Curso de Ciência da Computação
Universidade Estadual de Mato Grosso do Sul

Simulação de Partículas Utilizando WebGL

Gabriel Dantas Sigolo

Msc. Diogo Fernando Trevisan (Orientador)

Dourados - MS
2016

Curso de Ciência da Computação
Universidade Estadual de Mato Grosso do Sul

Simulação de Partículas Utilizando WebGL

Gabriel Dantas Sigolo
Novembro de 2016

Banca Examinadora:

Prof. MSc. Diogo Fernando Trevisan (Orientador)
Computação - UEMS

Prof. MSc. Jéssica Bassani de Oliveira
Computação - UEMS

Prof. Dr. Mercedes Rocío Gonzales Márquez
Computação - UEMS

Simulação de Partículas Utilizando WebGL

Gabriel Dantas Sigolo

Este exemplar corresponde à redação final da monografia da disciplina Projeto Final de Curso devidamente corrigida e defendida por Gabriel Dantas Sigolo e aprovada pela Banca Examinadora, como parte dos requisitos para a obtenção do título de Bacharel em Ciência da Computação.

Dourados, 29 de Setembro de 2016.

Prof Msc. Diogo Fernando Trevisan
(Orientador)

Resumo

Neste trabalho é demonstrada a implementação de um sistema de partículas utilizando WebGL. Este sistema pode ser usado para gerar efeitos como fogo, fumaça, entre outros elementos. Hoje em dia, é muito fácil conectar a internet via computador/notebook, e assim acessar páginas *web*, afim de trocar informações continuamente usuários nessa rede de escala global. A WebGL, juntamente com o HTML, pode proporcionar uma forma de incorporar aos sites maior realidade ao conteúdo *web*, isso quando envolve efeitos gráficos. Um sistema de partículas permite criar efeitos especiais e adicioná-los em aplicações gráficas. Será tratado também as características e limitações do WebGL.

Palavras-chave:WebGL, partículas, web, GLSL, HTML

Conteúdo

Lista de Figuras

Listagens

1 Introdução

1.1 Introdução

A *web* foi inventada em 1992, por Sir Tim Berners-Lee, que trabalhava na seção de Computação da Organização Europeia de Pesquisa Nuclear (MIT), quando iniciou pesquisas visando descobrir um método que possibilitasse aos cientistas do mundo inteiro compartilhar eletronicamente seus textos e pesquisas; ainda que tivesse a funcionalidade de interligar documentos [13].

Quando um computador é ligado, uma das primeiras coisas a ser feita é encontrar o browser para acessar páginas *web*. A *web* é referência em todos os aspectos, como comércio, aprendizado e informação, pois, qualquer novidade é divulgada instantaneamente pelo mundo.

O OpenGL, também surgiu na década de 90, desenvolvido na *Silicon Graphics Computer Systems*. É um software de interface baseado em hardware gráfico que permite a criação de programas interativos que produzem imagens de objetos tridimensionais. Com OpenGL é possível controlar a tecnologia em computação gráfica para produzir imagens realísticas. A OpenGL é utilizada para criar aplicações gráficas que podem acessar recursos da placa gráfica (GPU) [2].

A WebGl é a junção de duas tecnologias, a *web* e o OpenGL. Com a WebGL é possível viajar dentro de construçõ e vê-las interna e externamente. Pode-se conhecer detalhes do projeto sem precisar sair de casa. Isso vale para museus conhecidos internacionalmente, os quais disponibilizam uma viagem

na história em três dimensões, com a sensação de estar caminhando dentro do local, assim como na realidade.

Ainda não se consegue ver muitas aplicações disponibilizadas pelo fato da WebGL estar em desenvolvimento e ser relativamente nova. Pelo resultado que se pode obter com essa ferramenta, as páginas *web* tendem a ficar cada vez mais dinâmicas e interativas ao público.

