o objetivo de visualizar o aspeto do jogo num determinado instante.

- **Customização do ambiente de desenvolvimento**

O Editor deverá fornecer um mecanismo que permita modificar o ambiente de desenvolvimento do mesmo. Nesse sentido, deverá ser possível organizar os vários painéis disponíveis com o objetivo de satisfazer as preferências dos seus utilizadores.

- **Desfazer / Refazer ações**

O Editor deverá fornecer um mecanismo que permite desfazer ou refazer ações efetuadas. Por ex. deverá ser possível desfazer/refazer uma ação de translação sobre um objeto acabada de executar.

- **Criar conta de utilizador**

O Editor deverá fornecer um mecanismo que permita criar uma conta de utilizador. Essa conta poderá ser utilizada posteriormente para sincronização de ficheiros na cloud, entre outros.

- **Efetuar login com conta de utilizador**

O Editor deverá fornecer um mecanismo que permita aos utilizadores fazer login com a sua conta pessoal.

- **Utilização em modo offline**

O Editor deverá fornecer um mecanismo que permita aos utilizadores utilizar o Scarlett sem uma conexão ativa com a internet. Todos os dados e preferências devem ser guardados localmente.

4.4 Requisitos não funcionais

A seguinte lista inclui os requisitos não funcionais associados ao desenvolvimento de todo o ambiente envolvente ao Scarlett Game Studio.

- **Tecnologia**

A tecnologia a utilizar deverá conter um número de limitações reduzido possibilitando a execução em diferentes sistemas com o menor esforço associado possível.

- **Usabilidade**

O sistema deverá ser eficiente e fácil de utilizar por qualquer utilizador. Para isso deverá possuir uma interface de utilizador simples e prática que permita visualizar e executar com agilidade as funcionalidades disponíveis.

- **Fiabilidade**

O sistema deverá ser seguro, as operações consideradas não seguras deverão ser executadas num meio contido para não causar problemas críticos. É importante que não sejam perdidos dados durante a utilização. Devem ser utilizados testes ao longo do desenvolvimento do projeto para assegurar a fiabilidade do mesmo.

- **Eficiência**

O sistema deverá ser eficiente no sentido de utilizar a menor quantidade de recursos disponíveis para cada tarefa que tem a executar. Um motor de jogos é responsável por gerir bastante informação em tempo de execução e a sua eficiência é muito importante.

4.5 Casos de uso

A seguinte listagem apresenta os casos de uso relacionados com a interação do editor do Scarlett Game Studio:

- Criar projeto
- Carregar projeto
- Adicionar cenário

- Carregar cenário
- Editar propriedades de um cenário de jogo
- Remover cenário
- Adicionar objeto de jogo
- Selecionar objeto(s) de jogo
- Editar propriedades dos objetos selecionados
- Remover objeto de jogo
- Importar conteúdo para o jogo
- Adicionar script
- Remover script
- Compilar scripts
- Executar jogo
- Adicionar componente a objeto(s) de jogo
- Remover componente de objeto(s) de jogo
- Editar propriedades de um componente
- Efetuar login com conta de utilizador
- Efetuar registo de uma conta de utilizador
- Efetuar alterações na conta de utilizador (nome, imagem...)
- Efetuar logout de uma conta de utilizador
- Deslocar câmara do editor
- Efetuar zoom na câmara do editor
- Desfazer ação efetuada
- Refazer ação efetuada
- Modificar associação hierárquica de um objeto de jogo
- Exportar projeto

Tabela 1 - casos de uso

Criar Projeto	
Descrição	Esta ação permite criar um projeto associado ao Scarlett Game Studio. Deverá aparecer uma janela de navegação do sistema operativo que permita ao utilizador escolher a pasta do projeto. Em adição, o utilizador pode também definir o nome do projeto.
Pré-condições	• Espaço livre no sistema operativo. • Utilização de uma pasta de destino sem conteúdo.
Resultado	• A estrutura de ficheiros e pastas de um projeto é criada. • É gerado um ficheiro com as propriedades do projeto na pasta selecionada.

Carregar Projeto	
Descrição	Esta ação permite carregar um projeto associado ao Scarlett Game Studio. Deverá aparecer uma janela de navegação do sistema operativo que permita ao utilizador escolher a pasta do projeto a carregar.
Pré-condições	• Pelo menos um projeto do Scarlett criado.
Resultado	• A estrutura de ficheiros e pastas de um projeto é carregada. • O editor apresenta o conteúdo do projeto carregado.

Adicionar cenário	
Descrição	Esta ação permite adicionar um cenário de jogo ao projeto. Deverá existir um menu de navegação no editor que apresente a hierarquia de pastas no projeto.
Pré-condições	• Um projeto carregado.
Resultado	• Um ficheiro é criado na pasta do projeto com as propriedades associadas ao cenário. • Na hierarquia de navegação do editor é apresentado o cenário criado.

Carregar cenário	
Descrição	Esta ação permite carregar um cenário de jogo associado ao projeto. Deve ser utilizado o navegador de conteúdo do editor para que seja possível selecionar e abrir o cenário desejado.
Pré-condições	• Um projeto carregado. • Um cenário criado.
Resultado	• O editor carrega todo o conteúdo associado ao cenário. • O editor apresenta todo o conteúdo associado ao cenário atualizando os componentes de edição.

Editar propriedades de um cenário de jogo	
Descrição	Esta ação permite editar as propriedades de um cenário de jogo tais como cor de fundo, nome, etc.
Pré-condições	• Um projeto carregado. • Um cenário carregado.
Resultado	• O editor atualiza os componentes associados e apresenta as alterações efetuadas.

Remover cenário	
Descrição	Esta ação permite remover um cenário de jogo associado ao projeto. Deve ser utilizado o navegador de conteúdo do editor para que seja possível selecionar e remover o cenário desejado.
Pré-condições	• Um projeto carregado. • Um cenário criado.
Resultado	• O editor atualiza os componentes associados e apresenta as alterações efetuadas.

Adicionar objeto de jogo	
Descrição	Esta ação permite adicionar um objeto de jogo associado ao cenário selecionado. Deve ser utilizado o navegador de hierarquia do cenário na execução deste processo.
Pré-condições	• Um projeto carregado. • Um cenário carregado.
Resultado	• Um objeto de jogo é criado no cenário de jogo atualizando o menu de hierarquia de objetos.

Selecionar objeto(s) de jogo	
Descrição	Esta ação permite selecionar um ou vários objetos de jogo disponíveis num cenário selecionado. A seleção pode ser feita ou pelo menu de hierarquia de objetos ou pelo painel de visualização do cenário de jogo. Deverá existir um editor de propriedades que apresente as características que podem ser modificadas pelo utilizador. Quando vários objetos são selecionados ao mesmo tempo, as propriedades comuns que tenham valores diferentes devem ser diferenciadas.
Pré-condições	• Um projeto carregado. • Um cenário carregado. • Um ou vários objetos de jogo criados.
Resultado	• Os objetos selecionados são identificados visualmente nos componentes do editor. • O editor de propriedades apresenta as características que podem ser modificadas pelo utilizador. • Quando é selecionado mais do que um objeto, as propriedades comuns com valores diferentes são diferenciadas visualmente.

Editar propriedades de um objeto de jogo	
Descrição	Esta ação permite editar as propriedades de um objeto de jogo associado ao cenário selecionado. Deve ser utilizado um editor de propriedades com o objetivo de apresentar ferramentas distintas a cada uma das

	características do objeto (por ex. uma propriedade de cor deverá possuir um editor que permita selecionar e modificar de forma visual o valor da cor).
Pré-condições	• Um projeto carregado. • Um cenário carregado. • Um objeto selecionado.
Resultado	• O valor da propriedade editada é modificado. • Propriedades de carácter visual (por ex. rotação, posição, cor...) são automaticamente atualizadas no painel de visualização do cenário.

Remover objeto de jogo	
Descrição	Esta ação permite remover objetos de jogo no cenário ativo. Para a execução desta ação deve ser utilizado o menu de hierarquia de objetos.
Pré-condições	• Um projeto carregado. • Um cenário carregado. • Pelo menos um objeto criado.
Resultado	• O objeto é removido da hierarquia de objetos. • Caso seja um dos objetos selecionados, os componentes atualizam a sua informação tendo em consideração que o objeto foi removido.

Importar conteúdo para o jogo	
Descrição	Esta ação permite importar conteúdo (imagens, scripts...) para a pasta do jogo em desenvolvimento. Para a execução desta ação pode ser utilizado o menu de navegação de pastas do projeto.
Pré-condições	• Um projeto carregado. • Um cenário carregado. • Um ficheiro para importar.
Resultado	• O ficheiro é adicionado ao projeto. • O menu de navegação apresenta o ficheiro adicionado.

Adicionar Script	
Descrição	Esta ação permite adicionar um ficheiro de scripting ao jogo em desenvolvimento. Para a execução desta ação pode ser utilizado o menu de navegação de pastas do projeto.
Pré-condições	• Um projeto carregado.
Resultado	• O script é adicionado ao jogo. • O menu de navegação apresenta o script adicionado.

Remover script	
Descrição	Esta ação permite remover um ficheiro de scripting ao jogo em desenvolvimento. Para a execução desta ação pode ser utilizado o menu de navegação de pastas do projeto.
Pré-condições	• Um projeto carregado. • Um script criado
Resultado	• O script é removido do jogo. • Objetos de jogo com associação ao script são atualizados. • O menu de navegação atualiza a sua listagem sem incluir o ficheiro removido.

Compilar scripts	
Descrição	Esta ação permite compilar os scripts adicionados ao jogo em desenvolvimento. O motor de jogo deverá possuir uma listagem com todos os scripts associados.
Pré-condições	• Um projeto carregado. • Pelo menos um script criado
Resultado	• Os scripts são compilados. • Alterações efetuadas nas propriedades públicas dos scripts são atualizadas no editor de propriedades.

Executar jogo	
Descrição	Esta ação permite executar o cenário ativo em "modo de jogo" podendo o utilizador visualizar o resultado num formato jogável.
Pré-condições	• Um projeto carregado. • Um cenário carregado.
Resultado	• O cenário ativo é executado possibilitando a sua interação em "modo de jogo". • Enquanto o processo de execução está a decorrer, a edição não tem impacto sobre este modo.

Adicionar componente a objeto(s) de jogo	
Descrição	Após a compilação dos scripts, os mesmos podem ser adicionados a objetos de jogo em forma de componentes com propriedades editáveis.
Pré-condições	• Um projeto carregado. • Um cenário carregado. • Pelo menos um objeto de jogo criado. • Pelo menos um script criado e compilado.

Resultado	• O componente é adicionado ao objeto de jogo.

Remover componente de objeto(s) de jogo	
Descrição	Esta ação permite remover um componente associado a um objeto de jogo. Para a sua execução, deve ser utilizado o editor de propriedades de objetos.
Pré-condições	• Um projeto carregado. • Um cenário carregado. • Pelo menos um objeto de jogo criado. • Pelo menos um componente associado a um objeto de jogo. • Pelo menos um objeto selecionado.
Resultado	• O componente é removido do(s) objeto(s) de jogo. • O editor de propriedades atualiza a sua visualização com o objetivo de remover o componente da sua listagem.

Editar propriedades de um componente	
Descrição	Esta ação permite editar as propriedades públicas de um componente associado a um objeto de jogo. Para a sua execução, deve ser utilizado o editor de propriedades de objetos.
Pré-condições	• Um projeto carregado. • Um cenário carregado. • Pelo menos um objeto de jogo criado. • Pelo menos um componente associado a um objeto de jogo. • Pelo menos um objeto selecionado.
Resultado	• A propriedade do componente é alterada. • Propriedades de carácter visual (por ex. rotação, posição, cor...) são automaticamente atualizadas no painel de visualização do cenário.

Efetuar login com conta de utilizador	
Descrição	Esta ação permite aos utilizadores efetuar login com os seus dados de utilizador no editor do motor de jogo.
Pré-condições	• Editor aberto com nenhuma sessão ativa. • Ligação à internet.
Resultado	• Os dados do utilizador são apresentados no editor. • As preferências do utilizador são aplicadas.

Efetuar registo de uma conta de utilizador	
Descrição	Esta ação permite aos utilizadores efetuar um registo com o objetivo de obter uma conta de utilizador associada ao motor de jogo.
Pré-condições	• Editor aberto com nenhuma sessão ativa. • Ligação à internet.
Resultado	• Os dados do utilizador são criados remotamente na base de dados do motor de jogo.

Efetuar alterações na conta de utilizador	
Descrição	Esta ação permite aos utilizadores efetuar alterações aos seus dados pessoais (como por ex. imagem, nome).
Pré-condições	• Sessão de utilizador remota ativa. • Ligação à internet.
Resultado	• Os dados do utilizador são atualizados remotamente na base de dados do motor de jogo.

Efetuar logout de uma conta de utilizador	
Descrição	Esta ação permite aos utilizadores efetuar logout da sua conta de utilizador no motor de jogo.
Pré-condições	• Sessão de utilizador remota ativa. • Ligação à internet.
Resultado	• A conta de utilizador é fechada necessitando de um novo login para entrar na aplicação em modo online.

Deslocar câmara do editor	
Descrição	Esta ação permite aos utilizadores deslocar a câmara do editor possibilitando a navegação pelo cenário de jogo. Para a execução desta ação tanto pode ser usado o teclado ou o rato, ficando a decisão à preferência do utilizador.
Pré-condições	• Um projeto carregado. • Um cenário carregado.
Resultado	• A câmara de jogo é deslocada para a posição desejada.

Efetuar zoom na câmara do editor	
Descrição	Esta ação permite aos utilizadores efetuar zoom na câmara do editor possibilitando uma visão mais alargada ou próxima sobre o cenário de

	jogo. Para a execução desta ação tanto pode ser usado o teclado ou o rato, ficando a decisão à preferência do utilizador.
Pré-condições	• Um projeto carregado. • Um cenário carregado.
Resultado	• A câmara de jogo apresenta mais ou menos conteúdo conforme o zoom aplicado.

Desfazer ação efetuada	
Descrição	Esta ação permite aos utilizadores desfazer uma ação efetuada no editor. A operação inclui alterações em objetos e cenários. A execução desta ação pode ser feita pelo teclado ou pela interface do editor.
Pré-condições	• Um projeto carregado. • Um cenário carregado. • Pelo menos uma ação efetuada.
Resultado	• A última ação realizada é desfeita sendo que o estado anterior à ação é reposto.

Refazer ação efetuada	
Descrição	Esta ação permite aos utilizadores refazer uma ação desfeita no editor. A operação inclui alterações em objetos e cenários. A execução desta ação pode ser feita pelo teclado ou pela interface do editor.
Pré-condições	• Um projeto carregado. • Um cenário carregado. • Pelo menos uma ação desfeita.
Resultado	• A última ação realizada é refeita sendo que o estado sucessor à ação é reposto.

Modificar associação hierárquica de um objeto de jogo	
Descrição	Esta ação permite aos utilizadores mudar a hierarquia de objetos, entenda-se, alterar a posição hierárquica de um objeto relativamente a outro. A execução desta ação deve ser realizada pelo menu de hierarquia de objetos disponível no editor.
Pré-condições	• Um projeto carregado. • Um cenário carregado. • Pelo menos dois objetos criados
Resultado	• A posição hierárquica do objeto é alterada

Exportar projeto	
Descrição	Esta ação permite aos utilizadores exportar os jogos criados no Scarlett Game Studio para outras plataformas. A execução desta operação deve ser feita através de um menu de especifico de exportação no editor.
Pré-condições	• Um projeto carregado. • Um cenário criado.
Resultado	• É criada uma pasta com o conteúdo do jogo exportado e preparado para ser executado na plataforma alvo escolhida.

4.6 Tecnologias

Com o objetivo de satisfazer a necessidade de compatibilidade em diferentes plataformas com a menor redundância possível, as tecnologias escolhidas neste desenvolvimento são maioritariamente compatíveis com ambientes web.

4.6.1 Lado do cliente

Os recipientes web ("web container" em inglês) podem ser encontrados em qualquer sistema operativo relevante. Normalmente as aplicações web são executadas em navegadores de internet que possuem todas as funcionalidades necessárias para o funcionamento dos mesmos. Além disso, existem aplicações tais como o *Electron* que utilizam os motores de renderização disponíveis nos navegadores de internet existentes, mas com a vantagem de serem executados nativamente dando assim a possibilidade de estender as funcionalidades associadas. Resumidamente, permitem que aplicações web possam ser executadas nativamente com mais permissões sobre o sistema operativo em utilização.

Mais especificamente o *Electron* é um software que possui suporte a tecnologias web (*HTML*, *CSS* e *Javascript*) e utiliza o motor *Chromium* e *Node.JS*. Uma das características mais interessantes é a possibilidade de execução em diferentes plataformas incluindo Mac OS, Windows e Linux. Sendo que utiliza o *Chromium*, inclui também nas suas capacidades: atualizações automáticas, menus nativos, menus de depuração e avaliadores de performance. Desta forma torna-se evidente que a sua utilização neste desenvolvimento é uma boa decisão sobre os módulos presentes no lado do cliente.

Além do recipiente onde vai ser executado o editor do Scarlett Game Studio, existe a necessidade de escolher uma framework de desenvolvimento com o objetivo de agilizar e manter com maior aptidão o processo de criação.

Nos dias de hoje existem imensas bibliotecas e frameworks em auxilio do desenvolvimento web. Porém, não apenas por experiência pessoal, mas também por possuir uma comunidade interessante e a expectativa de atualizações futuras, a escolha de *AngularJS* no suporte ao desenvolvimento do editor sobressai sobre as demais.

Numa breve descrição, o *AngularJS* tem como principal objetivo estender as funcionalidades existentes em HTML dando a possibilidade de criar ambientes web dinâmicos. O *HTML* por natureza é uma boa escolha para a apresentação de documentos estáticos, mas não foi construído com o intuito de apresentar conteúdo dinâmico (normalmente utilizam-se bibliotecas de terceiros tais como o *jQuery* no sentido de ultrapassar esta limitação). Além disso, utiliza o padrão de desenvolvimento *MVC* (*Model-View-Controller*) que permite isolar a lógica da aplicação da apresentação da interface de utilizador (em capítulos posteriores irá ser detalhado com maior detalhe).

Por fim, vai ser utilizado WebGL para a renderização dos gráficos 2D do Scarlett Game Studio. A escolha desta *API* não foi necessariamente complicada pois é sem dúvida a que mais se destaca em ambientes web. Além do *WebGL* a *API* de renderização mais adequada seria o padrão *HTML5 Canvas* que possui um índice de performance muito abaixo da primeira opção.

Resumidamente, o *WebGL* é baseado no *OpenGL ES 2.0* e fornece uma interface de programação para gráficos 3D/2D. As suas semelhanças com *OpenGL* nativo são um grande beneficio pois poderá facilitar a integração futura com um motor de renderização de mais baixo nível. Existem algumas bibliotecas de auxilio ao desenvolvimento com *WebGL* (alguns utilizadores consideram demasiado complexo pois requer uma compreensão ampla para efetuar operações simples) embora neste desenvolvimento não serão utilizadas com o objetivo de possuir um maior controlo sobre o mesmo.

