

PONTIFÍCIA UNIVERSIDADE CATÓLICA DE GOIÁS
ESCOLA DE CIÊNCIAS EXATAS E DA COMPUTAÇÃO
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO



APLICAÇÃO MOBILE PARA SIMULAÇÕES COM PARTÍCULAS

MAURÍCIO JOSÉ BENIGNO DE ALMEIDA SOARES

GOIÂNIA

2018

MAURÍCIO JOSÉ BENIGNO DE ALMEIDA SOARES

APLICAÇÃO MOBILE PARA SIMULAÇÕES COM PARTÍCULAS

Trabalho de conclusão de curso apresentado à
Escola de Ciências Exatas e da Computação da
Pontifícia Universidade Católica de Goiás, como
parte dos requisitos para obtenção do título de
Bacharel em Ciência da Computação.

Orientador:

Prof. Me. Fabio Gomes de Assunção

Banca examinadora:

Prof. Me. Ana Flavia Marinho de Lima

Garrote

Prof. Me. Rafael Leal Martins

GOIÂNIA
2018

MAURÍCIO JOSÉ BENIGNO DE ALMEIDA SOARES

APLICAÇÃO MOBILE PARA SIMULAÇÕES COM PARTÍCULAS

Trabalho de Conclusão de Curso aprovado em sua forma final pela Escola de Ciências Exatas e da Computação, da Pontifícia Universidade Católica de Goiás, para obtenção do título de Bacharel em Ciência da Computação, em ____/____/____.

Orientador: Me. Fábio Gomes de Assunção

Prof. Me. Ludmilla Reis Pinheiro dos Santos
Coordenadora de Trabalho de Conclusão de Curso

GOIÂNIA
2018

A Deus pela minha vida e oportunidades.
A minha família, amigos e professores por toda lição, apoio e
críticas construtivas.
A todos que tiveram paciência e disponibilidade em me ajudar
e aconselhar.

AGRADECIMENTOS

Ao Professor Me. Fabio Gomes de Assunção, orientador acadêmico, por me aceitar como seu orientando e me guiar não só durante a produção deste, mas ao longo de quase todo o meu processo de formação.

Ao meu estimado amigo Gabriel Dias Vilela, que sempre se prontificou a me ajudar em meio às dúvidas acadêmicas e na elaboração deste trabalho.

Ao meu primo Rodrigo Ferreira de Almeida Wallauer, iniciamos o curso juntos, estudamos juntos, passamos pelos momentos complicados do curso juntos, sempre apoiando um ao outro de forma em que nos mantivemos no caminho.

A todos que tiveram participação na elaboração deste trabalho, seja ela direta ou indiretamente.

“Imaginação, muitas vezes, leva-nos a mundos que nunca existiram. Mas sem ela não vamos a lugar nenhum.”

Carl Sagan

RESUMO

Apresenta-se um estudo e desenvolvimento sobre Sistema de Partículas para o ambiente *mobile*, que faz parte de um campo de pesquisa importante para a Computação Gráfica. Neste trabalho é abordado um breve histórico sobre a trajetória da computação gráfica e alguns exemplos de seu uso em diversos segmentos. Também é apresentada uma introdução a API OpenGL ES, um pouco sobre suas versões e detalhes sobre o pipeline de execução e seus programas de shader. Não menos importante, também são mostradas algumas aplicações de Sistemas de Partículas no contexto de simulação de fluídos e da realidade aumentada. Por fim, o sistema de partículas desenvolvido é utilizado para simular efeitos especiais simples, com o intuito de mostrar seu funcionamento e estimular as pesquisas sobre partículas voltadas para dispositivos móveis.

Palavras-chaves: *Simulação, Sistema de Partículas, Aplicação Mobile, Efeitos especiais.*

ABSTRACT

It presents a study and development on Particle System for the mobile environment, which is part of an important research field for Computer Graphics. In this work a brief history about the path of computer graphics and some examples of its use in several segments is discussed. Also introduced is an introduction to the API OpenGL ES, a bit about its versions and details about the execution pipeline and its shader programs. Not least, some applications of Particle Systems in the context of fluid simulation and augmented reality are also shown. Finally, the developed particle system is used to simulate simple special effects, to show its operation and stimulate research on particles targeted to mobile devices.

Keywords: *Simulation, Particle System, Mobile Application, Special Effects.*

LISTA DE IMAGENS

FIGURA 1.1 – SEGMENTOS ONDE A CG ESTÁ PRESENTE	13
FIGURA 2.1 – MENU SIMCMED	18
FIGURA 2.2 – DEMONSTRANDO INTERAÇÃO COM PACIENTE VIRTUAL.....	19
FIGURA 2.3 – DEMONSTRANDO INSTRUMENTAÇÃO CIRURGICA.	19
FIGURA 2.4 – FILME: AS AVENTURAS PI (2012).....	20
FIGURA 2.5 – FILME: PLANETA DOS MACACOS: A ORIGEM (2011).....	21
FIGURA 2.6 – FILME: WOLVERINE: IMORTAL	21
FIGURA 2.7 – FILME: LIGA DA JUSTIÇA (2017).....	22
FIGURA 2.8 – BANQUETA 3D MODELADA A PARTIR DE UM MODELO 2D	23
FIGURA 2.9 – MODELO DE BANQUETA 2D USANDO PRIMITIVAS GEOMÉTRICAS.....	23
FIGURA 2.10 – TELA DO UNITY 3D COM UM PROJETO DE JOGO 2D.....	24
FIGURA 2.11 – PLATAFORMAS SUPORTADAS PELO UNITY	25
FIGURA 3.1 – VERSÃO SIMPLIFICADA DO PIPELINE DO OPENGL.....	27
FIGURA 3.2 – GRÁFICO DE IMPLEMENTAÇÃO DE API GRAFICO 3D.....	28
FIGURA 3.3 - TIPOS DE DADOS GSLS.....	31
FIGURA 3.4 - OPERADORES GLSL	32
FIGURA 3.5 CONTINUAÇÃO OPERADORES GLSL	32
FIGURA 4.1 – GENESIS EFFECT.....	33
FIGURA 4.2 – FOGOS DE ARTIFÍCIO COLORIDOS	34
FIGURA 5.1 – USO DE SISTEMA DE PARTÍCULAS EM REALIDADE AUMENTADA	36
FIGURA 5.2 - SIMULAÇÃO NO BLENDER 3D COM REPRESENTAÇÃO EM MULTI-LEVEL PARTITION OF UNITY IMPLICITS.....	37
FIGURA 5.3 - SIMULAÇÃO EM OPENGL COM REPRESENTAÇÃO EM MARCHING CUBES	37
FIGURA 6.1 – INFORMAÇÕES SOBRE O APARELHO	38
FIGURA 6.2 – INFORMAÇÕES SOBRE O SISTEMA DO APARELHO.....	39
FIGURA 6.3 – SELEÇÃO DE COMPONENTES - ANDROID STUDIO.....	96
FIGURA 6.4 – TELA DE CONFIGURAÇÃO DE COMPONENTES	97
FIGURA 6.5 – TELA DE BOAS-VINDAS - ANDROID STUDIO	97
FIGURA 6.6 – MENU DE OPÇÕES DA TELA DE BOAS-VINDAS.....	98
FIGURA 6.7 – CONFIGURAÇÕES DO SDK MANAGER	98
FIGURA 6.8 – MENU TOOLS.....	99
FIGURA 6.9 – CRIAÇÃO DE CONFIGURAÇÃO DE DISPOSITIVOS VIRTUAIS ANDROID	99
FIGURA 6.10 – CLASSES EXEMPLO.....	40
FIGURA 6.11 – CLASSE PARTÍCULA BÁSICA.....	41
FIGURA 6.12 – CLASSE PARTÍCULA COMPOSTA	41
FIGURA 6.13 – PARTÍCULA DE CHUVA.....	42
FIGURA 6.14 – MÉTODO CONSTRUTOR SISTEMAPARTICULAS	42

FIGURA 6.15 – MÉTODO DE ADIÇÃO DE PARTÍCULAS.....	43
FIGURA 6.16 – MÉTODO BINDDATA.....	43
FIGURA 6.17 – ARQUIVO VERTEX SHADER.....	44
FIGURA 6.18 – ARQUIVO FRAGMENT SHADER.....	45
FIGURA 6.19 – MÉTODO CONSTRUTOR DE PARTICLESHADERPROGRAM.....	45
FIGURA 6.20 – CONFIGURAÇÃO DA SUPERFÍCIE DE RENDERIZAÇÃO.....	46
FIGURA 6.21 – CONFIGURAÇÃO DO SISTEMA DE PARTÍCULAS.....	47
FIGURA 7.1 – MENU INICIAL DO APLICATIVO.....	48
FIGURA 7.2 – DICA PARA MUDAR EFEITO	49
FIGURA 7.3 – PIPELINE DE FUNCIONAMENTO APLICATIVO	50
FIGURA 7.4 – EFEITO CHUVA.....	51
FIGURA 7.5 – PARTICULARIDADE NO VERTEX SHADER DO EFEITO CHUVA	52
FIGURA 7.6 – DEFINIÇÃO DE SHADERS, DIREÇÃO E QUANTIDADE DE PARTÍCULAS	52
FIGURA 7.7- ALTERAÇÃO NO MÉTODO DE ADIÇÃO DE PARTÍCULAS DE FAÍSCA.....	53
FIGURA 7.8 - SIMULAÇÃO EFEITO DE FAÍSCA	53
FIGURA 7.9 - CÓDIGO DE INTERATIVIDADE DO EFEITO FAÍSCA	54
FIGURA 7.10 – ARQUIVO VERTEX SHADER DO EFEITO FAÍSCA.....	54
FIGURA 7.11 – ARQUIVO FRAGMENT SHADER DO EFEITO FAÍSCA	54
FIGURA 7.12 - SIMULAÇÃO EFEITO DE FOGO.....	55
FIGURA 7.13 - APLICAÇÃO DE CONFIGURAÇÕES INICIAIS, COMO COR, ANGULO E DIREÇÃO	56
FIGURA 7.14 - CÓDIGO PARA DIMINUIÇÃO DE PARTÍCULAS COM O TEMPO	56
FIGURA 7.15 - SIMULAÇÃO EFEITO DE FOGOS DE ARTIFÍCIO	57
FIGURA 7.16 - CÓDIGO DE GERAÇÃO RANDÔMICA DE COR RGB.....	57
FIGURA 7.17 - SIMULAÇÃO EFEITO DE FONTE	59
FIGURA 7.18 – EFEITO NEVE.....	60
FIGURA 7.19 – PARTICULARIDADE NO MÉTODO DE ADIÇÃO DE PARTÍCULAS	60

LISTA DE SIGLAS

2D	Bidimensional
3D	Tridimensional
AEP	Android Extension Pack
API	Application Programming Interface
CAD	Computer Aided Design
CG	Computação Gráfica
GLSL	OpenGL Shading Language
GLU	OpenGL Utility Library
GPU	Graphics Processing Unit
GUI	Graphical User Interface
IDE	Integrated Development Enviroment
OpenGL	Open Graphics Library
OpenGL ES	Open Graphics Library Embedded Systems
SimCMed	Simulador 3D de Cirurgias Medicas
RA	Realidade Aumentada
RV	Realidade Virtual
SO	Sistema Operacional
SP	Sistema de Partículas

SUMÁRIO

1. INTRODUÇÃO.....	13
1.1. Justificativa	15
1.2. Objetivos	16
1.2.1. Objetivo Geral.....	16
1.2.2. Objetivos Específicos	16
1.3. Metodologia	16
2. COMPUTAÇÃO GRÁFICA	17
2.1. Histórico da computação gráfica	17
2.2. Uso da computação gráfica na medicina	18
2.3. Uso da computação gráfica no cinema	20
2.4. Uso da computação gráfica na engenharia	22
2.5. Uso da computação gráfica em jogos	24
2.6. Uso da computação gráfica em dispositivos móveis	25
3. OPENGL ES	26
3.1. API OpenGL.....	26
3.2. Funcionamento API OpenGL	26
3.3. API OpenGL ES	27
3.4. API OpenGL ES 1.X.....	28
3.5. API OpenGL ES 2.X.....	28
3.6. API OpenGL ES 3.X.....	29
3.7. Shaders OpenGL ES	29
3.8. Vertex Shader	30
3.9. Fragment Shader.....	30
3.10. Shading Language	30
4. SISTEMAS DE PARTÍCULAS	33
4.1. O que são sistemas de partículas?.....	33
4.2. História dos sistemas de partículas	33
4.3. Composição	34
5. PESQUISAS QUE ENVOLVEM SISTEMAS DE PARTÍCULAS	36
5.1. Aplicação em Realidade Aumentada	36
5.2. Aplicação em simulação de fluídos.....	37

6.	SIMULAÇÃO DE PARTÍCULAS	38
6.1.	Ambiente de simulação	38
6.1.1.	Lenovo Vibe k6.....	38
6.2.	Preparação do ambiente de desenvolvimento e simulação.....	39
6.3.	Estrutura de uso do OpenGL ES.....	39
6.4.	Partícula	40
6.5.	Sistema de Partículas JAVA.....	42
6.6.	Arquivos de Shader	44
6.7.	Estrutura de Renderização.....	46
7.	RESULTADOS	48
7.1.	Aplicativo MBParticles.....	48
7.2.	Efeito Chuva.....	50
7.3.	Efeito Faísca	52
7.4.	Efeito Fogo.....	55
7.5.	Efeito Fogos de Artifício.....	56
7.6.	Efeito Fonte	58
7.7.	Efeito Neve	59
8.	PROBLEMAS E LIMITAÇÕES	61
8.1.	Efeitos simultâneos	61
8.2.	Enviar dados parametrizados para o shader	61
8.3.	Algoritmos não funcionaram no Shading.....	61
8.4.	Recuperar dados do Shading para o Java.....	62
9.	CONCLUSÃO	63
	REFERÊNCIAS.....	64
	APÊNDICES.....	67
	A – ACTIVITY.....	67
	A.1 Classe MainActivity (JAVA).....	67
	A.2 Classe ParticulasActivity (JAVA).....	69
	B – EFEITOS.....	71
	B.1 Classe Chuva (JAVA)	71
	B.2 Shader Chuva (GLSL).....	72
	B.3 Classe Faísca (JAVA)	73
	B.4 Shader Faísca (GLSL).....	74

B.5 Classe Fogo (JAVA)	75
B.6 Shader Fogo (GLSL).....	77
B.7 Classe Fogos (JAVA).....	78
B.10 Shader Neve (GLSL)	82
B.11 Particle Fragment Shader Com Tempo (GLSL)	82
B.12 Particle Fragment Shader sem Tempo (GLSL).....	82
C – LINK ENTRE JAVA E GPU.....	83
C.1 Classe ShaderProgram (JAVA)	83
C.2 Classe ParticleShaderProgram (JAVA)	83
D – SISTEMA DE PARTÍCULAS.....	85
D.1 Classe Particulas (JAVA)	85
D.2 Classe SistemaParticulas (JAVA)	87
E – SUPERFÍCIE DE RENDERIZAÇÃO	90
E.1 Classe ParticlesRenderer (JAVA).....	90
F – INSTAÇÃO ANDROID STUDIO E ANDROID VIRTUAL DEVICE.....	96
ANEXO	100
A – ARQUIVOS DE SHADER	100
A.1 Vertex Shader (GLSL).....	100
A.2 Fragment Shader (GLSL)	100
B – CLASSES UTIL.....	101
B.1 Classe Geometry (JAVA)	101
B.2 Classe LoggerConfig (JAVA).....	102
B.3 Classe MatrixHelper (JAVA)	102
B.4 Classe ShaderHelper (JAVA).....	102
B.5 Classe TextResourceReader (JAVA)	105
B.6 Classe TextureHelper (JAVA).....	105
B.7 Classe Constants (JAVA)	106

1. INTRODUÇÃO

Segundo Hughes (2013) a Computação Gráfica (CG) é a ciência e arte de se comunicar visualmente por meio de um computador e demais componentes que permitem a interação do usuário. Sendo uma área interdisciplinar, a CG está presente em diversas áreas do conhecimento tais como Física para a modelagem da luz e realização de simulações, na Matemática para o desenvolvimento de forma de objetos, entre outros, conforme Figura 1.1:

Figura 1.1 – Segmentos onde a CG está presente

Arte	Efeitos especiais, modelagens criativas, esculturas e pinturas
Medicina	Exames, diagnósticos, estudo, planejamento de procedimentos
Arquitetura	Perspectivas, projetos de interiores e paisagismo
Engenharia	Em todas as suas áreas (mecânica, civil, aeronáutica etc.)
Geografia	Cartografia, GIS, georreferenciamento, previsão de colheitas
Meteorologia	Previsão do tempo, reconhecimento de poluição
Astronomia	Tratamento de imagens, modelagem de superfícies
Marketing	Efeitos especiais, tratamento de imagens, projetos de criação
Segurança Pública	Definição de estratégias, treinamento, reconhecimento
Indústria	Treinamento, controle de qualidade, projetos
Turismo	Visitas virtuais, mapas, divulgação e reservas
Moda	Padronagem, estamparias, criação, modelagens, gradeamentos
Lazer	Jogos, efeitos em filmes, desenhos animados, propaganda
Processamento de Dados	Interface, projeto de sistemas, mineração de dados
Psicologia	Terapias de fobia e dor, reabilitação
Educação	Aprendizado, desenvolvimento motor, reabilitação

Fonte: Azevedo e Conci (2003)

Segundo Hughes (2013), a CG impactou as indústrias cinematográfica, televisiva, publicitária e de entretenimento, influenciando a forma na qual visualizamos as informações em nosso dia a dia, porém consideramos que a mudança mais significativa ocorreu graças às *Graphical User Interfaces* (GUI), ou interfaces gráficas do usuário.

Um dos frutos da CG são os efeitos especiais e, segundo Artero e Santos (2011), tais efeitos têm sido muito utilizados na televisão e no cinema, permitindo a realização de cenas impossíveis, que traziam grandes riscos à integridade física dos atores ou por conta de outras impossibilidades.

Segundo Azevedo e Conci (2003):

“Se você puder imaginar algo, isso pode ser gerado com a computação gráfica. A computação gráfica está a um passo de um mundo novo, repleto de aplicações, ainda desconhecidas, e muitas oportunidades de trabalho para designers, modeladores, animadores, iluminadores e programadores.”

Mesmo que a CG permita a criação de qualquer tipo de objeto ou cena, Azevedo e Conci (2003) afirmam que o desenvolvimento de CG pode parecer difícil, pois requer que os desenvolvedores possuam conhecimento tanto da teoria de CG, quanto de bibliotecas de desenvolvimento utilizadas, tais como *Open Graphics Library* (OpenGL), e dos designers o conhecimento adicional sobre técnicas de modelagem.

Segundo Coelho (2013), em relação às bibliotecas de desenvolvimento, a OpenGL se destaca por se tratar de uma Application Programming Interface (API) que permite a criação de elementos em 2D e 3D através da definição de objetos por meio da utilização de primitivas geométricas e comandos de manipulação.

De acordo com Brothaler (2013), existe a derivação do OpenGL chamada *Open Graphics Library Embedded Systems* (OpenGL ES), que é a API OpenGL voltada a dispositivos móveis. E é importante citar sua existência, visto que segundo a Opus Software (2016), existe uma demanda crescente por desenvolvimento de aplicativos, já que existe também uma crescente adoção do uso de smartphones.

Segundo Hughes (2013):

“O gráfico é um campo amplo, para entendê-lo, você precisa de informações de percepção, física, matemática e engenharia. Construir uma aplicação gráfica implica um trabalho de interface do usuário, uma certa quantidade de modelagem (isto é, fazendo uma representação de uma forma) e renderização (criação de imagens de formas).”

Desde o surgimento da CG, vários estudos foram realizados na área, auxiliando no seu desenvolvimento. Entre eles está o campo de estudo que foi o resultado da CG com a indústria cinematográfica, a modelagem de objetos não determinísticos.

Esses objetos são caracterizados pela impossibilidade de serem representados por meio da utilização de primitivas geométricas, sendo apenas possível com a utilização de formas complexas.

Dentro da relação CG e cinema, Reeves (1983) fez uma publicação, nomeada *Particles Systems - Technique for Modeling a Class of Fuzzy Object* (Sistema de Partículas - Técnica para modelar uma classe de objetos difusos), propondo um sistema de partículas (SP) para a modelagem de objetos não determinísticos, mostrando também sua aplicação prática em um filme.

Segundo Reeves (1983)

“Um SP é uma coleção de muitas minúsculas partículas que, juntas, representam um objeto difuso. Ao longo de um período, as partículas são geradas num sistema, se movimentam e interagem dentro dele, e morrem.”

Segundo Reeves (1983) um SP possui três particularidades em relação a representações comuns de imagens, sendo: (1) é representado por uma nuvem de partículas; (2) cada partícula possui um ciclo de vida e (3) é não-determinístico, ou seja, um SP pode assumir diversas formas. Usando SP, foi possível criar a cena *Genesis Effect* do filme *Star Trek II – The Wrath of Khan* (Star Trek II - A Ira de Khan).

1.1. Justificativa

Segundo a Opus Software (2016), com o crescente uso de Smartphones, faz-se necessário o desenvolvimento de aplicativos que atendam esta plataforma e auxiliem usuários nas atividades do dia a dia.

O desenvolvimento de um SP para smartphones possibilita uma atualização do estudo sobre partículas no contexto da programação *mobile* ao mesmo tempo em que estimula a criação de aplicativos voltados aos usuários da plataforma.

A escolha pelo ambiente *mobile* também tem como objetivo, demonstrar a aplicação da tecnologia OpenGL ES e sua capacidade de processamento, tal como os computadores são capazes de realizar.

1.2. Objetivos

1.2.1. Objetivo Geral

O objetivo deste trabalho é construir uma aplicação mobile para demonstrar e explicar o uso de sistemas de partículas, bem como as possibilidades que tal técnica permite simular.

1.2.2. Objetivos Específicos

- Analisar SP implementados feitos para computadores de mesa, seus pontos fortes e suas limitações.
- Analisar a plataforma mobile, bem como seus avanços na parte gráfica e seus limites.
- Compreender tanto a plataforma mobile quanto os sistemas existentes para desktop e os cálculos necessários para o funcionamento desse sistema.
- Implementar interface gráfica que permita interação do usuário para com o SP.
- Implementar exemplos de objetos não determinísticos para uso e exemplificação no SP.
- Demonstrar os resultados obtidos por meio de aplicativo.
- Detalhar dificuldades e limitações.

1.3. Metodologia

Por meio de pesquisa ação foram feitas análises de sistemas de partículas e tecnologia gráfica disponível para dispositivos *mobile*, com base nos resultados dessas pesquisas determinamos se é possível a implementação de um SP para o ambiente *mobile*, por fim caso possível, realizando a implementação e uso de SP para esse ambiente. Como instrumento de testes e coleta de dados serão utilizados dispositivos *mobile* e/ou maquinas virtuais com sistema operacional Android.

2. COMPUTAÇÃO GRÁFICA

2.1. Histórico da computação gráfica

De acordo com Azevedo e Conci (2003), na década de 50 surgiu o primeiro computador com recursos gráficos e o termo “computação gráfica”. Este computador era o Whirlwind I, utilizado para executar o sistema SAGE (Sistema de monitoramento e controle de voos) para fins acadêmico e militar. Enquanto que o termo “computação gráfica” surgiu no final da década de 50 por Verne Hudson enquanto criava um simulador de fatores humanos em aviões.

Ainda de acordo com Azevedo e Conci (2003), a década de 60 foi marcada por dois acontecimentos importantes para o desenvolvimento da computação gráfica. A primeira é a publicação da tese de Ivan Shuterland (*Sketchpad – A Man -Machine Graphical Communication System*), pois tal publicação continha uma série estruturas de dados e técnicas de interação. O segundo acontecimento veio em decorrência do primeiro, a tese publicada chamou a atenção da indústria automobilística e aeroespacial americana. Essa atenção levou a General Motors a desenvolver um programa de CAD (Computer Aided Design) que foi precursor para o desenvolvimento desse tipo de software por toda essa indústria automobilística.

Já a década de 70 temos a introdução da computação gráfica no cinema. De acordo com Rayne (2018), naquela década os filmes *Westworld* e *Futureworld*, fizeram uso de CGI 2D e 3D para simular a visão, face e mãos do robô. Também houve o surgimento da SIGGRAPH (Special Interest Group on GRAPHics and Interactive Technique), que segundo a ACM SIGGRAPH (2013) surgiu em 1974, com o objetivo de promover a inovação em computação gráfica e técnicas interativas.

Na década de 80, de acordo com Azevedo e Conci (2003) houve o surgimento de novas técnicas de iluminação, como o ray-tracing em 1980 e radiossidade em 1984. Não só técnicas de iluminação, mas objeto de estudo deste trabalho, a publicação do SP em 1983 e sua utilização em um longa-metragem no ano anterior.

Adentrando a década de 90, temos o lançamento da API OpenGL (vide histórico de versões OpenGL, Khronos Group (2017)), o lançamento de um longa-metragem totalmente 3D *Toy Story* (de acordo com Azevedo e Conci (2003)), e

segundo a Nvidia (2003), em 1999 introduziram o conceito de GPU (Graphics Processing Unit) no mercado com o lançamento da placa de vídeo GeForce 256.

2.2. Uso da computação gráfica na medicina

Segundo Hughes (2013) a CG alterou a forma como enxergamos as informações do nosso dia a dia, tais como na Medicina.

Azevedo e Conci (2003) afirmam que a “área média encontra na CG uma poderosa aliada. É possível simular o corpo humano e obter conclusões a partir disso”

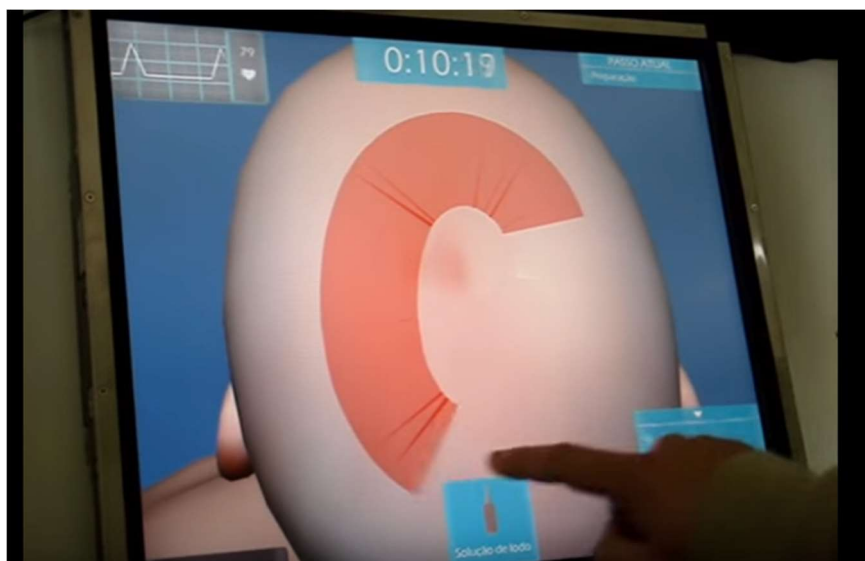
Figura 2.1 – Menu SimCMed



Fonte: Keepplay Game Studios (2010)

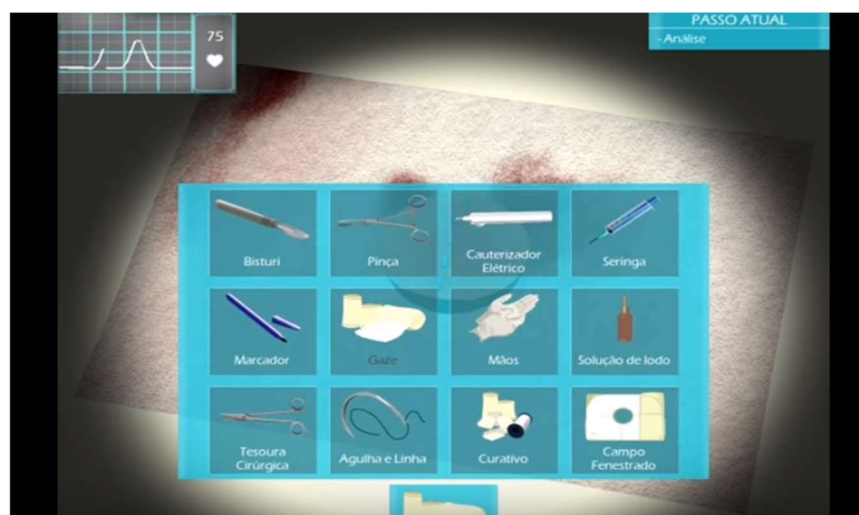
O Simulador 3D de Cirurgias Médicas (SimCMed), desenvolvido pela Keepplay Game Studios é um exemplo de como a CG pode ser utilizada para disponibilizar meios para que os alunos e profissionais da área médica possam aprender sobre conceitos de sua área de atuação e como lidar em possíveis situações durante procedimentos cirúrgicos, conforme Figura 2.1.

