UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

SISTEMA DE PARTÍCULAS PARA DISPOSITIVOS MÓVEIS NA PLATAFORMA ANDROID

ANGEL VITOR LOPES

BLUMENAU
2012

2012/1-08

ANGEL VITOR LOPES

SISTEMAS DE PARTÍCULAS PARA DISPOSTIVOS MÓVEIS NA PLATAFORMA ANDROID

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

Prof. Mauro Marcelo Mattos, Dr. - Orientador

BLUMENAU
2012

2012/1-08

SISTEMA DE PARTÍCULAS PARA DISPOSITIVOS MÓVEIS NA PLATAFORMA ANDROID

Por

ANGEL VITOR LOPES

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

Presidente:	Prof. Mauro Marcelo Mattos, Dr. – Orientador, FURB
Membro:	Prof. Dalton Solano dos Reis, M. Sc. – FURB
Membro:	Prof. Miguel Alexandre Wisintainer, M. Sc. – FURB

Blumenau, 10 de julho de 2012

Dedico este trabalho à minha família, à minha namorada, e a todos que incentivaram e ajudaram diretamente na realização deste.

AGRADECIMENTOS

A Deus, por iluminar-me durante toda esta caminhada.

Aos meus pais e irmãs, pelo apoio e compreensão.

À minha namorada, Mirela, pela motivação, paciência e carinho.

À empresa, Unimed de Brusque, por acreditar no meu potencial, e meu coordenador Valmir Coelho pelo incentivo e apoio aos estudos.

Ao meu orientador, Mauro Marcelo Mattos, por ter acreditado na conclusão deste trabalho.

A imaginação é mais importante que a ciência, porque a ciência é limitada, ao passo que a imaginação abrange o mundo inteiro.

Albert Einstein

RESUMO

Este trabalho apresenta o desenvolvimento de um framework para suporte a utilização de motores de sistema de partículas. Com os sistemas de partículas é possível modelar objetos cujas formas não são bem definidas como chuva, fogo, nuvens e fluídos. Os motores definem o fluxo para o funcionamento da simulação física das partículas. A plataforma alvo são os dispositivos móveis com sistema operacional Android. Para modelagem foi utilizado OpenGL ES 1.0, disponível no Android. Para demonstrar o funcionamento da simulação física foi realizado dois estudos de casos, simulação de fogos de artifício e simulação de gotas de água.

Palavras-chave: Android. Física. Dispositivos móveis. Computação gráfica.

ABSTRACT

This work presents the development of a framework to support the use of particles system motors. With particle systems you can model objects whose shapes are not well defined as rain, fire, clouds, fluids. The motor define the flow to operation the physical simulation of the particles. The target platform are the mobile devices with Android operating system. For modeling was utilized OpenGL ES 1.0, available on Android. To demonstrate the operation of the physical simulation was carried out two case studies, simulation of fireworks and simulation of water drops.

Key-words: Android. Physics. Mobile devices. Computer graphics.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE SIGLAS

ADT – Android *Development Tools*

API – *Application Programmer Interfaces*

APK – Android *Package*

AVD – Android *Virtual Device*

CPU – *Central Processing Unit*

DDMS – *Dalvik Debug Monitor Service*

DEX – *Dalvik Executable*

FPS – Frames Por Segundo

GHz – *Giga Hertz*

GPS – *Global Position System*

GPU – *Graphics Processing Unit*

IDE – *Integrated Development Environment*

JNI – *Java Native Interface*

JVM – *Java Virtual Machine*

MHz – *Mega Hertz*

NDK – *Native Development Kit*

OHA – *Open Handset Alliance*

OpenGL – *Open Graphics Library*

OpenGL ES – *Open Graphics Library Embedded System*

PSRE – *Particle System Rendering Engine*

RF – Requisito Funcional

RAM – *Random Access Memory*

RGB – *Red Green Blue*

RGBA – *Red Green Blue Alpha*

RNF – Requisito Não-Funcional

SDK – *Software Development Kit*

SMS – *Short Message System*

UML – *Unified Modeling Language*

SQL – *Structured Query Language*

XML – *eXtensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

Weiser (1991, p. 94), já em 1991, afirmava que tecnologia móvel, sem fio, computação onipresente, com recursos de som, vídeo, textos e gráficos estaria cada vez mais presente no cotidiano das pessoas. Com o crescimento da telefonia móvel, banda larga e redes sem fio, a mobilidade da computação em múltiplas plataformas e dispositivos tornou-se cada vez mais factível (WATSON et al., 2002).

A popularização que os dispositivos móveis alcançaram tornou-se algo que não pode ser tratado com irrelevância. O expressivo aumento na utilização de dispositivos móveis dos últimos anos vem da necessidade de mobilidade e agilidade de resposta, tanto no ambiente corporativo quanto na utilização pessoal. Muito além de instrumento que realiza chamadas telefônicas, os dispositivos móveis se destacam como ferramentas importantes no dia-a-dia das pessoas e empresas. Além das funcionalidades de comunicação, estes dispositivos estão se tornando ferramentas de produtividade e agilidade de processos, bem como dispositivos avançados de entretenimento.

Como destacado por Cearley (2012), as aplicações móveis e interfaces estão entre o *Top* 10 de tendências tecnológicas e estratégicas para 2012. Considerando os sistemas operacionais em dispositivos móveis, o Android está entre os três mais vendidos no mundo e comparando o crescimento nos últimos anos, percebe-se que o ganho de mercado da plataforma é o mais expressivo em relação aos concorrentes iOS da Apple e Symbian da Nokia (PETTEY; GOASDUFF, 2011).

Os avanços tecnológicos dos *hardwares* que compõe esses dispositivos permitem o desenvolvimento de aplicativos cada vez mais robustos e com desempenho aproximado ao de *hardwares* encontrados em computadores de mesa. Alguns *smartphones* e *tablets*, topo de linha, possuem um componente chamado *Graphics Processing Unit* (GPU). A GPU é um processador especializado para executar rotinas de computação gráfica, como funções de iluminação e transformação de vértices, e que trabalha paralelamente ao processador principal. Com isso o processador principal do dispositivo não precisa gastar tempo com processamento gráfico, aumentando consideravelmente a capacidade total de processamento do dispositivo (LUEBKE; HUMPHREYS, 2007, p. 126).

Esses avanços tecnológicos em *hardware* não podem vir sozinhos. Ao mesmo tempo em que o *hardware* evolui, se faz necessário a construção de *softwares* que os utilizem de forma eficiente, aproveitando todos os recursos que são oferecidos. Alguns exemplos são as

aplicações de simulação, jogos eletrônicos, animações gráficas e ferramentas de modelagem gráfica que dependem muito de ter um hardware eficiente, mas também é preciso que o software seja desenvolvido de modo adequado. Particularmente na área de jogos, o melhor resultado é aquele que transmite maior credibilidade e realismo.

Discutido neste trabalho, o sistema de partículas surgiu com a necessidade de resolver problemas de modelagem de objetos cujas formas não são bem definidas, tais como nuvens, fumaça, poeira, fogo, chuva, fluxo de fluídos, pois a modelagem desses elementos não se resolve com polígonos e superfícies curvas (STEIGLEDER, 1997, p. 15). Foi preciso encontrar outras técnicas de modelagem para fazer com que esses elementos do mundo real fossem transportados para o mundo virtual sem perder o realismo.

As partículas únicas são objetos pequenos desenhados na área gráfica que tem diversos atributos para que se possa simular seu comportamento. Essas partículas quando agrupadas, e interagindo entre si e também com o ambiente, formam o sistema de partículas (SANTOS, 2008). Na área de jogos em particular, a simulação de partículas é um assunto que demanda atenção contínua, pois são essenciais para manter o realismo. Schuytema (2008, p. 277) afirma que um jogo que não apresente realidade no universo, não é um jogo atrativo ao jogador. Um ambiente no qual se caminha com dificuldade, que precise de esforço e luta para atingir o objetivo, representa um papel enorme na experiência geral do jogo.

Diante do exposto, este trabalho descreve o desenvolvimento de um motor de partículas que simule o comportamento físico de elementos semelhantes ao mundo real. A plataforma alvo são os dispositivos móveis com sistema operacional Android. Foi realizado um estudo de caso de dois motores exemplos para validar o conceito de que é possível desenvolver essas aplicações em dispositivos móveis.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho foi o desenvolvimento de um framework de suporte a utilização de sistema de partículas em dispositivos baseados na plataforma Android.

Os objetivos específicos do trabalho são:

a) disponibilizar uma aplicação de demonstração de sistema de partículas;
b) validar a aplicação através de um estudo de caso.

1.2 ESTRUTURA DO TRABALHO

A estrutura deste trabalho está apresentada em quatro capítulos, onde o segundo capítulo contém a fundamentação teórica necessária para o entendimento deste trabalho.

O terceiro capítulo trata do desenvolvimento dos motores de partículas, iniciando com os requisitos e a especificação da aplicação. Fazem parte desta especificação os diagramas de pacotes, de classes, de estados e de sequência. Ainda no terceiro capítulo são comentados os resultados e problemas encontrados durante a implementação do sistema.

Por fim, no quarto capítulo são apresentadas as conclusões finais sobre o trabalho e sugestões para extensões.

2 FUNDAMENTAÇÃO TEÓRICA

Na seção 2.1 são apresentados os conceitos de sistema de partículas e sua dinâmica, as etapas básicas para seu desenvolvimento, aplicações para sistemas de partículas e princípios físicos. A seção 2.2 contém uma introdução a plataforma Android, incluindo os componentes do sistema operacional e definições sobre *Open Graphics Library Embedded System* (OpenGL ES). Na seção 2.3 são apresentados os princípios e aspectos relevantes do desenvolvimento para dispositivos móveis na plataforma Android. Por fim, a seção 2.4 traz os trabalhos correlatos.

2.1 SISTEMA DE PARTÍCULAS

A modelagem de objetos cujas formas não são bem definidas, tais como nuvens, fogo, fumaça, é difícil e trabalhosa se forem utilizados técnicas convencionais de modelagem, como por exemplo, modelagem através de polígonos e superfícies curvas. Steigleder (1997, p. 15) afirma que o uso de sistemas de partículas como uma técnica de modelagem proporciona uma modelagem simples e eficaz de tais objetos dispersos.

Reeves (1983, p. 92) inicialmente introduziu o sistema de partículas como uma técnica de modelagem de objetos. A modelagem através de sistemas de partículas pode ser definida como uma representação de um objeto através de um conjunto composto por inúmeras partículas que definem o volume deste objeto.

Steigleder (1997, p. 15) toma como exemplo, uma nuvem. Ela é composta por um conjunto de inúmeras gotículas de água suspensas no ar, sendo que à medida que estas gotículas se movimentam ao longo do tempo a forma da nuvem também se modifica. Estas gotículas de água podem ser imaginadas como partículas, sendo que estas partículas definem completamente a forma da nuvem em questão. Ainda estas partículas possuem um conjunto de atributos com propriedades semelhantes. Isto demonstra que sistemas de partículas são especialmente adequados para a modelagem deste tipo de fenômeno.

Existe uma enorme quantidade de fenômenos que podem ser modelados através de um sistema de partículas, sendo que eles possuem especialmente a característica de não apresentarem formas bem definidas e regulares. Dentre esses fenômenos pode-se citar

fumaça, fogo, água, nuvens, fluídos em geral, além de fenômenos mais específicos, como erupção vulcânica, chuva, fogos de artifício, etc. Pode-se afirmar, também, que sistemas de partículas são utilizados com frequência para a modelagem de uma grande variedade de fenômenos naturais (STEIGLEDER, 1997, p. 15).

2.1.1 Etapas de um sistema de partículas

De acordo com Steigleder (1997, p. 15), os procedimentos para animação de um sistema de partículas se resumem em cinco etapas, conforme mostra o Quadro 1.

❶ A cada passo na animação de um sistema de partículas, novas partículas são geradas. Uma aproximação da quantidade de novas partículas a serem geradas por passo é normalmente especificada pelo usuário;
❷ Para cada partícula gerada são associados atributos próprios. Os valores aproximados dos atributos são, na sua maioria, também especificados pelo usuário;
❸ Cada partícula que atinge alguma condição de extinção é eliminada do sistema de partículas. Dentre as condições de extinção mais comuns encontram-se: a expiração do tempo de vida, uma transparência muito grande, uma colisão contra um objeto com uma velocidade muito elevada e a localização fora de um determinado limite. Tais condições devem ser especificadas pelo usuário e normalmente são dependentes do efeito que se que simular;
❹ Os atributos das partículas restantes são modificados de acordo com as regras dinâmicas (para atributos posicionais) e de acordo com as regras específicas para o efeito ou fenômeno que se quer modelar;
❺ As partículas ativas são utilizadas na criação da imagem final de um quadro da animação.

Fonte: Steigleder (1997, p. 15).

Quadro 1 - Etapas básicas para animação de um sistema de partículas

A primeira etapa refere-se à geração de partículas. Cada instante de tempo corresponde a um quadro da animação. E a cada instante uma quantidade determinada de partículas é criada dentro do sistema de partículas, sendo que esta quantidade é controlada de forma aleatória. O número de partículas geradas a cada quadro é importante na medida em que influencia a densidade do objeto.

Basicamente, o processo de geração das partículas é especificado de duas maneiras. Na

primeira, são especificados o número médio de partículas a serem geradas a cada instante e sua variância. O número de novas partículas a serem criadas pode ser descrito pela equação mostrada no Quadro 2, onde NP é o número de novas partículas a serem geradas, μ_{Part} é a média de partículas a serem geradas, σ_{Part} é a variância de partículas a serem geradas e Δ é uma função de distribuição normalizada para o intervalo [-1,1].

$$NP = \mu_{Part} + \Delta.\sigma_{Part}$$

Fonte: Steigleder (1997, p. 16).

Quadro 2 - Equação 1 para geração de partículas

A segunda opção para geração de partículas leva em consideração a área total da tela ocupada pelo objeto modelado. Neste modelo, conforme mostrado no Quadro 3, a média e a variância de partículas a serem geradas não é absoluta, mas relativa à área do sistema de partículas. Onde NP é o número de partículas a serem geradas, μ_{sa} é a média de partículas a serem geradas por unidade de área do dispositivo de saída, σ_{sa} é a variância de partículas a serem geradas, também por unidade de área do dispositivo de saída, Δ é uma função de distribuição normalizada para o intervalo [-1,1] e SA é a área do dispositivo de saída ocupada pelo sistema de partículas.

$$NP = \left(\mu_{sa} + \Delta.\sigma_{sa}\right).SA$$

Fonte: Steigleder (1997, p. 16).

Quadro 3 - Equação 2 para geração de partículas

Os parâmetros μ_{Part} , σ_{Part} , μ_{sa} , σ_{sa} podem variar ao longo do tempo durante a animação, proporcionando a geração de mais ou menos partículas.

A segunda etapa refere-se aos atributos das partículas. Para cada nova partícula gerada no sistema de partículas, seus atributos devem ser especificados. Os atributos mais comuns são mostrados no Quadro 4.

1. Posição,
2. Velocidade (direção e módulo),
3. Tamanho,
4. Cor,
5. Transparência,
6. Forma e
7. Tempo de vida.

Fonte: Steigleder (1997, p. 17).

Quadro 4 - Atributos de partículas

De acordo com Steigleder (1997, p. 17), alguns atributos podem ser adicionados dependendo da aplicação e dos fenômenos que estão sendo simulados. Um atributo que merece particular atenção é o atributo posição, dado que um determinado sistema de partículas apresenta posição central (origem), assim como um sistema de coordenadas próprio (determinado por dois ângulos de rotação).

Um sistema de partículas também apresenta uma forma que define a região do espaço onde as partículas serão geradas. A forma de um sistema de partículas influencia na direção em que as partículas irão percorrer. Por exemplo, em sistemas de partículas com forma esférica ou pontual, as partículas se deslocam em todas as direções, e em sistemas triangulares ou retangulares, as partículas se deslocam em direções derivadas daquelas do vetor normal, com variação dependente do ângulo de ejeção (STEIGLEDER, 1997, p. 18).

Os demais atributos podem ser estipulados através de uma média e uma variância, conforme mostrado no Quadro 5. Onde `<atributo>` é o atributo que está referenciando, como por exemplo, cor, transparência, velocidade, tamanho. $\mu_{\texttt{<atributo>}}$ é a média do atributo, $\sigma_{\texttt{<atributo>}}$ é a variância do atributo e Δ é uma função de distribuição normalizada para o intervalo [-1,1].

$$< atributo >_{inicial} = \mu_{<atributo>} + \Delta.\sigma_{<atributo>}$$

Fonte: Steigleder (1997, p. 18).

Quadro 5 - Equação de início de atributos

A terceira etapa é a dinâmica das partículas. Ao longo do tempo de vida, cada partícula apresenta variações em seus atributos, não somente no atributo posição, mas também nos atributos, cor, transparência, tamanho e velocidade.

No que diz respeito ao atributo posição, o movimento de uma determinada partícula pode ser efetuado simplesmente adicionando o vetor velocidade ao vetor posição com

possíveis variações na direção, ou através de regras pré-definidas, tais como regras algorítmicas, ou baseadas em equações diferenciais ou mesmo através de movimentos aleatórios.

Obedecendo as regras físicas dos elementos e aumentando a complexidade das equações, há outros efeitos que podem ser adicionados na dinâmica de partículas, tais como efeito de gravidade, vento, ou demais forças externas, que resultam na modificação da direção e/ou no módulo do vetor velocidade das partículas.

Em continuidade as etapas de animação de partículas, Steigleder (1997, p. 20) sugere que a quinta etapa seja a extinção das partículas. Esta etapa ocorre por uma série de motivos. O motivo mais comum para a eliminação de uma partícula do sistema de partículas é a expiração do seu tempo de vida, porém outros motivos podem ser levados em consideração, como por exemplo, quando uma partícula, apresentar uma transparência muito alta, quando estiver a uma determinada distância do centro do sistema de partículas, ou quando colidir com uma determinada velocidade com um determinado objeto.

Os motivos para a extinção de uma partícula são normalmente dependentes da aplicação ou do fenômeno que se quer modelar.

Por fim, a última etapa é a geração da imagem final com o sistema de partículas. A cada instante de tempo no processo de animação do sistema de partículas, com as posições e os atributos de suas partículas atualizados, uma imagem do sistema de partículas e os demais objetos da cena é gerada (STEIGLEDER, 1997, p. 21).

2.1.2 Aplicações de Sistema de Partículas

Desde a introdução de sistemas de partículas como uma técnica de modelagem, sistemas de partículas têm sido utilizados para simular uma grande quantidade de fenômenos naturais. Também, sistemas de partículas têm sido utilizados como técnicas de controle de animações ou como técnica para visualização de fenômenos astrofísicos (STEIGLEDER, 1997, p. 21).

Steigleder (1997, p. 21) afirma que dentre as aplicações mais comuns de sistemas de partículas, encontram-se simulação de gases, fluídos, fogo, explosões, árvores e arbustos, lava vulcânica e arco-íris, etc. Ainda, sistemas de partículas têm sido utilizados na modelagem de objetos deformáveis, como na modelagem da fusão da cera de velas, simulação de movimento de nebulosas. Também, técnicas para simulação de forças externas, como o vento, vórtices e

gravidade, utilizam sistemas de partículas.

2.1.2.1 Modelagem de fogos de artifício

Uma aplicação para a qual sistemas de partículas são bastante adequados e muito utilizados é em simulação de fogos de artifícios. Teng-See Loke (LOKE, 1992, p. 34) desenvolveu um sistema para modelagem e exibição de fogos de artifício utilizando sistema de partículas.

Basicamente, o sistema desenvolvido por Loke armazena as informações que compõe o fogo de artifício em duas listas: uma para as partículas ativas e outra para as partículas extintas. Uma vez que as partículas podem ser reutilizadas, salvar as partículas que foram extintas é muito útil, economizando, deste modo, tempo no processo de alocação e realocação de memória para partículas com a mesma estrutura (STEIGLEDER, 1997, p. 26).

As partículas ativas são introduzidas em um mecanismo de manipulação de sistemas de partículas (*Particle System Rendering Engine* – PSRE). Este mecanismo consiste de vários módulos que manipulam as propriedades particulares de cada partícula, como por exemplo, cor, brilho, forma, rastro, entre outros. Cada módulo PSRE também exibe as partículas na área de vídeo, onde a exibição de cada partícula está sujeita às projeções e transformações perspectivas. De acordo com Steigleder (1997, p. 26) o PSRE consiste basicamente de oito componentes que manipulam os seguintes procedimentos:

a) controle de movimento: manipula a dinâmica das partículas. Este módulo move as partículas para a sua próxima posição e efetua os cálculos correspondentes para a aceleração ou desaceleração das partículas;
b) mudança de cor: modifica a cor e a transparência das partículas através do uso de um incremento *Red Green Blue Alpha* (RGBA) onde A representa um fator de transparência a ser aplicado sobre os termos R, G e B;
c) efeitos especiais: trata efeitos especiais;
d) contagem de estágios: controla a contagem regressiva do atributo relativo ao tempo de vida da partícula, mensurado em número de quadros;
e) recorte: remove as partículas que estão fora da área de interesse, assim como elimina as partículas que estão com pouco brilho;
f) cintilação/exibição: manipula o efeito de cintilação das partículas e também exibe as partículas que estão ativas na área de vídeo;

g) salvamento do quadro: captura cada quadro individual que foi totalmente exibido e transfere para o arquivo;

h) *spawning*: cria o rastro que a partícula deixa a medida que se move no espaço.

Considerando as características das partículas, o atributo cor é associado a cada partícula individualmente e é representado através dos componentes *Red Green Blue* (RGB). O quarto componente, o *alpha*, é introduzido de modo a controlar a transparência de cada partícula. O componente *alpha* é necessário devido ao fato que as partículas em fogos de artifício desaparecem gradualmente. Tal procedimento é modelado através do aumento da transparência das partículas à medida que o tempo passa (STEIGLEDER, 1997, p. 27).

O componente brilho das partículas modela a intensidade da partícula de um fogo de artifício, onde zero é equivalente à cor base da partícula e o valor unitário representa uma partícula extremamente brilhante. Deste modo, através de uma interpolação linear, obtêm-se as equações, conforme Quadro 6, para a modificação das cores. Onde C_r, C_g e C_b são componentes RGB da cor a ser exibida, R_0, G_0 e B_0 são componentes RGB da cor base da partícula e B é o brilho da partícula. O brilho de uma partícula começa com um valor próximo ao valor unitário e descreve gradualmente até zero (STEIGLEDER, 1997, p. 27).

$$C_r = (1.0 - R_0).B + R_0$$

$$C_g = (1.0 - G_0).B + G_0$$

$$C_b = (1.0 - B_0).B + B_0$$

Fonte: Steigleder (1997, p. 27).

Quadro 6 - Equação de modificação das cores para fogos de artifício

Normalmente as partículas luminosas dos fogos de artifício deixam um rastro. O rastro é modelado através da criação de partículas filhas idênticas à original, porém com maior transparência, de modo a modelar corretamente o desaparecimento do rastro.

Já o movimento das partículas é definido através das equações básicas mostradas no Quadro 7. Δt, Δv e Δs são respectivamente, os incrementos de tempo, velocidade e posição entre os quadros. $u(t)$ e $a(t)$ são respectivamente, a velocidade e a aceleração da partícula no instante de tempo t.

$$\Delta s = u(t).\Delta t$$

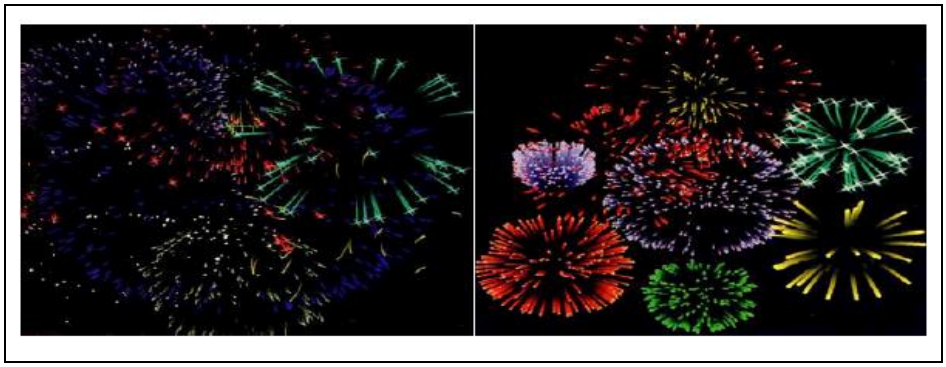
$$\Delta v = a(t).\Delta t$$

Fonte: Steigleder (1997, p. 28).

Quadro 7 - Equação do movimento de uma partícula

Fogos de artifício normalmente apresentam alguns efeitos, como o *mousing* e *starring*. Para simular esses efeitos devem ser aplicados alguns procedimentos. O *mousing* ocorre quando as partículas que estão prestes a desaparecer começam a se movimentar de forma desordenada. Uma pequena modificação aleatória no vetor velocidade é introduzida para simulação deste efeito. Já o efeito de *starring* ocorre quando uma partícula é extremamente brilhante e aparenta ser uma estrela.

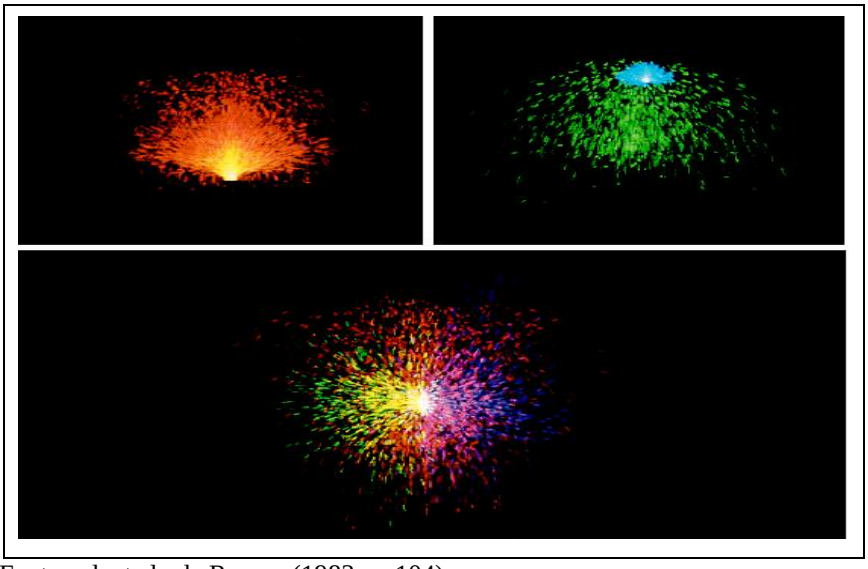
Exemplo de modelagem de fogos de artifício desenvolvidos por Loke (1992), apurado por Steigleder (1997), são mostrados na Figura 1.



