

UNIVERSIDADE REGIONAL DE BLUMENAU
CENTRO DE CIÊNCIAS EXATAS E NATURAIS
CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

TOWELJS: ENGINE 3D EM JAVASCRIPT USANDO
ARQUITETURA BASEADA EM COMPONENTES

GABRIEL ZANLUCA

BLUMENAU
2018

GABRIEL ZANLUCA

**TOWELJS: ENGINE 3D EM JAVASCRIPT USANDO
ARQUITETURA BASEADA EM COMPONENTES**

Trabalho de Conclusão de Curso apresentado ao curso de graduação em Ciência da Computação do Centro de Ciências Exatas e Naturais da Universidade Regional de Blumenau como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Prof. Dalton Solano dos Reis, M. Sc. - Orientador

**BLUMENAU
2018**

TOWELJS: ENGINE 3D EM JAVASCRIPT USANDO ARQUITETURA BASEADA EM COMPONENTES

Por

GABRIEL ZANLUCA

Trabalho de Conclusão de Curso aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II pela banca examinadora formada por:

Presidente: _____
Prof. Dalton Solano dos Reis, M. Sc. – Orientador, FURB

Membro: _____
Prof(a). Luciana Pereira de Araújo, Mestre – FURB

Membro: _____
Prof. Aurélio Faustino Hoppe, Mestre – FURB

Blumenau, 13 de julho de 2018

Dedico este trabalho a todos os meus familiares e amigos especialmente aos que sempre me apoiaram, também dedico a todos os educadores que ajudaram na minha formação

AGRADECIMENTOS

Primeiramente a Deus por ter me dado força e vontade de chegar até aqui.

À minha família por sempre estar do meu lado e me apoiado, especialmente aos meus pais e a minha irmã.

Aos meus amigos novos e antigos, especialmente ao João Paulo Serodio Gonçalves por ter me ajudado nos testes desse trabalho.

Ao meu orientador e amigo Dalton Solano dos Reis não apenas por ter acreditado na conclusão desse trabalho, mas também por ter acreditado naquele acadêmico recém-saído do segundo semestre para ser seu bolsista.

Também quero agradecer a todos os professores que fizeram parte da minha graduação.

O verdadeiro analfabeto é aquele que sabe ler,
mas não lê.

Mario Quitana

RESUMO

Esse trabalho apresenta a implementação de um motor de jogos utilizando JavaScript e WebGL, tendo como principal objetivo facilitar a implementação e aumentar o nível de abstração para aplicações desenvolvidas utilizando essas duas ferramentas. O motor por sua vez disponibiliza a criação de objetos gráficos (cubos e esferas) e luzes permitido juntar tudo numa cena, permite também a criação de dois tipos diferentes de câmera sintética (perspectiva e ortogonal). Tudo isso feito utilizando uma arquitetura baseada em componentes, que ajudou na organização e facilitará futuras expansões do código. Implementou-se junto comportamentos comuns em motores de jogos com o intuito de isola-los para que seja fácil de reaproveita-los sem passar por um grande trabalho a cada nova aplicação. O uso foi validado executando exemplos de código nos principais navegadores do mercado bem como foram feitos teste de performance comparando-o com a Three.js. Nesses testes observou-se que o modo como a arquitetura baseada em componentes foi implementado precisa ser revisto buscando uma melhora na performance.

Palavras-chave: Motor de jogos. Three.js. Desenvolvimento de aplicações gráficas.

ABSTRACT

This work presents the implementation of a game engine using JavaScript and WebGL, with the main objective of facilitating implementation and increasing the level of abstraction for applications developed using these two tools. The engine in turn offers the creation of graphic objects (cubes and spheres) and lights allowed to add everything in a scene, also allows the creation of two different types of synthetic camera (perspective and orthogonal). All of this is done using a component-based architecture that has helped in the organization and will facilitate future code expansions. Common behaviors in game engines were implemented with the aim of isolating them so that it is easy to reuse them without going through a great work with each new application. The use was validated by executing code examples in the main browsers of the market as well as were tested performance by compared it to Three.js. In these tests it was observed that the way the component-based architecture was implemented needs to be revised seeking an improvement in performance.

Key-words: Game engine. Three.js. Development of graphics applications.

LISTA DE FIGURAS

Figura 1 - Exemplo do uso Three.js (geometries)	21
Figura 2 - Uso do WebGLStudio.js	22
Figura 3 - Informações na aba Profiling	23
Figura 4 - Arquitetura VisEdu-Engine	24
Figura 5 - Utilização do motor de jogos VisEdu-Engine	25
Figura 6 - Arquitetura do motor de jogos TowelJS	27
Figura 7 - Diagrama de classe do pacote component	29
Figura 8 - Diagrama de classe do pacote game	30
Figura 9 - Diagrama de classe do pacote gameObject	31
Figura 10 - Diagrama do pacote geometric	32
Figura 11 - Diagrama de classe do pacote light	33
Figura 12 - Diagrama de classe do pacote system	34
Figura 13 - Diagrama de classe do pacote utils	34
Figura 14 - Diagrama de componentes utilizados pela classe GameObject	35
Figura 15 - Diagrama de sequência do loop de renderização do motor	35
Figura 16 - Visualização da cena implementada utilizando o motor desenvolvido	48
Figura 17 - Comparação da análise do cenário 6 utilizando a ferramenta DevTools	49
Figura 18 - Gráfico de consumo de memória	50
Figura 19 - Gráfico de comparação do tempo da criação das esferas em cada cenário	51
Figura 20 - Visualização do cenário 1 implementado com o motor desenvolvido	63
Figura 21 - Visualização do cenário 2 implementado com o motor desenvolvido	63
Figura 22 - Visualização do cenário 3 implementado com o motor desenvolvido	64
Figura 23 - Visualização do cenário 4 implementado com o motor desenvolvido	64
Figura 24 - Visualização do cenário 5 implementado com o motor desenvolvido	65
Figura 25 - Visualização do cenário 6 implementado com o motor desenvolvido	65
Figura 26 - Visualização do cenário 1 implementado com a Three.js	72
Figura 27 - Visualização do cenário 2 implementado com a Three.js	72
Figura 28 - Visualização do cenário 3 implementado com a Three.js	73
Figura 29 - Visualização do cenário 4 implementado com a Three.js	73
Figura 30 - Visualização do cenário 5 implementado com a Three.js	74
Figura 31 - Visualização do cenário 6 implementado com a Three.js	74

Figura 32 - Primeiro enunciado.....	75
Figura 33 - Primeira parte da segunda atividade	76
Figura 34 - Segunda parte da segunda atividade	76
Figura 35 - Enunciado da terceira atividade	77
Figura 36 - Instruções para obter-se o motor de jogos	78
Figura 37 - Primeira parte da explicação da criação de uma cena	78
Figura 38 - Segunda parte da explicação da criação de uma cena	79

LISTA DE QUADROS

Quadro 1 - Exemplo de código vertex shader	20
Quadro 2 - Exemplo de um fragment shader.....	20
Quadro 3 - Construtor do <code>GameObject</code>	37
Quadro 4 - Código da Classe <code>CubeGameObject.js</code>	38
Quadro 5 - Construtor padrão da luz	38
Quadro 6 - Construtor da <code>DirectionalLight</code>	38
Quadro 7 - Construtor do <code>PointLight</code>	39
Quadro 8 - Construtor do <code>Spotlight</code>	39
Quadro 9 - Código que adiciona um <code>GameObject</code> a cena	39
Quadro 10 - Código do construtor da câmera em perspectiva	40
Quadro 11 - Código do construtor da câmera ortogonal	40
Quadro 12 - Código da classe <code>RenderSystem</code>	41
Quadro 13 - Código da classe do componente padrão	42
Quadro 14 - Código da classe <code>RenderComponent</code>	43
Quadro 15 - Código do componente de translação.....	44
Quadro 16 - Construtor da classe <code>Game</code>	45
Quadro 17 - Código presente na classe <code>Game</code>	46
Quadro 18 - Código utilizando o motor de jogos	47
Quadro 19 - Comparação dos trabalhos correlatos com o trabalho desenvolvido	52
Quadro 20 - Cenário 1 implementado com o motor desenvolvido (50 esferas).....	57
Quadro 21 - Cenário 2 implementado com o motor desenvolvido (100 esferas).....	58
Quadro 22 - Cenário 3 implementado com o motor desenvolvido (150 esferas).....	59
Quadro 23 - Cenário 4 implementado com o motor desenvolvido (200 esferas).....	60
Quadro 24 - Cenário 5 implementado com o motor desenvolvido (250 esferas).....	61
Quadro 25 - Cenário 6 implementado com o motor desenvolvido (300 esferas).....	62
Quadro 26 - Cenário 1 implementado com a <code>Three.js</code> desenvolvido (50 esferas).....	66
Quadro 27 - Cenário 2 implementado com a <code>Three.js</code> desenvolvido (100 esferas).....	67
Quadro 28 - Cenário 3 implementado com a <code>Three.js</code> desenvolvido (150 esferas).....	68
Quadro 29 - Cenário 4 implementado com a <code>Three.js</code> desenvolvido (200 esferas).....	69
Quadro 30 - Cenário 5 implementado com a <code>Three.js</code> desenvolvido (250 esferas).....	70

Quadro 31 - Cenário 2 implementado com a Three.js desenvolvido (300 esferas)..... 71

LISTA DE TABELAS

Tabela 1 - Quantidade de quadro por segundo em cada cenário	49
---	----

LISTA DE ABREVIATURAS E SIGLAS

3D – 3 Dimensões

API – Application Programming Interface

FPS – Frame Per Second

GPU – Graphics Processing Unit

HTML – HyperText Markup Language

IA – Inteligência Artificial

RF – Requisito Funcional

RNF – Requisito Não Funcional

UML – Unified Modeling Language

WebGL – Web Graphics Library

SUMÁRIO

1 INTRODUÇÃO.....	16
1.1 OBJETIVOS.....	17
1.2 ESTRUTURA.....	17
2 FUNDAMENTAÇÃO TEÓRICA	18
2.1 MOTOR DE JOGOS.....	18
2.2 ARQUITETURA BASEADA EM COMPONENTES	18
2.3 WEBGL.....	19
2.4 TRABALHO CORRELATOS	20
2.4.1 Three.JS.....	20
2.4.2 WebGLStudio.js.....	21
2.4.3 VisEdu-Engine	23
3 DESENVOLVIMENTO DO MOTOR DE JOGOS	26
3.1 REQUISITOS.....	26
3.2 ESPECIFICAÇÃO	26
3.2.1 Arquitetura do motor de jogos	26
3.2.2 Diagramas de classe do motor de jogos	27
3.2.3 Diagrama de componentes do gameObject	34
3.2.4 Diagrama de sequência	35
3.3 IMPLEMENTAÇÃO	36
3.3.1 Técnicas e ferramentas utilizadas.....	36
3.3.2 O motor de jogos.....	37
3.3.3 Operacionalidade da implementação	46
3.4 ANÁLISE DOS RESULTADOS	48
3.4.1 Testes de desempenho.....	48
3.4.2 Análise da usabilidade.....	51
3.4.3 Comparativo entre os trabalhos correlatos	52
4 CONCLUSÕES.....	53
4.1 EXTENSÕES	54
REFERÊNCIAS	55
APÊNDICE A – CENÁRIOS IMPLEMENTADOS USANDO O MOTOR DESENVOLVIDO.....	57

APÊNDICE B – CENÁRIOS IMPLEMENTADOS USANDO O MOTOR DESENVOLVIDO	66
APÊNDICE C – ENUNCIADO DAS ATIVIDADES.....	75
APÊNDICE D – TUTORIAL CRIADO PARA EXPLICAR A UTILIZAÇÃO DO MOTOR DE JOGOS DESENVOLVIDO	78

1 INTRODUÇÃO

De acordo com a pesquisa realizada pela Newzoo (2017), em 2017 a previsão é que a indústria de jogos eletrônicos movimentará 108,9 bilhões de dólares. Isso representa um ganho de 7,8 bilhões comparado com o ano de 2016. Dessa forma, a indústria de jogos eletrônicos se demonstra como algo atrativo para se investir em desenvolvimento visto que as projeções, segundo a mesma pesquisa, para os próximos anos também indicam aumento.

Para que todo esse crescimento se concretize a busca e utilização de ferramentas adequadas para auxiliar no desenvolvimento se torna essencial, dentre elas pode-se mencionar os motores de jogos. Com a utilização de motores de jogos ocorre-se um ganho de tempo e um aumento no nível de abstração. O ganho de tempo acontece pelo fato de rotinas, comportamentos e algoritmos gerais voltados para o desenvolvimento de jogos já estarem implementados e testados, assim não necessitando refazê-los a cada nova aplicação. O maior grau de abstração ocorre porque vários comportamentos são encapsulados, a exemplo tem-se desenho de objetos gráficos, transformações geométricas, tratamento para entrada de periféricos, entre outros. Em todos os casos citados anteriormente o programador não necessitaria conhecer qual API gráfica está sendo usada para desenhar os objetos, como funciona as matrizes de transformação geométricas ou a forma como o clique do mouse é capturado e tratado. Com essas facilidades citadas anteriormente a tarefa do programador pode se focar na criação da história e regras da aplicação deixando preocupação gerais e iguais em todas as aplicações para o motor de jogos escolhido resolver.

Ainda pensando-se na parte do desenvolvimento, uma das opções a se levar em consideração é o uso de uma arquitetura baseada em componentes, que se mostra interessante pelo fato de ser “[...] caracterizado pela composição de partes já existentes, ou pela composição de partes desenvolvidas independentemente e que são integradas para atingir o objetivo final [...]” (FEIJÓ, 2007, p. 17). Os benefícios de uso de componentes se dão pelo fato de questões relacionadas a desenho de objetos poderem ser criados separadamente, poder criar um comportamento que mais tarde será usado por uma ou mais personagens e em ambos os casos se tem a opção de remover quando necessário. Por exemplo, um comportamento de pulo pode ser implementado como um componente de pulo e sempre adicionando quando os personagens necessitarem dele. Dessa forma o componente pode ser visto como um bloco de montar que pode ser encaixado de diferentes formas.

Visto todos os argumentos citados anteriormente, este trabalho apresenta um motor de jogos que auxilie na construção de jogos em 3D utilizando a linguagem JavaScripts e arquitetura baseada em componentes.

1.1 OBJETIVOS

O objetivo é desenvolver um motor de jogos 3D utilizando arquitetura baseada em componentes para facilitar o desenvolvimento de jogos em JavaScript.

Os objetivos específicos são:

- a) desenvolver componentes dedicados para contabilizar o *frame per second* (FPS) na cena e memória consumida pela aplicação;
- b) analisar a performance do motor desenvolvido comparado com a biblioteca Three.js.

1.2 ESTRUTURA

O trabalho está dividido em quatro capítulos. O segundo capítulo engloba a fundamentação teórica para o entendimento do mesmo, bem como a apresentação de trabalhos correlatos.

O terceiro capítulo apresenta a descrição de como ocorreu o desenvolvimento do motor de jogos, os diagramas de classes, diagrama de sequência e as explicações necessárias sobre a implementação. Ainda nesse capítulo são tratados os resultados obtidos bem como os testes executados.

Por fim, no capítulo quatro são apresentadas as conclusões referentes ao trabalho além de sugestões e melhorias para a continuação do mesmo.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo aborda os conceitos relevantes para o entendimento do trabalho. Na seção 2.1 é discutido os conceitos de motor de jogos. Na seção 2.2 apresenta-se os conceitos e formas de utilização de arquitetura baseada em componentes. A seção 2.3 mostra o funcionamento do WebGL. Por fim, a seção 2.3 traz os trabalhos correlatos.

2.1 MOTOR DE JOGOS

Com o passar do tempo o termo motor de jogos foi mudando o seu significado para abranger mais funcionalidades deixando de ser algo concentrado apenas na parte gráfica de uma aplicação. Segundo Eberly (2006, p. 2, tradução nossa), “motor de jogo passou a significar uma coleção de motores - para gráficos, física, IA, redes, *scripts* [...]”. Dentre exemplos populares no mercado que atende essa descrição pode-se citar a Unity (UNITY, 2017), Unreal Engine (UNREAL ENGINE, 2017) e Cryengine (CRYENGINE, 2017) entre outras. Dos três exemplos citados, eles não apenas fornecem auxílio na parte gráfica de aplicação, mas como já citado, ajudam em outras partes da aplicação como: simulação de física, criação de IA para as personagens não jogáveis e com tudo que seja necessário atualmente para se construir uma aplicação gráfica. Assim, um motor de jogos passa a ser algo bem maior e mais complexo atualmente tentando alcançar todas essas características.

A principal ideia de um motor de jogos “[...] é permitir que os recursos comuns a quase todos os jogos sejam reutilizados para cada novo jogo criado. Neste caso, a cada novo jogo, se implementa apenas seus requisitos particulares” (PESSOA, 2001, p. 4). Dessa forma ganha-se em tempo, no desenvolvimento, por não necessitar a todo momento construir novamente as estruturas já usadas anteriormente, assim tendo a opção de se manter o foco na jogabilidade.

Logo, juntando os fatos que foram apresentados sobre motores de jogos, mostra-se que seus recursos vão além do auxílio no desenho de objetos e implementações de algoritmos de computação gráfica. Demonstrem-se como ferramentas importante na construção de jogos eletrônicos, pois não ficam restritas ao auxílio de uma única área.

2.2 ARQUITETURA BASEADA EM COMPONENTES

A arquitetura baseada em componentes baseia-se no desenvolvimento usando componentes. Segundo Sametinger (1997, p.2, tradução nossa) “componentes são artefatos que claramente são identificados nos sistemas de software. Eles têm uma interface, encapsulamento interno detalhado e são documentados separadamente.”.

Ao adotar-se o uso de componentes combinando os fabricados com os adquiridos ocorre um aumento na qualidade e agiliza o desenvolvimento levando assim a uma entrega mais rápida (SZYPERSKI, 2002). Assim, segundo Feijó (2001 apud CHEESMAN, 2007), o desenvolvimento de sistemas baseado em componentes adere ao princípio da divisão e conquista, diminuindo a complexidade, pois um problema é dividido em partes menores, resolvendo estas partes menores e construindo soluções mais elaboradas a partir de soluções mais simples.

Um ponto que deve ficar bem claro é

[...] a diferença entre o desenvolvimento de componentes e desenvolvimento com componentes. No primeiro caso, os componentes são especificados e implementados, existindo a preocupação em gerar a documentação em projetá-los de forma a serem reusados. No segundo, os componentes já produzidos são utilizados para se conceber um novo sistema (FEIJÓ, 2007).

Dessa forma tanto o uso de componentes quanto o desenvolvimento de componentes têm como preocupação o reaproveitamento de código e ganho de tempo. Assim um componente deve ser projetado com foco em uma tarefa específica, tendo assim uma maior chance de ser reutilizado novamente.

2.3 WEBGL

O WebGL pode ser definido como “[...] um padrão web *cross-platform* e livre de royalties para uma API de gráficos 3D de baixo nível com base no OpenGL ES, desenvolvido em ECMAScript por meio do elemento Canvas HTML5” (KHRONOS, 2018, p. 1, tradução nossa). O WebGL 1.0 baseia-se na versão 2.0 do OpenGL ES, “trazendo 3D sem *plug-in* para web implementados diretamente no navegador.” (KHRONOS, tradução nossa, 2018, p. 1). Estando presente nos maiores navegadores do mercado.

Para programar-se em WebGL faz-se uso da linguagem GL Shader Language (GLSL), que fará com que o código seja executado pela GPU do computador. Para que isso aconteça usa duas funções: `vertex shader` e `fragment shader`.

“O trabalho do vertex shader é calcular as posições dos vértices.” (TAVARES, 2018a). Essa função é escrita em GLSL e chamada uma vez para cada vértice. Dentro dessa função pode-se fazer alguns cálculos para no final atribuir um valor a variável especial `gl_Position`, sendo esse valor que o vértice terá no espaço gráfico. (TAVARES, 2018b). No Quadro 1 pode-se observar um exemplo de código de vertex shader.

Quadro 1 - Exemplo de código vertex shader

```
attribute vec4 aVertexPosition;
uniform mat4 uModelViewMatrix;

void main() {
    gl_Position = uModelViewMatrix * aVertexPosition
}
```

Fonte: elaborado pelo autor.

Para o `fragment shader` o trabalho “é calcular uma cor para cada pixel da primitiva atualmente sendo desenhada.” (TAVARES, 2018a). Semelhante ao `vertex shader` essa função também deve ser escrita em GLSL, com a diferença que agora o nome da variável especial é `gl_FragColor`. No Quadro 2 é exibido um exemplo de código de `fragment shader` utilizando um atributo como `varying`, ele serve para compartilhar valores entre o `vertex shader` e o `fragment shader`. Nesse exemplo buscou-se colocar uma cor diferente para vértices utilizando a variável `v_color` compartilhada com o `vertex shader`.

Quadro 2 - Exemplo de um fragment shader

```
varying vec4 v_color;

void main() {
    gl_FragColor = v_color;
}
```

Fonte: elaborado pelo autor.

2.4 TRABALHO CORRELATOS

Nesta seção são apresentados trabalhos com características semelhantes aos principais objetivos do estudo desenvolvido. O primeiro é a biblioteca Three.js (THREE.JS, 2018), segundo é o editor WebGLStudio.js (WEBGLSTUDIOJS, 2017) e o terceiro é o motor de jogos VisEdu-Engine (HARBS, 2013).