Figura 2.2 – Demonstrando interação com paciente virtual



Fonte: Keeplay Game Studios (2010)

Figura 2.3 – Demonstrando instrumentação cirúrgica.



Fonte: Keeplay Game Studios (2010)

Segundo a Keeplay Game Studio (2010), o simulador permite a aplicação dos conhecimentos em pacientes virtuais, por meio da interatividade proporcionada por uma tela *touchscreen* e elementos 3D, conforme Figura 2.2 e 2.3:

Outro exemplo de simulador é o projeto Health Simulator (MELLO, STAHNKE e BEZ, 2015), assim como o SimCMed, propõe a simulação de pacientes virtuais.

Segundo Mello, Stahnke e Bez (2015):

“O projeto Health Simulator objetiva oferecer ao professor uma ferramenta de auxílio que visa desenvolver, através da prática e interatividade com o ambiente e situações clínicas comuns, o raciocínio clínico do aluno. É um ambiente que proporciona

experiências reais e cotidianas da rotina na área da saúde, sem a presença dos riscos usuais inerentes.”

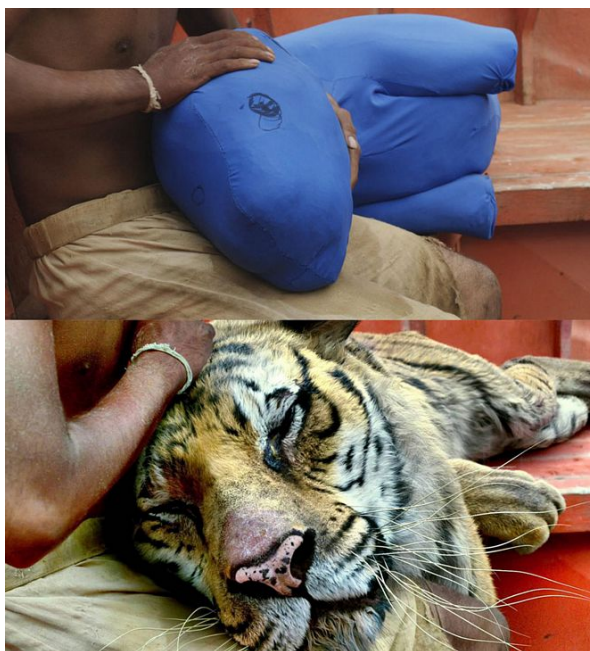
2.3. Uso da computação gráfica no cinema

De acordo com Linares (2012), a CG é tão disseminada que é usada em praticamente qualquer produto de entretenimento que assistimos. Além de abertura de novelas ou programas de televisão, podemos ver artes digitais em várias propagandas de televisão.

Segundo Artero e Santos (2011):

“Efeitos Especiais têm sido muito utilizados na televisão e no cinema, permitindo a realização de cenas impossíveis de serem feitas usando atores reais, devido à causa de grandes riscos à integridade física ou por causa de outras impossibilidades.”

Figura 2.4 – Filme: As Aventuras Pi (2012)



Fonte: Dias (2014)

O filme As aventuras de Pi (2012), fazia uso CG na interação entre o personagem principal e o tigre da história, conforme mostra a Figura 2.4. Já o filme Planeta dos Macacos (2011), o personagem Cesar, teve seu corpo construído usando CG, um ator fazia os movimentos característicos de um macaco e o computador aplicava os movimentos a animação (Figura 2.5).

Figura 2.5 – Filme: Planeta dos Macacos: A Origem (2011)



Fonte: Dias (2014)

Figura 2.6 – Filme: Wolverine: Imortal



Fonte: Dias (2014)

No filme *Wolverine Imortal* (2013), houve uso de CG na animação de regeneração de um corte profundo no rosto do personagem principal (Figura 2.6).

Vale mencionar também, uma curiosidade sobre a Figura 2.7 e o filme *Liga da Justiça* (2017), que segundo a revista eletrônica *Monet* (2017), o ator Henry Cavill estava preso a um contrato com outro filme que impedia ele de remover o bigode. Por conta disso foi feito uso de CG para remover digitalmente o bigode do ator.

Figura 2.7 – Filme: Liga da Justiça (2017)



Fonte: Monet (2017)

2.4. Uso da computação gráfica na engenharia

Conforme escrito na seção 2.1, a tese do Dr. Ivan Sutherland sobre CG chamou a atenção das indústrias automobilísticas e aeroespaciais americanas, sendo um dos motivos que levaram a General Motors a desenvolver o software precursor dos programas de CAD.

Segundo Linares (2012):

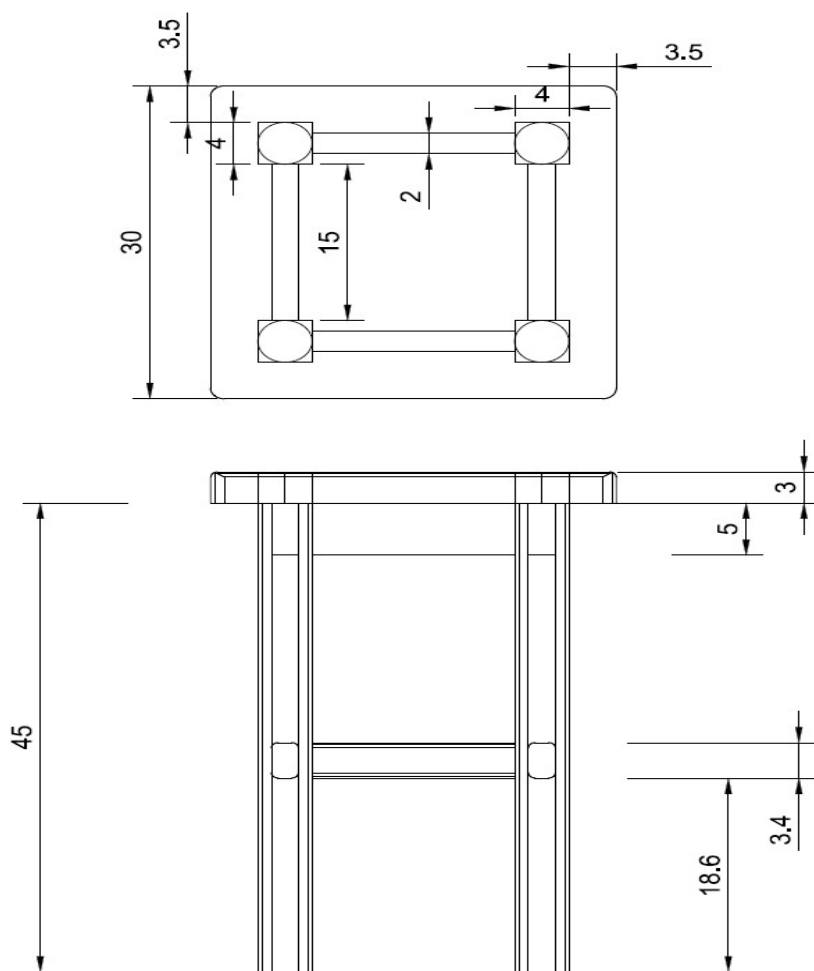
“Ferramentas mais modernas como o AutoCAD são capazes de gerar modelos tridimensionais complexos, realizar simulações de design sem uso de um modelo físico, criar desenhos 2D a partir de objetos 3D, além da possibilidade de conferir se desenhos já feitos obedecem a padrões pré-estabelecidos. Assim, o programa é usado na criação de qualquer elemento de engenharia técnica, por exemplo na criação de carros, pontes ou casas.”

Figura 2.8 – Banqueta 3D modelada a partir de um modelo 2D



Fonte: Tutorial 45 (2017)

Figura 2.9 – Modelo de banqueta 2D usando primitivas geométricas



Fonte: Tutorial45 (2017)

As Figuras 2.8 e 2.9 são exemplos de modelos de banquetas geradas no software AutoCAD. De acordo com o tutorial disponibilizado pelo Tutorial45 (2017), é feita a construção de um modelo 3D a partir da construção de um modelo 2D usando primitivas geométricas.

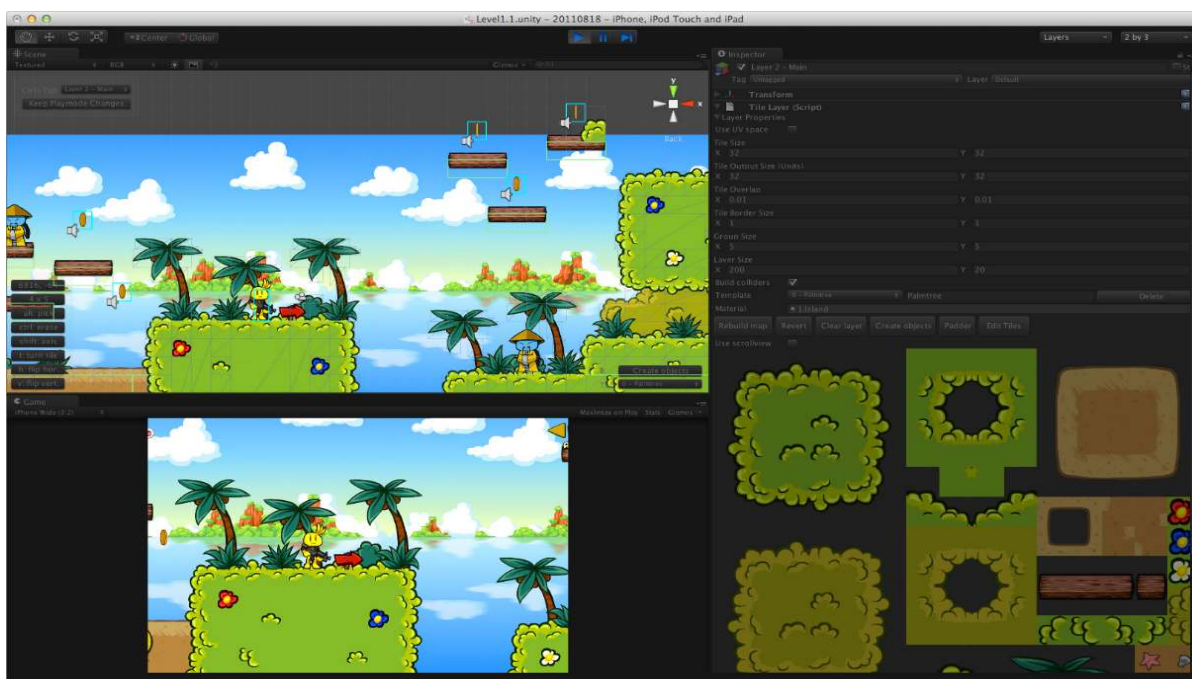
2.5. Uso da computação gráfica em jogos

Segundo Silva (2017):

“Uma *game engine* (motor de jogo), consiste em um conjunto de ferramentas capazes de facilitar o desenvolvimento de um jogo. Geralmente, esses softwares possuem desde recursos para criação de funções gráficas até opções para acrescentar física aos objetos, trilhas sonoras, entre outras ações.”

O site Portal Indie Game (2015) apresenta uma ferramenta popular de desenvolvimento de jogos. Trata-se do Unity 3D, que segundo o site é uma *game engine* muito popular e conta com uma vasta comunidade de usuários. Inclusive contando com ferramentas específicas para criação de jogos. A Figura 2.10 demonstra o uso do software no desenvolvimento de jogos.

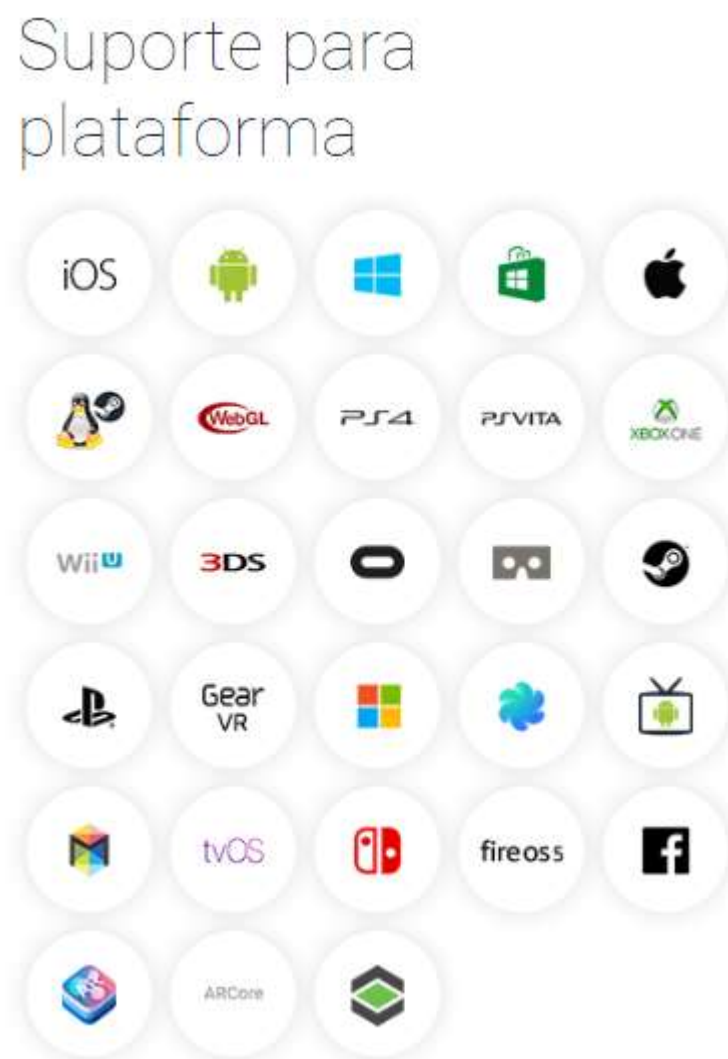
Figura 2.10 – Tela do Unity 3D com um projeto de jogo 2D.



Fonte: Indie Game (2015)

2.6. Uso da computação gráfica em dispositivos móveis

Figura 2.11 – Plataformas suportadas pelo Unity



Fonte: Unity Technologies (2018)

A empresa responsável pelo desenvolvimento do Unity 3D, Unity Technologies (2018), afirma que a ferramenta é multiplataforma, oferecendo suporte diversos sistemas móveis, tais como iOS e Android.

A empresa afirma que a ferramenta é um editor que permite organizar a narração de histórias de modo temporal, permitindo a criação a de conteúdo cinematográfico.

3. OPENGL ES

3.1. API OpenGL

Segundo Coelho (2013) a OpenGL é uma API gráfica que permite a criação de imagens gráficas 2D e 3D, através da definição de objetos por um conjunto de formas primitivas geométricas e rotinas para manipulação.

Segundo Manssour e Cohen (2006):

“As aplicações OpenGL variam de ferramentas CAD a programas de modelagem usados para criar personagens para o cinema, tal como um dinossauro. Além do desenho de primitivas gráficas, tais como linhas e polígonos, OpenGL dá suporte a iluminação, colorização, mapeamento de textura, transparência, animação, entre muitos outros efeitos especiais. Atualmente, OpenGL é reconhecida e aceita como um padrão API para desenvolvimento de aplicações gráficas 3D em tempo real.”

De acordo com Manssour e Cohen (2006) essa ferramenta possui especificação aberta e alta portabilidade, podendo funcionar desde potentes estações de trabalho a simples computadores pessoais. Tem funcionamento é semelhante ao de uma biblioteca C, fornecendo uma série de funcionalidades.

Normalmente se diz que um programa é baseado em OpenGL ou é uma aplicação OpenGL, o que significa que ele é escrito em alguma linguagem de programação que faz chamadas a uma ou mais bibliotecas OpenGL. (AZEVEDO e CONCI, 2003)

3.2. Funcionamento API OpenGL

De acordo com Manssour e Cohen (2006), quando a OpenGL é utilizada, não é necessário descrever com detalhes uma cena 2D ou 3D, basta especificar o conjunto de passos que devem ser seguidos para se obter a cena desejada. Esses passos

envolvem chamadas de rotinas na biblioteca da API OpenGL ou GLU (*OpenGL Utility Library*), que também faz parte da implementação OpenGL.

Manssour e Cohen (2006) também dizem que palavra *pipeline* é utilizada para descrever um processo que pode ter dois ou mais passos. Nesse caso a Figura 3.1 apresenta um pipeline simplificado do OpenGL. Detalhando, quando uma aplicação faz chamadas à API OpenGL, vários comandos, vértices, dados de textura etc. são colocados em um buffer. Este buffer quando preenchido, passa os dados para a próxima etapa, esvaziando-o.

Figura 3.1 – Versão simplificada do pipeline do OpenGL



Fonte: Manssour & Cohen (2006)

De acordo com Manssour e Cohen (2006) existem diferenças no processamento de dados geométricos e dados de imagem, porém mesmo com diferenças, todo dado processado passa pela etapa de *Rasterization*, que faz a conversão dos dados em operações de fragmentos.

As operações de fragmento tratam do processamento de cada posição da tela na memória da GPU, essas posições podem conter informações como cor, profundidade e textura. Cada fragmento tem sua parcela de contribuição para atualização dos pixels mostrados na tela.

3.3. API OpenGL ES

“O OpenGL ES, versão derivada da OpenGL para desktops, é o padrão para o desenvolvimento gráficos em dispositivos móveis, porém não mantém códigos para compatibilidade com versões anteriores”. (Brothaler, 2013)

Segundo a Khronos Group (2018):

O OpenGL ES é uma API multiplataforma, livre de royalties, para renderizar gráficos 2D e 3D avançados em sistemas

embarcados e móveis - incluindo consoles, telefones, eletrodomésticos e veículos. Ele consiste em um subconjunto bem definido do OpenGL de desktop adequado para dispositivos de baixa potência e fornece uma interface flexível e poderosa entre hardware de aceleração de software e gráficos.

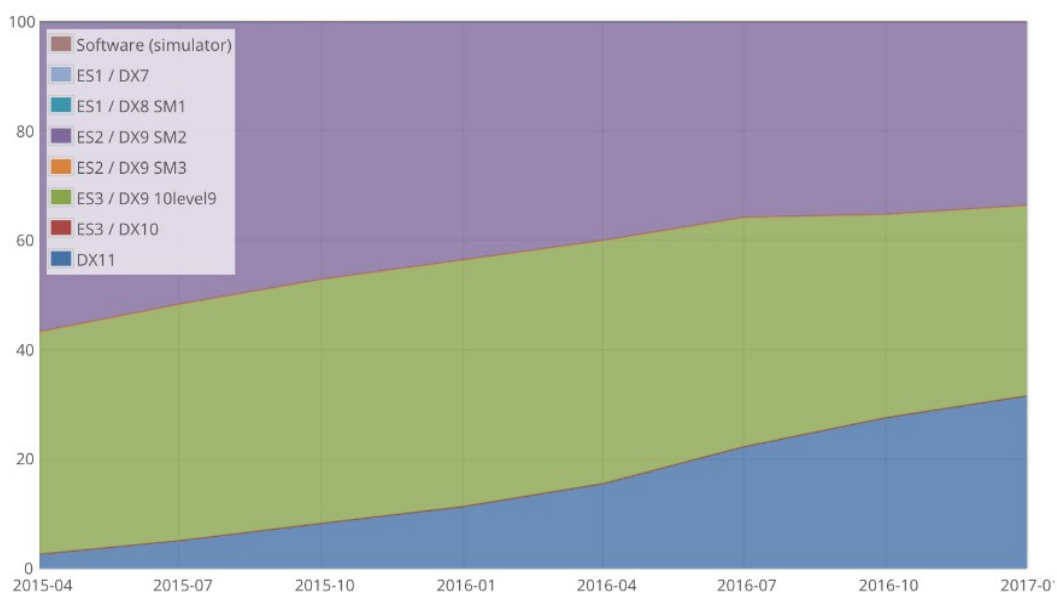
3.4. API OpenGL ES 1.X

De acordo com a Khronos Group (2018) as versões 1.0 e 1.1 foram as primeiras APIs gráficas desenvolvidas com o intuito de serem utilizadas em dispositivos móveis, sendo uma derivação direta da versão 1.5 do OpenGL para desktops. Nestas versões o pipeline fornecido é classificado de função fixa, ou não programável.

3.5. API OpenGL ES 2.X

“O OpenGL ES 2.0 foi a primeira API *mobile* a permitir *shaders* programáveis para o hardware gráfico. Ele continua sendo uma API predominante e mais amplamente disponível, sendo uma opção sólida para alcançar a maior variedade de dispositivos.” (Khronos Group, 2018)

Figura 3.2 – Gráfico de Implementação de API Gráfico 3D



Fonte: Khronos Group (2018)

De acordo com a Figura 3.2 divulgada pelo Khronos Group (2018), seu produto OpenGL ES é a API 3D mais amplamente implementada da história, existindo sempre um paralelo entre seu produto e o DirectX, além de uma longa lista de empresas que implementam a API.

3.6. API OpenGL ES 3.X

“O OpenGL ES 3.0 foi outro passo evolutivo do OpenGL ES, que adiciona múltiplos destinos de renderização, maior capacidade de texturização, buffers uniformes, instanciação e transformação afim.” (Khronos Group, 2018)

Entretanto na versão 3.1, apesar de ser apenas uma revisão, foi considerado como um marco para a API, visto que adicionou a capacidade de realizar computação de propósito geral, permitindo que a GPU dos dispositivos fosse utilizada para o processamento além de CG.

“A versão mais recente da série (OpenGL ES 3.2), adicionou funcionalidade baseada no *Android Extension Pack* (AEP) para OpenGL ES 3.1, que aproximou significativamente as funcionalidades da API com a versão desktop.” (Khronos Group, 2018)

3.7. Shaders OpenGL ES

. O funcionamento da API OpenGL ES é similar ao apresentado na seção 3.2, basicamente após a escrita de um programa é necessário fazer as chamadas à API e repassar os dados para o Buffer. Com suporte a shaders programáveis, é possível definir como esses dados serão armazenados e executados na GPU, porém para definir essas informações é necessário um link entre a GPU e os programas que serão executados na GPU (*vertex shader* e *fragments shader*). (Brothaler, 2013)

Segundo Brothaler (2013), cada programa possui uma utilidade, sendo o programa *vertex* responsável por organizar e processar as informações individuais de cada elemento enviado a GPU e o programa *fragments* organiza estes dados e instrui o OpenGL ES em como desenhar os fragmentos que compõe cada um dos elementos da tela (pontos, linhas e triângulos).

3.8. Vertex Shader

Segundo a Khronos Group (2017), o *vertex shader* trata-se de um estágio do *shader* onde o *pipeline* de renderização lida com o processamento de cada vértice individualmente. Os dados gerenciados são as informações especificadas em um *vertexarray*, neste trabalho é um vetor alimentado pelo SP, que após ser processado, ocasiona na geração de um vértice de saída para cada vértice de entrada.

Como explicado na seção 3.2, podem existir estágios adicionais, como por exemplo *Tessellation* e *Geometry Shader*, onde realizam algum processamento antes da *Rasterization*, que geram os fragmentos.

3.9. Fragment Shader

De acordo com a Khronos Group (2018), *fragment shader* trata-se de um estágio do *shader*, posterior a *rasterization* (Figura 3.1). Após o processamento dos estágios anteriores, a *rasterization* gera fragmentos, que são processados pelo *fragment shader* e traduzido em cores, posição na superfície de renderização e profundidade.

3.10. OpenGL Shading Language (GLSL)

O modelo de SP desenvolvido nesse trabalho tem como base o SP proposto por Brothaler (2013), porém com grandes modificações e um número expressivamente maior de programas de *shader*. O motivo é que, como os programas de *shader* tem como propriedade o processamento individual de cada entrada de dados, podemos aproveitar disso, e programar por meio dos *shaders*, um possível comportamento individual para cada partícula projetada na tela.

De acordo com Ginsburg e Purnomo (2014), não é tão simples construir os arquivos de *shaders*, visto que os mesmos possuem uma linguagem própria, a *OpenGL Shading Language* (GLSL), a qual compartilha determinadas semelhanças com a linguagem de programação C. Em seu livro aprendemos instruções como tipos de variáveis, variáveis construtoras, vetor, matriz, constantes, estruturas, etc.

Figura 3.3 - Tipos de dados GLSL

Variable Class	Types	Description
Scalars	<code>float</code> , <code>int</code> , <code>uint</code> , <code>bool</code>	Scalar-based data types for floating-point, integer, unsigned integer, and boolean values
Floating-point vectors	<code>float</code> , <code>vec2</code> , <code>vec3</code> , <code>vec4</code>	Floating-point-based vector types of one, two, three, or four components
Integer vector	<code>int</code> , <code>ivec2</code> , <code>ivec3</code> , <code>ivec4</code>	Integer-based vector types of one, two, three, or four components
Unsigned integer vector	<code>uint</code> , <code>uvec2</code> , <code>uvec3</code> , <code>uvec4</code>	Unsigned integer-based vector types of one, two, three, or four components
Boolean vector	<code>bool</code> , <code>bvec2</code> , <code>bvec3</code> , <code>bvec4</code>	Boolean-based vector types of one, two, three, or four components
Matrices	<code>mat2</code> (or <code>mat2x2</code>), <code>mat2x3</code> , <code>mat2x4</code> , <code>mat3x2</code> , <code>mat3</code> (or <code>mat3x3</code>), <code>mat3x4</code> , <code>mat4x2</code> , <code>mat4x3</code> , <code>mat4</code> (or <code>mat4x4</code>)	Floating-point based matrices of size 2×2 , 2×3 , 2×4 , 3×2 , 3×3 , 3×4 , 4×2 , 4×3 , or 4×4

Fonte: Ginsburg e Purnomo (2014)

A Figura 3.3 de Ginsburg e Purnomo (2014) demonstra os tipos de dados suportados pelo GLSL, como vetores e matrizes já forma de escrita é um pouco diferente da utilizada em C. Os tipos de dados como `int`, `bool` e `float` são correspondentes ao C.

Ginsburg e Purnomo (2014) mostram através das Figura 3.4 e 3.5 os operadores suportados pelo GLSL, como adição, subtração, divisão e multiplicação. Existem mais operadores como incremento, decremento, comparador de igualdade, diferença, maior igual, menor igual, todos com sintaxe similar a utilizada pela linguagem C.