Fonte: adaptado de Steigleder (1997, p. 28).

Figura 1 - Fogos de artifício modelados por Loke

Outro exemplo de modelagem de fogos de artifício foi desenvolvido por Reeves em 1983, conforme mostrado na Figura 2.



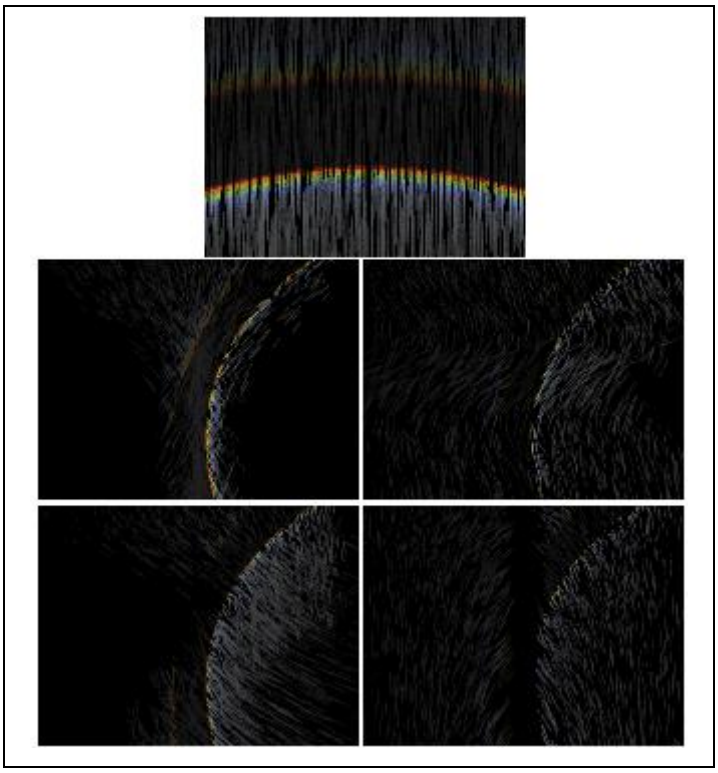
Fonte: adaptado de Reeves (1983, p. 104).

Figura 2 - Fogos de artifício modelados por Reeves

2.1.2.2 Modelagem de chuva

Um modelo importante desenvolvido para simular chuva e arco-íris foi construído por Alice Tull e Helios Tsoi (TULL; TSOI, 1995). O modelo considera o fato de que modelagem de chuva é um problema típico de sistemas de partículas, uma vez que a chuva é constituída de pequenas gotículas de água com atributos similares. Deste modo semelhante ao modelo introduzido por Reeves (1983), partículas são criadas a cada instante de tempo com atributos próprios (posição, velocidade, cor, etc.) e são extintas ao colidirem com o solo. O modelo proposto também possui a capacidade de modelar efeitos de forças externas como, por exemplo, vento (STEIGLEDER, 1997, p. 30).

Importante para trazer realismo à chuva é o efeito de um arco-íris em determinadas situações de luz. Ele é modelado através de um método fisicamente embasado. O modelo subdivide a cor dos raios de luz em uma parte de luz branca e uma parte de luz pura, sendo que uma tabela de consulta pré-calculada é utilizada para determinar a cor de determinada partícula do arco-íris. Os efeitos de luz branca e luz pura são consultados na tabela e suas cores combinadas no resultado final. Exemplos de modelagem de chuva com arco-íris são mostrados na Figura 3 (STEIGLEDER, 1997, p. 31).



Fonte: adaptado de Tull e Tsoi (1995).

Figura 3 - Modelagem de chuva com arco-íris de Alice Tull e Helios Tsoi

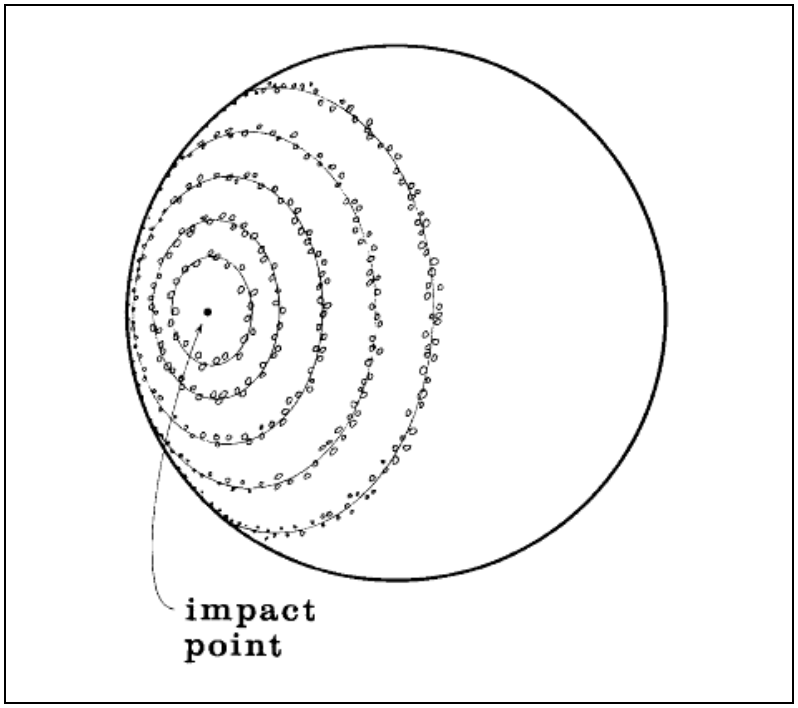
2.1.2.3 Outros exemplos de modelagem com sistema de partículas

Outros exemplos relevantes para aplicação de sistema de partículas são modelagem de fogo, fluídos, lava e erupções vulcânicas e fusão de materiais.

Para modelagem de fogo, Reeves (1983, p. 97) desenvolveu uma técnica para simular a explosão de uma bomba. A técnica introduzida foi um procedimento simples para ser utilizado no desenvolvimento da sequencia da bomba *Genesis* no filme *Star Trek II: The Wrath of Khan* (STAR TREK, 1982).

A simulação da bomba utilizou um *sistema de partículas hierárquico* de dois níveis. As partículas únicas geradas no primeiro nível se transformam em um sistema de partículas de segundo nível. Deste modo, pode-se utilizar o sistema de partículas de primeiro nível para controlar o movimento global e o sistema de partículas do segundo nível para controlar o movimento mais fino, reduzindo a complexidade do processo de modelagem (STEIGLEDER, 1997, p. 22).

O processo hierárquico de geração de partículas pode ser observado na Figura 4, onde o sistema de partículas de primeiro nível tem origem no ponto de impacto da bomba.

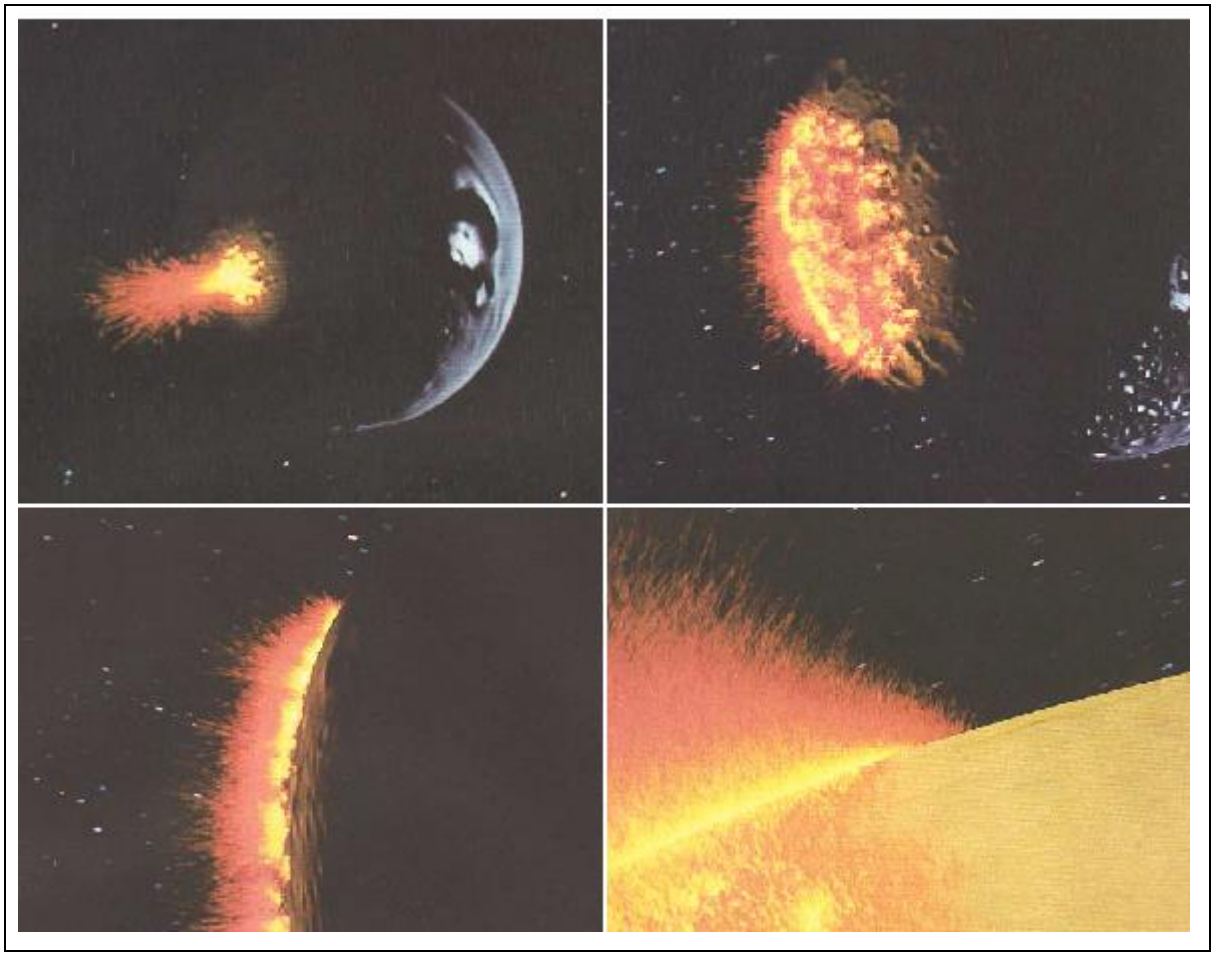


Fonte: Reeves (1983, p. 98).

Figura 4 - Distribuição do sistema de partículas na superfície do planeta

A partir do sistema de partículas de primeiro nível, o segundo nível é gerado na superfície do planeta, possibilitando assim a simulação da difusão do fogo na superfície do planeta. O exemplo do andamento deste método hierárquico de modelagem pode ser

observado na Figura 5.



Fonte: adaptado de Reeves (1983, p. 101).

Figura 5 - Efeito explosão simulado por Reeves para o filme Star Trek II

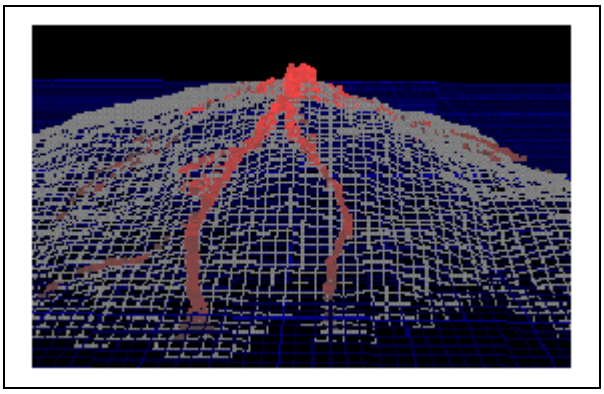
Ainda na modelagem de fogo, outra possibilidade é utilizar o conceito de *sistemas de partículas interdependentes*. Este conceito trata a combinação de outro elemento que resulta em um terceiro elemento. Por exemplo, a interação do fogo com a água formando a fumaça. Em um sistema de partículas interdependente, as partículas podem mudar os seus atributos baseados na presença, quantidade e proximidade das partículas vizinhas (STEIGLEDER, 1997, p. 22).

Na modelagem de fluídos a técnica clássica é utilizar o conceito de *sistema de partículas inversos*. Este conceito é utilizado para sistemas de controle de animação de gases, líquidos, e outros volumes de funções de densidade. A técnica possibilita o controle do movimento do fluído (STEIGLEDER, 1997, p. 24).

Outro exemplo de aplicação de sistema de partículas é na modelagem de fluxos de lava e erupções vulcânicas. Lucas Pereira e David Hsu (PEREIRA e HSU, 1995) desenvolveram um método simples e eficiente para modelagem. Basicamente as partículas são geradas no topo do vulcão, sendo que para cada partícula são atribuídos valores para temperatura e

velocidade inicial, de acordo com as características da erupção (STEIGLEDER, 1997, p. 28).

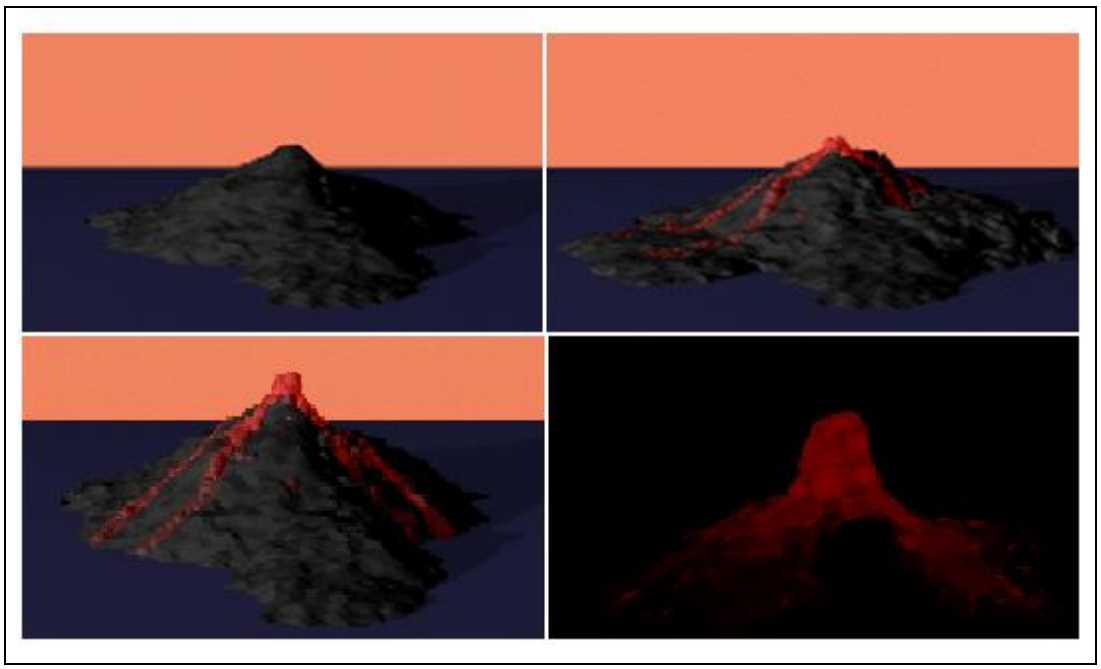
Para representação do vulcão, é utilizada uma grade bidimensional onde cada célula da grade possui uma determinada altura e temperatura, conforme mostrado na Figura 6. Cada partícula é associada a uma célula da grade de acordo com sua posição.



Fonte: Pereira e Hsu (1995).

Figura 6 - Grade bidimensional para modelagem de vulcão

Para a movimentação das partículas, é considerada uma série de fatores, porém os fatores que determinam a evolução das partículas são viscosidade, repulsão, atração, gravidade e fricção (STEIGLEDER, 1997, p. 29). O resultado da modelagem de Pereira e Hsu (1995) para fluxo de lava e erupção pode ser visualizado na Figura 7.



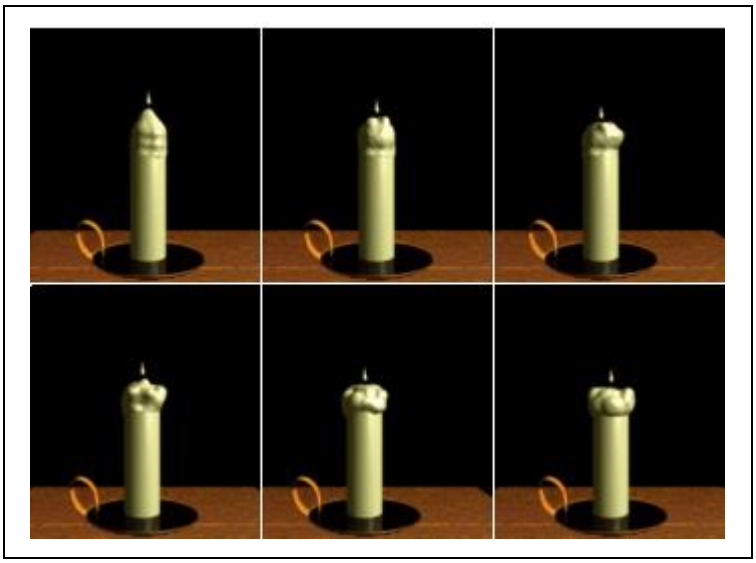
Fonte: Pereira e Hsu (1995).

Figura 7 - Modelagem de fluxo de lava e erupção vulcânica por Pereira e Hsu

Para modelagem de fusão de materiais, Gary Herman e David Redkey (HERMAN e REDKEY, 1995) desenvolveram um modelo para simular o comportamento físico da cera de vela. É considerada uma modelagem particularmente difícil, pois em um determinado instante de tempo, ela pode consistir tanto de componentes sólidos como de componentes líquidos que

interagem entre si (STEIGLEDER, 1997, p. 31).

O modelo proposto por Herman e Redkey decompõe o bloco de cera (no caso a vela) em um conjunto de esferas sólidas constituídas de partículas, sendo que estas partículas são mantidas agrupadas devido às forças de atração entre elas. Além das forças interpartículas, elas transferem calor entre si quando a diferença de calor. Deste modo, quando as partículas aquecem, as forças interpartículas diminuem. Assim que as partículas atingirem seu valor mínimo, as partículas desmembram-se das esferas, caracterizando a fusão da cera. A outros fatores externos introduzidos, como a gravidade, arraste e calor proveniente da chama (STEIGLEDER, 1997, p. 31). Um exemplo da modelagem da fusão da cera, desenvolvida por Herman e Redkey (1995), pode ser visualizado na Figura 8.



Fonte: Herman e Redkey (1995).

Figura 8 - Modelagem de fusão da cera de vela por Herman e Redkey

2.1.3 Física em sistema de partículas

Na física, o estudo da mecânica busca fornecer uma descrição precisa e consistente da dinâmica das partículas e dos sistemas de partículas, ou seja, busca um conjunto de leis físicas para descrever matematicamente os movimentos de corpos e agregados de corpos (THORNTON; MARION, 2011, p.43).

Para compreender a dinâmica das partículas, precisa-se de alguns conceitos fundamentais, como distância e tempo. A combinação destes conceitos nos permite definir a velocidade e a aceleração de uma partícula. O terceiro conceito fundamental é a massa.

Newton forneceu as leis fundamentais do movimento, as quais são (NEWTON, 1687,

p.13):

a) um corpo permanece em repouso ou em movimento uniforme, exceto sob a atuação de uma força;
b) um corpo sob a atuação de uma força se move de tal forma que a taxa temporal de variação da quantidade de movimento se iguala a força;
c) se dois corpos exercem forças entre si, essas forças são iguais em magnitude e opostas em termos de direção.

A primeira lei oferece um sentido preciso para força zero, ou seja, um corpo que permanece em repouso ou em movimento uniforme (isto é, não acelerado, retilíneo) não está sujeito a nenhuma força. Um corpo que se move dessa forma é denominado um corpo livre (ou partícula livre) (THORNTON; MARION, 2011, p. 44).

Na segunda lei, Newton definiu a quantidade de movimento (p) como sendo o produto da massa (m) com a velocidade (v) conforme mostra o Quadro 8.

$$p \equiv mv$$

Fonte: Thornton e Marion (2011, p. 44).
Quadro 8 - Equação quantidade de movimento da segunda lei de Newton

A terceira lei afirma que as acelerações de dois corpos sempre serão nas direções opostas e a relação entre as magnitudes da aceleração será constante. Essa relação constante é a relação inversa entre as massas dos corpos (THORNTON; MARION, 2011, p. 45).

Com a afirmação da terceira lei, obtêm-se a equação que resume a dinâmica de Newton para dois corpos isolados, 1 e 2, conforme mostra o Quadro 9.

$$F_1 = -F_2$$

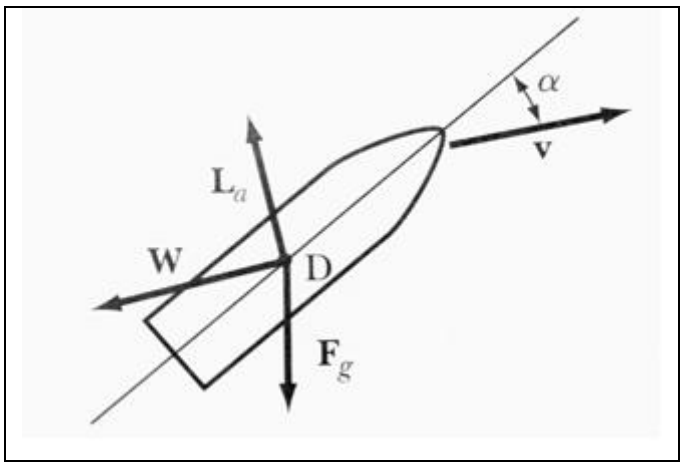
Fonte: Thornton e Marion (2011, p. 45).
Quadro 9 - Equação forças opostas da terceira lei de Newton

Para descrever o movimento de partículas, precisa-se de outros conceitos, como as forças de retardo que são as forças que alteram a direção e velocidade de uma partícula. Geralmente as forças de retardo aplicadas a uma partícula são três (THORNTON; MARION, 2011, p. 53):

a) gravitacional: geralmente a força atua para baixo;
b) resistiva do ar: também chamada de arrasto, força atua oposto a velocidade da partícula;
c) ascensional: atua perpendicular ao arrasto.

As 3 forças podem ser observadas na Figura 9 sendo aplicadas a um projétil balístico.

Onde W é o arrasto (força resistiva do ar), oposto à velocidade (v) do projétil. Já L_a é a força ascensional, perpendicular ao arrasto. E F_g é a força gravitacional, neste caso atuando para baixo. O ponto D é o centro de pressão. Caso o centro de pressão não se encontrasse no centro de massa do projétil, também existiria um torque em torno do centro de massa (THORNTON; MARION, 2011, p. 53).



Fonte: Thornton e Marion (2011, p. 53).

Figura 9 - Exemplo de forças de retardo aplicadas sobre um projétil balístico

A equação mostrada no Quadro 10 descreve a resistência do ar. Onde c_w é o coeficiente de arrasto adimensional, p é a densidade do ar, v é a velocidade e A é a área da seção transversal do objeto medida perpendicularmente à velocidade (THORNTON; MARION, 2011, p. 53).

$$W = \frac{1}{2} c_w p A v^2$$

Fonte: Thornton e Marion (2011, p. 53).

Quadro 10 - Equação para resistência do ar

O próximo conceito relevante é a gravidade. Newton formulou em 1687 a lei da gravitação, conforme abaixo.

> Cada partícula de massa atrai outra partícula do universo com uma força que varia diretamente conforme o produto das duas massas e inversamente como o quadrado da distância entre elas. (NEWTON, 1687, p.19).

A expressão matemática pode ser vista no Quadro 11. Onde r é a distância de uma partícula de massa M até outra partícula de massa m.

Trazendo o conceito para um grupo de partículas, a força resultante que atua sobre uma partícula no sistema (considerando que seja a α-ésima partícula) é, em geral, composta por duas partes. Uma parte é resultante de todas as forças cuja origem se encontra fora do sistema. Ela é chamada de *força externa* $F_\alpha^{(e)}$. A outra parte é a resultante das forças decorrentes da interação de todas as outras $n - 1$ partículas com a α-ésima partícula. Ela é chamada de

força interna f_α (THORNTON; MARION, 2011, p. 294).

$$F = -G\frac{mM}{r^2}e_r$$

Fonte: Thornton e Marion (2011, p. 161).
Quadro 11 - Equação gravitação de Newton

Algumas considerações conforme afirmados por Thornton e Marion (2011, p. 295):

a) o centro de massa de um sistema se move como se fosse uma única partícula de massa igual à massa do sistema, sob ação da força total externa, independente da natureza das forças internas;
b) a quantidade de movimento linear do sistema é a mesma de uma única partícula de massa M localizada na posição do centro de massa e se movendo da mesma maneira que o centro de massa;
c) a quantidade de movimento linear total de um sistema livre de forças externas é constante e igual à quantidade de movimento linear do centro de massa (lei da conservação da quantidade de movimento linear de um sistema).

2.2 PLATAFORMA ANDROID

O sistema operacional Android foi resultado da união de diversas empresas de tecnologia e mobilidade. Estas empresas criaram um grupo chamado *Open Handset Alliance* (OHA), atualmente conta com a participação de 84 empresas (OPEN HANDSET ALLIANCE, 2012).