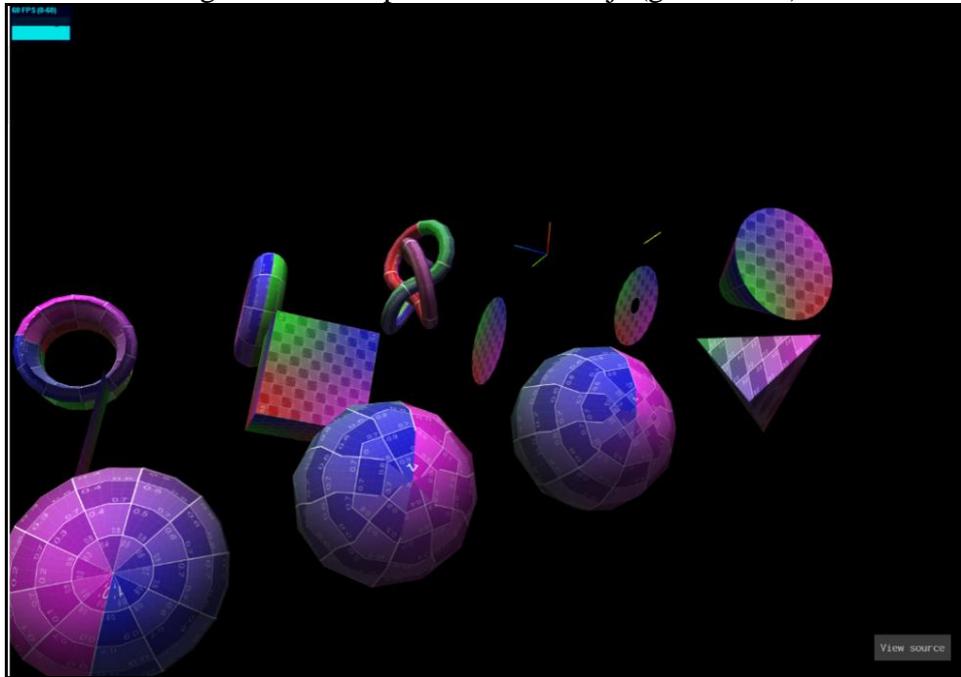
2.4.1 Three.JS

O objetivo da Three.js é “[...] criar uma biblioteca 3D leve e fácil de usar. A biblioteca fornece renderizadores <canvas>, <svg>, CSS3D e WebGL” (THREE.JS, 2017a, tradução nossa, p. 1). Desenvolvida em JavaScript e podendo ser vista como uma camada acima se comparado com uso do WebGL somente, porque permite um nível maior de abstração na questão gráfica do desenvolvimento da aplicação. Referente as questões para utilização, ela é gratuita e possui código fonte aberto disponível para alteração.

Dentre as características descritas na documentação da Three.js (2017d) pode-se listar a presença de formas básicas já definidas tais como: cubos, quadrados, círculos, cilindros, esferas, linhas, etc. Também já conta com elementos de luz, diferentes tipos de materiais além de permitir a adição de textura nos objetos e permite o carregamento de objetos externos

produzidos em programas de terceiros. Na Figura 1 tem-se um exemplo do uso da biblioteca com algumas das formas geométricas disponíveis e ainda com a utilização de iluminação.

Figura 1 - Exemplo do uso Three.js (geometries)



Fonte: Three.js (2017b).

A biblioteca conta também com recursos para ajudar na criação de animação de objetos. Segundo Three.js (2017c, tradução nossa, p. 1), “[...] pode-se animar várias propriedades dos modelos: *bones* de um modelo *skinned and rigged*, *morph targets*, propriedades diferentes do material (cores, opacidade, valores lógicos), visibilidade e transformações”. Além dessa facilidade, na hora de criar animações, questões mais simples como transformações geométricas também contam com algum auxílio.

O objetivo principal da Three.js é auxiliar com o desenho da cena. Dessa forma caso o desenvolvedor precisa de algo relacionado a simulação de física, IA para os personagens presentes na aplicação ou qualquer outra questão além do desenho da cena, precisará fazer uso de alguma biblioteca externa.

2.4.2 WebGLStudio.js

O WebGLStudio.js é um editor de gráficos 3D que pode ser acessado pelo navegador, que suporte o mesmo, para se criar os projetos. O editor possui seu código aberto e usa internamente como biblioteca gráfica a LiteScene (LITESCENE, 2017). Tanto o WebGLStudio.js quanto a LiteScene são implementados em JavaScript. As características a serem ressaltadas do WebGLStudio.js são:

- motor de gráficos 3D completo (LiteScene.js) que suporta múltiplas luzes,

shadowmaps, reflexões em tempo real, materiais personalizados, postFX, skinning, animação e muito mais;

- sistema baseado em componentes fácil de expandir (controlando a linha de renderização ou encaixar eventos para de interação);
- editor WYSIWYG fácil de usar, com todos os recursos em um só lugar (codificação, composição gráfica e linha de tempo);
- editor de gráficos para criar comportamentos interessantes ou efeitos de pós-produção;
- sistema virtual de arquivos para armazenar todos os recursos na web LiteFileSystem.js apenas arrastando-os (com cotas, usuários e pastas compartilhadas);
- fácil de exportar e compartilhar, apenas enviando um link (WEBGLSTUDIOJS, 2017, tradução nossa, p. 1).

Na Figura 2 pode-se observar o uso do WebGLStudio.js com uma cena já carregada com um objeto 3D modelado e a presença de luz em cena. A direita pode-se visualizar o grafo de cena montado de forma visual. Além disso, também há campos para configurar as propriedades do objeto selecionado.

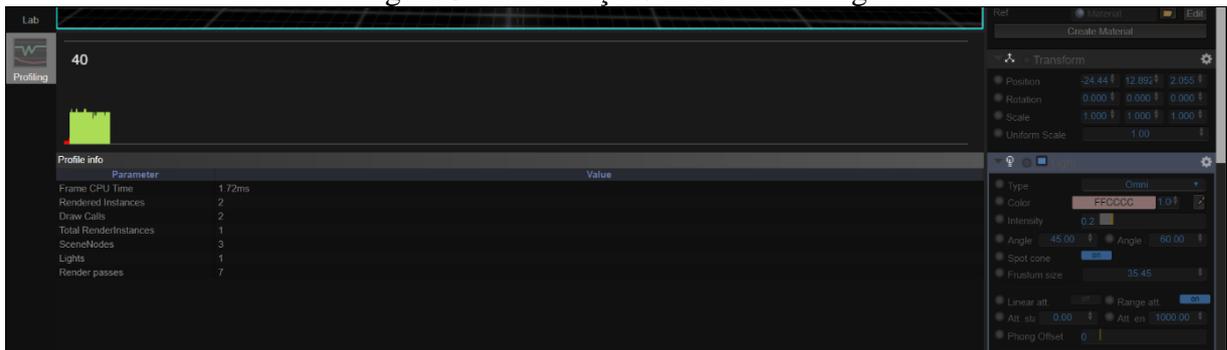
Figura 2 - Uso do WebGLStudio.js



Fonte: WebGLStudio.js (2017).

Semelhante à Three.js, o WebGLStudio.js não conta com recursos para tratar questões relacionadas a física, porém conta com uma aba no editor chamada Profiling que pode ser vista na Figura 3, na qual são exibidas algumas informações relativas a cena atual. Também conta com questões para criar as animações de personagens, podendo até se fazer uso da interface gráfica para configurar alguns dos parâmetros.

Figura 3 - Informações na aba Profiling



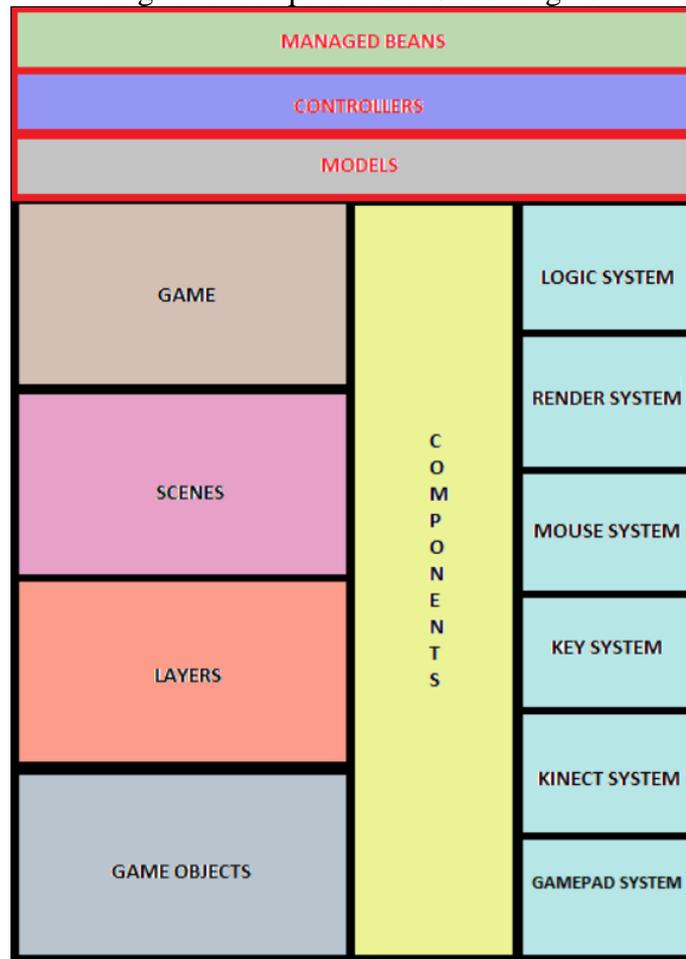
Fonte: WebGLStudio.js (2017).

2.4.3 VisEdu-Engine

O VisEdu-Engine é um motor de jogos 2D que possui as seguintes funcionalidades “[...] criação de cena, criação de camadas, gerenciamento de objetos, gerenciamento de recurso de imagens e áudio, detecção de colisão, física de corpos rígidos (utilizando a biblioteca Box2DJS) e uma arquitetura reutilizável orientada a componentes” (HARBS, 2013, p. 6). O motor é desenvolvido em JavaScript e conta com um editor que, segundo Harbs (2013), tem como objetivo facilitar o uso do motor de jogos para quem não tivesse um conhecimento tão aprofundado em programação.

Segundo Harbs (2013), a arquitetura do motor de jogos foi especificada de maneira orientada a componentes. Na Figura 4 pode-se observar o diagrama da arquitetura do VisEdu-Engine no qual a comunicação das camadas *Game*, *Scenes*, *Layers* e *Game Objects*, ocorre no sentido de cima para baixo, ou seja, o *Game* se comunica com *Scenes* que se comunica com *Layers* que por sua vez se comunica com *Game Objects*. Já todas as camadas com *system* no nome têm como objetivo criar interface com dispositivos externos e essas por sua vez se comunicam com as camadas citadas anteriormente por meio de componentes. Como também pode ser observado além do motor ter suporte a periféricos comuns como mouse e teclado, há também suporte para o Kinect e *joystick*.

Figura 4 - Arquitetura VisEdu-Engine



Fonte: Harbs (2013).

Segundo o trabalho desenvolvido por Harbs (2013) os objetos disponíveis no VisEdu-Engine são quadrados, círculos e polígonos, todos possuem seu respectivo corpo rígido também para que possa ocorrer o tratamento de colisões. Em questão de organização, a aplicação consiste em criar uma cena com várias camadas (*layers*), para que assim se tenha a impressão de algo estar no fundo ou mais à frente conseguindo assim simular o efeito de profundidade em cena. Na Figura 5 pode-se observar a criação de uma aplicação inspirada no jogo Space in Invaders utilizando o motor de jogos.

Figura 5 - Utilização do motor de jogos VisEdu-Engine



Fonte: Harbs (2013).

3 DESENVOLVIMENTO DO MOTOR DE JOGOS

Neste capítulo são demonstradas as etapas do desenvolvimento do motor de jogos denominado TowelJS. Na seção 3.1 são apresentados os requisitos funcionais e não funcionais do motor de jogos. A seção 3.2 demonstra a especificação do motor de jogos. A seção 3.3 detalha a implementação do motor de jogos, descrevendo e exibindo os trechos relevantes de códigos e ferramentas utilizadas. Por fim, a seção 3.4 apresenta os resultados dos testes.

3.1 REQUISITOS

O motor de jogos proposto nesse trabalho deverá:

- a) permitir a criação/renderização de objetos gráficos (Requisito Funcional - RF);
- b) possuir objetos gráficos como cubo e esferas já implementados (RF);
- c) permitir a criação ou extensão de objetos gráficos já criados (RF);
- d) permitir a criação de novos componentes (RF);
- e) implementar a estrutura de grafo de cena (RF);
- f) possuir componentes para analisar a performance e consumo de recursos (RF);
- g) contar com ao menos uma opção de câmera sintética (RF);
- h) possuir diferentes tipos de iluminação disponível para colocar-se em cena (RF);
- i) ser implementado utilizando JavaScript (Requisito Não Funcional - RNF);
- j) utilizar componentes para realizar a renderização dos objetos gráficos (RNF);
- k) ser implementado utilizando a arquitetura baseada em componentes (RNF).

3.2 ESPECIFICAÇÃO

Nesta seção é descrita a especificação do motor de jogos TowelJS. Inicialmente é apresentada a arquitetura do motor de jogos em seguida é apresentado os diagramas de classes, separados por pacotes, utilizando Unified Modeling Language (UML). Em seguida apresenta-se um diagrama de componentes mostrando a relação dos componentes utilizados para ter-se um objeto gráfico dentro do motor. Por fim, é apresentado um diagrama de sequência da função de desenho da cena.

3.2.1 Arquitetura do motor de jogos

Na Figura 6 pode-se observar a arquitetura do motor desenvolvido. Ela é baseada na arquitetura presente no trabalho realizado por Harbs (2013, p39) que também utilizou uma arquitetura baseada em componentes. O principal elemento dessa arquitetura é o `Game`. Ele é o

responsável por iniciar e controlar o loop principal do motor, também é nele que cada classe `System` é cadastrada para tratar um evento específico do navegador. Por cada aplicação é permitido apenas um game, isso foi garantido utilizando-se o padrão `singleton`. Por fim, o game conta ainda com uma cena e uma câmera que juntas com o game realizam o desenho de objetos na tela. A câmera é a representação de uma câmera sintética dentro do motor, podendo escolher entre duas opções: perspectiva ou ortogonal. A cena funciona como um agregador de `gameObject` e de luzes. Já os `gameObject` são as representações de objetos gráficos dentro do motor. Eles também contam com uma lista de `gameObject` com a finalidade de realizar o grafo de cena. Todos os itens citados anteriormente possuem uma lista de componentes. Os componentes para serem criados dentro do motor precisam herdar de `Component` e implementar seus métodos de acordo com suas necessidades. Por fim, o `System` representa as classes que serão responsáveis por receber os eventos externos ao motor de jogos e propagá-los pelos componentes da aplicação.

Figura 6 - Arquitetura do motor de jogos TowelJS



Fonte: elaborado pelo autor.

3.2.2 Diagramas de classe do motor de jogos

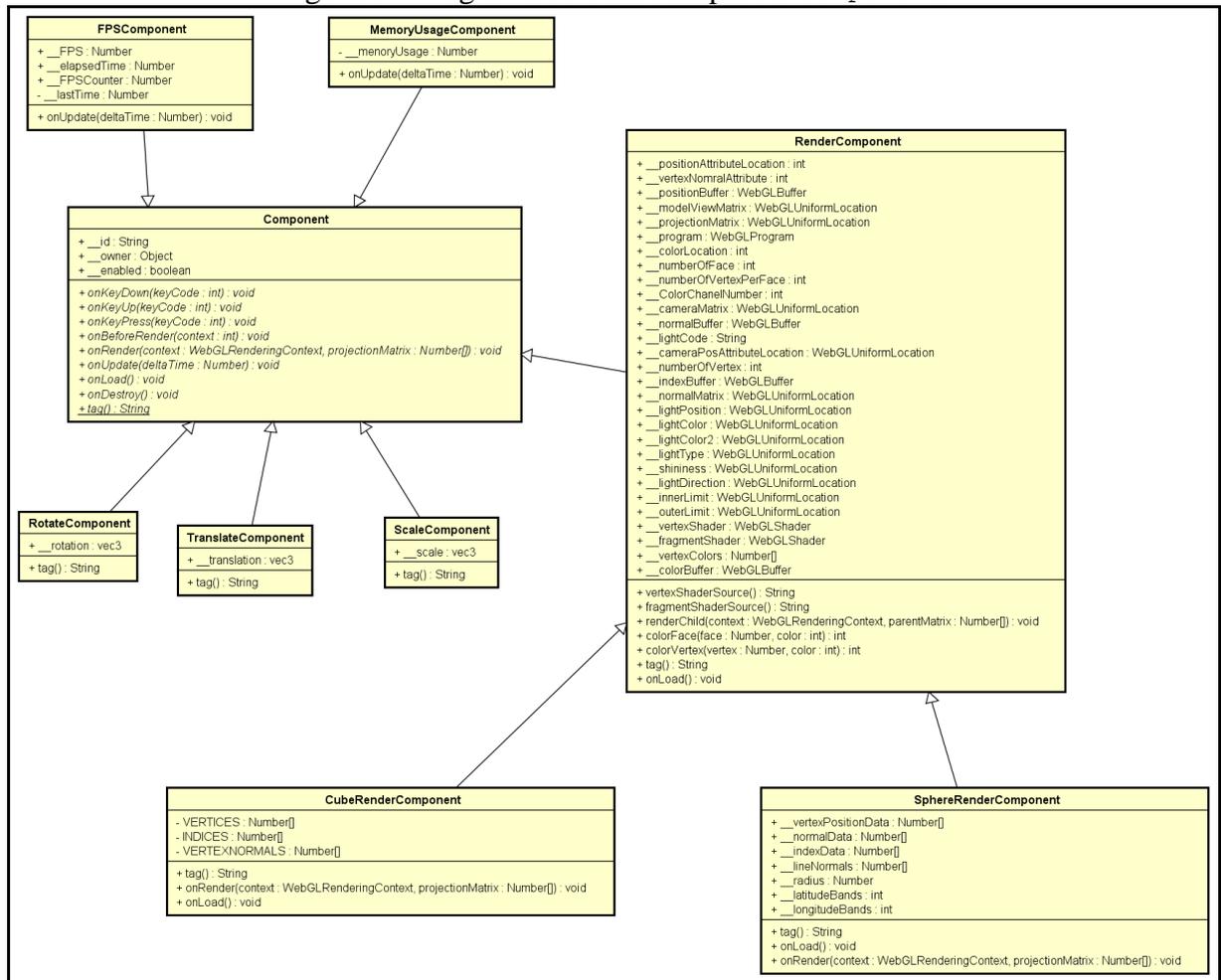
Nesta seção são apresentadas as classes que constituem o motor de jogos. Para que a visualização seja facilitada optou-se por separar por pacotes.

3.2.2.1 Pacote `component`

O pacote `component` possui a classe necessária para a criação de um novo componente dentro do motor de jogos. Nesse pacote também existem outros componentes previamente disponibilizados, como podem ser vistos na Figura 7.

A classe `Component` é a classe em que estão todos os métodos padrões de um componente, é dela que se deve herdar caso queria fazer-se um novo componente. As classes que herdam devem sobrescrever seus métodos para alcançar a funcionalidade desejada, pois por padrão não há nada implementado nos eventos. Durante o desenvolvimento notou-se a necessidade de criar-se uma classe padrão para armazenar todos os atributos e métodos referentes ao desenho dos objetos em cena. Dessa forma criou-se uma especialização do `Component`, a classe `RenderComponent`. O objetivo ao cria-la era agrupar todos os atributos e funções comuns para o desenho do `gameObject` em cena numa única classe. Dessa forma facilitou-se sua reutilização e manutenção. Como classes concretas para desenho de objetos já desenvolvidas tem-se a `CubeRenderComponent`, que se associa a um `CubeGameObject`, e `SphereRenderComponent`, que se associa a um `SphereGameObject`, para realizar o desenho na tela de um cubo e de uma esfera, respectivamente. As transformações geométricas dos objetos que compõe cena também foram realizadas por meio de classes de componentes. São elas: `RotateComponent`, `TranslateComponent` e `ScaleComponent`. Elas realizam respectivamente, rotação, translação e escala. Por fim nesse pacote encontram-se as classes de componentes dedicados exclusivamente para análise de performance, que são as classes: `FPSComponent` e `MemoryUsageComponent`. Esses componentes são responsáveis por calcular o *Frame per second* (FPS) da cena e o uso de memória da aplicação respectivamente.

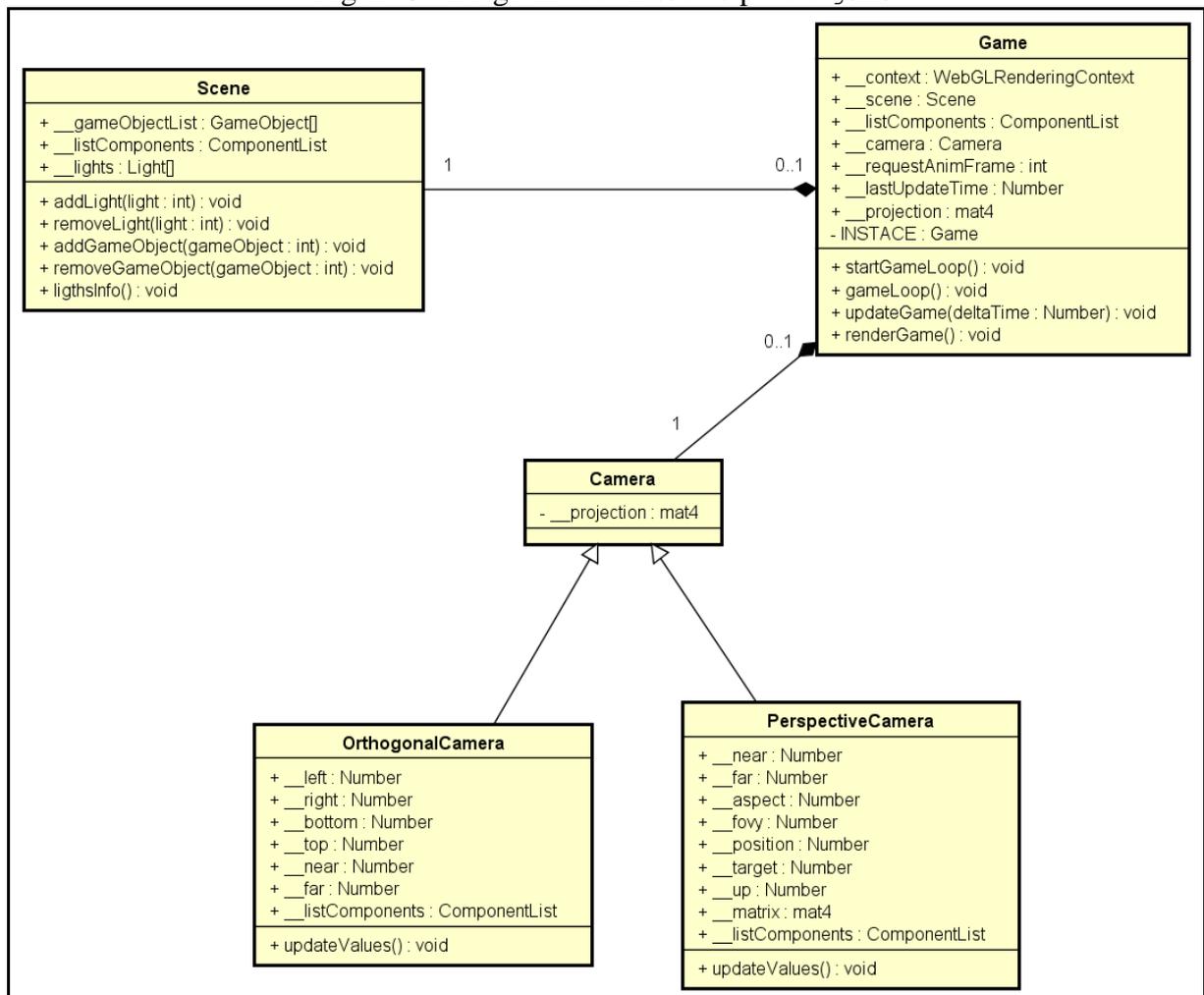
Figura 7 - Diagrama de classe do pacote component



Fonte: elaborado pelo autor.