Em conclusão, as tecnologias principais associadas ao desenvolvimento no lado do cliente serão as seguintes:

- *Electron*
- *Node.JS*
- *AngularJS*
- *Javascript*
- *HTML*
- *CSS*
- *WebGL*

Em segundo plano, vão ser utilizadas outras tecnologias de suporte que vão ser apresentadas em maior detalhe nos seguintes capítulos e incluem: *Ruby*, *SASS*, *Grunt*. Além disso, serão incluídas várias dependências (especialmente no editor) de bibliotecas de terceiros com o objetivo de agilizar o desenvolvimento. A organização das mesmas será feita pelo gestor de pacotes *NPM* que permite instalar, atualizar e especificar as dependências envolvidas em cada um dos módulos. A descrição das dependências e versões utilizadas está presente num ficheiro (tipicamente chamado *package.json*) que faz parte dos ficheiros da aplicação.

4.6.2 Lado do servidor

O serviço remoto será desenvolvido em *PHP* fornecendo uma API SOAP que pode ser invocada pelo editor da aplicação. A escolha de *PHP* recai principalmente por experiência pessoal,

capacidade de acesso as funcionalidades do sistema operativo e pela facilidade de incorporação em servidores de licenciamento básico.

O módulo de persistência (responsável por armazenar toda a informação associada ao motor de jogo incluindo os dados dos utilizadores) será em *MySQL*.

4.7 Controlo de versões

Com o objetivo principal de armazenar as alterações realizadas no projeto ao longo do seu desenvolvimento, será utilizado o *Git*, um sistema de controlo de versões. Em particular será utilizado o serviço *Bitbucket* pois permite a criação de projetos privados e a definição de tarefas e problemas associados ao desenvolvimento.

A vantagem de utilizar um sistema de versões relativamente à básica tarefa de copiar manualmente os arquivos por fases, é não ser suscetível a erros e permitir uma integração com vários elementos de equipa sem grandes preocupações. Com a utilização de uma aplicação de visualização sobre as alterações é também possível identificar com facilidade as tarefas realizadas pelos diferentes elementos (e identificar alterações indesejadas no comportamento das aplicações quando os há).

5 Framework

A framework é o esqueleto do Scarlett Game Studio, além de possuir as estruturas do modelo de negócio da aplicação, é responsável pela renderização e atualização do conteúdo de jogo. É importante relembrar que este módulo pode ser utilizado individualmente sem o auxilio do editor caso o utilizador assim prefira, sendo implicada a necessidade de uma API em Javascript que exponha as funcionalidades implementadas.

5.1 Preparação

Este módulo está dividido em múltiplos ficheiros Javascript que contêm o modelo de dados e funcionalidades relacionadas. Com o objetivo de integrar com diferentes contextos, existe a necessidade de concatenar todos os ficheiros existentes em apenas um. Para esse efeito utilizou-se o executador de tarefas *Javascript Grunt* que executa um script desenvolvido especificamente para automatizar este processo sempre que se efetuam alterações:

```
module.exports = function(grunt) {

    var sortDependencies = require("sort-dependencies");
    var copyToDirectory = "<path_to_export>";

    // Project configuration.
    grunt.initConfig({
        pkg: grunt.file.readJSON('package.json'),
        uglify: {
            options: {
                banner: '/*! <%= pkg.name %> <%= grunt.template.today("yyyy-mm-dd") %> */\n'
            },
            build: {
                src: 'build/<%= pkg.name %>.js',
                dest: 'build/<%= pkg.name %>.min.js'
            }
        },
        concat: {
            options: {
                separator: ';'
            },
            dist: {
                src: [
                    'node_modules/matter-js/build/matter.js',
                    sortDependencies.sortFiles("src/**/*.js")],
                dest:
                    'build/<%= pkg.name %>.js'
            }
        },
        copy: {
            main: {
                src: 'build/<%= pkg.name %>.js',
                dest:  copyToDirectory + '<%= pkg.name %>.js'
            }
        },
        jshint: {
            beforeconcat: ['src/**/*.js']
        },
        watch: {
            scripts: {
                files: ['src/**/*.js'],
                tasks: ['dev-concat', 'copy-to'],
                options: {
                    interrupt: true
                }
            }
        }
    });
```

```
    // Load the plugins here
    grunt.loadNpmTasks('grunt-contrib-jshint');
    grunt.loadNpmTasks('grunt-contrib-uglify');
    grunt.loadNpmTasks('grunt-contrib-concat');
    grunt.loadNpmTasks('grunt-contrib-watch');
    grunt.loadNpmTasks('grunt-contrib-copy');

    // Default task(s).
    grunt.registerTask('default', ['jshint', 'concat', 'uglify']);
    grunt.registerTask('watcher', ['watch']);
    grunt.registerTask('dist', ['uglify']);
    grunt.registerTask('dev', ['concat', 'watch']);
    grunt.registerTask('dev-concat', ['concat']);
    grunt.registerTask('copy-to', ['copy']);

};
```

Ao executar a tarefa definida por defeito, todo o código dentro da pasta '/src/' é validado (jshint) e em caso de sucesso concatenado e "minificado" sendo que o ficheiro resultante é a framework em formato comprimido.

Também foram criadas tarefas para copiar automaticamente o resultado deste processo para uma pasta de destino (normalmente a do editor).

5.2 Fluxo de jogo

Como previamente mencionado, uma das principais funções da framework é a capacidade de executar um videojogo sendo responsável pelo seu fluxo durante o processo.

Tipicamente o fluxo de um videojogo é comum a todos os motores de jogos. Inicialmente são carregados os conteúdos do jogo (imagens, scripts, sons...) e é realizada a inicialização do primeiro cenário. Após o arranque, o cenário é atualizado e apresentado continuamente até que aconteça uma situação de paragem (por ex. o jogador termina a aplicação ou o cenário é alterado). Por fim, todo o conteúdo é libertado da memória e o fluxo é terminado ou reiniciado.

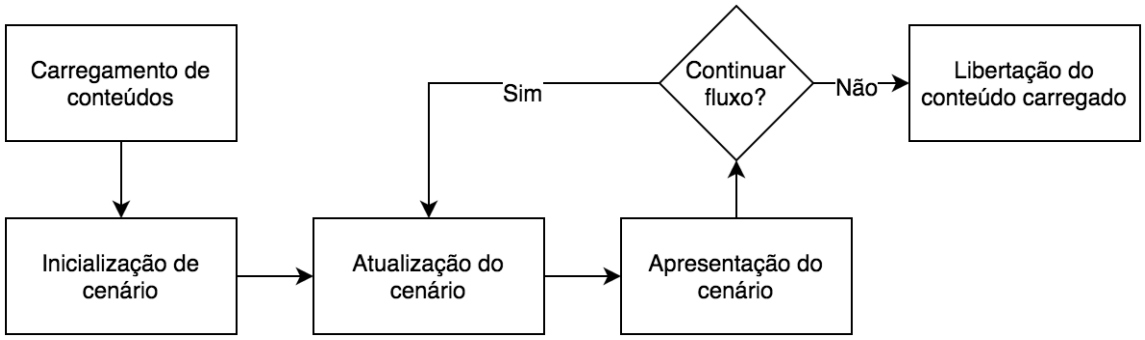

Figura 10 - fluxo de jogo no Scarlett Game Studio

As fases de atualização e apresentação do cenário ocorrem múltiplas vezes num curto espaço de tempo, normalmente são contabilizados o número de ciclos para apresentar ao utilizador a quantidade de quadros apresentados num segundo (conhecido por FPS – frames per second).

Na fase de atualização, toda a lógica associada aos objetos de jogo é executada. Durante este processo, acontecem varias operações, por exemplo, o sistema de física faz a sua simulação e os componentes executam as ações programadas.

Na fase de apresentação, a informação sobre os objetos (vértices e texturas) de jogo é enviada para a placa gráfica que trata de renderizar o pedido para o ecrã de visualização.

Em Javascript, as fases de atualizar e apresentar devem ser apenas executadas quando existe necessidade para tal. Por exemplo, caso o utilizador esteja com o browser minimizado ou numa outra aba pode não ser preciso a sua realização. Afortunadamente, podemos requisitar um novo quadro de animação ao recipiente web que estamos a usar que internamente resolve essas condições:

```
requestAnimationFrame(callback);
```

A função recebe como parâmetro uma outra função que irá ser executada na próxima chamada de apresentação. Nesta última, é onde colocamos todo o fluxo de jogo detalhado no inicio deste subcapítulo. Em adição, na função de animação é enviado por parâmetro o tempo em que (em milissegundos) foi registado o pedido.

No final da execução da nossa função de apresentação, realizamos novamente o pedido de animação com o objetivo de executar continuadamente o processo:

```
Game.prototype._onAnimationFrame = function (timestamp) {

   // atualizar cenário:
   this._gameScene.update(timestamp);

   // apresentar cenário:
   this._gameScene.draw(timestamp);

   // requisitar quadro de animação:
   requestAnimationFrame(this._onAnimationFrame.bind(this));
};

Game.prototype._init = function() {
   // chamar a primeira vez com o timestamp a 0:
   this._onAnimationFrame(0);
};
```

O código em cima apresenta um exemplo muito elementar do processo efetuado num quadro de apresentação. Num contexto real seria necessário adicionar algumas condições de paragem e de validação (como por ex. se é preciso mudar de cenário). Como vamos ver mais à frente neste capitulo, o Scarlett Game Studio possui várias fases de atualização e de apresentação que também não estão contempladas no exemplo.

O tempo de execução tem um papel muito importante na fase de atualização do jogo. Como é obvio, os dispositivos têm capacidades de processamento diferentes, ou seja, uns vão executar mais quadros por segundo do que outros. Nesse sentido é importante que os objetos de jogo possuam o mesmo comportamento apesar das diferenças. Não é desejado que um objeto de jogo se desloque mais rápido num dispositivo apenas porque tem uma maior capacidade de processamento. Tomemos em consideração o seguinte exemplo hipotético:

Dispositivo A	**Dispositivo B**
• CPU “3000 Z Turbo” • 300 quadros por segundo	• CPU “Eco-Basix” • 24 quadros por segundo

E a seguinte execução num cenário de jogo:

```
GameScene.prototype.update = function() {
   funnyFace.position.x += 10;
};
```

Sem nenhum tratamento especial, o que iria acontecer à posição do objeto no final de 5 segundos seria o seguinte:

Dispositivo A	**Dispositivo B**
Posição X: (300*5*10) = 15.000	Posição X: (24*5*10) = 1.200

De forma a obter o mesmo comportamento independentemente do dispositivo onde está a ser executado, é efetuada uma relação com o tempo decorrido. Ao valor desta unidade chamamos de 'delta' que não é nada mais nada menos do que o tempo decorrido desde a ultima execução. Dessa forma quando multiplicamos o valor de deslocamento por 'delta' obtemos um valor normalizado adequado à capacidade de processamento do dispositivo.

```
GameScene.prototype.update = function(delta) {
   funnyFace.position.x += (10 * delta);
};
```

O valor 'delta' não é fornecido automaticamente, na função de animação apenas recebemos por parâmetro uma unidade de tempo que corresponde à data em que foi efetuado o pedido da sua execução. Para a sua aquisição guardamos sempre o valor da ultima unidade de tempo fornecida e subtraímos da mais atual:

```
Game.prototype._onAnimationFrame = function (timestamp) {
    // primeira execução?
    if (this._totalElapsedTime === null) {
        this._totalElapsedTime = timestamp;
    }

    // calcular valor de delta:
    var delta = timestamp - this._totalElapsedTime;
    this._totalElapsedTime = timestamp;

    // ...
}
```

5.3 WebGL

O *WebGL* é a biblioteca utilizada para a renderização dos gráficos 2D do Scarlett Game Studio. Esta biblioteca fornece uma API baseada no *OpenGL ES 2.0* sem que seja necessária a utilização de Plug-ins. Os programas em *WebGL* consistem em código escrito em Javascript com a adição de *Shaders* que são executados na unidade de processamento dos dispositivos (Parisi, 2012).

Neste subcapítulo vão ser apresentadas as noções básicas associadas a esta biblioteca juntamente com alguns detalhes da sua utilização no Scarlett Game Studio. É importante saber que existem muitas semelhanças com *OpenGL* (salvo a preparação do seu contexto) e grande parte do que é apresentado funciona de forma semelhante em ambas as bibliotecas.

5.3.1 Preparação do contexto WebGL

O primeiro aspeto a considerar quando se utiliza WebGL é o recipiente onde o resultado do seu processamento vai ser apresentado. Esta biblioteca possui mecanismos de integração com elementos HTML e neste caso especifico utiliza-se o elemento "*canvas*" que suporta o contexto WebGL. Ao estabelecer a ligação da biblioteca com este elemento, estamos a definir que toda a apresentação do conteúdo de jogo irá ser mostrada nesse bloco.

```
<body onload="initWebGL()">
<canvas id="glcanvas" width="100%" height="100%”>
    O seu navegador web não suporta WebGL
</canvas>
</body>
```

Pode ser utilizado CSS (ou atribuição em linha diretamente no código HTML) para estilizar o elemento '*canvas*' selecionado podendo aplicar algumas propriedades como posição e tamanho.

Na função de inicialização '*initWebGL()*', que é chamada após o carregamento da página web, inicializamos o contexto *WebGL* efetuando assim a primeira fase necessária para o uso desta biblioteca:

```
function initWebGL() {
   // Em primeira instancia vamos buscar o elemento canvas
   var canvas = document.getElementById("glcanvas");

   // Alguns navegadores possuem identificadores diferentes para referenciar o WebGL,
dessa
   // forma utilizamos vários com o objetivo de atingir o maior número possível de casos.
   var gl = canvas.getContext("webgl") || canvas.getContext("experimental-webgl");

   // WebGL inicializado com sucesso?
   if (!gl) {
      return;
   }

   // Definição da cor de fundo:
   gl.clearColor(0.0, 0.0, 0.0, 1.0);

   // Limpa o conteúdo já renderizado no canvas (caso exista) aplicando a cor de fundo
   // definida em cima:
   gl.clear(gl.COLOR_BUFFER_BIT);
}
```

A definição da resolução de visualização (em pixéis) pode ser alterada conforme o necessário usando a função 'viewport' do contexto *WebGL*:

```
gl.viewport(0, 0, largura, altura);
```

Os valores definidos para altura e largura não precisam de ser os mesmos do elemento recipiente. Conforme a diferença existente é efetuado um redimensionamento virtual ao tamanho real do elemento 'canvas' da página.

Com a execução do código em cima apresentado, deverá ser possível visualizar um retângulo preto na página do navegador.

5.3.2 Renderizar conteúdo de jogo

O *WebGL* não possui uma pipeline fixa no que toca à apresentação de conteúdos de jogo, porém oferece uma pipeline programável (Parisi, 2012) que apesar de ser mais complicada de utilizar e compreender é mais dinâmica e preponderante.

Possuindo uma pipeline programável, significa que o programador assume a responsabilidade de juntar e associar toda a informação relevante ao cenário que pretende apresentar. Por informação, entenda-se, vértices, texturas, etc. Um dos elementos mais importantes que fazem parte desta pipeline são os *Shaders*, que não são nada mais do que pequenas aplicações executadas inteiramente nas placas gráficas dos dispositivos que executam operações sobre os vértices fornecidos.

Existem dois tipos de *Shaders*:

- *Vertex Shaders (vs)*
- *Fragment Shaders (fs)*

O *Vertex Shader* recebe todos os vértices facultados pelo programador na pipeline e tem como única função determinar a sua posição final no ecrã (Parisi, 2012). Para essa finalidade é utilizado o vetor de quatro dimensões (x, y, z, w) *gl_Position* do *Shader*, que contém a posição 3D onde vai ser apresentado o vértice. Em jogos 2D como é o caso, tipicamente apenas se utiliza os primeiros dois elementos X e Y do vetor.

Um *Fragment Shader* tal como o *Vertex Shader* apenas tem uma função, mas neste caso em vez de determinar a posição dos vértices , determina a cor a aplicar (Parisi, 2012). Para essa finalidade é utilizado o vetor de quatro dimensões (neste caso para as cores vermelho, verde, azul e a respetiva opacidade) *gl_FragColor*.

Outro aspeto importante a considerar são os tipos de variáveis utilizados nos *Shaders*. Existem três tipos diferentes com os seguintes nomes:

- *Uniform*
- *Attribute*
- *Varying*

O tipo *Uniform*, é utilizado com o objetivo de capacitar ambos os *Shaders* (*vs* e *fs*) sendo que os seus valores permanecem constantes durante todo o processamento de uma renderização. Deve ser utilizado sempre que não haja necessidade de variar o valor, como por exemplo, a posição de um objeto estático num cenário.

As variáveis do tipo *Attribute* são valores aplicados individualmente a cada vértice fornecido. Note-se que este apenas pode ser utilizado em *Vertex Shaders*.

Depois de ler a afirmação anterior, surge certamente a questão sobre como passar valores individuais a cada um dos vértices ao *Fragment Shader*. Para resolver esse problema são utilizadas as variáveis do tipo *Varying* que permitem a partilha de valores do *Vertex Shader* para um *Fragment Shader*.

Os seguintes dois exemplos apresentam um *Vertex* e *Fragment Shader* respetivamente:

```
// vertex shader
attribute vec2 position;

uniform mat4 uMatrix;
uniform mat4 uTransformMatrix;

void main() {
   gl_Position = uMatrix * uTransformMatrix * vec4(position, 0.0, 1.0);
}
```

```
// fragment shader
void main() {
   gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
}
```

É de salientar que no Vertex Shader estão a ser utilizadas matrizes no cálculo do *gl_Position*. A utilização destas estruturas é muito pertinente quando se prende calcular a posição que o pixel vai ter no ecrã. Na verdade, são utilizadas com o objetivo de representar transformações (posição, dimensão, rotação) aos objetos que tencionamos apresentar. Como já foi explicado anteriormente neste documento, a escolha de matrizes neste tipo de operações deve-se principalmente ao facto de serem eficientemente calculadas nos dispositivos que as executam e também por conseguirem representar várias transformações na mesma estrutura. É também importante saber que algumas unidades de processamento gráfico são otimizadas para operações com vetores o que torna a utilização de matrizes ainda mais eficiente.

Inicialmente pode parecer curioso existirem duas matrizes aplicadas no cálculo. Na realidade uma representa a transformação da câmara no cenário (*uMatrix*) e a outra (*uTransformMatrix*) a transformação do objeto que estamos a apresentar.

Por fim, existe a necessidade de saber como integrar e executar estes programas no código *Javascript* da framework. Como foi descrito anteriormente, os *Shaders* são programas que são executadas nas unidades de processamento gráfico. Resumidamente, para os executar, começamos por criar uma nova instancia de um programa ao qual associamos os *Shaders* que desejamos utilizar. Depois de inicializado, sempre que quisermos utilizar o programa, vinculamos a sua instancia ao contexto *WebGL*. No anexo 1 está presente uma classe de auxilio criada especificamente para tratar destes assuntos associados ao WebGL onde poderá encontrar exemplos da sua inicialização.