O objetivo do grupo é definir uma plataforma única, aberta, moderna e flexível, com foco na usabilidade e na mais avançada tecnologia. Buscando manter uma plataforma-padrão onde todas as novas tendências do mercado estejam englobadas em uma única solução. Trazendo benefícios aos usuários finais e corporações (LECHETA, 2010, p. 21).

Cumprindo com o objetivo, o Android possui uma arquitetura flexível e aberta, ao ponto de que qualquer um pode integrar aplicações nativas com sua aplicação desenvolvida. Por exemplo, é possível fazer a integração da agenda de contatos facilmente com sua aplicação, ou até substituir a interface nativa pela sua interface desenvolvida (LECHETA, 2010, p. 22).

2.2.1 Arquitetura do sistema operacional

O sistema operacional do Android foi baseado no *kernel* 2.6 do Linux, e é responsável por gerenciar a memória, os processos, *threads*, e a segurança dos arquivos e pastas, além de redes e drives (LECHETA, 2010, p. 23).

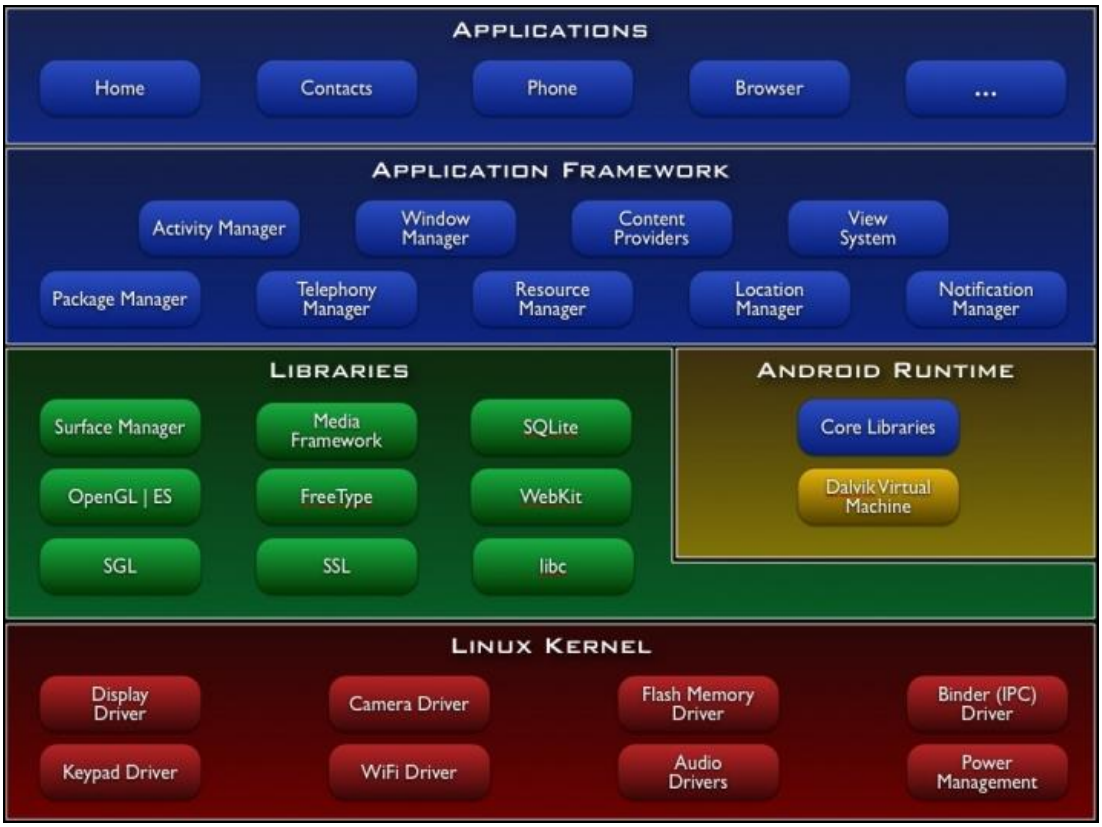
Na gerencia de processos, diversos processos podem ser executados simultaneamente, e o *kernel* do sistema operacional é responsável por realizar todo o controle de memória. Cada aplicativo do Android dispara um novo processo no sistema operacional. Alguns processos podem exibir uma tela para o usuário, e outros podem ficar executando em segundo plano por tempo indeterminado. A qualquer momento o sistema operacional pode decidir encerrar algum processo para liberar memoria e recursos, e talvez até reiniciar o mesmo processo posteriormente quando a situação estiver controlada (LECHETA, 2010, p. 24).

A segurança é baseada no na segurança do Linux. Cada aplicação é executada em um único processo, e cada processo por sua vez possuiu uma *thread* dedicada. Para cada instalação no sistema operacional Android, é criado um usuário de sistema operacional para ter acesso a sua estrutura de diretórios. Assim, nenhum outro usuário pode ter acesso a essa aplicação (LECHETA, 2010, p. 24).

O Android tem seu sistema operacional dividido em quatro camadas (*Linux Kernel*, *Libraries*, *Application Framework*, *Applications*), conforme mostrado na Figura 10. Sendo que uma camada superior não utiliza os recursos da camada inferior (GOOGLE, 2012a).

A camada mais próxima do hardware é a Linux *Kernel*. É a base em que o sistema operacional é construído, responsável pela segurança, gerenciamento de memória e processos, empilhamento de pacotes de rede e comunicação com drivers. Essa camada é a abstração entre o dispositivo físico e a camada de software (GOOGLE, 2012a).

A camada *Libraries* contém o conjunto de bibliotecas que auxiliam na execução das aplicações. Essas bibliotecas foram escritas nas linguagens C/C++ e customizadas para dispositivos móveis. Nesta camada, existe a máquina virtual Dalvik, que gerencia a execução dos aplicativos de forma a garantir um baixo consumo de memória. Uma implementação baseado no *Open Graphics Library Embedded System* (OpenGL ES) está presente nesta camada (GOOGLE, 2012a).



Fonte: Google (2012a).

Figura 10 - Arquitetura do sistema operacional Android

A camada *Application Framework* é composta por *Application Programmer Interfaces* (APIs) desenvolvidas em Java para abstrair o uso das bibliotecas da segunda camada. Estas APIs invocam as bibliotecas da segunda camada através das interfaces *Java Native Interface* (JNI) (GOOGLE, 2012a).

Na ultima camada, *Applications*, estão os aplicativos disponíveis ao usuário final. Como cliente de e-mail, calendário, programa de mensagem instantânea, mapa e contatos (GOOGLE, 2012a).

2.2.2 OpenGL ES

A *Open Graphics Library Embedded System* (OpenGL ES) é uma extensão da biblioteca *Open Graphics Library* (OpenGL) projetada para sistemas embarcados (GOOGLE, 2012b). O Android suporta OpenGL tanto através de sua biblioteca da camada *Libraries* quanto pelo *Native Development Kit* (NDK).

OpenGL ES oferece suporte para renderização de gráficos em 2 ou 3 dimensões. O OpenGL ES é mantido pelo Khronos Group (KHRONOS, 2010).

A OpenGL ES 1.0 foi projetada de acordo com a versão 1.3 da OpenGL. Em relação ao Android, a partir da versão 1.0 foi incluído o suporte ao OpenGL ES 1.0. Somente a partir do Android 2.2 (API *level* 8) foi incluído suporte à OpenGL ES 2.0. A característica mais marcante que diferencia o OpenGL ES do OpenGL é a remoção das chamadas `glBegin` e `glEnd`, favorecendo assim a utilização de *vertex arrays*. Outra característica marcante é a introdução do tipo numérico ponto fixo para coordenadas dos vértices e atributos, com o intuito de oferecer melhor suporte aos sistemas embarcados que não suportam o tipo numérico de ponto flutuante (GOOGLE, 2012b).

Para utilização da OpenGL ES no Android, há duas classes fundamentais para manipular gráficos, `GLSurfaceView` e `GLSurfaceView.Renderer`.

A classe `GLSurfaceView` é uma `View` onde os objetos são desenhados e manipulados. Para utilizá-la, é preciso criar uma instância de `GLSurfaceView`, em seguida adicionar o `Renderer` a ela. No entanto, se for necessário capturar eventos de toque na tela, será preciso estender a classe `GLSurfaceView` e implementar os *listeners* de toque (GOOGLE, 2012b).

A classe `GLSurfaceView.Renderer` é uma interface que define os métodos necessários para desenhar os gráficos em OpenGL. Deve ser feito uma implementação desta interface como classe separada e instanciá-la na classe `GLSurfaceView` através de `GLSurfaceView.setRenderer()` (GOOGLE, 2012b).

Depois de ter estabelecido uma `View` usando `GLSurfaceView` e `GLSurfaceView.Renderer`, deve-se começar a chamar as classes da OpenGL ES.

A OpenGL ES 1.0/1.1 é dividida em 2 pacotes semelhantes: `android.opengl` e `javax.microedition.khronos.opengles`. O pacote `android.opengl` contém algumas interfaces estáticas do OpenGL ES 1.0/1.1, que apresentam melhor desempenho. Já o pacote `javax.microedition.khronos.opengles` contém a implementação padrão do OpenGL ES (GOOGLE, 2012b).

A OpenGL ES 2.0 é disponibilizada através do pacote `android.opengl.GLES20` (GOOGLE, 2012b).

2.2.3 Máquina virtual Dalvik

Para desenvolver aplicativos para Android é utilizado a linguagem Java, mas não foi incluída uma *Java Virtual Machine* (JVM) no seu sistema operacional. Foi incluída uma

máquina virtual chamada Dalvik, que é uma máquina virtual desenvolvida especialmente para execução em dispositivos móveis (LECHETA, 2010, p. 24). Ela é diferente das máquinas virtuais Java existentes em outras plataformas, especialmente por ser baseada em registros e não em pilhas. A máquina virtual Dalvik utiliza-se de funcionalidades do *kernel* do sistema para tarefas básicas, como o controle de processos e gerenciamento de memória (DALVIKVM, 2008).

Essa máquina virtual também permite a execução de diversas instâncias simultâneas e independentes. Por padrão, cada aplicação é executada em seu próprio processo e instância da maquina virtual para que o código de um aplicativo seja executado isoladamente. Assim, quando o primeiro componente de uma aplicação é executado, o sistema operacional do Android inicia um novo processo Linux contendo uma instância da máquina virtual Dalvik. Esse processo continuará existindo até o sistema operacional decidir que este deve ser encerrado para liberar espaço na memória. Para alterar esse comportamento, é preciso criar explicitamente novas *threads* dentro de uma instância da máquina virtual ou definir componentes a serem executados em novas instâncias da máquina virtual (GOOGLE, 2012c).

A máquina virtual Dalvik possui também um *garbage collector* para evitar que fiquem objetos não mais utilizados na memória (GOOGLE, 2012d).

No desenvolvimento de aplicativos todos os recursos da linguagem Java estão disponíveis normalmente. Em seguida, o *bytecode* (`class`) é compilado e depois é convertido para o formato `dex` (*Dalvik Executable*), que representa uma aplicação Android compilada. Em seguida, os arquivos `dex` e os outros recursos como imagens são compactados em um único arquivo com extensão `apk` (*Android Package*) (LECHETA, 2010, p. 24). Esse arquivo pode então ser instalado e executado no sistema operacional Android.

2.2.4 Desenvolvimento na plataforma Android

O desenvolvimento de aplicações para Android é feito através do Android *Software Development Kit* (SDK) na linguagem Java. Outro ponto marcante é a forte utilização de arquivos escritos em *eXtensible Markup Language* (XML). O Android SDK possui um conjunto completo de bibliotecas para o desenvolvimento, aplicativos para a compilação e geração dos executáveis e um emulador do sistema operacional Android (LECHETA, 2010, p. 30).

Junto com o Android SDK foi incluído o Android *Virtual Device* (AVD) *Manager*,

que é utilizado para gerenciar os simuladores do Android. É possível criar um simulador de qualquer API *Level* (versão da plataforma Android), desde que se tenha atualizado e instalado a versão em questão. Para baixar novas versões da plataforma Android e atualizações para as APIs já existentes, é utilizado o SDK *Manager* que acompanha o SDK do Android.

Uma ferramenta importante para desenvolvedores que está incluída no Android SDK é o *Dalvik Debug Monitor Service* (DDMS). Com ela é possível controlar o emulador, simular o envio de uma mensagem *Short Message System* (SMS), analisar os processos, *threads*, memória, *heap*, visualizar log da aplicação, entre outros recursos (LECHETA, 2010, p. 66).

Para integração com o ambiente de desenvolvimento é preciso instalar o *plug-in* Android *Development Tools* (ADT) que facilita no desenvolvimento, nos testes e na compilação do projeto (LECHETA, 2010, p. 28).

No desenvolvimento de aplicações, o Android possuiu um recurso chamado de `Intent` que representa a intenção de uma aplicação realizar uma determinada tarefa. A classe `android.content.Intent` representa a ação que a aplicação deseja executar, e cabe ao sistema operacional interpretar essa mensagem e tomar as providências necessárias, que pode ser para abrir uma simples aplicação, fazer uma ligação telefônica, ou até abrir o browser em uma determinada página da internet (LECHETA, 2010, p. 135).

A classe `R` (`R.java`) contém as constantes para os todos os recursos de um projeto. Um recurso, por exemplo, pode ser uma imagem ou um arquivo XML que define alguma tela da aplicação. Essa classe é criada automaticamente pelo *plug-in* ADT quando criamos um projeto novo ou quando alteramos qualquer recurso do projeto (LECHETA, 2010, p. 72).

O arquivo `AndroidManifest.xml` é a base de uma aplicação Android. Este arquivo é no formato XML e deve existir em todos os projetos, com exatamente este nome. Esse arquivo fica no diretório raiz do projeto, contendo todas as configurações necessárias para executar a aplicação, como o nome do pacote utilizado, o nome das classes de cada *activity* e várias outras configurações (LECHETA, 2010, p. 74).

Há quatro tipos diferentes de componentes de aplicação: *activities*, *services*, *broadcast receivers* e *content providers*.

A classe `Activity` (`android.app.Activity`) é similar a classe `JFrame` do *Swing* e representa basicamente uma tela da aplicação. Uma tela é composta por diversos elementos visuais. Os quais no Android são representados pela classe `android.view.View`. Essas classes sempre estão juntas. A `Activity` define que existe uma tela, controla seu estado e a passagem de parâmetros de uma tela para a outra, define os métodos que serão chamados quando o usuário pressionar algum botão etc. Mas a finalidade de desenhar algo na tela é da

`View`. Uma *view* pode ser um simples componente gráfico (botão, *checkbox*, imagem) ou uma *view* complexa, que atua como um gerenciador de *layout* na qual pode conter várias *view*-filhas e tem a função de organizar as mesmas na tela (LECHETA, 2010, p. 71).

Para fazer a ligação entre *activity* e *view*, utiliza-se o método `setContentView(view)` passando como parâmetro a *view* que será exibida na tela. Esse comando sempre deve ser colocado dentro do método `onCreate(bundle)` da *activity* (LECHETA, 2010, p. 72).

A classe `Service` (`android.app.Service`) é utilizada para executar um processamento em segundo plano por tempo indeterminado, chamado popularmente de serviço. Um serviço geralmente faz um alto consumo de recursos, memória e *Central Processing Unit* (CPU). Não precisa interagir com o usuário e consequentemente não precisa de interface gráfica (LECHETA, 2010, p. 317). Outros componentes podem comunicar-se com um *service* em execução e inclusive solicitar a sua interrupção. Por padrão, um serviço será executado na mesma *thread* que disparar a sua execução. Para modificar esse comportamento é necessário definir a criação de uma nova *thread* ou um novo processo explicitamente (GOOGLE, 2012e).

O desenvolvimento de *services* é feito através da criação de subclasses de `android.app.Service` e deve obrigatoriamente implementar o método `IBinder onBind(intent)` e sugerida a implementação dos métodos `onCreate()`, `onStart()` e `onDestroy()` (LECHETA, 2010, p. 320).

De acordo com Lecheta (2010, p. 317), geralmente um serviço é iniciado a partir de um `BroadcastReceiver`, o qual precisa executar rapidamente e retornar um método `onReceiveIntent(context, intent)` o mais breve possível. Dessa forma, o `BroadcastReceiver` inicia um serviço para executar algum processamento demorado assíncrono e pode retornar.

A classe `BroadcastReceiver` (`android.content.BroadcastReceiver`) é utilizada para que aplicações possam reagir a determinados eventos gerados por uma `Intent`. Ela sempre é executada em segundo plano durante pouco tempo e sem utilizar uma interface gráfica (LECHETA, 2010, p. 290).

Para desenvolver um `BroadcastReceiver`, é preciso estender a classe `android.content.BroadcastReceiver`, e apenas o método `onReceive(contexto, intente)` precisa ser implementado. Existem duas formas de se configurar o `BroadcastReceiver`. De forma estática no arquivo `AndroidManifest.xml`, é utilizada a *tag* `<receiver>` em conjunto com a *tag* `<intente-filter>` para definir uma ação e categoria.

Ou de forma dinâmica, utilizando o método `context.registerReceiver(receiver, filtro)` dentro do código para registrar dinamicamente o `BroadcastReceiver`. O primeiro parâmetro chamado de `receiver` é uma instância de uma classe-filha da `IntentReceiver`, e o segundo é uma instância da classe `IntentFilter`, que por sua vez tem a configuração da ação e categoria (LECHETA, 2010, p. 292).

O método `sendBroadcast(intent)` deve ser implementado para que a aplicação envie uma `Intent` invocando algum `BroadcastReceiver`. Essa mensagem é enviada para o sistema operacional do Android e todas as aplicações que implementem esta `Intent` estão aptas a receber (LECHETA, 2010, p. 292).

Lecheta (2010, p. 412) afirma que cada aplicação Android possuiu sua própria área de dados, independente das demais aplicações e com acesso restrito a aplicação que criou as informações. O *Content Provider* (provedor de conteúdo) é criado para que determinadas informações sejam públicas, acessível por qualquer outra aplicação.

A criação de `ContentProviders` é feita através de uma subclasse da classe `android.content.ContentProvider` e implementa alguns métodos como `query(...)`, `insert(...)`, `update(...)` e `delete(...)` (LECHETA, 2010, p. 412).

De acordo com Lecheta (2010, p. 368) as opções para armazenamento de dados localmente são três. Através da integração com o banco de dados SQLite, possiblidade de escrever e ler arquivos e um sistema simples de persistência de chave e valor chamado preferências.

O SQLite é um leve e poderoso banco de dados escrito na linguagem C. Sua licença é de domínio público e é, portanto, livre para uso em qualquer finalidade, comercial ou privada. Ele é um motor de banco de dados *Structured Query Language* (SQL) embutido. Ele não tem um servidor de dados separado, ele lê e escreve diretamente nos arquivos de dados (SQLITE, 2012). Cada aplicação pode criar um ou mais banco de dados que ficam localizados numa pasta relativa ao nome do pacote do projeto, por exemplo `data/data/nome_pacote/databases/` (LECHETA, 2010, p. 368).

Outra forma de armazenamento é através da leitura e salvamento de arquivos. Para manipular os arquivos são utilizadas as tradicionais classes `java.io.FileInputStream` e `java.io.FileOutputStream`. Quando um arquivo é criado, ele fica armazenado no local `/data/data/nome_do_pacote/files` (LECHETA, 2010, p. 401).

Por fim, a terceira forma de armazenar dados é através das preferências. Este método utiliza a classe `android.content.SharedPreferences`. Ela funciona como uma `HashTable`

que armazena a estrutura de chave e valor para tipos primitivos, e os valores armazenados são automaticamente persistidos para a aplicação. Caso a aplicação seja encerrada e aberta posteriormente, os valores salvos nas preferências estarão lá (LECHETA, 2010, p. 407).

2.3 TRABALHOS CORRELATOS

Nesta seção são apresentados dois trabalhos de conclusão de curso e uma dissertação de mestrado com tecnologias ou funcionalidades similares àquelas que foram implementadas. O primeiro trabalho de conclusão é um estudo sobre realidade aumentada para a plataforma Android (VASSELAI, 2010). O segundo é uma simulação de corpos rígidos em 3D (MUELLER, 2010). Por fim é apresentada a dissertação de mestrado sobre integração de sistemas de partículas com detecção de colisão em ambiente *Ray Tracing* (STEIGLEDER, 2010).

2.3.1 Um estudo sobre realidade aumentada para a plataforma Android

O desafio computacional destacado pela autora está em utilizar dispositivos de hardware como câmera de vídeo, sensores de localização, acelerômetros, etc., para a entrada de dados, em seguida realizar o processamento adequado e apresentar uma saída no vídeo oferecendo um tempo de resposta curto o suficiente para que seja mantida a noção de realidade (VASSELAI, 2010, p. 15).

No trabalho foi definido três objetivos: “elucidar o potencial da plataforma Android frente ao conceito de realidade aumentada; aplicar o conceito de realidade aumentada que utiliza o registro de objetos virtuais através de coordenadas geográficas; utilizar os recursos de câmera de vídeo, *Global Position System* (GPS), acelerômetro e bússola disponibilizados na plataforma Android, para uma melhor interação do usuário com a realidade aumentada” (VASSELAI, 2010, p. 16). De acordo com Vasselai (2010, p. 96), todos os objetivos propostos foram cumpridos. Dentre os objetivos, a viabilidade de desenvolver aplicativos com realidade aumentada dentro do simulador do Android é destacada, sendo que este objetivo não fazia parte da definição inicial.

O trabalho utiliza coordenadas geográficas para registro dos objetos virtuais, chamados

pontos de interesse. A impressão de realidade aumentada é proporcionada com a utilização de uma câmera de vídeo, na qual objetos virtuais são desenhados na camada acima da imagem da câmera. A interação é feita com a movimentação do dispositivo, acionando a bússola e o acelerômetro. A Figura 11 apresenta uma demonstração da aplicação em execução.

Fonte: Vasselai (2010, p. 85).

Figura 11 - Demonstração de realidade aumentada para Android

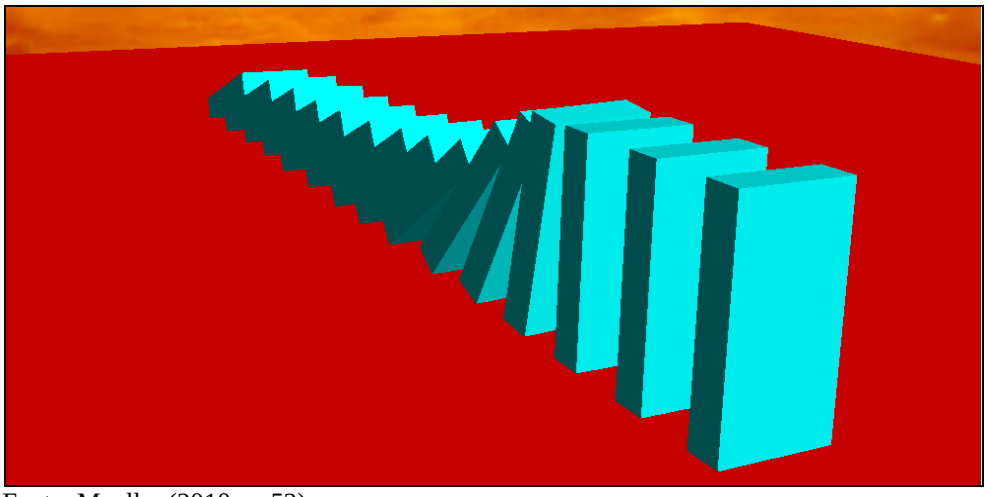
Algumas das dificuldades encontradas pela autora são: desenhar texto através da biblioteca OpenGL ES, a sobreposição de camadas de tela (camada OpenGL ES mais a camada da câmera mais a camada das ferramentas) e a dificuldade em deixar a camada da câmera translúcida. Outra dificuldade foi na utilização da API da OpenGL ES 1.0 portada para plataforma Android, com limitações, escassez de documentação e *bugs*. Por último, a dificuldade de desenvolver no simulador do dispositivo por apresentar baixo desempenho (VASSELAI, 2010, p. 91).

2.3.2 Simulação física de corpos rígidos em 3D

Identificado o problema em criar ambientes virtuais que simulem o mundo real, Mueller (2010) desenvolveu um motor de simulação física de corpos rígidos em 3D. Foi utilizado a linguagem C++ para o desenvolvimento do trabalho e que segundo o autor, proporcionou controle absoluto sobre a memória e as operações realizadas pelo programa. Foi utilizado também a biblioteca gráfica OpenGL 1.1 e o ambiente escolhido foi o Visual Studio 2010.

Mueller (2010, p. 57) complementou um trabalho anterior, desenvolvido por ele mesmo, para servir de aplicação exemplo, fazendo assim poupar esforços, pois as classes de matemática e câmera já estavam prontas.

O autor enfatiza o tratamento de colisões em seu trabalho. O motor utiliza o algoritmo `XenoCollide` para detecção de colisões e segue as leis físicas do mundo real, como considerar a massa, o centro de massa, e o momento de inércia dos corpos rígidos. O tratamento de colisão é um problema físico que determina novas velocidades, linear e angular, dos objetos após colidirem, com o objetivo de separá-los (MUELLER, 2010, p. 57). Um exemplo do motor é apresentado na Figura 12.



Fonte: Mueller (2010, p. 52).