3.2.2.2 Pacote game

O pacote denominado `game` possui as classes principais do motor de jogos para gerenciar a aplicação construída. Pode-se visualizar o diagrama de classe desse pacote na Figura 8.

Figura 8 - Diagrama de classe do pacote `game`

Fonte: elaborado pelo autor.

A classe `Scene` é responsável por armazenar os `GameObject` e as luzes presentes na cena atual. Na classe `Camera` estão localizados os métodos e atributos em comum presente nas suas classes filhas, que são as classes `OrthogonalCamera` e `PerspectiveCamera`. Essas duas classes por sua vez são responsáveis por representar uma câmera ortogonal e perspectiva dentro do motor de jogos. Ambas permitindo que sejam transladas ou rotacionadas pelo usuário, para que possa mudar sua visualização da cena. Por fim, a classe `Game` é onde a câmera se junta com a cena. Também faz parte de sua responsabilidade controlar a aplicação atual e dar início à função que desenhará a cena.

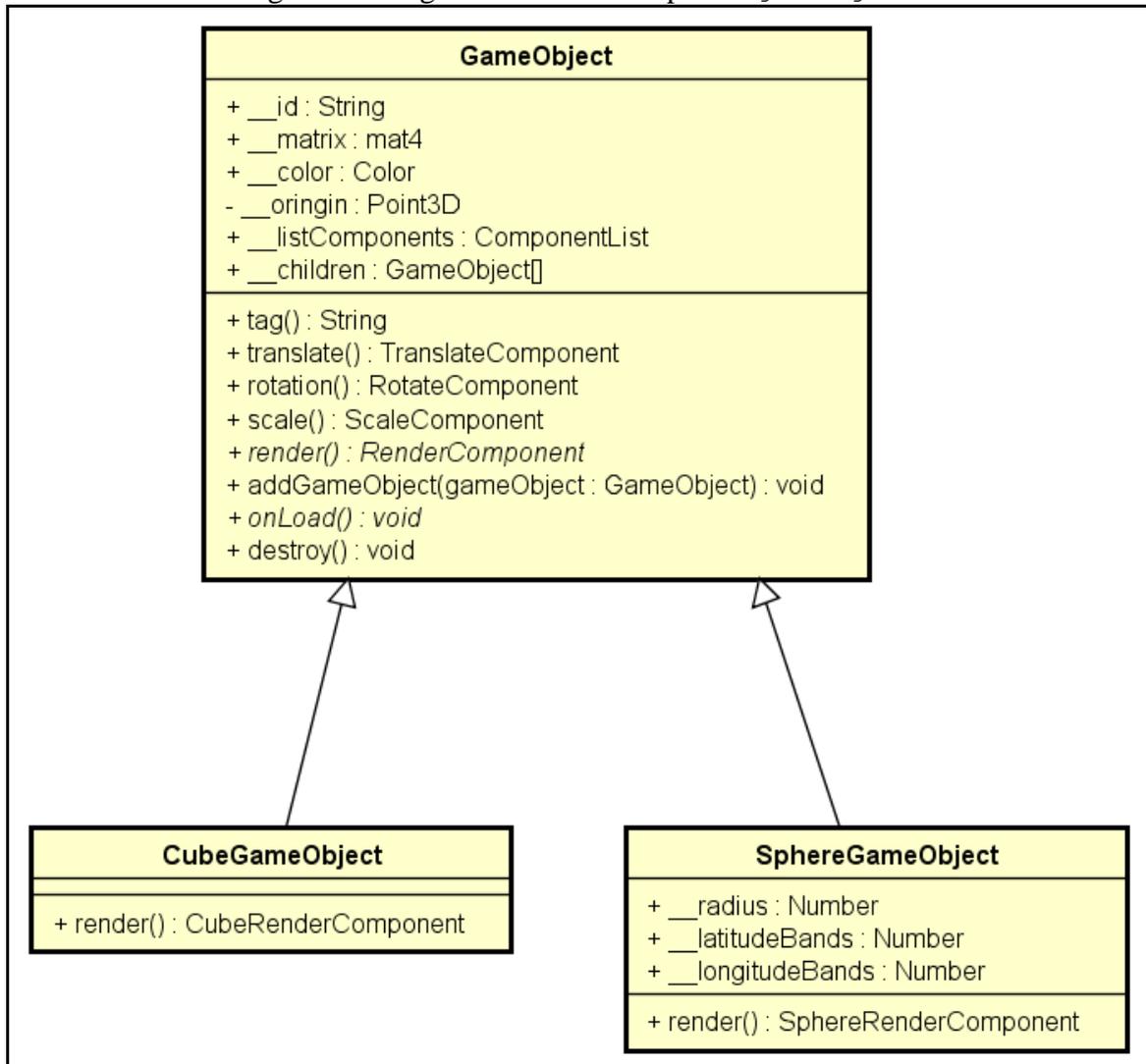
3.2.2.3 Pacote `gameObject`

No pacote `gameObject` está presente a classe padrão para definir todos os objetos dentro do motor de jogos, como pode ser observado na Figura 9.

A classe `GameObject` concentra todos os atributos e métodos comuns para definir um objeto dentro do motor. Dessa forma a classe `CubeGameObject` apenas precisa sobrescrever o

método `render` para retornar um componente responsável por desenhar um cubo. Já a classe `SphereGameObject`, precisa além de retornar seu componente responsável por desenhar uma esfera, também tem atributos a mais que serão consumidos pelo mesmo, sendo eles o raio e número de segmentos da esfera.

Figura 9 - Diagrama de classe do pacote `gameObject`

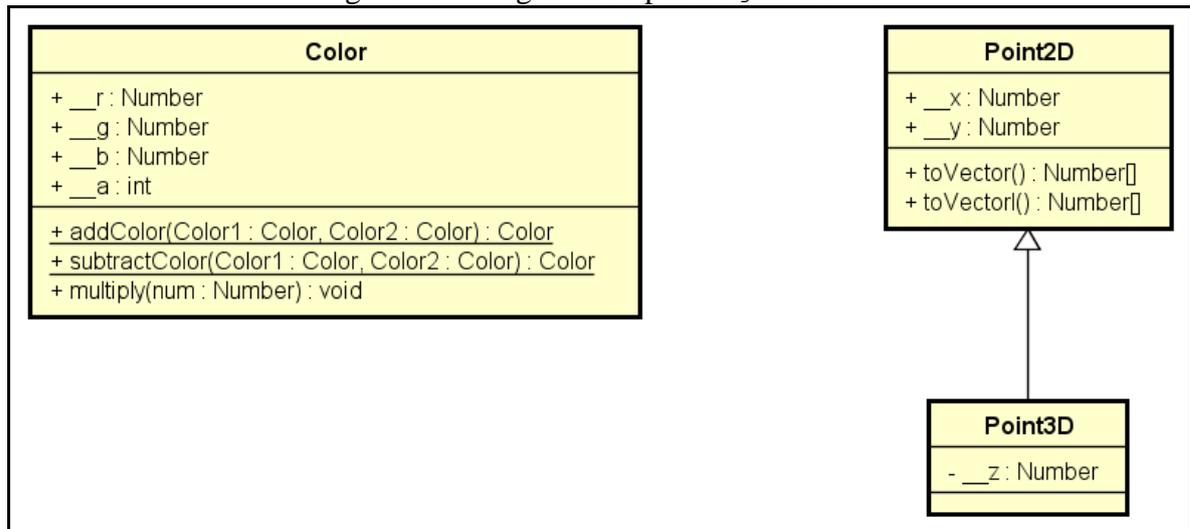


Fonte: elaborado pelo autor.

3.2.2.4 Pacote `geometric`

Nesse pacote estão localizadas as classes que representam pontos no plano e cores, como pode ser observado na Figura 10.

A classe `Color` possui os atributos e métodos para representar uma cor dentro do motor de jogos. A classe `Point2D` tem a função de representar um ponto em duas dimensões dentro do motor de jogos. Por fim, a classe `Point3D`, que estende a classe `Point2D`, tem como função representar um ponto em três dimensões dentro do motor de jogos.

Figura 10 - Diagrama do pacote `geometric`

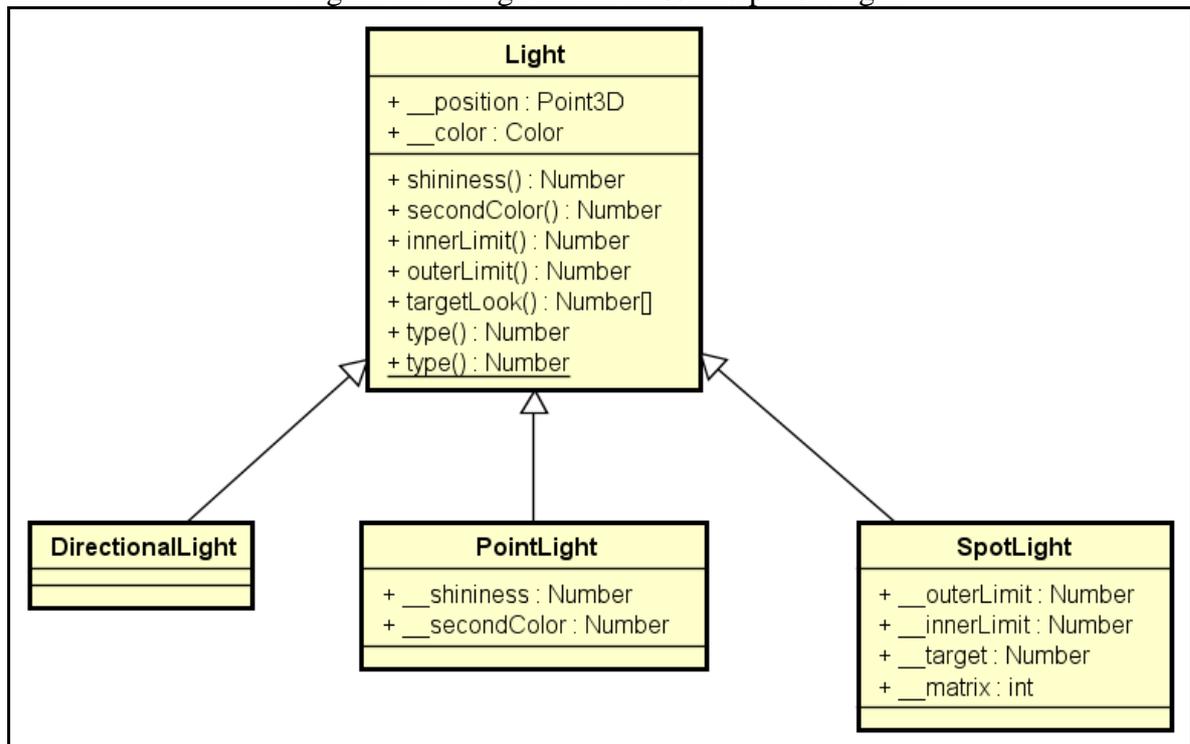
Fonte: elaborado pelo autor.

3.2.2.5 Pacote `light`

No pacote denominado `light` encontra-se a classe padrão para representação da luz dentro do motor de jogos, bem como os três tipos de luz já implementados. Na Figura 11 pode-se observar o diagrama deste pacote.

A classe `Light` é a classe padrão para definir uma luz dentro do motor de jogos. Nela estão todos os métodos que uma luz possa ter e cada classe que herde dela deverá implementar seu retorno conforme a necessidade. A classe `DirectionalLight` representa uma luz direcional dentro do motor de jogos. Já a classe `PointLight` é a representação de um ponto de luz. Por fim a classe `SpotLight` consiste na implementação de um holofote dentro do motor de jogos.

Figura 11 - Diagrama de classe do pacote light

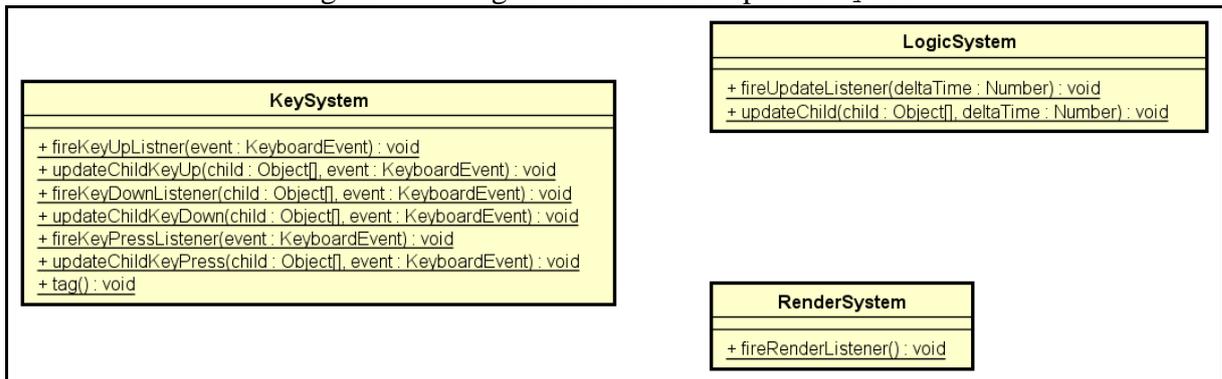


Fonte: elaborado pelo autor.

3.2.2.6 Pacote system

As classes presentes no pacote `system` são as responsáveis por receber os eventos externos ao motor de jogos e propaga-los pelos componentes. O diagrama deste pacote pode ser observado na Figura 12.

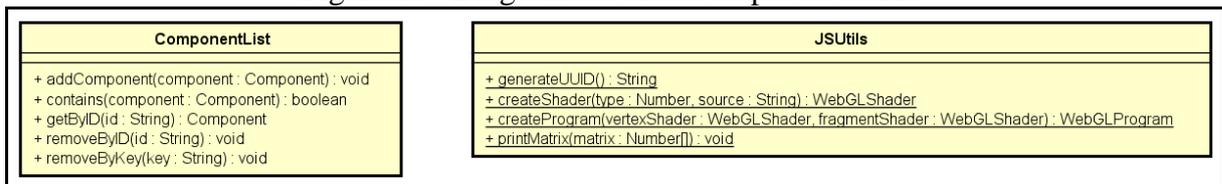
A classe `RenderSystem` é responsável por propagar o evento de renderização da cena utilizando o método `fireRenderListener` que por sua vez chama de cada componente o método `onRender(context, projectionMatrix)`. A função da classe `LogicSystem` propaga os eventos de atualização a cada *frame* utilizando o método `fireUpdateListener(deltaTime)`, que chamará de cada componente o método `onUpdate(deltaTime)`. Optou-se por separar o evento de renderização e de atualização a cada *frame* para que se pudesse ter um isolamento melhor. A classe `KeySystem` fica encarregada da propagação dos eventos ao pressionar uma tecla capturados pelo navegador.

Figura 12 - Diagrama de classe do pacote `system`

Fonte: elaborado pelo autor.

3.2.2.7 Pacote `utils`

No pacote denominado `utils` estão as classes utilitárias utilizadas para auxiliar nas operações realizadas pelo motor de jogos, como pode ser visto na Figura 13.

Figura 13 - Diagrama de classe do pacote `utils`

Fonte: elaborado pelo autor.

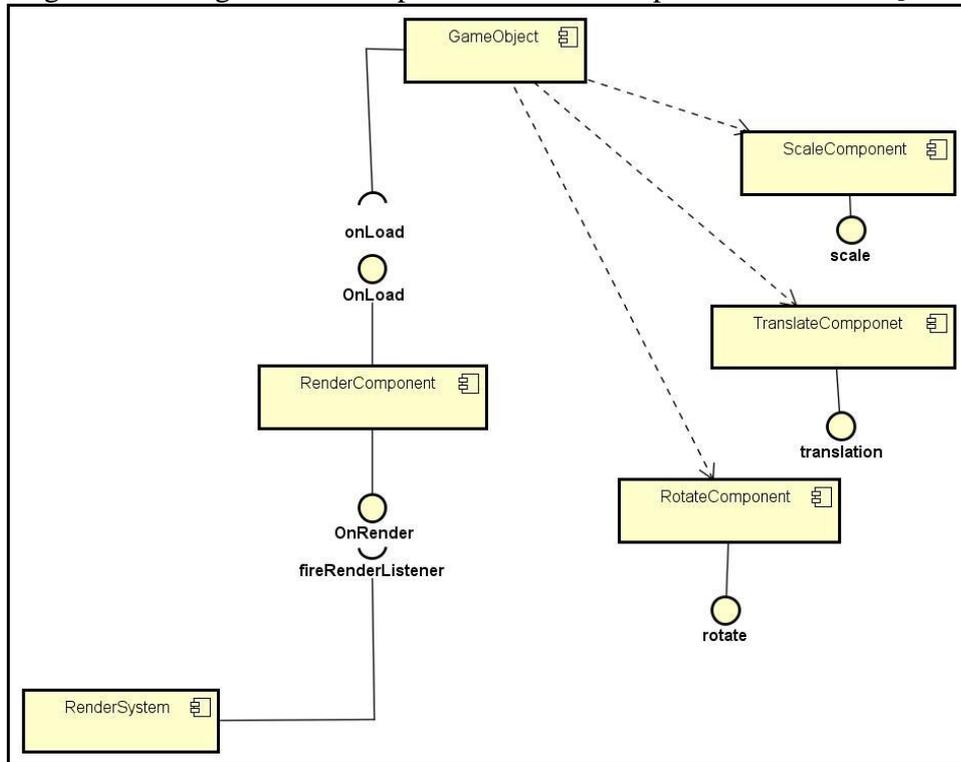
A classe `JSUtils` possui métodos para gerar o `id` de cada componente e cada `GameObject`. Também é nela que se gera o `shader` e o `program` do `WebGL` usado pela classe `RenderComponent`. Por fim conta com um método que imprime no console do navegador a matriz unidimensional de transformação do objeto em formato bidimensional. A classe `ComponentList` estende a classe padrão do JavaScript `Array` para formar uma lista de `Component` e contar com alguns métodos específicos para manipula-los.

3.2.3 Diagrama de componentes do `gameObject`

Esta seção apresenta o diagrama de componentes utilizados pela classe `GameObject` e por todas que herdam dela também. Na figura X pode-se observar o diagrama de componentes envolvidos na criação, manipulação e no desenho do `GameObject`. Por padrão todo `GameObject` já possuem os seguintes componentes na sua lista, o `TranslateComponent`, `ScaleComponent` e o `RotateComponent`. Os três são responsáveis por realizar as transformações geométricas do `GameObject`. Outro componente presente nos `GameObject` é o `RenderComponent` esse por sua vez é adicionado em cada classe específica visto que cada um deve implementar a função de desenho de forma diferente. Ao instanciar-se um `GameObject` o

método `onLoad` do seu `RenderComponent` é executado, responsável por inicializar a popular as variáveis necessárias para utilização do WebGL. Por fim, o método `fireRenderListener` é o responsável por executar o método `onRender`. Esse método é o responsável por realizar o desenho do objeto na tela.

Figura 14 - Diagrama de componentes utilizados pela classe `GameObject`

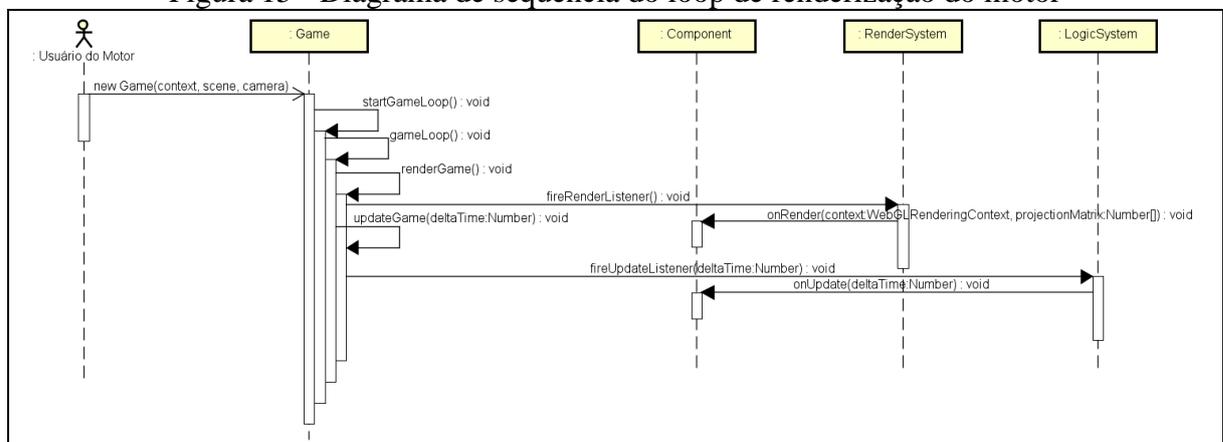


Fonte: elaborado pelo autor.

3.2.4 Diagrama de sequência

Esta seção apresenta o diagrama de sequência do loop de renderização gerenciado pelo motor. O diagrama pode ser observado na Figura 15.

Figura 15 - Diagrama de sequência do loop de renderização do motor



Fonte: elaborado pelo autor.