Figura 3.4 - Operadores GLSL

Table 5-2 OpenGL ES Shading Language Operators

Operator Type	Description
*	Multiply
/	Divide
%	Modulus
+	Add
-	Subtract
++	Increment (prefix and postfix)
--	Decrement (prefix and postfix)
=	Assignment
+=, -=, *=, /=	Arithmetic assignment

Fonte: Ginsburg e Purnomo (2014)

Figura 3.5 Continuação operadores GLSL

Operator Type	Description
==, !=, <, >, <=, >=	Comparison operators
&&	Logical and
^^	Logical exclusive or
	Logical inclusive or
<<, >>	Bit-wise shift
&, ^,	Bit-wise and, xor, or
?:	Selection
,	Sequence

Fonte: Ginsburg e Purnomo (2014)

4. SISTEMAS DE PARTÍCULAS

4.1. O que são sistemas de partículas?

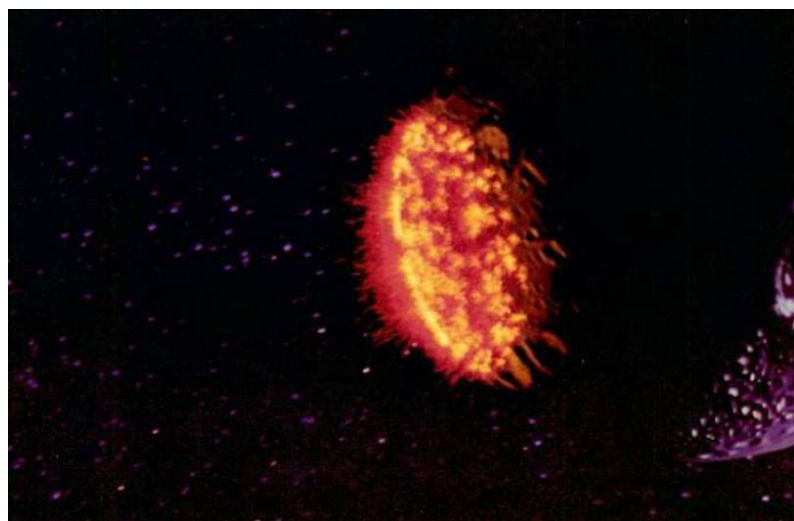
“Um SP é uma coleção de diversas partículas que, juntas, representam um objeto difuso. Ao longo de um período, as partículas são geradas em um sistema, se movimentam e interagem dentro dele, e morrem.” (Reeves, 1983)

4.2. História dos sistemas de partículas

Segundo Hughes (2013), a CG impactou as indústrias cinematográfica, televisiva, publicitária e de entretenimento. Dentro da relação CG com o cinema, partindo da necessidade de uma tecnologia que facilitasse a representação de objetos não determinísticos, Reeves (1983), fez uma publicação chamada *Particles Systems - Technique for Modeling a Class of Fuzzy Object* em que descrevia um SP e como sua implementação tornou possível a simulação de um efeito especial em um filme de 1982.

A cena que ficou conhecida como *Genesis Effect*, a qual utilizava das partículas para simular efeito de fogo, conforme mostrado na Figura 4.1

Figura 4.1 – Genesis Effect

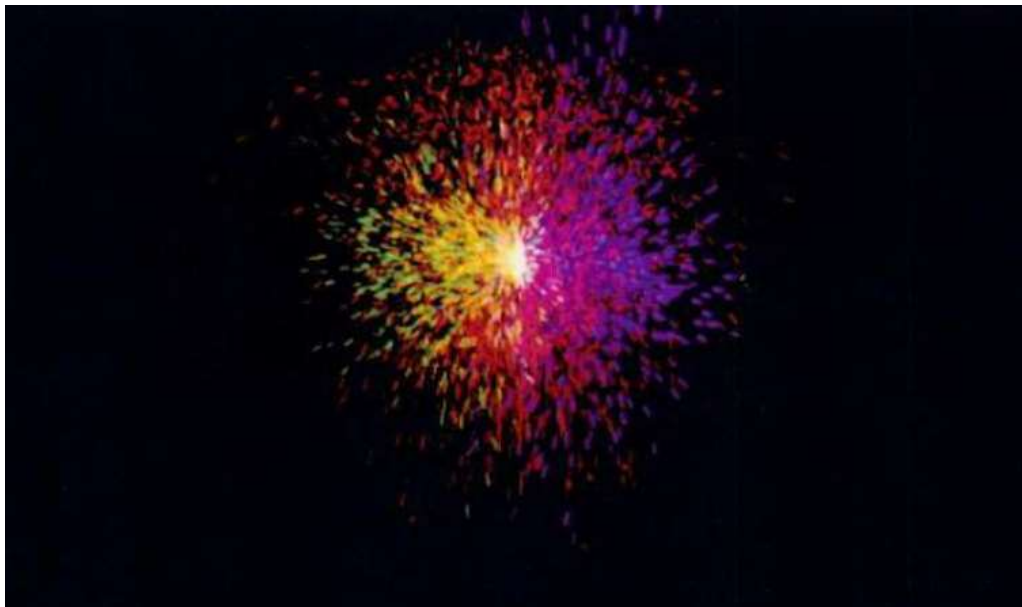


Fonte: REEVES (1983).

Reeves (1983) também demonstrou outras aplicações com SP, mostrando que era possível realizar simulações diversificadas, não se limitando somente ao efeito de

fogo. Conforme pode ser visto na Figura 4.2, o SP foi capaz de simular fogos de artifício coloridos.

Figura 4.2 – Fogos de artifício coloridos



Fonte: REEVES (1983).

4.3. Composição

De acordo com Reeves (1983), a representação de SP se difere de três formas em relação as representações que normalmente eram utilizados naquela época.

- Os objetos formados nesse sistema não são representados por primitivas geométricas, mas sim por nuvens de partículas.
- Os objetos desse sistema não são formas estáticas, logo suas partículas possuem um tempo de vida.
- Um objeto desse sistema não é determinístico, já que sua forma não é completamente definida.

Essas definições permitem que o SP possa ser usado para representar diferentes formas, movimentar-se na tela e sumir e aparecer dependendo do seu tempo de vida, como as Figuras 4.1 e 4.2 demonstraram, o mesmo sistema com duas representações distintas.

Reeves (1983) também detalha as vantagens da composição de uma partícula, segundo ele: (1) uma partícula é mais simples que um polígono simples, sendo formado pela primitiva mais básica (em OpenGL ES é *gl_point*), por consequência o tempo de computação é curto, permitindo que várias primitivas básicas sejam usadas

para produzir uma representação complexa. (2) o número de partículas em um SP não é fixo, assim, o nível de detalhamento pode ser ajustado de acordo com os parâmetros desejados. (3) as partículas que estão “vivas” dentro do SP podem mudar de forma ao longo de seu período.

Um SP também é composto por um processo de geração de partículas, onde deve ser especificado quantas partículas devem ser introduzidas por quadro de vídeo. Um detalhe importante é que o número de partículas geradas nesse processo é o que define a densidade da nuvem de partículas.

Outras informações que Reeves (1983) disponibiliza refere-se aos atributos de uma partícula, sendo elas: (1) posição inicial, (2) velocidade e direção inicial, (3) tamanho inicial, (5) transparência inicial, (6) forma e (7) tempo de vida.

O aplicativo produzido neste trabalho é baseado no SP de Brothaler, dessa forma, algumas propriedades não ficam diretamente associados à partícula, mas aos arquivos de *shader* que serão discutidos nos capítulos 6 e 7.

5. PESQUISAS QUE ENVOLVEM SISTEMAS DE PARTÍCULAS

5.1. Aplicação em Realidade Aumentada

Proposto por Assunção (2008), o trabalho 'Construindo Ambientes com Realidade Aumentada Utilizando Sistemas de Partículas', demonstra a combinação de diversas tecnologias para produzir um resultado de acordo com o que o título sugere.

Nesse trabalho foi feito o uso de tecnologia de Realidade Virtual (RV), Realidade Aumentada (RA) e um SP para promover a interação com as tecnologias. Os sistemas de RV e RA, foram implementados usando biblioteca ARToolKit combinado com OpenGL (usado na implementação do SP).

Aplicação de RA consiste na inserção de objetos virtuais estáticos no ambiente real como um complemento para a realidade. Uma característica de SP é que seus objetos não são estáticos (capítulo 4, seção 4.3), dessa forma a combinação entre SP e uma aplicação de RA tornaria possível a inserir objetos virtuais não estáticos no ambiente real.

Como exemplo, Assunção (2008) descreve uma possível aplicação de RA combinado SP, citando que com RA seria possível colocar uma vela sob uma mesa usando objetos cilíndricos, e um SP poderia tornar essa vela realista adicionando fogo e derretimento.

Figura 5.1 – Uso de sistema de partículas em realidade aumentada



Fonte: de Assunção (2008).

Como resultado de seu trabalho, Assunção (2008) construiu uma nave espacial usando primitivas geométricas e para simular os propulsores dessa nave, foram

utilizadas duas fontes de partículas, com efeito de Fogo. A Figura 5.1, demonstra o uso combinado de SP e RA.

5.2. Aplicação em simulação de fluídos

A dissertação ‘Animação computacional de escoamento de fluídos utilizando o método SPH’, escrito por Queiroz (2008), faz a descrição bem sobre o método *Smoothed Particle Hydrodynamics (SPH)*, que faz uso de partículas para realizar a simulação de fluídos.

Em seu trabalho Queiroz (2008) dedica as sessões 2.3.2 e 2.6 do capítulo 2 para abordar sobre os métodos baseados em partículas e SPH, justificando uso de SP no método SPH, argumentando que as propriedades de SPs permitem a aplicação de funções de aproximação por meio de integrais.

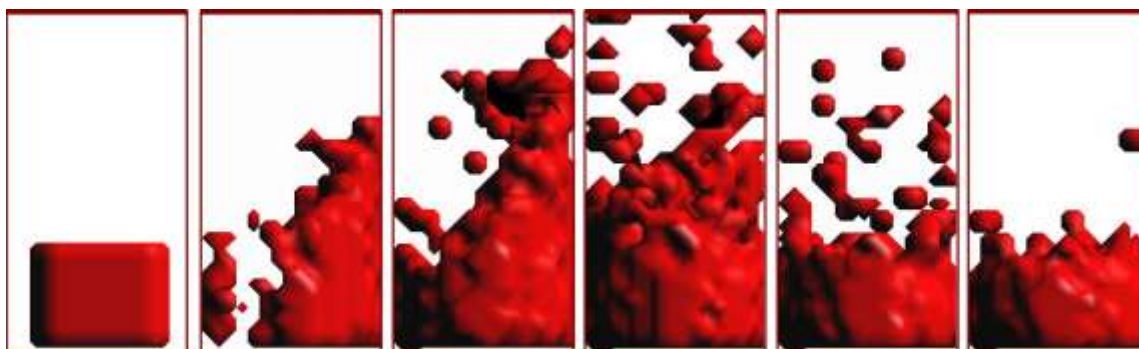
Para a aplicação prática de sua dissertação, Queiroz (2008) faz uso de softwares como Blender 3D e API OpenGL. As Figuras 5.2 e 5.3 demonstram alguns dos resultados de suas simulações utilizando diferentes representações de superfície.

Figura 5.2 - Simulação no Blender 3D com representação em *Multi-level partition of unity implicit*



Fonte: Queiroz (2008).

Figura 5.3 - Simulação em OpenGL com representação em *Marching Cubes*



Fonte: Queiroz (2008).

6. SIMULAÇÃO DE PARTÍCULAS

6.1. Ambiente de simulação

O ambiente de simulação, é um dispositivo *mobile* onde o aplicativo será instalado para efetuar a simulação dos efeitos especiais, neste projeto foi usado o dispositivo Lenovo K6 (Figura 6.1).

6.1.1. Lenovo Vibe k6.

Figura 6.1 – Informações sobre o aparelho

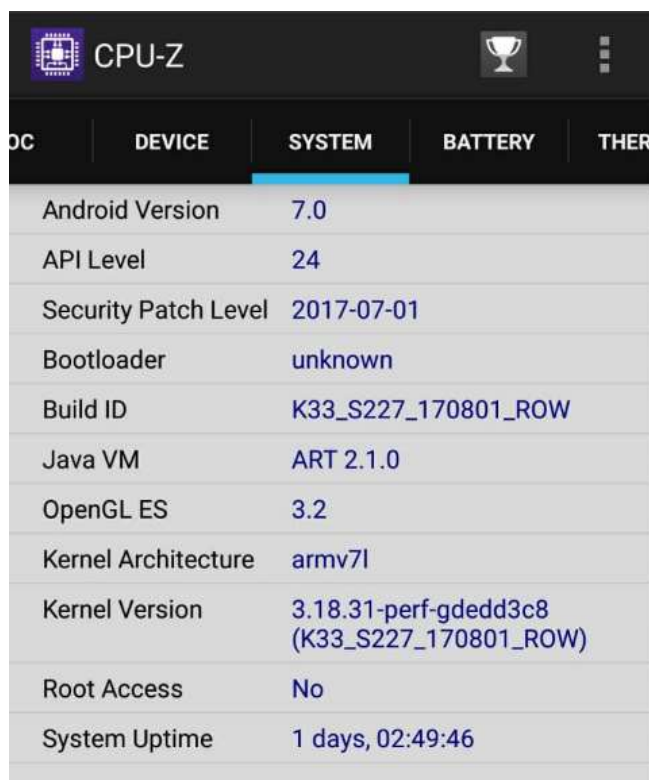


SOC	DEVICE	SYSTEM	BATTERY
Model	Lenovo K6 (Lenovo K33b36)		
Board	S82937DA1		
Hardware	qcom		
Screen Size	5,00 inches		
Screen Resolution	1080 x 1920 pixels		
Screen Density	440 dpi		
Dimensions	141,9 x 70,3 x 8,2 mm		
Weight	140 g		
Total RAM	1882 MB		
Available RAM	590 MB (31%)		
Internal Storage	23,99 GB		
Available Storage	4,49 GB (18%)		
Release Date	2016-11-01		

Fonte: O autor (2018).

O modelo de SP de Brothaler (2013), necessita de um dispositivo com suporte ao OpenGL ES 2.0 por fazer uso de shaders programáveis. O dispositivo Lenovo k6 conta com suporte ao OpenGL ES 3.2 e Android 7.0 (conforme pode ser visto na Figura 6.2) além de 2GB de RAM e processador Octacore de 1,4GHZ. Na seção seguinte é apresentado a preparação do ambiente de desenvolvimento com uma configuração que permite também a simulação através de um dispositivo virtual.

Figura 6.2 – Informações sobre o sistema do aparelho



The image shows the CPU-Z application interface. At the top, there's a header with the CPU-Z logo, a trophy icon, and a menu icon. Below the header is a navigation bar with tabs: CPU, DEVICE, SYSTEM (highlighted), BATTERY, and THERMAL. The main content area displays system information in a table-like format.

Android Version	7.0
API Level	24
Security Patch Level	2017-07-01
Bootloader	unknown
Build ID	K33_S227_170801_ROW
Java VM	ART 2.1.0
OpenGL ES	3.2
Kernel Architecture	armv7l
Kernel Version	3.18.31-perf-gdedd3c8 (K33_S227_170801_ROW)
Root Access	No
System Uptime	1 days, 02:49:46

Fonte: O autor (2018).

6.2. Preparação do ambiente de desenvolvimento e simulação

O conteúdo acerca da preparação do ambiente de desenvolvimento e dos dispositivos virtuais para simulação se encontram no Anexo F deste documento.

6.3. Estrutura de uso do OpenGL ES

De acordo com Ginsburg e Purnomo (2014), para construir um código OpenGL ES funcional é necessário:

- Criar uma superfície de renderização na tela usando EGL (Interface entre OpenGL ES e o sistema de telas do Android).
- Carregar os shaders (neste projeto são os programas *vertex* e *fragment shader*).
- Criar um objeto e o anexar aos programas de shader (*vertex* e *fragment shaders*).
- Definir a janela.

- Apagar a memória de cores.
- Processar uma primitiva geométrica.
- Tornar o conteúdo na superfície de tela EGL.

Segundo Ginsburg e Purnomo (2014), o OpenGL ES é baseado em *shaders*, fazendo com que haja mais códigos de configuração do que a versão para computadores, visto que sem que todos os *shaders* apropriados estejam carregados, nenhuma primitiva pode ser processada.

Ginsburg e Purnomo (2014) tentam simplificar a configuração apresentando um código exemplo. A estrutura exemplo é composta das classes *MainActivity*, *MyGLSurfaceView*, *MyGLRenderer*, *Square* e *Triangle*. *MainActivity* contém a instanciação de um objeto *GLSurfaceView*, bem como o ajuste de suas dimensões. Já as classes *MyGLSurfaceView* e *MyGLRenderer* fazendo uso do EGL, definem a janela e criam a superfície de renderização. Por fim tanto *Triangle* quanto *Square* são classes de primitivas geométricas. A estrutura pode ser observada na Figura 6.10.

Figura 6.10 – Classes exemplo



Fonte: O autor (2018).

6.4. Partícula

De acordo com Reeves (1983), uma partícula surge num ponto, se movimenta e “morre”. Na seção 4.3 Reeves também descreve as informações básicas de uma partícula, porém no ambiente *mobile* a forma de controlar uma partícula é diferente, então para construir uma partícula, as informações básicas que podemos definir, são as coordenadas de ponto de partida, direção de movimento, tempo de vida e velocidade de movimento. Desta forma, uma representação básica seria similar a apresentada na Figura 6.11.

Figura 6.11 – Classe Partícula básica

```

1  public class Particula {
2      private float [] ponto_inicial;
3      private float [] direcao_movimento;
4      private float [] velocidade_movimento;
5      private int tempo_vida;
6
7      public Particula(float[] ponto_inicial, float[] direcao_movimento,
8          float[] velocidade_movimento, int tempo_vida) {
9          this.ponto_inicial = ponto_inicial;
10         this.direcao_movimento = direcao_movimento;
11         this.velocidade_movimento = velocidade_movimento;
12         this.tempo_vida = tempo_vida;
13     }
14 }
15

```

Fonte: O autor (2018).

A pretensão é que no momento da simulação, as nuvens de partículas sejam capazes de representar pelo menos fogo, chuva e neve. Para que essas simulações sejam possíveis, também será necessário atribuir característica de cores e gravidade. A classe representada na Figura 6.12 atende a essa necessidade:

Figura 6.12 – Classe Partícula composta

```

1  public class Particula {
2      private float [] ponto_inicial;
3      private float [] direcao_movimento;
4      private float [] velocidade_movimento;
5      private float [] cores_particula;
6      private float gravidade;
7      private int tempo_vida;
8
9      public Particula(float[] ponto_inicial, float[] direcao_movimento,
10         float[] velocidade_movimento, float[] cores_particula, float gravidade,
11         int tempo_vida) {
12         this.ponto_inicial = ponto_inicial;
13         this.direcao_movimento = direcao_movimento;
14         this.velocidade_movimento = velocidade_movimento;
15         this.cores_particula = cores_particula;
16         this.gravidade = gravidade;
17         this.tempo_vida = tempo_vida;
18     }
19 }
20

```

Fonte: O autor (2018).

O desenvolvimento final da partícula nesse trabalho, produziu diversos efeitos com alguma particularidade, porém as características que todos possuem em comum serviram de base para definir o modelo de partícula dessa aplicação. Como exemplo, a Figura 6.13 demonstra a partícula do efeito chuva, seu método construtor é constituído por informações de posição, direção, cor, ângulo e velocidade. As responsabilidades de tempo de vida e gravidade são repassadas para o SP e programas de *Shader*.

Figura 6.13 – Partícula de Chuva

```

public Chuva(Geometry.Point posicao, Geometry.Vector direcao, int cor, float angulo, float velocidade) {
    this.posicao = posicao;
    this.direcao = direcao;
    this.cor = cor;
    // Atribuindo valores a matriz de direção, velocidade e angulo de variação
    this.angulo = angulo;
    this.velocidade = velocidade;
    vetorDeDirecao[0] = direcao.x;
    vetorDeDirecao[1] = direcao.y;
    vetorDeDirecao[2] = direcao.z;
    //Iniciando os extras
    extra = new Geometry.Point( x: 0.0f, y: 0.0f, z: 3.6f);
}

```

Fonte: O autor (2018).

6.5. Sistema de Partículas JAVA

O SP é o responsável pela emissão de partículas, alocação de memória e controle de particularidades dos efeitos. O modelo de SP de Brothaler (2013) possui 4 funções principais em Java, neste trabalho representados como: (1) método construtor SistemaParticulas, (2) addParticula, (3) *bindData* e (4) desenhar além de 12 variáveis de controle.

Figura 6.14 – Método construtor SistemaParticulas

```

public SistemaParticulas(int maximoDeParticulas) {
    /* Calcula o espaço de memoria de uma particula em um vetor estatico
    * o calculo se dá pelo numero maximo de particulas que serão usadas
    * multiplicado pelo espaço que os atributos dessa particula irá ocupar*/
    particulas = new float[maximoDeParticulas * CONTAGEM_TOTAL_COMPONENTES];
    /* aloca no vetor o espaço necessário armazenamento das particulas */
    vertexArray = new VertexArray(particulas);
    /* Atribui a contagem maxima de particulas ao atributo do Sistema de particulas*/
    this.maximoParticulas = maximoDeParticulas;
    /* Atribui valores zerados para o Extra*/
    this.extra = new Geometry.Point( x: 0.0f, y: 0.0f, z: 0.0f);
}

```

Fonte: O autor (2018).

O método construtor serve para construir o objeto SistemaParticulas, recebendo a quantidade máxima de partículas como parâmetro e inicializando as 4 variáveis exibidas na Figura 6.14. A variável *particulas* é um vetor que possui a quantidade de posições referente a quantidade máxima de partículas multiplicada pelo espaço de memória ocupado por seus componentes. A variável *vertexArray*, por sua vez é um objeto que recebe em seu parâmetro o vetor *particulas*. As duas últimas

variáveis definem a quantidade máxima de partículas do SP e um vetor de 3 posições para informações extras.

Figura 6.15 – Método de adição de partículas

```
public void addParticula(Geometry.Point posicao, int cor, Geometry.Vector direcao, float tempoInicial) {
    /* As variáveis de deslocamento servem para calcular o deslocamento total de memória entre os
    * componentes internos da partícula adicionada */
    final int deslocamentoInicial = proximaParticula * CONTAGEM_TOTAL_COMPONENTES;
    int deslocamentoAtual = deslocamentoInicial;
    proximaParticula++; // Guarda o ID da próxima partícula
    // Verifica se o número de partículas já chegou ao máximo
    if (particulaAtual < maximoParticulas) {
        particulaAtual++;
    }
    if (proximaParticula == maximoParticulas) {
        // Volta a contagem pra zero, matando as partículas antigas e reiniciando o envio de partículas
        proximaParticula = 0;
    }
    /* DADOS DAS PARTICULAS */
    // -- [ POSIÇÃO ] --
    particulas[deslocamentoAtual++] = posicao.x;
    particulas[deslocamentoAtual++] = posicao.y;
    particulas[deslocamentoAtual++] = posicao.z;
    // -- [ COR ] --
    particulas[deslocamentoAtual++] = Color.red(cor) / 255f;
    particulas[deslocamentoAtual++] = Color.green(cor) / 255f;
    particulas[deslocamentoAtual++] = Color.blue(cor) / 255f;
}
```

Fonte: O autor (2018).

O método `addParticula` (Figura 6.15) por sua vez, recebe como parâmetro, informações de cor, direção e o tempo atual em que a partícula nasceu. Dentro desse método é feito um controle para saber se a quantidade máxima de partículas já foi atingida, visto que se atingida, as partículas mais antigas morrem a fim de possibilitar o nascimento de novas partículas. Este método também coloca os dados recebidos no vetor de partículas e faz chamada ao objeto `vertexArray` para atualizar seus dados.

Figura 6.16 – Método `bindData`

```
public void bindData(ParticleShaderProgram programaDeParticulas)
{
    int dadoAtual = 0; // Coordenada do atributo atual (que esta sendo manipulado)
    // --[ Coordenadas de posição ]--
    vertexArray.setVertexAttribPointer(dadoAtual, programaDeParticulas.getLocalPosicao(),
        COMPONENTE_POSICAO, BYTESTOTAIS_STRIDE);
    dadoAtual += COMPONENTE_POSICAO;
    // --[ Coordenadas de cores ]--
    vertexArray.setVertexAttribPointer(dadoAtual, programaDeParticulas.getLocalCor(),
        COMPONENTE_COR, BYTESTOTAIS_STRIDE);
    dadoAtual += COMPONENTE_COR;
    // --[ Coordenadas de direção ]--
    vertexArray.setVertexAttribPointer(dadoAtual, programaDeParticulas.getLocalVetorDeDirecao(),
        COMPONENTE_VETOR, BYTESTOTAIS_STRIDE);
    dadoAtual += COMPONENTE_VETOR;
    // --[ Atributo de tempo de vida ]--
    vertexArray.setVertexAttribPointer(dadoAtual, programaDeParticulas.getLocalTempoInicialParticula(),
        COMPONENTE_TEMPO_INICIAL_PARTICULA, BYTESTOTAIS_STRIDE);
    dadoAtual += COMPONENTE_TEMPO_INICIAL_PARTICULA;
    // --[ Coordenadas de Extras ]--
    vertexArray.setVertexAttribPointer(dadoAtual, programaDeParticulas.getLocalInformacoesExtras(),
        PARTICULA_EXTRA_VEC3, BYTESTOTAIS_STRIDE);
    dadoAtual += PARTICULA_EXTRA_VEC3;
}
```

Fonte: O autor (2018).

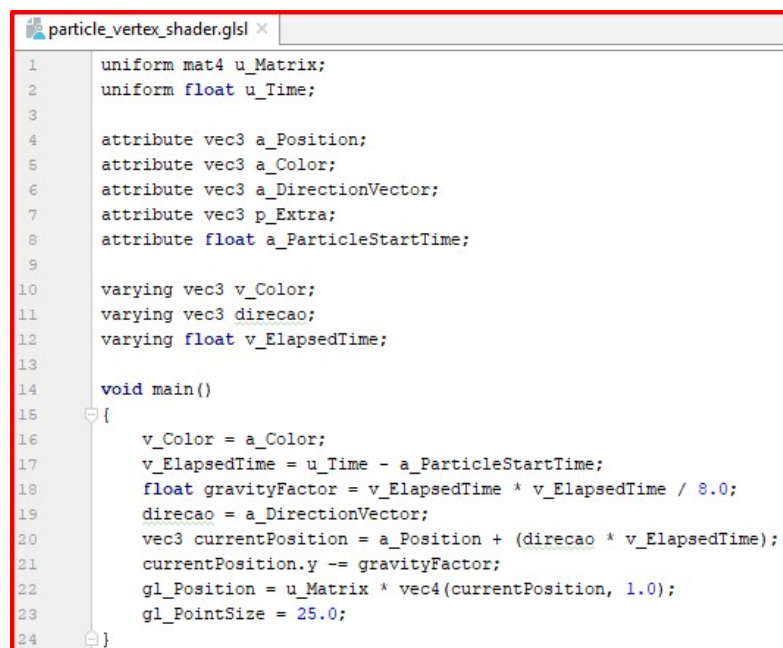
A Figura 6.16 demonstra o método *bindData*, que recebe um objeto *ParticleShaderProgram* como parâmetro, possibilitando que os atributos armazenados em *vertexArray* sejam repassados para a GPU e processados pelos programas de *shader*. Por fim, o método *desenho* dá o comando para que as partículas processadas pelos shaders sejam exibidas na superfície de renderização.

6.6. Arquivos de Shader

Os arquivos de *shader* são programas que possuem um papel importante neste projeto, segundo Brothaler (2013), eles informam o GPU como desenhar os dados recebidos (linhas, pontos, triângulos). Cada efeito deste trabalho é constituído por um programa *vertex shader* e um *fragment shader* escritos em GLSL. A codificação desses programas é similar à da linguagem C e cada arquivo vai ter uma responsabilidade diferente (capítulo 3, seção 3.10).

O arquivo *vertex shader* (capítulo 3, seção 3.8), fica responsável por processar a posição da partícula na tela, o tempo, direção e velocidade. Também é possível atribuir outros comportamentos, que podem ir desde a simulação de gravidade até a manipulação do tamanho da partícula na tela. A Figura 6.17 exibe um dos programas *vertex shader* deste trabalho.