Figura 12 - Demonstração de simulação física de corpos rígidos em 3D

2.3.3 Integração de sistemas de partículas com detecção de colisão em ambiente *Ray Tracing*

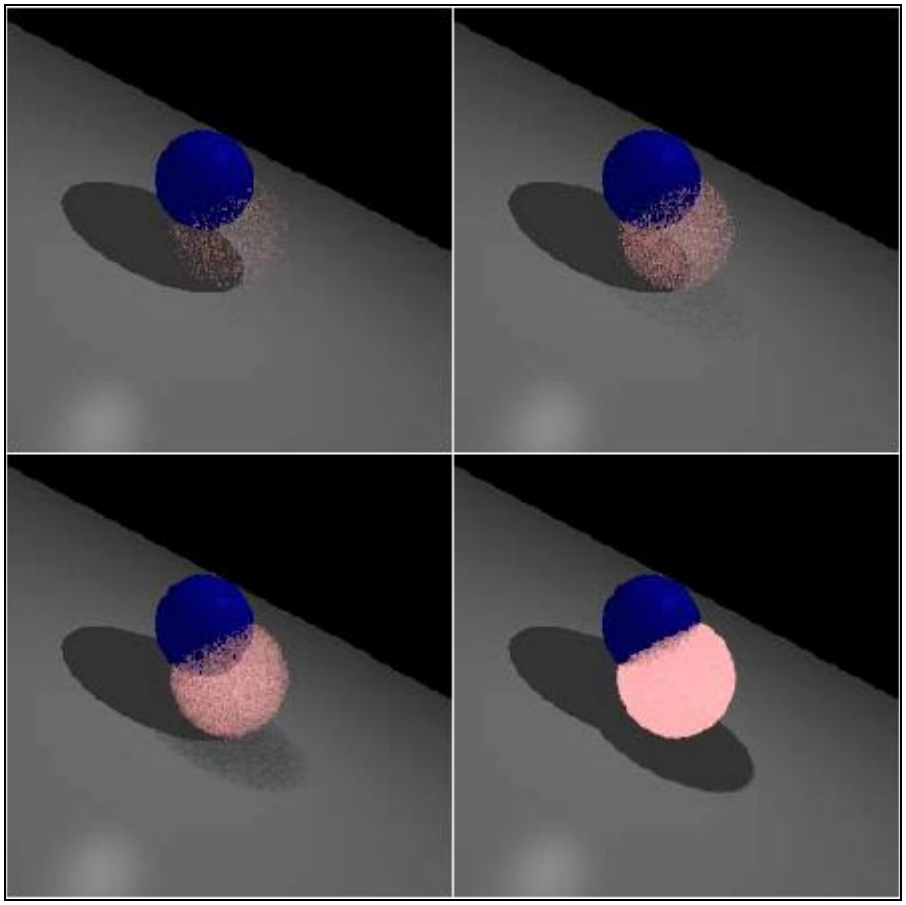
O autor destaca a meta da Computação Gráfica de encontrar um modo de criar imagens foto realísticas. Steigleder (1997, p. 9) afirma que os aspectos que possuem maior importância são a modelagem e a iluminação.

Considerando a modelagem, o autor destaca a dificuldade quando se pretende obter realismo da modelagem de objetos cujas formas não são bem definidas, como o fogo, fumaça, nuvens, etc. O autor sugere a utilização de sistemas de partículas para esta finalidade por se tratarem de entidades dinâmicas. Ainda dentre as principais vantagens da utilização deste, o autor cita a facilidade da obtenção de efeitos sobre as partículas, a necessidade de poucos dados para a modelagem global do fenômeno, o controle por processos aleatórios, o nível de detalhamento ajustável e a possibilidade de grande controle sobre suas deformações

(STEIGLEDER, 1997, p. 9).

Dentre as limitações e restrições, o autor destaca o pouco desenvolvimento de algoritmos específicos nesta área. Outras limitações estão na dificuldade de obtenção de efeitos realísticos de sombra e reflexão, o alto consumo de memória e o fato dos sistemas de partículas possuírem um processo de animação especifico para cada efeito que se quer modelar (STEIGLEDER, 1997, p. 9).

O trabalho desenvolvido pelo autor apresenta métodos para soluções destes problemas. Ele apresenta um método para tornar viável a integração de sistemas de partículas em ambiente *Ray Tracing*[1], através do uso de uma grade tridimensional. Também apresenta técnicas para eliminação de memória exigida para o armazenamento dos sistemas de partículas. A Figura 13 apresenta uma das simulações desenvolvidas por Steigleder.



Fonte: Steigleder (1997, p. 74).

Figura 13 - Demonstração de sistema de partículas com detecção de colisão

[1] *Ray Tracing* é uma técnica que tenta simular o efeito produzido pelos raios de luz emitidos, refletidos e transmitidos dentro de um determinado ambiente (STEIGLEDER, 1997, p. 41).

3 DESENVOLVIMENTO

Neste capítulo são abordados as etapas do desenvolvimento do projeto. A primeira seção descreve a instalação dos componentes do ambiente de desenvolvimento. A segunda seção apresenta os principais requisitos do problema trabalhado. A terceira seção descreve a especificação da solução através de diagramas da *Unified Modeling Language* (UML). A quarta seção apresenta a implementação da solução, incluindo os principais trechos de código fonte e exemplos de utilização da aplicação. Por fim, a quinta seção aborda resultados deste trabalho.

3.1 AMBIENTE DE DESENVOLVIMENTO

O aplicativo foi desenvolvido utilizando o Android SDK, e o ambiente de desenvolvimento padrão da plataforma. Detalhes sobre a instalação do Android SDK podem ser obtidos através da referência de Lecheta (2010, p. 31).

Lecheta recomenda a utilização do *Integrated Development Environment* (IDE) Eclipse nas versões 3.4 (Ganymede) e 3.5 (Galileo), mas não funcionaram. A versão utilizada foi 3.6.2 (Helios *Service Release* 2). Instruções para instalação do plug-in ADT para o IDE Eclipse podem ser obtidas na referência de Lecheta (2010, p. 46).

Por fim, para a simulação e testes, a referência de Lecheta (2010, p. 38), trás detalhes sobre a configuração de um Android *Virtual Device* (AVD) e a referência de Lecheta (2010, p. 42) trás detalhes sobre a utilização do emulador Android.

3.2 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

O aplicativo de simulação de partículas deverá:

a) disponibilizar uma aplicação exemplo para demonstrar sua utilização e visualizar a simulação (RF);
b) exibir a simulação de partículas na tela do dispositivo (Requisito Funcional – RF);

c) permitir alterar os parâmetros de cálculo (RF);
d) permitir pausar e continuar a execução durante a simulação de partículas (RF);
e) permitir a troca de dois parâmetros em tempo de execução com toque na tela (RF);
f) considerar um sentido e força para gravidade na simulação das partículas (RF);
g) informar a quantidade de Frames Por Segundo (FPS) na simulação (RF);
h) ser implementado para a plataforma Android (Requisito Não-Funcional – RNF);
i) ser desenvolvido na linguagem Java com o IDE Eclipse (RNF);
j) ser desenvolvido com base no Android SDK 2.2 API Level 8 (RNF);
k) ser executado pelo hardware de um dispositivo móvel sem comprometer o desempenho do sistema operacional (RNF).

3.3 ESPECIFICAÇÃO

A especificação deste trabalho foi desenvolvida utilizando diagramas UML em conjunto com a ferramenta Enterprise Architect 6.5.802 para elaboração dos diagramas de estados, de sequência, de classes e de pacotes.

3.3.1 Diagramas de classes

Nesta seção serão descritas as classes necessárias para o desenvolvimento do sistema de partículas deste trabalho, o relacionamento entre elas e as suas estruturas. Para facilitar o entendimento de como as classes estão reunidas, na Figura 14 são apresentados os pacotes, suas dependências bem como as classes que os compõem.

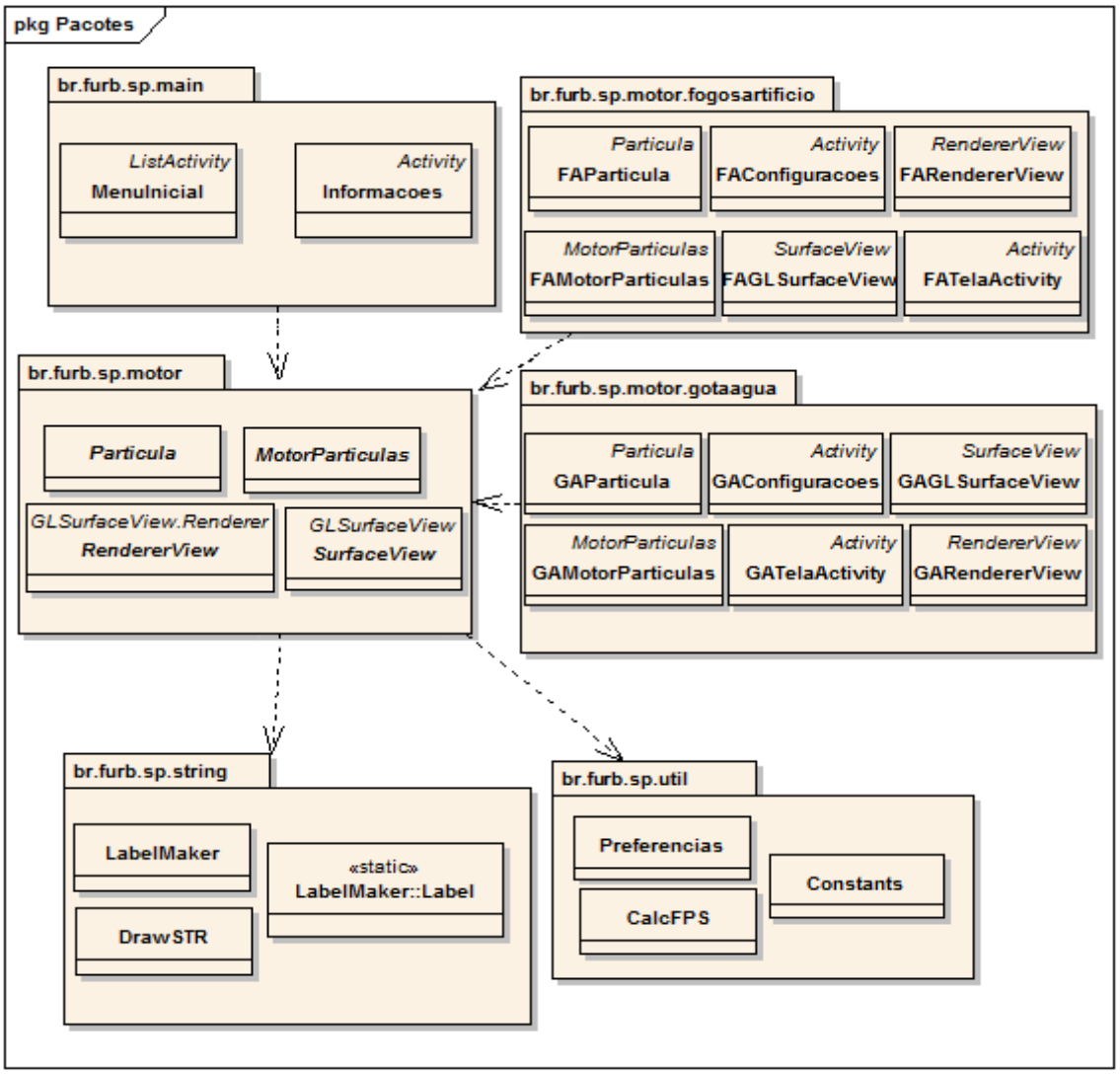


Figura 14 - Diagrama de Pacotes

3.3.1.1 Pacote `br.furb.sp.motor`

A Figura 15 apresenta a especificação das classes do *framework* para sistema de partículas, contido no pacote `br.furb.sp.motor`. Em seguida é feita uma análise sobre a funcionalidade de cada uma delas.

Atendendo a definição de *framework*, a classes do pacote `br.furb.sp.motor` tem por objetivo definir um modelo para as funcionalidades do sistema de partículas e fornecer uma implementação genérica das suas funcionalidades, isto é, elas contém as funcionalidades que são comuns a todas as implementações de motores de partículas. Estas classes são abstratas, portanto não podem ser instanciadas.

A classe `Particula` representa a partícula única dentro do sistema de partículas. Todos

os atributos que definem a partícula estão contidos nesta classe. Os atributos `x`. `y` e `z` são as coordenadas da partícula num sistema cartesiano tridimensional. `vx`, `vy` e `vz` são os vetores velocidade nas respectivas direções. Os demais atributos indicam o tempo de vida, as cores RGB, identificam se a partícula está ativa, se existe um efeito aplicado e qual o tamanho da partícula. Por fim, o atributo `subParticulas` que representa um vetor de partículas. A existência desse vetor de partículas tem por objetivo fornecer a implementação de um sistema de partículas hierárquico.

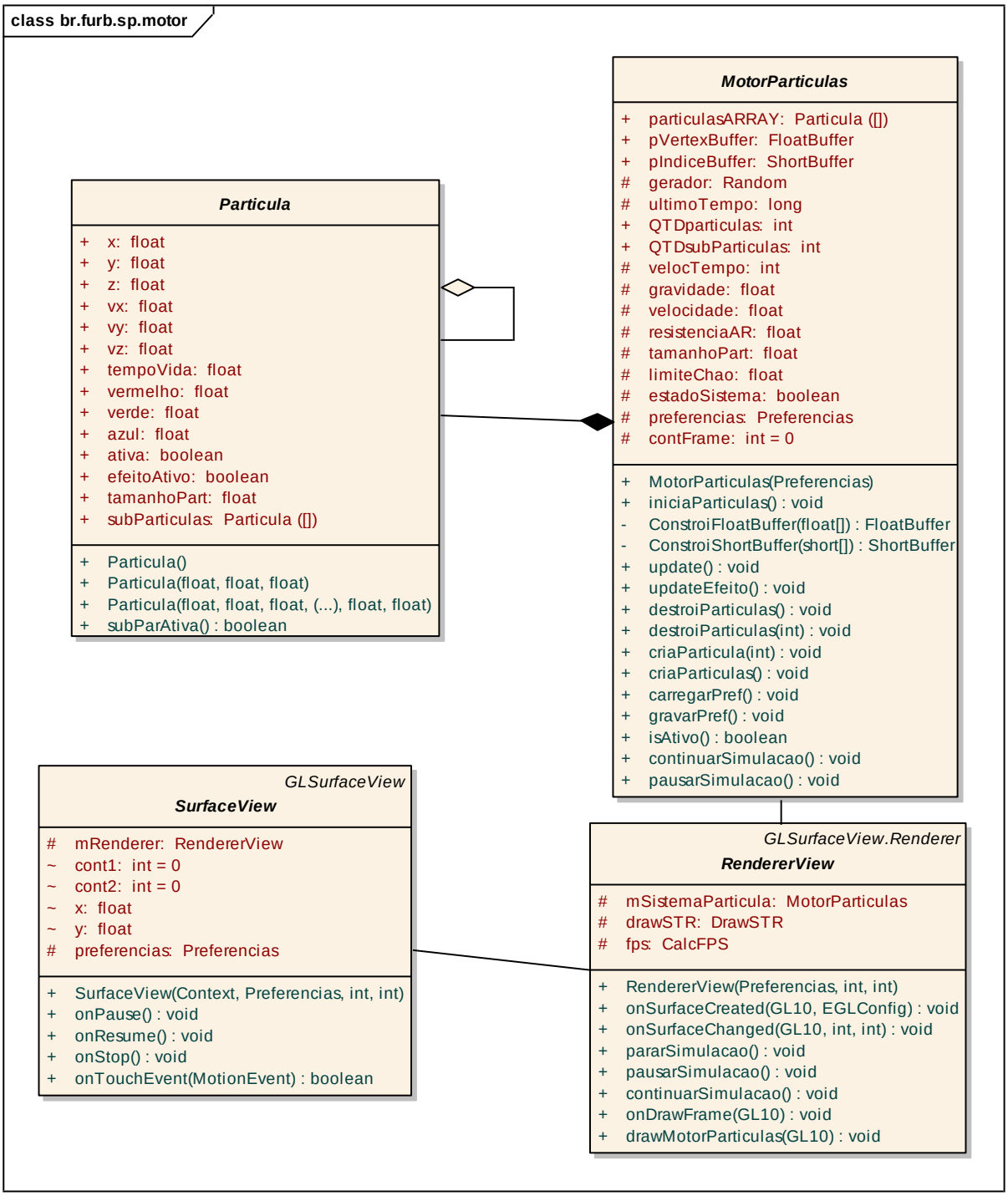


Figura 15 - Classes do pacote `br.furb.sp.motor`

A classe `MotorParticulas` define as funcionalidades mais importantes para a simulação. Nesta classe estão os parâmetros que definem o ambiente e os métodos que instanciam e simulam o comportamento das partículas.

A classe `RendererView` trás uma implementação padrão para a interface `GLSurfaceView.Renderer`. Essa classe é responsável por desenhar e manipular os gráficos em OpenGL ES. Além disso, essa classe contém instancias para classes de utilitários, como desenho de texto na tela e cálculo de Frames Por Segundo FPS.

Assim como a classe `RendererView`, a classe `SurfaceView` também implementa uma interface, a `GLSurfaceView`. Essa classe descende da classe `View`, portanto, é ela que contém os componentes da interface do usuário e é responsável pelo desenho do `RendererView` na tela. Essa classe é responsável por manipular os eventos de toque na tela, para isso ela implementa o método `onTouchEvent(MotionEvent)`. O método `onTouchEvent (MotionEvent)` é utilizado para alterar dois parâmetros em tempo de execução utilizando movimentos de toque na tela na horizontal e na vertical. A partir desta classe que é disparado os eventos de pausar e continuar a simulação através dos métodos `onPause()` e `onResume()`, respectivamente.

3.3.1.2 Pacote `br.furb.sp.main`

A Figura 16 apresenta a especificação do pacote `br.furb.sp.main`.

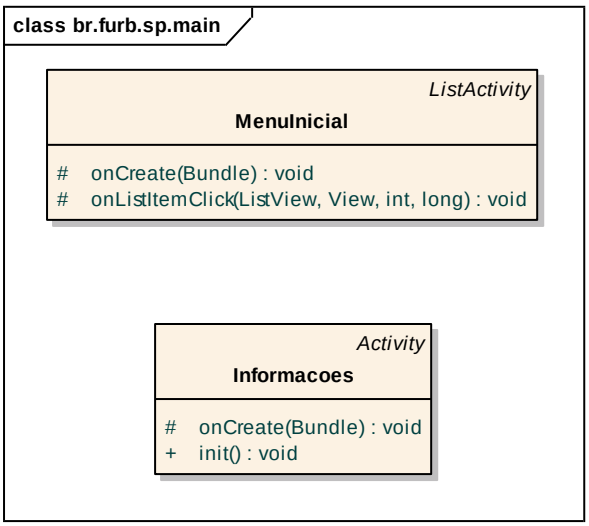


Figura 16 - Classes do pacote `br.furb.sp.main`

A classe `MenuInicial` é a `Activity` inicial. Ela é disparada quando o usuário inicia o aplicativo. A função dessa classe é prover uma lista de simulações disponíveis, para isso, esta

classe estende a classe `ListActivity`. A classe `ListActivity` estende uma `Activity` e possui a estrutura de lista, portanto, a implementação de um *layout* de tela para uma lista torna-se simples e rápida já que ela contém todos os métodos para manipulação de eventos de seleção de itens.

A classe `Informacoes` é uma simples `Activity` que exibe informações sobre a autoria do trabalho.

3.3.1.3 Pacote `br.furb.sp.string`

O OpenGL ES não oferece suporte a desenho de texto. Para isso foi necessário a criação de classes auxiliares que transformem texto em imagens *canvas*. A Figura 17 mostra a ligação e dependências entre as classes.

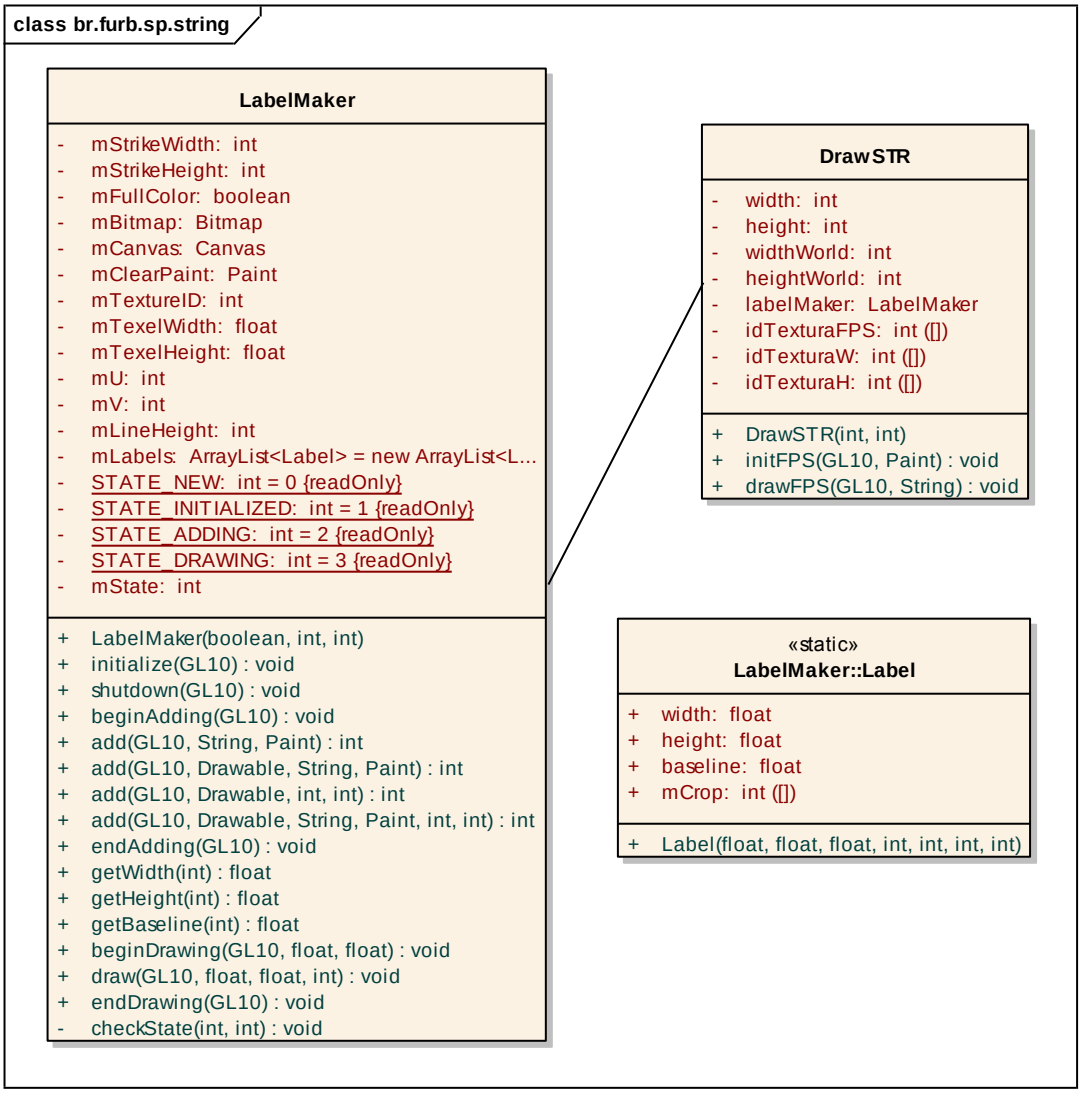


Figura 17 - Classes do pacote `br.furb.sp.string`

A classe `LabelMaker` é fornecida pelo Google como alternativa ao desenho de texto em OpenGL ES e pode ser obtida na referência Google (2007). `Label` é uma classe privada auxiliar também fornecida pelo Google e está no mesmo arquivo fonte da classe `LabelMaker`. A classe `LabelMaker` precisou de adaptações para ser incluída a este trabalho.

A classe `DrawSTR` abstrai a complexidade dos métodos da classe `LabelMaker`, utilizando apenas os métodos necessários ao desenho de texto. Com isso ela fornece três métodos apenas. Sua utilização se limita ao desenho da informação de Frames Por Segundo FPS.

3.3.1.4 Pacote `br.furb.sp.util`

A Figura 18 apresenta o pacote `br.furb.sp.util`. Este pacote contém os utilitários para o sistema.

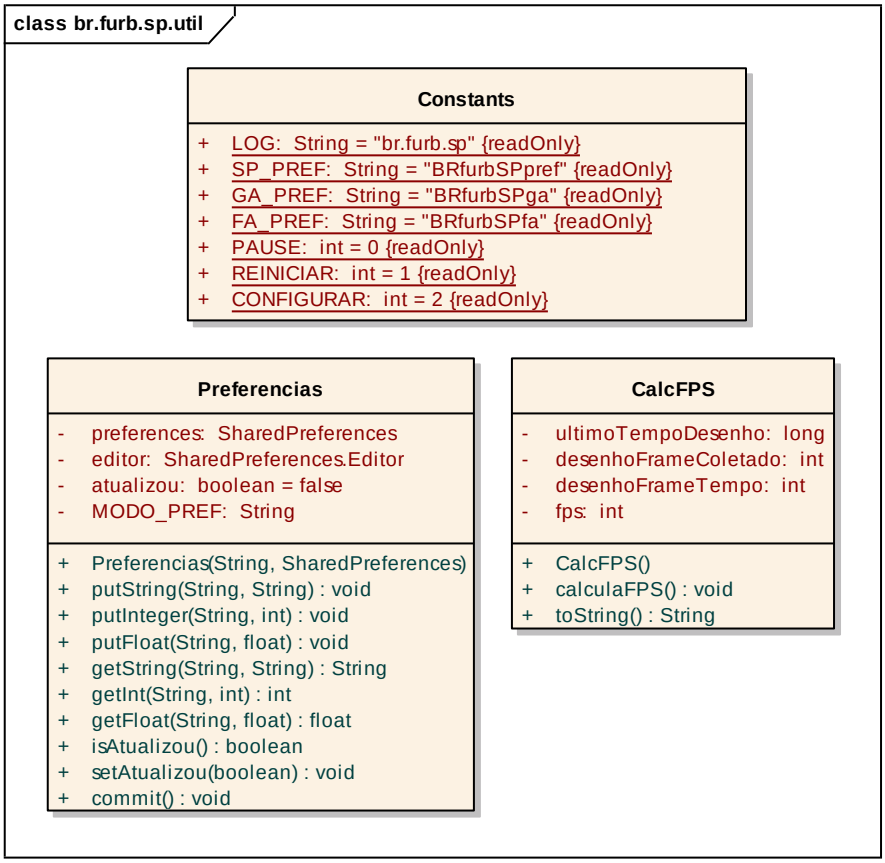


Figura 18 - Classes do pacote `br.furb.sp.util`

A classe `Constants` mantém valores que são utilizadas por diversas classes do sistema. A constante `LOG` define uma nomenclatura padrão para o pacote raiz do projeto, ela é

utilizada como *tag* padrão nas chamadas de gravação do log. As constantes `SP_PREF`, `GA_PREF` e `FA_PREF` são nomenclaturas para a gravação de preferências de cada parte do projeto. As demais constantes `PAUSE`, `REINICIAR` e `CONFIGURAR` definem o comportamento das opções de lista nas *activies* da simulação.