Quando se instancia pela primeira vez um objeto da classe `Game` o loop já tem seu início. Começando pelo método `startGameLoop()` responsável por cadastrar a método `gameLoop()` na função `window.requestAnimationFrame()` do navegador. Essa função é chamada a cada *frame* e recebe uma outra função como parâmetro que também será executada a cada *frame*. O `gameLoop()` por sua vez chama os métodos `renderGame()` e `updateGame(deltaTime)`, também fica responsável por calcular o tempo gasto para desenhar cada *frame*. O método `renderGame()` é quem dispara o evento de renderização invocando o método `fireRenderListener` da classe `RenderSystem`. Esse método por sua vez é o responsável por chamar o método `onRender` de cada componente presente. Por fim o método `updateGame(deltaTime)` invoca o método `fireUpdateListener(deltaTime)`, que será o responsável por chamar o método `onUpdate` de cada componente.

3.3 IMPLEMENTAÇÃO

A seguir são mostradas as técnicas e ferramentas utilizadas no desenvolvimento do motor, e a operacionalidade da implementação.

3.3.1 Técnicas e ferramentas utilizadas

Para o desenvolvimento do motor de jogos utilizou-se JavaScript na sua versão ECMAScript 8 e como API gráfica o WebGL 1.0. Como ambiente de desenvolvimento optou-se pelo editor de texto Visual Studio Code na sua versão 1.24.1 (MICROSOFT, 2018). Utilizou-se também o gerenciador de pacotes npm na versão 5.6.0 (NPM, 2018) para controlar as dependências do motor de jogos de código externo. Para abstrair as operações que são feitas sobre matrizes optou-se pela biblioteca `gl-matrix` (GLMATRIX, 2018).

Como a versão do ECMAScript 8 utilizada para desenvolver o motor de jogos é mais recente do que a implementada nos navegadores precisou-se utilizar um *transpiler* para que o código fosse convertido para uma versão que a maioria dos navegadores do mercado suporte. Visto esse fato o *transpiler* escolhido foi o Webpack (WEBPACK, 2018). Para os testes foram escolhidos os navegadores Google Chrome (versão 67.0.3396.99), Opera (versão 53.0.2907.99) e FireFox Developer Edition (versão 62.0b4). Todos testes foram executados num computador notebook com processador Intel(R) Core(TM) i7-6700HQ de 2.60GHz e 8 núcleos, memória de 16 GB DDR3 e uma placa de vídeo Nvidia GeForce GTX 960M.

3.3.2 O motor de jogos

Nessa seção serão explicados os principais elementos do motor de jogos. Eles serão explicados na ordem inversa da arquitetura, começando pelo `GameObject` e terminando com o `Game`.

3.3.2.1 GameObject

O `GameObject` é o responsável pela parte lógica dos objetos gráficos do motor de jogos. Nele ficam armazenadas informações essenciais para principalmente ser usada no desenho de cada objeto em cena. Dentre elas pode-se destacar a matriz de transformação (linha 3), cor (linha 5), lista de componentes (linha 6) e a lista de filhos (linha 11). No Quadro 3 pode-se visualizar o código do construtor da classe do `GameObject` padrão.

Quadro 3 - Construtor do `GameObject`

```

1 constructor({origin = new Point3D(0, 0, 0), color = new Color()}) {
2     this.__id = JSUtils.generateUUID();
3     this.__matrix = mat4.create();
4     this.__origin = origin;
5     this.__color = color;
6     this.__listComponents = new ComponentList();
7     this.__listComponents.addComponent(new TranslateComponent({owner:
this}));
8     this.__listComponents.addComponent(new RotateComponent({owner:
this }));
9     this.__listComponents.addComponent(new ScaleComponent({owner: this
}));
10    this.translate.translation = [origin.x, origin.y, origin.z];
11    this.__children = [];
12 }

```

Fonte: elaborado pelo autor.

Além dos atributos já citados há também o `id` (linha 1) gerado de forma aleatória buscando assim possuir um para cada `GameObject` criado para que se possa recupera-lo por esse `id` caso seja necessário. Além disso no construtor já são criados os componentes responsáveis pelas transformações geométricas (linhas 7, 8 e 9) e assim dessa forma o componente de translação já é usado para colocar o objeto na posição que for passada como parâmetro, pois todos os objetos de jogo são criados por padrão com `x`, `y` e `z` zerado.

Por fim para que um `GameObject` seja completo em cada classe que herde da classe padrão, precisa que no seu construtor adicione na lista de componentes o componente responsável por desenhar tal objeto. No Quadro 4 encontra-se o código pertencente a classe `CubeGameObject`. Ela é responsável por representar um cubo dentro do motor de jogos. Para que tudo funcione de forma correta é necessário instanciar um objeto de

CubeRenderComponent (linha 3), logo após isso é necessário chamar o método `onLoad` dele e atribuir a cor a ele, dessa forma o cubo será desenhado na cena de forma correta.

Quadro 4 - Código da Classe `CubeGameObject.js`

```

1 class CubeGameObject extends GameObject{
2     constructor({point = new Point3D(0,0,0), color = new Color()}) {
3         super({origin : point, color});
4         this.__listComponents.addComponent(new
5         CubeRenderComponent({owner : this}));
6         this.render.onLoad();
7         this.render.color = color;
8     }
9     get render() {
10        return this.listComponents[CubeRenderComponent.tag];
11    }
12    get tag(){
13        return "CUBE_OBJECT";
14    }
15    }

```

Fonte: elaborado pelo autor.

3.3.2.2 Luz

Semelhante ao `GameObject` a luz é a representação lógica de iluminação dentro do motor de jogos. Foram implementados três tipos diferentes: `DirectionalLight`, `PointLight` e `Spotlight`. Representando uma luz direcional, um ponto de luz e um holofote respectivamente. No Quadro 5 pode ser observado o código do construtor padrão da luz.

Quadro 5 - Construtor padrão da luz

```

1 constructor({color = new Color({ r: 0, b: 0, g: 0 }), position = new
2 Point3D(0, 0, 0)}) {
3     this.__color = color;
4     this.__position = position;
5 }

```

Fonte: elaborado pelo autor.

Para cada tipo diferente de luz seu construtor muda, nos Quadro 6, Quadro 7 e Quadro 8 pode ser visto os construtores da `DirectionalLight`, `PointLight` e `Spotlight` respectivamente.

Quadro 6 - Construtor da `DirectionalLight`

```

1 constructor({color = new Color({r:0, b:0, g : 0}), position = new
2 Point3D(0,0,0) }) {
3     super({color : color, position : position});
4 }

```

Fonte: elaborado pelo autor.

Quadro 7 - Construtor do `PointLight`

```

1 constructor({color = new Color({r: 1, b: 1, g: 1}), position = new
  Point3D(0, 0, 0), shininess = 1, secondColor = new Color({r: 1, g: 1,
  b: 1}) }) {
2   super({color: color, position: position});
3   this.__shininess = shininess;
4   this.__secondColor = secondColor
5 }

```

Fonte: elaborado pelo autor.

Quadro 8 - Construtor do `Spotlight`

```

1 constructor({ position = new Point3D(0, 0, 0), color = new Color({ r:
  1, g: 1, b: 1 }), innerLimit = 0, outerLimit = 0, target = new
  Point3D(0, 0, 0) }) {
2   super({ color: color, position: position });
3   this.__outerLimit = (Math.PI / 180) * outerLimit;
4   this.__innerLimit = (Math.PI / 180) * innerLimit;
5   this.__target = target;
6   this.__shininess = 1;
7   this.__matrix = mat4.create();
8   mat4.lookAt(this.__matrix, [position.x, position.y, position.z],
  target.toVector(), [0, 1, 0]);
9 }

```

Fonte: elaborado pelo autor.

Para a `DirectionalLight` o construtor padrão já basta, agora para o `PointLight` há também atributos relacionados para a segunda cor que a luz emite no seu centro e a intensidade do brilho. Para o `Spotlight` além dos atributos da classe padrão há também os atributos para controlar o degrade da luz entre o interior e o exterior e a posição para onde a luz deverá apontar.

3.3.2.3 Cena

A cena funciona como um agregador dos `GameObject` junto com as luzes para que depois essas informações possam ser acessadas pelas funções responsáveis pelo desenho. No Quadro 9 pode-se observar o código responsável por adicionar um `GameObject` na cena. Nesse momento que todos os componentes adicionados nele executam o seu método `onLoad()`.

Quadro 9 - Código que adiciona um `GameObject` a cena

```

1 addGameObject(gameObject) {
2   this.__gameObjectList.push(gameObject);
3   for (let componentKey in gameObject.listComponents) {
4     let component = gameObject.listComponents[componentKey];
5     component.onLoad();
6   }
7 }

```

Fonte: elaborado pelo autor.

Quando uma luz é adicionada ou removida da cena é necessário alterar o código do *vertex shader* presente nos componentes de desenho da cena, pois as informações da luz adicionada ou removida precisam ser alteradas.

3.3.2.4 Câmera

O motor de jogos oferece duas opções de câmera sintética, ortogonal e perspectiva. Para se criar a matriz de cada câmera fez-se uso da biblioteca glmatrix. Nos Quadro 10 e Quadro 11 apresenta-se os códigos dos construtores da câmera em perspectiva e ortogonal respectivamente.

Quadro 10 - Código do construtor da câmera em perspectiva

```

1 constructor({ aspect = 1, near = 0, far = 0, fovy = 0, position = new
  Point3D(0, 0, 0) }) {
2   super();
3   this.__near = near;
4   this.__far = far;
5   this.__aspect = aspect;
6   this.__fovy = fovy;
7   this.__position = position;
8   this.__projection = mat4.create();
9   this.__matrix = mat4.create();
10  mat4.perspective(this.__projection, fovy, aspect, near, far);
11  mat4.lookAt(this.__matrix, [position.x, position.y, position.z],
    [0, 0, 0], [0, 1, 0]);
12 }

```

Fonte: elaborado pelo autor.

Quadro 11 - Código do construtor da câmera ortogonal

```

1 constructor({ left, right, bottom, top, near, far }) {
2   super();
3   this.__projection = mat4.create();
4   mat4.ortho(this.__projection, left, right, bottom, top, near, far);
5 }

```

Fonte: elaborado pelo autor.

3.3.2.5 Classes System

As classes `system` são responsáveis por receber e propagar os eventos externos, como por exemplo o pressionar de uma tecla, a chamada do sistema para desenhar a cena, entre outros. Para demonstração no Quadro 12 tem-se o código do `RenderSystem` responsável por propagar o evento de desenho de cada *frame*.

Cada propagação de evento executa uma função específica do componente e sempre segue essa ordem: começa pelo `Game`, seguido pela `cena` e por fim os `GameObject`. As classes `system` presentes atualmente no motor de jogos são: `RenderSystem`, `LogicSystem` e `KeyboardSystem`. Responsáveis por propagar os eventos de desenhar a cada *frame*, atualização de cada *frame*, clique ou movimento do mouse e pressionar de uma tecla respectivamente.

Quadro 12 - Código da classe `RenderSystem`

```

1 class RenderSystem {
2     static fireRenderListener(){
3         let game = new Game();
4         game.context.clearColor(1.0, 1.0, 1.0, 1.0);
5         game.context.enable(game.context.CULL_FACE);
6         game.context.clear( game.context.COLOR_BUFFER_BIT |
game.context.DEPTH_BUFFER_BIT);
7         game.context.clearDepth(1.0);
8         game.context.enable(game.context.DEPTH_TEST);
9         game.context.depthFunc(game.context.LEQUAL);
10        for (let key in game.listComponents) {
11            let component = game.listComponents[key];
12            component.onRender(game.context, game.projection);
13        }
14        let scene = game.scene;
15        if (scene) {
16            for (let key in scene.listComponents){
17                let component = scene.listComponents[key];
18                component.onRender(game.context, game.projection);
19            }
20            for (let gameObject of scene.gameObjectList) {
21                if (gameObject instanceof GameObject) {
22                    for (let index in gameObject.listComponents) {
23                        let component =
gameObject.listComponents[index];
24                        component.onRender(game.context,
game.projection);
25                    }
26                }
27            }
28        }
29        let camera = game.camera;
30
31        if (camera){
32            for (let key in camera.listComponent) {
33                let component = camera.listComponent[key];
34                component.onRender(game.context, game.projection);
35            }
36        }
37    }
38    static get tag () {
39        return "RENDER_SYSTEM";
40    }

```

Fonte: elaborado pelo autor.

3.3.2.6 Componentes

Os componentes são onde a arquitetura baseada em componentes realmente se faz presente. Cada componente herda da classe padrão `Component` e nela estão presentes os métodos que são responsáveis por tratar os eventos recebidos pela classe `System`. Atualmente o `Game`, a cena, a câmera e os `GameObject` podem possuir componentes. No Quadro 13 pode-se observar o código da classe `Component` padrão.

Quadro 13 - Código da classe do componente padrão

```

1 class Component {
2     constructor({ owner }) {
3         this.__id = JSUtils.generateUUID();
4         this.__enabled = true;
5         this.__owner = owner;
6     }
7     get id() {
8         return this.__id;
9     }
10    get enabled() {
11        return this.__enabled;
12    }
13    set enabled(enabled) {
14        this.__enabled = enabled;
15    }
16    get owner() {
17        return this.__owner;
18    }
19    set owner(owner) {
20        this.__owner = owner;
21    }
22    onKeyDown(keyCode) { }
23    onKeyUp(keyCode) { }
24    onKeyPress(keyCode) { }
25    onClick(x, y, wich) { }
26    onMouseDown(x, y, wich) { }
27    onMouseMove(x, y) { }
28    onBeforeRender(context) { }
29    onRender(context, projectionMareix) { }
30    onUpdate(delta) { }
31    onLoad() { }
32    onDestroy() { }
33    get tag() {
34        return "COMPONENT";
35    }
36 }

```

Fonte: elaborado pelo autor.

Dentro do motor de jogos dois tipos de componentes já são especificados e precisam estar presentes para que ocorra o funcionamento correto, podendo ou não ser os implementados nesse trabalho. São eles os componentes responsáveis por desenhar os `GameObject` de jogo e os responsáveis por fazer as transformações geométricas. Para que o grau de abstração seja melhor, criou-se uma camada a mais entre a classe `Component` e as classes específicas de desenho de cada `GameObject`. Essa camada é a classe `RenderComponent`. E é por ela que todas as outras classes responsáveis pelo desenho herdam. No Quadro 14, pode-se visualizar os métodos principais dessa classe.

Os principais métodos a serem sobrescritos por cada classe que herdar de `RenderComponent` são: `vertexShaderSource` e `fragmentShaderSource`, responsáveis respectivamente por retornar o código para o vertex shader e fragment shader específico de cada objeto.

Quadro 14 - Código da classe `RenderComponent`

```

1  ...
2  export class RenderComponent extends Component {
3      ...
4
5      vertexShaderSource() {
6          return "Please implement abstract method vertexShaderSource."
7      };
8      onLoad(){
9          ...
10
11         this.__vertexShader = JSUtils.createShader(gl.VERTEX_SHADER,
12         ligths + this.vertexShaderSource());
13
14         this.__fragmentShader =
15         JSUtils.createShader(gl.FRAGMENT_SHADER, this.fragmentShaderSource());
16     }
17     fragmentShaderSource() {
18         return "Please implement abstract method
19         fragmentShaderSource."
20     };
21     get vertexShader() {
22         return this.__vertexShader;
23     }
24     get fragmentShader () {
25         return this.__fragmentShader;
26     }
27     get program() {
28         return this.__program;
29     }
30     renderChild(context, parentMatrix) {
31         for (let i = 0; i < this.owner.child.length; i++) {
32             const child = this.owner.child[i];
33             child.render.onRender(context, parentMatrix);
34         }
35     }
36     ...
37 }

```

Fonte: elaborado pelo autor.

As transformações geométricas também são efetuadas por meio de componentes e igualmente o desenho dos `GameObject` há uma opção já implementada. As transformações geométricas por usarem matrizes foram implementadas com auxílio da biblioteca `gl-matrix`. Mesmo assim pensou-se em fazer da forma mais desacoplada possível para que fosse fácil realizar a troca da biblioteca ou do próprio componente em si. No Quadro 15 observa-se o código do componente de translação. Os outros componentes de transformação geométrica, rotação e escala, seguem o mesmo princípio apenas mudando a chamada do método da biblioteca.

Por fazer-se uso da biblioteca `gl-matrix` as transformações ocorrem exatamente como foram implementadas nela que é de forma relativa. Dessa forma, toda transformação é a

acumulação do estado anterior com o novo valor. Exemplo ao se fazer a translação de um objeto em x acrescentando 2, com x igual a 5, o resultado será x igual a 7.

Quadro 15 - Código do componente de translação

```

1 class TranslateComponent extends Component{
2     constructor({owner}) {
3         super({owner : owner});
4         this.__translation = vec3.create();
5     }
6     set translation(translation){
7         vec3.set(this.__translation, translation[0], translation[1],
translation[2]);
8         this.translate(this.owner.matrix);
9     }
10    get translation() {
11        return this.__translation;
12    }
13    get x() {
14        return this.__translation[0];
15    }
16    get y() {
17        return this.__translation[1];
18    }
19    get z() {
20        return this.__translation[2];
21    }
22    set x(x) {
23        this.__translation[0] = x;
24        mat4.translate(this.owner.matrix, this.owner.matrix, [x, 0,
0]);
25    }
26    set y(y) {
27        this.__translation[1] = y;
28        mat4.translate(this.owner.matrix, this.owner.matrix, [0, y,
0]);
29    }
30    set z(z) {
31        this.__translation[2] = z;
32        mat4.translate(this.owner.matrix, this.owner.matrix, [0, 0,
z]);
33    }
34    translate(matrix){
35        mat4.translate(matrix, matrix, this.__translation);
36    }
37    get tag(){
38        return TranslateComponent.tag;
39    }
40    static get tag(){
41        return "TRANSLATE_COMPONENT";
42    }
43 }

```

Fonte: elaborado pelo autor.

3.3.2.7 Game

O Game é a estrutura principal do motor de jogos é nele que as classes system são cadastradas para tratar os eventos externo. Também é nele que a câmera e a cena juntam-se para haver o desenho dos objetos na tela. Além disso o Game é responsável por dar início ao

loop principal do motor de jogos. Desta forma optou-se por limitar a existência de apenas um objeto da classe `Game` por aplicação e para isso usou-se o padrão singleton. Visto que além disso vários dos atributos da classe `Game` são compartilhados com diferentes parte do motor de jogos. A forma que o padrão singleton foi implementada pode-se observar no Quadro 16 presente no construtor da classe `Game`. Nele a variável `INSTANCE`, acessada apenas dentro do arquivo da classe, é responsável por controlar se ao chamar o construtor da classe `Game` deve-se criar uma nova instancia ou retornar a já criada anteriormente (linha 2).

Quadro 16 - Construtor da classe `Game`

1	<code>constructor(context = undefined, scene = undefined, camera = undefined) {</code>
2	<code>if (!INSTANCE) {</code>
3	<code>this.__context = context;</code>
4	<code>this.__scene = scene;</code>
5	<code>this.__listComponents = new ComponentList();</code>
6	<code>this.__requestAnimFrame = undefined;</code>
7	<code>this.__lastUpdateTime = 0;</code>
8	<code>this.__projection = mat4.create();</code>
9	<code>this.__camera = camera;</code>
10	<code>INSTANCE = this;</code>
11	<code>this.startGameLoop();</code>
12	<code>}</code>
13	
14	<code>return INSTANCE;</code>
15	<code>}</code>

Fonte: elaborado pelo autor.

No quadro Quadro 17 pode-se visualizar os principais métodos da classe `Game` bem como suas respectivas implementações. O método `startGameLoop` é chamado logo após um objeto de `Game` ser criado, ele é responsável por cadastrar a função de renderização na função do navegador `window.requestAnimationFrame` e também por cadastrar a classe `KeySystem` para receber os retornos do evento do teclado. O método `gameLoop()` é o loop principal do motor responsável por disparar os eventos na classe `RenderSystem` e na classe `LogicSystem`

Quadro 17 - Código presente na classe Game

```

1  ...
2  export class Game {
3
4  ...
5      startGameLoop() {
6          let Loop = () => {
7              this.__requestAnimFrame =
window.requestAnimationFrame(Loop);
8              this.gameLoop();
9          };
10         Loop();
11         window.addEventListener("keypress",
KeySystem.fireKeyPressListener);
12         window.addEventListener("keydown",
KeySystem.fireKeyDownListener);
13         window.addEventListener("keyup", KeySystem.fireKeyUpListner);
14     }
15     ...
16     gameLoop() {
17         this.renderGame();
18         let now = Date.now();
19         let deltaTime = (now - this.__lastUpdateTime) / 1000;
20         this.updateGame(deltaTime);
21         this.__lastUpdateTime = Date.now();
22     }
23 }

```

Fonte: elaborado pelo autor

3.3.3 Operacionalidade da implementação

Para apresentar o uso do motor de jogos escolheu-se um cenário (Quadro 18) simples. Ele consiste de uma cena com um cubo e uma esfera sendo essa esfera filha do cubo. Nesse cenário há também a presença de uma luz e optou-se por usar a câmera em perspectiva.

A primeira coisa a se fazer para usar o motor de jogos é ter um elemento de `canvas` declarado na página HTML. Esse elemento é responsável por exibir a cena e usando ele é possível ter acesso ao uso da GPU. Com esse elemento criado deve-se acessá-lo pelo código (linha 11) e em seguida criar o contexto (linha 13), como o motor de jogos desenvolvido trabalha em 3D os parâmetros possíveis para a função `getContext` são: "webgl", "experimental-webgl" e "webgl2".

Seguindo a explicação, cria-se uma cena (linha 15) e após isso uma câmera (linha 16), nesse caso em perspectiva, com isso pode-se criar uma instância da classe `Game` (linha 17). Em seguida cria-se três instâncias da classe `Cor`, uma para cor vermelha, azul e branca, para serem usadas como cor dos objetos e luz. Para os objetos de jogo criou-se uma instância da classe `CubeGameObject` (linha 24) e outra de `SphereGameObject` (linha 25) usando as cores criadas anteriormente e como não se passou nenhum parâmetro para indicar a posição ambos serão criados na posição `x`, `y` e `z` igual zero. Cria-se em seguida uma instância da classe

`DirectionalLight` (linha 27) para que haja incidência de luz em cena, mesmo que não seja necessário ter luz para que o objeto apareça.