1.2 Objetivos

O objetivo deste trabalho é estudar e implementar um sistema de partículas utilizando WebGL - que consiste em um código de controle escrito em JavaScript e códigos de efeitos especiais (*shader*). Também será feita uma avaliação de desempenho do sistema.

1.2.1 Objetivos Específicos

- Utilizar WebGL para criar um Sistema de Partículas.
- Criar um sistema de partículas genérico, onde a mudança de efeitos e formas sejam alteradas de maneira simples e rápida.
- Analisar o desempenho gráfico do sistema criado.

1.3 Justificativa

Os sistemas de partículas são muito utilizados em jogos e outras aplicações gráficas para gerar efeitos especiais como fumaça, fogo, entre outros elementos. Neste trabalho, objetiva-se verificar as limitações da WebGL para a implementação de sistemas de partículas e também analisar o desempenho desta, visto que tais sistemas são comumente utilizados em aplicações 3D.

1.4 Metodologia

Este trabalho foi realizado através das seguintes etapas:

- Estudo sobre WebGL.
- Estudo sobre o funcionamento de *Shaders*.
- Estudo sobre Sistemas de Partículas.
- Implementação de código para gerar o sistema de partículas.
- Testes do sistema de partículas.

Além disso, durante todas as etapas foi realizada a escrita e documentação do trabalho.

1.5 Organização do Texto

O texto do projeto está organizado em um único volume, contendo cronograma de atividades, listas de figuras e tabelas. Além deste Capítulo, o volume é organizado em outros cinco capítulos. No capítulo 2, é realizado uma introdução ao OpenGL. No capitulo 3, são apresentados os conceitos de *WebGL*. No capítulo 4, é exposto o sistema de partículas, com conceitos e características a serem trabalhadas no desenvolvimento. No capítulo 5 é apresentado o sistema implementado e, por fim, no capítulo 6 são apresentadas as conclusões do trabalho.

2 OpenGL

2.1 Introdução

O OpenGL é uma API para criação de aplicações gráficas, definida como um programa de interface para hardware gráfico. Pode ser descrita também como uma biblioteca de rotinas gráficas e de modelagem bidimensionais (2D) e tridimensionais (3D), sendo extremamente portável e rápida. Esta permite que os desenvolvedores de software criem aplicações gráficas de alta-performance e visualmente atraentes [5].

A API (*Application Programming Interface*) do OpenGL sofreu várias mudanças desde sua criação. A cada nova versão, novos recursos e funcionalidades foram adicionados[8].

2.2 Pipeline Fixa

Um conjunto de estados de processamento configurável presente em versões mais antigas do OpenGL é chamado de *Pipeline* de funções fixas. Esta fornece uma série de funcionalidades para as operações de vértices e *pixel* de manipulação que não podem ser alteradas, pois são predefinidas ou fixas.

Até o OpenGL versão 2.0, em 2004, a funcionalidade do *pipeline* de gráficos era fixa, ou seja, o *hardware* era limitado por um conjunto de operações e era impossível modificar a *pipeline* de gráficos.

O diagrama da Figura 2.1 mostra como o OpenGL trabalha com o pro-

cessamento de dados. Primeiramente, dados geométricos (vértices, linhas e polígonos) são passados através da linha de estágios para os avaliadores e, logo a seguir, as operações por vértices montam as primitivas, enquanto os dados de *pixel* (*pixels* e imagens) recebem tratamento diferenciado por parte do processo. Ambos os tipos de dados passam pelas mesmas etapas finais (rasterização e operações por fragmento), antes que os *pixels* finais sejam escritos no *buffer* de quadro [2].