No Scarlett Game Studio existem dois tipos de *Shaders* diferentes, um para apresentação de objetos gráficos primitivos (linhas, círculos, retângulos) ao qual denominamos de *PrimitiveShader* e outro para a apresentação de objetos com textura ao qual denominamos de *TextureShader*. A especial diferença entre os dois é que o *TextureShader* recebe como variável a informação sobre a textura que tencionamos renderizar enquanto que o *PrimitiveShader* apenas recebe cores sólidas. Ambos podem ser consultados no anexo 2.

A Figura 11 ilustra os pedidos executados à API do *WebGL* durante a realização de um processo individual de apresentação de um objeto. Considera-se que os *Shaders* já estão inicializados.

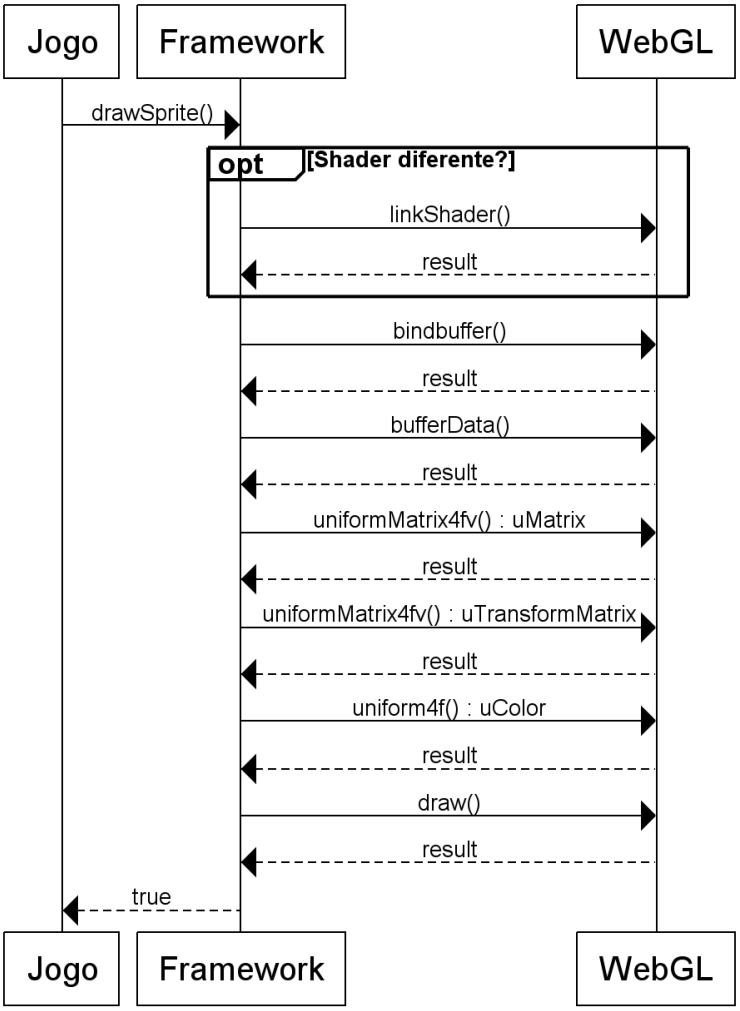

Figura 11 - fluxo de apresentação individual em WebGL

Note-se que o *Shader* a utilizar apenas é vinculado caso o ultimo usado seja diferente, não sendo necessária a operação quando contrário. Esta validação deve ser efetuada, por razões de modularidade, com auxilio de um gestor estático de Shaders. No Scarlett Game Studio foi criada uma classe chamada *ShaderManager* que possui a noção sobre qual o ultimo *Shader* vinculado podendo ser utilizada para a sua validação.

5.3.3 Spritebatch

Sem dúvida que um dos aspetos mais importantes de um motor de jogo é a sua performance. No ponto anterior explicou-se que quando desejamos apresentar conteúdo num ecrã necessitamos de enviar essa informação para a unidade de processamento. O que não foi mencionado é que esse passo deve ser efetuado o mínimo de vezes possível. É preferível, por ser mais rápido, enviar o máximo de informação possível de uma só vez do que parcialmente. Nesse sentido, foi criado um *Spritebatch* que na sua essência tem como principal função armazenar toda a informação que vamos pedindo à framework para apresentar, para que no final de cada ciclo de jogo (ou caso o buffer atribuído seja totalmente preenchido) seja descarregada toda a informação de uma vez só

para a unidade de processamento gráfico. Desta forma minimizamos o número de pedidos que é realizado melhorando assim a rapidez de processamento gráfico.

Observando a Figura 11, conseguimos perceber que sempre que precisarmos de desenhar um objeto vão ser executadas as funções de apresentação do WebGL. Ou seja, caso quisermos mostrar 3 objetos de seguida, todo o processo apresentado vai ser executado três vezes.

Em comparação, pode-se verificar na Figura 12 o comportamento agrupado que foi descrito contextualizado com o *Spritebatch* implementado no Scarlett Game Studio.

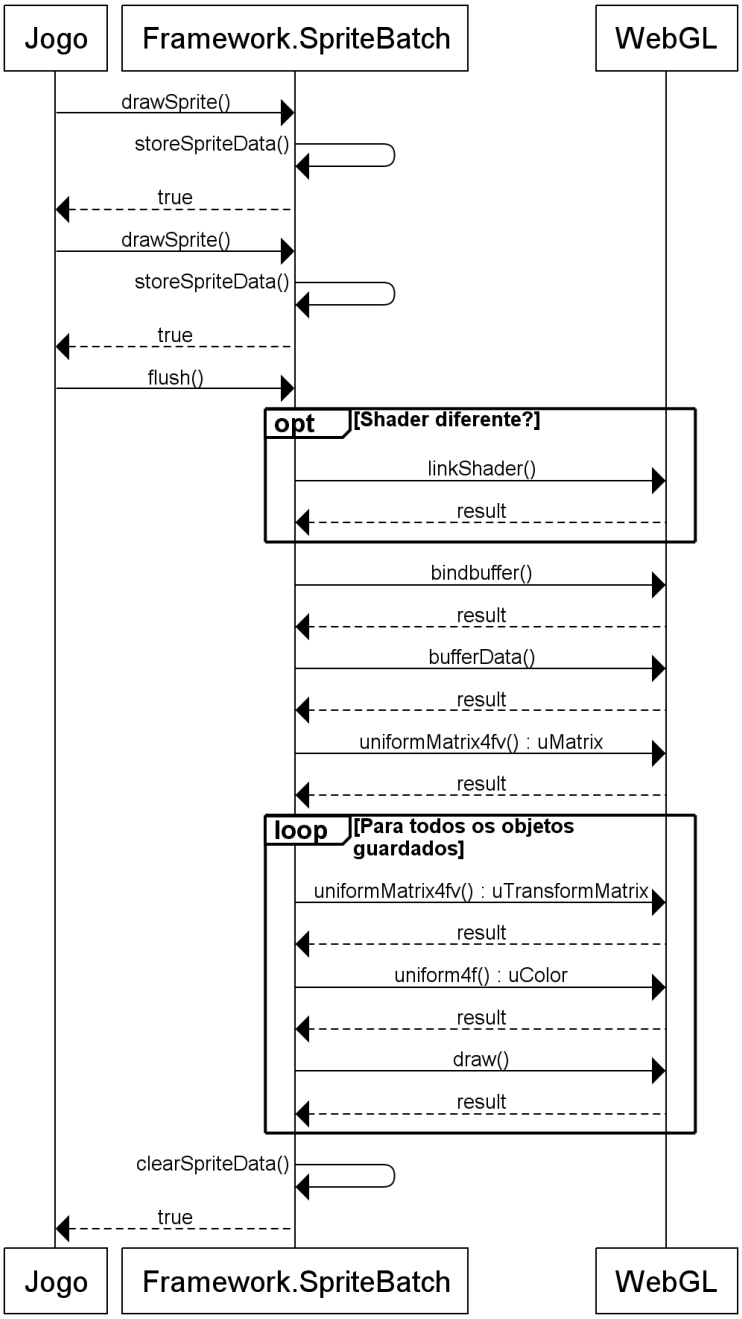

Figura 12 - fluxo de apresentação agrupada em WebGL

5.4 Arquitetura geral

A Figura 13 apresenta as classes envolvidas na framework do Scarlett Game Studio. Nos seguintes subcapítulos vão ser apresentadas em detalhe algumas das funcionalidades mais relevantes associadas à arquitetura deste módulo.

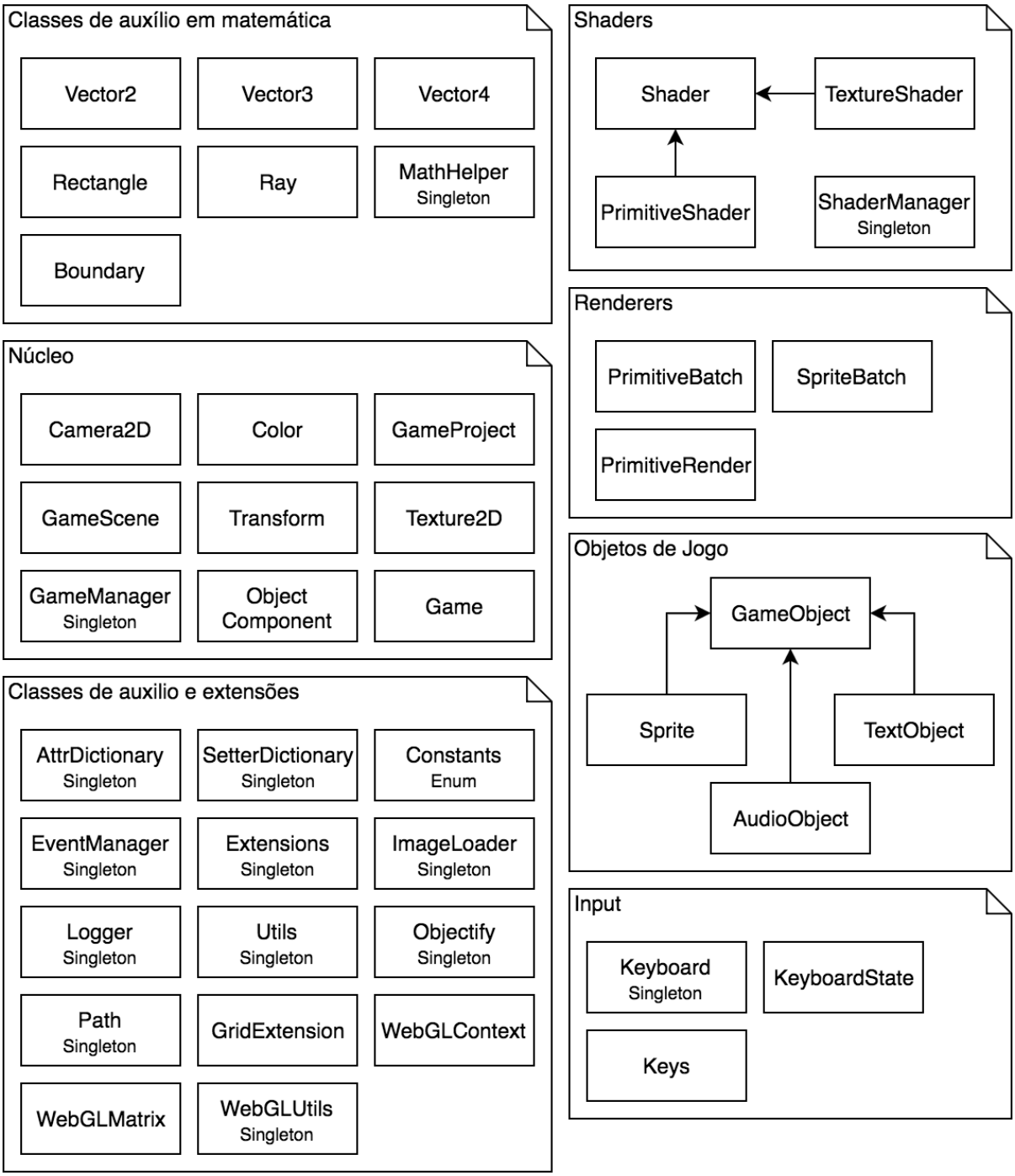

Figura 13 - diagrama de classes da framework

5.5 Objetos de Jogo

Os objetos de jogo de um motor de jogo podem ser descritos como estruturas de informação que contêm características especificas consoante o objetivo a empenhar no fluxo de processamento e apresentação. Tipicamente, e como é o caso, os objetos podem ser agrupados hierarquicamente podendo um ser "pai" de uma quantidade variável de instancias e assim sucessivamente.

Outro aspeto interessante a conhecer é o fluxo de um objeto de jogo num cenário em execução. Tal como o cenário, o objeto possui varias fases de processamento associadas. Por exemplo, quando o cenário é iniciado, todos os objetos pertencentes também o são, ou em outro caso, quando o cenário é atualizado ou apresentado.

No Scarlett Game Studio existem vários tipos de objetos disponíveis com a seguinte relação hierárquica:

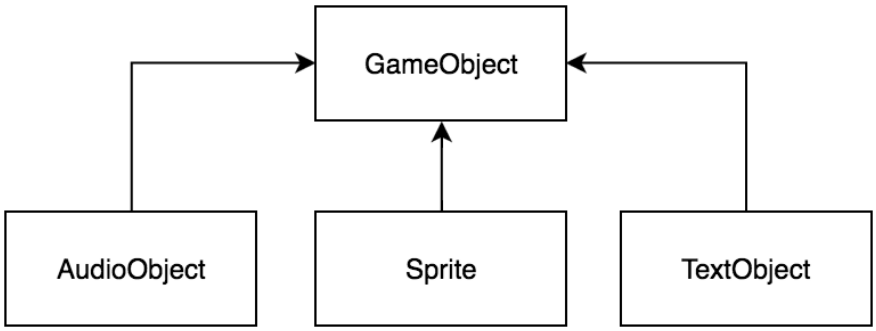

Figura 14 - hierarquia de objetos

- **GameObject** – Não possui nenhuma característica em especial sendo que serve de base para todas os outros tipos de objetos existentes. É nesta classe que podem ser encontradas definições genéricas tais como o nome, identificação de parentesco e transformação da instância do objeto.

- **AudioObject –** Permite a reprodução de áudio (por exemplo .mp3 ou .ogg) num cenário 2D com a particularidade de transmitir diferentes valores associados ao volume dependendo da posição da câmara de visualização ativa (quanto mais longe menor será o volume).

- **Sprite –** Possivelmente o tipo de objeto mais utilizado, permite apresentar uma textura no ecrã com algumas propriedades particulares. Por exemplo, pode-se definir a cor, opacidade, etc. É empregado em vários casos como representação de personagens, menus e até mesmo interfaces simples de utilizador.

- **TextObject –** Permite a apresentação de texto no ecrã podendo definir certos parâmetros tais como tamanho e tipo de letra.

5.6 Dispositivos de entrada

Com o objetivo de que os utilizadores da aplicação consigam utilizar dispositivos de entrada no motor de jogo (por ex. o teclado) foi criada uma interface genérica que permite mapear o estado do conjunto de teclas ou botões num determinado momento. Com o conhecimento sobre o estado é possível efetuar validações como por exemplo, saber se uma determinada tecla está a ser pressionada ou não.

A atribuição do estado não é efetuada pela framework sendo que fica sobre a responsabilidade do recipiente que a executa. No caso do editor, quando uma tecla é pressionada, essa informação é partilhada com a framework para que na validação de estado seguinte possa ser verificada a condição presente do dispositivo de entrada.

A Figura 15 ilustra o fluxo existente que é efetuado quando uma tecla é pressionada no editor e de seguida largada.

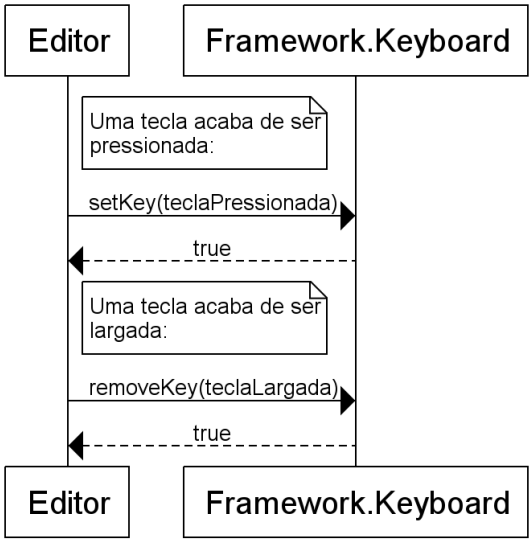

Figura 15 - fluxo de atribuição de estado num dispositivo de entrada

A Figura 16 ilustra o fluxo existente que é efetuado quando o estado de uma tecla é verificado no contexto de um jogo.

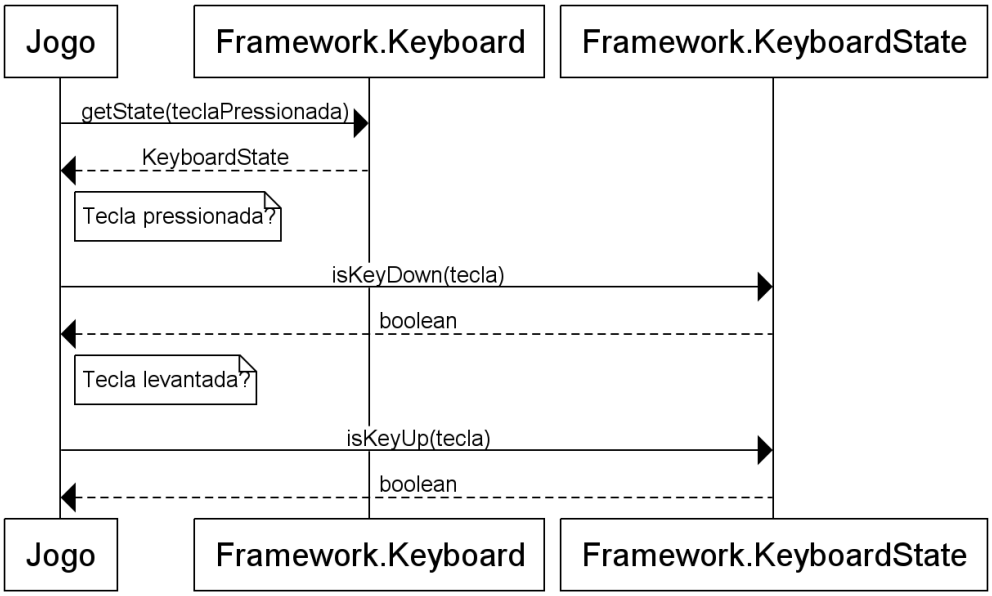

Figura 16 - fluxo de validação de estado num dispositivo de entrada

A lógica aplicada nos exemplos demonstrados em cima aplica-se de forma similar a outros dispositivos de entrada como por exemplo um controlador (em inglês é comum chamar de *Joystick*).