Quadro 18 - Código utilizando o motor de jogos

```

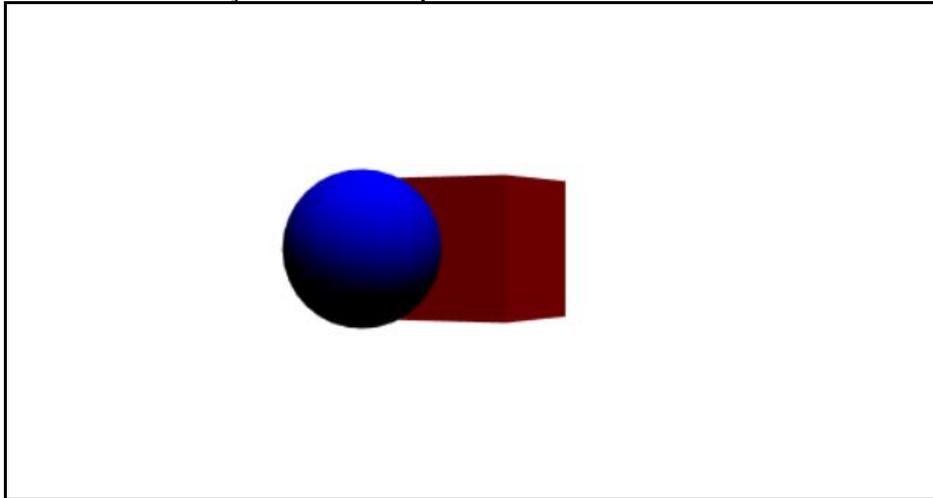
1 import { Scene } from "../game/Scene";
2 import { Point3D } from "../geometric/Point3D";
3 import { PerspectiveCamera } from "../game/PerspectiveCamera";
4 import { Game } from "../game/Game";
5 import { Color } from "../geometric/Color";
6 import { CubeGameObject } from "../gameObject/CubeGameObject";
7 import { PointLight } from "../Light/PointLight";
8 import { SphereGameObject } from "../gameObject/SphereGameObject";
9 import { DirectionalLight } from "../Light/DirectionalLight";
10
11 let canvas = document.getElementById("glCanvas");
12
13 let context = canvas.getContext("webgl2");
14
15 let scene = new Scene();
16 let camera = new PerspectiveCamera({near: 0.1, far : 500, aspect : 1,
17   fovy : 45 * Math.PI / 180, position : new Point3D(0, 0, 15)});
18 let game = new Game(context, scene, camera);
19
20 let red = new Color({r : 1});
21 let blue = new Color({b : 1});
22 let white = new Color({r : 1, g : 1, b : 1});
23
24 let cube = new CubeGameObject({color : red});
25 let sphere = new SphereGameObject({color : blue});
26
27 let directLight = new DirectionalLight({color : white, position : new
28   Point3D(2, 8, 5)});
29 scene.addLight(directLight);
30
31 cube.rotation.onUpdate = (deltaTime) => {
32   cube.rotation.y = 0.9 * deltaTime;
33 }
34
35 cube.translate.z = -5;
36 sphere.translate.x = 3;
37
38 scene.addGameObject(cube);
39 cube.addGameObject(sphere);

```

Fonte: elaborado pelo autor.

Nesse exemplo aproveitou-se do próprio componente de rotação para que a cada `frame` seja atualizada a rotação do objeto (linha 31). Por fim, adiciona-se os objetos de jogos na cena usando o método `addGameObject` (linha 38) e para adicionar a esfera como filho do cubo no grafo de cena usa-se o mesmo método, porém chame-se ele através da instância do cubo (linha 39). A visualização desta cena pode ser vista na Figura 16.

Figura 16 - Visualização da cena implementada utilizando o motor desenvolvido



Fonte: elaborado pelo autor.

3.4 ANÁLISE DOS RESULTADOS

Nessa seção serão apresentados os testes para analisar o desempenho do motor de jogos, bem como sua usabilidade.

3.4.1 Testes de desempenho

Para se analisar o desempenho do motor de jogos montou-se seis cenários, seus respectivos códigos podem ser visualizados no Apêndice A. Para confrontar os resultados obtidos implementou-se os mesmos cenários usando o trabalho correlato Three.js, os códigos desenvolvidos podem ser visualizados no Apêndice B. A essência de todos os cenários é a mesma, possuem uma câmera em perspectiva, uma luz direcional e todos os objetos em cena rotacionando. O que muda em cada um é a quantidade de esferas, no cenário 1 usou-se 50 esferas, no cenário 2 foram 100 esferas, assim por diante incrementando de cinquenta em cinquenta até que no último colocou-se 300 esferas simultâneas. Optou-se por esferas por serem os objetos mais complexos presente no motor implementado e 300 foi o máximo presente em cena antes de acontecer estouro de memória. Os cenários funcionaram em todos os navegadores, mas os testes de desempenho foram feitos só no Google Chrome (versão 67.0.3396.99). Esta escolha foi porque este navegador tem uma ferramenta chamada DevTools que auxilia na análise das estáticas da página, assim garantindo que quando for comparar os resultados com o Three.js não haverá interferência do uso dos componentes de análise de performance.

3.4.1.1 Análise do FPS

As análises seguiram da seguinte forma, esperava-se a cena carregar por completo e então utilizava-se a ferramenta DevTools para analisar o FPS por cinco segundos. O resultado exibido na Tabela 1 é a média de quadros por segundo durante esse tempo.

Tabela 1 - Quantidade de quadro por segundo em cada cenário

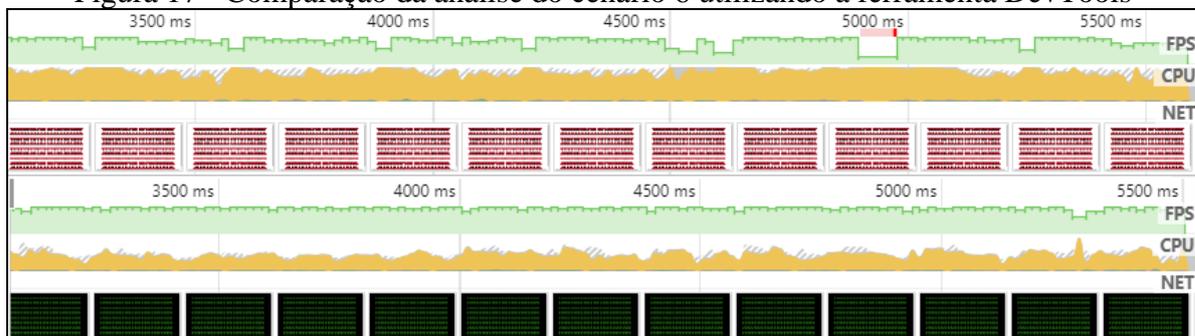
Cenários	TowelJS	Three.js
Cenário 1	60	60
Cenário 2	60	60
Cenário 3	59	60
Cenário 4	57	60
Cenário 5	50	60
Cenário 6	46	60

Fonte: elaborado pelo autor.

Como pode-se observar não ocorre variação de FPS nos cenários implementados usando Three.js. Contudo, nos implementados usando o motor desenvolvido nos dois primeiros cenários a taxa de quadros é igual. E a partir do terceiro cenário começa a ficar cada vez menor.

Ao olhar com mais calma a análise feita pela ferramenta pode-se observar que o loop realizado pelo TowelJS no cenário 6 há quadros marcados com vermelho nos 5000ms, como pode-se observar na Figura 17. Isso indica que essa parte está causando um gargalo no desenho da cena. A parte em questão pertence à classe `SphereRenderComponent` no seu método `onRender`. Tendo como possível causa dessa diferença a diferença de arquitetura empregada entre o motor desenvolvido e a Three.js. Visto que quando analisado o cenário implementado com a Three.js não apresenta a mesma marcação.

Figura 17 - Comparação da análise do cenário 6 utilizando a ferramenta DevTools



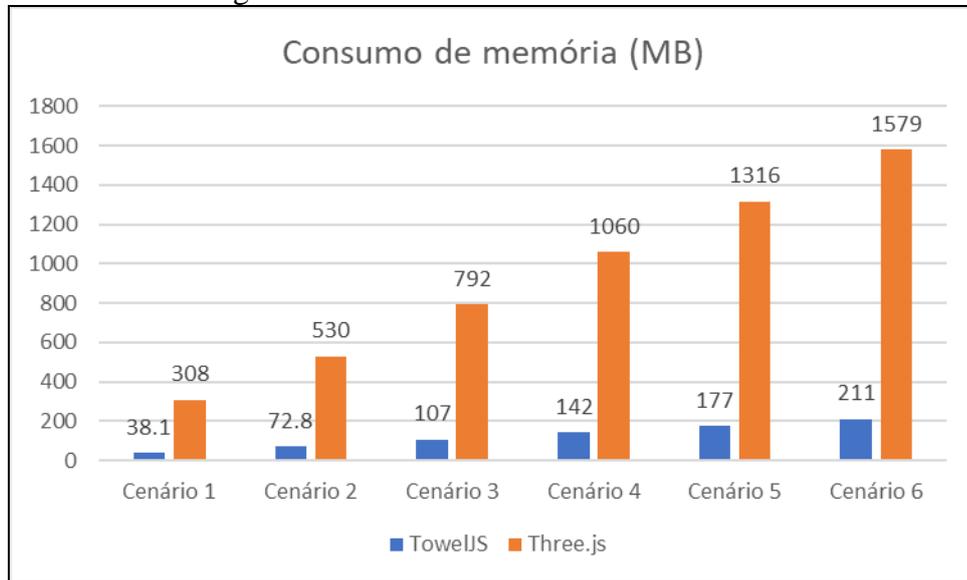
Fonte: elaborado pelo autor.

3.4.1.2 Análise do uso de memória

De forma semelhante aos testes de FPS, os testes de memória também foram feitos comparando o código desenvolvido com a Three.js. Para analisar o uso de memória fez-se uso dos mesmos cenários já descritos. Os dados foram obtidos após a cena ser carregada por

completa usando a ferramenta DevTools. Na Figura 18 pode-se visualizar um gráfico com os valores obtidos.

Figura 18 - Gráfico de consumo de memória



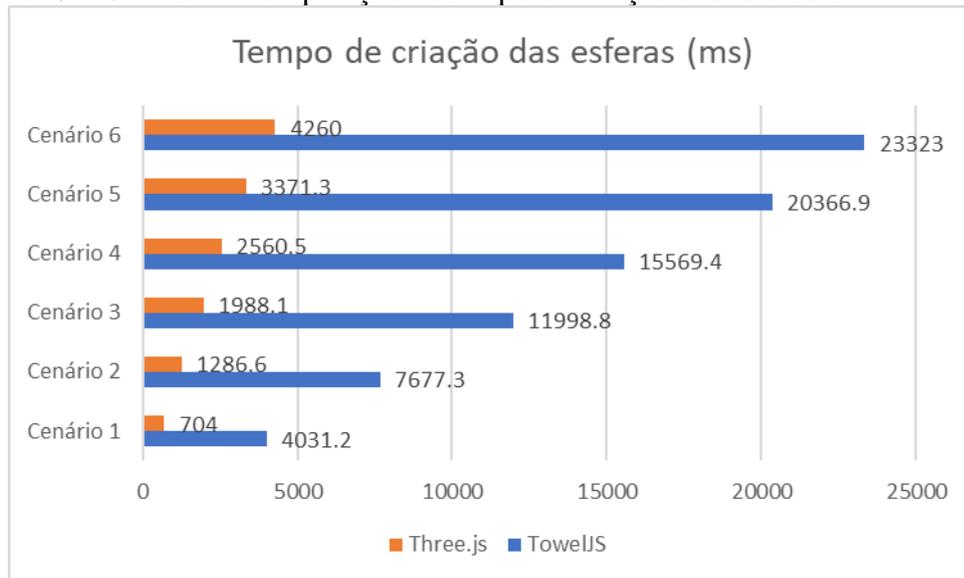
Fonte: elaborado pelo autor.

Como pode-se observar o motor desenvolvido consumiu menos memória em todos os testes. O principal fator para isso ter ocorrido pode ser o fato de que o TowelJS é um motor de jogos mais simples, com menos funcionalidades e recursos se comparado a Three.js. Dessa forma, os recursos a mais presente na Three.js já podem estar carregados em memória pronto para uso mesmo que os códigos dos cenários não os utilizem.

3.4.1.3 Análise do tempo de carregamento

Por fim foram analisados o tempo de criação das esferas em cada cenário. Para isso, o código responsável por criar as esferas foi isolada e observou-se o tempo de execução desse método. Na Figura 19 pode-se observar os tempos obtidos.

Figura 19 - Gráfico de comparação do tempo da criação das esferas em cada cenário



Fonte: elaborado pelo autor.

Como pode-se observar o tempo de criação das esferas utilizando Three.js é muito inferior ao utilizando o motor desenvolvido. Isso pode ocorrer pelo mesmo motivo que a diminuição do FPS, o uso de componentes. Ao usar componentes para tudo se adiciona mais uma camada de acesso e isso pode acabar implicando no tempo de execução já que agora não há um acesso direto aos atributos. Ou até mesmo pelo código do objeto estar separado em mais de uma classe.

3.4.2 Análise da usabilidade

Para avaliação qualitativa do motor de jogos realizou-se uma entrevista com o acadêmico Gonçalves (2018), ele já havia sido aprovado na disciplina de computação gráfica e utilizado o Unity3D, um motor gráfico reconhecido e usado comercialmente. Assim ele teria um conhecimento de como funciona a implementação de uma aplicação gráfica. Inicialmente a ideia era que ele receberia três atividades¹ (Apêndice C), para realizar com auxílio apenas do tutorial² (Apêndice D). Porém isso não pode ser feito ao observar-se que o tutorial não explicava alguns pontos importantes a serem realizados nas atividades. Dessa forma as atividades foram realizadas com o auxílio do desenvolvedor do motor de jogos. Os pontos apontados por Gonçalves (2018), se mostram positivos, com ressalva ao tutorial que estava incompleto ou vago em certos pontos. Quando perguntado se achava que o motor poderia ajudar no ensino de computação gráfica, sua resposta foi positiva, justificando ser mais fácil

¹ disponível em: <https://goo.gl/forms/hschqwJwbsXY0dPc2>

² <https://gzanluca.bitbucket.io/>

que o OpenGL. Outro ponto apontado seria o fato do motor não possuir um editor gráfico, que poderia facilitar a edição das cenas. Por fim, algo que foi descoberto com essas atividades é que mesmo a cena contar com uma luz no navegador Opera no sistema operacional MacOS essa função não funciona.

3.4.3 Comparativo entre os trabalhos correlatos

Nessa seção será comparado o trabalho desenvolvido com os trabalhos apresentados na seção 2.4.

Como pode ser observado no Quadro 19, o trabalho desenvolvido contempla quase todas as principais características dos trabalhos correlatos, sendo dois deles a Three.js e o WebGLStudio.js aplicações com intuito mais comercial. Da lista apresentada apenas não foram contemplados a opção de fazer a seleção de um objeto com auxílio do mouse e possuir um editor que facilitasse o desenvolvimento. Todos os trabalhos inclusive o desenvolvido são implementados utilizando JavaScript e usam o navegador para executar sua aplicação. Isso ajuda no fato de facilitar o acesso.

Quadro 19 - Comparação dos trabalhos correlatos com o trabalho desenvolvido

Características	Three.js (2018)	WebGLStudio.js (2018)	VisEdu-Engine (2013)	TowelJS (2018)
implementado em JavaScript	Sim	Sim	Sim	Sim
possuir grafo de cena	Sim	Sim	Não	Sim
sistema baseado em componentes	Não	Sim	Sim	Sim
gráfico em 3D	Sim	Sim	Não	Sim
possui um editor	Sim	Sim	Sim	Não
iluminação	Sim	Sim	Não	Sim
câmera sintética	Sim	Sim	Sim	Sim
seleção de objetos com o mouse	Sim	Sim	Sim	Não

Fonte: elaborado pelo autor.

Mesmo não sendo tão completo quanto as opções apresentadas, o fato dele ser mais simples pode facilitar o entendimento por quem não tem tanto conhecimento em computação gráfica. O uso de componentes pode realmente ajudar na reutilização de código, mas ainda precisa ser analisado como melhorar o desempenho.

4 CONCLUSÕES

Este trabalho apresentou o desenvolvimento de um motor de jogos utilizando JavaScript, WebGL e arquitetura baseada em componentes. Dentre as dificuldades durante o desenvolvimento vale o destaque para a forma como foram implementadas as transformações geométricas e as luzes. Para evitar-se processamento desnecessário do recálculo da matriz de transformação do objeto a cada vez que desenhasse-o optou-se por recalculá-la somente quando realmente houve-se alguma alteração nos seus valores. Com isso as transformações funcionam de forma relativa e não absoluta, isso pode causar uma pequena confusão no início ao utilizar o motor. A dificuldade com relação as luzes, se deve ao fato de que inicialmente haviam sido validadas apenas para os cubos. Quando se incluiu as esferas haviam detalhes referente aos vetores normais que só foram reparados nesse momento. Mesmo com essas dificuldades o motor cumpriu seu objetivo como ferramenta para facilitar e abstrair certas partes no desenvolvimento de aplicações gráficas em JavaScript.

O uso da arquitetura baseada em componentes se mostrou eficiente para ser utilizado num motor de jogos. Por alcançar-se um baixo nível de acoplamento e um grande grau de reutilização dos componentes. Contudo ainda é preciso analisar melhor as questões referentes ao desempenho, para que quando haja muitos objetos em cena ou processos mais complexos entre cada *frame* o FPS não seja afetado. Contemplando o primeiro objetivo específicos disponibilizou-se junto ao trabalho dois componentes exclusivos para analisar a performance. O primeiro chamado de `FPSComponent` tem a função de contabilizar os FPS na cena. O segundo denominado de `MemoryUsageComponent` tem por sua vez a função de exibir a memória que a aplicação atual está utilizando.

Durante os testes de usabilidade do motor de jogos desenvolvido por outro desenvolvedor, percebeu-se que o tutorial criado para que os auxiliasse, no uso do motor, precisava de melhoria. A principal dificuldade estava no entendimento da criação e manipulação dos componentes. Após melhorias no tutorial o motor se mostrou como uma ferramenta com potencial para ser usado por pessoas que não tenham um conhecimento tão aprofundado em computação gráfica. Visto que um nível de abstração conseguiu ser alcançado graças ao encapsulamento do código responsável pelo desenho e comportamento dos `gameObject` dentro dos seus respectivos componentes.

Por esta ser a primeira versão do motor implementada ele conta com algumas limitações como: não possuir uma classe `System` para tratar os eventos do mouse presentes no navegador. Logo dessa forma também não permite a seleção de objetos ou manipulação de

objetos pelo mouse. Outro ponto limitado é o fato de contar apenas com duas opções de `gameObject` (cubo e esfera). Algo que deixa o motor desenvolvido longe das opções usadas comercialmente e o fato de não permitir a aplicação de textura nos objetos. Isso faz com que a aparência seja limitada por cores solidas ou a interpolação feita pelo próprio WebGL entre as cores de cada vértice.

Os testes feitos em comparação ao Three.js mostraram resultados satisfatórios considerando que o mesmo é usado comercialmente e com muito mais tempo de desenvolvimento. Mesmo que em sua maioria o motor desenvolvido tem tido um resultado inferior. Esses resultados não se mostraram tão distantes com relação ao FPS. No quesito uso de memória o motor desenvolvido teve um resultado melhor muito provável pela sua simplicidade maior em comparação ao Three.js. Já o tempo de criação dos cenários foi onde ocorreu a maior diferença mostrando que o componente de desenho precisa ser melhor analisado.

Por fim, o motor desenvolvido ser mais simples e limitado comparado as opções comerciais pode-se mostrar como uma vantagem para ser usado em sala de aula. Isso deve-se a exatamente ao fato de ser mais simples e limitado, pois dessa forma pode se tornar mais fácil seu entendimento ao mesmo tempo que ajuda no entendimento de conceitos básicos de computação gráfica. Não necessitando de conhecimentos profundos nessa área visto que motor encapsulou os principais códigos necessários para isso.

4.1 EXTENSÕES

Para trabalhos futuros, são sugeridas as seguintes extensões:

- a) permitir a troca de componentes de transformações geométricas e de desenho de objetos em tempo de execução;
- b) adicionar novos tipos de objetos, implementado seu `gameObject` e `RenderComponent` específico;
- c) possuir uma forma de selecionar objetos com o mouse;
- d) importar objetos feitos em software de terceiros como Blender ou Maya;
- e) permitir aplicar texturas;
- f) construir mais testes para análise de performance;
- g) analisar melhor o uso de componentes;
- h) fazer teste em outros navegadores e em outros sistemas operacionais;
- i) construir um editor gráfico para facilitar o uso do motor o motor desenvolvido.