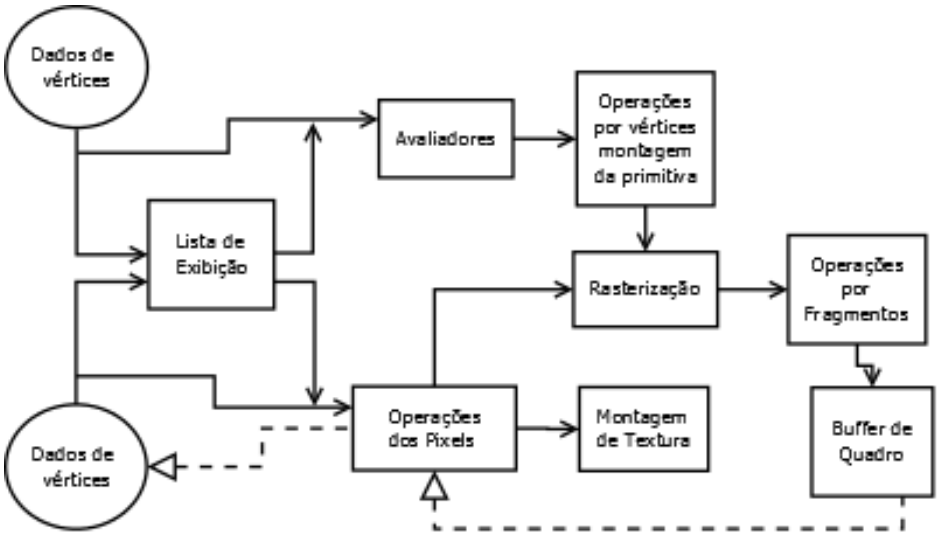

Figura 2.1: *Pipeline* OpenGL na versão 1.1

2.3 Pipeline Programável

Ainda na versão 2.0, os objetos de *Shader* foram introduzidos pela primeira vez. Isso permitiu que os desenvolvedores pudessem modificar o *pipeline* de gráficos através de programas especiais chamados de *Shaders*, esses, escritos em uma linguagem especial chamada OpenGL Shading Language (GLSL)[8]. A Figura 2.2 mostra o diagrama da *pipeline* de *rendering* da versão mais recente do OpenGL, a v.4.5.

Primeiramente, acontece a preparação dos dados do conjunto de vértices. Em seguida, nos três próximos passos, o processo onde *Shader dos vértices*

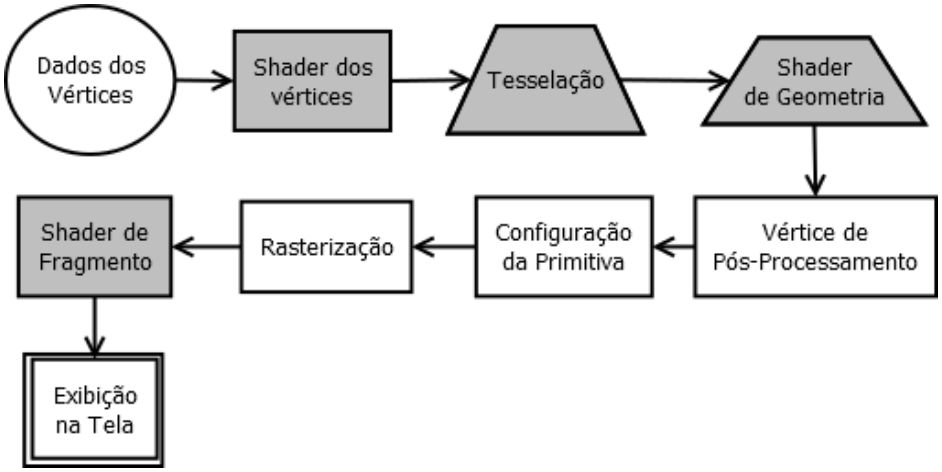

Figura 2.2: *Pipeline* OpenGL na versão 4.5. Adaptado de [9]

realiza o processamento de cada vértice para a saída; após esta etapa tem-se o estágio de tesselação das primitivas, e, por fim, em *Shader de Geometria* a saída de uma sequência de primitivas. Em um segundo momento acontece o pós-processamento do vértice, que ajusta as saídas do último estágio ou os envia para um local diferente. O processo de *Rasterização* converte o conjunto de primitivas em uma sequência de fragmentos que calcula os dados finais para um *pixel* no *buffer* de quadros de saída. No *Shader de Fragmento*, cada fragmento é processado e gera um número de saída [9].