5.7 Componentes

A utilização de componentes em motores de jogos é uma prática bastante comum. Na sua essência, componentes são procedimentos programáveis que podem ser atribuídos a objetos de jogo com o objetivo de tornar o seu comportamento dinâmico. No Scarlett Game Studio, a implementação deste mecanismo está associada à construção de scripts que sucintamente descritos, são pedaços de código que os utilizadores podem escrever com o objetivo de definir comportamentos específicos em componentes criados.

O fluxo de um componente é bastante similar ao de um objeto de jogo. Quando um objeto é inicializado, todos os componentes atribuídos também o são. O mesmo acontece quando existe uma atualização ou apresentação. Durante estas diferentes fases, é de a responsabilidade do componente efetuar operações sobre o objeto de jogo a que está atribuído ou a qualquer outro do cenário.

A criação de um componente pode ser efetuada sem o auxilio do editor, foi criada uma classe chamada "*ObjectComponent*" que serve de base na construção dos scripts. A única propriedade existente na classe mãe é o objeto de jogo a qual o componente é atribuído. Após a criação do componente, o mesmo pode ser adicionado a qualquer objeto de jogo utilizando a função "*addObjectComponent*()" existente nos mesmos. De seguida o motor de jogo fica responsável por invocar todas as fases de processamento associadas.

5.8 Sistema de Física

O sistema que permite detetar colisões no Scarlett Game Studio está associado a um sistema de física com características complexas. Particularmente, nesta implementação existe a dependência do motor de física *Matter-JS,* uma biblioteca cujo objetivo é fornecer capacidades orientadas a este âmbito.

O sistema de física pode ser compreendido como um componente individual que tal como os objetos é atualizado em cada quadro de processamento. Para cada cenário de jogo existe um mundo virtual com configurações típicas de um universo material (por ex. gravidade). Depois de instanciado, podem ser adicionadas entidades ao mundo que interagem fisicamente entre si (colisões, ligações...). É importante notar que as entidades do mundo "físico" podem não conter a totalidade de objetos existentes num cenário. Inicialmente os objetos de jogo não possuem características físicas nem estão associados a nenhuma entidade. Esta possibilidade é apenas verdadeira quando o utilizador associa um componente de física a um objeto.

Quando um componente de física é associado a um objeto de jogo, as suas transformações de rotação, dimensão e posição passam a ser controladas pelo motor de física. Mais concretamente, quando um objeto está associado a uma entidade, essas propriedades estão sincronizadas.

As entidades podem conter diversas formas, tipicamente as mais utilizadas são os retângulos e círculos embora seja possível criar polígonos complexos a partir de um conjunto de pontos.

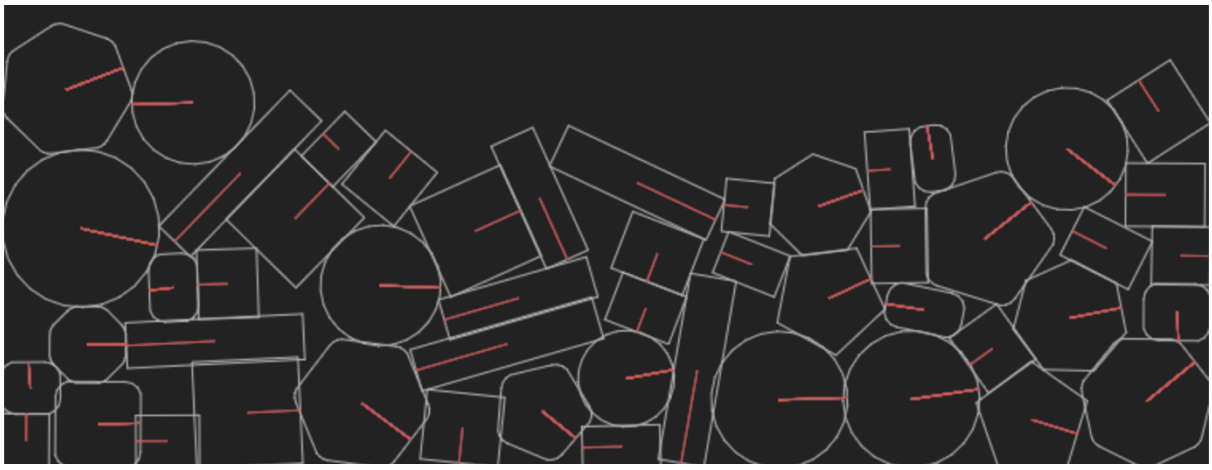

Figura 17 - exemplo de diferentes formatos em entidades existentes no Scarlett

5.9 Deteção de colisões com sobreposição

Além da utilização do sistema de física existente no Scarlett Game Studio, também é possível testar colisões com os limites geométricos dos objetos usando uma implementação do teorema de separação de eixos.

Existem várias técnicas que permitem verificar se uma figura geométrica sobrepõe outra. O procedimento mais simples passa por criar um retângulo de colisão (ou caixa de colisão) em torno de uma área desejada (como por exemplo as dimensões brutas de um objeto de jogo) e verificar se os seus limites entram em colisão com outra área fornecida.

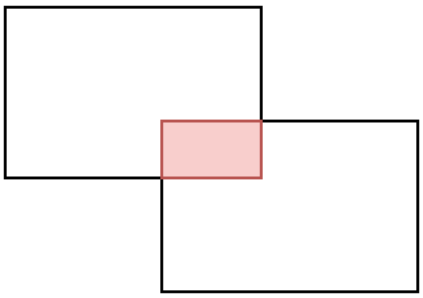

Figura 18 - exemplo de sobreposição de retângulos

Um problema associado a este procedimento é que vamos obter resultados inesperados quando o objeto de jogo que desejamos verificar possui, por exemplo, uma transformação de rotação aplicada. Em resultado, a caixa de colisão vai exceder os limites supostos pois é calculada com o objetivo de incluir simplesmente as dimensões do objeto não possuindo a sua rotação:

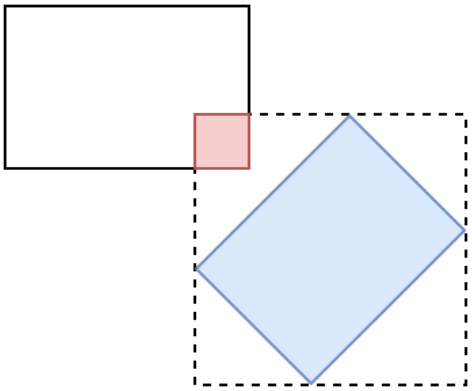

Figura 19 - exemplo de deteção de colisão indesejada (objeto de jogo representado a azul)

Uma das técnicas disponíveis para resolver este problema é a aplicação do teorema de separação de eixos, implementado em contexto prático no Scarlett Game Studio. Este teorema afirma que, para um par de polígonos convexos que não se encontrem em colisão, existe um eixo perpendicular a uma aresta de um dos polígonos que não tem qualquer sobreposição entre os vértices projetados dos dois polígonos (Huynh, 2009). Esta solução funciona em qualquer tipo de colisão, incluindo casos em que o alvo a detetar se encontra transformado:

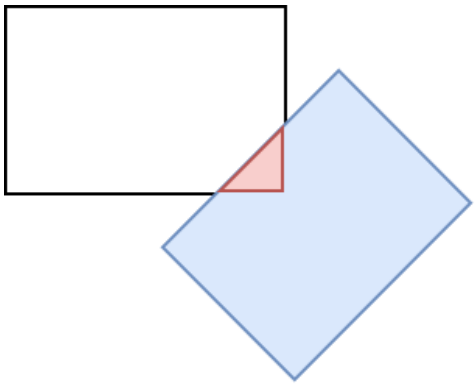

Figura 20 - exemplo de deteção de colisão desejada (com utilização do teorema de separação de eixos)

O seguinte bloco de código representa a implementação efetuada no Scarlett Game Studio com base no teorema descrito:

```
Boundary.overlap = function (boundaryA, boundaryB) {
    var normA = boundaryA.getNormals();
    var normB = boundaryB.getNormals();

    function getMinMax(boundary, norm) {
        var probeA = boundary.topRight.dot(norm);
        var probeB = boundary.bottomRight.dot(norm);
        var probeC = boundary.bottomLeft.dot(norm);
        var probeD = boundary.topLeft.dot(norm);

        return {
            max: Math.max(probeA, probeB, probeC, probeD),
            min: Math.min(probeA, probeB, probeC, probeD)
        }
    }

    var p1, p2, normNode, norm;
    for (var i = 0; i < 4; i++) {
        normNode = i >= 2 ? normB : normA;
        norm = i % 2 == 0 ? normNode.bottom : normNode.right;
        p1 = getMinMax(boundaryA, norm);
        p2 = getMinMax(boundaryB, norm);

        if (p1.max < p2.min || p2.max < p1.min) {
            return false;
        }
    }

    return true;
};
```

6 Serviço remoto

O serviço remoto do Scarlett Game Studio é distribuído por dois módulos principais, uma base de dados que contém toda a informação sobre os dados associados ao motor de jogo e um serviço web que permite ao editor invocar pedidos remotamente sobre informação na base de dados e sistema de ficheiros num servidor de conteúdos.

6.1 Base de dados

A base de dados foi implementada em *MySQL* e guarda a informação sobre os utilizadores, equipas, projetos e *tokens* de autenticação. Quando um utilizador efetua o processo de autenticação no editor, é gerado um *token* único na base de dados que fica associado ao utilizador e ao seu IP sendo apenas válido nessas condições.

A Figura 21 ilustra o modelo da base de dados da aplicação.

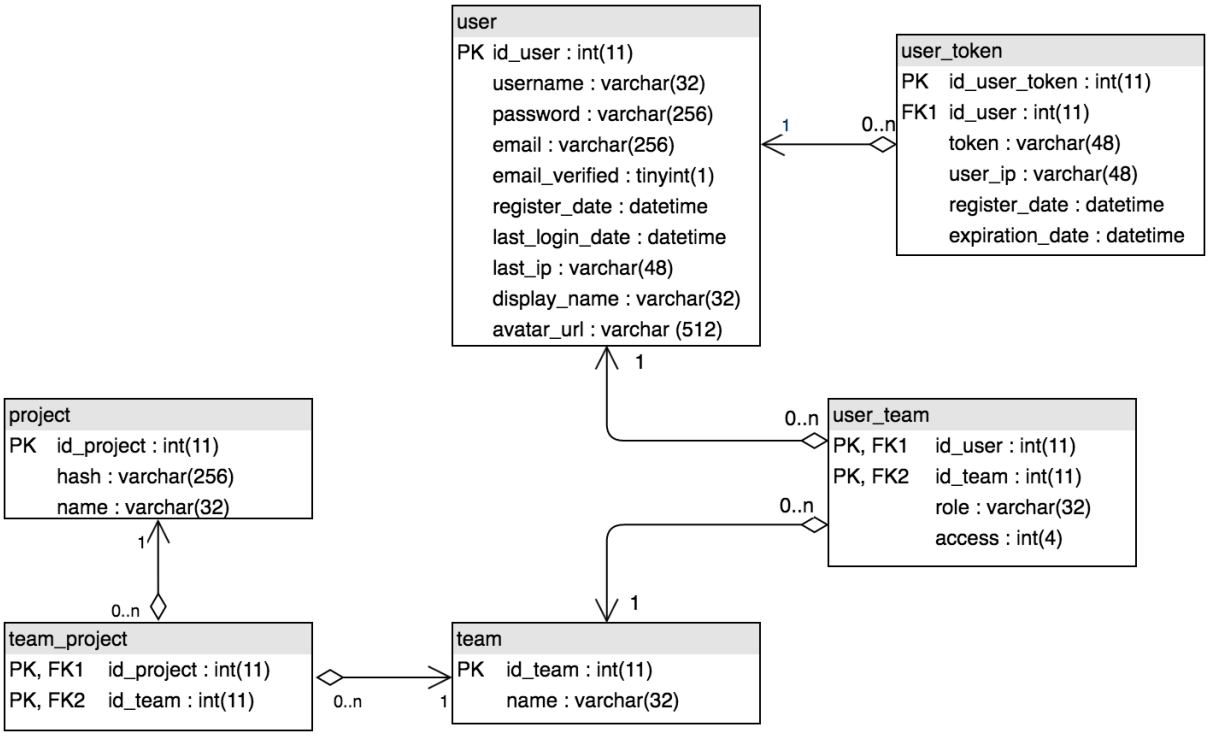

Figura 21 - diagrama da base de dados do Scarlett

Tabela "user"

- **id_user –** chave primária
- **username –** nome de utilizador
- **password –** senha de utilizador (encriptada com sha-256)
- **email –** email do utilizador
- **email_verified –** identifica se o email foi confirmado
- **register_date –** data de registo

- **last_login_date –** data do último login efetuado no editor
- **last_ip –** último IP do utilizador
- **display_name –** nome visível do utilizador
- **avatar_url –** endereço da imagem do utilizador

Tabela "user_team"

- **id_user –** elemento da chave primária e chave estrangeira de *user.id_user*
- **id_team –** elemento da chave primária e chave estrangeira de *team.id_team*
- **role –** posição na equipa atribuída pelo o utilizador
- **access –** nível de acesso sobre as permissões da equipa

Tabela "team"

- **id_team –** chave primária
- **name –** nome da equipa

Tabela "team_project"

- **id_project –** elemento da chave primária e chave estrangeira de *project.id_project*
- **id_team –** elemento da chave primária e chave estrangeira de *team.id_team*

Tabela "project"

- **id_project –** chave primária
- **hash –** identificador único do projeto
- **name –** nome do projeto

6.2 Serviço Web

O serviço web oferece várias funcionalidades que podem ser invocadas remotamente usando uma interface SOAP desde que exista acesso à internet:

- **Criar Conta** – Esta funcionalidade permite que os utilizadores criem contas no sistema que são armazenadas na base de dados.
- **Efetuar Login** – Esta funcionalidade permite que os utilizadores efetuem login no sistema. Aquando desta ação, é gerado um token que apenas o utilizador pode utilizar com o objetivo de efetuar operações.
- **Modificar dados de Utilizador** – Esta funcionalidade permite que os utilizadores alterem os seus dados incluindo nome e imagem.
- **Criar equipa –** Esta funcionalidade permite que os utilizadores criem equipas de trabalho.
- **Modificar dados de equipa –** Esta funcionalidade permite que os utilizadores alterem os dados da equipa como por exemplo o nome.
- **Adicionar elemento a uma equipa –** Esta funcionalidade permite que os utilizadores adicionem elementos a uma equipa de trabalho

- **Remover elemento de uma equipa –** Esta funcionalidade permite que os utilizadores removam elementos de uma equipa de trabalho.
- **Criar Projeto** – Esta funcionalidade permite que os utilizadores criem projetos para os armazenar remotamente.
- **Carregar Projeto** – Esta funcionalidade permite que os utilizadores obtenham um projeto associado a uma equipa de trabalho a que pertencem.
- **Guardar Projeto** – Esta funcionalidade permite que os utilizadores guardem um projeto remotamente.

A Figura 22 ilustra o fluxo do processo de login apresentando os diferentes componentes envolvidos na operação:

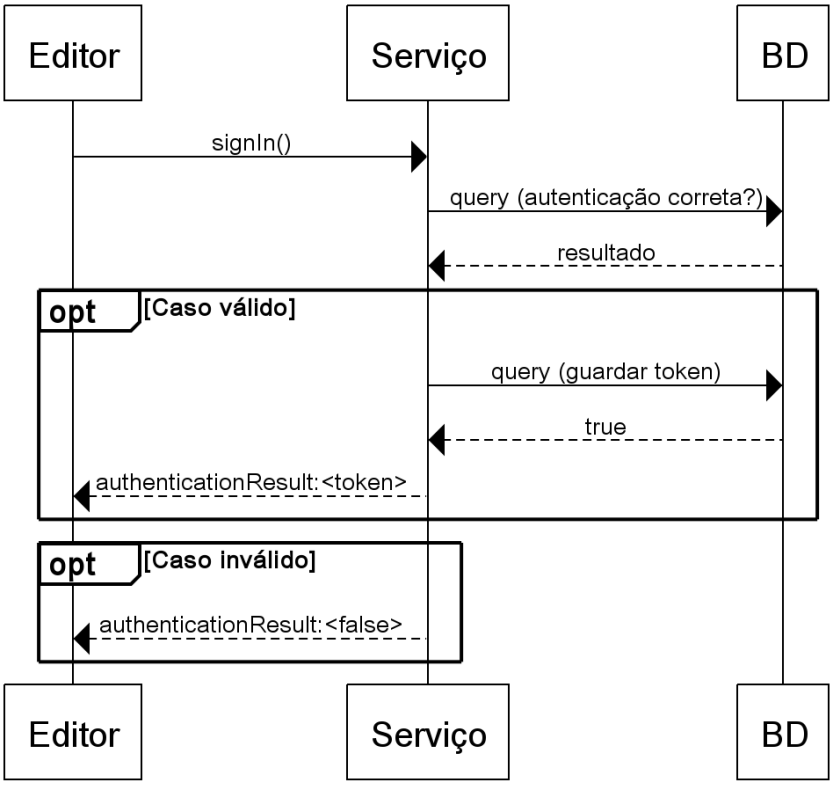

Figura 22 - fluxo de login

6.2.1 SOAP

SOAP é um acrónimo de "*Simple Object Access Protocol*". É um sistema de troca de mensagens baseado em *XML* (e segue a especificação *XML*) que permite a troca informação entre dispositivos diferentes. Foi desenhado com base na especificação *XML-RPC* sendo que utiliza alguns dos seus mecanismos e padrões relacionados com a troca e especificação das mensagens (Scibner & Stiver, 2000).

O protocolo de transporte utilizado não faz parte da sua especificação, mas tipicamente é realizado via *HTTP*, como é o caso desta implementação. A funcionalidade de um serviço SOAP é

independente da linguagem de programação ou plataforma sendo que apenas especifica principalmente as seguintes camadas:

- Formato da mensagem (a estrutura do corpo do conteúdo que é transmitido)
- O padrão de troca de mensagens (conhecido como *MEP – Message Exchange Pattern*).
- Os modelos de processamento de mensagem.