REFERÊNCIAS

- CRYENGINE, **CRYENGINE** | The complete solution for next generation game development by Crytek. [S.l.], 2017. Disponível em: < <https://www.cryengine.com/>>. Acesso em: 25 out. 17.
- EBERLY, David H. **3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics**. 2. ed. New York: Morgan Kaufmann, 2006
- FEIJÓ, R. H. B. **Uma arquitetura de software baseada em componentes para visualização de informações industriais**. 2007. 88 f. Dissertação (Mestrado em Ciências) - Curso de Pós-graduação em Engenharia Elétrica, Universidade Federal de Rio Grande do Norte, Natal.
- GLMATRIX. **GLMatrix**. [S.l.], 2018. Disponível em: <<http://glmatrix.net/>>. Acesso em: 1 jul. 2018.
- GONÇALVES, João P. S. **Entrevista sobre usabilidade do motor de jogos**. Entrevistador: Gabriel Zanluca. Indaial. 2018. Entrevista feita através de conversação – não publicada.
- HARBS, Marcos. **Motor para Jogos 2D Utilizando HTML5**. 2013. 78 f. Trabalho de Conclusão de curso (Bacharelado em Ciência da Computação) – Centro de Ciências e Exatas Naturas, Universidade Regional de Blumenau, Blumenau.
- KHRONOS. **WebGL Overview**: The Khronos Group Inc. [S.l.], 2018. Disponível em: <<https://www.khronos.org/webgl/>>. Acesso em: 26 jun. 2018.
- LITESCENE. **litescene.js**. [S.l.], 2017. Disponível em: < <https://github.com/jagenjo/litescene.js>>. Acesso em: 13 set. 2017.
- MICROSOFT. **Visual Studio Code**. [S.l.], 2018. Disponível em: <<https://code.visualstudio.com/docs>>. Acesso em: 1 jul. 2018.
- NEWZOO. **The Global Games Market Will Reach \$108.9 Billion In 2017 With Mobile Taking 42%** [S.l.], 2017 Disponível em: <https://newzoo.com/insights/articles/the-global-games-market-will-reach-108-9-billion-in-2017-with-mobile-taking-42>. Acesso em: 9 set. 2017.
- NPM. **Npm**. [S.l.], 2018. Disponível em: <<https://www.npmjs.com/>>. Acesso em: 1 jul. 2018.
- PESSOA, Carlos A. C. **wGEM: um framework de desenvolvimento de jogos para dispositivos móveis**. 2001. 58f. Dissertação (Mestrado em Informática) – Centro de Informática – Universidade Federal de Pernambuco, Pernambuco.
- SAMETINGER, Johannes. **Software Engineering with Reusable Components**. New York: Springer, 1997.
- SZYPERSKI Clemens, **Component Software: Beyond Object-Oriented Programming** (2nd Edition). Great Britain: Addison-Wesley Professional, 2002.
- TAVARES, Gregg. **WebGL How It Works**. [S.l.], 2018a. Disponível em: < <https://webglfundamentals.org/webgl/lessons/webgl-fundamentals.html>>. Acesso em: 26 jun. 2018.
- TAVARES, Gregg. **WebGL How It Works**. [S.l.], 2018b. Disponível em: <<https://webglfundamentals.org/webgl/lessons/webgl-how-it-works.html>>. Acesso em: 26 jun. 2018.
- THREE.JS. **three.js**. [S.l.], 2018. Disponível em: < <https://github.com/mrdoob/three.js/>>. Acesso em: 24 jun. 2018.

THREE.JS. **three.js**. [S.l.], 2017a. Disponível em: < <https://github.com/mrdoob/three.js/>>. Acesso em: 9 set. 2017.

THREE.JS. **three.js / examples**. [S.l.], 2017b. Disponível em: < https://threejs.org/examples/#webgl_geometries>. Acesso em: 9 set. 2017.

THREE.JS. **three.js docs - Animation System**. [S.l.], 2017c. Disponível em: < <https://threejs.org/docs/index.html#manual/introduction/Animation-system>>. Acesso em: 13 set. 2017.

THREE.JS. **three.js docs**. [S.l.], 2017d. Disponível em: < <https://threejs.org/docs/index.html>>. Acesso em: 25 out. 2017.

UNITY, **Unity – Game Engine**. [S.l.], 2017. Disponível em: < <https://unity3d.com/pt> >. Acesso em: 25 out. 17.

UNREAL ENGINE, **Game Engine Technology by Unreal**. [S.l.], 2017. Disponível em: < <https://www.unrealengine.com/en-US/what-is-unreal-engine-4>>. Acesso em: 25 out. 17.

WEBGLSTUDIOJS. **webglstudio.js**. [S.l.], 2017. Disponível em: < <https://github.com/jagenjo/webglstudio.js/>>. Acesso em: 9 set. 2017.

WEBPACK. **Webpack**. [S.l.] 2018. Disponível em: <<https://webpack.js.org/>>. Acesso em: 1 jul. 2018.

APÊNDICE A – Cenários implementados usando o motor desenvolvido

Neste apêndice são apresentados os códigos (Quadro 19 ao 24) relativos a cada cenário usado nos testes implementados com o motor desenvolvido (Figura 19 a 24).

Quadro 20 - Cenário 1 implementado com o motor desenvolvido (50 esferas)

```
import { Scene } from "../game/Scene";
import { Point3D } from "../geometric/Point3D";
import { PerspectiveCamera } from "../game/PerspectiveCamera";
import { Game } from "../game/Game";
import { Color } from "../geometric/Color";
import { SphereGameObject } from "../gameObject/SphereGameObject";
import { DirectionalLight } from "../Light/DirectionalLight";
import { Component } from "../component/Component";
class RotateSphere extends Component{
  onUpdate(deltaTime){
    this.owner.rotation.x = 0.1 * deltaTime;
    this.owner.rotation.y = 0.1 * deltaTime;
  }
}
function initSphere(){
  for (let i = 0; i < 10; i++) {
    for (let j = 0; j < 5; j++) {
      let sphere = new SphereGameObject({ color: green, radius: 1,
latitudeBands: 60, longitudeBands: 60, position : new Point3D(positionX,
positionY, 0)});
      spheres.push(sphere);
      positionX += 2;
      scene.addGameObject(sphere);
    }
    positionY -= 4;
    positionX = -30;
  }
}
let scene = new Scene();
let canvas = document.getElementById("glCanvas");
canvas.width = window.innerWidth * window.devicePixelRatio;
canvas.height = window.innerHeight * window.devicePixelRatio;
let context = canvas.getContext("webgl");
let camera = new PerspectiveCamera({ near: 0.1, far: 1000, aspect:
window.innerWidth / window.innerHeight, fovy: 75 * Math.PI / 180,
position: new Point3D(0, 0, 25) });
let game = new Game(context, scene, camera);
let spheres = [];
let green = new Color({ r: 1 });
let white = new Color({ r: 1, g: 1, b: 1 });
let directLight = new DirectionalLight({ color: white, position: new
Point3D(-.24, 3.34, 7.5) });
scene.addLight(directLight);
var positionX = -30;
var positionY = 15;
initSphere();
for (let i = 0; i < spheres.length; i++) {
  let component = new RotateSphere({owner :spheres[i] });
  spheres[i].listComponents.addComponent(component);
}
}
```

Fonte: elaborado pelo autor.

Quadro 21 - Cenário 2 implementado com o motor desenvolvido (100 esferas)

```

import { Scene } from "../game/Scene";
import { Point3D } from "../geometric/Point3D";
import { PerspectiveCamera } from "../game/PerspectiveCamera";
import { Game } from "../game/Game";
import { Color } from "../geometric/Color";
import { SphereGameObject } from "../gameObject/SphereGameObject";
import { DirectionalLight } from "../Light/DirectionalLight";
import { Component } from "../component/Component";
class RotateSphere extends Component{
  onUpdate(deltaTime){
    this.owner.rotation.x = 0.1 * deltaTime;
    this.owner.rotation.y = 0.1 * deltaTime;
  }
}
function initSphere(){
  for (let i = 0; i < 10; i++) {
    for (let j = 0; j < 10; j++) {
      let sphere = new SphereGameObject({ color: green, radius: 1,
latitudeBands: 60, longitudeBands: 60, position : new Point3D(positionX,
positionY, 0)});
      spheres.push(sphere);
      positionX += 2;
      scene.addGameObject(sphere);
    }
    positionY -= 4;
    positionX = -30;
  }
}
let scene = new Scene();
let canvas = document.getElementById("glCanvas");
canvas.width = window.innerWidth * window.devicePixelRatio;
canvas.height = window.innerHeight * window.devicePixelRatio;
let context = canvas.getContext("webgl");
let camera = new PerspectiveCamera({ near: 0.1, far: 1000, aspect:
window.innerWidth / window.innerHeight, fovy: 75 * Math.PI / 180,
position: new Point3D(0, 0, 25) });
let game = new Game(context, scene, camera);
let spheres = [];
let green = new Color({ r: 1 });
let white = new Color({ r: 1, g: 1, b: 1 });
let directLight = new DirectionalLight({ color: white, position: new
Point3D(-.24, 3.34, 7.5) });
scene.addLight(directLight);
var positionX = -30;
var positionY = 15;
initSphere();
for (let i = 0; i < spheres.length; i++) {
  let component = new RotateSphere({owner :spheres[i] });
  spheres[i].listComponents.addComponent(component);
}

```

Fonte: elaborado pelo autor.

Quadro 22 - Cenário 3 implementado com o motor desenvolvido (150 esferas)

```

import { Scene } from "../game/Scene";
import { Point3D } from "../geometric/Point3D";
import { PerspectiveCamera } from "../game/PerspectiveCamera";
import { Game } from "../game/Game";
import { Color } from "../geometric/Color";
import { SphereGameObject } from "../gameObject/SphereGameObject";
import { DirectionalLight } from "../Light/DirectionalLight";
import { Component } from "../component/Component";
class RotateSphere extends Component{
  onUpdate(deltaTime){
    this.owner.rotation.x = 0.1 * deltaTime;
    this.owner.rotation.y = 0.1 * deltaTime;
  }
}
function initSphere(){
  for (let i = 0; i < 10; i++) {
    for (let j = 0; j < 15; j++) {
      let sphere = new SphereGameObject({ color: green, radius: 1,
latitudeBands: 60, longitudeBands: 60, position : new Point3D(positionX,
positionY, 0)});
      spheres.push(sphere);
      positionX += 2;
      scene.addGameObject(sphere);
    }
    positionY -= 4;
    positionX = -30;
  }
}
let scene = new Scene();
let canvas = document.getElementById("glCanvas");
canvas.width = window.innerWidth * window.devicePixelRatio;
canvas.height = window.innerHeight * window.devicePixelRatio;
let context = canvas.getContext("webgl");
let camera = new PerspectiveCamera({ near: 0.1, far: 1000, aspect:
window.innerWidth / window.innerHeight, fovy: 75 * Math.PI / 180,
position: new Point3D(0, 0, 25) });
let game = new Game(context, scene, camera);
let spheres = [];
let green = new Color({ r: 1 });
let white = new Color({ r: 1, g: 1, b: 1 });
let directLight = new DirectionalLight({ color: white, position: new
Point3D(-.24, 3.34, 7.5) });
scene.addLight(directLight);
var positionX = -30;
var positionY = 15;
initSphere();
for (let i = 0; i < spheres.length; i++) {
  let component = new RotateSphere({owner :spheres[i] });
  spheres[i].listComponents.addComponent(component);
}

```

Fonte: elaborado pelo autor.

Quadro 23 - Cenário 4 implementado com o motor desenvolvido (200 esferas)

```

import { Scene } from "../game/Scene";
import { Point3D } from "../geometric/Point3D";
import { PerspectiveCamera } from "../game/PerspectiveCamera";
import { Game } from "../game/Game";
import { Color } from "../geometric/Color";
import { SphereGameObject } from "../gameObject/SphereGameObject";
import { DirectionalLight } from "../Light/DirectionalLight";
import { Component } from "../component/Component";
class RotateSphere extends Component{
  onUpdate(deltaTime){
    this.owner.rotation.x = 0.1 * deltaTime;
    this.owner.rotation.y = 0.1 * deltaTime;
  }
}
function initSphere(){
  for (let i = 0; i < 10; i++) {
    for (let j = 0; j < 20; j++) {
      let sphere = new SphereGameObject({ color: green, radius: 1,
latitudeBands: 60, longitudeBands: 60, position : new Point3D(positionX,
positionY, 0)});
      spheres.push(sphere);
      positionX += 2;
      scene.addGameObject(sphere);
    }
    positionY -= 4;
    positionX = -30;
  }
}
let scene = new Scene();
let canvas = document.getElementById("glCanvas");
canvas.width = window.innerWidth * window.devicePixelRatio;
canvas.height = window.innerHeight * window.devicePixelRatio;
let context = canvas.getContext("webgl");
let camera = new PerspectiveCamera({ near: 0.1, far: 1000, aspect:
window.innerWidth / window.innerHeight, fovy: 75 * Math.PI / 180,
position: new Point3D(0, 0, 25) });
let game = new Game(context, scene, camera);
let spheres = [];
let green = new Color({ r: 1 });
let white = new Color({ r: 1, g: 1, b: 1 });
let directLight = new DirectionalLight({ color: white, position: new
Point3D(-.24, 3.34, 7.5) });
scene.addLight(directLight);
var positionX = -30;
var positionY = 15;
initSphere();
for (let i = 0; i < spheres.length; i++) {
  let component = new RotateSphere({owner :spheres[i] });
  spheres[i].listComponents.addComponent(component);
}

```

Fonte: elaborado pelo autor.

Quadro 24 - Cenário 5 implementado com o motor desenvolvido (250 esferas)

```

import { Scene } from "../game/Scene";
import { Point3D } from "../geometric/Point3D";
import { PerspectiveCamera } from "../game/PerspectiveCamera";
import { Game } from "../game/Game";
import { Color } from "../geometric/Color";
import { SphereGameObject } from "../gameObject/SphereGameObject";
import { DirectionalLight } from "../Light/DirectionalLight";
import { Component } from "../component/Component";
class RotateSphere extends Component{
  onUpdate(deltaTime){
    this.owner.rotation.x = 0.1 * deltaTime;
    this.owner.rotation.y = 0.1 * deltaTime;
  }
}
function initSphere(){
  for (let i = 0; i < 10; i++) {
    for (let j = 0; j < 25; j++) {
      let sphere = new SphereGameObject({ color: green, radius: 1,
latitudeBands: 60, longitudeBands: 60, position : new Point3D(positionX,
positionY, 0)});
      spheres.push(sphere);
      positionX += 2;
      scene.addGameObject(sphere);
    }
    positionY -= 4
    positionX = -30;
  }
}
let scene = new Scene();
let canvas = document.getElementById("glCanvas");
canvas.width = window.innerWidth * window.devicePixelRatio;
canvas.height = window.innerHeight * window.devicePixelRatio;
let context = canvas.getContext("webgl");
let camera = new PerspectiveCamera({ near: 0.1, far: 1000, aspect:
window.innerWidth / window.innerHeight, fovy: 75 * Math.PI / 180,
position: new Point3D(0, 0, 25) });
let game = new Game(context, scene, camera);
let spheres = [];
let green = new Color({ r: 1 });
let white = new Color({ r: 1, g: 1, b: 1 });
let directLight = new DirectionalLight({ color: white, position: new
Point3D(-.24, 3.34, 7.5) });
scene.addLight(directLight);
var positionX = -30;
var positionY = 15;
initSphere();
for (let i = 0; i < spheres.length; i++) {
  let component = new RotateSphere({owner :spheres[i] });
  spheres[i].listComponents.addComponent(component);
}

```

Fonte: elaborado pelo autor.

Quadro 25 - Cenário 6 implementado com o motor desenvolvido (300 esferas)

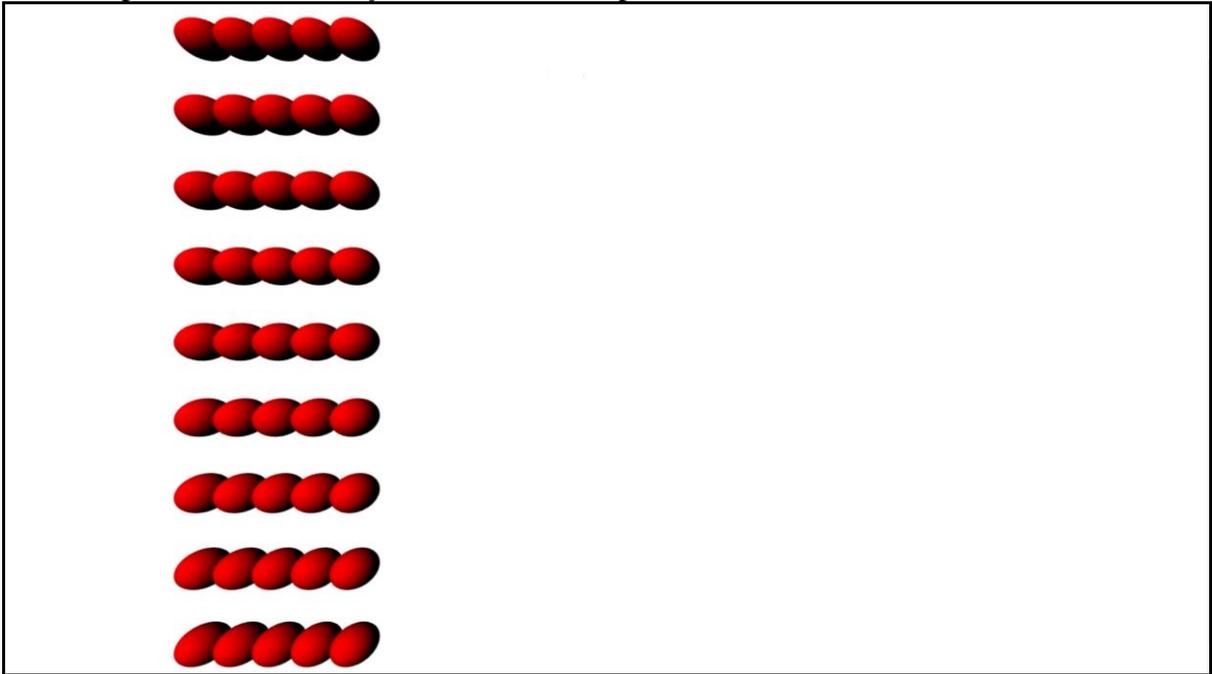
```

import { Scene } from "../game/Scene";
import { Point3D } from "../geometric/Point3D";
import { PerspectiveCamera } from "../game/PerspectiveCamera";
import { Game } from "../game/Game";
import { Color } from "../geometric/Color";
import { SphereGameObject } from "../gameObject/SphereGameObject";
import { DirectionalLight } from "../Light/DirectionalLight";
import { Component } from "../component/Component";
class RotateSphere extends Component{
  onUpdate(deltaTime){
    this.owner.rotation.x = 0.1 * deltaTime;
    this.owner.rotation.y = 0.1 * deltaTime;
  }
}
function initSphere(){
  for (let i = 0; i < 10; i++) {
    for (let j = 0; j < 30; j++) {
      let sphere = new SphereGameObject({ color: green, radius: 1,
latitudeBands: 60, longitudeBands: 60, position : new Point3D(positionX,
positionY, 0)});
      spheres.push(sphere);
      positionX += 2;
      scene.addGameObject(sphere);
    }
    positionY -= 4;
    positionX = -30;
  }
}
let scene = new Scene();
let canvas = document.getElementById("glCanvas");
canvas.width = window.innerWidth * window.devicePixelRatio;
canvas.height = window.innerHeight * window.devicePixelRatio;
let context = canvas.getContext("webgl");
let camera = new PerspectiveCamera({ near: 0.1, far: 1000, aspect:
window.innerWidth / window.innerHeight, fovy: 75 * Math.PI / 180,
position: new Point3D(0, 0, 25) });
let game = new Game(context, scene, camera);
let spheres = [];
let green = new Color({ r: 1 });
let white = new Color({ r: 1, g: 1, b: 1 });
let directLight = new DirectionalLight({ color: white, position: new
Point3D(-.24, 3.34, 7.5) });
scene.addLight(directLight);
var positionX = -30;
var positionY = 15;
initSphere();
for (let i = 0; i < spheres.length; i++) {
  let component = new RotateSphere({owner :spheres[i] });
  spheres[i].listComponents.addComponent(component);
}

```

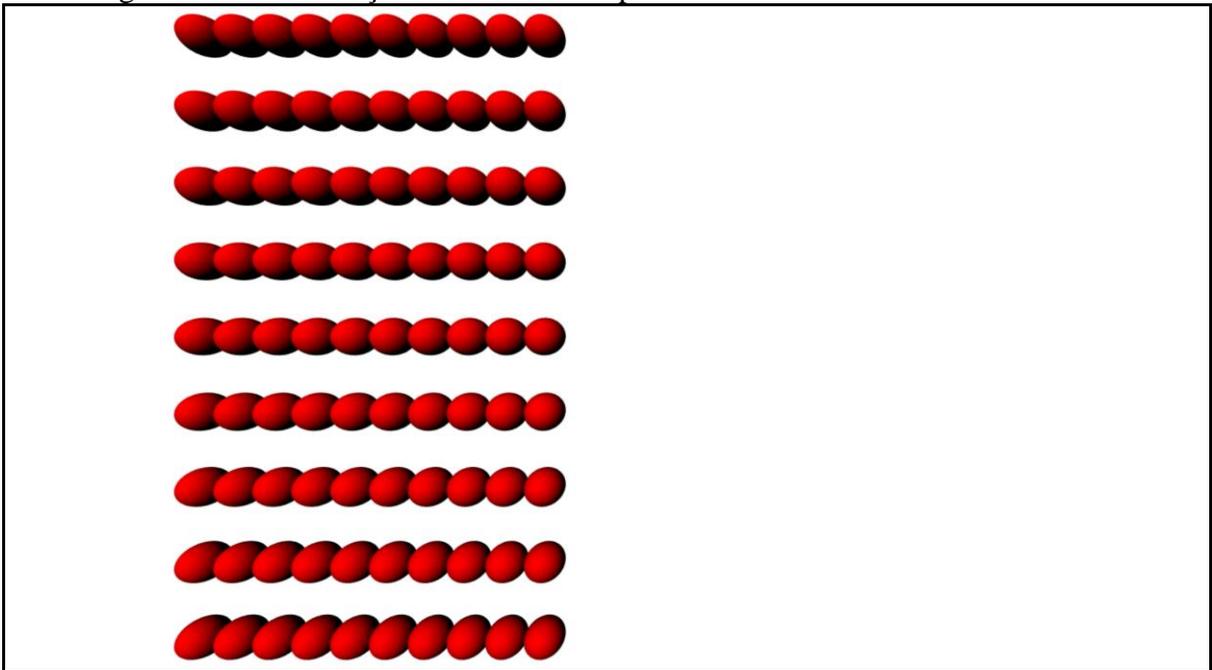
Fonte: elaborado pelo autor.