3 WebGL

3.1 Introdução

A origem do WebGL começou há 24 anos, quando lançada a versão 1.0 do OpenGL. É uma maravilhosa e excitante tecnologia que permite criar poderosos gráficos 3D dentro de um navegador Web. Isso é alcançado utilizando uma API JavaScript que interage com a Unidade de Processamento Gráfico (GPU - *Graphics Procesment Unit*). Essa API de *rendering* 3D é derivada do OpenGL ES 2.0, mas em um contexto HTML, permitindo criar primitivas flexíveis que podem ser aplicadas em qualquer caso de uso[6].

Como o contexto da API é obtido a partir do `<canvas>` do HTML, significa que não é preciso nenhum *plugin*. O programa usa GLSL (*GL Shading Language*), que tem linguagem semelhante ao c++ e é compilado em tempo de execução. Trabalhar com WebGL requer dedicação, porém, oferece a recompensa de ser muito rápida e trazer realismo às páginas WEB[1].

3.2 HTML5

HTML é a sigla para *HiperText Markup Language*, que significa linguagem para marcação de hipertexto. Hipertexto é considerado todo conteúdo inserido em um documento para web, que tem como principal característica interligar outros documentos web, através de *links* presentes nas páginas dos sites [12].

A web foi inventada em 1992 por Sir Tim Berners-Lee. Ele trabalhava em uma pesquisa visando descobrir um método que possibilitasse aos cientistas do mundo inteiro compartilhar eletronicamente seus textos e pesquisas, e, que também tivesse a funcionalidade de interligar os documentos. Assim, já estava criada a noção de *web* e *links* [12].

Desde a invenção da web, o HTML evoluiu por sete versões, estas listadas abaixo:

- HTML
- HTML +
- HTML 2.0
- HTML 3.0
- HTML 4.0
- HTML 4.01
- HTML5 (versão atual)

Nesta última versão foi inserido o elemento *canvas* que é utilizado para gerar conteúdo 3D com WebGL. A estrutura básica do HTML5 continua sendo praticamente a mesma das versões anteriores, mas com menos código. A Listagem 3.1 mostra a estrutura básica de um documento HTML5 [3].

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4    <title></title>
5    <meta charset="utf-8">
6  </head>
7  <body>
8
9  </body>
10 </html>
```

Listagem 3.1: Estrurura básica de código HTML5

3.2.1 Elemento Canvas

Antes do HTML5, para exibir uma imagem em uma página web, a única abordagem para isso era usando a tag `<img>`, que embora seja uma ferramenta conveniente, é restrita a imagens estáticas. Ela não permite o desenho de forma dinâmica e também exibição em tempo real [7].

A introdução da tag `<canvas>` mudou tudo, agora oferecendo uma maneira apropriada para desenhar gráficos de computador dinamicamente usando JavaScript. Como os artistas usam telas para a pintura, o `<canvas>` define uma área para desenho em uma página web [7].

A Listagem 3.2 mostra como funciona a tag `<canvas>`. Ela tem alguns parâmetros essenciais para que se possa produzir desenhos em uma página.

```
1 <canvas id="example" width="400" height="400"></canvas>
```

Listagem 3.2: Criando o elemento canvas.

Para expandir um pouco mais nosso conhecimento, será criado então um ambiente para desenho através do elemento *canvas*, como demonsta a Listagem 3.3. Em seguida, o resultado da codificação é gerado e ilustrado na Figura 3.1.