Figura 6.17 – Arquivo Vertex Shader



```

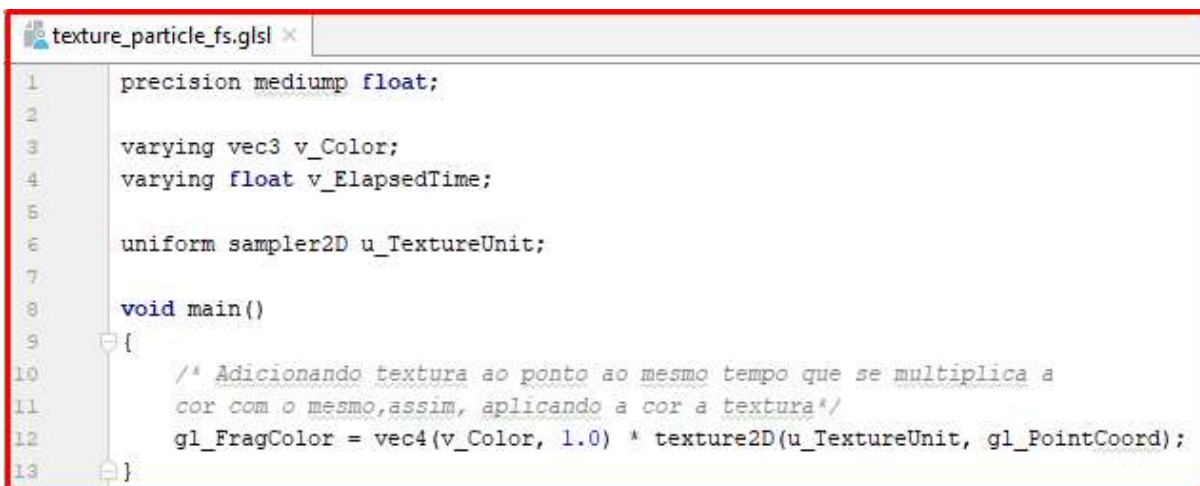
1  uniform mat4 u_Matrix;
2  uniform float u_Time;
3
4  attribute vec3 a_Position;
5  attribute vec3 a_Color;
6  attribute vec3 a_DirectionVector;
7  attribute vec3 p_Extra;
8  attribute float a_ParticleStartTime;
9
10 varying vec3 v_Color;
11 varying vec3 direcao;
12 varying float v_ElapsedTime;
13
14 void main()
15 {
16     v_Color = a_Color;
17     v_ElapsedTime = u_Time - a_ParticleStartTime;
18     float gravityFactor = v_ElapsedTime * v_ElapsedTime / 8.0;
19     direcao = a_DirectionVector;
20     vec3 currentPosition = a_Position + (direcao * v_ElapsedTime);
21     currentPosition.y -= gravityFactor;
22     gl_Position = u_Matrix * vec4(currentPosition, 1.0);
23     gl_PointSize = 25.0;
24 }

```

Fonte: O autor (2018).

Os arquivos de *fragment shader* (capítulo 3, seção 3.9) tem o objetivo de controlar as cores nos pedaços de tela ocupados pelas partículas. As definições de informações desse arquivo, podem ser tanto para permitir uso de textura, quanto de cor e se a cor muda com o tempo. A Figura 6.18 demonstra a codificação de um arquivo *fragment shader*.

Figura 6.18 – Arquivo Fragment Shader



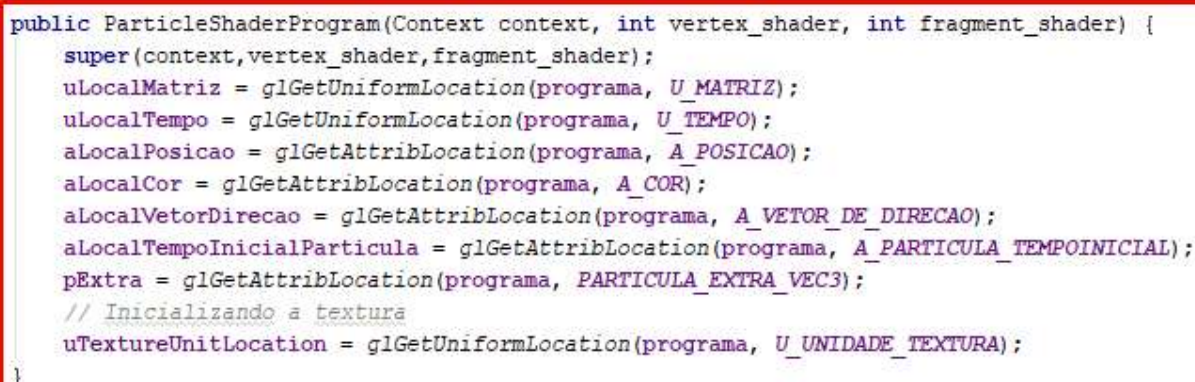
```

1 precision mediump float;
2
3 varying vec3 v_Color;
4 varying float v_ElapsedTime;
5
6 uniform sampler2D u_TextureUnit;
7
8 void main()
9 {
10     /* Adicionando textura ao ponto ao mesmo tempo que se multiplica a
11     cor com o mesmo, assim, aplicando a cor a textura*/
12     gl_FragColor = vec4(v_Color, 1.0) * texture2D(u_TextureUnit, gl_PointCoord);
13 }
  
```

Fonte: O autor (2018).

Existe uma classe que coordena a relação entre esses programas GLSL e o Java, trata-se do *ParticleShaderProgram*. Esta classe recebe três parâmetros em seu método construtor; (1) *context* (Interface de informações globais do aplicativo), (2) *vertex shader* e (3) *fragment shader*. Por meio dessas informações essa classe carrega arquivos de shader na memória da GPU (Figura 6.19).

Figura 6.19 – Método construtor de ParticleShaderProgram



```

public ParticleShaderProgram(Context context, int vertex_shader, int fragment_shader) {
    super(context, vertex_shader, fragment_shader);
    uLocalMatriz = glGetUniformLocation(programa, U_MATRIZ);
    uLocalTempo = glGetUniformLocation(programa, U_TEMPO);
    aLocalPosicao = glGetAttribLocation(programa, A_POSICAO);
    aLocalCor = glGetAttribLocation(programa, A_COR);
    aLocalVetorDirecao = glGetAttribLocation(programa, A_VETOR_DE_DIRECAO);
    aLocalTempoInicialParticula = glGetAttribLocation(programa, A_PARTICULA_TEMPOINICIAL);
    pExtra = glGetAttribLocation(programa, PARTICULA_EXTRA_VEC3);
    // Inicializando a textura
    uTextureUnitLocation = glGetUniformLocation(programa, U_UNIDADE_TEXTURA);
}
  
```

Fonte: O autor (2018).

6.7. Estrutura de Renderização

De acordo com Ginsburg e Purnomo (2014), existe uma estrutura para que um conjunto de códigos OpenGL ES seja funcional. Até o momento, nas seções anteriores do presente capítulo, foram cumprimos quase todos os requisitos descritos na seção 6.3, faltando definir a tela, criar uma superfície de renderização por intermédio do EGL, apagar as cores (para que possam ser redesenhadas) e escrever os dados na tela.

Para coordenar os itens que faltam, nesse projeto foram criadas algumas classes, denominadas *ParticulasActivity* e *ParticlesRenderer*. A primeira é responsável por executar as configurações iniciais a fim de permitir que uma tela seja definida. Também há a conversão de coordenadas de pixels as coordenadas geométricas do OpenGL, o código pode ser encontrado no apêndice A2. A segunda classe, fica responsável por definir a superfície de renderização usando a implementação do EGL e inicializar as demais configurações do SP.

Figura 6.20 – Configuração da superfície de renderização



```

32  glSurfaceView = new GLSurfaceView(getApplicationContext());
33  /* Pegar configurações do dispositivo e verificar se suporta Opengl ES 2 ou superior */
34  ActivityManager activityManager = (ActivityManager) getSystemService(Context.ACTIVITY_SERVICE);
35  /* pega informações do dispositivo a partir do contexto. */
36  ConfigurationInfo configurationInfo = activityManager.getDeviceConfigurationInfo();
37  /* Verifica se o dispositivo suporta openGL a partir da versão digitada e guarda um resultado bool */
38  final boolean supportsEs2 = configurationInfo.reqGlEsVersion >= 0x20000;
39  //final ParticlesRenderer particlesRenderer = new ParticlesRenderer(this);
40  final ParticlesRenderer particlesRenderer = new ParticlesRenderer(getApplicationContext());
41  if (supportsEs2) {
42      // Seta a informação de qual versão do OpenGL será usada na superfície
43      glSurfaceView.setEGLContextClientVersion(2);
44      // Inicia a superfície de renderização
45      glSurfaceView.setRenderer(particlesRenderer);
46      rendererSet = true;
47  }

```

Fonte: O autor (2018).

Demonstrado na Figura 6.21, na classe *ParticlesRenderer* é definido o tipo de efeito, a quantidade de partículas, a seleção dos arquivos de *shader* a serem usados, controle de tempo, textura, a promoção da limpeza das cores e o redesenho delas na tela.

Figura 6.21 – Configuração do sistema de Partículas

```

case "Fogo":
    if (usoTextura == 0) { //Não usa textura
        particleProgram = new ParticleShaderProgram(context, R.raw.efeito_fogo_particle_vs,
            R.raw.particle_fragment_shader);
        texture = TextureHelper.loadTexture(context, R.drawable.texture_fumaca);
    }
    else {
        particleProgram = new ParticleShaderProgram(context, R.raw.efeito_fogo_particle_vs,
            R.raw.texture_particle_fs);
        texture = TextureHelper.loadTexture(context, R.drawable.texture_fumaca);
    }
    direcaoParticulas = new Geometry.Vector(x: 1.3f, y: -2.0f, z: 0.0f);
    sistemaParticulas = new SistemaParticulas(maximoDeParticulas: 5000);
    break;

```

Fonte: O autor (2018).

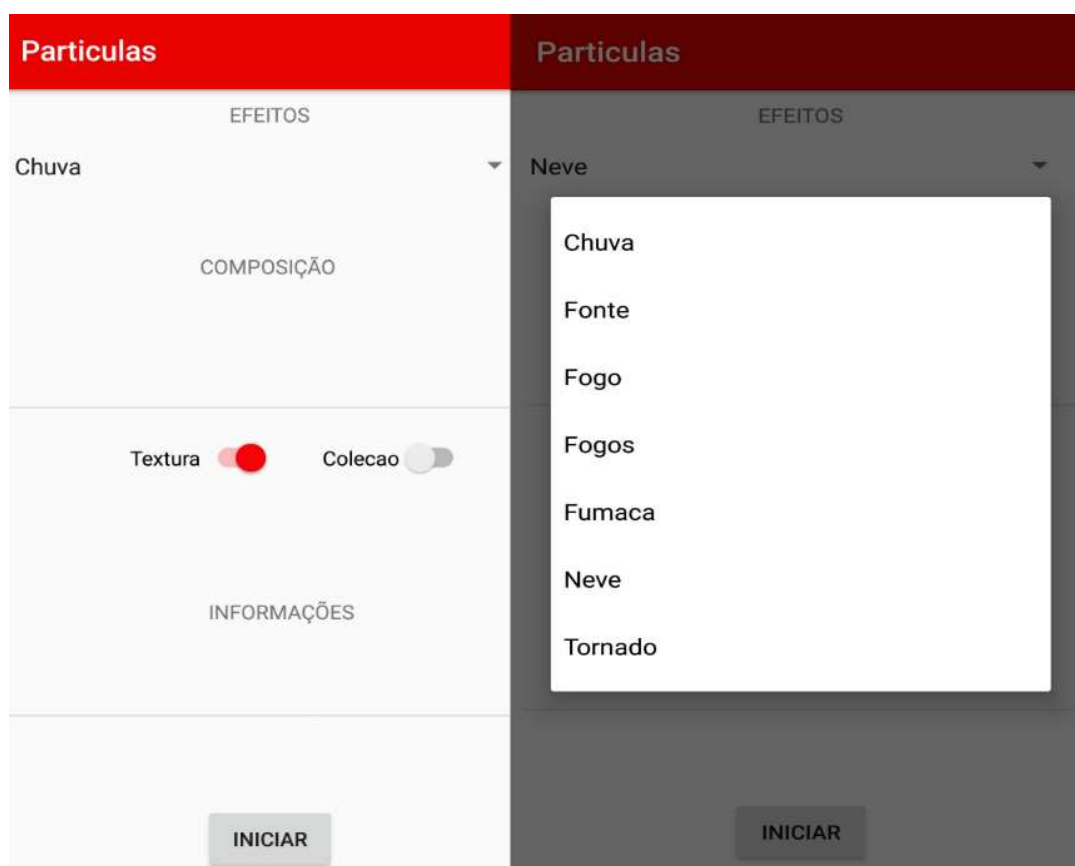
Esse conjunto de classes e códigos, são suficientes para que exista um SP, arquivos de configuração e a superfície necessária para executar as simulações. Toda a codificação do aplicativo *MBParticles* encontram-se nos Apêndices e Anexos.

7. RESULTADOS

7.1. Aplicativo MBParticles

O aplicativo MBParticles é o resultado da codificação feita nesse projeto, sua interface conta com um menu simples trazendo uma lista de efeitos e opção para realizar a simulação com ou sem texturas uso de texturas além de um botão para iniciar a simulação, conforme observado na Figura 7.1.

Figura 7.1 – Menu inicial do aplicativo

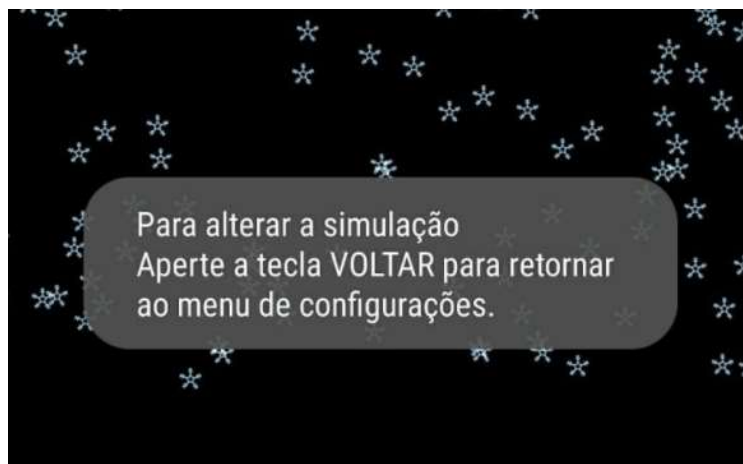


Fonte: O autor (2018).

Após tocar no botão iniciar, as opções escolhidas são salvas em arquivo usando a biblioteca *SharedPreferences*. Uma nova tela é iniciada já com uma superfície de renderização configurada, dentro dessa nova tela uma outra classe chamada *ParticlesRenderer* é instanciada e executada, as configurações salvas em arquivo são recuperadas e o efeito desejado no SP para execução imediata.

Para interromper a simulação ou escolher outro efeito, basta pressionar a tecla voltar do dispositivo e o menu será apresentado novamente. A Figura 7.2 demonstra a mensagem informando a possibilidade de alternância entre os efeitos.

Figura 7.2 – Dica para mudar efeito



Fonte: O autor (2018).

A sequência de funcionamento do MBParticles está representada no Pipeline da Figura 7.3. O primeiro item do pipeline é o Menu em seguida a Superfície de Renderização. Entre esses dois itens é onde ocorre o envio de informações (efeito, textura) definidas no menu e a definição das configurações de tela (superfície de renderização).

O segundo item do pipeline é a Superfície de Renderização, que tem entre suas responsabilidades, mostrar na tela o resultado do que é processado nos estágios posteriores.

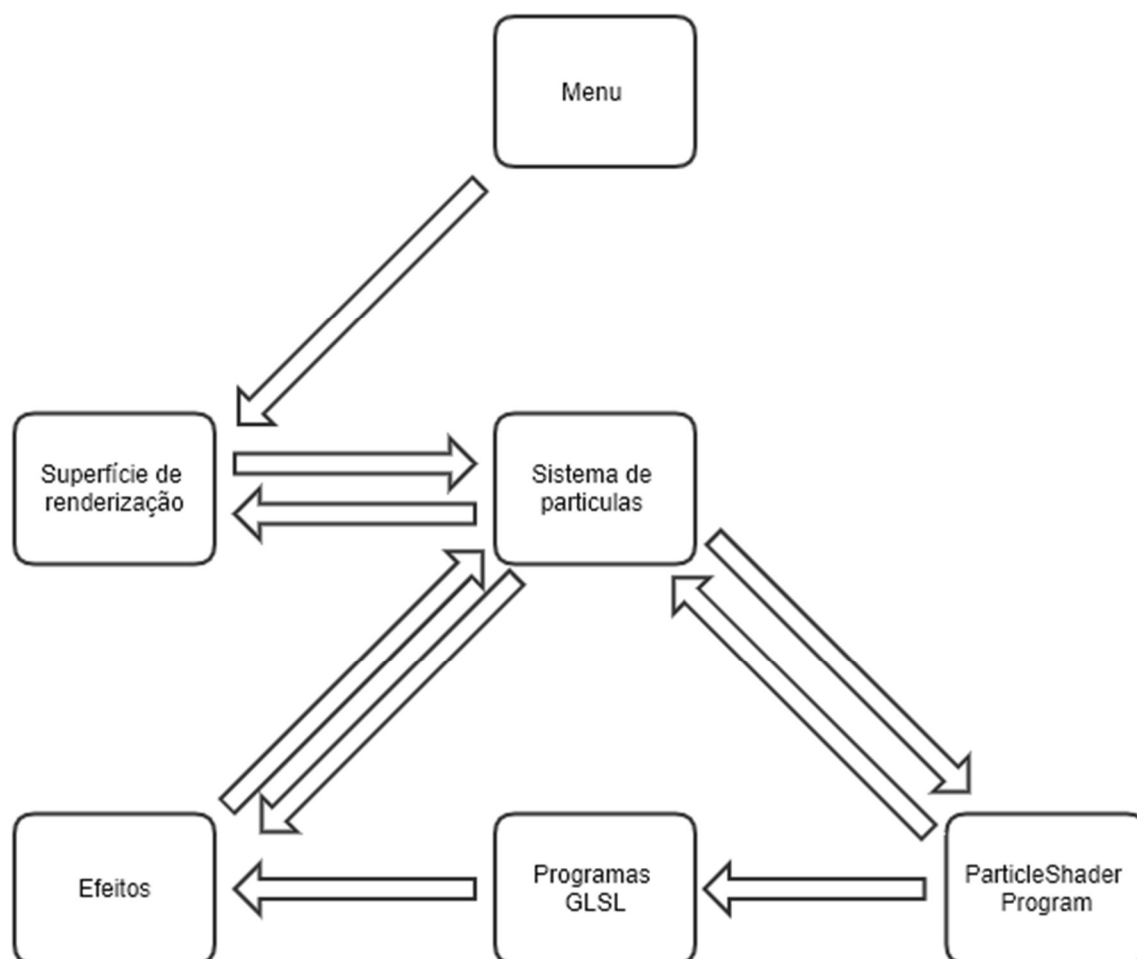
O próximo item, trata-se do Sistema de Partículas, onde são definidos as quantidades de fontes e partículas e o nome dos programas de *shader* relativos ao efeito que será simulado. As informações repassadas para o próximo item são os nomes dos programas de *shader* escolhidos.

O item *ParticleShaderProgram*, recebe os nomes dos programas e os localiza, uma vez localizados os carrega no pipeline do OpenGL ES (na GPU), constituindo então o item Programas GLSL. Caso nenhum erro ocorra, o *ParticleShaderProgram* retorna ao item Sistema de Partículas.

O Sistema de Partículas, com base nas informações reunidas, passa a lidar com os efeitos, o Item Efeitos representa as classes de efeitos e seus métodos, sendo um desses métodos o de adição de partículas no SP. O método de adição de partículas faz uso dos programas GLSL para processamento e emissão na Superfície

de Renderização. A representação no pipeline segue então o fluxo, onde o Item Sistema Partículas envia um pedido de emissão de partículas para o Item Efeitos, já esse item faz uso de informações contidas em Programas GLSL e então retorna pelo Sistema Partículas para exibir o conteúdo em Superfície de Renderização.

Figura 7.3 – Pipeline de funcionamento Aplicativo



Fonte: O autor (2018).

7.2. Efeito Chuva

Os efeitos criados nesse trabalho, são representados em 2D, dessa forma o eixo Z sempre terá valor 0, dessa forma lista de informações irá desconsiderar o eixo Z. No efeito Chuva, as partículas surgem no ponto inicial e percorrem um caminho na diagonal da esquerda para a direita de cima para baixo, também é feita a aplicação de textura, para que cada partícula tenha similaridade com um pingo. As informações

principais desse estão listadas abaixo. O Resultado da simulação pode ser visto na Figura 7.4.

- Quantidade partículas: 3 mil
- Posição inicial eixo X: randômico entre -1,3 e 4
- Posição inicial eixo Y: 3,8
- Direção eixo X: -1,5
- Direção Eixo Y: -3
- Variação de ângulo: 3°
- Arquivo *Vertex*: efeito_chuva_particle_vs
- Arquivo *Fragment*: particle_fragment_shader

Figura 7.4 – Efeito chuva



Fonte: O autor (2018).

A nível de código, o efeito Chuva, tem modificações no método de adição de partículas (Java), usando um atributo extra, um número randômico é enviado para o

Shader. No *vertex shader* esse número é usado para determinar o tamanho da partícula, assim cada pingo de chuva tem um tamanho em pixel, já o arquivo *fragment shader* usado faz o mapeamento de textura, dando um aspecto de pingo de chuva para cada partícula.

Figura 7.5 – Particularidade no Vertex shader do efeito Chuva

```
spawn = a_Position;

vec3 currentPosition = spawn + (direcao * v_ElapsedTime);

gl_Position = u_Matrix * vec4(currentPosition, 1.0);

gl_PointSize = p_Extra.x;
```

Fonte: O autor (2018).

Outros detalhes a serem observados, são as de configurações de quantidade máxima de 3 mil partículas, velocidade randomizando entre 1 e 2 e variação de ângulo de 3°. A adoção desses valores, permitem que cada partícula tenha uma velocidade e angulo diferente ao realizar sua trajetória na tela, sem que “morram” pelo controle de tempo de vida do SP.

Figura 7.6 – Definição de shaders, direção e quantidade de partículas

```
// -- [ Inicializa o Sistema de partículas e o Shader ] --
switch (this.tipoEfeito) {
    case "Chuva":
        if(usoTextura==0) { //Não usa textura
            particleProgram = new ParticleShaderProgram(context, R.raw.efeito_chuva_particle_vs,
                R.raw.particle_fragment_shader);
            texture = TextureHelper.loadTexture(context, R.drawable.texture_pingo);
        }
        else {
            texture = TextureHelper.loadTexture(context, R.drawable.texture_pingo);
            particleProgram = new ParticleShaderProgram(context, R.raw.efeito_chuva_particle_vs,
                R.raw.texture_particle_fs);
        }
        direcaoParticulas = new Geometry.Vector( x: -2.0f, y: -3.0f, z: 0.0f);
        sistemaParticulas = new SistemaParticulas( maximoDeParticulas: 3000);
        break;
```

Fonte: O autor (2018).

7.3. Efeito Faísca

O efeito Faísca possui interatividade, ou seja, quem manusear o aplicativo, poderá de alguma forma influenciar a simulação durante sua execução. O ponto de partida é definido no momento em que ocorre um toque na tela, nesse momento as partículas irão surgir e seguir em qualquer direção dentro do raio de 360 graus, conforme expresso através de setas na Figura 7.8.

- Quantidade partículas: 500
- Posição inicial eixo X: onde ocorrer toque na tela.
- Posição inicial eixo Y: onde ocorrer toque na tela.
- Variação de ângulo: 360°
- Direção eixo X: -2
- Direção Eixo Y: -3
- Arquivo *Vertex*: efeito_faísca_vs
- Arquivo *Fragment*: particle_fs_com_tempo

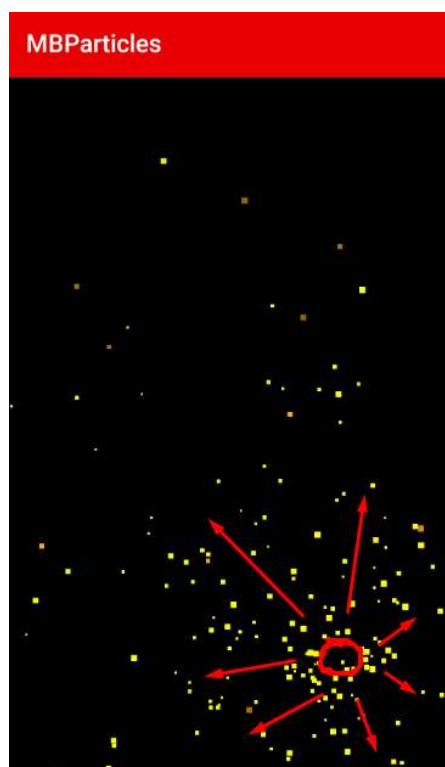
O código Java do efeito em questão, possui uma chamada à classe Randomizador no método de adição de partículas, seu uso visa gerar um número num intervalo de 5 a 15 *pixels* para que as partículas tenham tamanhos diferentes entre si, logo após esse valor é adicionado ao SP, a Figura 7.7 ilustra essas modificações.

Figura 7.7- Alteração no método de adição de partículas de Faísca

```
Randomizador a = new Randomizador();  
extra.x = extra.y = extra.z = a.randomNumero( max: 5, min: 15);  
sistemaParticulas.setExtra(extra);
```

Fonte: O autor (2018).

Figura 7.8 - Simulação efeito de Faísca



Fonte: O autor (2018).

Outro trecho de código em Java na classe *ParticlesRenderer* faz o tratamento dos dados captados na interação com a tela, trata-se do método *handleTouchDrag* que recebe e aplica os valores dos eixos X e Y para o efeito em questão. A figura 7.9 indica o código responsável.

Figura 7.9 - Código de interatividade do efeito Faísca

```
case "Faísca":
    partícula[0].setPosicao(new Geometry.Point(coordX, coordY, z: 0.0f));
    toqueDisplay=true;
    break;
```

Fonte: O autor (2018).

As últimas particularidades se encontram nos arquivos de shader, onde *vertex shader* aplica gravidade e recebe as informações extras contendo o tamanho aleatório das partículas e *fragment shader* aplica variação da cor conforme de acordo com o tempo da partícula, amarelado quando nova alternando ao laranja enquanto envelhece. As Figuras 7.10 e 7.11 mostram os arquivos *vertex* e *fragment shader* respectivamente.

Figura 7.10 – Arquivo vertex shader do efeito Faísca

```
void main()
{
    v_Color = a_Color;
    v_ElapsedTime = u_Time - a_ParticleStartTime;
    float gravityFactor = v_ElapsedTime * v_ElapsedTime / 8.0;
    vec3 currentPosition = a_Position + (a_DirectionVector * v_ElapsedTime);
    currentPosition.y -= gravityFactor;
    gl_Position = u_Matrix * vec4(currentPosition, 1.0);
    gl_PointSize = p_Extra.x;
}
```

Fonte: O autor (2018).