Uma mensagem SOAP é basicamente um documento XML que contém os seguintes elementos:

Tabela 2 - blocos de uma mensagem SOAP

Elemento	Descrição	Obrigatório
Envelope	Identifica o documento XML presente como uma mensagem SOAP.	Sim
Header	Contém a informação sobre o cabeçalho do documento.	Não
Body	Contém toda a informação relacionada com a mensagem transmitida.	Sim
Fault	Identifica possíveis erros que possam ter ocorrido durante o processamento e troca da mensagem.	Não

Exemplo de uma mensagem *SOAP* devolvida pelo serviço do Scarlett Game Studio:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <SOAP-ENV:Body>
        <ns1:requestResponse
xmlns:ns1="http://anlagehub.com/scarlett_ws/service.php?wsdl">
            <return xsi:type="xsd:string">{"result":{"code":5,"message":"Invalid
parameters"}}</return>
        </ns1:requestResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

6.2.2 Princípios de Implementação

A implementação do serviço web foi realizada em *PHP*. A disposição e descrição das funcionalidades disponíveis pelo serviço, foram realizadas usando a biblioteca *nuSOAP* que é uma das alternativas atuais para a criação simplificada de interfaces *SOAP* nesta linguagem.

O registo de funções na biblioteca nuSOAP é efetuado da seguinte forma:

```
$server = new soap_server();
$server->configureWSDL("server", SERVICE_NAMESPACE);
$server->soap_defencoding = 'UTF-8';

// example function:
function request($request) {
   // ..

   return x;
```

```
}

$server->register("request",
                  array("request" => "xsd:string"), // lista de parametros
                  array("return" => "xsd:string")); // lista de objetos resultante
```

No caso em cima apresentado, especificamos que existe um argumento (*request*) que é do tipo *String* e também que o tipo de objeto devolvido pela função é do tipo *String*.

Dependendo da funcionalidade a executar, no serviço do Scarlett existem várias definições com argumentos diferentes. Por exemplo, no caso do registo de uma conta de utilizador, é enviada informação sobre o nome de utilizador, password, email, etc.

Com o objetivo de devolver um resultado composto (incluindo pelo menos o tipo de resultado e respetiva mensagem explicativa), o conteúdo devolvido pelas funções do serviço está transformado no formato *JSON* contendo por tanto múltiplos elementos.

Exemplo de uma mensagem devolvida pelo serviço:

```
{"result":{"code":5,"message":"Invalid parameters"}}
```

O tipo de uma mensagem resultante pode ser um dos seguintes:

- ***Success*** – Pedido efetuado com sucesso.
- ***NotAuthorized*** – Pedido sem autorização (acontece por norma quando o token enviado não está correto, ou encontra-se inválido)
- ***ApiError*** – Indica que ocorreu um erro interno no processamento do pedido.
- ***NoInformation*** – Indica que o processo foi executado, mas não gerou nenhum resultado.
- ***InvalidParameters*** – Indica que a função foi invocada com um número ou formato errado de parâmetros.

6.2.3 Integração com a base de dados

A base de dados apresentada no subcapítulo 6.1 é operada diretamente pelo PHP usando a extensão *MySQLi*. Com o objetivo de manter alguma modularidade e consistência, foi desenvolvido uma classe de auxilio à utilização desta extensão (pode ser consultada no anexo 3) que permite executar consultas e efetuar alterações no conteúdo da base de dados.

6.2.4 Segurança

Todos os pedidos realizados pelo o editor (salvo os de registo e login) passam por uma verificação de token de utilizador. O token é um código com 48 carateres gerado e fornecido ao editor quando o utilizador efetua o processo de login. Esse valor é depois enviado em todos os pedidos subsequentes sendo validado sempre antes de qualquer execução.

O token é associado ao utilizador e ao seu IP remoto no momento em que efetuou o login. Desta forma, mesmo que alguém seja capaz de "apanhar" o token de outro utilizador, não o pode utilizar

caso não esteja na mesma rede. É importante saber que o IP de utilizador é resolvido no lado do servidor tornando ainda mais segura a validação.

Toda a comunicação com o serviço é sob o protocolo *HTTPS*, sendo que todo o conteúdo transferido é encriptado e não observável. A ligação é estabelecida em *TLS* sendo que existe um *handshake* inicial onde são trocadas as informações e cifras de segurança associadas a esta camada de comunicação.

A Figura 23 ilustra uma captura realizada usando a aplicação *Wireshark* com o objetivo de observar o conteúdo transferido no serviço do Scarlett Game Studio ao invocar uma função da API. Como é possível verificar, todo o conteúdo capturado está encriptado não facilitando ações maliciosas por outros indivíduos.

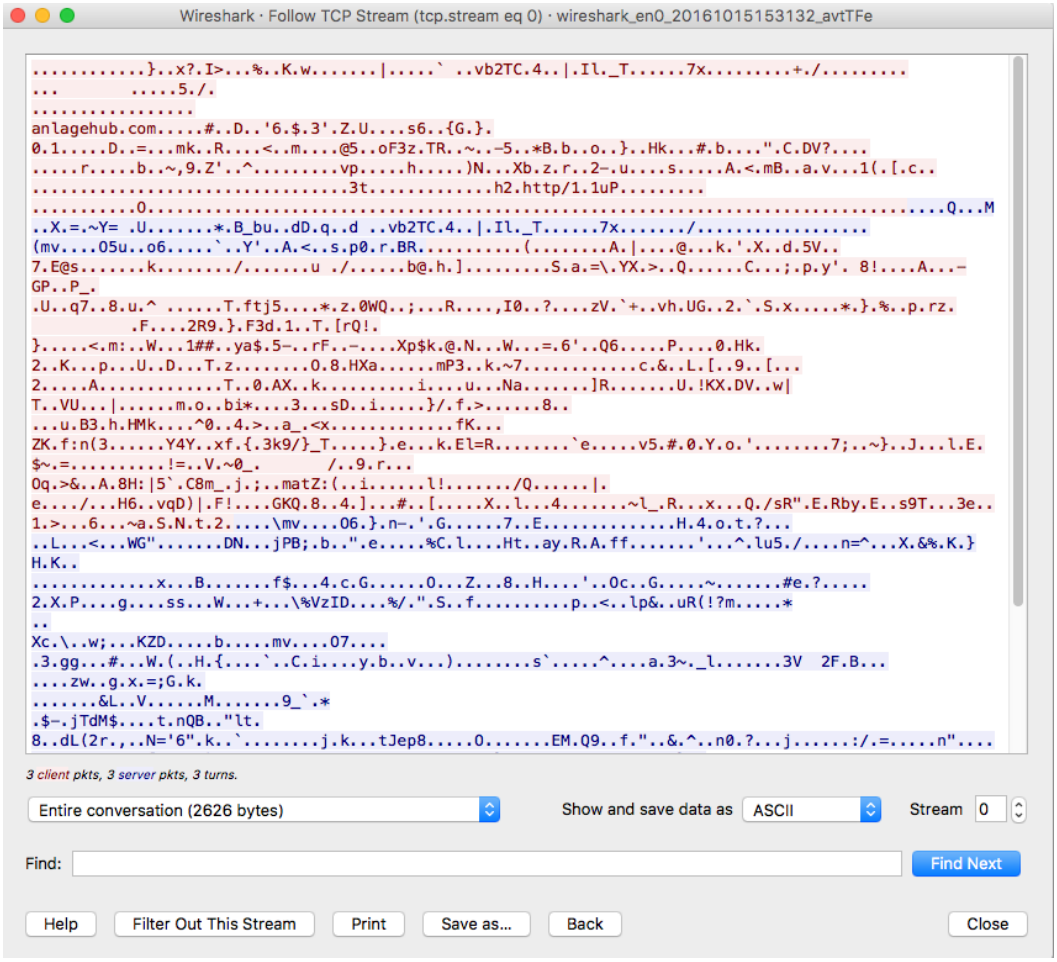

Figura 23 - exemplo de captura sobre o conteúdo encriptado do serviço

6.3 Escalabilidade

Todo este módulo foi criado com a consciência de escalabilidade. Tanto o serviço web como a base de dados podem ser distribuídas horizontalmente por diferentes maquinas no sentido de aumentar a capacidade de utilizadores ativos num determinado momento.

Com a utilização de sistemas de balanceamento de carga (como por exemplo *HAProxy*), é possível distribuir eficientemente trafego de rede (*HTTP, TCP*...) por vários servidores. A base de dados pode ser alocada numa só máquina ou num cluster e o serviço web pode ser facilmente colocado em diferentes servidores. É de a responsabilidade do sistema de balanceamento de carga atribuir um IP de um utilizador a uma das máquinas associadas sendo a sua descrição e aplicação fora do âmbito deste desenvolvimento.

A Figura 24 ilustra um exemplo de distribuição de múltiplos serviços com um sistema de balanceamento de carga:

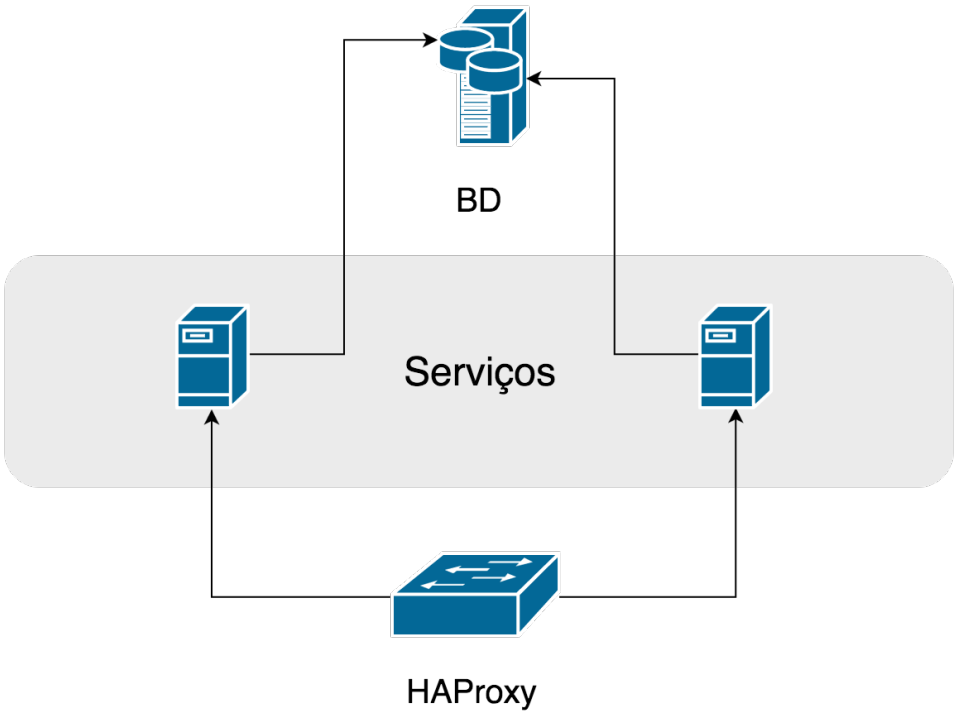

Figura 24 - distribuição de serviços web

7 Editor

O editor do Scarlett Game Studio é construído com a utilização de múltiplas tecnologias. A interface de utilizador foi desenvolvida usando tecnologias web (*HTML, Javascript, CSS*...) podendo ser executadas em qualquer recipiente apropriado para o efeito (por ex. *Chromium*). Em adição, foi criada uma interface nativa que permite interagir nativamente com os diferentes sistemas operativos suportados (Windows, Mac OS e Linux).

7.1 Recipiente de execução

O recipiente de execução do Scarlett Game Studio é o *Electron* que utiliza o *Chromium* e *Node.JS* fornecendo assim um ambiente de desenvolvimento completo com suporte a tecnologias web. Este recipiente encontra-se em código aberto e conta com uma elevada comunidade de contribuidores.

Uma das grandes vantagens do *Electron* é a sua compatibilidade com diferentes sistemas operativos incluindo Mac OS, Windows e Linux, cumprindo assim um dos requisitos principais do Scarlett.

Outra capacidade interessante é a sua interação nativa com os diferentes sistemas operativos, como já foi mencionado, este recipiente utiliza *Node.JS* o que permite executar determinadas operações nos sistemas dos utilizadores.

É importante notar que existem dois processos principais numa aplicação executada pelo Electron. Um é o processo de apresentação onde o conteúdo é mostrado ao utilizador (semelhante ao processo disponível quando uma página é executada num navegador web) e existe também o processo principal que fornece um acesso menos restrito sobre o sistema operativo. A comunicação entre estes dois processos é possível com a utilização de um canal IPC fornecido pelo Electron, o que é desejável pois existe a necessidade neste caso de uma interação desimpedida sobre o sistema operativo (por exemplo para obter os ficheiros de uma pasta).

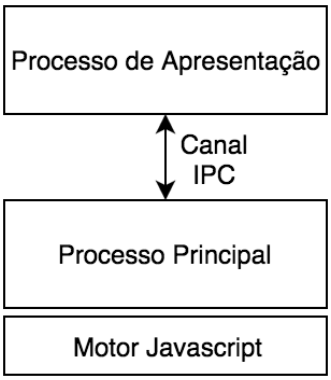

Figura 25 - processos do Electron

No Scarlett Game Studio existem módulos pertencentes a ambos os processos. No processo principal foram criadas interfaces distintas de acesso aos diferentes sistemas operativos tendo em

consideração as diferenças relevantes que existem entre eles. No processo de apresentação esta implementada toda a estrutura relativa à interface de utilizador.

Os seguintes blocos de código apresentam um exemplo de comunicação (usando o canal IPC) entre ambos os processos:

```
// No processo principal:
const {ipcMain} = require('electron');
ipcMain.on('asynchronous-message', (event, arg) => {
   // ...
})

// no processo de apresentação:
const {ipcRenderer} = require('electron');
ipcRenderer.send('asynchronous-message', "hello world");
```

7.2 AngularJS

Com o objetivo de tornar o desenvolvimento do editor mais ágil e declarativo, foi utilizada a framework *AngularJS* que possui um conjunto de mecanismos apropriados a este género de implementação.

O *AngularJS* utiliza o padrão de desenvolvimento *MVC* (*Model-View-Controller*) que permite isolar a lógica da aplicação da apresentação da interface de utilizador. O *HTML* por si não fornece mecanismos de declaração dinâmica de conteúdos. Um dos pontos mais fortes do *AngularJS* é a possibilidade de estender o vocabulário existente no *HTML* tornando-o dinâmico e mais expressivo (Karpov & Netto, 2015). Esta framework pode ser integrada com outras bibliotecas Javascript e qualquer das suas funcionalidades implementadas podem ser alteradas conforme necessário.

Outro aspeto muito interessante do *AngularJS* é a sua capacidade de *Data Binding* bidirecional. O conteúdo de apresentação é compilado pelo recipiente contendo os valores definidos no documento automaticamente. Desta forma não é necessário atualizar o conteúdo manualmente sempre que um determinado valor é alterado, a framework fica responsável por essa tarefa (Karpov & Netto, 2015).

7.2.1 Arquitetura base de uma aplicação em AngularJS

Na sua essência, uma aplicação que utiliza AngularJS consiste em três componentes que podem ser descritos da seguinte forma:

- **Template** – o *template* é a estrutura *HTML* da aplicação ou fragmento. Construir um template é exatamente o mesmo que escrever uma página *HTML* exceto que contém sintaxe adicional permitindo que informação seja apresentada do modelo de dados. De uma certa forma é um pouco semelhante a uma experiencia de desenvolvimento em páginas geradas no lado do servidor (por exemplo com *PHP*). Porém uma das grandes diferenças é que em *AngularJS* a informação pode ser atualizada sem nunca haver uma atualização na página.

- **Controller** – o *controller* (controlador) oferece uma posição de suporte nas aplicações AngularJS onde a lógica de um determinado template é definida.

- **Scope** – o *scope* é um componente que representa o modelo da aplicação/controlador. Contém propriedades que guardam valores que depois podem ser apresentados no template. Cada controlador possui um *scope* distinto que possui noção hierárquica sobre os modelos que o precedem, sendo que é possível aceder diretamente aos dados existentes em níveis hierárquicos anteriores.

O seguinte bloco de código apresenta um exemplo de um template *HTML* em *AngularJS:*

```
<html ng-app="myApp">
<body>
    <p>{{myText}}</p>
</body>
</html>
```

O seguinte bloco de código mostra um exemplo de um controlador em Javascript associado ao template em cima apresentado:

```
angular.module('myApp', [])
.controller("index", ["$scope", function ($scope) {
   $scope.myText = "This will appear on the template :)";
}]);
```

7.2.2 Serviços

No subcapítulo anterior foi apresentado um exemplo de um controlador muito básico cuja única função era a definição de um valor do *scope*. Como é possível verificar, a instancia do *scope* foi passada por argumento, em *AngularJS* denominamos este mecanismo por injeção de dependências.

As dependências tipicamente são serviços que em *AngularJS* podem ser descritos como componentes que permitem extrair a lógica da aplicação dos controladores com o objetivo de a encapsular em contentores disponíveis a todos os constituintes da aplicação.

Um exemplo prático de uma boa utilização de um serviço é, por exemplo, uma interface de comunicação com um servidor. Não seria prático nem ideal implementar toda a lógica de ligação e transferência de conteúdos em cada controlador, sendo a utilização de um serviço a melhor opção pois com uma implementação única capacitam-se todos os componentes.

O seguinte bloco de código apresenta o serviço de comunicação *SOAP* implementado no Scarlett Game Studio:

```
app.factory("soapSvc", function ($q, config, logSvc) {
   return {
      invoke: function (action, data) {
         var deferred = $q.defer();
         var param = {action: action, data: data};
         var soapParams = new SOAPClientParameters();
         soapParams.add("request", JSON.stringify(param));
         SOAPClient.invoke(config.API.ADDRESS, "request", soapParams, true,
            function (response) {
               if (typeof response !== "undefined" && response != false) {
                  try {
                     deferred.resolve(JSON.parse(response));
                  } catch (error) {
                     logSvc.error("parse error (api call): " + error.message);
                     deferred.reject(error.message);
```

```
                }
            } else {
                deferred.reject(response);
            }
        });
        return deferred.promise;
    }
  }
});
```

7.3 Janelas de apresentação

O editor do Scarlett Game Studio possui quatro janelas de apresentação principais em que o utilizador interage diretamente.

- **Login** – Onde pode ser realizado o processo de login com os dados do utilizador, ou caso pretenda, iniciar em modo offline.

- **Registo** – Onde pode ser realizado o processo de registo. Nesta janela o utilizador introduz os dados pessoais num formulário apropriado.

- **Hub** – Onde são apresentados os últimos projetos carregados/desenvolvidos sendo possível a criação de novos.

- **Editor** – Onde todo o processo de desenvolvimento é realizado. Esta janela contém diversos componentes distintos que entre si fornecem um ambiente de criação completo a jogos de duas dimensões.