```
1  <!doctype html>
2  <html>
3  <head>
4    <title>A blank canvas</title>
5    <style>
6      body{ background-color: grey; }
7      canvas{ background-color: white; }
8    </style>
9    <script>
10     document.getElementById('example');
11   </script>
12 </head>
13 <body>
14   <canvas id="example" width="400" height="400">
15     Your browser does not support the HTML5 canvas element.
```

Listagem 3.3: Exemplo de canvas

Figura 3.1: Exemplo de canvas.

A tag `<style>` é código de estilo CSS e é usada para definir a cor de fundo do corpo da página e a cor do fundo do *canvas*. O *id* dentro do elemento *canvas* é utilizado posteriormente para a manipulação no ambiente JavaScript, que a partir de então oferece seus recursos para desenhos e transformações.

3.3 JavaScript

A linguagem JavaScript foi criada pela Natscape em parceria com a Sun Microsystems, com a finalidade de fornecer um meio de adicionar interatividade a uma página web. Desenvolvida para rodar no lado do cliente, isto é, o funcionamento da linguagem depende de funcionalidades hospedadas no navegador do usuário. Isso acontece porque existe um interpretador JavaScript no navegador[11].

3.3.1 Manipulando o canvas com JavaScript

O DOM(Document Object Model) representa todos os objetos de uma página HTML. É uma linguagem neutra e independente de plataforma que permite que o conteúdo e o estilo da página possam ser atualizados depois de já processados no navegador. O DOM é acessível através do JavaScript[4].

Um dos métodos mais usados do JavaScript para acessar um nó do DOM é o **document.getElementById()**, o qual pode ser utilizado para mudar,

deletar, criar, copiar elementos HTML entre outras funções. Na Listagem 3.4 é mostrado um exemplo de como usar essa função na prática.