Figura 20 - Visualização do cenário 1 implementado com o motor desenvolvido



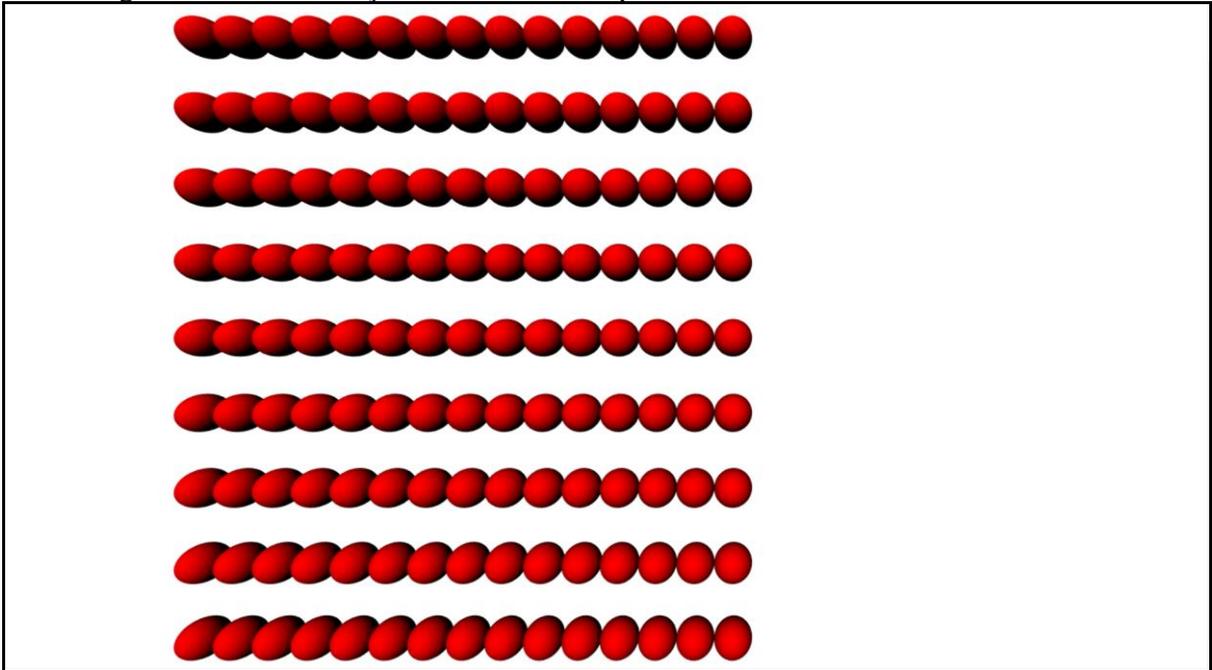
Fonte: elaborado pelo autor.

Figura 21 - Visualização do cenário 2 implementado com o motor desenvolvido



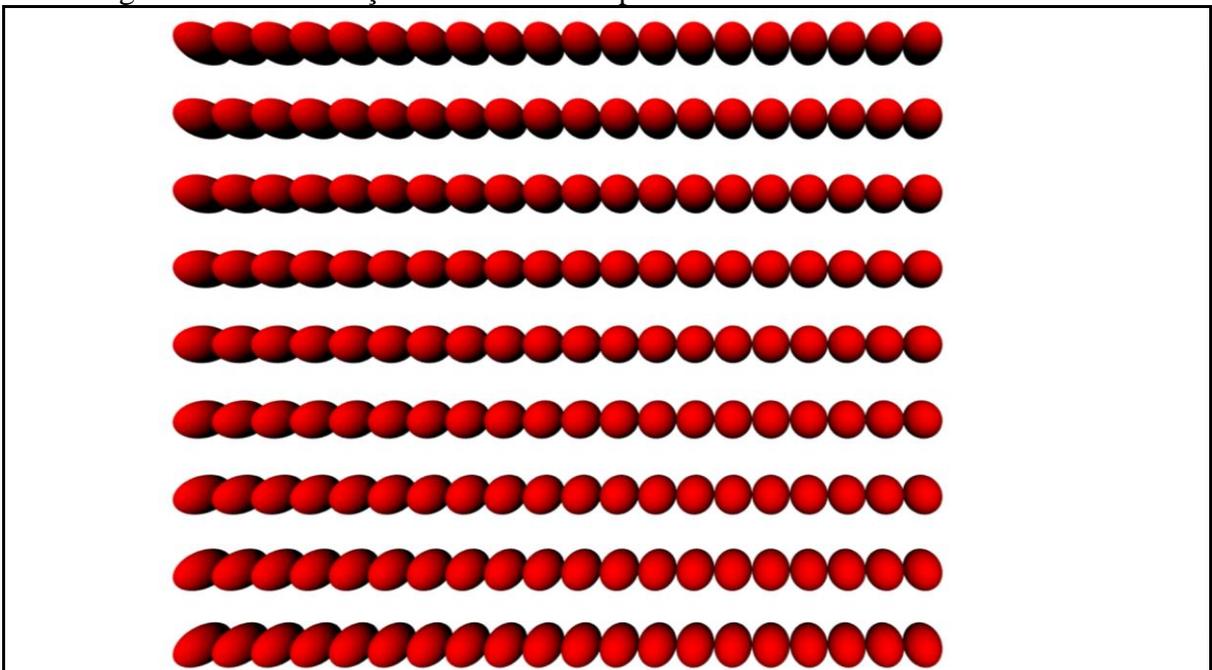
Fonte: elaborado pelo autor.

Figura 22 - Visualização do cenário 3 implementado com o motor desenvolvido



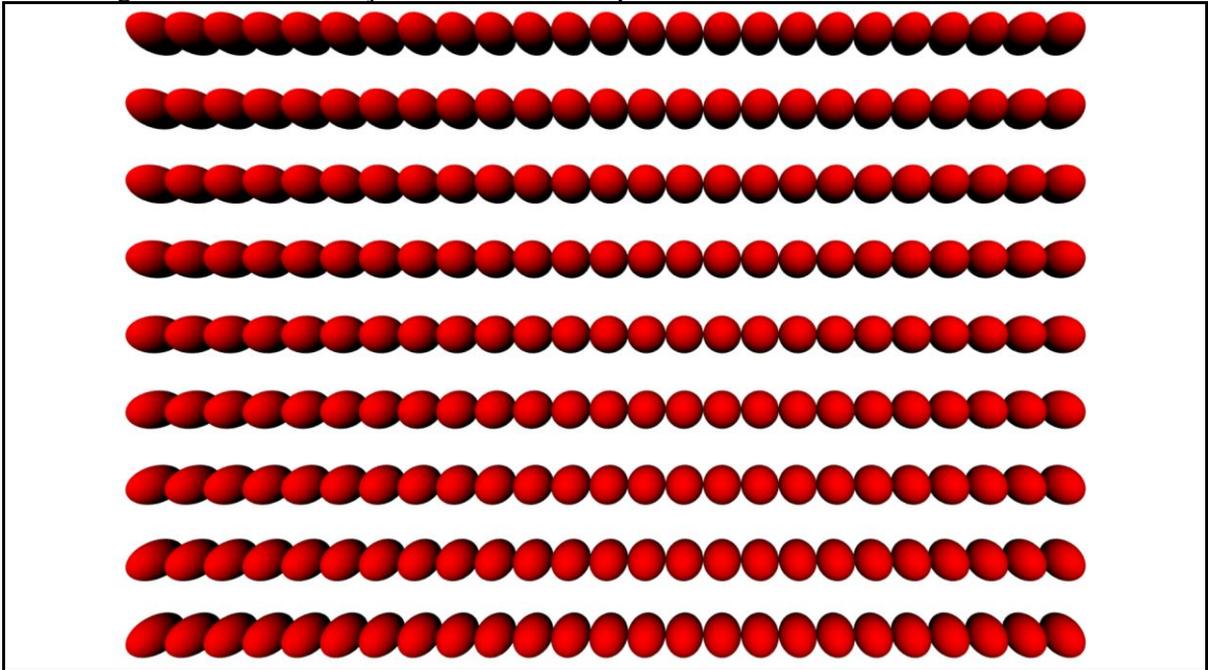
Fonte: elaborado pelo autor.

Figura 23 - Visualização do cenário 4 implementado com o motor desenvolvido



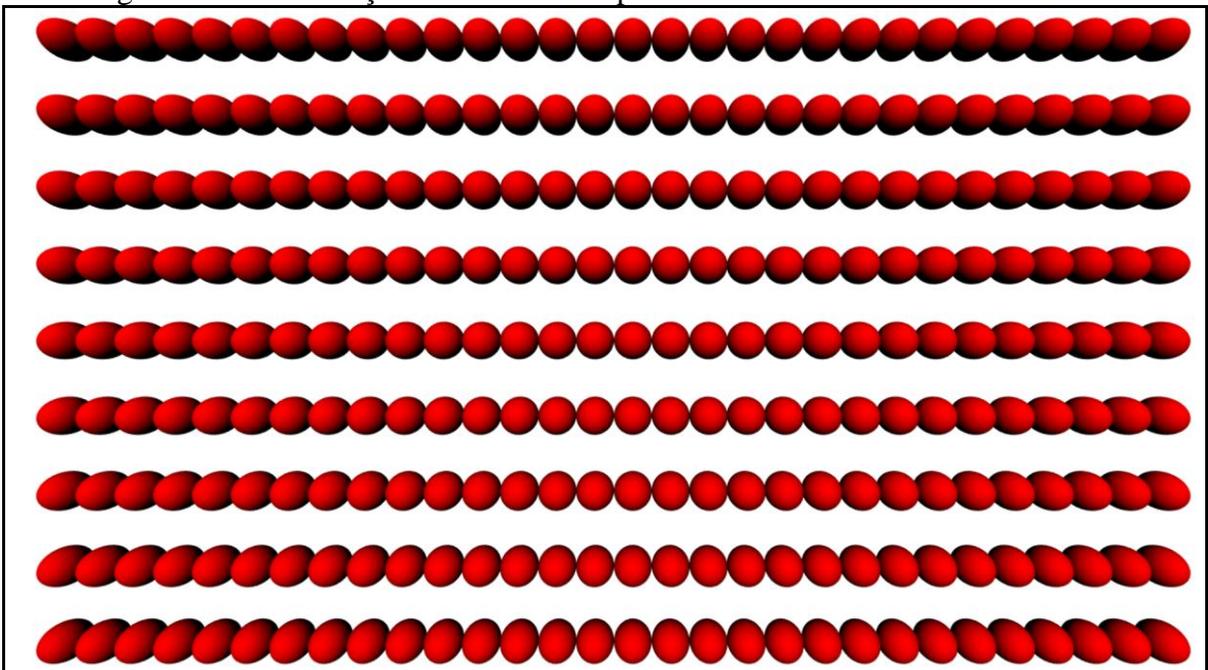
Fonte: elaborado pelo autor.

Figura 24 - Visualização do cenário 5 implementado com o motor desenvolvido



Fonte: elaborado pelo autor.

Figura 25 - Visualização do cenário 6 implementado com o motor desenvolvido



Fonte: elaborado pelo autor.

APÊNDICE B – Cenários implementados usando o motor desenvolvido

Neste apêndice são apresentados os códigos(Quadro 25 ao 30) relativos a cada cenário usado nos testes implementados com Three.js (Figura 26 a 31).

Quadro 26 - Cenário 1 implementado com a Three.js desenvolvido (50 esferas)

```
function initSphres(){
  for (let i = 0; i < 10; i++) {
    for (let index = 0; index < 5; index++) {
      var geometry = new THREE.SphereGeometry(1, 60, 60);
      var material = new THREE.MeshLambertMaterial({ color: 0x00ff00
    });
      var sphere = new THREE.Mesh(geometry, material);
      sphere.position.set(positionX, positionY, 0);
      positionX += 2;
      spheres.push(sphere);
      scene.add(sphere);
    }
    positionY -= 4
    positionX = -30;
  }
};

var spheres = [];

var scene = new THREE.Scene();
var camera = new THREE.PerspectiveCamera(75, window.innerWidth /
window.innerHeight, 0.1, 1000);

var renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);

var positionX = -30;
var positionY = 15;

initSphres();

var light = new THREE.DirectionalLight(0xffffffff, 0.5);
light.position.set(-.24, 3.34, 7.5);
scene.add(light);

camera.position.z = 25;

function animate() {
  requestAnimationFrame(animate);
  for (let index = 0; index < spheres.length; index++) {
    spheres[index].rotation.x += 0.01;
    spheres[index].rotation.y += 0.01;
  }
  renderer.render(scene, camera);
}
animate();
```

Fonte: elaborado pelo autor.

Quadro 27 - Cenário 2 implementado com a Three.js desenvolvido (100 esferas)

```

function initSphres(){
  for (let i = 0; i < 10; i++) {
    for (let index = 0; index < 10; index++) {
      var geometry = new THREE.SphereGeometry(1, 60, 60);
      var material = new THREE.MeshLambertMaterial({ color: 0x00ff00
    });
      var sphere = new THREE.Mesh(geometry, material);
      sphere.position.set(positionX, positionY, 0);
      positionX += 2;
      spheres.push(sphere);
      scene.add(sphere);
    }
    positionY -= 4
    positionX = -30;
  }
};

var spheres = [];

var scene = new THREE.Scene();
var camera = new THREE.PerspectiveCamera(75, window.innerWidth /
window.innerHeight, 0.1, 1000);

var renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);

var positionX = -30;
var positionY = 15;

initSphres();

var light = new THREE.DirectionalLight(0xffffff, 0.5);
light.position.set(-.24, 3.34, 7.5);
scene.add(light);

camera.position.z = 25;

function animate() {
  requestAnimationFrame(animate);
  for (let index = 0; index < spheres.length; index++) {
    spheres[index].rotation.x += 0.01;
    spheres[index].rotation.y += 0.01;
  }
  renderer.render(scene, camera);
}
animate();

```

Fonte: elaborado pelo autor.

Quadro 28 - Cenário 3 implementado com a Three.js desenvolvido (150 esferas)

```

function initSphres(){
  for (let i = 0; i < 10; i++) {
    for (let index = 0; index < 15; index++) {
      var geometry = new THREE.SphereGeometry(1, 60, 60);
      var material = new THREE.MeshLambertMaterial({ color: 0x00ff00
    });
      var sphere = new THREE.Mesh(geometry, material);
      sphere.position.set(positionX, positionY, 0);
      positionX += 2;
      spheres.push(sphere);
      scene.add(sphere);
    }
    positionY -= 4
    positionX = -30;
  }
};

var spheres = [];

var scene = new THREE.Scene();
var camera = new THREE.PerspectiveCamera(75, window.innerWidth /
window.innerHeight, 0.1, 1000);

var renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);

var positionX = -30;
var positionY = 15;

initSphres();

var light = new THREE.DirectionalLight(0xffffff, 0.5);
light.position.set(-.24, 3.34, 7.5);
scene.add(light);

camera.position.z = 25;

function animate() {
  requestAnimationFrame(animate);
  for (let index = 0; index < spheres.length; index++) {
    spheres[index].rotation.x += 0.01;
    spheres[index].rotation.y += 0.01;
  }
  renderer.render(scene, camera);
}
animate();

```

Fonte: elaborado pelo autor.

Quadro 29 - Cenário 4 implementado com a Three.js desenvolvido (200 esferas)

```

function initSphres(){
  for (let i = 0; i < 10; i++) {
    for (let index = 0; index < 20; index++) {
      var geometry = new THREE.SphereGeometry(1, 60, 60);
      var material = new THREE.MeshLambertMaterial({ color: 0x00ff00
    });
      var sphere = new THREE.Mesh(geometry, material);
      sphere.position.set(positionX, positionY, 0);
      positionX += 2;
      spheres.push(sphere);
      scene.add(sphere);
    }
    positionY -= 4
    positionX = -30;
  }
};

var spheres = [];

var scene = new THREE.Scene();
var camera = new THREE.PerspectiveCamera(75, window.innerWidth /
window.innerHeight, 0.1, 1000);

var renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);

var positionX = -30;
var positionY = 15;

initSphres();

var light = new THREE.DirectionalLight(0xffffff, 0.5);
light.position.set(-.24, 3.34, 7.5);
scene.add(light);

camera.position.z = 25;

function animate() {
  requestAnimationFrame(animate);
  for (let index = 0; index < spheres.length; index++) {
    spheres[index].rotation.x += 0.01;
    spheres[index].rotation.y += 0.01;
  }
  renderer.render(scene, camera);
}
animate();

```

Fonte: elaborado pelo autor.

Quadro 30 - Cenário 5 implementado com a Three.js desenvolvido (250 esferas)

```

function initSphres(){
  for (let i = 0; i < 10; i++) {
    for (let index = 0; index < 25; index++) {
      var geometry = new THREE.SphereGeometry(1, 60, 60);
      var material = new THREE.MeshLambertMaterial({ color: 0x00ff00
    });
      var sphere = new THREE.Mesh(geometry, material);
      sphere.position.set(positionX, positionY, 0);
      positionX += 2;
      spheres.push(sphere);
      scene.add(sphere);
    }
    positionY -= 4
    positionX = -30;
  }
};

var spheres = [];

var scene = new THREE.Scene();
var camera = new THREE.PerspectiveCamera(75, window.innerWidth /
window.innerHeight, 0.1, 1000);

var renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);

var positionX = -30;
var positionY = 15;

initSphres();

var light = new THREE.DirectionalLight(0xffffff, 0.5);
light.position.set(-.24, 3.34, 7.5);
scene.add(light);

camera.position.z = 25;

function animate() {
  requestAnimationFrame(animate);
  for (let index = 0; index < spheres.length; index++) {
    spheres[index].rotation.x += 0.01;
    spheres[index].rotation.y += 0.01;
  }
  renderer.render(scene, camera);
}
animate();

```

Fonte: elaborado pelo autor.

Quadro 31 - Cenário 2 implementado com a Three.js desenvolvido (300 esferas)

```

function initSphres(){
  for (let i = 0; i < 10; i++) {
    for (let index = 0; index < 30; index++) {
      var geometry = new THREE.SphereGeometry(1, 60, 60);
      var material = new THREE.MeshLambertMaterial({ color: 0x00ff00
    });
      var sphere = new THREE.Mesh(geometry, material);
      sphere.position.set(positionX, positionY, 0);
      positionX += 2;
      spheres.push(sphere);
      scene.add(sphere);
    }
    positionY -= 4
    positionX = -30;
  }
};

var spheres = [];

var scene = new THREE.Scene();
var camera = new THREE.PerspectiveCamera(75, window.innerWidth /
window.innerHeight, 0.1, 1000);

var renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);

var positionX = -30;
var positionY = 15;

initSphres();

var light = new THREE.DirectionalLight(0xffffff, 0.5);
light.position.set(-.24, 3.34, 7.5);
scene.add(light);

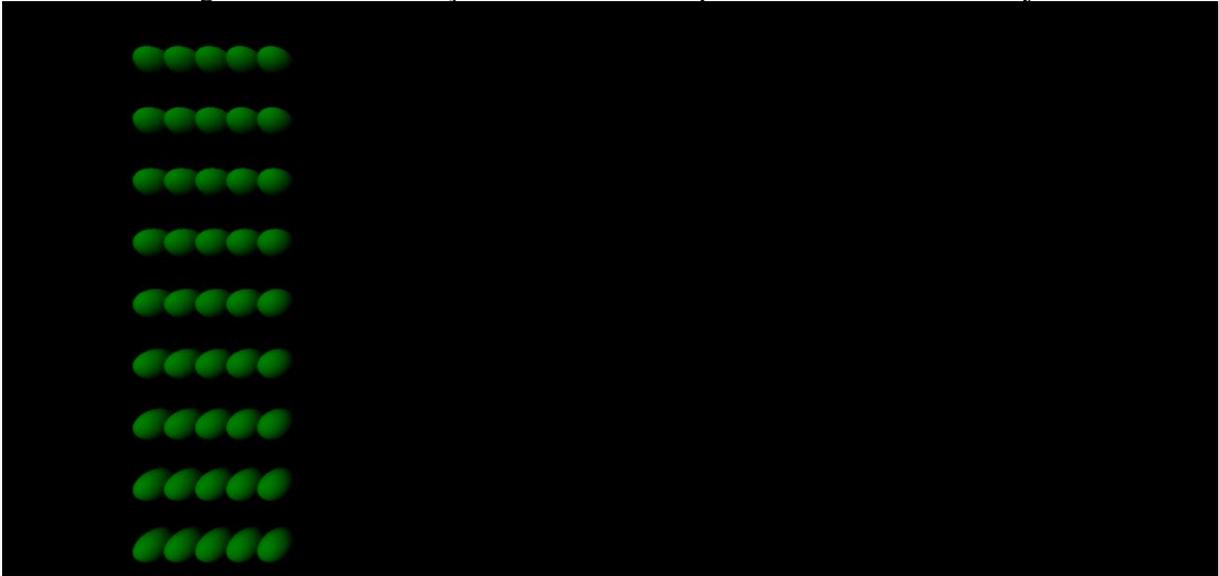
camera.position.z = 25;

function animate() {
  requestAnimationFrame(animate);
  for (let index = 0; index < spheres.length; index++) {
    spheres[index].rotation.x += 0.01;
    spheres[index].rotation.y += 0.01;
  }
  renderer.render(scene, camera);
}
animate();

```

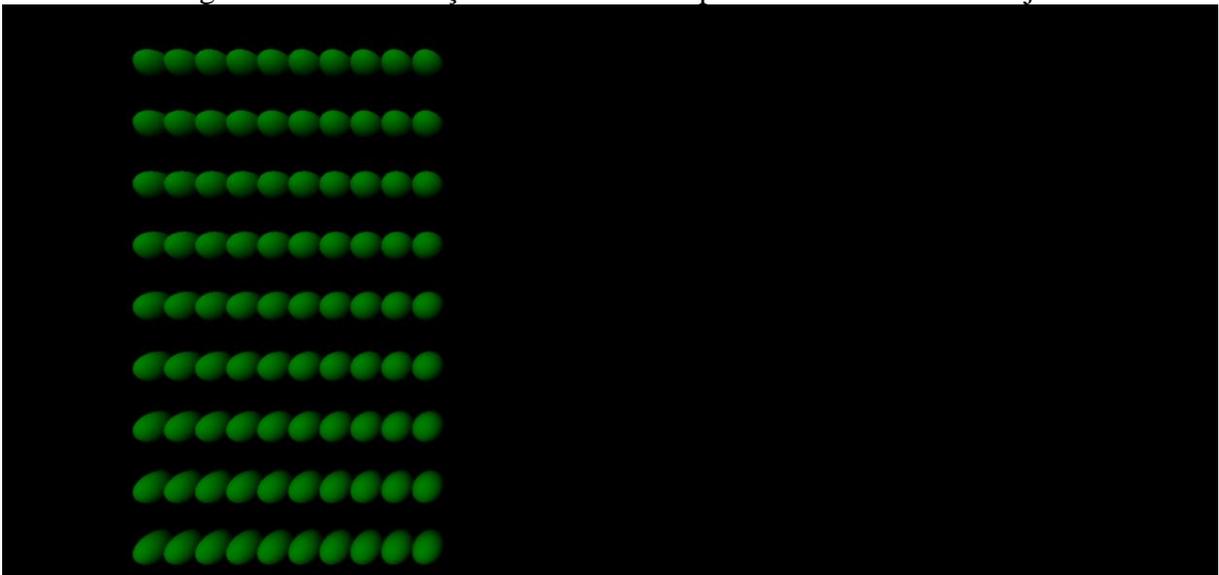
Fonte: elaborado pelo autor.