Na classe `CalcFPS` é implementado o cálculo de FPS. Esta classe deve ser instanciada na aplicação que se deseja obter o FPS e a cada execução deve ser chamado o método `calculaFPS()`. O método `toString()` retorna o valor do FPS convertido em texto.

A classe `Preferencias` implementa todos os métodos necessários para gravação e leitura das preferências. O atributo `preferences`, de tipo `SharedPreferences`, é o responsável por buscar as preferencias gravadas no dispositivo. Já o atributo `editor`, do tipo `SharedPreferences.Editor`, é responsável pela gravação das preferências. Antes das preferências serem gravadas elas são persistidas pelo `editor`, para garantir a consistência dos valores.

3.3.1.5 Pacote `br.furb.sp.motor.fogosartificio`

O pacote `br.furb.sp.motor.fogosartificio`, apresentado na Figura 19, contém uma implementação de um motor de sistema de partículas para simular fogos de artifício.

As classes `FAGLSurfaceView`, `FARendererView`, `FAMotorParticulas` e `FAParticulas` são implementações das classes abstratas contidas no pacote `br.furb.sp.motor`. Apenas alguns métodos foram reimplementados para realizar o efeito de fogos de artifício.

A classe `FATelaActivity` é a `Activity` que representa a tela do dispositivo no momento que essa aplicação está em execução. Nesta classe está implementado um menu com três opções: pausar ou continuar; reiniciar; abrir a tela de configurações.

Na classe `FAConfiguracoes` é onde estão os parâmetros que podem ser alterados para mudar o comportamento das partícula. Esta classe é uma `Activity`. Seu *layout* é baseado num xml `configfogosart.xml` que está nos *resources* (`res/layout`) do projeto. Neste *layout* foram adicionados componentes `SeekBar`, `TextView` e `EditText` para oferecer uma interface gráfica amigável ao usuário. Esta classe também mantém uma instância da classe `Preferencias` para poder carregar os parâmetros na inicialização e grava os parâmetros no saída. Ainda nesta classe, foi incluído um menu oferendo opção de *reset* para os parâmetros

da simulação.

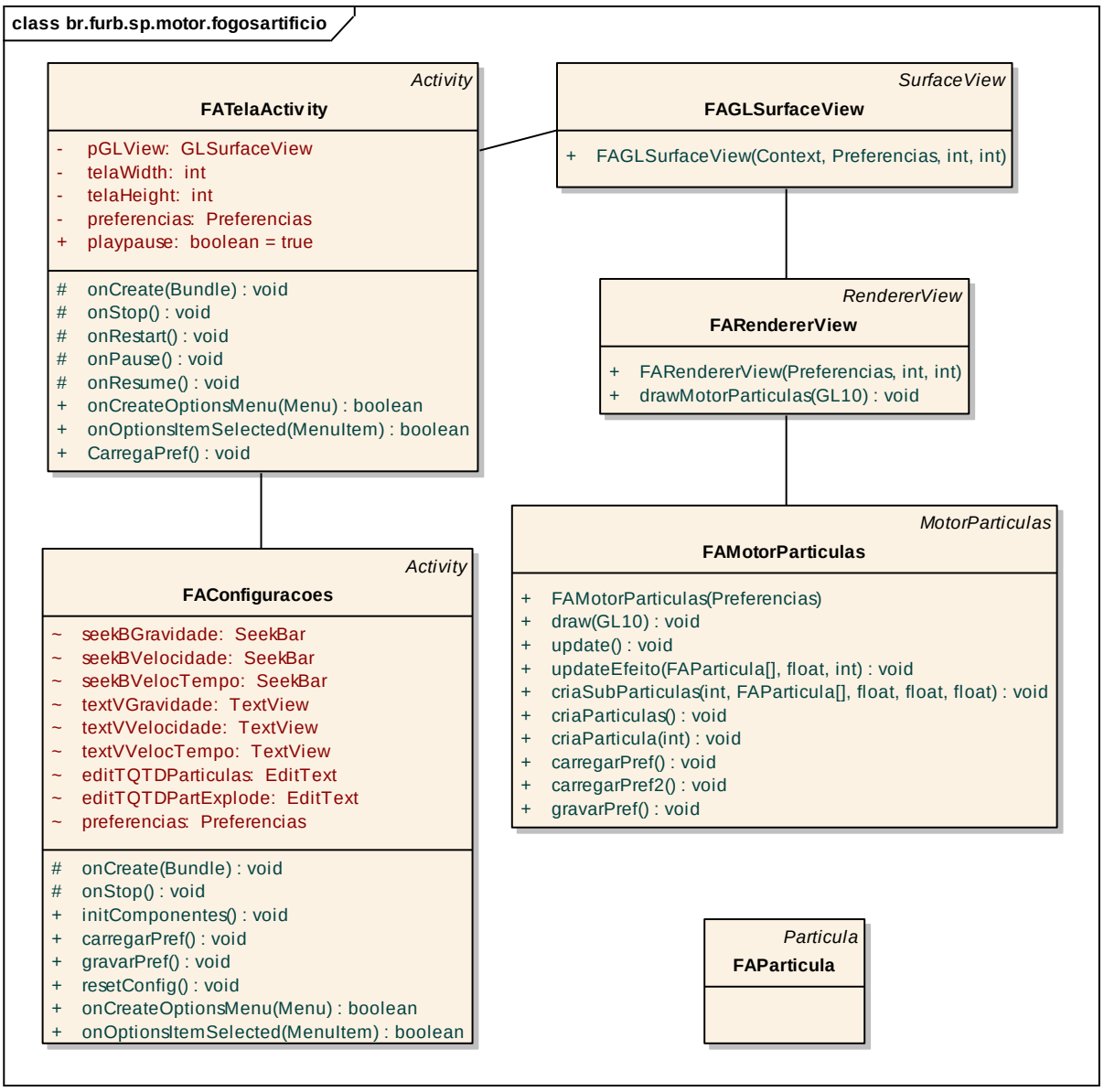


Figura 19 - Classes do pacote `br.furb.sp.motor.fogosartificio`

3.3.1.6 Pacote `br.furb.sp.motor.gotaagua`

O pacote `br.furb.sp.motor.gotaagua`, apresentado na Figura 20, contém as classes que implementam um motor para sistema de partículas que simulam gotas de água caindo e se sumindo.

Sua implementação obedece a mesma estrutura das classes abstratas do pacote br.furb.sp.motor e as classes adicionadas foram as mesmas que o simulador de fogos de artificio implementou, com alguns ajustes para simular o efeito de gotas de água.

Diferente do simulador de fogos de artifício, a classe `GARendererView` reimplementou o método `onSurfaceCreated(GL10, EGLConfig)` para aumentar a distância entre o observador e a animação.

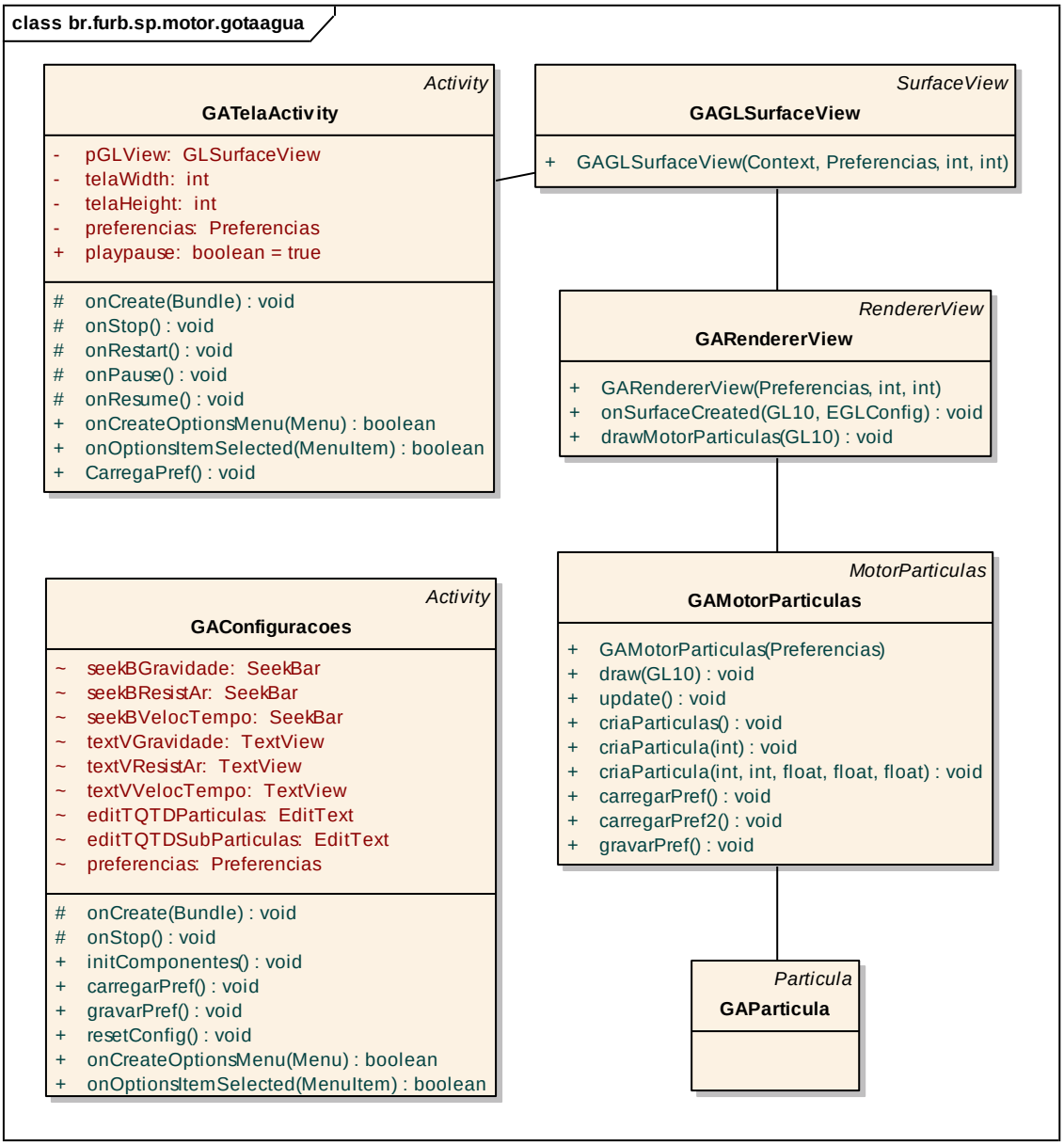
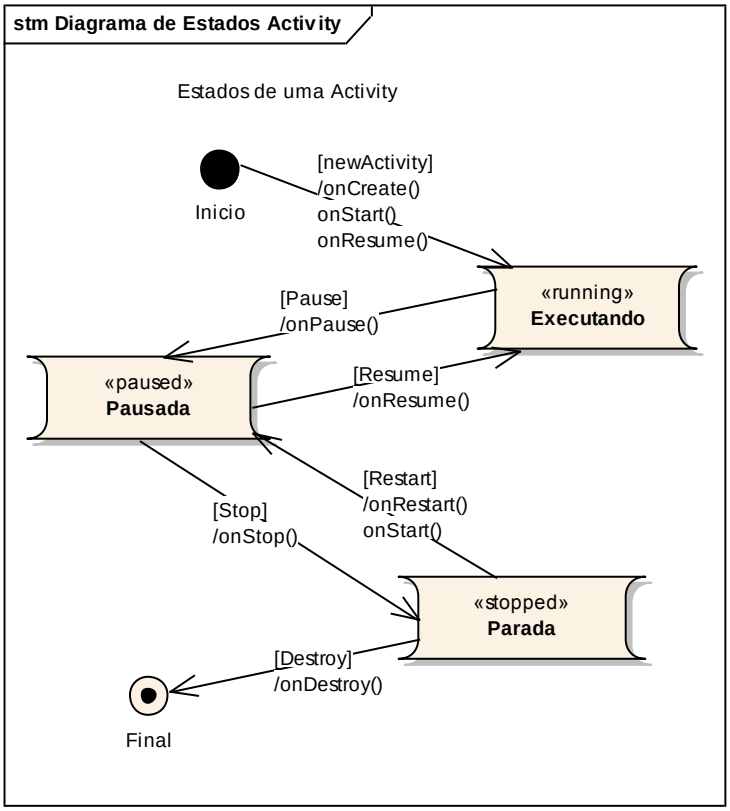


Figura 20 - Classes do pacote `br.furb.sp.motor.gotaagua`

3.3.2 Diagrama de estados

Como a ação de pausar e continuar é um requisito funcional do motor de partículas estudou-se quais componentes ofereceriam suporte para isto. Foi identificado que o ciclo de

vida de uma `Activity` se comporta de forma semelhante com a que o motor precisa se comportar. O diagrama de estados do ciclo de vida da `Activity` é apresentado na Figura 21.



Fonte: adaptado de Motorola (2009).

Figura 21 - Diagrama de estados da `Activity`

A Figura 22 apresenta o diagrama de estados do motor de partículas. Ele contém três estados, semelhantes à `Activity`.

O primeiro estado apresentado é o `Executando` e determina que o motor de partículas esta em execução. O estado `Executando` é atribuído quando:

a) a simulação inicia pela chamada do menu inicial. É executado método `iniciaParticulas()` do próprio motor de partículas;
b) ocorrer ação reiniciar através botão *reiniciar* que está no menu da `Activity` do motor de partículas, então executa o método `onRestart()` da `Activity` e em seguida executa novamente o método `iniciaParticulas()` do motor de partículas;
c) ocorrer ação continuar através do botão *pause/play* que está no menu da `Activity` do motor de partículas, então executa o método `onResume()` da `Activity` e o `onResume()` do motor de partículas.

O estado `Pausado` identifica quando a simulação está em espera. Neste estado o cáclulo de FPS também para de calcular. O estado `Pausado` é atribuído quando:

a) ocorrer ação pausar através do botão *pause/play* que está no menu da `Activity` do motor de partículas, então executa o método `onPause()` da `Activity` e o `onPause()` do motor de partículas;

b) o usuário pressionar o botão *voltar* ou *home* do dispositivo, então executa o método `onPause()` da `Activity` e o `onPause()` do motor de partículas;

c) o usuário entrar na tela de configuração dos parâmetros, então executa o método `onPause()` da `Activity` e o `onPause()` do motor de partículas.

Por fim, o estado `Parado` é atribuído quando o motor de partículas é finalizado. As situações que fazem entrar em estado `Parado` são:

a) quando outra tela é colocada na frente, então executa o método `onPause()` e o `onStop()` da `Activity`, em seguida executa o `destroiParticulas()` do motor de partículas;

b) quando o processo ficar em segundo plano por um período longo e o sistema operacional do Android precisar limpar os recursos para liberar memória.

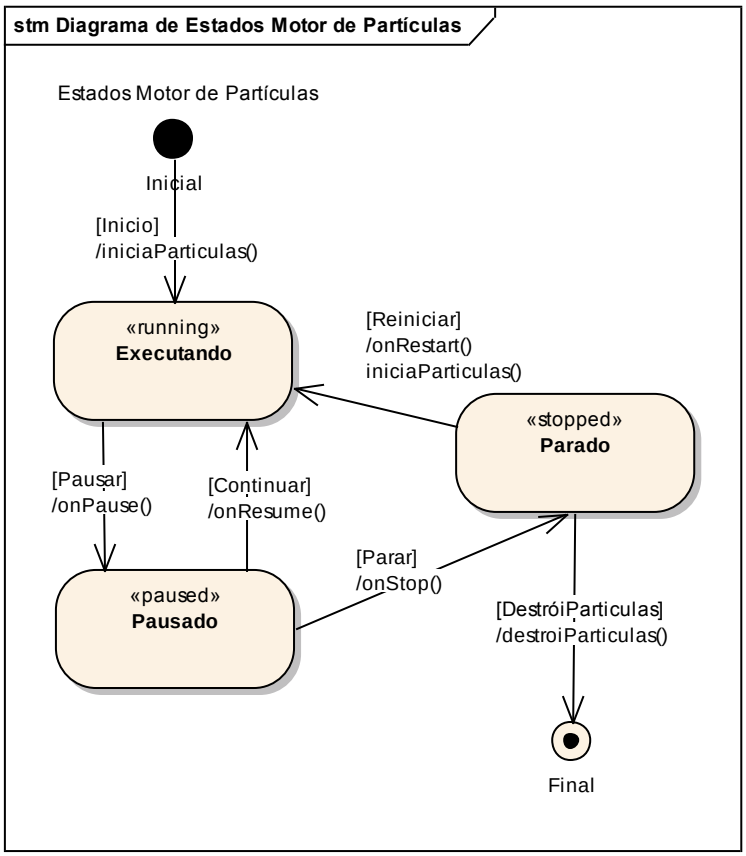


Figura 22 - Diagrama de estados do motor de partículas

3.3.3 Diagrama de sequência

O diagrama de sequência da Figura 23 mostra a interação do `Usuário` com o sistema de partículas.

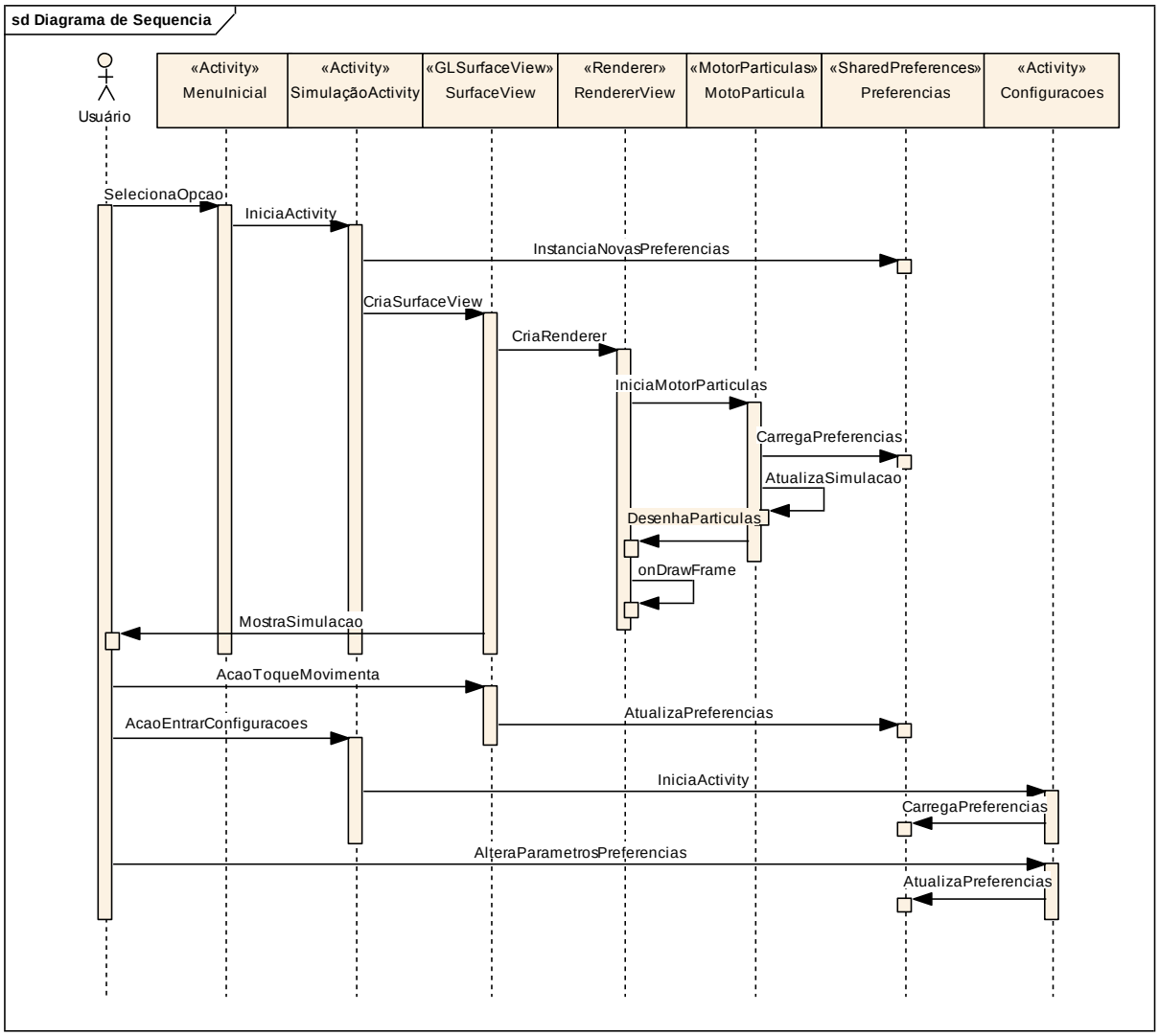


Figura 23 - Diagrama de sequência

Primeiramente o usuário deve selecionar o tipo de simulação que deseja visualizar. Selecionando uma opção, a Activity é disparada. Em seguida é criado uma nova instância de `Preferencias`, `SurfaceView`, `RendererView` e do `MotorParticulas`. Com o `MotorParticulas` criado, é carregado as `Preferencias`, a simulação recebe o estado de `Executando` e começa o *loop* da atualização do comportamento das partículas e do desenho na tela.

Também na Figura 23, é apresentado a ação do `Usuário` alterar os parâmetros da simulação, através do toque na tela ou acessando a tela de configurações que está no menu da

simulação das partículas.

3.4 IMPLEMENTAÇÃO

A seguir são descritas as técnicas e ferramentas utilizadas na implementação e os detalhes das principais classes e rotinas implementadas na aplicação exemplo.

3.4.1 Técnicas e ferramentas utilizadas

O desenvolvimento do *framework* de sistema de partículas foi feito na linguagem Java no ambiente de desenvolvimento Eclipse na versão 3.6.2 (Helios *Service Release* 2) (ECLIPSE FOUNDATION, 2012) combinado com o Android SDK da versão 2.2 (GOOGLE, 2012f) e os plug-in Android *Development Tools* (ADT) da revisão 17.

O desenvolvimento foi feito para a plataforma Android 2.2 (API *Level* 8), conhecido também pelo codinome Froyo. Foi utilizado OpenGL ES 1.0 pois, de acordo com a referência do Google (2012g), a OpenGL ES 2.0 não é suportada pela atual versão do emulador de Android, o que dificultaria a execução e testes da aplicação.

Para execução e depuração da aplicação foi utilizado o emulador de Android na versão 2.2 e um dispositivo *smartphone* da fabricante Motorola do modelo XT860, também conhecido como Milestone 3, na versão 2.3.2 do Android.

O dispositivo possui um processador TI OMAP4430 de 2 núcleos com 1 *Giga Hertz* (GHz) cada, 512 *Mega Bytes* (MB) de *Random Access Memory* (RAM), resolução de 540 x 960 *pixels* e tela de 4 polegadas. Sua GPU é PowerVR SGX 540 de 200 *Mega Hertz* (MHz).

3.4.2 Arquivo `AndroidManifest.xml`

O arquivo `AndroidManifest.xml` desempenha um papel essencial para uma aplicação desenvolvida na plataforma Android. Nele está contido as informações básicas para a aplicação. Algumas são opcionais, como o pacote, o código da versão, nome da versão, e outras são obrigatórias, como os nomes das *activities* e qual é a *activity* de partida da

aplicação. O Quadro 12 apresenta o inicio do arquivo `AndroidManifest.xml`.

Dentro da *tag* `<manifest>`, está declarado o pacote principal do projeto com o código `package="br.furb.sp"` e o código e nome da versão, `android:versionCode="2"`, `android:versionName="2.5"`, respectivamente. Em seguida é informado qual a versão do Android mínima necessária para poder instalar e executar a aplicação, com a *tag* `<uses-sdk android:minSdkVersion="8" />`. Logo abaixo estão as informações sobre o ícone padrão e o nome da aplicação nos atributos `android:icon` e `android:label` dentro da *tag* `<application>`.

As *tags* `<activity>` contém a especificação de cada *activity* que compõe a aplicação. O atributo `android:name="br.furb.sp.main.MenuInicial"` indica o nome da *activity*. Dentro da *tag* `<intent-filter>`, estão identificadas algumas configurações da *activity*. A *tag* `<action android:name="android.intent.action.MAIN" />` define que essa *activity* pode ser iniciada isoladamente como ponto de partida da aplicação. A *tag* `<category android:name="android.intent.category.LAUNCHER" />` define que a categoria dessa *activity* é `LAUNCHER`, e estará disponível para o usuário na tela inicial junto com os outros aplicativos.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="br.furb.sp"
    android:versionCode="2"
    android:versionName="2.5" >
    <uses-sdk android:minSdkVersion="8" />
    <application  android:icon="@drawable/ic_launcher"
                  android:label="@string/app_name" >
        <activity
            android:name=".main.MenuInicial"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category
                  android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".motor.fogosartificio.FATelaActivity"
                  android:label="@string/app_fa" >
        </activity>
        <!-- ... -->
        <activity android:name=".motor.gotaagua.GAConfiguracoes" >
        </activity>
        <activity android:name=".main.Informacoes"
                  android:label="@string/app_about" >
        </activity>
    </application>
</manifest>
```

Quadro 12 - Arquivo `AndroidManifest.xml`

Ainda no Quadro 12, após definir a *activity* inicial, estão as declarações das demais

activities da aplicação. O atributo `android:label="@string/app_fa"` define o nome padrão para a *activity* que será exibido no topo da aplicação.

3.4.3 Recursos auxiliares

Os arquivos contidos no diretório `/res/layout` são os recursos de *layout* para interface gráfica. Esses recursos de *layout* são arquivos XML que servem para separar o design de tela da codificação e manipulação dos dados.

Parte do arquivo que contém o *layout* da tela de configuração do simulador de fogos de artifício pode ser observado no Quadro 13.

A *tag* `<ScrollView>` é utilizada para inserir uma rolagem dos itens da tela, utilizada quando se tem muitos elementos na tela. Os atributos `android:layout_width` e `android:layout_height` indicam os tamanhos de largura e altura. O valor `"fill_parent"` indica que o componente irá ocupar todo o tamanho definido por seu *layout* pai. Já o valor `"wrap_content"` indica que o componente irá usar apenas o tamanho necessário na tela.

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content" >
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical"
        android:paddingLeft="10px"
        android:paddingRight="10px" >
        <TextView
            android:id="@+id/textView6"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Quantidade Particulas"
            android:textAppearance=
                            "?android:attr/textAppearanceLarge" />
        <EditText
            android:id="@+id/FAeditTQTDParticulas"
            android:layout_width="125dp"
            android:layout_height="wrap_content"
            android:ems="10"
            android:inputType="number" >
            <requestFocus />
        </EditText>
      <!-- ... -->
```

Quadro 13 - Primeiro trecho do arquivo `configfogosart.xml`

A *tag* `<LinearLayout>` é utilizada para criar um gerenciador de *layout* onde é possível organizar os componentes na horizontal ou na vertical. O atributo `android:orientation` é o

responsável por controlar a orientação do *layout*.