```
<script>
  document.getElementById('example');
</script>
```

Listagem 3.4: Exemplo de como usar o *document.getElementById()*.

3.4 GLSL

GLSL (GL Shading Language) é uma linguagem de programação usada para o desenvolvimento de *shaders* dentro da *pipeline* do OpenGL. Padronizada pela Khronos juntamente com a API OpenGL, faz parte de qualquer implementação OpenGL que seja compatível com a versão 2.0 ou posterior da API.

A OpenGL Shading Language (GLSL) 4 traz o poder e a flexibilidade para criação de programas modernos, interativos e com alta qualidade gráfica. Permite aproveitar o poder da Unidade de Processamento Gráfico(GPU), fornecendo uma linguagem simples e ao mesmo tempo uma API poderosa [14].

3.5 Componentes WebGL

Nesta seção serão abordados os componentes utilizados na GLSL: o *buffer de desenho*, *tipos primitivos* e *dados de um vértice*.

3.5.1 Buffer de Desenho

Um *buffer* é um bloco de memória que tem como objetivo armazenar dados temporários. Um buffer de cor armazena informações de cores como vermelho, verde e azul e como opção um valor alpha, que armazena a quantidade de transparência/opacidade. O buffer de profundidade armazena a informação de um pixel(valor z) [1].

O buffer de estêncil é usado para esboçar áreas que podem ser processadas ou não. Quando uma área da imagem é marcada para não ser processada, o buffer de estêncil funciona como mascaramento nessa área. A imagem completa, incluindo as porções mascaradas, formam o estêncil. O buffer de estêncil também pode ser usado juntamente com o buffer de profundidade para otimizar o desempenho, não desenhando porções de uma cena que está determinada a não ser visível [1].

3.5.2 Tipos Primitivos

Os tipos primitivos formam todos os modelos de uma linguagem gráfica. Na WebGL existem três tipos de primitivos, que são: *points*, *lines* e *triangles* e existem sete maneiras de usá-los: POINTS, LINES, LINE_STRIP, LINE_LOOP, TRIANGLE, TRIANGLE_STRIP e TRIANGLE_FAN.

Os pontos (POINTS) são vértices renderizados um a um. As linhas (LINES) são formadas por pares de vértices. Na Figura 3.2 pode-se observar que duas linhas compartilham o mesmo vértice. LINE_STRIP é uma coleção de vértices em que, com exceção da primeira linha, o ponto de partida de cada nova linha é ponto final da linha anterior, de forma que é possível criar quatro linhas com cinco vértices. Um LINE_LOOP é semelhante ao LINES_TRIP. A diferença é que o primeiro vértice e o ultimo são unidos, formando assim um ıloop. TRIANGLE são trios de vértices. Um TRIANGLE_STRIP utiliza os últimos dois vértices, juntamente com um vértice seguinte para formar um novo triângulo. E, finalmente, o TRIANGLE_FAN, que usa o primeiro vértice como parte de cada triângulo [1].

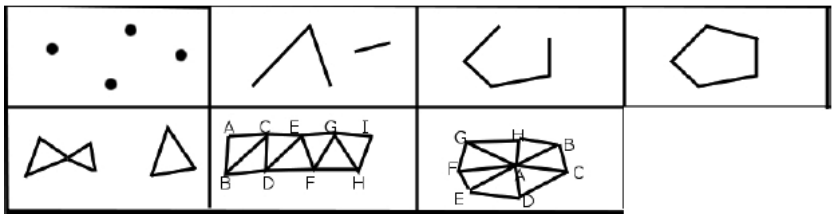

Figura 3.2: Tipos Primitivos.

3.5.3 Dados do Vértice

A WebGL usa *shaders* programáveis, e, por este motivo, todos os dados associados a um vértice precisam ser transmitidos da API JavaScript para a unidade de processamento gráfico (GPU). É necessário criar objetos de buffer de vértices (VBOs) que irão atribuir aos vértices posição de cor, vetor normal e coordenadas de texturas. Estes buffers de vértices são então enviados para o programa *shader* que irá manipular os dados [1].

3.6 Vertex Buffer Objects (VBOs)

Um VBO armazena dados sobre um tipo de atributo de seus vértices. Esse pode ser de posição, cor, vetor normal ou coordenadas de texturas. Também pode ter múltiplos atributos interligados.

Para a criação de um VBO são usadas as seguintes funções:

- *createBuffer()*: o primeiro passo é cria-lo com essa chamada.
- *bindBuffer(GLenum target, WebGLBuffer buffer)*: função que vincula o buffer criado. O parâmetro *target*, pode ser do tipo ARRAY_BUFFER que é usado para atributos de vértices, como posição e cor. Também pode ser ELEMENT_ARRAY_BUFFER, que pode ser usado quando o buffer contém índices de vértices. *buffer* é a variável na qual foi atribuída a função anterior. O segundo parâmetro é o *buffer*.