Figura 26 - Visualização do cenário 1 implementado com a Three.js



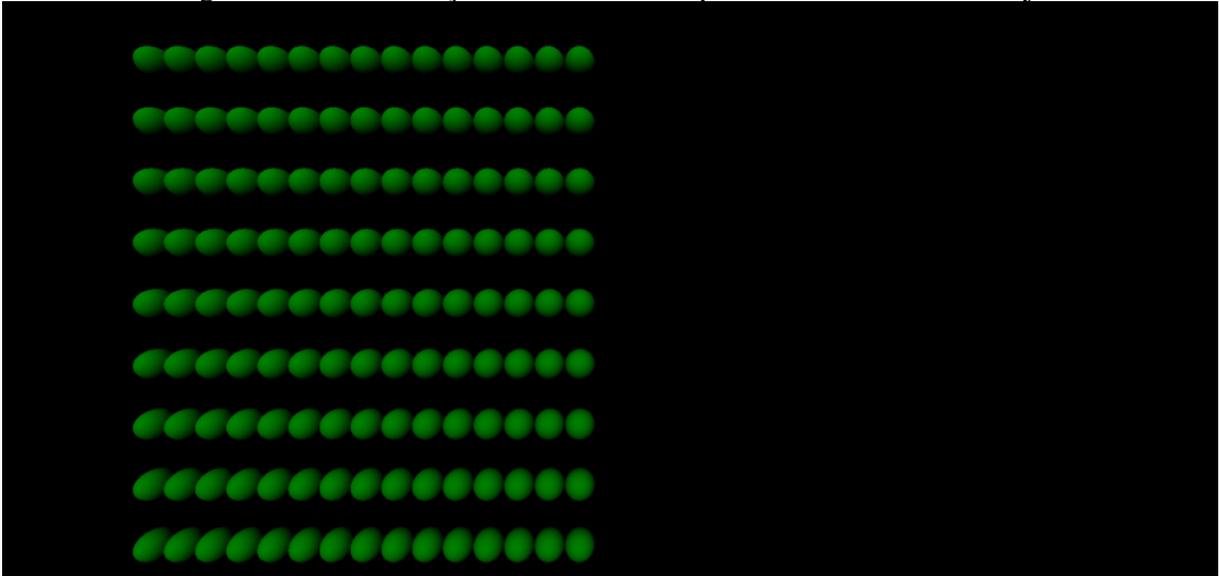
Fonte: elaborado pelo autor.

Figura 27 - Visualização do cenário 2 implementado com a Three.js



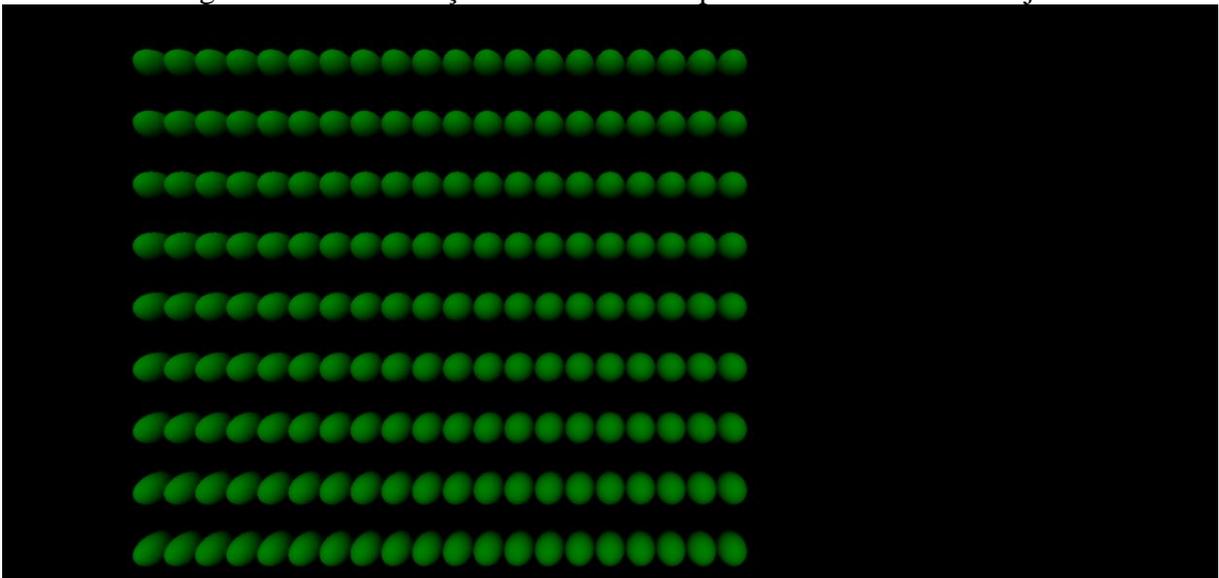
Fonte: elaborado pelo autor.

Figura 28 - Visualização do cenário 3 implementado com a Three.js



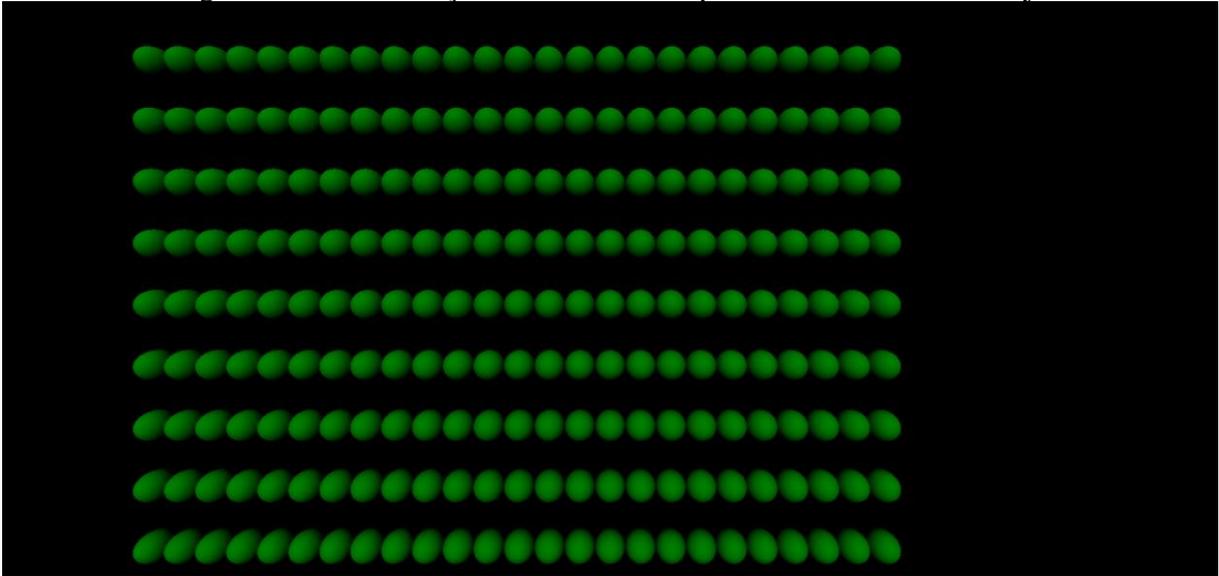
Fonte: elaborado pelo autor.

Figura 29 - Visualização do cenário 4 implementado com a Three.js



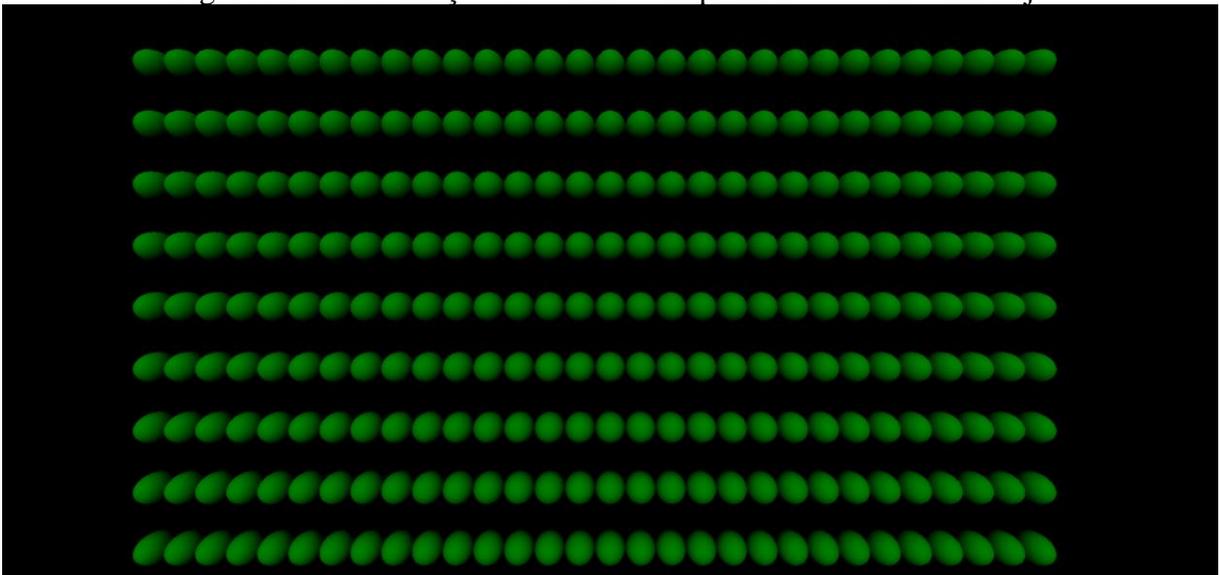
Fonte: elaborado pelo autor.

Figura 30 - Visualização do cenário 5 implementado com a Three.js



Fonte: elaborado pelo autor.

Figura 31 - Visualização do cenário 6 implementado com a Three.js

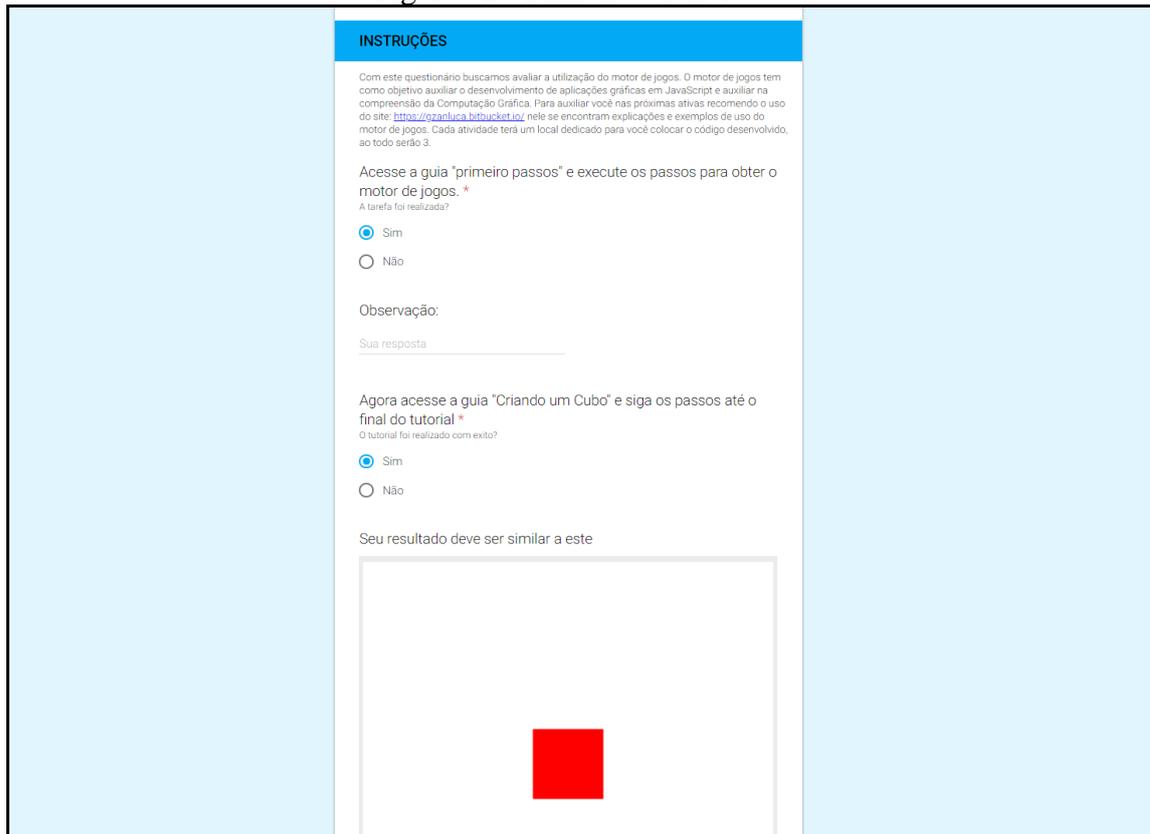


Fonte: elaborado pelo autor.

Apêndice C – Enunciado das atividades

Neste apêndice são apresentados os enunciados das atividades solicitadas no teste de usabilidade do motor desenvolvido. A Figura 32 corresponde a primeira atividade. Consistia em obter o motor de jogos desenvolvido e seguir o primeiro tutorial para criar uma cena com um cubo.

Figura 32 - Primeiro enunciado.



INSTRUÇÕES

Com este questionário buscamos avaliar a utilização do motor de jogos. O motor de jogos tem como objetivo auxiliar o desenvolvimento de aplicações gráficas em JavaScript e auxiliar na compreensão da Computação Gráfica. Para auxiliar você nas próximas ativas recomendo o uso do site: <https://grafica.tobuCKET.io/>, nele se encontram explicações e exemplos de uso do motor de jogos. Cada atividade terá um local dedicado para você colocar o código desenvolvido, ao todo serão 3.

Acesse a guia "primeiro passos" e execute os passos para obter o motor de jogos. *

A tarefa foi realizada?

Sim

Não

Observação:

Sua resposta

Agora acesse a guia "Criando um Cubo" e siga os passos até o final do tutorial *

O tutorial foi realizado com sucesso?

Sim

Não

Seu resultado deve ser similar a este



Fonte: elaborado pelo autor.

A Figura 33 e Figura 34 consistem na segunda atividade. Ela consistia em criar-se uma esfera como filha do cubo e fazer com que o cubo rotacionasse utilizando um componente. Por fim, pedia-se que se move a câmera para as posições indicadas na Figura 34.

Figura 33 - Primeira parte da segunda atividade

Instruções

Agora nessa atividade você irá usar duas funções do motor de jogos que são: o grafo de cena e a adição de luz em cena. Para isso crie uma cena com um cubo tendo como filho uma esfera e adicione um dos tipos de luzes disponível no motor de jogos a cena, e por fim rotacione apenas o cubo a cada frame.

Resultado Esperado



Você conseguiu criar o grafo de cena? *

Sim
 Não

Você conseguiu adicionar e visualizar a incidência da luz em cena? *

Sim
 Não

Qual tipo de luz você adicionou? *

DirectionalLight
 PointLight
 SpotLight

Consegui rotacionar o cubo? *

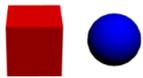
Sim
 Não

Fonte: elaborado pelo autor.

Figura 34 - Segunda parte da segunda atividade

Agora para finalizar posicione a câmera em x=5, y= 8 e z= 15.

Resultado final esperado



Sua cena está similar a apresentada na foto *

Sim
 Não

Você teve alguma dificuldade em realizar essa tarefa? Se sim qual?

Sua resposta

Código fonte (HTML)

Sua resposta

Código fonte (JavaScript)

Fonte: elaborado pelo autor.

A terceira e última atividade consistia em criar um componente para receber a entrada do teclado e com ela fizesse a câmera mudar de posição. Na pode-se observar o enunciado dessa atividade.

Figura 35 - Enunciado da terceira atividade

TOWELJS: ENGINE 3D EM JAVASCRIPT USANDO ARQUITETURA BASEADA EM COMPONENTES

*Obrigatório

Instrução

Como última atividade crie uma cena (pode utilizar uma das anteriores) e um componente para mover a câmera usando as setas do teclado. Dica, os códigos das setas são: 37 (esquerda), 38 (cima), 39 (direita), 40 (baixo). E use apenas os métodos onKeyDown ou onKeyUp do componente, pois o navegador não captura as setas no evento keypress. E também não esqueça que para ter uma sensação de movimento completo da cena você precisa alterar a mesma quantidade o valor da posição da câmera e o alvo para onde ela está olhando (target).

Você conseguiu criar o componente? *

Sim

Não

Você conseguiu usar algum dos eventos do teclado para mover a câmera? *

Sim

Não

Código fonte (HTML)

Sua resposta

Código fonte (JavaScript)

Sua resposta

 Página 4 de 5

Nunca envie senhas pelo Formulários Google.

Fonte: elaborado pelo autor.

Apêndice D – Tutorial criado para explicar a utilização do motor de jogos desenvolvido

Neste apêndice é apresentado duas telas do tutorial feito para auxiliar no uso do motor de jogos desenvolvido por outros desenvolvedores. A Figura 36 é a tela inicial do tutorial. É nela que explica como obter-se o motor e quais são outros itens necessários para sua utilização. As Figura 37 e Figura 38 fazem parte do mesmo tutorial explicando a construção de uma cena passo-a-passo.

Figura 36 - Instruções para obter-se o motor de jogos

Exemplos	Primeiros Passos
Primeiros passos Criando um cubo Criando uma esfera Usando transformações Adicionando luz na cena Criando um componente Grafo de cena Câmeras	<p>Requisitos:</p> <p>Node.js</p> <p>Usando o motor de jogos</p> <p>Para começar a usar a biblioteca você deve baixa-la através desse link aqui</p> <p>Após efetuar o download descompacte a pasta em algum diretório e execute o comando a seguir dentro da pasta raiz:</p> <pre>npm install</pre> <p>Da pasta dentro do diretório raiz a que mais importa nesse momento é a pasta <code>.app</code> e a pasta <code>.build</code>. Na pasta <code>app</code> é onde por padrão ficam os códigos da aplicação e a na pasta <code>build</code> encontra-se o código após passado pelo transpilador (webpack). Outro ponto importante é o arquivo <code>webpack.config.js</code>, é nele que deve indicar a localização de todo o código que você produziu para sua aplicação para que depois use o comando:</p> <pre>npm run webpack</pre> <p>Coloque os arquivos gerados na sua página HTML. Fique tranquilo essa parte será melhor esclarecida nos próximos tutoriais.</p>

Fonte: elaborado pelo autor.

Figura 37 - Primeira parte da explicação da criação de uma cena

Exemplos	Criando um Cubo
Primeiros passos Criando um cubo Criando uma esfera Usando transformações Adicionando luz na cena Criando um componente Grafo de cena Câmeras	<p>A primeira coisa a ser feita é declarar um elemento de <code><canvas></code> na sua página HTML. Ele será responsável por renderizar a sua aplicação.</p> <pre><canvas id="myCanvas" width="640" height="640"></pre> <p>Depois disso no seu código javascript importe os seguintes arquivos:</p> <pre>import { Scene } from "../game/Scene"; import { Point3D } from "../geometric/Point3D"; import { PerspectiveCamera } from "../game/PerspectiveCamera"; import { Game } from "../game/Game"; import { Color } from "../geometric/Color"; import { CubeGameObject } from "../gameObject/CubeGameObject";</pre> <p>Após isso recupere o elemento canvas:</p> <pre>let canvas = document.getElementById("myCanvas");</pre> <p>E defina qual tipo de contexto você quer:</p> <pre>let context = canvas.getContext("webgl");</pre> <p>Com esses arquivos importados e essas definições comece criando uma cena:</p> <pre>let scene = new Scene();</pre> <p>Depois crie uma câmera. Nesse tutorial optou-se por uma câmera em perspectiva porque ela se assemelha mais com a visão humana.</p> <pre>let camera = new PerspectiveCamera({near: 0.1, far : 500, aspect : 1, fovy : 45 * Math.PI / 180, position : new Point3D(0, 0, 15)});</pre> <p>Com tudo isso pronto você pode instanciar um Game. Ele é a parte principal do motor de jogos e você terá apenas um por aplicação.</p> <pre>let game = new Game(context, scene, camera);</pre> <p>Tudo que foi visto até esse ponto e basicamente comum para qualquer aplicação que você irá fazer utilizando esse motor de jogos. Daqui para frente será mostrado a criação do cubo.</p> <p>Para se criar um cubo você deve instanciar um objeto da classe <code>CubeGameObject</code>. No construtor dessa classe você pode ou não informar a cor (color) e a posição (position). Caso não sejam informados será atribuído valores padrão, que para cor será preto e para a posição (0,0,0). No exemplo será criado um cubo vermelho na posição (0,0,0).</p>

Fonte: elaborado pelo autor.

Figura 38 - Segunda parte da explicação da criação de uma cena

Criando a cor:

```
let red = new Color({r : 1});
```

Criando o cubo:

```
let cube = new CubeGameObject({color : red});
```

Para que você visualize o cubo basta por fim adicionar ele na cena:

```
scene.addGameObject(cube);
```

Agora com código pronto basta indicar o caminho dele no arquivo **webpack.config.js** dessa forma:

```
entry : {  
  ...  
  nomeDoArquivo : "./app/nomeDoArquivo.js",  
  ...  
}
```

Com isso feito execute o comando:

```
npm run webpack
```

E por fim coloque na sua página HTML o arquivo no formato **nomeDoArquivo.bundle.js** localizado na pasta **./build**.

Não esqueça que o caminho para a baste e relativo onde você colocou sua página HTML. Esse tutorial pressupõem que as pastas **./app** e **./build** estão no mesmo nível que sua página HTML.

Outro detalhe a se observar é o fato que o a tag `<script src="./build/nomeDoArquivo.bundle.js"></script>` deve estar após a declaração do canvas, para que assim dê tempo de o canvas carregar antes de tentar recupera-lo

```
<!DOCTYPE html>  
<html>  
<head>  
<title>Criando um cubo</title>  
</head>  
<body>  
  ...  
<canvas id="glCanvas" width="640" height="640"></canvas>  
<script src="./build/cube.bundle.js"></script>  
</body>  
</html>
```

Fonte: elaborado pelo autor.