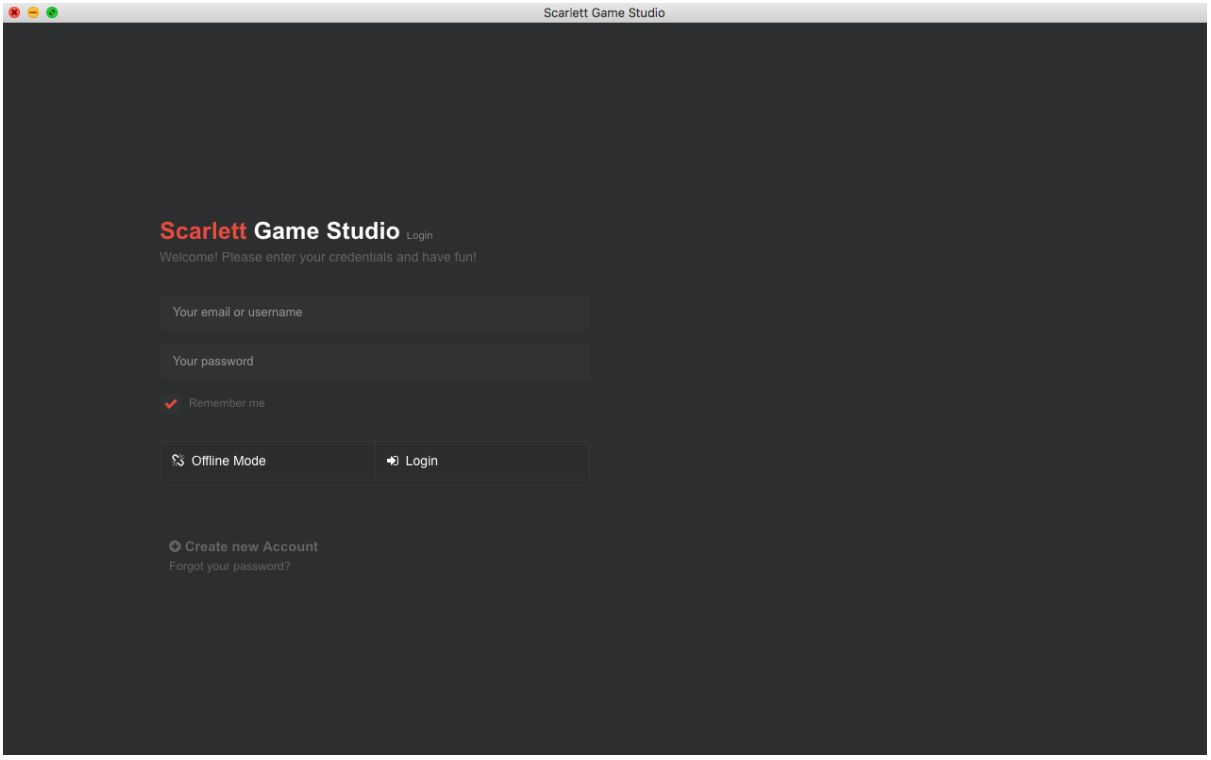

Figura 26 - ecrã de login

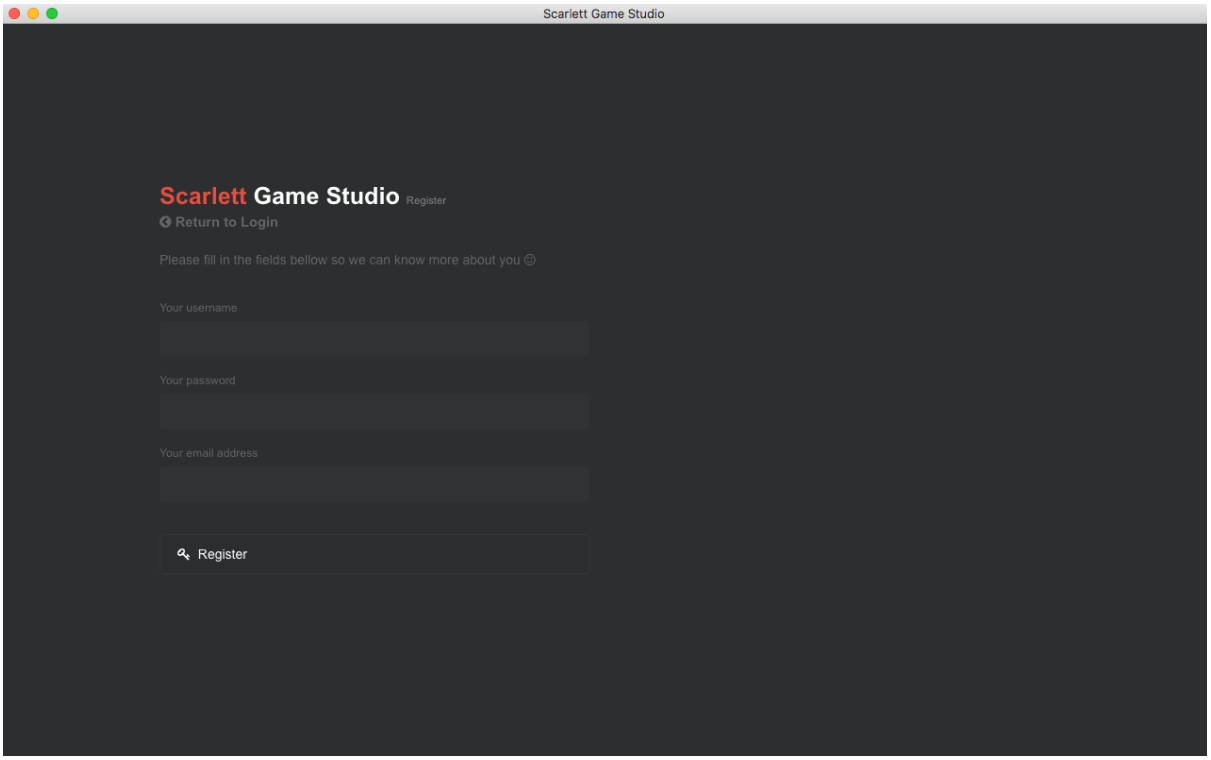

Figura 27 - ecrã de registo

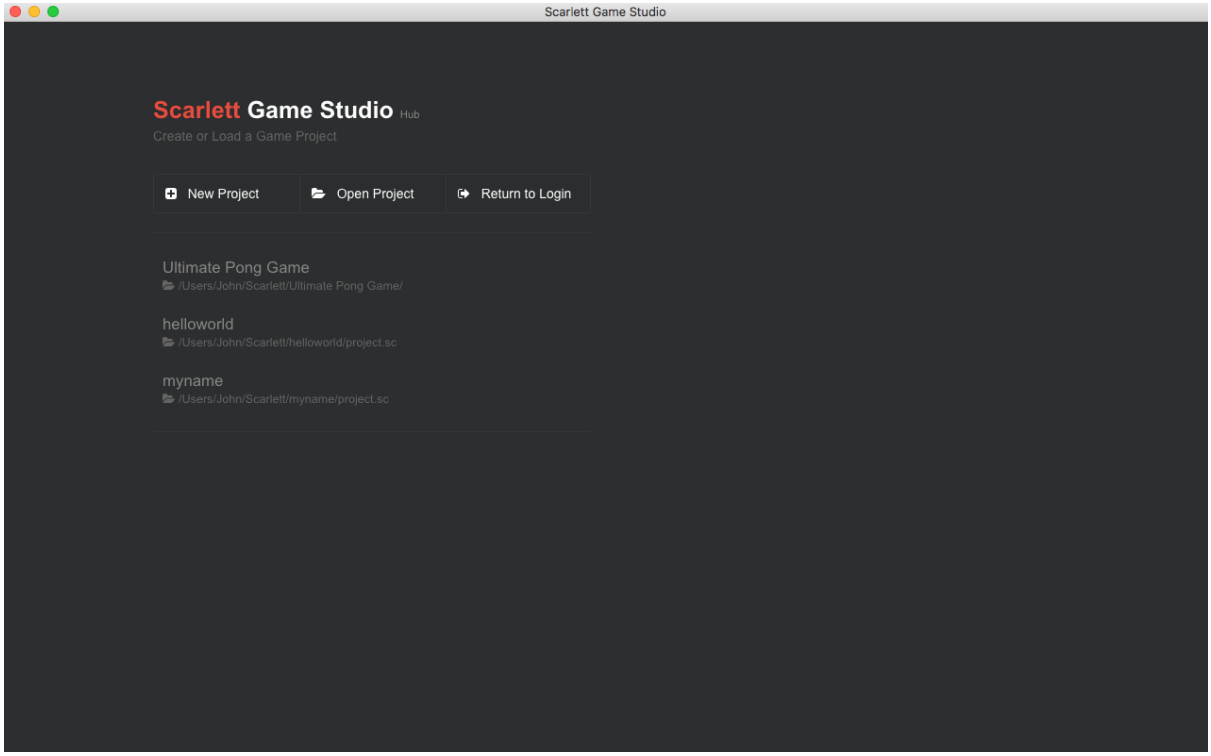

Figura 28 - ecrã de entrada

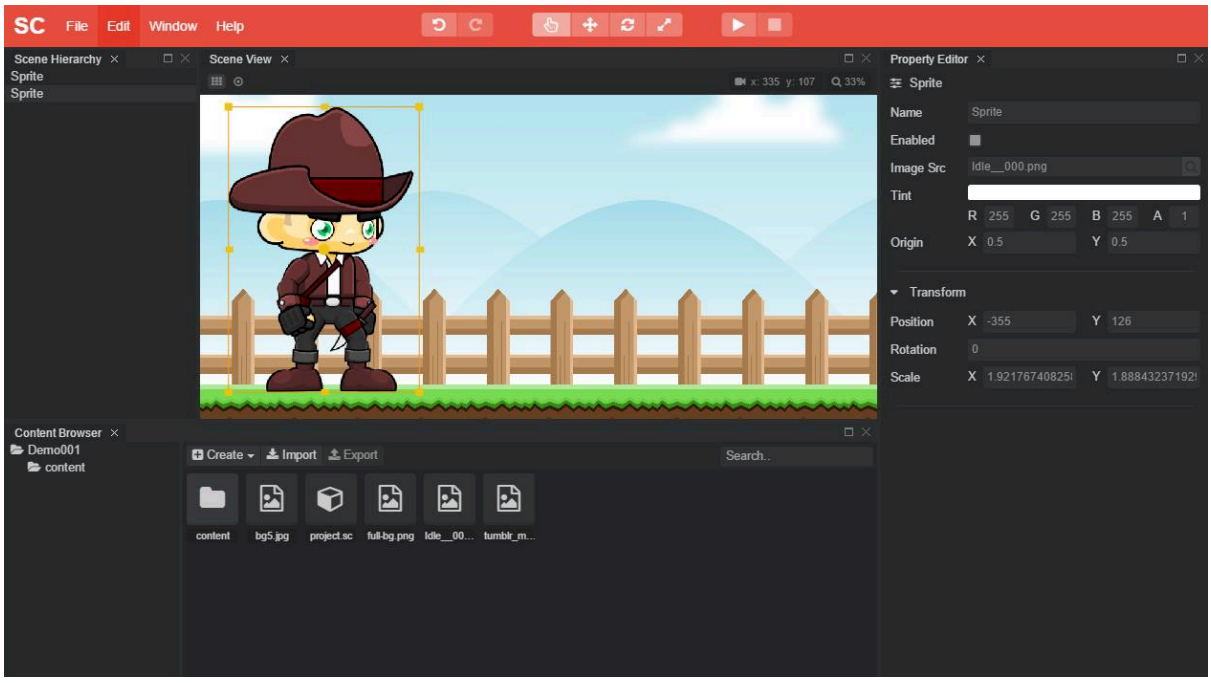

Figura 29 - ecrã do editor

7.4 Componentes do Editor

O editor está dividido em diferentes seções implementadas exclusivamente para este desenvolvimento ao qual denominamos por componentes ou ferramentas de edição. A dimensão e posição de cada componente pode ser alterada visualmente (por ex. ao arrastar o rato) pelo utilizador conforme as suas preferências. Neste subcapítulo estes vão ser apresentados com algum detalhe e particularidades da sua implementação.

7.4.1 Hierarquia do Cenário

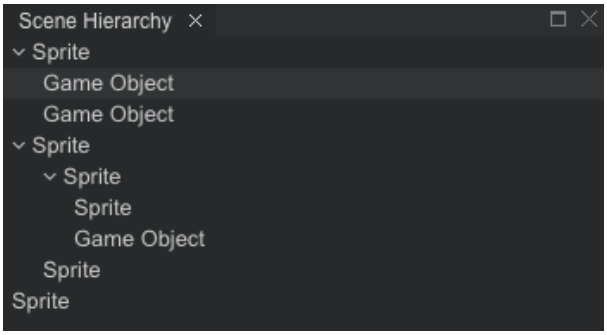

Figura 30 – componente de hierarquia do cenário

Este componente permite adicionar, gerir e visualizar a hierarquia de objetos de jogo existente no cenário em edição. A sua implementação foi realizada em *Javascript* e está integrado com *AngularJS*. Na sua essência é uma árvore visual de elementos em que cada um pode possuir ramificações com subelementos.

O template *HTML* definido para cada elemento é o seguinte:

```
<script type="text/ng-template" id="hierarchyNodeTemplate.html">

    <!-- header -->
    <div ng-class="{'selected' : isSelected()}">
        <div cz-tree-node-header context-menu="itemContextMenuOptions">
        <span>
            <i ng-if="node.nodes.length > 0" class="fa clickable"
               ng-class="{'fa-angle-down' : !collapsed, 'fa-angle-right' : collapsed}"
               ng-click="toggleCollapse()" aria-hidden="true"></i>
            {{node.gameObject.name}}
        </span>
        </div>
    </div>

    <!-- subelementos -->
    <div ng-class="{'hidden': collapsed}">
        <div ng-repeat="node in node.nodes" cz-tree-node uid="{{node.id}}"
             attachment="{{node.gameObject.getUID()}}" ng-
include="''hierarchyNodeTemplate.html''"></div>
    </div>

</script>
```

A estrutura em cima apresentada é chamada recursivamente até que todos os elementos tenham sido apresentados no ecrã. Associado a cada elemento existe um controlador que possibilita algumas operações como por exemplo selecionar e colapsar.

7.4.2 Explorador de conteúdo

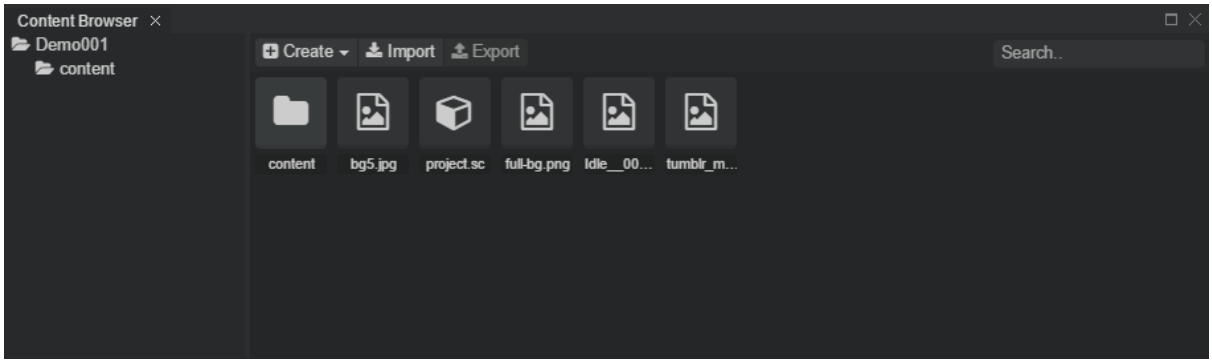

Figura 31 – explorador de conteúdo do projeto

Este componente permite gerir e visualizar todo o conteúdo associado ao projeto de desenvolvimento ativo no editor. O controlo está divido por duas seções, à esquerda possui uma árvore de navegação onde as pastas são apresentadas, e à direita são apresentados os ficheiros existentes na pasta selecionada.

A implementação da navegação de pastas é bastante similar à existente no componente de hierarquia do cenário possuindo as mesmas funcionalidades. Em contraste, possui um leque de operações mais vasto sendo mesmo possível adicionar mais pastas, scripts ou cenários:

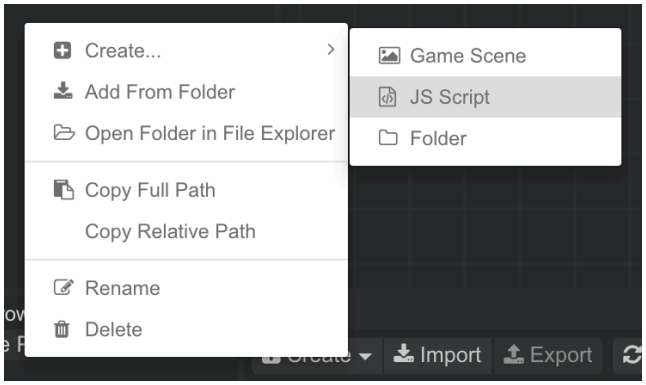

Figura 32 – menu de contexto do explorador de conteúdos

7.4.3 Editor de propriedades

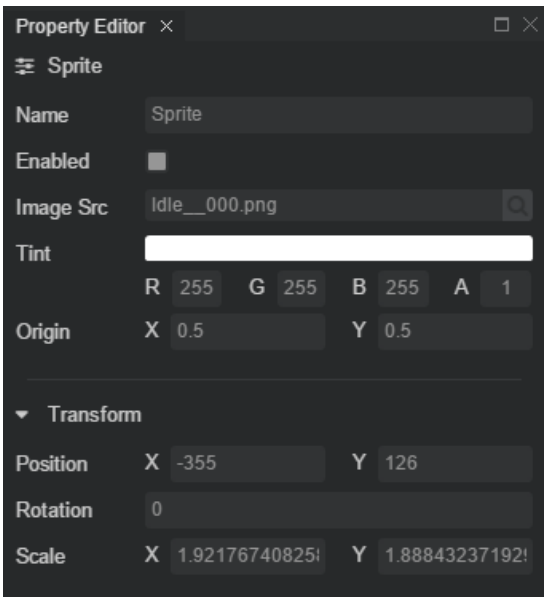

Figura 33 – editor de propriedades

Este componente permite visualizar e editar as diferentes propriedades existentes num ou vários objetos de jogo. Os componentes adicionados aos objetos de jogo também podem ser visualizados e editados neste painel. Cada tipo de propriedade possui um editor apropriado com funcionalidades relevantes na edição do mesmo.