No Quadro 14 é apresentado outro trecho do arquivo `configfogosart.xml` que trás outros componentes do *layout* de tela. Esses componentes representam os parâmetros do simulador de partículas, sendo eles:

a) `<TextView>`: representa um texto na tela, exibe uma informação;

b) `<EditText>`: representa um campo de texto para digitar informações;

c) `<SeekBar>`: representa uma barra de progresso com um ícone ajustável, exibe e altera um parâmetro numérico com valores limite superiores e inferiores.

O atributo `android:id` é o responsável por manter a referencia para manipular o componente dentro de uma *activity* que usa o *layout*.

```
        <!-- ... -->
        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content" >
            <EditText
                android:id="@+id/FAeditTQTDPartExplode"
                android:layout_width="127dp"
                android:layout_height="wrap_content"
                android:ems="10"
                android:inputType="number" />
            <TextView
                android:id="@+id/textView8"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="    ( + random(20) )"
    android:textAppearance="?android:attr/textAppearanceMedium" />
        </LinearLayout>
        <!-- ... -->
        <SeekBar
            android:id="@+id/FAseekBGravidade"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:max="2000"
            android:progress="1" />

        <TextView
            android:id="@+id/textView13"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text=" "
          android:textAppearance="?android:attr/textAppearanceSmall" />
```

Quadro 14 - Segundo trecho do arquivo `configfogosart.xml`

O arquivo `configgotaagua.xml`, que representa a configuração do simulador de gotas de água, repete a estrutura do arquivo `configfogosart.xml`, com algumas diferenças na organização e nomenclatura do itens exibidos na tela.

Também como recurso de *layout*, o arquivo `informacoes.xml` foi criado com a finalidade de se obter uma referência para o objetivo e autoria do trabalho dentro da

aplicação.

Conforme apresentado no Quadro 15, outro recurso auxiliar ao projeto é o arquivo de valores `strings.xml` que fica no diretório `/res/values/`. Foram utilizados esses recursos para definir os nomes da aplicação e nome das telas *activies*.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">TCC Sistema de Partículas Angel Vitor</string>
  <string name="app_fa">Simular Fogos Artifício</string>
  <string name="app_ga">Simular Gota de Água</string>
  <string name="app_about">Informações TCC Angel Vitor Lopes</string>
</resources>
```

Quadro 15 - Recurso auxiliar de valores strings.xml

3.4.4 Partículas

A classe mais básica dos motores de simulação de partículas é a `Particula`, pois elas representam as partículas únicas. Ela é responsável por manter os atributos de cada partícula que está em simulação.

No Quadro 16 é apresentado um trecho do código onde estão os atributos da `Particula`, lembrando que esta classe é abstrata pois serve de referência para todos os motores de partículas criados e está contida no pacote `br.furb.sp.motor`. Os atributos são públicos visando a simplicidade e otimização sobre processamento e memória.

```
package br.furb.sp.motor;

public abstract class Particula {

      public float      x, y, z;    // posição
      public float      vx, vy, vz; // velocidade
      public float      tempoVida;  // tempo de vida
      public float      vermelho,   // Cores RGB
                        verde,
                        azul;
      public boolean    ativa,      // Indica atividade
                        efeitoAtivo;// indica efeito ativo
      public float      tamanhoPart;      // tamanho da partícula
      public Particula[]
                       subParticulas;     // Particulas
```

Quadro 16 - Atributos da classe `Particula`

Esta classe possui três construtores:

a) `Particula()`: somente seta os atributos `ativa` e `efeitoAtivo` como falso;

b) `Particula(float novaPosX, float novaPosY, float novaPosZ)`: seta os atributos posição das partículas;

c) `Particula(float novaPosX, float novaPosY, float novaPosZ, float velX, float velY, float velZ, float vermelho, float verde, float azul)`: inicializa a maioria dos atributos da partícula.

O método `subParAtiva()`, conforme apresentado no Quadro 17 , percorre o *array* de subpartículas procurando por partículas ativas, caso alguma esteja ativa, ele retorna imediatamente verdadeiro, se chegar ao final do método ele retorna falso.

```
/**
 * Verifica se existe pelo menos uma sub particula ativa
 * @return
 */
public boolean subParAtiva(){
      for (int i = 0; i < subParticulas.length; i++) {
            if (subParticulas[i].ativa)
                  return true;
      }
      return false;
}
```

Quadro 17 - Método `subParAtiva()` da classe `Particula`

3.4.5 Motor de partículas

Dentre todas as classes da aplicação, pode-se dizer que a `MotorParticulas` é a que acumula a maior quantidade de funções. Também, lembrando que esta classe é abstrata pois serve de referência para todos os motores de partículas criados e está contida no pacote `br.furb.sp.motor`.

No Quadro 18 contém a codificação dos atributos da classe `MotorParticulas`. Seguindo a proposta de simplicidade e otimização do desempenho, vários atributos também são públicos.

No Quadro 19 é apresentado o método `iniciaParticulas()`. Este método é responsável por criar novas instancias de atributos auxiliares e inicializar os *buffers* de ordem nativa dos vértices para desenho das partículas.

Os outros métodos são implementados nos motores descendentes deste, pois, a maioria necessita de controle específicos para modelar seu efeito de simulação.

```
public abstract class MotorParticulas {

	// Array para armazenar as partículas
	public Particula[] particulasARRAY;
	// Buffers para desenhar as particulas
	public FloatBuffer pVertexBuffer;
	public ShortBuffer pIndiceBuffer;
	// Gerador
	protected Random gerador;
	// Variavel auxiliar para calcular o framerate
	protected long ultimoTempo;
	// Variável Auxiliar para Update
	protected int contFrame = 0;

	// Quantidade de partículas e SubParticulas
	public int QTDparticulas;
	public int QTDsubParticulas;
	// Velocidade Tempo
	protected int velocTempo;
	// Gravidade
	protected float gravidade;
	// Velocidade das Particuals
	protected float velocidade;
	// Resistencia do Ar
	protected float resistenciaAR;
	// Tamanho da Particula
	protected float tamanhoPart;
	// Chão
	protected float limiteChao;
	// Estado Em execução Pausado Parado
	protected boolean estadoSistema;
	// Preferencias
	protected Preferencias preferencias;
```

Quadro 18 - Atributos da classe `MotorParticulas`

```
	public void iniciaParticulas() {

		// Random para gerador de partículas
		gerador = new Random(System.currentTimeMillis());

		// Inicializa Particulas
		criaParticulas();

		// cria triangulo para servir de particula
		float[] coord = {
				-this.tamanhoPart, 0.0f, 0.0f,
				this.tamanhoPart, 0.0f, 0.0f,
				0.0f, this.tamanhoPart, 0.0f };

		// coordenadas indices
		short[] icoord = { 0, 1, 2 };

		// Constroi buffers nativos para os pontos das partículas
		pVertexBuffer = ConstroiFloatBuffer(coord);
		pIndiceBuffer = ConstroiShortBuffer(icoord);

		this.ultimoTempo = System.currentTimeMillis();
	}
```

Quadro 19 - Método `iniciaParticulas()` da classe `MotorParticulas`

3.4.5.1 Motor de partículas para fogos de artifício

A classe `FAMotorParticulas`, que é responsável por implementar a simulação de partículas de fogos de artifício, está no pacote `br.furb.sp.motor.fogosartificio`. Esta classe estende de `br.furb.sp.motor.MotorParticulas`.

Assim que a classe é instanciada, ela faz uma chamada para o método `iniciaParticulas()`, que lhe foi herdado da classe pai, e em seguida já inicia o *loop* da atualização do comportamento das partículas, método `update()`. O algoritmo de `update()` é apresentado em duas partes. A primeira é apresentada no Quadro 20, foi abstraído alguns trechos de código irrelevantes que não interferem no entendimento do mesmo.

Logo que o código inicia, é feito uma verificação para saber se alguma das preferências que são alteradas em tempo de execução foram alteradas, essas são as preferências que são atualizadas com o toque na tela. Essa verificação é executada apenas a cada 5 *updates* para não prejudicar o processamento.

Em seguida, se o sistema de partículas ainda estiver ativo, calcula o `frameRate` (fator de tempo, referência para o tempo percorrido durante a simulação). Ele é calculado pegando o tempo atual, subtraindo pelo `ultimoTempo` que foi executado, tudo em milissegundos, e em seguida é feito a divisão pelo `velocTempo` que é o parâmetro para definir uma medida para o fator de tempo da simulação.

Com o fator de tempo calculado, começa a simulação do comportamento. Então é executado uma repetição para percorrer todos as partículas ativas do sistema.

Dentro da repetição, o algoritmo verifica se a partícula percorrida está ativa. Em seguida verifica se o efeito da partícula está ativo. Caso o efeito esteja ativo, é chamado o método `updateEfeito(FAParticula[], float, int)`, para executar somente os cálculos do efeito das partículas.

O Quadro 21 contém a segunda parte da implementação do método `update()`.

O fator que faz as partículas mudarem de forma e explodirem é a coordenada `y`, que representa a coordenada vertical. Foi considerado que o limite vertical seria o limite máximo do tempo de vida da partícula. Então, a partícula irá começa a subir com a ajuda do vetor velocidade aplicado no início, e quando começar a cair foi porque atingiu o ponto máximo de altitude, então é chamado o método de criar subpartículas e a partícula pai morre. Na prática, essa partícula é marcada como inativa no vetor de partículas e quando for lançada novamente ela troca alguns atributos principais e repete o processo de lançamento de partícula. Foi

utilizado este método para se ganhar tempo na simulação com a reutilização de objetos.

O primeiro parâmetro a ser calculado é o vetor velocidade. Como a partícula está se movendo em um ambiente onde são consideradas as três dimensões, os três atributos de vetor de velocidade devem ser atualizados, `particulasARRAY[i].vx`, `particulasARRAY[i].vy` e `particulasARRAY[i].vz`.

```
Public void update() {

// Tempo de execução a cada 5 updates, mas somente se foi atualizado
(...)
carregarPref2();
// Calcula tempo dos frames para definir velocidade
(...)

// Somente quando o sistema estiver Ativo
if (this.estadoSistema) {

   // Calcula o Frame Rate (Fator de tempo)
     float frameRate = (float) (tempoAtual - this.ultimoTempo) /
                                this.velocTempo;
     // Movimento das partículas
     // Percorre todas as partículas do sistema
     for (int i = 0; i < this.QTDparticulas; i++) {

        // Somente quando a partícula estiver ativa
        if (particulasARRAY[i].ativa) {
          // Verifica se a partícula tem efeito
          if (particulasARRAY[i].efeitoAtivo) {

            updateEfeito((FAParticula[])
                         particulasARRAY[i].subParticulas, frameRate,
                         i);
          } else {
          // Atualiza somente as coordenadas da partícula
```

Quadro 20 - Primeira parte do método `update()` da classe `FAMotorParticulas`

Sobre o vetor velocidade, primeiro é subtraído a resistência do ar nas três dimensões. Não foi considerado a massa da partícula, a densidade do ar, nem a área da seção transversal do corpo.

Em seguida é aplicado a gravidade ao vetor velocidade. Foi considerado que a gravidade apenas atua no sentido para baixo, pois estamos desprezando a massa das partículas, então a gravidade é subtraída do vetor velocidade `y` (`vy`).

Após ser calculado a gravidade, já é calculada a nova posição de `y` isoladamente, pois será necessário ter esse valor atualizado antes dos demais para saber o limite de altura.

A atualização da posição é obtida através da posição anterior mais o vetor velocidade multiplicado pelo fator de tempo. Em seguida é verificado se a partícula ainda está se movimentando na direção do vetor velocidade, ou se a partícula começou a descer.

Caso ela tenha começado a descer, são instanciadas as novas subpartículas e começa a

simulação a ter efeito.

Caso contrário, se a partícula ainda estiver subindo apenas se atualiza os valores posição `x`, `y` e `z`.

Por fim, caso o sistema não esteja ativo, apenas atualiza o fator de tempo, pois o tempo continua a percorrer, apenas foi pausado a simulação.

```
            // Atualiza velocidades reduzindo da Resistencia do AR
            particulasARRAY[i].vx = particulasARRAY[i].vx
                 - (particulasARRAY[i].vx * resistenciaAR * frameRate);
            particulasARRAY[i].vy = particulasARRAY[i].vy
                 - (particulasARRAY[i].vy * resistenciaAR * frameRate);
            particulasARRAY[i].vz = particulasARRAY[i].vz
                 - (particulasARRAY[i].vz * resistenciaAR * frameRate);

            // Atualiza o vetor velocidade do eixo Y
            // Aplicando a Gravidade
            particulasARRAY[i].vy = particulasARRAY[i].vy
                                  - (gravidade * frameRate);

              // Calular a coordenada Y antes pra poder detectar
              // quando estiver atingido o ponto máximo em Y
              float novoY = particulasARRAY[i].y
                          + (particulasARRAY[i].vy * frameRate);

              // Verifica quando a partícula atingiu o ponto máximo e
              // começa a descer
              if (novoY < particulasARRAY[i].y) {

                // Explode Partícula
                particulasARRAY[i].subParticulas = new
                                  FAParticula[gerador.nextInt(20) +
                                              this.QTDsubParticulas];

                // Executa criação das partículas de explosão
                criaSubParticulas( i, (FAParticula[])
                            particulasARRAY[i].subParticulas,
                            particulasARRAY[i].x, particulasARRAY[i].y,
                            particulasARRAY[i].z);

                // Marca que efeito está ativo
                particulasARRAY[i].efeitoAtivo = true;
              } else {
                // movimento conforme a velocidade
                particulasARRAY[i].x = particulasARRAY[i].x
                     + (particulasARRAY[i].vx * frameRate);
                particulasARRAY[i].z = particulasARRAY[i].z
                     + (particulasARRAY[i].vz * frameRate);
                // Altura
                particulasARRAY[i].y = novoY;
              }
            } // end if efeito
          }
        }
      } else {
        this.ultimoTempo = tempoAtual;
}
```

Quadro 21 - Segunda parte do método `update()` da classe `FAMotorParticulas`

No Quadro 22 é apresentado um trecho do método `updateEfeito()`.

Os métodos de atualização do comportamento é semelhante a partícula pai, a maior diferença fica no tempo de vida.

No método de criação de subpartículas é atribuído um tempo de vida utilizando uma função randômica. Quando esse tempo de vida termina, a partícula recebe status de inativa e executa o método `criaParticula(ind)`, que faz criar novamente a partícula pai.

```
  public void updateEfeito(FAParticula[] parExplode, float frameRate,
int ind) {
       // Percorre todas as subpartículas
    for (int i = 0; i < parExplode.length; i++) {
       // Aplica Resistencia do AR
       parExplode[i].vx = parExplode[i].vx
            - (parExplode[i].vx * resistenciaAR * frameRate);
       parExplode[i].vy = parExplode[i].vy
            - (parExplode[i].vy * resistenciaAR * frameRate);
       parExplode[i].vz = parExplode[i].vz
            - (parExplode[i].vz * resistenciaAR * frameRate);

       // Move a Particula de acordo com a velocidade
       parExplode[i].x = parExplode[i].x + (parExplode[i].vx *
                                                    frameRate);
       parExplode[i].y = parExplode[i].y + (parExplode[i].vy *
                                                    frameRate);
       parExplode[i].z = parExplode[i].z + (parExplode[i].vz *
                                                    frameRate);
       // Efeito de Gravidade, diminui a gravidade
       parExplode[i].y = parExplode[i].y - (gravidade * frameRate);

       // Diminui Tempo de Vida
       parExplode[i].tempoVida = parExplode[i].tempoVida - frameRate;

       // Se a partícula atingiu seu tempo limite desativa
       if (parExplode[i].tempoVida < 0f) {
         particulasARRAY[ind].efeitoAtivo = false;
         particulasARRAY[ind].ativa = false;
         // Cria novamente a partícula que lança os fogos para o alto
         criaParticula(ind);
       }
    }
  }
```

Quadro 22 - Método `updateEfeito()` da classe `FAMotorParticulas`

Outro método importante é o `draw(GL10 gl)` que executa as rotinas de desenho da simulação em OpenGL ES 1.0. O método `draw(GL10 gl)` é apresentado no Quadro 23.

Este método apresenta as funções básicas para desenho de um buffer de vértices e esta dividido em duas partes. A primeira contém a repetição onde se percorre todas as subpartículas, caso o efeito esteja em execução, e as desenha. Na segunda parte, são desenhados apenas as partículas únicas, caso não esteja com efeito em execução.

Para as duas partes de código, a forma de desenhar é idêntica, desde o método `gl.glPushMatrix()` até o `gl.glPopMatrix()`.

```
  public void draw(GL10 gl) {
    gl.glVertexPointer(3, GL10.GL_FLOAT, 0, pVertexBuffer);
    // percorre todas as partículas
    for (int i = 0; i < QTDparticulas; i++) {
      // somente desenha quando estiver ativa
      if (particulasARRAY[i].ativa) {
        // Desenha a Particula, caso ela nao tenha explodido
        if (!particulasARRAY[i].efeitoAtivo) {
          gl.glPushMatrix();
          gl.glColor4f(particulasARRAY[i].vermelho,
               particulasARRAY[i].verde, particulasARRAY[i].azul,
               1f);
          gl.glTranslatef(particulasARRAY[i].x, particulasARRAY[i].y,
               particulasARRAY[i].z);
          gl.glDrawElements(GL10.GL_TRIANGLES, 3,
               GL10.GL_UNSIGNED_SHORT, pIndiceBuffer);
          gl.glPopMatrix();
          // desenha o efeito da partícula
        } else {
          for (int j=0; j<particulasARRAY[i].subParticulas.length; j++) {
            gl.glPushMatrix();
            gl.glColor4f(
                 particulasARRAY[i].subParticulas[j].vermelho,
                 particulasARRAY[i].subParticulas[j].verde,
                 particulasARRAY[i].subParticulas[j].azul,
                 1f);
            gl.glTranslatef(particulasARRAY[i].subParticulas[j].x,
                 particulasARRAY[i].subParticulas[j].y,
                 particulasARRAY[i].subParticulas[j].z);
            gl.glDrawElements(GL10.GL_TRIANGLES, 3,
                 GL10.GL_UNSIGNED_SHORT, pIndiceBuffer);
            gl.glPopMatrix();
          }
          (...)
  }
```

Quadro 23 - Método `draw(GL10 gl)` da classe `FAMotorParticulas`

3.4.5.2 Motor de partículas para gotas de água

A metodologia da implementação do motor de partículas das gotas de água é semelhante a de fogos de artifício, a diferença é que cada gota já é composta de subpartículas, então no meio da execução ela não tem sua forma alterada. Somente quando atinge um limite baixo, representado como chão, é simulado uma colisão e as partículas são desmembradas e espalhadas.

No Quadro 24 é apresentado o algoritmo do método `update()` da classe `GAMotorParticulas`. Foram abstraídos trechos de código irrelevantes que são semelhantes ao motor de fogos de artifício, tais como os cálculos da velocidade, gravidade e resistência do ar.

Uma consideração importante, é que o valor do vetor velocidade inicial das partículas

de gota de água é zero.

```
public void update() {
  // Tempo de execução a cada 5 updates, mas somente se foi atualizado
  (...)

  if (this.estadoSistema) {

    // Calcula Frame Rate
    (...)
    // Percorre Todas as Partículas
    for(int partInd=0; partInd<this.particulasARRAY.length;partInd++){

      // Percorre Todas as SubPartículas que compõe a gota
      for (int subInd = 0; subInd < this.particulasARRAY[partInd].
                                    subParticulas.length; subInd++) {

        // somente atualiza quando elas tiverem ativas
        If(this.particulasARRAY[partInd].subParticulas[subInd].ativa){

          // Referencia para facilitar as chamadas da subparticula
          GAParticula particula = (GAParticula)
                 this.particulasARRAY[partInd].subParticulas[subInd];

          // Aplica Resistencia do AR
          (...)
          // Aplica velocidade
          (...)
          // Aplica Gravidade
          (...)
          // Colisao com o Chao
          if (particula.y <= this.limiteChao) {

            // Cria uma simples dispersão das partículas
            particula.y = this.limiteChao;
            particula.vy = particula.vy + (gerador.nextFloat() *
                                                1.1f) * -.5f;
            particula.vx = particula.vx + (gerador.nextFloat() *
                                                6f) - 3f;
            particula.efeitoAtivo = true;
          }
          // depois de colidir com o chão, diminui o tempo de vida
          if (this.particulasARRAY[partInd].efeitoAtivo) {
            // Diminui Tempo de Vida
            particula.tempoVida = partícula.tempoVida - frameRate;
          }
          // // Se a partícula atingiu seu tempo limite desativa
          if (particula.tempoVida < 0f) {
            particula ativa = false;
            // Verifica se todas as Partículas morrreram, e cria
            // uma nova caso verdadeiro
            if (this.particulasARRAY[partInd].subParAtiva()) {
              this.particulasARRAY[partInd].ativa = false;
              criaParticula(partInd);
            }
            (...)
```

Quadro 24 - Método `update()` da classe `GAMotorParticulas`

Para colisão com o chão foi utilizado um simples método de verificação se o valor da coordenada `y` atingiu um valor determinado para o chão. Se ocorrer a colisão então é atribuído

novos valores aos vetores velocidade.

Assim que as partículas colidem com o chão, começa a subtrair o tempo de vida delas.

Quando o método `subParAtiva()` da classe `Particula` retornar que não existe mais nenhuma subpartícula ativa, então ele cria uma nova partícula pai, e reinicia todo o processo.

Outros algoritmos relevantes dentro da classe `GAMotorParticulas` estão dentro do método `criaParticula(int partindo)` que é chamado para iniciar cada partícula. O código no Quadro 25 define a aleatoriedade da posição `x` em que as partículas são criadas.

```
          // Posição X é Aleatória
          if (gerador.nextBoolean())
               x = (this.gerador.nextFloat() * (-3f +

     this.gerador.nextFloat()));
          else if (gerador.nextBoolean())
               x = (this.gerador.nextFloat() * (3f +

     this.gerador.nextFloat()));
          else if (gerador.nextBoolean()) x = (this.gerador.nextFloat());
          else x = (-this.gerador.nextFloat());
          // Posição Y é Aleatória mais uma posição 1.9 fixado para altura
          y = this.gerador.nextFloat() + 1.9f;
          z = 0f;
```

Quadro 25 - Início do método `criaParticula()` da classe `GAMotorParticulas`

Como as partículas de gotas apresentam uma forma específica, foi implementado um método para modelar tal efeito. O método desenvolvido é ilustrado na Figura 24.

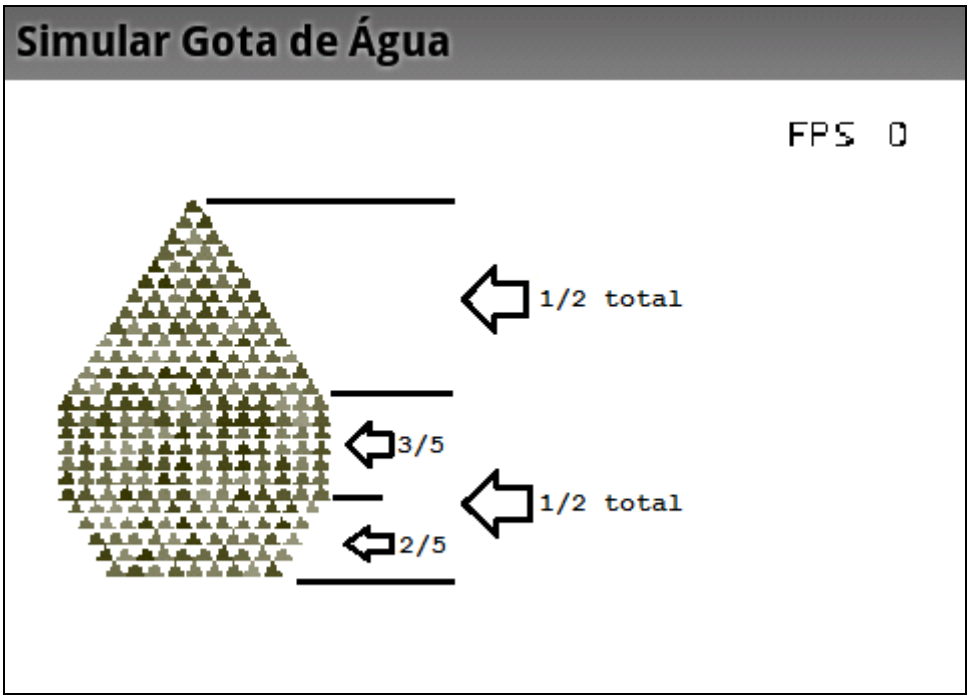


Figura 24 - Método de modelagem da gota de água

O Quadro 26 apresenta o código utilizado para criar dinamicamente as subpartículas no formato de gota.