Figura 7.11 – Arquivo fragment shader do efeito Faísca

```
void main()
{
    //Aplicando cor com variação ao tempo
    gl_FragColor = vec4(v_Color / v_ElapsedTime, 1.0);
}
```

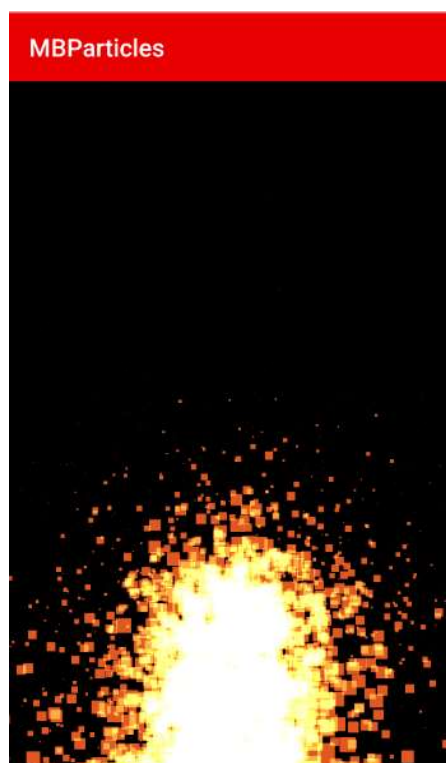
Fonte: O autor (2018).

7.4. Efeito Fogo

O efeito Fogo possui 4 fontes de partículas, dessa forma foi possível dar um aspecto símil a um fogo real. Esse efeito também possui interatividade, o código é similar ao apresentado na Figura 7.9, porém aplicados as 4 fontes de partículas. O resultado desta simulação é demonstrado na Figura 7.12.

- Quantidade partículas: 5 mil
- Posição inicial eixo X: randômico onde ocorrer toque na tela.
- Posição inicial eixo Y: de -0,6 a 0,4 e onde ocorrer toque na tela.
- Variação de ângulo: 180°
- Direção eixo X: 1,3
- Direção Eixo Y: -2
- Arquivo *Vertex*: efeito_faísca_vs
- Arquivo *Fragment*: particle_fs_com_tempo

Figura 7.12 - Simulação efeito de Fogo



Fonte: O autor (2018).

As informações listadas acima somadas as cores RGB em 226, 88 e 34 ajudam a dar cor e comportamento parecido com uma fogueira. A nível de código Java, a definição das informações iniciais ocorrem de forma similar a exibida nas Figuras 7.6 e 7.13.

Figura 7.13 - Aplicação de configurações iniciais, como cor, angulo e direção

```
case "Fogo":
    partícula [0] = new Particula(tipoEfeito,new Geometry.Point( x: 0f, y: 0.4f, z: 0f), direcaoParticulas,
        Color.rgb( red: 226, green: 88, blue: 34), angulo: 180, velocidade: 0.15f);
    partícula [1] = new Particula(tipoEfeito,new Geometry.Point( x: 0f, y: 0.2f, z: 0f), direcaoParticulas,
        Color.rgb( red: 226, green: 88, blue: 34), angulo: 180, velocidade: 0.15f);
    partícula [2] = new Particula(tipoEfeito,new Geometry.Point( x: 0f, y: -0.2f, z: 0f), direcaoParticulas,
        Color.rgb( red: 226, green: 88, blue: 34), angulo: 180, velocidade: 0.15f);
    partícula [3] = new Particula(tipoEfeito,new Geometry.Point( x: 0f, y: -0.6f, z: 0f), direcaoParticulas,
        Color.rgb( red: 226, green: 88, blue: 34), angulo: 180, velocidade: 0.25f);
    break;
```

Fonte: O autor (2018).

No método de adição de partículas, o efeito de fogo realiza algumas randomizações similares a que foi demonstrada na Figura 7.7, porém com valores do eixo X e dos argumentos extras. No shader existe uma repetição de códigos para fazer com que a partícula diminua de tamanho conforme seu tempo de vida passa. A repetição de códigos se fez necessária, visto as limitações descritas no capítulo 8 seção 8.3. A repetição de código pode ser vista na Figura 7.14.

Figura 7.14 - Código para diminuição de partículas com o tempo

```
gl_PointSize = 55.0-v_ElapsedTime-v_ElapsedTime-v_ElapsedTime-v_ElapsedTime
-v_ElapsedTime-v_ElapsedTime-v_ElapsedTime-v_ElapsedTime-v_ElapsedTime
-v_ElapsedTime-v_ElapsedTime-v_ElapsedTime-v_ElapsedTime-v_ElapsedTime
-v_ElapsedTime-v_ElapsedTime-v_ElapsedTime-v_ElapsedTime-v_ElapsedTime;
```

Fonte: O autor (2018).

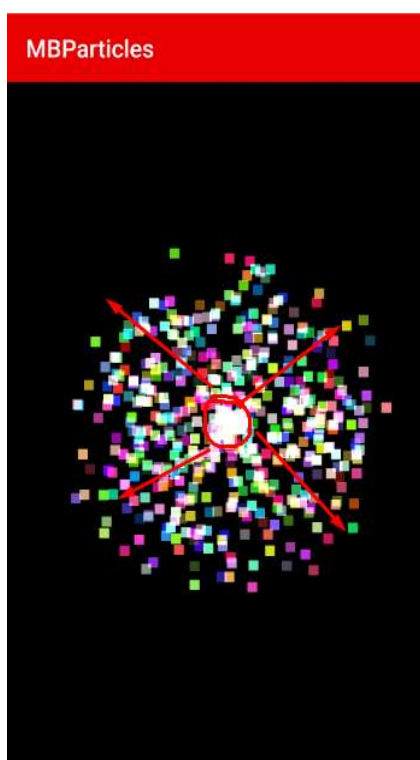
7.5. Efeito Fogos de Artifício

O efeito de Fogos, é similar ao efeito de Faísca, possui uma fonte de partículas e interatividade, diferindo no que diz respeito as informações iniciais, tamanho fixo de 25pixels no arquivo de *vertex shader* e uma adição de código no método *addParticulas*. O arquivo *fragment shader* não faz variação de cores com o tempo, assim conservando a cor inicial de cada partícula. O efeito pode ser observado na Figura 7.15, as setas desenhadas indicam a variação de ângulo em 360f graus.

- Quantidade partículas: 3mil
- Posição inicial eixo X: onde ocorrer toque na tela.

- Posição inicial eixo Y: onde ocorrer toque na tela.
- Variação de ângulo: 360°
- Direção eixo X: 0
- Direção Eixo Y: 3
- Arquivo *Vertex*: particle_vertex_shader
- Arquivo *Fragment*: particle_fragment_shader

Figura 7.15 - Simulação efeito de Fogos de Artifício



Fonte: O autor (2018).

No método de adição de partículas, a modificação de código, trata-se da geração individual de cores, são gerados 3 números inteiros aleatórios num intervalo de 0 a 255, cada resultado é somado em uma variável chamada *rgb* e são feitas operações de multiplicação usando operadores bit a bit, o resultado corresponde a uma posição de cor RGB. O código responsável por essa geração, pode ser visto na Figura 7.16.

Figura 7.16 - Código de geração randômica de cor RGB

```
Randomizador a = new Randomizador();
int rgb = a.randomNumeroInt( max: 255, min: 0); // Vermelho
rgb = (rgb << 8) + a.randomNumeroInt( max: 255, min: 0); //Verde
rgb = (rgb << 8) + a.randomNumeroInt( max: 255, min: 0); //Azul
```

Fonte: O autor (2018).

7.6. Efeito Fonte

Nesta simulação foram usadas 3 fontes de partículas, cada uma com sua própria direção, velocidade e quantidade de emissão. Existe também um mecanismo de interação, porém difere dos outros efeitos já se aplica apenas a fonte central, abaixando ou aumentando a distância em que as partículas sobem antes que a gravidade as faça descer.

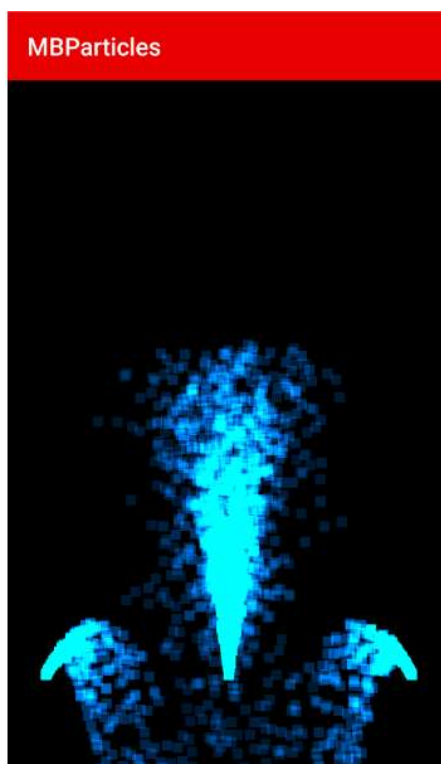
- Quantidade partículas: 5mil
- Posição inicial eixo X: onde ocorrer toque na tela.
- Posição inicial eixo Y: onde ocorrer toque na tela.
- Variação de ângulo: laterais 1°, centro 10°
- Direção eixo X: -0,1 lateral esquerda, 0,1 lateral direita e 0 central
- Direção Eixo Y: 0,2 para ambas laterais direita e 0,5 central
- Arquivo *Vertex*: particle_vertex_shader
- Arquivo *Fragment*: particle_fs_com_tempo

Todas as fontes possuem velocidade definida em 1, já a quantidade partículas emitidas nas fontes laterais foi configurada para uma por chamada do método *onDrawFrame* e a fonte central para 5. A cor definida foi Azul com RGB nos valores 0, 127 e 255. O efeito gerado pode ser observado na Figura 7.17.

Sobre o código em Java, possui uma leve modificação no método *handleTouchDrag* (citado na seção 7.3), para que as coordenadas captadas na interação sejam enviadas como informação extra para a fonte do meio, já dentro do método *addParticulas* essas informações extras são utilizadas como direção.

Os arquivos de *shader* fixam o tamanho das partículas em 25 *pixels* e variação de cores com o tempo.

Figura 7.17 - Simulação efeito de Fonte



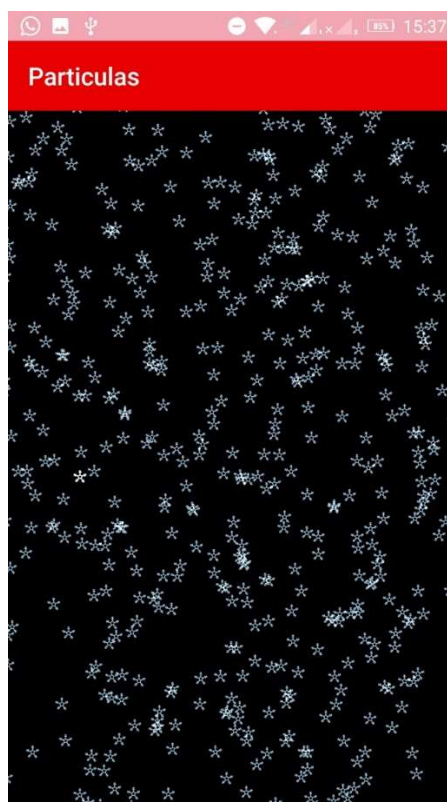
Fonte: O autor (2018).

7.7. Efeito Neve

No efeito de Neve as partículas surgem num ponto superior e descem na diagonal, porém a direção é randômica entre direita e esquerda, dessa forma apenas uma fonte de partículas é usada ao mesmo tempo que se obtém um comportamento parecido com o cair de neve. Outras características são; (1) uso de textura e (2) tamanho fixo em 40 *pixels* no *vertex shader*. Dessa forma dessa forma cada partícula assume a aparência de floco de neve e possuem um tamanho fixo para facilitar sua visualização. Outras informações importantes estão listadas abaixo e o resultado da simulação pode ser conferido na Figura 7.18.

- Quantidade partículas: 5mil
- Posição inicial eixo X: randômico entre -2,3 e 2,3.
- Posição inicial eixo Y: 3,8
- Variação de ângulo: 3°
- Direção eixo X: 1
- Direção Eixo Y: -3
- Arquivo *Vertex*: efeito_neve_particle_vs
- Arquivo *Fragment*: texture_particle_fs

Figura 7.18 – Efeito neve



Fonte: O autor (2018).

A nível de código, o efeito Neve, tem modificações no método `addParticulas` (Java), trata-se de inversão randômica no eixo X, para que cada partícula emitida possa seguir uma direção diferente, a Figura 7.19 exhibe o trecho de código.

Figura 7.19 – Particularidade no método de adição de partículas

```
// Alterando um eixo da direção
Randomizador inteiro = new Randomizador();
int num = inteiro.randomNumeroInt( max: 1, min: -1);
if (num==0)
    num=1;
direcaoParticula.x = direcaoParticula.x*num;
```

Fonte: O autor (2018).

8. PROBLEMAS E LIMITAÇÕES

8.1. Efeitos simultâneos

Cada efeito possui sua particularidade que é desenvolvida tanto no código em Java (no momento da emissão da partícula) quanto nos programas GLSL associado ao efeito. O SP permite apenas um arquivo *vertex shader* e *fragment shader* associados por vez, dessa forma apenas um efeito consegue conservar suas particularidades, impedindo que outros efeitos sejam simulados ao mesmo tempo.

8.2. Enviar dados parametrizados para o shader

Uma das primeiras ideias para o aplicativo era de parametrizar um espaço de coordenadas, onde as partículas iriam interagir de alguma forma, porém uma das dificuldades encontradas foi a de realizar o envio desses dados para o *shader*.

O envio não foi possível, por dois motivos; (1) alocação de memória para o *shader* ocorre quando o SP é instanciado, (2) o GLSL não possui estruturas complexas de vetor.

O primeiro impedimento ocorre por não ter como conhecer a quantidade exata de dados que serão captados durante na parametrização, não sendo possível então saber a quantidade de memória necessária a ser alocada.

O segundo impedimento se dá pela quantidade de dados obtidos durante a parametrização ser sempre superior a maior estrutura de dados disponível em GLSL, que até a versão 4.6 é um vetor de 4 posições (*vec4*).

8.3. Algoritmos não funcionaram no Shading

Em alternativa a parametrização, dois algoritmos foram implementados dentro do *vertex shader*, um para randomizar números reais e outro o de *bresenham* para selecionar coordenadas dado um grau inclinação. Contudo sempre que o aplicativo era compilado com algum desses algoritmos dentro de qualquer *vertex shader*, mesmo não gerando erros, o processamento do efeito na GPU não era realizado.

Também houve limitação ao tentar manipular variáveis *varyng*, impossibilitando que algumas multiplicações fossem feitas, o que implicou em repetição de códigos no efeito Fogo para redução do tamanho das partículas.

8.4. Recuperar dados do Shading para o Java

Outra ideia que não funcionou, foi a implementação de código em Java que fosse capaz de recuperar informações de partículas nos programas GLSL após já terem sido enviadas para a GPU. Com a recuperação de informações era esperado conseguir implementar duas funcionalidades:

- Permitir a geração de novas partículas a partir de partículas que atingissem um certo tempo de vida ou coordenada específica na tela. Com essa funcionalidade seria possível implementar respingos no efeito Chuva, quando a partícula chegasse a determinada coordenada. E com o controle de tempo, seria possível uma partícula partindo de um ponto e ao morrer gerar uma explosão com o efeito de fogos de artifício.
- Uma nova tentativa de usar a parametrização pelo Java ao invés de GLSL, observando quando as partículas alcançassem alguma coordenada parametrizada e então gerar alguma ação.

9. CONCLUSÃO

Percebeu-se a possibilidade de desenvolver SPs em aplicações *mobile*, com o uso de programas GLSL, por intermédio da API OpenGL ES e que, embora limitados em sua modelagem, tanto matemática, quanto nos requisitos de interação, obteve-se resultados razoáveis para os efeitos propostos. Assim, entendemos como promissora a pesquisa para a modelagem, desenvolvimento e interação deste tipo de renderização na plataforma abordada, atualizando, portanto, o estudo sobre SP.

Embora o SP desenvolvido não seja tão complexo e elaborado quanto outros SPs aplicados no mercado, parte da ideia contida na palavra promissora do parágrafo anterior diz respeito a melhoramentos deste SP. Com um SP que suporte a simulações de efeitos em 3D, poderia ser possível expandir sua aplicação, por exemplo, permitindo que usos similares aos descritos no capítulo 5 sejam desenvolvidos no contexto da programação *mobile*.

Seria interessante a aplicação deste SP em conjunto com tecnologias similares ao ARToolKit para aplicação de RA (vide capítulo 5 seção 5.1). Nesse sentido vale comentar que existem linhas de pesquisa atualizadas no contexto da programação *mobile*, por exemplo o artigo *Augmented Reality and ARToolkit for Android: the First Steps* proposto por Deminova (2016), demonstra a combinação de códigos Java com as bibliotecas ARToolKit em C, permitindo o uso do ARToolKit em dispositivos Android.

Outra ferramenta que pode ser usada para RA é a plataforma ARCore disponibilizada pela Google Developers (2018), porém os dispositivos demandarão de uma especificação superior à mínima necessária deste trabalho, o SP também deverá ser atualizado para OpenGL ES 3.0 ou mais recente.

Outra forma interessante de melhoria para este SP, seria a adição de algoritmos que permitissem o tratamento de colisão entre as partículas, a fim de tornar possível a aplicação do método SPH. Com aplicação do método SPH seria possível realizar simulações de fluídos.

Com a aplicação em conjunto das melhorias citadas acima, este SP poderia resultar na criação de um aplicativo de RA com um grande potencial de se integrar com realismo e naturalidade à realidade.

REFERÊNCIAS

- ACM SIGGRAPH. **About ACM SIGGRAPH**, 2013. Disponível em: <<https://www.siggraph.org/about/about-acm-siggraph>>. Acesso em: 2 maio 2018.
- ARTERO, A. O.; SANTOS, B. M. D. EFEITOS ESPECIAIS EM COMPUTAÇÃO GRÁFICA – MORPHING. **Colloquium Exactarum**, Presidente Prudente, v. 3, n. 2, p. 85-92, dez. 2011.
- ASSUNÇÃO, F. G. D. **Construindo Ambientes com Realidade Aumentada Utilizando Sistemas de Partículas**. Universidade Federal de Goiás. Catalogação. 2008.
- AZEVEDO, E.; CONCI, A. **Computação Gráfica: teoria e pratica**. São Paulo: Editora Campus, 2003.
- BROTHALER, K. **OpenGL ES 2 for Android: A Quick-Start Guide**. Dallas: The Pragmatic Bookshelf, 2013.
- COELHO, A. M. **Introdução a OpenGL**. Instituto Federal de Ciência e Tecnologia. Rio Pomba, p. 47. 2013.
- DEMINOVA, L. **Augmented Reality and ARToolkit for Android: the First Steps**. SHS Web Conf.29 02010. [S.l.]: SHS Web of Conferences. 2016. p. 1-4.
- DIAS, R. **40 Cenas de filmes antes e depois do poder dos efeitos especiais**, 2014. Disponível em: <<https://www.tudointeressante.com.br/2014/06/40-cenas-de-filmes-antes-e-depois-do-poder-dos-efeitos-especiais.html>>. Acesso em: 12 abr. 2018.
- GINSBURG, D.; PURNOMO, B. **OpenGL ES 3.0 Programming Guide**. 2ª. ed. Crawfordsville: Addison-Wesley Professional, 2014.
- GOOGLE DEVELOPERS. **Quickstart Android | ARCore | Google Developers**, 2018. Disponível em: <<https://developers.google.com/ar/develop/java/quickstart>>. Acesso em: 16 nov. 2018.
- HUGHES, J. F. et al. **Computer Graphics**. 3. ed. Willard: Addison-Wesley, 2013.
- INDIE GAME. **Unity 3D – Desenvolva seus jogos com maestria!**, 2015. Disponível em: <<http://www.indiegame.com.br/unity-3d-desenvolva-seus-jogos-com-maestria/>>. Acesso em: 26 mar. 2018.
- KEEPLAY GAME STUDIOS. **SimCMed Simulador 3D de Cirurgias Médicas**, 2010. Disponível em: <<http://www.keeplay.com/keeplay/serious-game/simcmmed>>. Acesso em: 6 abr. 2018.

KHRONOS GROUP. **Vertex Shader**, 2017. Disponível em:

<https://www.khronos.org/opengl/wiki/Vertex_Shader>. Acesso em: 25 out. 2018.

KHRONOS GROUP. **History of OpenGL**, 2017. Disponível em:

<http://www.khronos.org/opengl/wiki_opengl/index.php?title=History_of_OpenGL&oldid=14013>. Acesso em: 2 maio 2018.

KHRONOS GROUP. **Fragment Shader**, 2018. Disponível em:

<https://www.khronos.org/opengl/wiki/Fragment_Shader>. Acesso em: 25 out. 2018.

KHRONOS GROUP. **Visão Geral do OpenGL ES**, 04 abr. 2018. Disponível em:

<<https://www.khronos.org/opengles/>>. Acesso em: 3 abr. 2018.

LINARES, G. **O que é CGI e computação gráfica?**, 2012. Disponível em:

<<https://canaltech.com.br/software/O-que-e-CGI-e-computacao-grafica/>>. Acesso em: 18 abr. 2018.

MANSSOUR, I. H.; COHEN, M. **OpenGL Uma Abordagem Prática e Objetiva**. São Paulo: Novatec, 2006.

MELLO, B.; STAHNKE, F.; BEZ, M. R. **Projeto para desenvolvimento do Simulador**. Universidade Feevale. Novo Hamburgo. 2015.

MONET. **Bigode apagado digitalmente de Superman em 'Liga da Justiça' vira piada nas redes: "Shrek, é você?"**, 2017. Disponível em:

<<https://revistamonet.globo.com/Filmes/noticia/2017/11/bigode-apagado-digitalmente-de-superman-em-liga-da-justica-vira-piada-nas-redes-shrek-e-voce.html>>. Acesso em: 18 abr. 2018.

NVIDIA. **GeForce 256**, 2003. Disponível em:

<<http://www.nvidia.com/page/geforce256.html>>. Acesso em: 2 maio 2018.

OPUS SOFTWARE. **A Nova Geração de Aplicativos Móveis**, 2016. Disponível em:

<<https://www.opus-software.com.br/a-nova-geracao-de-aplicativos-moveis>>. Acesso em: 25 out. 2018.

QUEIROZ, T. E. **Animação computacional de escoamento de fluidos utilizando o método SPH**. Instituto de Ciências Matemáticas e de Computação. São Carlos. 2008.

RAYNE, E. **Firsts Futureworld and the futuristic evolution of 3D CGI effects in genre movies**, 2018. Disponível em: <<http://www.syfy.com/syfywire/firsts-futureworld-and-the-futuristic-evolution-of-3d-cgi-effects-in-genre-movies>>. Acesso em: 2 maio 2018.

REEVES, W. T. Particle Systems - Technique for Modeling a Class of Fuzzy Objects. **ACM Transactions on Graphics**, v. 2, n. 2, p. 91-106, Julho 1983.

SILVA, R. D. D. **Unity - Guia Completa sobre a Game Engine**, 2017. Disponível em: <<https://producaodejogos.com/unity/>>. Acesso em: 12 abr. 2018.

TUTORIAL 45. **3D Design project for beginners – AutoCAD**, 2017. Disponível em: <<http://tutorial45.com/3d-design-project-for-beginners-autocad>>. Acesso em: 10 abr. 2018.

UNITY TECHNOLOGIES. **Unity - Fast Facts**, 2018. Disponível em: <<https://unity3d.com/pt/public-relations>>. Acesso em: 24 abr. 2018.

APÊNDICES

A – ACTIVITY

A.1 Classe MainActivity (JAVA)

```
package com.particles.android;

import android.annotation.SuppressLint;
import android.content.Intent;
import android.content.SharedPreferences;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.Spinner;
import android.widget.Switch;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class MainActivity extends AppCompatActivity {
    // --- [ BOTÕES ] ---
    private Button bt_iniciar;
    // --- [ EFEITOS ] ---
    private Spinner comboEfeitos;
    // --- [ TEXTURA ] ---
    private Switch simTextura;
    private Switch naoTextura;
    // --- [ ARQUIVO ] ---
    private static final String ARQUIVO_CONFIGURACAO = "configParametro";
    private SharedPreferences;
    SharedPreferences.Editor arquivoEditor;

    @SuppressLint("ClickableViewAccessibility")
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // --- [ ARQUIVO ] ---
        sharedPreferences = getSharedPreferences(ARQUIVO_CONFIGURACAO, 0);
        //Arquivo e modo de leitura (0 é privado)
        arquivoEditor = sharedPreferences.edit();
        // --- [ BOTÕES ] ---
        bt_iniciar=(Button)findViewById(R.id.bt_iniciar);
        // --- [ EFEITOS ] ---
        comboEfeitos=(Spinner)findViewById(R.id.comboEfeitos);
        List<String> listaEfeitos = new
        ArrayList<>(Arrays.asList("Chuva", "Faisca", "Fogo", "Fogos", "Fonte", "Ne
ve"));
        ArrayAdapter<String> dataAdapter = new ArrayAdapter<String>(this,
        android.R.layout.simple_spinner_item, listaEfeitos );
        dataAdapter.setDropDownViewResource(android.R.layout.simple_spinner_d
ropdown_item);
        comboEfeitos.setAdapter(dataAdapter);
    }
}
```

```

// --- [ SELECAO TEXTURA ] ---
// Link dos objetos Switch
simTextura = (Switch)findViewById(R.id.sParaDes);
simTextura.setChecked(true);
naoTextura = (Switch)findViewById(R.id.sParaLin);
// --- [ BOTÕES ] ---
bt_iniciar.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        /* Grava em arquivo o Efeito escolhido e se o mesmo fará
        uso de textura ou não*/

arquivoEditor.putString("efeito", comboEfeitos.getSelectedItem().toString())
;;

        if(simTextura.isChecked()) {
            arquivoEditor.putInt("textura", 1);
        }
        if(naoTextura.isChecked()) {
            arquivoEditor.putInt("textura", 0);
        }
        arquivoEditor.commit(); // gravando configurações
        // Sai do menu e entra na tela de Partículas
        Intent = new Intent(getApplicationContext(),
ParticulasActivity.class);
        startActivity(intent);
    }
});
simTextura.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        if(simTextura.isChecked())
            naoTextura.setChecked(false);
        else
            naoTextura.setChecked(true);
    }
});
naoTextura.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        if(naoTextura.isChecked())
            simTextura.setChecked(false);
        else
            simTextura.setChecked(true);
    }
});
}
}

```

A.2 Classe ParticulasActivity (JAVA)

```

package com.particles.android;

import android.annotation.SuppressLint;
import android.app.ActivityManager;
import android.content.Context;
import android.content.SharedPreferences;
import android.content.pm.ConfigurationInfo;
import android.opengl.GLSurfaceView;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.MotionEvent;
import android.view.View;
import android.widget.Toast;

public class ParticulasActivity extends AppCompatActivity {
    private GLSurfaceView;
    private boolean rendererSet = false;
    // --- [ ARQUIVO ] ---
    private static final String ARQUIVO_CONFIGURACAO = "configParametro";
    //constant
    private SharedPreferences sharedPreferences;
    SharedPreferences.Editor arquivoEditor;

    @SuppressLint("ClickableViewAccessibility")
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_particulas);
        // --- [ ARQUIVO ] ---
        sharedPreferences = getSharedPreferences(ARQUIVO_CONFIGURACAO, 0);
        //Arquivo e modo de leitura (0 é privado)
        arquivoEditor = sharedPreferences.edit();
        // GL Surface
        glSurfaceView = new GLSurfaceView(getApplicationContext());
        /* Pegar configurações do dispositivo e verificar se suporta OpenGL
        ES 2 ou superior */
        ActivityManager activityManager = (ActivityManager)
        getSystemService(Context.ACTIVITY_SERVICE);
        /* pega informações do dispositivo a partir do contexto. */
        ConfigurationInfo configurationInfo =
        activityManager.getDeviceConfigurationInfo();
        /* Verifica se o dispositivo suporta OpenGL a partir da versão
        digitada e guarda um resultado bool */
        final boolean supportsEs2 = configurationInfo.reqGlEsVersion >=
        0x20000;
        //final ParticlesRenderer particlesRenderer = new ParticlesRenderer(this);
        final ParticlesRenderer particlesRenderer = new
        ParticlesRenderer(getApplicationContext());
        if (supportsEs2) {
            // Seta a informação de qual versão do OpenGL será usada na superfície
            glSurfaceView.setEGLContextClientVersion(2);
            // Inicia a superfície de renderização
            glSurfaceView.setRenderer(particlesRenderer);
            rendererSet = true;
        }
    }
}