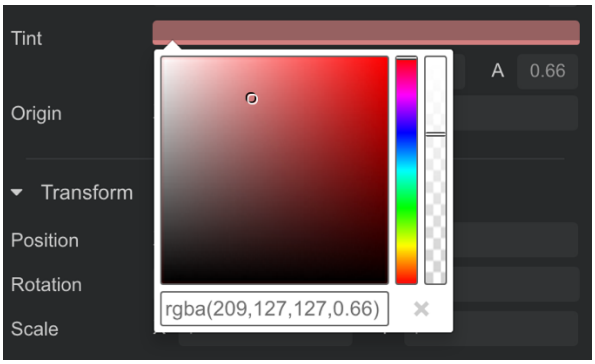

Figura 34 – editor especifico à seleção de uma cor

Uma particularidade muito interessante deste componente é a capacidade de edição de vários objetos em simultâneo. Quando mais do que um objeto é selecionado, o editor agrupa todos os componentes que sejam comuns na seleção e identifica possíveis casos em que existam diferenças nos valores nas propriedades associadas. No caso de existirem diferenças, elas são apresentadas ao utilizador visualmente:

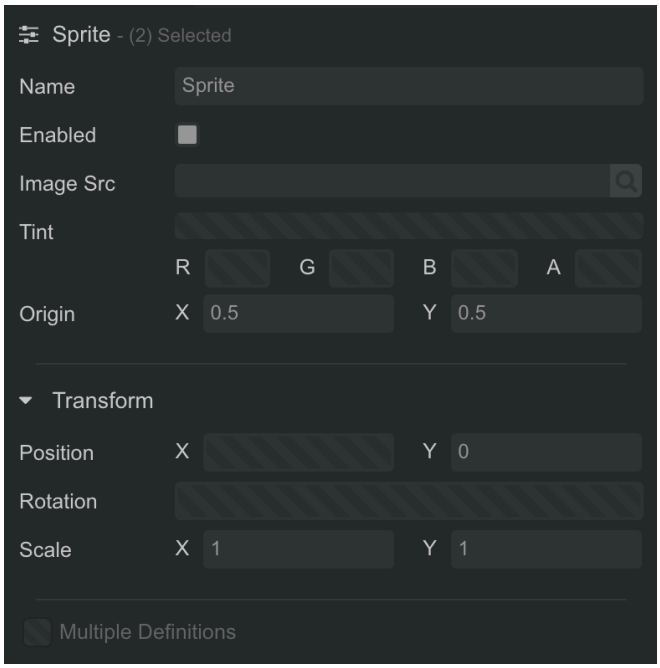

Figura 35 – editor de propriedades (múltipla seleção)

As propriedades que possuem valores diferentes no editor podem ser na mesma editadas, e quando isso acontece todos os objetos da seleção ficam com o mesmo valor. Desta forma é possível aplicar um valor desejado a vários objetos de jogo simultaneamente numa só operação.

Este componente é atualizado sempre que é realizada a operação de seleção de objetos no editor. A operação de seleção pode ser realizada de diferentes formas incluindo o componente de hierarquia do cenário. Sempre que a operação é realizada, é lançado um evento global com a notificação do mesmo e este editor é um dos recetores à escuta de tal ocorrência.

Com o objetivo de possuir um maior controlo sobre o que é efetivamente apresentado nas propriedades dos objetos, foi implementada a lógica de identificar as suas características. Originalmente o componente apresentava todas as propriedades existentes num objeto, sendo que o Javascript não possui o conceito de variáveis privadas tornando-se complicado limitar diretamente pela linguagem. Para resolver este problema e adicionar outras possibilidades (como por exemplo definir o nome de visualização da propriedade no editor de propriedades), foi criado um dicionário de propriedades que os utilizadores podem efetivamente definir algumas regras para as propriedades nos objetos.

O seguinte bloco de código apresenta um exemplo da utilização do dicionário de regras de propriedades existente no Scarlett Game Studio:

```
AttributeDictionary.inherit("sprite", "gameobject");
AttributeDictionary.addRule("sprite", "_textureSrc", {displayName: "Image Src", editor:
"filepath"});
AttributeDictionary.addRule("sprite", "_tint", {displayName: "Tint"});
AttributeDictionary.addRule("sprite", "_texture", {visible: false});
```

Ao adicionar uma regra especifica-se qual o tipo de objeto referido e respetiva propriedade. É também possível herdar definições aplicadas em outros objetos usando o método *inherit* que recebe como parâmetro os tipos dos dois objetos que devem ser associados.

Quando as propriedades estão a ser carregadas pelo editor, a existência de regras é validada e aplicada.

7.4.4 Editor de cenário

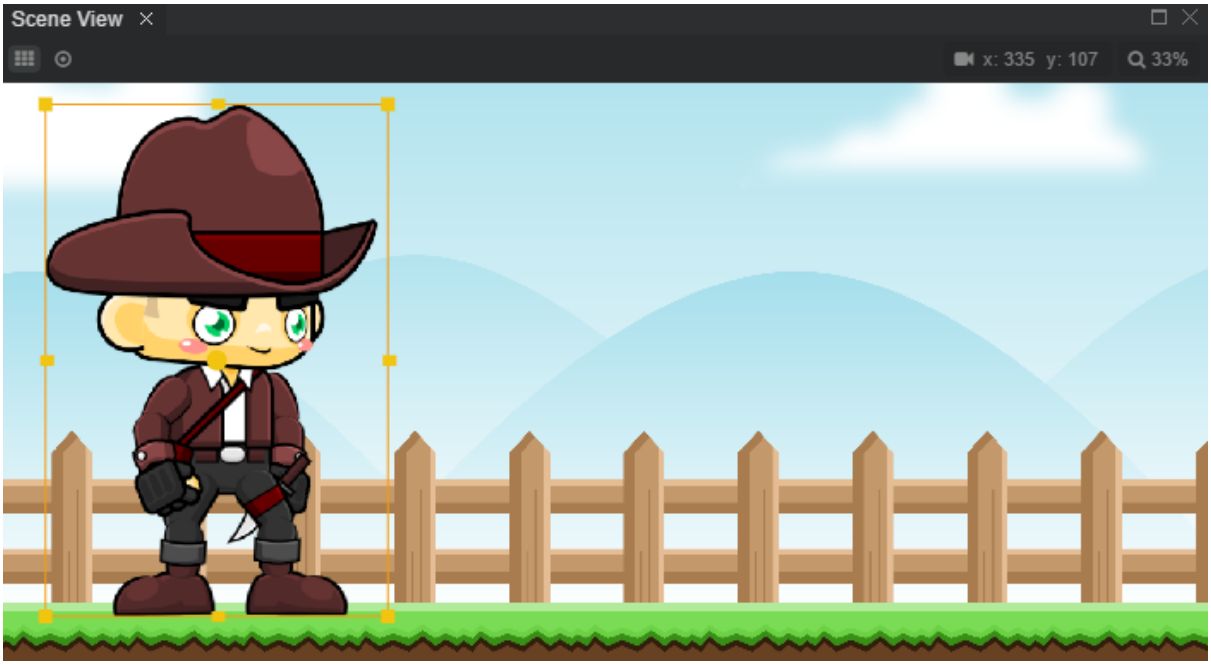

Figura 36 – editor de cenário

Este componente apresenta uma representação de todos os objetos de jogo existentes no cenário de jogo ativo no editor. Todo o conteúdo é apresentado com a utilização de *WebGL* sendo o seu contexto integrado com os componentes do editor.

Existem várias operações que podem ser realizadas sobre este componente estando associada a utilização de uma barra de ferramentas no topo do editor que permite filtrar as mesmas.

Figura 37 – barra de ferramentas (operações de transformação)

Quando o primeiro elemento está selecionado todas as operações de transformação e seleção estão disponíveis. Caso o utilizador pretenda limitar as operações a deslocação, rotação ou dimensão, deverá selecionar os botões correspondentes.

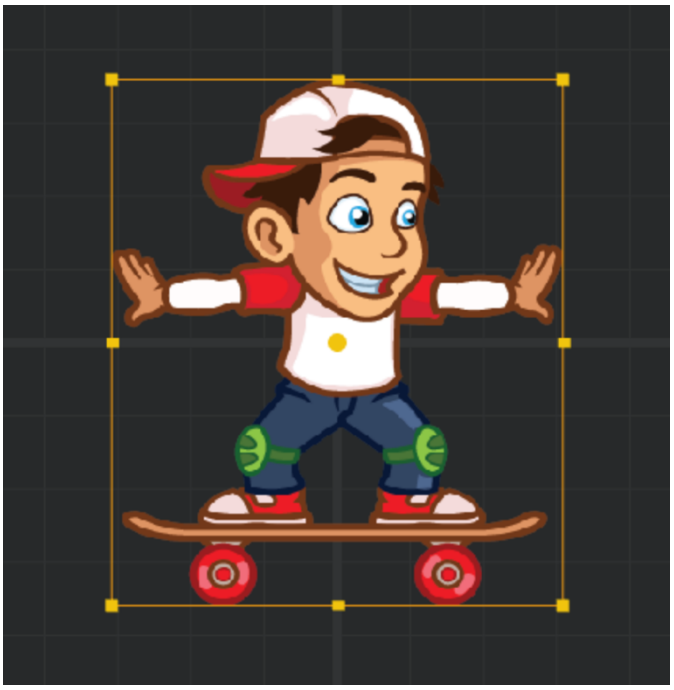

Figura 38 – exemplo de seleção de um objeto

Quando um objeto é selecionado é desenhado um retângulo com as dimensões do mesmo. Note-se que existem pontos de controlo nas extremidades da seleção, estas permitem efetuar operações de transformação sobre o objeto.

Com o objetivo de auxiliar a criação de cenários, foi também implementada uma grelha infinita que acompanha o editor de visualização independentemente da posição ou zoom da câmara.

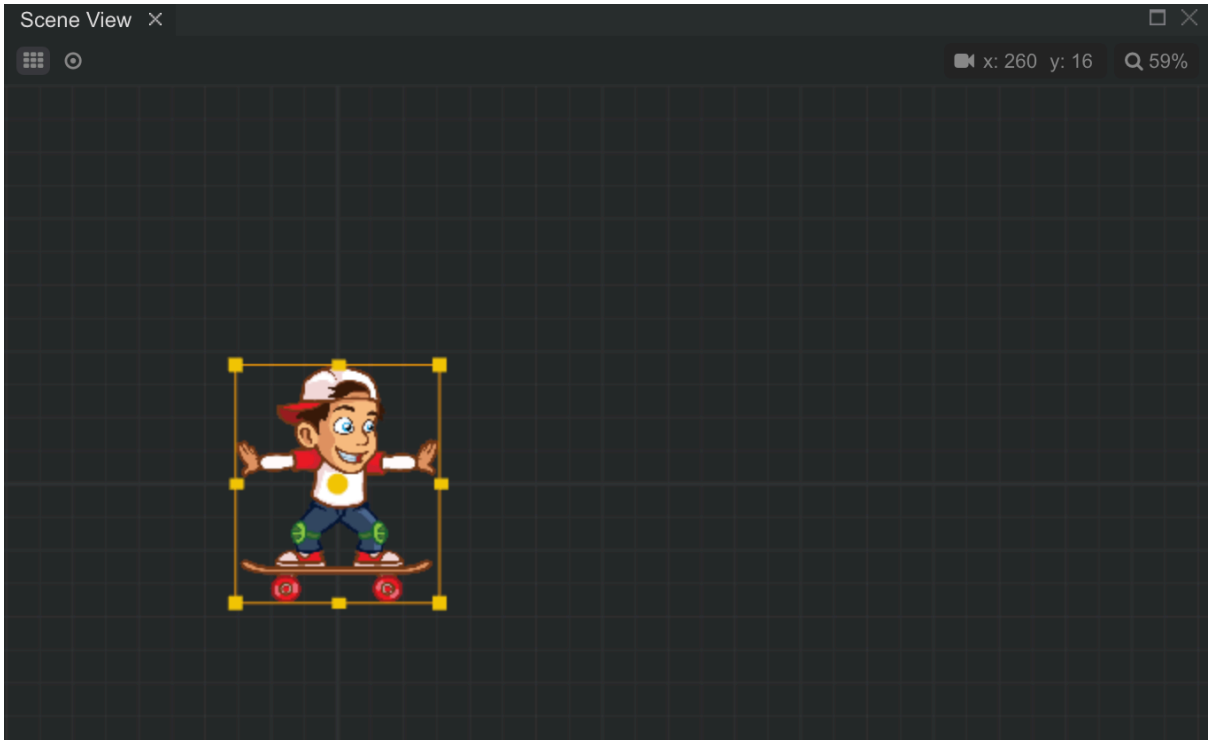

Figura 39 – grelha do editor de cenário

8 Avaliação

O processo de avaliação existente no Scarlett Game Studio pode ser dividido em diferentes categorias. Por um lado, existe uma avaliação técnica (por ex. testes unitários) sobre as funcionalidades implementadas, esta é realizada com o objetivo de validar se a aplicação possui um comportamento esperado ao efetuar uma determinada seleção de operações. Noutro sentido, é também muito importante a indicação e comentários dos utilizadores que utilizam a aplicação. Na presente data, a aplicação ainda não se encontra aberta ao público em geral, no entanto tem sido partilhada informação com uma seleção reduzida de indivíduos no sentido de validar as diferentes metodologias aplicadas.

8.1 Testes unitários

A realização de testes unitários no Scarlett Game Studio é efetuada com a biblioteca de testes *Jasmine* que possibilita a realização de ensaios orientados ao comportamento da aplicação.

Mais detalhadamente, o *Jasmine* é uma framework que permite a execução de testes em *Javascript* sem a necessidade de introduzir ou associar elementos DOM. A sintaxe aplicada é bastante simples e assemelha-se a uma representação textual comum.

O seguinte bloco de código apresenta um exemplo da utilização do Jasmine com uma validação de uma ação (logout) possível no editor do Scarlett Game Studio:

```
describe("User logout", function() {
    it("should logout", function() {
        expect(userSvc.logout()).toBe(true);
    });
});
```

Os testes podem ser definidos com uma determinada sequencia permitindo incluir pré-condições especificas baseadas em ações efetuadas. Por exemplo, para efetuar o teste de logout, é primeiro necessário efetuar um login.

A tabela seguinte apresenta algumas das operações de teste mais relevantes no contexto deste desenvolvimento e implicam todos os módulos da aplicação. Note-se que todos os testes apresentados foram validados com sucesso sobre os resultados expectáveis definidos:

Tabela 3 - casos de teste

C01: Criação de projeto numa pasta sem conteúdos	
Descrição	Esta ação valida a criação de um projeto sobre uma pasta sem conteúdos associados (vazia).
Resultado expectável	A operação é efetuada com sucesso.

C02: Criação de projeto numa pasta com conteúdos	
Descrição	Esta ação valida a criação de um projeto sobre uma pasta com conteúdos associados (preenchida).
Resultado expectável	A operação falha devido à pasta não estar vazia.

C03: Carregar Projeto	
Descrição	Esta ação valida o carregamento de um projeto associado ao Scarlett Game Studio.
Resultado expectável	A operação é efetuada com sucesso.

C04: Login com dados de utilizador válidos	
Descrição	Esta ação valida a operação de login com dados de utilizador válidos.
Resultado expectável	A operação é efetuada com sucesso.

C05: Login com dados de utilizador inválidos	
Descrição	Esta ação valida a operação de login com dados de utilizador inválidos.
Resultado expectável	A operação falha devido aos dados de utilizador não estarem corretos.

C06: Logout	
Descrição	Esta ação valida a operação de login com dados de utilizador válidos
Pré-condições	C04
Resultado expectável	A operação é efetuada com sucesso.

C07: Compilar scripts	
Descrição	Esta ação valida a operação de compilação de scripts associados a um projeto carregado.
Pré-condições	C03
Resultado expectável	A operação é efetuada com sucesso.

C08: Verificação da disponibilidade do serviço remoto	
Descrição	Esta ação valida se o serviço remoto está operacional ao receber uma resposta de uma mensagem modelo enviada.
Resultado expectável	A operação é efetuada com sucesso.

C09: Guardar cenário	
Descrição	Esta ação valida a funcionalidade de guardar cenários de jogo.
Pré-condições	C03
Resultado expectável	A operação é efetuada com sucesso.

8.2 Feedback e fases de avaliação

Associado ao desenvolvimento do Scarlett Game Studio pretende-se que haja uma linha de contacto aberta com os utilizadores da aplicação. Existem vários mecanismos preparados para que seja possível aos utilizadores fornecerem feedback sobre a sua experiencia de utilização:

- **Fórum** – neste meio os utilizadores podem fornecer sugestões e esclarecer possíveis dúvidas de utilização.

- **Avaliação integrada** – está prevista a inclusão de um sistema de avaliação integrado no editor da aplicação que irá permitir aos utilizadores avaliar a aplicação consoante alguns parâmetros (usabilidade, praticidade, customização...).

- **Questionários públicos** – após ser lançada a primeira versão pública da aplicação, será apresentado aos utilizadores um sistema de questionários com o objetivo de entender as necessidades e preferências dos mesmos.

Com o objetivo de causar um bom impacto inicial, está planeado em primeira instância o lançamento da aplicação a uma seleção reduzida de utilizadores. Esta fase tem como objetivo detetar possíveis problemas existentes na aplicação antes de ser lançada publicamente. Além disso é uma ótima oportunidade para perceber as preferências dos utilizadores e também o grau de satisfação sobre as funcionalidades presentes (vai ser fornecido um questionário a todos os utilizadores desta fase). Toda a informação sobre o processo de inscrição para esta fase foi disponibilizado no website do Scarlett Game Studio onde também é possível efetuar a subscrição associada através de um formulário online.

9 Trabalhos relacionados

Durante o processo de desenvolvimento, foram surgindo algumas tarefas e trabalhos relacionados que vão ser apresentados neste capítulo sucintamente.

9.1.1 Website Scarlett Game Studio

Com o objetivo de partilhar com o mundo o motor de jogo, foi criado um website com a informação e documentação associada ao projeto.

O mesmo pode ser consultado em: https://scarlett.anlagehub.com/

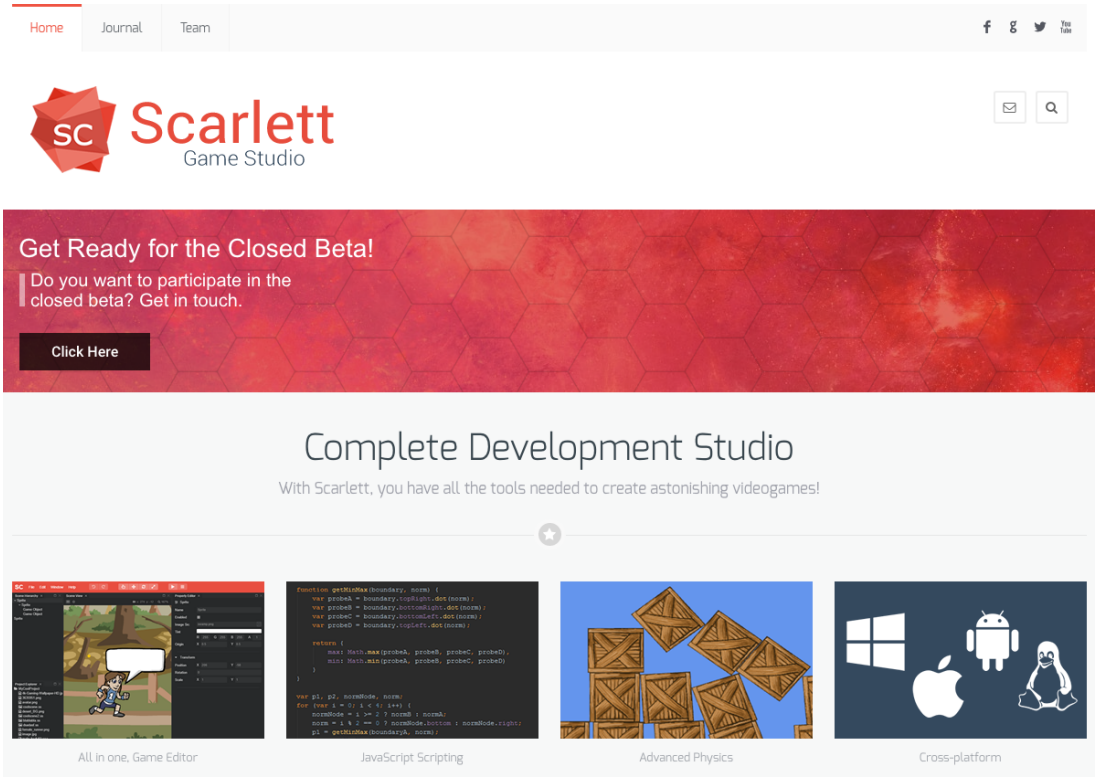

Figura 40 – captura de ecrã da página web principal do Scarlett

9.1.2 Criação de conteúdo associado

Durante o desenvolvimento existiu a necessidade de criar conteúdo adicional como por exemplo áudio e imagens. O logotipo também fez parte do processo:

Figura 41 – logotipo completo do Scarlett Game Studio

9.1.3 Diversos minijogos

A melhor forma de testar o desenvolvimento realizado foi com a implementação de minijogos que de uma forma ou de outra ajudaram a validar o comportamento e funcionalidade dos vários componentes envolvidos no motor de jogo.