```
        String raizSTR = Math.sqrt(this.QTDsubParticulas) + "";
        raizSTR = raizSTR.substring(0, raizSTR.indexOf("."));
        // Divide a raiz quadrada das particulas em 4 partes
        float raizQtdPart = Float.parseFloat(raizSTR) / 4;

        // Variaveis temporarias para o posicionamento da gota
        int j = 0, q = 1, qx = 0;
        float xTemp = x, yTemp = y;
        boolean volta1 = true, volta2 = true;

        // Cria array das subpartículas
        particulasARRAY[partInd].subParticulas = new
                                        GAParticula[this.QTDsubParticulas];
        // Percorre todas as partículas e cria instancias de cada uma
        for (int subInd = 0; subInd < particulasARRAY[partInd].
                                          subParticulas.length; subInd++){

          // Cria particula e codigo abaixa ja prepara a proxima posicao
          criaParticula(partInd, subInd, x, y, z);
          // Rotina para saber a posição que pintar a próxima subparticula
          j++; // Marca que pintou um elemento
          x = x - 0.1f; // diminui 1 da posicao x da proxima partícula

          // Regra pula linha para desenhar as particulas
          if (q==j && q <= (raizQtdPart * 3 + raizQtdPart / 2) && volta1){
            xTemp = xTemp + this.tamanhoPart; // Posiciona Coluna
            yTemp = yTemp - this.tamanhoPart; // Posiciona Linha
            x = xTemp; y = yTemp; q++; j = 0;

          } else if (q==j && qx <= raizQtdPart + raizQtdPart/2 && volta2){
            volta1 = false;
            yTemp = yTemp - this.tamanhoPart; // Posiciona Linha
            x = xTemp; y = yTemp;
            qx++; // indica quantos espaços da parte igual vai pintar
            j = 0;

          } else if (q == j) {
            volta2 = false;
            xTemp = xTemp - this.tamanhoPart; // Posiciona Coluna
            yTemp = yTemp - this.tamanhoPart; // Posiciona Linha
            x = xTemp; y = yTemp; q--; j = 0;
          }
        }
```

Quadro 26 - Final do método `criaParticula()` da classe `GAMotorParticulas`

3.4.6 Preferências

A classe `Preferencias` é a responsável por fazer a comunicação dos parâmetros do Android com a aplicação.

O Quadro 27 apresenta trecho de código da classe `Preferencia`, alguns métodos foram abstraídos pois só mudam o tipo em relação aos que são mostrados.

```
public class Preferencias {

	// (...)
	// Armazena uma String, item é nome do atributo, e pref é o valor
	public void putString(String item, String pref){
		this.editor = preferences.edit();
		editor.putString(MODO_PREF + item, pref);
		commit();
	}
	// (...)
	// Retorna uma String, item é nome do atributo, e defValue é o
	// valor padrão caso não tenha sido atribuído anteriormente
	public String getString(String item, String defValue){
		return preferences.getString(MODO_PREF + item, defValue);
	}
	// (...)
	// Faz o commit e salva as preferencias
	public void commit(){
		editor.commit();
	}

}
```

Quadro 27 - Trecho de código da classe `Preferencias`

3.4.7 Cálculo de Frames Por Segundo

Calcular Frames Por Segundo (FPS) significa obter o número de quadros registrados, ou processados, ou exibidos por um dispositivo por uma unidade de tempo qualquer. Muito utilizado como instrumento de comparação para medir o desempenho em aplicações gráficas.

Para o cálculo de FPS na aplicação de simulação de partículas, foi implementada uma classe (`CalcFPS`) para armazenar e atualizar seu valor. O método utilizado para calcular o FPS é apresentado no Quadro 28.

```
    (...)
    public void calculaFPS() {
        long now = System.currentTimeMillis();
        if (this.ultimoTempoDesenho != 0) {
            int time = (int) (now - this.ultimoTempoDesenho);
            this.desenhoFrameTempo += time;
            this.desenhoFrameColetado++;
            if (this.desenhoFrameColetado == 10) {
                this.fps = (int) (10000 / this.desenhoFrameTempo);
                this.desenhoFrameTempo = 0;
                this.desenhoFrameColetado = 0;
        }}
        this.ultimoTempoDesenho = now;
    }
```

Quadro 28 - Método `calculaFPS()` da classe `CalcFPS`

3.4.8 Desenho de texto na OpenGL

Foram incluídas no trabalho, as classes fornecidas pela referência do Google (2007) e adaptadas a necessidade.

A classe `LabelMaker` gerencia os textos dentro de um objeto `Canvas`. Para fazer o desenho em OpenGL ES, ela cria texturas para cada texto e desenha um objeto `GL10`. Seus objetos tiveram seu escopo alterado para `package br.furb.sp.string`. O método `draw(GL10, Float, Float, int)` utilizado para desenho é apresentado no Quadro 29.

```
    /**
     * Draw a given label at a given x,y position, expressed in pixels,
     *   with the
     * lower-left-hand-corner of the view being (0,0).
     *
     * @param gl
     * @param x
     * @param y
     * @param labelID
     */
    public void draw(GL10 gl, float x, float y, int labelID) {
        checkState(STATE_DRAWING, STATE_DRAWING);
        Label label = mLabels.get(labelID);
        gl.glEnable(GL10.GL_TEXTURE_2D);
        ((GL11)gl).glTexParameteriv(GL10.GL_TEXTURE_2D,
                GL11Ext.GL_TEXTURE_CROP_RECT_OES, label.mCrop, 0);
        ((GL11Ext)gl).glDrawTexiOES((int) x, (int) y, 0,
                (int) label.width, (int) label.height);
    }
```

Quadro 29 - Método `draw(GL10, Float, Float, int)` da classe `LabelMaker`

Para simplificar sua utilização, foi desenvolvida a classe `DrawSTR`. No construtor é preciso informar o tamanho em pixels da tela do dispositivo. Em seguida é preciso iniciar os componentes através do método `initFPS(GL10, Paint)` informando a instancia da `GL10` e um objeto `Paint`, onde será desenhado o texto. Neste método já são instanciadas componentes para os caracteres: `F`, `P`, `S`, os números de 0-9 e um espaço em vazio para representar a espaço entre o texto e o número do FPS. Todos esses componentes são adicionados no `LabelMaker` e retornam uma referência para o elemento. Essa referência é guardada dentro do `DrawSTR` em forma de *array*, também é armazenada a largura e altura de cada elemento inserido para facilitar no desenho posteriormente. Para realizar o desenho na tela do dispositivo, utiliza-se o método `drawFPS(GL10, String)`, o parâmetro `String` é o valor do FPS convertido em texto. Trechos do método `initFPS` e `drawFPS` são apresentados no Quadro 30, parte do código foi abstraída pois é apenas uma repetição do código que é exibido, apenas se altera o caractere informado.

```
public void initFPS(final GL10 gl10, final Paint paint) {
   this.idTexturaFPS = new int[14];
   this.idTexturaW = new int[14];
   this.idTexturaH = new int[14];
   this.width = 512;// Largura da Label
   this.height = 128;// Altura da Label

   this.labelMaker = new LabelMaker(false, this.width, this.height);
   // Inicializa e configura LabelMaker para Receber os dados
   this.labelMaker.initialize(gl10);
   this.labelMaker.beginAdding(gl10);
   String caracter = "";
   int idLBSTR = 0;
   // Percorre os Decimais de 0 a 9 e adiciona no LabelMaker
   for (int cont = 0; cont < 10; cont++) {
      caracter = String.valueOf(cont);
      idLBSTR = this.labelMaker.add(gl10, caracter, paint);
      this.idTexturaFPS[cont] = idLBSTR;
      this.idTexturaW[cont] = (int) Math.ceil(this.labelMaker
           .getWidth(idLBSTR));
      this.idTexturaH[cont] = (int) Math.ceil(this.labelMaker
           .getHeight(idLBSTR));
   }
   // Armazena também os caracteres F P S
   caracter = String.valueOf("F");
   idLBSTR = this.labelMaker.add(gl10, caracter, paint);
   this.idTexturaFPS[10] = idLBSTR;
   this.idTexturaW[10] = (int) Math.ceil(this.labelMaker
        .getWidth(idLBSTR));
   this.idTexturaH[10] = (int) Math.ceil(this.labelMaker
        .getHeight(idLBSTR));
   (...)
   // Configura o LaberMaker para encerrar a recepcao
   this.labelMaker.endAdding(gl10);
}