```

```

else { // não suporta OpenGL ES da versão
    Toast.makeText(getApplicationContext(), "Esse dispositivo não
suporta OpenGL ES 2.0 ou superior.", Toast.LENGTH_LONG).show();
    return;
}
glSurfaceView.setOnTouchListener(new View.OnTouchListener()
{
    @Override
    public boolean onTouch(View v, MotionEvent event) {
        if (event != null) {
            /* Converte as coordenadas de pixel para coordenadas
reconhecidas pelo OpenGL
            * Obs: No Android as coordenadas de pixel são
invertidas em relação ao OpenGL */
            float cordX = (float) v.getWidth()/2.6f;
            cordX = event.getX()/cordX;
            cordX = cordX-1.3f;
            final float coordenadaGeometricaX = cordX;
            /* 2,6 foi o intervalo encontrado entre o menor valor
vizível em X e o maior (-1,3 a 1,3) */
            float cordY = (float) v.getHeight()/4.2f;
            cordY = ((event.getY()-v.getHeight())*-1.0f)/cordY;
            cordY = cordY-0.6f;
            final float coordenadaGeometricaY = cordY;
            if (event.getAction() == MotionEvent.ACTION_MOVE) {
                glSurfaceView.queueEvent(new Runnable()
                {
                    @Override
                    public void run()
                    {
                        particlesRenderer.handleTouchDrag(coordenadaGeometricaX,
coordenadaGeometricaY);
                    }
                });
            }
            return true;
        } else {
            return false;
        }
    }
});
Toast.makeText(getApplicationContext(), "Para alterar a simulação"
+
    "\nAperte a tecla VOLTAR para retornar" +
    "\nnao menu de configurações.", Toast.LENGTH_SHORT).show();
setContentView(glSurfaceView);
}
@Override
protected void onPause() {
    super.onPause();
    if (rendererSet) {
        glSurfaceView.onPause();
    }
}
@Override
protected void onResume() {
    super.onResume();
    if (rendererSet) {
        glSurfaceView.onResume();
    }
}

```


B – EFEITOS

B.1 Classe Chuva (JAVA)

```
package com.particles.android.efeitos;

import com.particles.android.objects.SistemaParticulas;
import com.particles.android.util.Geometry;
import com.particles.android.util.Randomizador;

import java.util.Random;

import static android.opengl.Matrix.multiplyMV;
import static android.opengl.Matrix.setRotateEulerM;

public class Chuva {
    private Geometry.Point posicao;
    private final Geometry.Vector direcao;
    private final int cor;
    /* Nesta versão, o atirador de particulas agora terá funções que
    randomizam
    * algumas informações como; angulo, velocidade, direção e etc*/
    private final float angulo; // Variação de Angulo
    private final float velocidade; // Variação de velocidade
    private final Random random = new Random(); // Randomizador
    private float[] matrizDeRotacao = new float[16]; // Matriz de rotação
    private float[] vetorDeDirecao = new float[4]; // Matriz de direção
    private float[] vetorResultante = new float[4]; // Matriz resultante
    // -- [ INFORMAÇÕES VEC3 EXTRA ] --
    private Geometry.Point extra;
    // -- [ MÉTODO CONSTRUTOR ] --
    public Chuva(Geometry.Point posicao, Geometry.Vector direcao, int cor,
    float angulo, float velocidade) {
        this.posicao = posicao;
        this.direcao = direcao;
        this.cor = cor;
        // Atribuindo valores a matriz de direção, velocidade e angulo de
    variação
        this.angulo = angulo;
        this.velocidade = velocidade;
        vetorDeDirecao[0] = direcao.x;
        vetorDeDirecao[1] = direcao.y;
        vetorDeDirecao[2] = direcao.z;
        //Iniciando os extras
        extra = new Geometry.Point(0.0f,0.0f,3.6f);
    }
    // -- [ ADICIONA PARTICULAS AO SISTEMA DE PARTICULAS ] --
    public void addParticulas(SistemaParticulas sistemaParticulas, float
    tempoAtual, int quantidade) {
        for (int i = 0; i < quantidade; i++) {
            // Altera o angulo para onde a partícula ta sendo atirada
            setRotateEulerM(matrizDeRotacao, 0, (random.nextFloat() - 0.5f) *
    angulo,
                (random.nextFloat() - 0.5f) * angulo, (random.nextFloat() -
    0.5f) * angulo);
            // É feita a multiplicação da matriz para que o vetor fique "rodado"
            multiplyMV(vetorResultante, 0, matrizDeRotacao, 0, vetorDeDirecao,
    0);
        }
    }
}
```

```

// Ajuste de velocidade
    float ajusteVelocidade = 1f + random.nextFloat() * velocidade;
    // Ajuste de angulo
    Geometry.Vector direcaoParticula = new
Geometry.Vector(vetorResultante[0] * ajusteVelocidade,
                vetorResultante[1] * ajusteVelocidade,
vetorResultante[2] * ajusteVelocidade);
    // Adicionando particula
    Randomizador a = new Randomizador();
    float posX= a.randomNumero(-1.3f,4.0f);
    extra.x = extra.y = extra.z = a.randomNumero(30.0f,55.5f);
    sistemaParticulas.setExtra(extra);
    sistemaParticulas.addParticula(new
Geometry.Point(posX,posicao.y,posicao.z), cor, direcaoParticula,
tempoAtual);
    }
    /*Obs: Uma simples estrutura de repetição que adiciona a quantidade
definida na fonte de particulas ao sistema*/
    }

    public void setPosicao(Geometry.Point posicao) {
        this.posicao = posicao;
    }

    public void setExtra(Geometry.Point extra) {
        this.extra = extra;
    }
}

```

B.2 Shader Chuva (GLSL)

```

uniform mat4 u_Matrix;
uniform float u_Time;

attribute vec3 a_Position;
attribute vec3 a_Color;
attribute vec3 a_DirectionVector;
attribute float a_ParticleStartTime;
attribute vec3 p_Extra;

varying vec3 v_Color;
varying vec3 direcao;
varying vec3 spawn;
varying float v_ElapsedTime;

void main()
{
    v_Color = a_Color;
    v_ElapsedTime = u_Time - a_ParticleStartTime;

    direcao = a_DirectionVector;

    spawn = a_Position;

    vec3 currentPosition = spawn + (direcao * v_ElapsedTime);

    gl_Position = u_Matrix * vec4(currentPosition, 1.0);

    gl_PointSize = p_Extra.x;
}

```

B.3 Classe Faísca (JAVA)

```

package com.particles.android.efeitos;

import com.particles.android.objects.SistemaParticulas;
import com.particles.android.util.Geometry;
import com.particles.android.util.Randomizador;
import java.util.Random;
import static android.opengl.Matrix.multiplyMV;
import static android.opengl.Matrix.setRotateEulerM;

public class Faísca {
    private Geometry.Point posicao;
    private final Geometry.Vector direcao;
    private final int cor;
    private final float angulo; // Variação de Angulo
    private final float velocidade; // Variação de velocidade
    private final Random random = new Random(); // Randomizador
    private float[] matrizDeRotacao = new float[16]; // Matriz de rotação
    private float[] vetorDeDirecao = new float[4]; // Matriz de direção
    private float[] vetorResultante = new float[4]; // Matriz resultante
    // -- [ INFORMAÇÕES VEC3 EXTRA ] --
    private Geometry.Point extra;
    // -- [ MÉTODO CONSTRUTOR ] --
    public Faísca(Geometry.Point posicao, Geometry.Vector direcao, int cor,
float angulo, float velocidade)
    {
        this.posicao = posicao;
        this.direcao = direcao;
        this.cor = cor;
        // Atribuindo valores a matriz de direção, velocidade e angulo de
variação
        this.angulo = angulo;
        this.velocidade = velocidade;
        vetorDeDirecao[0] = direcao.x;
        vetorDeDirecao[1] = direcao.y;
        vetorDeDirecao[2] = direcao.z;
        //Inicializando os extras
        extra = new Geometry.Point(0.0f,0.0f,3.6f);
    }

    // -- [ ADICIONA PARTICULAS AO SISTEMA DE PARTICULAS ] --
    public void addParticulas(SistemaParticulas sistemaParticulas, float
tempoAtual, int quantidade) {
        for (int i = 0; i < quantidade; i++) {
            // Altera o angulo para onde a partícula ta sendo atirada
            setRotateEulerM(matrizDeRotacao, 0, (random.nextFloat() - 0.5f) *
angulo, (random.nextFloat() - 0.5f) * angulo, (random.nextFloat() - 0.5f) *
angulo);
            // É feita a multiplicação da matriz para que o vetor fique
"rodado"
            multiplyMV(vetorResultante, 0, matrizDeRotacao, 0, vetorDeDirecao,
0);
            // Ajuste de velocidade
            float ajusteVelocidade = 1f + random.nextFloat() * velocidade;

```

```

// Ajuste de angulo
    Geometry.Vector direcaoParticula = new
Geometry.Vector(vetorResultante[0] * ajusteVelocidade,
                vetorResultante[1] * ajusteVelocidade,
vetorResultante[2] * ajusteVelocidade);
    // Adicionando particula
    Randomizador a = new Randomizador();
    extra.x = extra.y = extra.z = a.randomNumero(5,15);
    sistemaParticulas.setExtra(extra);
    sistemaParticulas.addParticula(new
Geometry.Point(posicao.x,posicao.y,posicao.z), cor, direcaoParticula,
tempoAtual);
    }
    /*Obs: Uma simples estrutura de repetição que adiciona a quantidade
definida na fonte de particulas ao sistema*/
}

    public void setPosicao(Geometry.Point posicao) {
        this.posicao = posicao;
    }

    public void setExtra(Geometry.Point extra) {
        this.extra = extra;
    }
}

```

B.4 Shader Faísca (GLSL)

```

uniform mat4 u_Matrix;
uniform float u_Time;

attribute vec3 a_Position;
attribute vec3 a_Color;
attribute vec3 a_DirectionVector;
attribute vec3 p_Extra;
attribute float a_ParticleStartTime;

varying vec3 v_Color;
varying float v_ElapsedTime;

void main()
{
    v_Color = a_Color;
    v_ElapsedTime = u_Time - a_ParticleStartTime;
    float gravityFactor = v_ElapsedTime * v_ElapsedTime / 8.0;
    vec3 currentPosition = a_Position + (a_DirectionVector *
v_ElapsedTime);
    currentPosition.y -= gravityFactor;
    gl_Position = u_Matrix * vec4(currentPosition, 1.0);
    gl_PointSize = p_Extra.x;
}

```

B.5 Classe Fogo (JAVA)

```

package com.particles.android.efeitos;

import com.particles.android.objects.SistemaParticulas;
import com.particles.android.util.Geometry;
import com.particles.android.util.Randomizador;
import java.util.Random;
import static android.opengl.Matrix.multiplyMV;
import static android.opengl.Matrix.setRotateEulerM;

public class Fogo {
    private Geometry.Point posicao;
    private final Geometry.Vector direcao;
    private final int cor;
    private final float angulo; // Variação de Angulo
    private final float velocidade; // Variação de velocidade
    private final Random random = new Random(); // Randomizador
    private float[] matrizDeRotacao = new float[16]; // Matriz de rotação
    private float[] vetorDeDirecao = new float[4]; // Matriz de direção
    private float[] vetorResultante = new float[4]; // Matriz resultante
    // -- [ INFORMAÇÕES VEC3 EXTRA ] --
    private Geometry.Point extra;
    // -- [ MÉTODO CONSTRUTOR ] --
    public Fogo(Geometry.Point posicao, Geometry.Vector direcao, int cor,
float angulo, float velocidade) {
        this.posicao = posicao;
        this.direcao = direcao;
        this.cor = cor;
        // Atribuindo valores a matriz de direção, velocidade e angulo de
        variação
        this.angulo = angulo;
        this.velocidade = velocidade;
        vetorDeDirecao[0] = direcao.x;
        vetorDeDirecao[1] = direcao.y;
        vetorDeDirecao[2] = direcao.z;
        //Iniciando os extras
        extra = new Geometry.Point(0.0f,0.0f,3.6f);
    }
    // -- [ ADICIONA PARTICULAS AO SISTEMA DE PARTICULAS ] --
    public void addParticulas(SistemaParticulas sistemaParticulas, float
tempoAtual, int quantidade) {
        for (int i = 0; i < quantidade; i++) {
            // Altera o angulo para onde a partícula ta sendo atirada
            setRotateEulerM(matrizDeRotacao, 0, (random.nextFloat() - 0.5f) *
angulo,
                (random.nextFloat() - 0.5f) * angulo, (random.nextFloat() -
0.5f) * angulo);
            // Ajuste de velocidade
            float ajusteVelocidade = -velocidade;

```

```

        // Adicionando particula
        Randomizador a = new Randomizador();
        float posX= a.randomNumero(-0.3f,0.3f);
        float extraX= a.randomNumero(-2.3f,2.3f);
        float extraY= a.randomNumero(1.3f,3.5f);
        float extraZ= a.randomNumero(0.3f,0.0f);
        extra.x=extraX;
        extra.y=extraY;
        extra.z=extraZ;
        multiplyMV(vetorResultante, 0, matrizDeRotacao, 0,
vetorDeDirecao, 0);
        // Ajuste de velocidade
        Geometry.Vector direcaoParticula = new
Geometry.Vector(vetorResultante[0] * ajusteVelocidade,
                vetorResultante[1] * ajusteVelocidade,
vetorResultante[2] * ajusteVelocidade);
        sistemaParticulas.addParticula(new
Geometry.Point(posX+posicao.x,posicao.y,posicao.z), cor, direcaoParticula,
tempoAtual);
    }
    /*Obs: Uma simples estrutura de repetição que adiciona a quantidade
definida na fonte de particulas ao sistema*/
}

    public void setPosicao(Geometry.Point posicao) {
        this.posicao = posicao;
    }

    public void setExtra(Geometry.Point extra) {
        this.extra = extra;
    }
}

```

B.6 Shader Fogo (GLSL)

```

uniform mat4 u_Matrix;
uniform float u_Time;

attribute vec3 a_Position;
attribute vec3 a_Color;
attribute vec3 a_DirectionVector;
attribute float a_ParticleStartTime;
attribute vec3 p_Extra;

varying vec3 v_Color;
varying vec3 direcao;
varying vec3 spawn;
varying float v_ElapsedTime;

void main()
{
    v_Color = a_Color;
    v_ElapsedTime = u_Time - a_ParticleStartTime;

    direcao = a_DirectionVector;

    spawn = a_Position;

    direcao.x = direcao.x+0.05;
    direcao.y = direcao.y+0.05;

    vec3 currentPosition = spawn + (direcao * v_ElapsedTime);

    gl_Position = u_Matrix * vec4(currentPosition, 1.0);

    gl_PointSize = 55.0-v_ElapsedTime-v_ElapsedTime-v_ElapsedTime-
v_ElapsedTime
-v_ElapsedTime-v_ElapsedTime-v_ElapsedTime-v_ElapsedTime-v_ElapsedTime
-v_ElapsedTime-v_ElapsedTime-v_ElapsedTime-v_ElapsedTime-v_ElapsedTime
-v_ElapsedTime-v_ElapsedTime-v_ElapsedTime-v_ElapsedTime-v_ElapsedTime;
}

```

B.7 Classe Fogos (JAVA)

```

package com.particles.android.efeitos;

import com.particles.android.objects.SistemaParticulas;
import com.particles.android.util.Geometry;
import com.particles.android.util.Randomizador;
import java.util.Random;
import static android.opengl.Matrix.multiplyMV;
import static android.opengl.Matrix.setRotateEulerM;

public class Fogos {
    private Geometry.Point posicao;
    private final Geometry.Vector direcao;
    private final int cor;
    private final float angulo; // Variação de Angulo
    private final float velocidade; // Variação de velocidade
    private final Random random = new Random(); // Randomizador
    private float[] matrizDeRotacao = new float[16]; // Matriz de rotação
    private float[] vetorDeDirecao = new float[4]; // Matriz de direção
    private float[] vetorResultante = new float[4]; // Matriz resultante
    // -- [ INFORMAÇÕES VEC3 EXTRA ] --
    private Geometry.Point extra;
    // -- [ MÉTODO CONSTRUTOR ] --
    public Fogos(Geometry.Point posicao, Geometry.Vector direcao, int cor,
float angulo, float velocidade) {
        this.posicao = posicao;
        this.direcao = direcao;
        this.cor = cor;
        // Atribuindo valores a matriz de direção, velocidade e angulo de
        variação
        this.angulo = angulo;
        this.velocidade = velocidade;
        vetorDeDirecao[0] = direcao.x;
        vetorDeDirecao[1] = direcao.y;
        vetorDeDirecao[2] = direcao.z;
        //Iniciando os extras
        extra = new Geometry.Point(0.0f,0.0f,3.6f);
    }
    // -- [ ADICIONA PARTICULAS AO SISTEMA DE PARTICULAS ] --
    public void addParticulas(SistemaParticulas sistemaParticulas, float
tempoAtual, int quantidade) {
        for (int i = 0; i < quantidade; i++) {
            // Altera o angulo para onde a partícula ta sendo atirada
            setRotateEulerM(matrizDeRotacao, 0, (random.nextFloat() - 0.5f) *
angulo,
                (random.nextFloat() - 0.5f) * angulo, (random.nextFloat() -
0.5f) * angulo);
            // É feita a multiplicação da matriz para que o vetor fique
            "rodado"
            multiplyMV(vetorResultante, 0, matrizDeRotacao, 0, vetorDeDirecao,
0);
            // Ajuste de velocidade
            float ajusteVelocidade = 1f + random.nextFloat() * velocidade;

```



```

// Ajuste de angulo
    Geometry.Vector direcaoParticula = new
Geometry.Vector(vetorResultante[0] * ajusteVelocidade,
                vetorResultante[1] * ajusteVelocidade,
vetorResultante[2] * ajusteVelocidade);
    // Adicionando particula
    Randomizador a = new Randomizador();
    int rgb = a.randomNumeroInt(255,0); // Vermelho
    rgb = (rgb << 8) + a.randomNumeroInt(255,0); //Verde
    rgb = (rgb << 8) + a.randomNumeroInt(255,0); //Azul
    sistemaParticulas.addParticula(new
Geometry.Point(posicao.x,posicao.y,posicao.z), rgb, direcaoParticula,
tempoAtual);
}
    /*Obs: Uma simples estrutura de repetição que adiciona a quantidade
definida na fonte de particulas ao sistema*/
}

    public void setPosicao(Geometry.Point posicao) {
        this.posicao = posicao;
    }

    public void setExtra(Geometry.Point extra) {
        this.extra = extra;
    }
}

```

B.8 Classe Fonte (JAVA)

```

package com.particles.android.efeitos;

import com.particles.android.objects.SistemaParticulas;
import com.particles.android.util.Geometry;
import java.util.Random;
import static android.opengl.Matrix.multiplyMV;
import static android.opengl.Matrix.setRotateEulerM;

public class Fonte {
    private Geometry.Point posicao;
    private final Geometry.Vector direcao;
    private final int cor;
    private final float angulo; // Variação de Angulo
    private final float velocidade; // Variação de velocidade
    private final Random random = new Random(); // Randomizador
    private float[] matrizDeRotacao = new float[16]; // Matriz de rotação
    private float[] vetorDeDirecao = new float[4]; // Matriz de direção
    private float[] vetorResultante = new float[4]; // Matriz resultante
    // -- [ INFORMAÇÕES VEC3 EXTRA ] --
    private Geometry.Point extra;
    // -- [ MÉTODO CONSTRUTOR ] --
    public Fonte(Geometry.Point posicao, Geometry.Vector direcao, int cor,
float angulo, float velocidade)
    {
        this.posicao = posicao;
        this.direcao = direcao;
        this.cor = cor;

        // Atribuindo valores a matriz de direção, velocidade e angulo de
variação

```

```

this.angulo = angulo;
this.velocidade = velocidade;
vetorDeDirecao[0] = direcao.x;
vetorDeDirecao[1] = direcao.y;
vetorDeDirecao[2] = direcao.z;
extra= new Geometry.Point(direcao.x, direcao.y, direcao.z);
}
// -- [ ADICIONA PARTICULAS AO SISTEMA DE PARTICULAS ] --
public void addParticulas(SistemaParticulas sistemaParticulas, float
tempoAtual, int quantidade) {
    for (int i = 0; i < quantidade; i++) {
        // Altera o angulo para onde a partícula ta sendo atirada
        setRotateEulerM(matrizDeRotacao, 0, (random.nextFloat() - 0.5f)
* angulo,
                        (random.nextFloat() - 0.5f) * angulo,
        (random.nextFloat() - 0.5f) * angulo);
        // É feita a multiplicação da matriz para que o vetor fique
        "rodado"
        multiplyMV(vetorResultante, 0, matrizDeRotacao, 0,
vetorDeDirecao, 0);
        // Ajuste de velocidade
        float ajusteVelocidade = 1f + random.nextFloat() * velocidade;
        Geometry.Vector direcaoParticula = new Geometry.Vector(vetorResultante[0] *
ajusteVelocidade,
                        vetorResultante[1] * ajusteVelocidade, vetorResultante[2] * ajuste-
Velocidade);
        // Adicionando partícula
        sistemaParticulas.addParticula(posicao, cor, direcaoParticula, tempoAtual);
    }
    /*Obs: Uma simples estrutura de repetição que adiciona a quantidade definida
na fonte de partículas ao sistema*/
}

public void setPosicao(Geometry.Point posicao) {
    this.posicao = posicao;
}

public void setExtra(Geometry.Point extra) {
    this.extra = extra;
    vetorDeDirecao[1] = extra.y;
}
}

```

B.9 Classe Neve (JAVA)

```

package com.particles.android.efeitos;
import com.particles.android.objects.SistemaParticulas;
import com.particles.android.util.Geometry;
import com.particles.android.util.Randomizador;
import java.util.Random;
import static android.opengl.Matrix.multiplyMV;
import static android.opengl.Matrix.setRotateEulerM;
public class Neve {
    private Geometry.Point posicao;
    private final Geometry.Vector direcao;
    private final int cor;
    private final float angulo; // Variação de Angulo
    private final float velocidade; // Variação de velocidade
    private final Random random = new Random(); // Randomizador
    private float[] matrizDeRotacao = new float[16]; // Matriz de rotação
    private float[] vetorDeDirecao = new float[4]; // Matriz de direção

```

```

private float[] vetorResultante = new float[4]; // Matriz resultante
// -- [ INFORMAÇÕES VEC3 EXTRA ] --
private Geometry.Point extra;
// -- [ MÉTODO CONSTRUTOR ] --
public Neve(Geometry.Point posicao, Geometry.Vector direcao, int cor,
float angulo, float velocidade) {
    this.posicao = posicao;
    this.direcao = direcao;
    this.cor = cor;
    // Atribuindo valores a matriz de direção, velocidade e angulo de
    variação
    this.angulo = angulo;
    this.velocidade = velocidade;
    vetorDeDirecao[0] = direcao.x;
    vetorDeDirecao[1] = direcao.y;
    vetorDeDirecao[2] = direcao.z;
    //Inicializando os extras
    extra = new Geometry.Point(0.0f,0.0f,3.6f);
}
// -- [ ADICIONA PARTICULAS AO SISTEMA DE PARTICULAS ] --
public void addParticulas(SistemaParticulas sistemaParticulas, float
tempoAtual, int quantidade) {
    for (int i = 0; i < quantidade; i++) {
        // Atribuindo as novas coordenadas com valores randomicos
        // Altera o angulo para onde a partícula ta sendo atirada
        setRotateEulerM(matrizDeRotacao, 0, (random.nextFloat() - 0.5f)
* angulo, (random.nextFloat() - 0.5f) * angulo, (random.nextFloat() - 0.5f)
* angulo); // É feita a multiplicação da matriz para que o vetor fique
"rodado"
        multiplyMV(vetorResultante, 0, matrizDeRotacao,
0, vetorDeDirecao, 0); // Ajuste de velocidade
        float ajusteVelocidade = velocidade;

// Ajuste de angulo
        Geometry.Vector direcaoParticula = new
Geometry.Vector(vetorResultante[0] * ajusteVelocidade,
                vetorResultante[1] * ajusteVelocidade,
vetorResultante[2] * ajusteVelocidade);
        // Alterando um eixo da direção
        Randomizador inteiro = new Randomizador();
        int num = inteiro.randomNumeroInt(1,-1);
        if (num==0)
            num=1;
        direcaoParticula.x = direcaoParticula.x*num;
        // Randomizando valores de posição X e extras partícula
        Randomizador a = new Randomizador();
        float posX= a.randomNumero(-2.3f,2.3f);
        extra.x = extra.y = extra.z = a.randomNumero(25.0f,35.5f);
        sistemaParticulas.setExtra(extra);
        // Adicionando partícula
        sistemaParticulas.addParticula(new
Geometry.Point(posX,posicao.y,posicao.z), cor, direcaoParticula,
tempoAtual);
    }
    /*Obs: Uma simples estrutura de repetição que adiciona a quantidade
    definida na fonte de partículas ao sistema*/
}

public void setPosicao(Geometry.Point posicao) {
    this.posicao = posicao;
}

```

B.10 Shader Neve (GLSL)

```
uniform mat4 u_Matrix;
uniform float u_Time;

attribute vec3 a_Position;
attribute vec3 a_Color;
attribute vec3 a_DirectionVector;
attribute float a_ParticleStartTime;
attribute vec3 p_Extra;

varying vec3 v_Color;
varying vec3 direcao;
varying vec3 spawn;
varying float v_ElapsedTime;

void main()
{
    v_Color = a_Color;
    v_ElapsedTime = u_Time - a_ParticleStartTime;
    direcao = a_DirectionVector;
    spawn = a_Position;
    vec3 currentPosition = spawn + (direcao * v_ElapsedTime);
    gl_Position = u_Matrix * vec4(currentPosition, 1.0);
    gl_PointSize = 40.0;
}
```

B.11 Particle Fragment Shader Com Tempo (GLSL)

```
precision mediump float;

varying vec3 v_Color;
varying float v_ElapsedTime;

uniform sampler2D u_TextureUnit;

void main()
{
    //Aplicando cor com variação ao tempo
    gl_FragColor = vec4(v_Color / v_ElapsedTime, 1.0);
}
```

B.12 Particle Fragment Shader sem Tempo (GLSL)

```
precision mediump float;

varying vec3 v_Color;
varying float v_ElapsedTime;

uniform sampler2D u_TextureUnit;

void main()
{
    //Aplicando cor sem variação ao Fragmento
    gl_FragColor = vec4(v_Color, 1.0);
}
```

C – LINK ENTRE JAVA E GPU

C.1 Classe ShaderProgram (JAVA)

```
package com.particles.android.programs;

import android.content.Context;
import com.particles.android.util.ShaderHelper;
import com.particles.android.util.TextResourceReader;
import static android.opengl.GLES20.glUseProgram;

public class ShaderProgram {
    // Uniform constants
    protected static final String U_MATRIZ = "u_Matrix";
    protected static final String U_UNIDADE_TEXTURE = "u_TextureUnit";
    protected static final String U_COR = "u_Color";
    // Attributs constants
    protected static final String A_POSICAO = "a_Position";
    protected static final String A_COR = "a_Color";
    protected static final String A_TEXTURE_COORDINATES =
"a_TextureCoordinates";
    protected static final String U_TEMPO = "u_Time";
    protected static final String A_VETOR_DE_DIRECAO = "a_DirectionVector";
    protected static final String A_PARTICULA_TEMPOINICIAL =
"a_ParticleStartTime";
    protected static final String PARTICULA_EXTRA_VEC3 = "p_Extra";
    // Programa shader
    protected final int programa;

    protected ShaderProgram(Context context, int vertexShaderResourceId,
int fragmentShaderResourceId) {
        // Compila os shaders e linka ao programa
        programa =
ShaderHelper.buildProgram(TextResourceReader.readTextFileFromResource(context, vertexShaderResourceId),
        TextResourceReader.readTextFileFromResource(context,
fragmentShaderResourceId));
    }
    public void useProgram() {
        // Seta o shader atual ao programa
        glUseProgram(programa);
    }
}
```