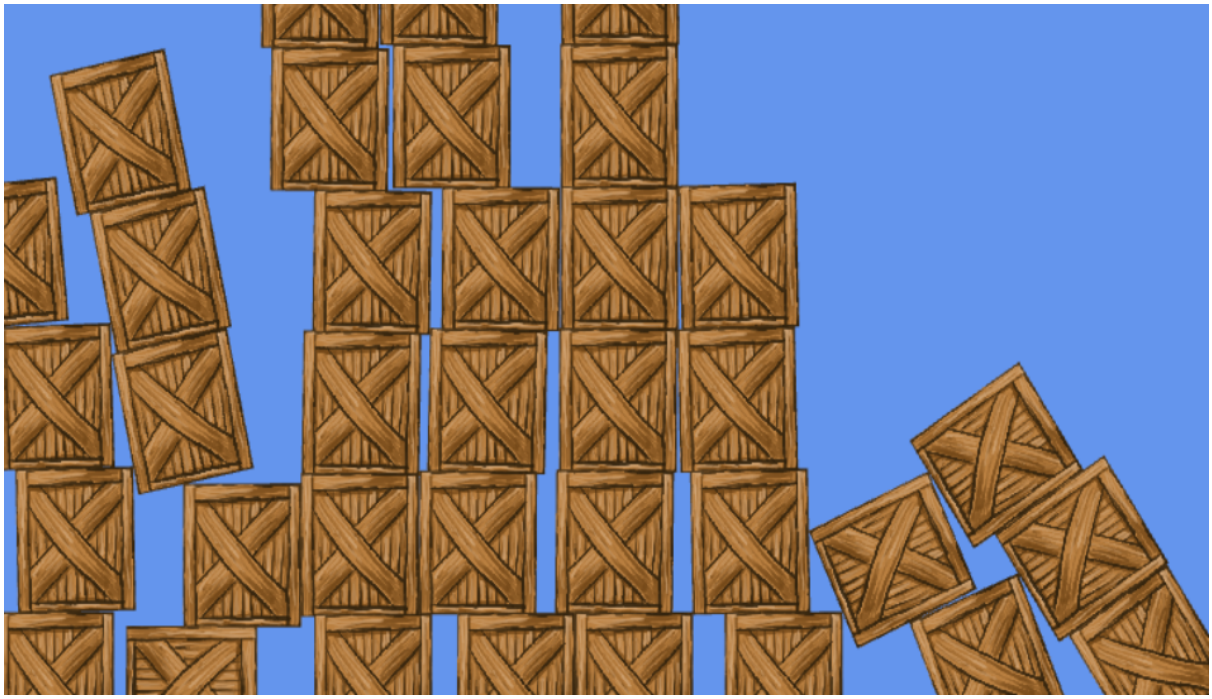

Figura 42 – captura de ecrã de um minijogo criado (*BoxAttack – utilização do sistema de física*)

9.1.4 Alojamento e manutenção de servidores

Associado a todos os componentes do lado do servidor (serviço, website...) faz parte o processo de alojamento e manutenção dos servidores envolvidos. Deste processo são realizadas várias tarefas como limpeza regular e execução de backups.

10 Conclusões finais

Sem dúvida que todo o desenvolvimento, planeamento e descrição deste projeto foram bastante gratificantes possibilitando a aprendizagem e aplicação de novos conhecimentos e estratégias. Confirma-se que todos os objetivos principais foram concluídos sendo que ainda estão traçadas algumas tarefas para desenvolvimento futuro. Durante o processo de implementação foram aprendidas diferentes técnicas e tecnologias que por fim foram elevadas ao âmbito proposto tornando-se também numa criação pessoal e evolutiva. Pode-se, portanto, considerar que foi acima de tudo uma experiencia enriquecedora a todos os níveis.

10.1 Objetivos realizados

Inicialmente foi proposto o desenvolvimento de um motor de jogo a duas dimensões com um leque de funcionalidades complexo e diverso. Associado ao processo de implementação foi planeada a criação e estrutura de diferentes módulos que em conjunto formam o Scarlett Game Studio:

- Framework
- Editor
- Serviço Remoto

A implementação destes módulos foi completada com sucesso podendo afirmar que o motor de jogo Scarlett Game Studio está preparado para ser utilizado na criação de videojogos em meios profissionais e pessoais.

Os requisitos principais e princípios de organização estabelecidos associados ao desenvolvimento também foram cumpridos, destacando-se os seguintes:

- Capacidade de desenvolver em diferentes plataformas
- Capacidade de execução em diferentes plataformas
- Implementação efetuada de forma modular
- Capacidade de manter informação e dados remotamente
- Flexibilidade de desenvolvimento associada à preferência dos utilizadores

10.2 Desenvolvimento futuro

Em adição a todas as funcionalidades implementadas, ainda existem outras que estão previstas para um desenvolvimento futuro. A seguinte lista apresenta algumas das funcionalidades planeadas mais interessantes:

- **Scripting Visual** – Capacidade de criação de scripts com ferramentas visuais.
- **Editor temático** – Possibilidade de editar um projeto com base num contexto ou tema especifico incluindo ferramentas distintas.
- **Animador incorporado** – Inclusão de um editor de animações no editor.

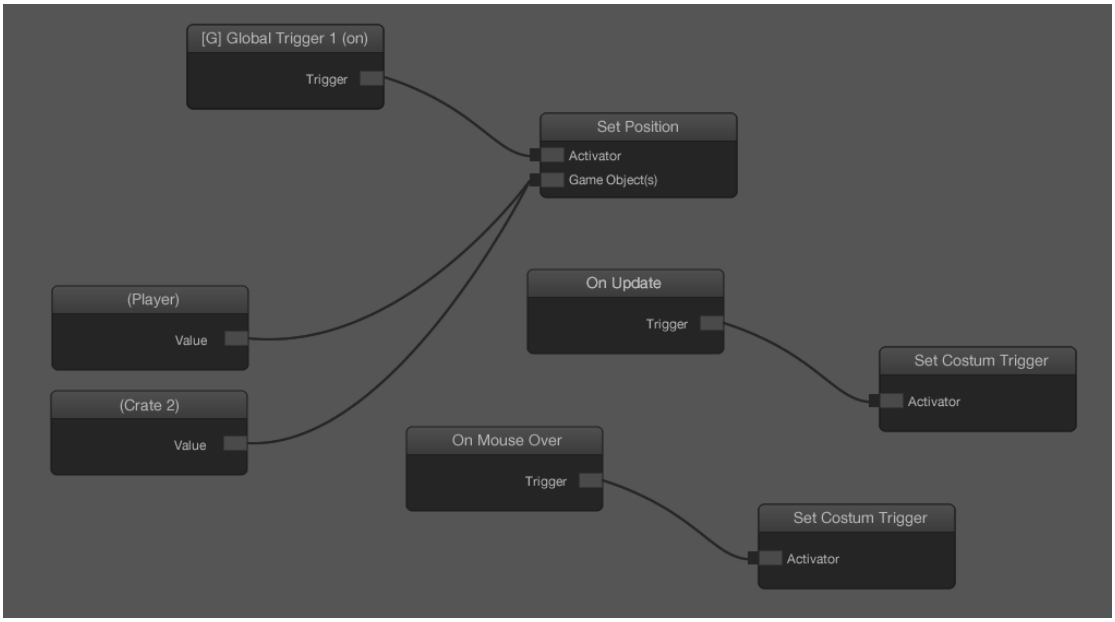

Figura 43 – esboço do sistema de scripting visual planeado

Referências

Kent, S. (2010). The Ultimate History of Video Games: from Pong to Pokemon and beyond... the story behind the craze that touched our lives and changed the world. Three Rivers Press.

Eberly, D. H. (2006). 3D game engine design: a practical approach to real-time computer graphics. CRC Press.

Lewis, M., & Jacobson, J. (2002). Game engines. Communications of the ACM.

Gregory, J. (2009). Game engine architecture. CRC Press.

Bourg, D. M. (2002). Physics for game developers. " O'Reilly Media, Inc."

Creighton, R. H. (2010). Unity 3D game development by example: A seat-of-your-pants manual for building fun, groovy little games quickly. Packt Publishing Ltd.

Dunn, F., & Parberry, I. (2015). 3D math primer for graphics and game development. CRC Press.

Hill, F., & Kelley, S. (2007). Computer graphics using OpenGL, 3/E. Pearson.

Huynh, J. (2009). Separating Axis Theorem for Oriented Bounding Boxes.

Parisi, T. (2012). WebGL: up and running. " O'Reilly Media, Inc.".

Scribner, K., Scribner, K., & Stiver, M. C. (2000). Understanding Soap: Simple Object Access Protocol. Sams.

Karpov, V., & Netto, D. (2015). Professional AngularJS.

Anexo 1 – WebGL Utils

```
/**
 * WebGL Utils class
 *
 */
function WebGLUtils() {
    // private fields
    this._logger = new Logger(arguments.callee.name);
}

/**
 * Compiles a shader
 * @param gl
 * @param shaderSource
 * @param shaderType
 */
WebGLUtils.prototype._compileShader = function(gl, shaderSource, shaderType) {
    // Create the shader object
    var shader = gl.createShader(shaderType);

    // Load the shader source
    gl.shaderSource(shader, shaderSource);

    // Compile the shader
    gl.compileShader(shader);

    // Check the compile status
    var compiled = gl.getShaderParameter(shader, gl.COMPILE_STATUS);
    if (!compiled) {
        // Something went wrong during compilation; get the error
        var lastError = gl.getShaderInfoLog(shader);

        this._logger.error("Error compiling shader '" + shader + "':" + lastError);

        gl.deleteShader(shader);

        return null;
    }

    return shader;
};

/**
 * Creates a program from 2 shaders.
 * @param gl
 * @param vertexShader
 * @param fragmentShader
 * @returns {WebGLProgram}
 */
WebGLUtils.prototype.createProgram = function (gl, vertexShader, fragmentShader) {
    // create a program.
    var program = gl.createProgram();

    // attach the shaders.
    gl.attachShader(program, vertexShader);
    gl.attachShader(program, fragmentShader);

    // link the program.
    gl.linkProgram(program);

    // Check if it linked.
    var success = gl.getProgramParameter(program, gl.LINK_STATUS);
    if (!success) {
        // something went wrong with the link
        this._logger.error("Program filed to link:" + gl.getProgramInfoLog(program));
    }

    return program;
};
```

```
/**
 * Creates a shader from the script string
 * @param gl
 * @param script
 * @param shaderType
 * @returns {null}
 */
WebGLUtils.prototype.createShader = function (gl, script, shaderType) {
    // If we didn't pass in a type, use the 'type' from
    // the script tag.
    var glShaderType;
    if (shaderType === "vertex") {
        glShaderType = gl.VERTEX_SHADER;
    } else if (shaderType === "fragment") {
        glShaderType = gl.FRAGMENT_SHADER;
    } else if (!shaderType) {
        this._logger.warn("Shader type not set, discarding..");
        return null;
    }

    return this._compileShader(gl, script, glShaderType);
};

/**
 * Creates a shader from the content of a script tag
 * @param gl
 * @param scriptId
 * @param shaderType
 */
WebGLUtils.prototype.createShaderFromScript = function (gl, scriptId, shaderType) {
    // look up the script tag by id.
    var shaderScriptElem = document.getElementById(scriptId);
    if (!shaderScriptElem) {
        this._logger.warn("Unknown script target element, discarding..");
        return null;
    }

    // extract the contents of the script tag.
    this.createShader(gl, shaderScriptElem.text, shaderType);
};

/**
 * Creates a program based on both vertex and fragment given scripts
 * @param gl
 * @param vertexScript
 * @param fragmentScript
 */
WebGLUtils.prototype.createProgramFromScripts = function(gl, vertexScript,
fragmentScript) {
    var vshader = this.createShader(gl, vertexScript, "vertex");
    var fshader = this.createShader(gl, fragmentScript, "fragment");

    if(isObjectAssigned(vshader) && isObjectAssigned(fshader)) {
        return this.createProgram(gl, vshader, fshader);
    } else {
        this._logger.warn("Could not create program because scripts could not be
compiled, discarding..");
    }

    // clean up shaders
    gl.deleteShader(vshader);
    gl.deleteShader(fshader);

    return null;
};

/**
 * Creates a program based on both vertex and fragment given elements
 * @param gl
 * @param vertexScriptId
 * @param fragmentScriptId
 */
WebGLUtils.prototype.createProgramFromScriptElements = function(gl, vertexScriptId,
fragmentScriptId) {
```

```
    var vshader = this.createShaderFromScript(gl, vertexScriptId, "vertex");
    var fshader = this.createShaderFromScript(gl, fragmentScriptId, "fragment");

    if(isObjectAssigned(vshader) && isObjectAssigned(fshader)) {
        return this.createProgram(gl, vshader, fshader);
    } else {
        this._logger.warn("Could not create program because scripts could not be
compiled, discarding..");
    }

    // clean up shaders
    gl.deleteShader(vshader);
    gl.deleteShader(fshader);

    return null;
};
```

Anexo 2 – Primitive e Texture Shader

```
/**
 * PrimitiveShader class
 * @depends shader.js
 */
function PrimitiveShader() {
    Shader.call(this,
        // inline-vertex shader:
        [
            'attribute vec2 aVertexPosition;',

            'uniform mat4 uMatrix;',
            'uniform mat4 uTransform;',
            'uniform float uPointSize;',

            'void main(void) {',
            '   gl_PointSize = uPointSize;',
            '   gl_Position = uMatrix * uTransform * vec4(aVertexPosition, 0.0, 1.0);',
            '}'
        ].join('\n'),
        // inline-fragment shader
        [
            'precision mediump float;',

            'uniform vec4 uColor;',

            'void main(void) {',
            '   gl_FragColor = uColor;',
            '}'
        ].join('\n'),
        // uniforms:
        {
            uMatrix: {type: 'mat4', value: mat4.create()},
            uTransform: {type: 'mat4', value: mat4.create()},
            uColor: [0.0, 0.0, 0.0, 1.0],
            uPointSize: 2
        },
        // attributes:
        {
            aVertexPosition: 0
        });
}

inheritsFrom(PrimitiveShader, Shader);
```

```
/**
 * TextureShader class
 * @depends shader.js
 */
function TextureShader() {
    Shader.call(this,
        // inline-vertex shader:
        [
            'precision mediump float;',

            'attribute vec2 aVertexPosition;',
            'attribute vec2 aTextureCoord;',

            'uniform mat4 uMatrix;',
            'uniform mat4 uTransform;',

            'varying vec2 vTextureCoord;',

            'void main(void){',
            '   gl_Position = uMatrix * uTransform * vec4(aVertexPosition, 0.0, 1.0);',
            '   vTextureCoord = aTextureCoord;',
            '}'
        ].join('\n'),
```

```
        // inline-fragment shader
        [
            'precision mediump float;',

            'varying vec2 vTextureCoord;',
            'varying vec4 vColor;',

            'uniform sampler2D uSampler;',
            'uniform vec4 uColor;',

            'void main(void){',
            '   gl_FragColor = texture2D(uSampler, vTextureCoord) * uColor;',
            '}'
        ].join('\n'),
        // uniforms:
        {
            uSampler: {type: 'tex', value: 0},
            uMatrix: {type: 'mat4', value: mat4.create()},
            uTransform: {type: 'mat4', value: mat4.create()},
            uColor: [1.0, 1.0, 1.0, 1.0]
        },
        // attributes:
        {
            aVertexPosition:    0,
            aTextureCoord:      0
        });
}

inheritsFrom(TextureShader, Shader);
```

Anexo 3 – MySQL Connector

```
<?php

class MySQLConnector {
  private $db_conn = null;
  private $initialized = false;
  private $db_servername = "";
  private $db_username = "";
  private $db_password = "";
  private $db_name = "";

  private function initialize() {
    $this->db_servername = DB_SERVERNAME;
    $this->db_username = DB_USERNAME;
    $this->db_password = DB_PASSWORD;
    $this->db_name = DB_NAME;

    $this->initialized = true;
  }

  public function inserted_id() {
    return mysqli_insert_id($this->db_conn);
  }

  public function open_db() {
    if($this->initialized == false) {
      $this->initialize();
    }

    $this->db_conn = new mysqli($this->db_servername, $this->db_username, $this-
>db_password);

    // check connection :
    if($this->db_conn->connect_error) {
      error_log(mysqli_connect_error());
      return false;
    }

    mysqli_set_charset($this->db_conn, "utf8");

    if(!mysqli_select_db($this->db_conn, $this->db_name)) {
      error_log("could not select database (DB_NAME): " . mysqli_error($this->db_conn));
      return false;
    }

    return true;
  }

  public function select_db($target_db_name) {
    if(!mysqli_select_db($this->db_conn, $target_db_name)) {
      error_log("could not select database (DB_NAME): " . mysqli_error($this->db_conn));
      return false;
    }

    return true;
  }

  public function close_db() {
    if($this->db_conn != null && !$this->db_conn->connect_errno)  {
      $this->db_conn->close();
      $this->db_conn = null;

      return true;
    }

    return false;
  }

  public function escape_string($value) {
    return trim(mysqli_real_escape_string($this->db_conn, $value));
  }
```

```
  public function execute_nonquery($sql) {
    if ($this->db_conn->query($sql) === TRUE) {
      return true;
    } else {
      $error = mysqli_error($this->db_conn);
      error_log("Error in: $sql \nError Message: $error");
      return false;
    }
  }

  public function execute_query($sql) {
    return $this->db_conn->query($sql);
  }

  public function get_error() {
    return mysqli_errno($this->db_conn) . "_:_" . mysqli_error($this->db_conn);
  }
}

?>
```