public void drawFPS(final GL10 gl, final String numFPS) {
   // Posicoes que serao desenhadas
   int  x = this.widthWorld - 90,
        y = this.heightWorld - 75;
   int codigoSTR = 10;
   // Inicia Desenho dos caracteres
   this.labelMaker.beginDrawing(gl, this.width, this.height);
   // primeiro os caracteres F P S
   // F
   this.labelMaker.draw(gl, x, y, this.idTexturaFPS[codigoSTR]);
   x += this.idTexturaW[codigoSTR];
   codigoSTR++;
   (...)
   // Desenha o valor do FPS
   codigoSTR = Integer.parseInt((String) numFPS.subSequence(0, 1));
   this.labelMaker.draw(gl, x, y, this.idTexturaFPS[codigoSTR]);
   x += this.idTexturaW[codigoSTR];
   // Se o valor for de 2 digitos
   if (numFPS.trim().length() > 1){
      codigoSTR = Integer.parseInt((String) numFPS.subSequence(1, 2));
      this.labelMaker.draw(gl, x, y, this.idTexturaFPS[codigoSTR]);
   }
   this.labelMaker.endDrawing(gl);
}
```

Quadro 30 - Trechos de código da classe `DrawSTR`

3.4.9 Problemas encontrados

Entre os problemas encontrados durante o desenvolvimento deste trabalho, três se destacam: tempo em milissegundos com funcionalidade de pausar simulação; desenho de texto em OpenGL ES; os valores dos parâmetros para cálculos do comportamento das partículas.

Na implementação do botão pause e continue dos simuladores, foi adotada a ação a partir dos métodos `onPause()` e `onResume()` da *Activity*. Verificou-se que após executar o `onResume()`, a simulação não continuava do ponto em que ela havia congelado na tela, ela era desenhada muito a frente, como se o tempo não tivesse parado, e era exatamente este o problema. Quando se utiliza como referência o tempo atual para realizar um cálculo, esse fator precisa ser levado em consideração. Para se calcular o `frameRate` no motor de partículas, utiliza-se o instante atual através do método `System.currentTimeMillis()` que retorna o tempo atual em milissegundos. Então, quando se pausa e continua, o tempo não para. Para resolver este problema, foram reimplementados os métodos `onPause()` e `onResume()` da *Activity* para que quando fossem acionados, enviassem o comando `pararSimulacao()` para o motor de partículas alterar seu estado para *Pausado*. No motor de partículas o método `update()` foi alterado para que, mesmo quando estivesse com o estado *Pausado*, continuasse atualizando o atributo `Float tempoAtual`. Para quando retornar a execução, a diferença entre o ultimo tempo calculado e o tempo atual (`frameRate`) não seja prejudicada.

Para o desenho de OpenGL ES foi implementado a classe `DrawSTR` que converte os textos em objetos `Canvas` e os projeta na `OpenGL ES`, utilizando das classes disponibilizadas na referência do Google (2007).

O principal problema foi encontrar os valores para parametrizar os cálculos de física dentro dos métodos da simulação das partículas, sendo que nenhum dos valores das referências pesquisadas obteve resultado positivo. Várias referências fixavam os valores conforme a sua aplicação de simulação, fazendo que, quando fossem reimplementados em outro ambiente, não trouxessem os mesmos resultados, como exemplo, o Quadro 26, no trecho onde ocorre a dispersão das partículas, teve que ser inserido valores que se adequem a simulação e tragam um resultado coerente com o mundo real.

3.4.10 Operacionalidade da aplicação

A operacionalidade da aplicação é apresentada na forma de funcionalidades sendo representados através de imagens. Serão apresentados imagens tanto da aplicação funcionando no simulador quanto no dispositivo Milestone 3, ambos utilizados durante o desenvolvimento do trabalho.

3.4.10.1 Tela inicial

A aplicação foi implementada de forma a possuir uma tela inicial em forma de lista que permite escolher qual simulação deseja executar, ou se deseja abrir a tela de informações sobre autoria do trabalho. No momento que a aplicação é inicia é aberta a lista de menu que possui três opções conforme a Figura 25.

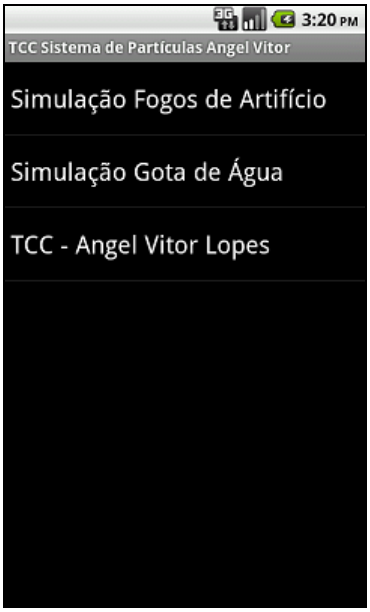


Figura 25 - Tela inicial da aplicação com a lista de opções

Selecionando a primeira opção acessa a tela de simulação de fogos de artifício, a segunda opção permite acesso a tela de simulação de gota de água e a terceira opção exibe informações sobre autoria do trabalho, conforme mostrado na Figura 26.

Figura 26 - Tela de informações do autor

3.4.10.2 Simulação de fogos de artifício

Ao acessar a tela de simulação dos fogos de artifício, é imediatamente iniciada a simulação, conforme mostra na Figura 27.

A origem das partículas sempre é na área inferior da tela, no centro.

Conforme ilustra a Figura 28, quando pressionado o botão *menu* é exibido três opções:

a) `Pause/Play`: quando em execução, o simulador entra em estado *Pausado*, pelo contrário entra em estado *Executando*;
b) `Reiniciar`: o simulador reinicia a simulação;
c) `Configurar`: abre a tela de configurações dos parâmetros.

A tela de configurações é apresentada na Figura 29. Esta tela concentra todos os parâmetros editáveis para a simulação, tais como:

a) `quantidade de partículas`: define a quantidade inicial de partículas que serão simuladas;
b) `qtd. partículas explode`: quantidade de subpartículas que serão criadas quando o fogo de artifício explodir, acrescentado de uma função de distribuição normalizada no intervalo de [1,20];
c) `gravidade`: a gravidade possui limite fixo para seleção com valor de -5.0 para 15.0, quanto menor o número, menor será a incidência da gravidade, quando o valor se tornar negativo, as partículas não cairão mais;

d) `velocidade`: esse valor representa a velocidade inicial em que as partículas serão criadas quanto maior o número, mais alto e mais rápido as partículas serão lançadas;

e) `fator do tempo`: define a velocidade em que o tempo percorre na simulação, reduzindo esse número a aplicação executa mais rapidamente, quanto maior, mais lento será a simulação.

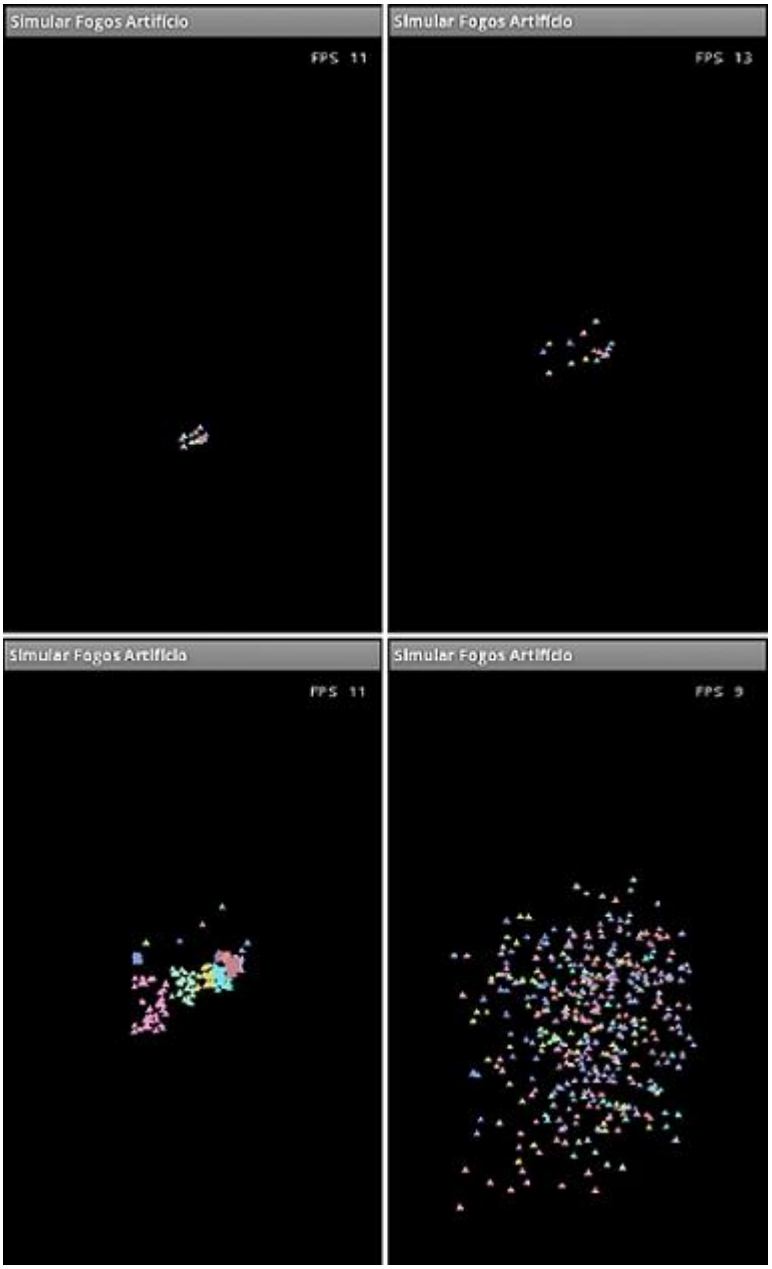


Figura 27 - Simulador de fogos de artifício em execução

A opção de `reset configurações`, apresentada na Figura 29 executa um método que redefine todos os parâmetros alteráveis para valores padrão. Esses valores padrão ajustam a simulação para que obtenha um desempenho de FPS normal.

Figura 28 - Menu do simulador de fogos de artifício

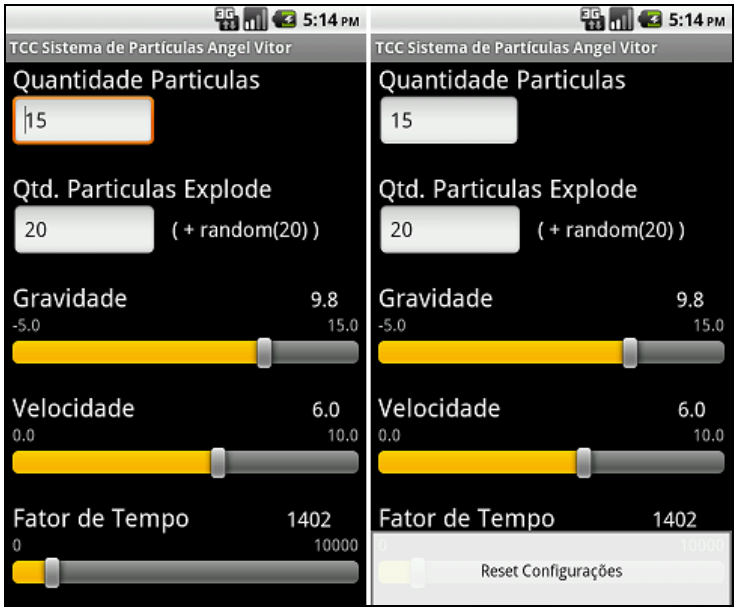


Figura 29 - Tela de configuração e menu dos parâmetros da simulação de fogos de artifício

3.4.10.3 Simulação de gotas de água

Semelhante ao simulador de fogos de artifício o simulador de gotas de água, apresentado na Figura 30, possui as mesmas funcionalidades, desde a tela de menu na simulação (Figura 31), até a tela de configurações de parâmetros e menu de parâmetros (Figura 32).

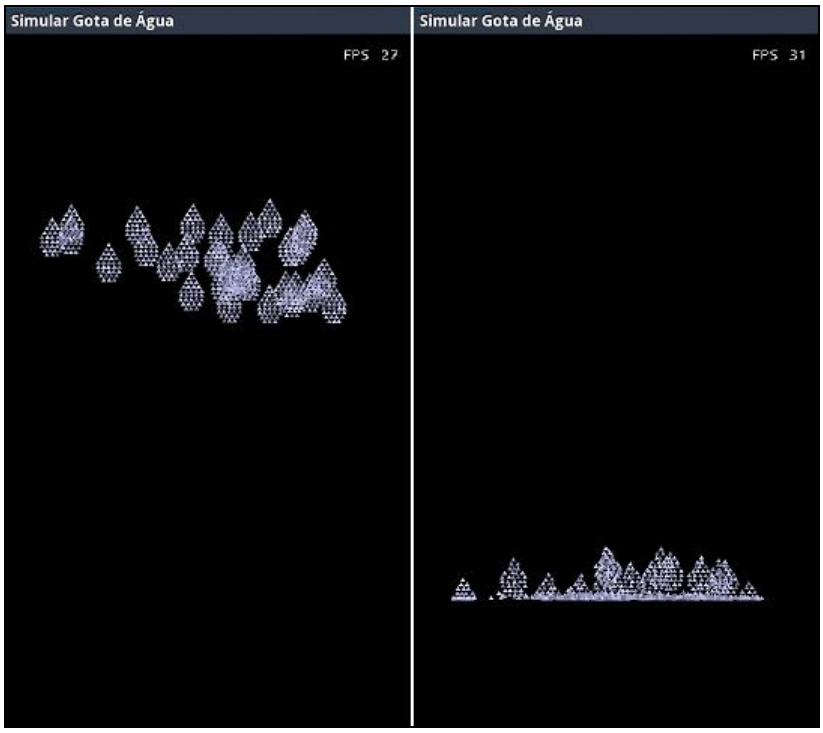


Figura 30 - Tela de simulação de gotas de água em execução

A diferença entre elas está no conteúdo dos parâmetros, apresentado na Figura 32. Para simulação de gotas de água foi priorizado os parâmetros:

a) `quantidade de partículas gotas`: define a quantidade de partículas que serão simuladas;
b) `qtd. subpartículas gotas`: quantidade de subpartículas que as gotas possuem;
c) `gravidade`: semelhante aos fogos de artifício, a gravidade possui limite fixo para seleção com valor de -5.0 para 15.0;
d) `resistência do ar`: esse valor representa o valor da resistência do ar, com limite fixo de 0 a 1.0, quando maior o número, mais próximas as partículas ficarão uma das outras, quanto menor, maior a chance delas dispersarem;
e) `fator do tempo`: também semelhante aos fogos de artifício, define a velocidade em que o tempo percorre na simulação, reduzindo esse número a aplicação executa mais rapidamente, quanto maior, mais lento será a simulação.

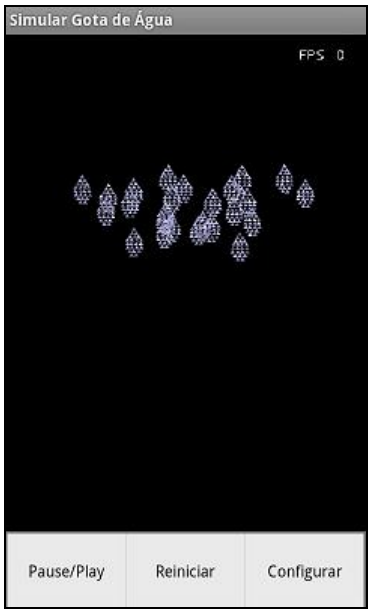


Figura 31 - Menu da tela de simulação de gotas de água

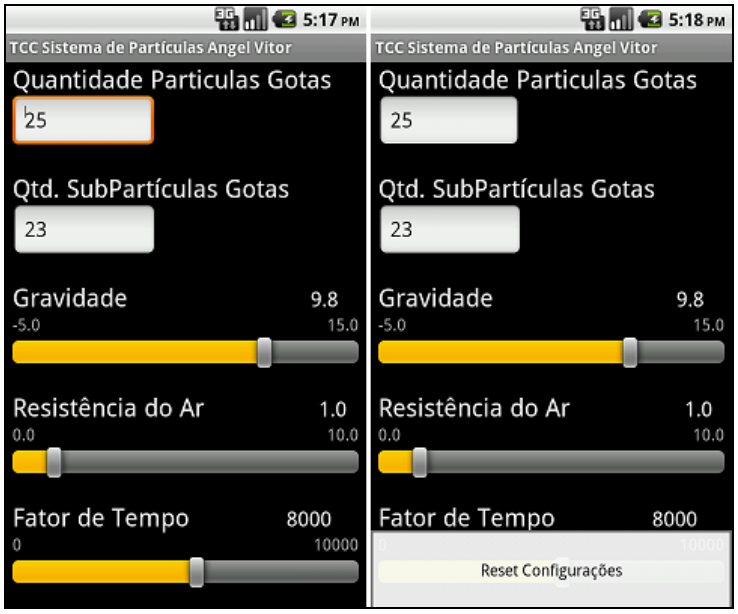


Figura 32 - Tela de configuração e menu dos parâmetros da simulação de gotas de água

3.4.10.4 Alteração de parâmetros em tempo de execução

Como requisito funcional, foi apontado a possibilidade de implementar a troca de parâmetros em tempo de execução. Esta funcionalidade foi disponibilizada para dois parâmetros nas duas simulações, `gravidade` e `fator de tempo`.

A Figura 33 apresenta o funcionamento desse método de troca de parâmetros. Como forma de ilustrar o método, foi utilizado círculos representando o toque na tela e setas indicando a direção em que deve ser deslizado o dedo. O funcionamento é o seguinte:

a) movimento horizontal para a direita: diminui o fator de tempo, fazendo aumentar a velocidade do tempo;
b) movimento horizontal para a esquerda: aumenta o fator de tempo, fazendo diminuir a velocidade o tempo;
c) movimento vertical para cima: diminui a gravidade, fazendo as partículas ficarem mais tempo no ar;
d) movimento vertical para baixo: aumenta a gravidade, fazendo as partículas caírem mais rapidamente.

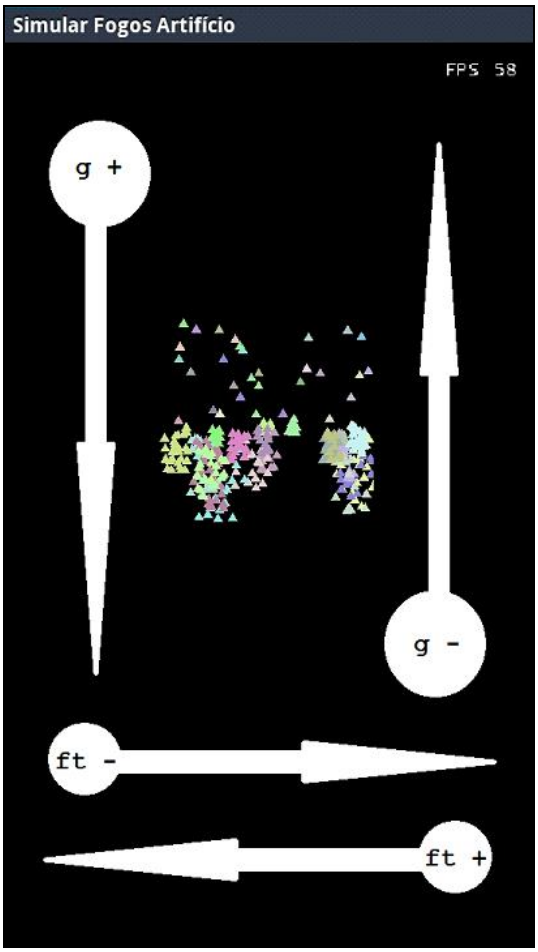


Figura 33 - Funcionamento do toque na tela que altera parâmetros em tempo de execução

3.5 RESULTADOS E DISCUSSÃO

Primeiramente, foi proposto como objetivo específico a criação de uma biblioteca de suporte a sistemas de partículas. No entanto, foi verificado que as característica de *framework* se adequariam por inteiro com o objetivo da proposta.

Um *framework* apresenta uma solução para um conjunto de problemas semelhantes, com classes e interfaces que decompõe o problema. As associações e o fluxo de informações entre essas classes são definidos pelo próprio *framework*, sendo que ele dita a arquitetura do sistema. Para isso, o *framework* deve disponibilizar funcionalidades abstratas, a serem completadas, essas funcionalidades definirão as características de cada aplicação (SAUVÉ, 2010).

Diante do exposto, pode-se verificar que o presente trabalho atendeu parte das características de um framework. A partir da identificação de um conjunto de necessidades para as simulações de partículas foi desenvolvido um conjunto de classes objetivando atender a essas necessidades. As classes desenvolvidas apresentam associações e fluxo de informações definidos e apresentam funcionalidades parcialmente abstratas. Neste último quesito, das funcionalidades abstratas, muitos métodos não puderam ser generalizados pois o algoritmo para cada simulação de comportamento é diferente. Muitas modelagens de partículas precisam ser implementadas de forma específica, desde a criação de partículas, passando pela atualização do comportamento, até o ponto de extinção da mesma.

Dentre as dificuldades e problemas encontrados, descritas na seção 3.4.9, os valores fixos nos cálculos de física foi o mais impactante, sendo necessário muitos testes e pesquisas procurando o melhor resultado. A referência de Thornton e Marion (2011) trás muitos conceitos e fórmulas para se chegar aos cálculos precisos da dinâmica das partículas, mas quando aplicados na computação, é preciso adaptar e abstrair o essencial e funcional.

Outra dificuldade, que demandou pesquisa e muitos testes repetitivos até ocorrer a adequação ao trabalho, foi o OpenGL ES para Android não oferecer suporte ao desenho de texto. Embora o problema já havia sido documentado no trabalho de Vasselai (2011), descrito na seção 2.3.1, foram encontrados problemas na implementação.

Um dos fatores que resultou em muita pesquisa foi a escassez de material bibliográfico para sistemas de partículas, assim como nos métodos utilizados para a resolução de problemas.

O simulador do Android é um limitador de desempenho e de funcionalidades. Dentre as diversas limitações, a que foi mais impactante é a falta de suporte a OpenGL ES 2.0, já descrita na referência do Google (2012g), onde é recomendada a utilização de dispositivos físicos com Android acima da API *Level* 8 (Android 2.2) para executar testes com a OpenGL ES 2.0. Outro item referente as dificuldades no simulador do Android é o baixo desempenho no processamento das tarefas, que dificultou muito em encontrar as fórmulas e valores adequadas na simulação física.

3.5.1 Resultados obtidos nos testes de desempenho

O método utilizado para medir o desempenho nas simulações de partículas foi a verificação da taxa de Frames Por Segundo (FPS) em relação à quantidade de partículas em simulação. A medição de FPS foi realizada nos dois motores de partículas. As medições foram levantadas a partir do dispositivo móvel Milestone 3.

Identificada a necessidade de realização precisa de medição, foi implementada uma rotina para gravação de um arquivo texto a partir do botão `pause/play`. Seu funcionamento é ativado quando se pressiona o botão `pause/play`. Em seguida motor de partículas verifica todas as partículas ativas na aplicação e grava a informação no arquivo texto, por fim é realizado o cálculo de FPS e seu valor escrito no arquivo também.

Pela diferença na implementação dos efeitos entre os dois motores, precisam ser considerados alguns fatores. Na simulação de fogos de artifício existe uma variação da quantidade de subpartículas que não pode ser quantificada precisamente, a causa é a quantidade de subpartículas ser atribuída por uma função de distribuição normalizada para o intervalo [0,20]. Então a quantidade de subpartículas de fogos de artifício não podem ser previamente quantificadas com precisão. Outro fator importante nos fogos de artifício é o fato das subpartículas serem ativadas após a partícula principal explodir então no decorrer da simulação, entre pequenos intervalos de tempo pode-se ter poucas ou muitas partículas em animação.

O primeiro teste para a simulação de fogos de artifício considera a mesma quantidade de partículas pai para toda a amostragem, somente variando a quantidade de subpartículas. Os resultados da primeira medição podem ser conferidos na Tabela 1.

Tabela 1 - Primeira medição da simulação de fogos de artifício

Quantidade de partículas	Quantidade de subpartículas	Total de partículas + Aleatório[1,20]	Média FPS
15	20	444	58
15	30	612	52
15	40	720	44
15	50	893	36
15	60	1049	31
15	70	1147	27
15	80	1334	25
15	90	1519	21
15	100	1667	20
15	150	2443	15

A partir primeira medição de fogos de artifício foi possível gerar o gráfico da Figura 34. É perceptível que a quantidade total de partículas interfere no desempenho da simulação.

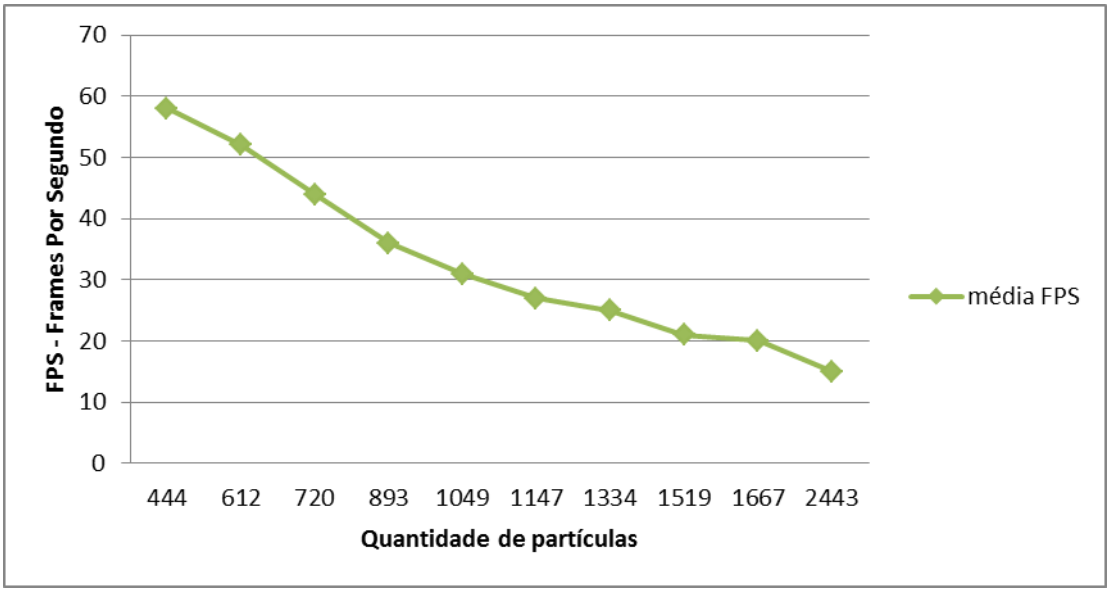


Figura 34 - Gráfico da primeira medição de FPS da simulação de fogos de artifício

Para obter mais dados para a comparação, a simulação foi submetida novamente ao teste de desempenho, mantendo os mesmos valores de subpartículas, mas, a quantidade de partículas pai foi aumentada.

O segundo teste teve uma quantidade relativamente maior de partículas simulada, conforme pode ser conferido na Tabela 2, por outro lado, o desempenho foi inferior ao obtido no primeiro teste.

Tabela 2 - Segunda medição da simulação de fogos de artifício

Quantidade de partículas	Quantidade de subpartículas	Total de partículas + Aleatório[1,20]	Média FPS
15	20	447	59
20	30	805	43
25	40	1186	31
30	50	1209	27
35	60	1779	24
40	70	3215	22
45	80	4017	13
50	90	4947	9
55	100	6004	7
60	150	9535	4

O gráfico da Figura 35 ilustra os dados apresentados na segunda medição.

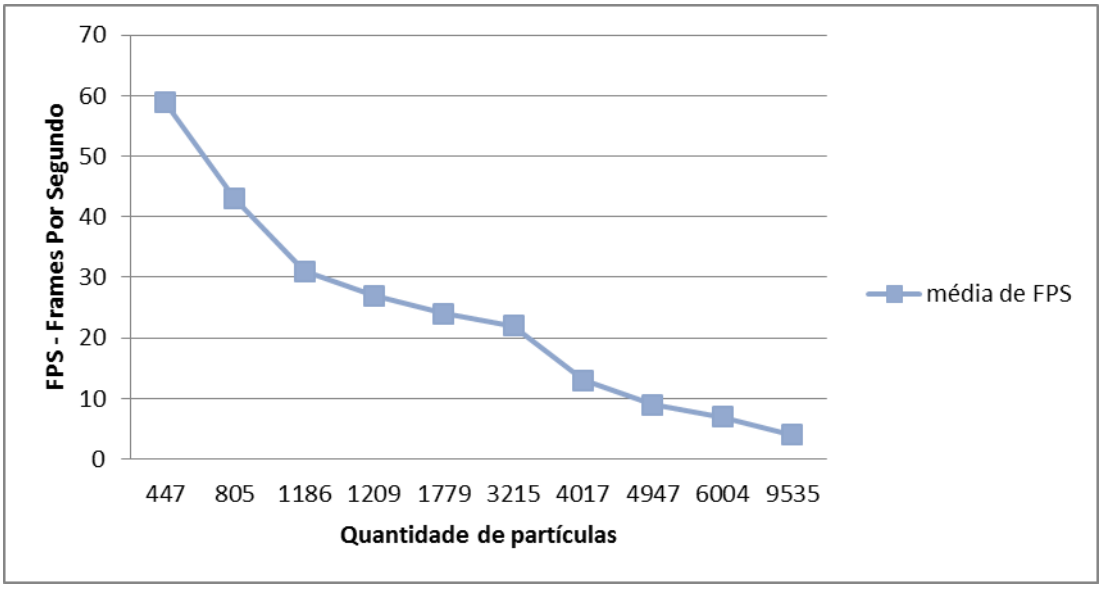


Figura 35 - Gráfico da segunda medição de FPS da simulação de fogos de artifício

Para a simulação de gotas de água, foi utilizada a mesma metodologia. A Tabela 3 apresenta os resultados obtidos no teste de desempenho e observa-se que o resultado assemelha-se ao obtido no primeiro teste da simulação de fogos de artifício.

Tabela 3 - Medição da simulação de gotas de água

Quantidade de partículas	Quantidade de subpartículas	Total de partículas	Média FPS
15	20	300	59
15	30	450	59
15	40	600	53
15	50	750	47
15	60	900	36
15	70	1050	30
15	80	1200	27
15	90	1350	24
15	100	1500	20
15	150	2250	15

O gráfico da Figura 36 ilustra os resultados obtidos no teste da simulação de gotas de água.

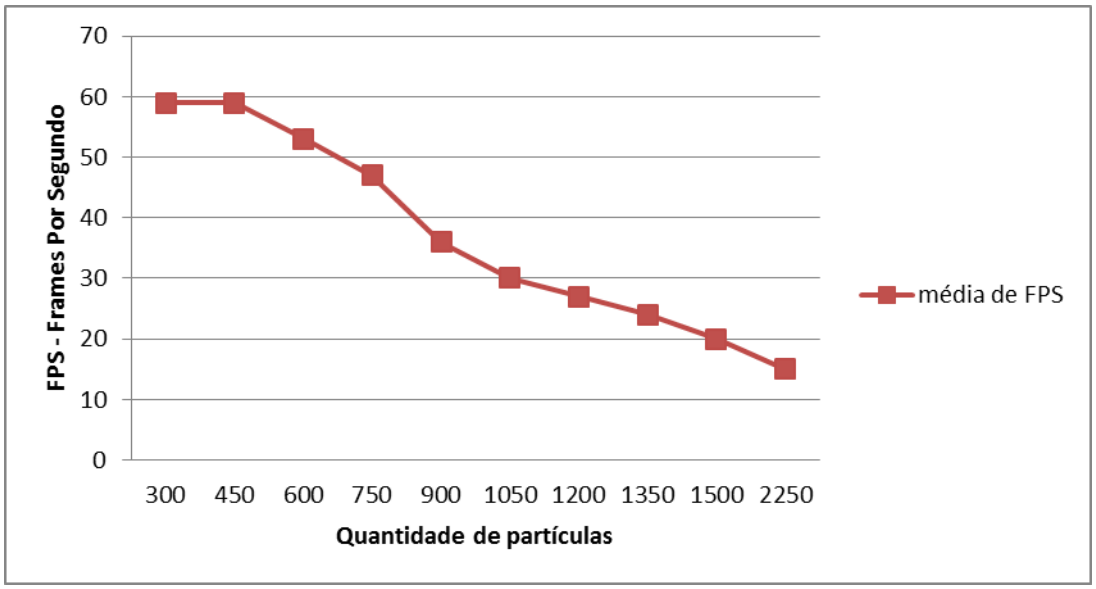


Figura 36 - Gráfico da medição de FPS da simulação de gotas de água

Analisando os três testes, pode-se afirmar que a variância da média de FPS é maior nos primeiros itens das amostragens e menor nos itens com muitas partículas. Tal comportamento deve-se ao fato de que quando se aumenta a quantidade de partículas em simulação, é necessário realizar mais cálculos físicos, fazendo aumentar os intervalos em que é feito o desenho na tela.

Confrontando os resultados com os obtidos nos trabalhos correlatos, observa-se que o desempenho da aplicação sempre é uma questão relevante e muito discutida.

Pode-se concluir então que quantidades maiores de partículas afetam o desempenho do dispositivo. Então é necessário levar em consideração o tempo entre cálculos físicos e desenho das partículas na tela.

4 CONCLUSÕES

O presente trabalho apresentou um *framework* que atende a uma infraestrutura de suporte para o desenvolvimento de sistema de partículas e dois estudos de casos, um simulador de fogos de artifício, e um simulador de gotas de água.

A plataforma Android mostrou possuir grande potencial para desenvolvimento de aplicativos. Contendo uma API bastante robusta e ferramentas de desenvolvimento que permitem inclusive depurar código fonte dentro do dispositivo.

Os pontos fracos encontrados do Android SDK estão no simulador Android, que não oferece suporte para OpenGL ES 2.0 e apresenta um desempenho muito baixo.

Um dos principais desafios para o desenvolvimento deste trabalho foi a escassez de material bibliográfico sobre sistemas de partículas e seus métodos de solução de problemas. Muitos documentos pesquisados traziam uma descrição para resolução de algum problema especifico que não podia ser aplicado a este trabalho. Foi necessário um esforço de pesquisa para juntar equações e valores que se adequassem ao sistema de partículas proposto. A simulação segue as leis físicas do mundo real, levando em consideração a gravidade, resistência do ar e velocidade de partículas.

O framework desenvolvido neste trabalho apresentou funcionalidades semelhantes às dos trabalhos correlatos. Sendo destacada a utilização de física e desenvolvimento de um motor de partículas e a utilização do ambiente de desenvolvimento Android.

Por fim, a plataforma Android mostrou-se capaz de modelar elementos com sistema de partículas. As limitações na quantidade de partículas simuladas por taxa de FPS também são encontradas em *hardwares* comuns, em escalas diferentes, então pode-se considerar o Android um dispositivo cada vez mais apto a receber aplicações que necessitem de alto desempenho. Este trabalho definiu um *framework* básico para ser trabalhado em sistema de partículas, demonstrou seu funcionamento através de dois estudos de casos, e reuniu fundamentação para fortalecer a ligação entre a plataforma Android e modelagem por sistema de partículas.

4.1 EXTENSÕES

A seguir são apresentados alguns pontos que podem ser agregados ou melhorados, tanto no *framework* do motor de partículas, quando nos motores dos estudos de casos. Como sugestões, seguem:

a) acrescentar suporte para modelagem de fluídos no *framework*, com adição de outros atributos nas partículas e alterando a forma de armazenamento no motor de partículas, e também adicionar o efeito de fusão em fluídos;
b) acrescentar efeitos na modelagem de fogos de artifício, procurando trazer mais realismo a simulação;
c) portar o mesmo *framework* para outras plataformas, como iOS da Apple ou Windows Phone;
d) acrescentar efeitos na modelagem de gotas de água, como arco-íris refletindo nas gotas e aplicar efeitos de vento;
e) modelar uma simulação de partículas para representar uma nuvem de nêutrons que tem partículas girando ao redor uma partícula central, considerando atributos como massa, centro de massa e gravidade;
f) acrescentar rotação em três dimensões nas simulações, pode-se modelar rotação da câmera quando as partículas modelarem uma forma em três dimensões. Com isso pode-se expandir muitos conceitos na simulação em dispositivos móveis;
g) simular os efeitos produzidos pelos raios de luz emitidos, refletidos e transmitidos dentro de um ambiente;
h) simular um redemoinho de partículas;
i) acrescentar um sistema de colisões de partículas ao *framework*;
j) desenvolver alguns componentes da animação em C++ buscando otimizar o desempenho;
k) acrescentar mais fenômenos físicos para serem modelados.

REFERÊNCIAS BIBLIOGRÁFICAS

CEARLEY, David. **Top 10 strategic technology trends for 2012**. [S.l.], 2012. Disponível em: <http://www.gartner.com/technology/research/top-10-technology-trends/>. Acesso em: 22 maio 2012.

DALVIKVM. **Dalvik virtual machine**. [S.l.], 2008. Disponível em: <http://www.dalvikvm.com>. Acesso em: 30 maio 2012.

ECLIPSE FOUNDATION. Ottawa, 2012. Disponível em: <http://www.eclipse.org/helios/>. Acesso em: 01 fev. 2012.

GOOGLE. **What is Android?** [S.l.], 2012a. Disponível em: <http://developer.android.com/guide/basics/what-is-android.html>. Acesso em: 30 maio 2012.

_____. **OpenGL**. [S.l.], 2012b. Disponível em: <http://developer.android.com/guide/topics/graphics/opengl.html>. Acesso em: 30 maio 2012.

_____. **Application fundamentals**. [S.l.], 2012c. Disponível em: <http://developer.android.com/guide/topics/fundamentals.html>. Acesso em: 31 maio 2012.

_____. **Designing for performance**. [S.l.], 2012d. Disponível em: <http://developer.android.com/guide/practices/design/performance.html>. Acesso em: 31 maio 2012.

_____. **Services**. [S.l.], 2012e. Disponível em: <http://developer.android.com/guide/topics/fundamentals/services.html>. Acesso em: 31 maio 2012.

_____. **Download the Android SDK**. [S.l.], 2012f. Disponível em: <http://developer.android.com/sdk/index.html>. Acesso em: 01 fev. 2012.

_____. **OpenGL ES 2.0**. [S.l.], 2012g. Disponível em: <http://developer.android.com/resources/tutorials/opengl/opengl-es20.html>. Acesso em: 05 mar. 2012.

_____. **LabelMaker**. [S.l.], 2007. Disponível em: <http://developer.android.com/resources/samples/ApiDemos/src/com/example/android/apis/graphics/spritetext/LabelMaker.html>. Acesso em: 02 fev. 2012.

HERMAN, Gary; REDKEY, David. **A particle system representation of candle wax**. Stanford, 1995. Disponível em: <http://www-graphics.stanford.edu/courses/cs348c-95-fall/projects/redkey_herman/writeup.html>. Acesso em: 28 maio 2012.

KHRONOS. **Khronos OpenGL ES API registry**. Oregon, 2010. Disponível em: <http://www.khronos.org/registry/gles>. Acesso em: 30 maio 2012.

LECHETA, Ricardo R. **Google Android**: aprenda a criar aplicações para dispositivos móveis com o Android SDK. 2. ed. São Paulo: Novatec, 2010.

LOKE, Teng-See. Rendering fireworks displays. **IEEE Computer graphics and applications**, New York, v. 12, n. 3, p. 33-43, 1992.

LUEBKE, David; HUMPHREYS, Greg. Hows gpus works. **Computer**, Washington, v. 40, n. 2, p. 126-130, Feb. 2007.

MOTOROLA, Inc. **Android applications for java me developers**. [S.l.], 2009. Disponível em: <http://developer.motorola.com/docs/android_applications_for_java_me_developers>. Acesso em: 31 maio 2012.

MUELLER, Victor A. **Simulação física de corpos rígidos em 3D**. 2010. 67 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau. Disponível em: <http://www.bc.furb.br/docs/MO/2011/346089_1_1.pdf>. Acesso em: 05 set. 2011.

NEWTON, Isaac. **Philosophiae naturalis principia mathematica**. London, 1687. Disponível em: <http://astro.if.ufrgs.br/newton/principia.pdf>. Acesso em: 28 maio 2012.

OPEN HANDSET ALLIANCE. **Alliance**. [S.l.], 2012. Disponível em: <http://www.openhandsetalliance.com/oha_members.html>. Acesso em: 29 maio 2012.

PEREIRA, Lucas; HSU, David. **Modeling lava flows with a particle system**. Stanford, 1995. Disponível em: <http://www-graphics.stanford.edu/courses/cs348c-95-fall/projects/pereira_hsu/implementation.html>. Acesso em: 28 maio 2012.

PETTEY, Christy; GOASDUFF, Laurence. **Gartner says sales of mobile devices in second quarter of 2011**. Eghan, 2011. Disponível em: <http://www.gartner.com/it/page.jsp?id=1764714>. Acesso em: 07 set. 2011.

REEVES, William T. Particle systems: a technique for modeling a class of fuzzy objects. **ACM Transactions on Graphics**, New York, v. 2, n. 2, p. 91-108, Apr. 1983.

SANTOS, Christiano L. **Uma introdução a sistema de partículas**. [S.l.], 2008. Disponível em: <http://www.blog.programadoresdejogos.com/2008/02/uma-introducao-asistemas-de-particulas/>. Acesso em: 11 set. 2011.

SAUVÉ, Jacques. **Frameworks**. Campina Grande, [2010]. Disponível em: <http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/frame/oque.htm>. Acesso em: 31 maio 2012.

SCHUYTEMA, Paul. **Design de games**: uma abordagem prática. Tradução de Cláudia Mello Belhassof. São Paulo: Cengage Learning, 2008.

SQLITE. [S.l.], 2012. Disponível em: <http://www.sqlite.org/about.html>. Acesso em: 31 maio 2012.

STAR TREK II: the wrath of Khan. Direção: Nicholas Meyer. Produção: Robert Sallin. Roteiro: Jack B. Sowards; Nicholas Meyer. Estados Unidos: Paramount Pictures, 1982. 116min. son. color.

STEIGLEDER, Mauro. **Integração de sistemas de partículas com detecção de colisões em ambientes de Ray Tracing**. 1997. 104 f. Dissertação (Mestrado em Ciência da Computação) – Universidade Federal do Rio Grande do Sul, Porto Alegre.

THORNTON, Stephen T.; MARION, Jerry B. **Dinâmica clássica de partículas e sistemas**. Tradução de All Tasks. Revisão técnica de Fábio Raia. 5. ed. São Paulo: Cengage Learning, 2011.

TULL, Alice; TSOI, Helios. **Controlable rain, rainbown and oil films**. Stanford, 1995. Disponível em: <http://www-graphics.stanford.edu/courses/cs348c-95-fall/projects/tull_tsoi/writeup.html>. Acesso em 26 maio 2012.

VASSELAI, Gabriela T. **Um estudo sobre realidade aumentada para a plataforma Android**. 2010. 102 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau. Disponível em: <http://www.bc.furb.br/docs/MO/2011/346536_1_1.pdf>. Acesso em: 06 set. 2011.

WATSON, Richard T. et al. UCommerce: extending the universe of marketing. **Journal of the Academy of Marketing Science**. [S.l.], v. 30, p. 329-343, Oct. 2002.

WEISER, Mark. The computer for the 21st century. **Scientific American**, [S.l.], v. 265, n. 3, p. 94-104, Sept. 1991.