C.2 Classe ParticleShaderProgram (JAVA)

```
package com.particles.android.programs;

import android.content.Context;
import static android.opengl.GLES20.GL_TEXTURE0;
import static android.opengl.GLES20.GL_TEXTURE_2D;
import static android.opengl.GLES20.glActiveTexture;
import static android.opengl.GLES20.glBindTexture;
import static android.opengl.GLES20.glGetAttribLocation;
import static android.opengl.GLES20.glGetUniformLocation;
import static android.opengl.GLES20.glUniform1f;
import static android.opengl.GLES20.glUniform1i;
import static android.opengl.GLES20.glUniformMatrix4fv;
```

```

public class ParticleShaderProgram extends ShaderProgram{
    // Localização das variaveis Uniform no shader
    private final int uLocalMatriz;
    private final int uLocalTempo;
    // Localização das variaveis Attribute no shader
    private final int aLocalPosicao;
    private final int aLocalCor;
    private final int aLocalVetorDirecao;
    private final int aLocalTempoInicialParticula;
    private final int pExtra;
    private final int uTextureUnitLocation;

    public ParticleShaderProgram(Context context, int vertex_shader, int
fragment_shader) {
        super(context, vertex_shader, fragment_shader);
        uLocalMatriz = glGetUniformLocation(programa, U_MATRIZ);
        uLocalTempo = glGetUniformLocation(programa, U_TEMPO);
        aLocalPosicao = glGetAttribLocation(programa, A_POSICAO);
        aLocalCor = glGetAttribLocation(programa, A_COR);
        aLocalVetorDirecao = glGetAttribLocation(programa,
A_VETOR_DE_DIRECAO);
        aLocalTempoInicialParticula = glGetAttribLocation(programa,
A_PARTICULA_TEMPOINICIAL);
        pExtra = glGetAttribLocation(programa, PARTICULA_EXTRA_VEC3);
        // Inicializando a textura
        uTextureUnitLocation = glGetUniformLocation(programa,
U_UNIDADE_TEXTURA);
    }
    public void setUniforms(float[] matrix, float elapsedTime, int textureId) {
        glUniformMatrix4fv(uLocalMatriz, 1, false, matrix, 0);
        glUniform1f(uLocalTempo, elapsedTime);
        // -- [ APLICAÇÃO DE TEXTURA ] --
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, textureId);
        glUniform1i(uTextureUnitLocation, 0);
    }
    public int getLocalPosicao() {
        return aLocalPosicao;
    }

    public int getLocalCor() {
        return aLocalCor;
    }

    public int getLocalVetorDeDirecao() {
        return aLocalVetorDirecao;
    }

    public int getLocalTempoInicialParticula() {
        return aLocalTempoInicialParticula;
    }

    public int getLocalInformacoesExtras() { return pExtra; }

    public int getProgramID() { return programa; }
}

```

D – SISTEMA DE PARTÍCULAS

D.1 Classe Particulas (JAVA)

```
package com.particles.android.objects;

import com.particles.android.efeitos.Chuva;
import com.particles.android.efeitos.Faisca;
import com.particles.android.efeitos.Fogo;
import com.particles.android.efeitos.Fogos;
import com.particles.android.efeitos.Fonte;
import com.particles.android.efeitos.Neve;
import com.particles.android.util.Geometry;

public class Particula {
    private String tipoEfeito;
    // -- [ EFEITOS ] --
    private Chuva efeitoChuva;
    private Faisca efeitoFaisca;
    private Fogo efeitoFogo;
    private Fogos efeitoFogos;
    private Fonte efeitoFonte;
    private Neve efeitoNeve;
    // -- [ MÉTODO CONSTRUTOR ] --
    public Particula(String tipoEfeito, Geometry.Point posicao,
        Geometry.Vector direcao, int cor, float angulo, float velocidade) {
        this.tipoEfeito = new String(tipoEfeito);
        switch (this.tipoEfeito) {
            case "Chuva":
                efeitoChuva = new
                Chuva(posicao, direcao, cor, angulo, velocidade);
                break;
            case "Faisca":
                efeitoFaisca = new
                Faisca(posicao, direcao, cor, angulo, velocidade);
                break;
            case "Fogo":
                efeitoFogo = new
                Fogo(posicao, direcao, cor, angulo, velocidade);
                break;
            case "Fogos":
                efeitoFogos = new
                Fogos(posicao, direcao, cor, angulo, velocidade);
                break;
            case "Fonte":
                efeitoFonte = new
                Fonte(posicao, direcao, cor, angulo, velocidade);
                break;
            case "Neve":
                efeitoNeve = new
                Neve(posicao, direcao, cor, angulo, velocidade);
                break;
        }
    }
}
```

```

// -- [ ADICIONA PARTICULAS AO SISTEMA DE PARTICULAS ] --
    public void addParticulas(SistemaParticulas sistemaParticulas, float
tempoAtual, int quantidade) {
        switch (this.tipoEfeito) {
            case "Chuva":
efeitoChuva.addParticulas(sistemaParticulas,tempoAtual,quantidade);
                break;
            case "Faisca":
efeitoFaisca.addParticulas(sistemaParticulas,tempoAtual,quantidade);
                break;
            case "Fogo":
efeitoFogo.addParticulas(sistemaParticulas,tempoAtual,quantidade);
                break;
            case "Fogos":
efeitoFogos.addParticulas(sistemaParticulas,tempoAtual,quantidade);
                break;
            case "Fonte":
efeitoFonte.addParticulas(sistemaParticulas,tempoAtual,quantidade);
                break;
            case "Neve":
efeitoNeve.addParticulas(sistemaParticulas,tempoAtual,quantidade);
                break;
        }
    }

    public void setPosicao(Geometry.Point posicao) {
        switch (this.tipoEfeito) {
            case "Faisca":
efeitoFaisca.setPosicao(posicao);
                break;
            case "Fogo":
efeitoFogo.setPosicao(posicao);
                break;
            case "Fogos":
efeitoFogos.setPosicao(posicao);
                break;
            case "Fonte":
efeitoFonte.setPosicao(posicao);
                break;
        }
    }

    public void setExtra(Geometry.Point extra) {
        switch (this.tipoEfeito) {
            case "Chuva":
efeitoChuva.setExtra(extra);
                break;
            case "Faisca":
efeitoFaisca.setExtra(extra);
                break;
            case "Fogo":
efeitoFogo.setExtra(extra);
                break;
            case "Fogos":
efeitoFogos.setExtra(extra);
                break;
            case "Fonte":
efeitoFonte.setExtra(extra);
                break;
        }
    }
}

```


D.2 Classe SistemaParticulas (JAVA)

```

package com.particles.android.objects;

import android.graphics.Color;
import com.particles.android.data.VertexArray;
import com.particles.android.programs.ParticleShaderProgram;
import com.particles.android.util.Geometry;
import static android.opengl.GLES20.GL_POINTS;
import static android.opengl.GLES20.glDrawArrays;
import static com.particles.android.Constants.BYTES_PER_FLOAT;

public class SistemaParticulas {
    // Variáveis para contagem de posição de memória dos componentes
    private static final int COMPONENTE_POSICAO = 3;
    private static final int COMPONENTE_COR = 3;
    private static final int COMPONENTE_VETOR = 3;
    private static final int COMPONENTE_TEMPO_INICIAL_PARTICULA = 1;
    private static final int PARTICULA_EXTRA_VEC3 = 3;
    private static final int CONTAGEM_TOTAL_COMPONENTES =
        COMPONENTE_POSICAO
        + COMPONENTE_COR
        + COMPONENTE_VETOR
        + COMPONENTE_TEMPO_INICIAL_PARTICULA
        + PARTICULA_EXTRA_VEC3;

    private static final int BYTESTOTAIS_STRIDE =
        CONTAGEM_TOTAL_COMPONENTES * BYTES_PER_FLOAT;
    private final float[] particulas;
    private final VertexArray vertexArray;
    private final int maximoParticulas;
    private int particulaAtual;
    private int proximaParticula;
    // -- [ INFOS EXTRAS ] --
    private Geometry.Point extra;

    public SistemaParticulas(int maximoDeParticulas) {
        /* Calcula o espaço de memória de uma partícula em um vetor
        estatico
        * o calculo se dá pelo numero maximo de particulas que serão usadas
        * multiplicado pelo espaço que os atributos dessa partícula irá
        ocupar*/
        particulas = new float[maximoDeParticulas *
        CONTAGEM_TOTAL_COMPONENTES];
        /* aloca no vetor o espaço necessário armazenamento das particulas
        */
        vertexArray = new VertexArray(particulas);
        /* Atribui a contagem maxima de particulas ao atributo do Sistema
        de particulas*/
        this.maximoParticulas = maximoDeParticulas;
        /* Atribui valores zerados para o Extra*/
        this.extra = new Geometry.Point(0.0f, 0.0f, 0.0f);
    }

    public void addParticula(Geometry.Point posicao, int cor,
        Geometry.Vector direcao, float tempoInicial) {

```

```

/* As variaveis de deslocamento servem para calcular o deslocamento total
de memoria entre os
    * componentes internos da partícula adicionada */
    final int deslocamentoInicial = proximaParticula *
CONTAGEM_TOTAL_COMPONENTES;
    int deslocamentoAtual = deslocamentoInicial;
    proximaParticula++; // Guarda o ID da proxima partícula
    // Verifica se o numero de particulas já chegou ao maximo
    if (particulaAtual < maximoParticulas) {
        particulaAtual++;
    }
    if (proximaParticula == maximoParticulas) {
        // Volta a contagem pra zero, matando as particulas antigas e
reiniciando o envio de particulas
        proximaParticula = 0;
    }
    /* DADOS DAS PARTICULAS */
    // -- [ POSIÇÃO ] --
    particulas[deslocamentoAtual++] = posicao.x;
    particulas[deslocamentoAtual++] = posicao.y;
    particulas[deslocamentoAtual++] = posicao.z;
    // -- [ COR ] --
    particulas[deslocamentoAtual++] = Color.red(cor) / 255f;
    particulas[deslocamentoAtual++] = Color.green(cor) / 255f;
    particulas[deslocamentoAtual++] = Color.blue(cor) / 255f;
    // -- [ DIRECAO ] --
    particulas[deslocamentoAtual++] = direcao.x;
    particulas[deslocamentoAtual++] = direcao.y;
    particulas[deslocamentoAtual++] = direcao.z;
    // -- [ INICIO DO TEMPO DE VIDA ] --
    particulas[deslocamentoAtual++] = tempoInicial;
    // -- [ TESTE DE INFORMAÇÕES EXTRAS ] --
    particulas[deslocamentoAtual++] = extra.x;
    particulas[deslocamentoAtual++] = extra.z;
    particulas[deslocamentoAtual++] = extra.y;
    // -- [ ENVIADO PARTICULA PARA O VETOR "memoria" ] --
    vertexArray.updateBuffer(particulas, deslocamentoInicial,
CONTAGEM_TOTAL_COMPONENTES);
}
/* -- [ ESSA FUNÇÃO, LINKA O VETOR DE PARTICULA COM O PROGRAMA OPENGGL ] --
    * Esse link é feito fazendo com que cada informação seja atribuída ao seu
    lugar correto
    * evitando assim que as coordenadas se misturem, por exemplo cord de cores
    com direção.*/
public void bindData(ParticleShaderProgram programaDeParticulas)
{
    int dadoAtual = 0; // Coordenada do atributo atual (que esta sendo
manipulado)
    // --[ Coordenadas de posição ]--
    vertexArray.setVertexAttribPointer(dadoAtual,
programaDeParticulas.getLocalPosicao(),
        COMPONENTE_POSICAO, BYTESTOTAIS_STRIDE);
    dadoAtual += COMPONENTE_POSICAO;
    // --[ Coordenadas de cores ]--
    vertexArray.setVertexAttribPointer(dadoAtual,
programaDeParticulas.getLocalCor(),
        COMPONENTE_COR, BYTESTOTAIS_STRIDE);
    dadoAtual += COMPONENTE_COR;
}

```

```

// --[ Coordenadas de direção ]--
    vertexArray.setVertexAttribPointer(dadoAtual,
programaDeParticulas.getLocalVetorDeDirecao(),
        COMPONENTE_VETOR, BYTESTOTAIS_STRIDE);
    dadoAtual += COMPONENTE_VETOR;
    // --[ Atributo de tempo de vida ]--
    vertexArray.setVertexAttribPointer(dadoAtual,
programaDeParticulas.getLocalTempoInicialParticula(),
        COMPONENTE_TEMPO_INICIAL_PARTICULA, BYTESTOTAIS_STRIDE);
    dadoAtual += COMPONENTE_TEMPO_INICIAL_PARTICULA;
    // --[ Coordenadas de Extras ]--
    vertexArray.setVertexAttribPointer(dadoAtual,
programaDeParticulas.getLocalInformacoesExtras(),
        PARTICULA_EXTRA_VEC3, BYTESTOTAIS_STRIDE);
    dadoAtual += PARTICULA_EXTRA_VEC3;
}
/* --[ FUNÇÃO DESENHO NA TELA ] -- */
public void desenhar() {
    glDrawArrays(GL_POINTS, 0, particulaAtual);
    /* Obs: As particulas estão sendo apresentadas como pontos na tela
    * Dentro do OpenGL (dentro dos arquivos vertex shader), é possível
    definir o tamanho em pixels, controlar
    * influenciar no trajeto, entre outras coisas*/
}
/* --[ FUNÇÃO ATRIBUI EXTRA ] -- */
public void setExtra(Geometry.Point extra)
{
    this.extra = extra;
}
}

```

E – SUPERFÍCIE DE RENDERIZAÇÃO

E.1 Classe ParticlesRenderer (JAVA)

```
package com.particles.android;

import android.content.Context;
import android.content.SharedPreferences;
import android.graphics.Color;
import android.opengl.GLSurfaceView;
import android.util.Log;
import com.particles.android.objects.Particula;
import com.particles.android.objects.SistemaParticulas;
import com.particles.android.programs.ParticleShaderProgram;
import com.particles.android.util.Geometry;
import com.particles.android.util.MatrixHelper;
import com.particles.android.util.Randomizador;
import com.particles.android.util.TextureHelper;
import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;
import static android.opengl.GLES20.GL_BLEND;
import static android.opengl.GLES20.GL_COLOR_BUFFER_BIT;
import static android.opengl.GLES20.GL_ONE;
import static android.opengl.GLES20.glBlendFunc;
import static android.opengl.GLES20.glClear;
import static android.opengl.GLES20.glClearColor;
import static android.opengl.GLES20.glEnable;
import static android.opengl.GLES20.glViewport;
import static android.opengl.Matrix.multiplyMM;
import static android.opengl.Matrix.setIdentityM;
import static android.opengl.Matrix.translateM;

public class ParticlesRenderer implements GLSurfaceView.Renderer
{
    // Contexto ou tela atual que o Android está executando
    private final Context context;
    // Matrizes de projeção para a correta visualização da tela
    private final float[] projectionMatrix = new float[16];
    private final float[] viewMatrix = new float[16];
    private final float[] viewProjectionMatrix = new float[16];
    // Objetos que fazem referencia ao Sistema de Particulas e os códigos
    de Shader
    private ParticleShaderProgram particleProgram;
    private SistemaParticulas sistemaParticulas;
    // Fontes de partículas
    private Particula [] particula = new Particula[5];
    // Variavel de medida de tempo
    private long globalStartTime;
    // Aplicação de textura na particula para melhorar a visualização
    private int texture;
    // Identificar se houve toque na tela
    boolean toqueDisplay;
    // --- [ ARQUIVO ] ---
    private static final String ARQUIVO_CONFIGURACAO = "configParametro";
    //constant
    private SharedPreferences sharedPreferences;
    SharedPreferences.Editor arquivoEditor;
    String tipoEfeito;
    int usoTextura;
    Geometry.Vector direcaoParticulas;
```

```

// Método construtor
public ParticlesRenderer(Context context) {
    this.context = context;
    // inicializando a classe que gravará o arquivo
    sharedPreferences =
context.getSharedPreferences(ARQUIVO_CONFIGURACAO,0); //Arquivo e modo de
leitura (0 é privado)
    arquivoEditor = sharedPreferences.edit();
    toqueDisplay=false;
}

@Override
public void onSurfaceCreated(GL10 glUnused, EGLConfig config) {
    // Limpa a cor de fundo para a cor indicada 0,0,0
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    // Permite a mistura de cores
    glEnable(GL_BLEND);
    // Dita a forma como a mistura vai acontecer
    glBlendFunc(GL_ONE, GL_ONE);
    // -- [ Recupera alguns dados de arquivo ] --
    tipoEfeito = sharedPreferences.getString("efeito","Fogos");
    usoTextura = sharedPreferences.getInt("textura",1);
    // -- [ Inicializa o Sistema de particulas e o Shader ] --
    switch (this.tipoEfeito) {
        case "Chuva":
            if(usoTextura==0) { //Não usa textura
                particleProgram = new ParticleShaderProgram(context,
R.raw.efeito_chuva_particle_vs,
R.raw.particle_fragment_shader);
                texture = TextureHelper.loadTexture(context,
R.drawable.texture_pingo);
            }
            else {
                texture = TextureHelper.loadTexture(context,
R.drawable.texture_pingo);
                particleProgram = new ParticleShaderProgram(context,
R.raw.efeito_chuva_particle_vs,
R.raw.texture_particle_fs);
            }
            direcaoParticulas = new Geometry.Vector(-2.0f,-3.0f,0.0f);
            sistemaParticulas = new SistemaParticulas(3000);
            break;
        case "Faisca":
            if(usoTextura==0) { //Não usa textura
                particleProgram = new ParticleShaderProgram(context,
R.raw.efeito_faisca_vs, R.raw.particle_fs_com_tempo);
                texture = TextureHelper.loadTexture(context,
R.drawable.texture_pingo);
            }
            else {
                texture = TextureHelper.loadTexture(context,
R.drawable.texture_pingo);
                particleProgram = new ParticleShaderProgram(context,
R.raw.efeito_faisca_vs, R.raw.particle_fs_com_tempo);
            }
            direcaoParticulas = new Geometry.Vector(-2.0f,-3.0f,0.0f);
            sistemaParticulas = new SistemaParticulas(500);
            break;
    }
}

```

```

case "Fogo":
    if(usoTextura==0) { //Não usa textura
        particleProgram = new ParticleShaderProgram(context,
R.raw.efeito_fogo_particle_vs, R.raw.particle_fragment_shader);
        texture = TextureHelper.loadTexture(context,
R.drawable.texture_fumaca);
    }else {
        particleProgram = new ParticleShaderProgram(context,
R.raw.efeito_fogo_particle_vs, R.raw.texture_particle_fs);
        texture = TextureHelper.loadTexture(context,
R.drawable.texture_fumaca);
    }
    direcaoParticulas = new Geometry.Vector(1.3f,-2.0f,0.0f);
    sistemaParticulas = new SistemaParticulas(5000);
    break;
case "Fogos":
    if(usoTextura==0) { //Não usa textura
        texture = TextureHelper.loadTexture(context,
R.drawable.texture_pingo);
        particleProgram = new ParticleShaderProgram(context,
R.raw.particle_vertex_shader, R.raw.particle_fragment_shader);
    }else {
        texture = TextureHelper.loadTexture(context,
R.drawable.texture_pingo);
        particleProgram = new ParticleShaderProgram(context,
R.raw.particle_vertex_shader, R.raw.texture_particle_fs);
    }
    direcaoParticulas = new Geometry.Vector(0.0f,3.0f,0.0f);
    sistemaParticulas = new SistemaParticulas(3000);
    break;
case "Fonte":
    Log.d("PARTICLEPROGRAM","Fonte");
    if(usoTextura==0) //Não usa textura
        particleProgram = new
ParticleShaderProgram(context,R.raw.particle_vertex_shader,R.raw.particle_f
s_com_tempo);
    else
        particleProgram = new
ParticleShaderProgram(context,R.raw.particle_vertex_shader,R.raw.particle_f
s_com_tempo);
    direcaoParticulas = new Geometry.Vector(0.0f,0.5f,0.0f);
    sistemaParticulas = new SistemaParticulas(5000);
    break;
case "Neve":
    if(usoTextura==0) { //Não usa textura
        particleProgram = new ParticleShaderProgram(context,
R.raw.efeito_neve_particle_vs, R.raw.particle_fragment_shader);
        texture = TextureHelper.loadTexture(context,
R.drawable.texture_snow);
    }
    else {
        particleProgram = new ParticleShaderProgram(context,
R.raw.efeito_neve_particle_vs, R.raw.texture_particle_fs);
        texture = TextureHelper.loadTexture(context,
R.drawable.texture_snow);
    }
    direcaoParticulas = new Geometry.Vector(1.0f,-3.0f,0.0f);
    sistemaParticulas = new SistemaParticulas(5000);
    break;
}

```

```

// -- [ Inicializa a variavel de controle de tempo ] --
globalStartTime = System.nanoTime();
// -- [ Inicializa as fontes de particulas ] --
switch (this.tipoEfeito) {
    case "Chuva":
        particula [0] = new Particula(tipoEfeito,new Geometry.Point(0f,
3.8f, 0f), direcaoParticulas,
        Color.rgb(226, 243, 255), 3f, -2);
        break;
    case "Faisca":
        particula [0] = new Particula(tipoEfeito,new Geometry.Point(0f,
3.8f, 0f), direcaoParticulas,
        Color.rgb(255, 171, 0), 360, 0.5f);
        break;
    case "Fogo":
        particula [0] = new Particula(tipoEfeito,new Geometry.Point(0f,
0.4f, 0f), direcaoParticulas,
        Color.rgb(226, 88, 34), 180, 0.15f);
        particula [1] = new Particula(tipoEfeito,new Geometry.Point(0f,
0.2f, 0f), direcaoParticulas,
        Color.rgb(226, 88, 34), 180, 0.15f);
        particula [2] = new Particula(tipoEfeito,new Geometry.Point(0f, -
0.2f, 0f), direcaoParticulas,
        Color.rgb(226, 88, 34), 180, 0.15f);
        particula [3] = new Particula(tipoEfeito,new Geometry.Point(0f, -
0.6f, 0f), direcaoParticulas,
        Color.rgb(226, 88, 34), 180, 0.25f);
        break;
    case "Fogos":
        particula [0] = new Particula(tipoEfeito,new Geometry.Point(0f,
3.8f, 0f), direcaoParticulas,
        Color.rgb(255, 171, 0), 360, 0.02f);
        break;
    case "Fonte":
        particula [0] = new Particula(tipoEfeito,new Geometry.Point(0f,
0.0f, 0f), direcaoParticulas,
        Color.rgb(0,127,255), 10f, 1f);
        particula [1] = new Particula(tipoEfeito,new Geometry.Point(1.1f,
0.0f, 0f),
            new Geometry.Vector(-0.1f,0.2f,0.0f),
            Color.rgb(0,127,255), 1f, 1f);
        particula [2] = new Particula(tipoEfeito,new Geometry.Point(-1.1f,
0.0f, 0f),
            new Geometry.Vector(0.1f,0.2f,0.0f),
            Color.rgb(0,127,255), 1f, 1f);
        break;
    case "Neve":
        particula [0] = new Particula(tipoEfeito,new Geometry.Point(0f,
3.8f, 0f), direcaoParticulas,
        Color.rgb(226, 243, 255), 3f, 0.2f);
        break;
}
/* No momento que os objetos são instanciados, são informadas as coordenadas
ponto de partida,
* bem como sua direção e por fim a cor.
* Obs: A posição de partida também pode variar, isso irá depender do
tipo de simulação selecionada.*/
}

```

```

@Override
public void onSurfaceChanged(GL10 glUnused, int width, int height) {
    glViewport(0, 0, width, height);
    MatrixHelper.perspectiveM(projectionMatrix, 45, (float) width / (float)
height, 1f, 10f);
    setIdentityM(viewMatrix, 0);
    translateM(viewMatrix, 0, 0f, -1.5f, -5f);
    multiplyMM(viewProjectionMatrix, 0, projectionMatrix, 0, viewMatrix,
0);
    /* Os dados aqui são arranjados de forma a definir uma visualização
padrão*/
}

@Override
public void onDrawFrame(GL10 glUnused) {
    // Limpa a superficie de renderização, para que possa ser desenhada
novamente
    glClear(GL_COLOR_BUFFER_BIT);
    // Faz a conversão de nano segundos para segundos
    float tempoAtual = (System.nanoTime() - globalStartTime) / 1000000000f;
    /* Informa ao Objeto que fará uso do sistema de particulas, qual
sistema utiliza
    * qual o tempo atual e quantas particulas devem ser criadas*/
    switch (this.tipoEfeito) {
        case "Chuva":
            particula[0].addParticulas(sistemaParticulas, tempoAtual, 5);
            break;
        case "Faisca":
            if(toqueDisplay) {
                Randomizador randomizador = new Randomizador();
                particula[0].addParticulas(sistemaParticulas, tempoAtual,
randomizador.randomNumeroInt(20,5));
            }
            toqueDisplay=false;
            break;
        case "Fogo":
            particula[0].addParticulas(sistemaParticulas, tempoAtual, 1);
            particula[1].addParticulas(sistemaParticulas, tempoAtual, 2);
            particula[2].addParticulas(sistemaParticulas, tempoAtual, 5);
            particula[3].addParticulas(sistemaParticulas, tempoAtual, 10);
            break;
        case "Fogos":
            if(toqueDisplay) {
                Randomizador randomizador = new Randomizador();
                particula[0].addParticulas(sistemaParticulas, tempoAtual,
randomizador.randomNumeroInt(300,150));
            }
            toqueDisplay=false;
            break;
        case "Fonte":
            particula[0].addParticulas(sistemaParticulas, tempoAtual, 5);
            particula[1].addParticulas(sistemaParticulas, tempoAtual, 1);
            particula[2].addParticulas(sistemaParticulas, tempoAtual, 1);
            break;
        case "Neve":
            particula[0].addParticulas(sistemaParticulas, tempoAtual, 2);
            break;
    }
}

```



```

// vincula o Sistema de particulas com o Shader de OpenGL
particleProgram.useProgram();
// verificar o que é
particleProgram.setUniforms(viewProjectionMatrix, tempoAtual,
texture);
// Organiza a ordem dos dados para o programa OpenGL
sistemaParticulas.bindData(particleProgram);
// Da a ordem de desenhar as particulas
sistemaParticulas.desenhar();
}
/* -- [ Capturando coordenadas em arrasto de dedo na tela ] --*/
public void handleTouchDrag(float coordX, float coordY) {
    switch (this.tipoEfeito) {
        case "Chuva":
            break;
        case "Faisca":
            particula[0].setPosicao(new
Geometry.Point(coordX, coordY, 0.0f));
            toqueDisplay=true;
            break;
        case "Fogo":
            particula[0].setPosicao(new
Geometry.Point(coordX, coordY+1.0f, 0.0f));
            particula[1].setPosicao(new
Geometry.Point(coordX, coordY+0.6f, 0.0f));
            particula[2].setPosicao(new
Geometry.Point(coordX, coordY+0.4f, 0.0f));
            particula[3].setPosicao(new
Geometry.Point(coordX, coordY, 0.0f));
            break;
        case "Fogos":
            particula[0].setPosicao(new
Geometry.Point(coordX, coordY, 0.0f));
            toqueDisplay=true;
            break;
        case "Fonte":
            particula[0].setExtra(new
Geometry.Point(coordX, coordY, 0.0f));
            break;
        case "Neve":
            break;
    }
}
}

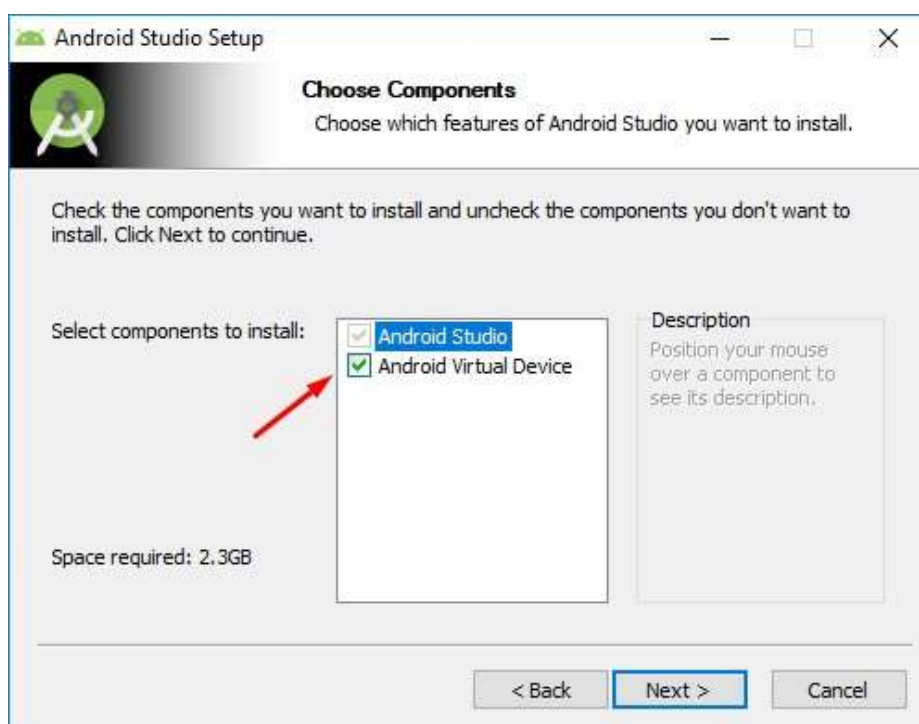
```

F – INSTAÇÃO ANDROID STUDIO E ANDROID VIRTUAL DEVICE

Para o desenvolvimento dos códigos, foi utilizada a IDE (*Integrated Development Enviroment*) Android Studio (indicada pelo site oficial do sistema Android), dada sua facilidade instalação, download de componentes e atualizações.

Para ter acesso ao Android Studio, basta fazer o download do site oficial, e após esse processo, executar e avançar a instalação, caso haja interesse em fazer uso de um simulador virtual de um dispositivo Android, selecione a opção Android Virtual Device na segunda tela, conforme Figura 6.3.

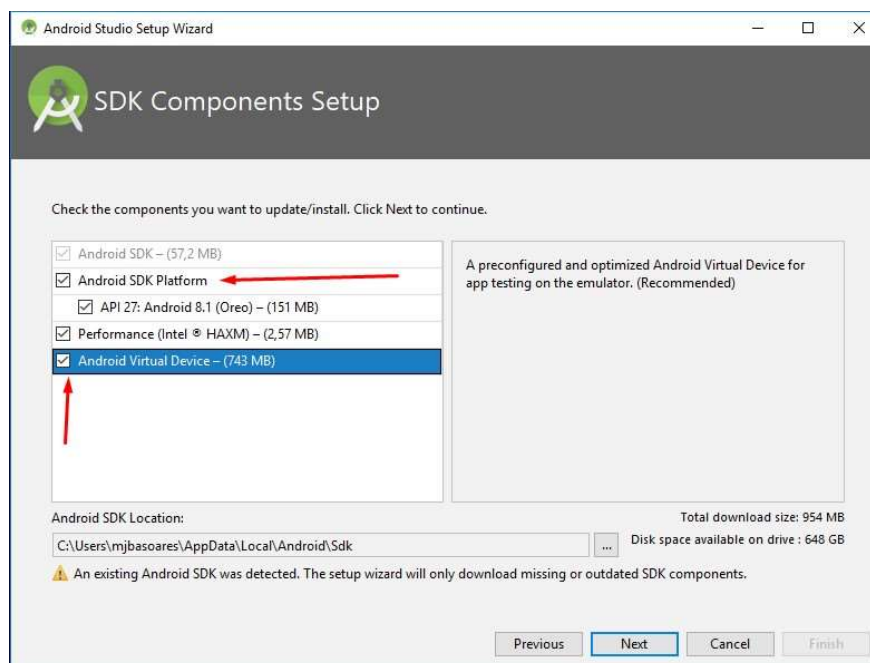
Figura 6.3 – Seleção de componentes - Android Studio



Fonte: O autor (2018).

Após a conclusão da instalação, inicia-se o Android Studio e novas telas de configuração serão apresentadas. A opção Android SDK já aparecerá marcada, marque também as opções Android SDK Platform e Android Virtual Device, conforme Figura 6.4.

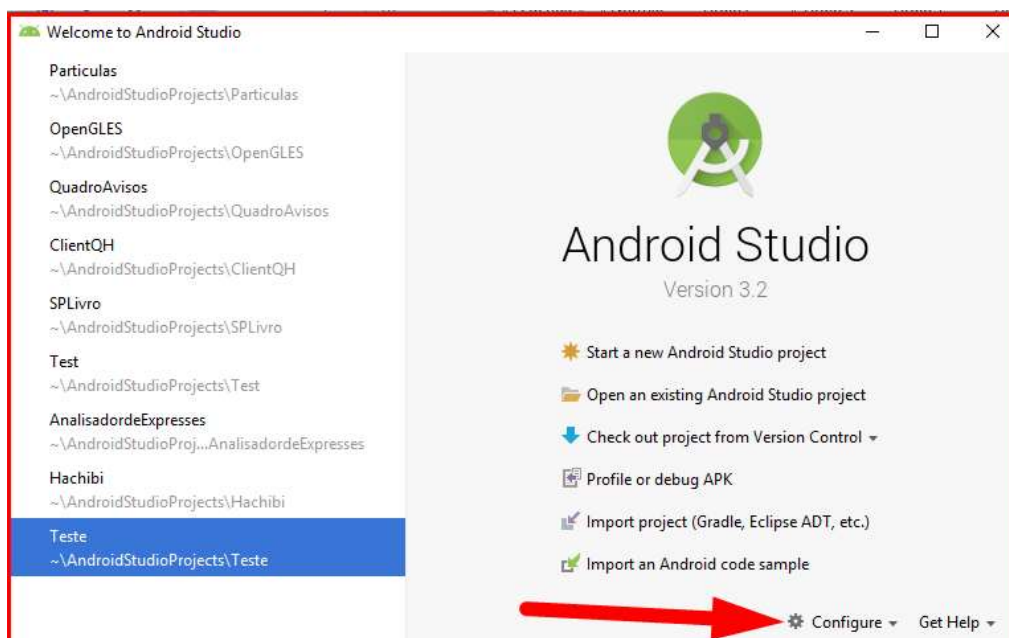
Figura 6.4 – Tela de Configuração de componentes



Fonte: O autor (2018).

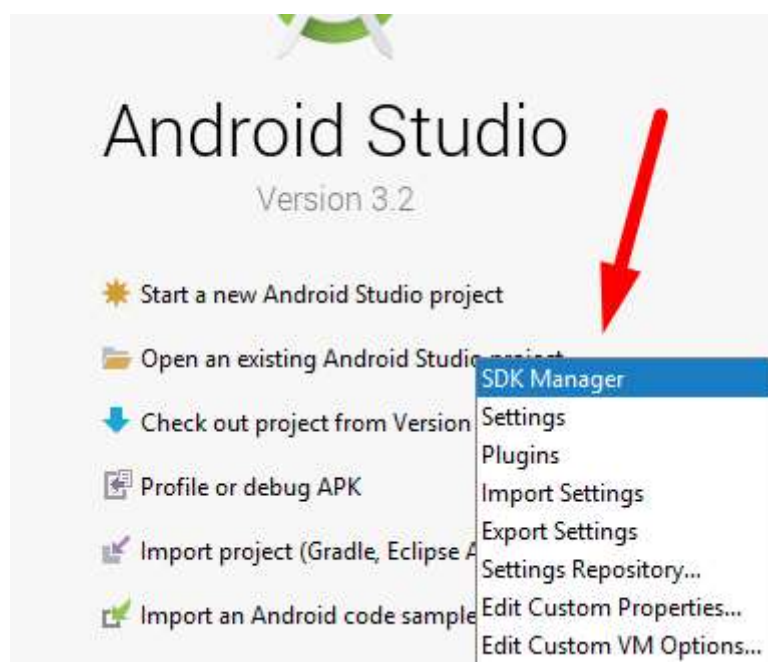
Após o finalizar o *download* dos componentes, na tela de boas-vindas da IDE, clique em *configure* em seguida em *SDK Manager*, para iniciar a configuração de dispositivos virtuais, conforme pode se observar nas Figuras 6.5 e 6.6.

Figura 6.5 – Tela de boas-vindas - Android Studio



Fonte: O autor (2018)

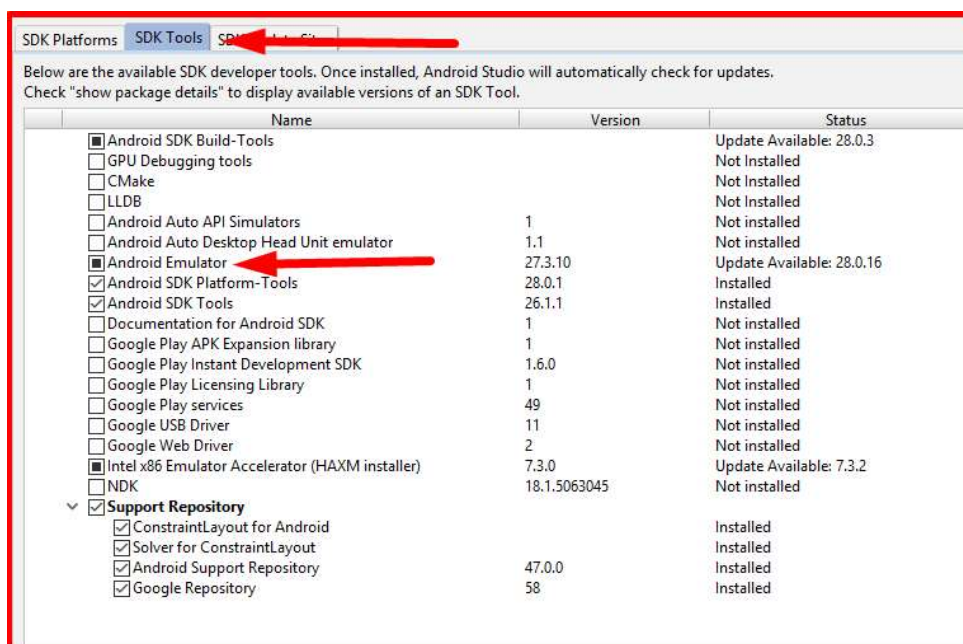
Figura 6.6 – Menu de opções da tela de boas-vindas



Fonte: O autor (2018).

Quando o *SDK Manager* abrir, selecione a aba *SDK Tools*, e verifique se a opção *Android Emulator* está ativa, assim como na Figura 6.7, caso não esteja, marque e aplique. Essa configuração é importante para habilitar o suporte à criação e uso dos dispositivos virtuais.

Figura 6.7 – Configurações do SDK Manager

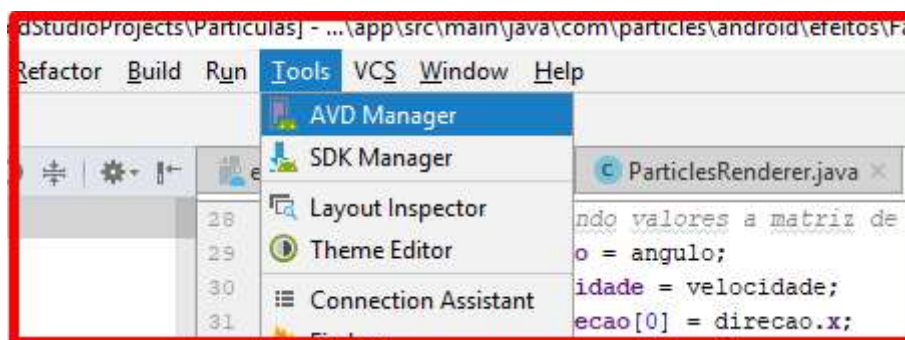


Fonte: O autor (2018).

Após essas configurações crie um projeto para ter acesso a IDE, dessa forma a opção de criar dispositivos virtuais fica disponível no ambiente de desenvolvimento.

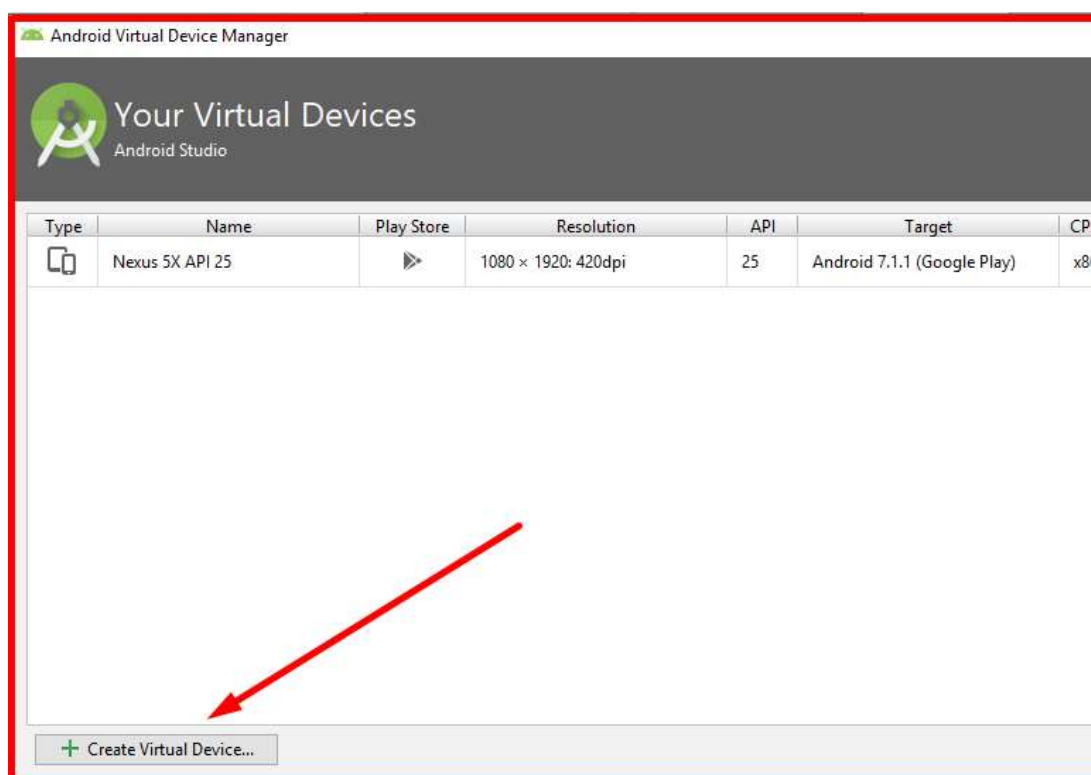
Para criar o dispositivo virtual, acesse o menu *Tools*, em seguida *AVD Manager*, assim como na Figura 6.8. Na tela seguinte, clique na opção *Create Virtual Device*, como na Figura 6.9.

Figura 6.8 – Menu Tools



Fonte: O autor (2018).

Figura 6.9 – Criação de configuração de dispositivos virtuais Android



Fonte: O autor (2018).

A parte de criação é simplificada, clique na categoria *Phone* e escolha algum dos dispositivos com configurações já definidas, logo após escolha uma versão do sistema Android e clique em *finish*.

ANEXO

A – ARQUIVOS DE SHADER

A.1 Vertex Shader (GLSL)

```
uniform mat4 u_Matrix;
uniform float u_Time;

attribute vec3 a_Position;
attribute vec3 a_Color;
attribute vec3 a_DirectionVector;
attribute vec3 p_Extra;
attribute float a_ParticleStartTime;

varying vec3 v_Color;
varying vec3 direcao;
varying float v_ElapsedTime;

void main()
{
    v_Color = a_Color;
    v_ElapsedTime = u_Time - a_ParticleStartTime;
    float gravityFactor = v_ElapsedTime * v_ElapsedTime / 8.0;
    direcao = a_DirectionVector;
    vec3 currentPosition = a_Position + (direcao * v_ElapsedTime);
    currentPosition.y -= gravityFactor;
    gl_Position = u_Matrix * vec4(currentPosition, 1.0);
    gl_PointSize = 25.0;
}
```

A.2 Fragment Shader (GLSL)

```
precision mediump float;

varying vec3 v_Color;
varying float v_ElapsedTime;

uniform sampler2D u_TextureUnit;

void main()
{
    /* Adicionando textura ao ponto ao mesmo tempo que se multiplica a
    cor com o mesmo, assim, aplicando a cor a textura*/
    gl_FragColor = vec4(v_Color, 1.0) * texture2D(u_TextureUnit,
    gl_PointCoord);
}
```

B – CLASSES UTIL

B.1 Classe Geometry (JAVA)

```
package com.particles.android.util;

public class Geometry {
    // Classe ponto
    public static class Point {
        public float x, y, z;
        public Point(float x, float y, float z) {
            this.x = x;
            this.y = y;
            this.z = z;
        }
        public Point translateY(float distance)
        {
            return new Point(x, y + distance, z);
        }
        // Translação de ponto
        public Point translate(Vector vector) {
            return new Point(x + vector.x, y + vector.y, z + vector.z);
        }
    }
    // Vetor
    public static class Vector {
        public float x, y, z;
        public Vector(float x, float y, float z) {
            this.x = x;
            this.y = y;
            this.z = z;
        }
        // comprimento
        public float length()
        {
            return (float) Math.sqrt(x * x + y * y + z * z);
        }
        // Produto cruzado
        public Vector crossProduct(Vector other) {
            return new Vector((y * other.z) - (z * other.y), (z * other.x)
- (x * other.z), (x * other.y) - (y * other.x));
        }
        // Produto escalar entre dois vetores
        public float dotProduct(Vector other)
        {
            return x * other.x + y * other.y + z * other.z;
        }
        // Scala
        public Vector scale(float f)
        {
            return new Vector(x * f, y * f, z * f);
        }
    }
}
```

B.2 Classe LoggerConfig (JAVA)

```
package com.particles.android.util;

public class LoggerConfig {
    public static final boolean ON = true;
}
```

B.3 Classe MatrixHelper (JAVA)

```
package com.particles.android.util;

public class MatrixHelper {
    public static void perspectiveM(float[] m, float yFovInDegrees, float
aspect, float n, float f) {
        final float angleInRadians = (float) (yFovInDegrees * Math.PI /
180.0);
        final float a = (float) (1.0 / Math.tan(angleInRadians / 2.0));
        m[0] = a / aspect;
        m[1] = 0f;
        m[2] = 0f;
        m[3] = 0f;

        m[4] = 0f;
        m[5] = a;
        m[6] = 0f;
        m[7] = 0f;

        m[8] = 0f;
        m[9] = 0f;
        m[10] = -((f + n) / (f - n));
        m[11] = -1f;

        m[12] = 0f;
        m[13] = 0f;
        m[14] = -((2f * f * n) / (f - n));
        m[15] = 0f;
    }
}
```

B.4 Classe ShaderHelper (JAVA)

```
package com.particles.android.util;

import android.util.Log;
import static android.opengl.GLES20.GL_COMPILE_STATUS;
import static android.opengl.GLES20.GL_FRAGMENT_SHADER;
import static android.opengl.GLES20.GL_LINK_STATUS;
import static android.opengl.GLES20.GL_VALIDATE_STATUS;
import static android.opengl.GLES20.GL_VERTEX_SHADER;
import static android.opengl.GLES20.glAttachShader;
import static android.opengl.GLES20.glCompileShader;
import static android.opengl.GLES20.glCreateProgram;
import static android.opengl.GLES20.glCreateShader;
import static android.opengl.GLES20.glDeleteProgram;
import static android.opengl.GLES20.glDeleteShader;
import static android.opengl.GLES20.glGetProgramInfoLog;
import static android.opengl.GLES20.glGetProgramiv;
```



```

import static android.opengl.GLES20.glGetShaderInfoLog;
import static android.opengl.GLES20.glGetShaderiv;
import static android.opengl.GLES20.glLinkProgram;
import static android.opengl.GLES20.glShaderSource;
import static android.opengl.GLES20.glValidateProgram;

public class ShaderHelper {
    private static final String TAG = "ShaderHelper";

    public static int compileVertexShader(String shaderCode) {
        return compileShader(GL_VERTEX_SHADER, shaderCode);
    }
    public static int compileFragmentShader(String shaderCode) {
        return compileShader(GL_FRAGMENT_SHADER, shaderCode);
    }
    private static int compileShader(int type, String shaderCode) {
        final int shaderObjectId = glCreateShader(type);
        if (shaderObjectId == 0) {
            if (LoggerConfig.ON) {
                Log.w(TAG, "Não foi possível instanciar novo shader.");
            }
            return 0;
        }
        // carrega o código opengl de shader
        glShaderSource(shaderObjectId, shaderCode);
        // compila o código carregado
        glCompileShader(shaderObjectId);

        final int[] compileStatus = new int[1];
        glGetShaderiv(shaderObjectId, GL_COMPILE_STATUS, compileStatus, 0);

        if (LoggerConfig.ON) {
            // Pega mensagens de eventos do shared e coloca no Logcat
            Log.v(TAG, "Resultado da compilação é:" + "\n" + shaderCode +
"\n:"
                + glGetShaderInfoLog(shaderObjectId));
        }

        if (compileStatus[0] == 0) {
            // Se a compilação falhar, o objeto ID atual será excluído
            (limpando a memória)
            glDeleteShader(shaderObjectId);
            if (LoggerConfig.ON) {
                Log.w(TAG, "A compilação do shader falhou.");
            }
            return 0;
        }
        return shaderObjectId;
    }
    public static int linkProgram(int vertexShaderId, int fragmentShaderId) {
        final int programObjectId = glCreateProgram();
        if (programObjectId == 0) {
            if (LoggerConfig.ON) {
                Log.w(TAG, "Não foi possível criar o novo programa");
            }
            return 0;
        }
        // Anexa ps Shaders ao programa
        glAttachShader(programObjectId, vertexShaderId);
        glAttachShader(programObjectId, fragmentShaderId);
    }

```

```

// Link os shaders ao programa
glLinkProgram(programObjectId);
final int[] linkStatus = new int[1];
glGetProgramiv(programObjectId, GL_LINK_STATUS, linkStatus, 0);
if (LoggerConfig.ON) {
    Log.v(TAG, "Resultado do link dos programas :\n" +
glGetProgramInfoLog(programObjectId));
}

if (linkStatus[0] == 0) {
    glDeleteProgram(programObjectId);
    if (LoggerConfig.ON) {
        Log.w(TAG, "O link de programas falhou.");
    }
    return 0;
}
// retorna o programa com os shaders linkados
return programObjectId;
}

public static boolean validateProgram(int programObjectId) {
    glValidateProgram(programObjectId);
    final int[] validateStatus = new int[1];
    glGetProgramiv(programObjectId, GL_VALIDATE_STATUS, validateStatus,
0);
    Log.v(TAG, "Resultados da validação do programa: " +
validateStatus[0]
        + "\nLog:" + glGetProgramInfoLog(programObjectId));
    return validateStatus[0] != 0;
}

public static int buildProgram(String vertexShaderSource, String
fragmentShaderSource) {
    int program;
    // Compilar os shaders.
    int vertexShader = compileVertexShader(vertexShaderSource);
    int fragmentShader = compileFragmentShader(fragmentShaderSource);
    // Linkar programa de shader.
    program = linkProgram(vertexShader, fragmentShader);
    if (LoggerConfig.ON) {
        validateProgram(program);
    }
    return program;
}
}

```

B.5 Classe TextResourceReader (JAVA)

```
package com.particles.android.util;

import android.content.Context;
import android.content.res.Resources;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;

public class TextResourceReader {
    public static String readTextFileFromResource(Context context, int
resourceId) {
        StringBuilder body = new StringBuilder();
        try {
            InputStream inputStream =
context.getResources().openRawResource(resourceId);
            InputStreamReader inputStreamReader = new
InputStreamReader(inputStream);
            BufferedReader bufferedReader = new
BufferedReader(inputStreamReader);
            String nextLine;
            while ((nextLine = bufferedReader.readLine()) != null) {
                body.append(nextLine);
                body.append('\n');
            }
        } catch (IOException e) {
            throw new RuntimeException("Could not open resource: " +
resourceId, e);
        } catch (Resources.NotFoundException nfe) {
            throw new RuntimeException("Resource not found: " + resourceId,
nfe);
        }
        return body.toString();
    }
}
```

B.6 Classe TextureHelper (JAVA)

```
package com.particles.android.util;

import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.util.Log;
import static android.opengl.GLES20.GL_LINEAR;
import static android.opengl.GLES20.GL_LINEAR_MIPMAP_LINEAR;
import static android.opengl.GLES20.GL_TEXTURE_2D;
import static android.opengl.GLES20.GL_TEXTURE_MAG_FILTER;
import static android.opengl.GLES20.GL_TEXTURE_MIN_FILTER;
import static android.opengl.GLES20.glBindTexture;
import static android.opengl.GLES20.glDeleteTextures;
import static android.opengl.GLES20.glGenTextures;
import static android.opengl.GLES20.glGenerateMipmap;
import static android.opengl.GLES20.glTexParameterf;
import static android.opengl.GLUUtils.texImage2D;
```

```

public class TextureHelper {
    private static final String TAG = "TextureHelper";
    public static int loadTexture(Context context, int resourceId) {
        final int[] textureObjectIds = new int[1];
        glGenTextures(1, textureObjectIds, 0);
        if (textureObjectIds[0] == 0) {
            if (LoggerConfig.ON) {
                Log.w(TAG, "Não foi possível gerar um novo objeto de
textura OpenGL.");
            }
            return 0;
        }
        final BitmapFactory.Options options = new BitmapFactory.Options();
        options.inScaled = false;
        final Bitmap bitmap =
BitmapFactory.decodeResource(context.getResources(), resourceId, options);
        if (bitmap == null) {
            if (LoggerConfig.ON) {
                Log.w(TAG, "Resource ID " + resourceId + " could not be
decoded.");
            }
            glDeleteTextures(1, textureObjectIds, 0);
            return 0;
        }
        // Configura o OpenGL para configurar a textura como 3D, indica o
ID de textura.
        glBindTexture(GL_TEXTURE_2D, textureObjectIds[0]);
        // Suavização de texturas usando mipMaps e filtragens lineares ou
trilineares
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR); // Minimização com filtragem trilinear
        glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
// maximização com filtragem linear
        // carregamento de textura
        texImage2D(GL_TEXTURE_2D, 0, bitmap, 0);
        // Limpar a memória imediatamente após o carregamento ao OpenGL
        bitmap.recycle(); // Assim evita-se lixo ou um trabalho extra ao
Dalvik cache
        glGenerateMipmap(GL_TEXTURE_2D);
        glBindTexture (GL_TEXTURE_2D, 0);
        return textureObjectIds[0];
    }
}

```

B.7 Classe Constants (JAVA)

```

package com.particles.android;

public class Constants {
    public static final int BYTES_PER_FLOAT = 4;
}

```