- *bufferData(GLenum target, ArrayBuffer data, GLenum usage)*: *terget* é o mesmo usado na função anterior. *data* são os dados para serem armazenados dentro do buffer. *usage* pode ser STATIC_DRAW, que define os dados uma única vez e não mudam ao longo da aplicação ou DYNAMIC_DRAW usa os dados muitas vezes na aplicação, estes sendo reespecificados para nova utilização. Ainda, pode ser STREAM_DRAW, que é similar ao primeiro, porém é utilizado somente algumas vezes na aplicação.
- *deleteBuffer()*: é possível excluir um buffer após a utilização do mesmo com essa chamada.

3.7 *Shaders*

Os *shaders* foram adicionados ao OpenGL na versão 2.0, oferecendo maior flexibilidade às aplicações. Com eles, é possível executar códigos diretamente na GPU, proporcionando um alto grau de paralelismo [14].

Com *shaders* em uma cena, tem-se um grande ganho de realismo. *Shaders* são escritos em uma linguagem de programação própria, chamada GLSL.

O OpenGL possui os seguintes estágios de processamento [2]:

1. O estágio de *Vertex shading* recebe os dados especificados pelo programador em seus ıbuffers de vértice. O processo acontece em cada vértice separadamente. Essa é uma etapa obrigatória em todos os programas OpenGL.

2. O estágio de *Tessellation shading* é opcional, gerando uma geometria adicional dentro do pipeline do OpenGL. Se ativado, ele usa a saída do estágio anterior e processa os vértices.

3. O estágio de *Geometry shading* também é opcional, podendo modificar primitivas geométricas inteiras dentro do OpenGL. A partir de uma entrada primitiva é possível gerar mais de uma geometria. Um exemplo é converter um triângulo em linhas.

4. Nessa etapa são processado os fragmentos individuais gerados por rasterização no OpenGL. Os valores de cor e profundidade de um fragmento são computados e enviados a um processamento adicional posteriormente.

5. Essa etapa não faz parte dos estágios citados acima, mas considera-se como etapa única no programa. Ela Pode processar buffers criados e consumidos por outros programas shaders em sua aplicação. Isso pode incluir efeitos de pós-processamento.

O diagrama apresentado na Figura 3.3 demonstra uma visão simplificada da *pipeline* do OpenGL, ilustrando apenas o *shader* de vértice e o de fragmento.

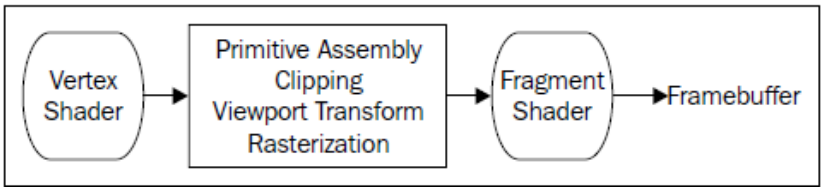

Figura 3.3: Demonstração da *Pipeline* OpenGL apenas com *Shaders*.

Dados sobre o vértice são enviados à *pipeline* e chegam ao *shader* de vértice via atributos. Isso corresponde a enviar os dados a um *shader* usando atributos e VBO (*Vertex Buffer Objects*). Também é possível enviar valores uniformes para o *shader*, sendo estes constantes para todos vértices, podendo ser uma matriz de visualização, matriz de projeção, posição de luz, dentre outras configurações. Como o *shader* é compilado externamente, é preciso fazer referência à sua localização dentro do programa. Uma vez obtida a localização de uma variável, pode-se então enviar dados para o *shader* a partir da aplicação *web* [1].

4 Sistemas de Partículas

4.1 Introdução

No inicio da década de 1980, durante um projeto de efeitos especiais para a longa metragem *A Ira de Khan* da série *Jornada nas Estrelas*, o cientista Willian T. Reeves recebeu o desafio de criar um efeito de simulação do que foi denominado de Efeito Gênesis, uma cena dotada de uma diversidade de detalhes altamente complexos. Em análise ao problema, Reeves percebe que a computação gráfica por si só não poderia proporcionar tal representação. Reeves confrontou tais conhecimentos com outras teorias e deu origem a uma técnica surpreendente e robusta, a qual denominou de Sistemas de Partículas, que é capaz de representar incontáveis fenômenos dotados de dinamicidade e complexidade, como fogo, fumaça, explosões, fluídos, gases e muitos outros elementos [10].

4.2 Características de um Sistema de Partículas

Os sistemas de partículas têm sido utilizados para simular uma grande gama de fenômenos naturais, e, se diferem em três formas básicas de uma representação normalmente utilizada na construção de uma imagem. Primeiro, um objeto não é representado por um conjunto de elementos primitivos, como polígonos, mas como nuvens de partículas primitivas que definem o seu volume. Em segundo lugar, um sistema de partículas não é uma enti-

dade estática, ela muda de forma e se move com o passar do tempo. Novas partículas "nascem"e antigas "morrem", também podendo ser restauradas as mortas. Em terceiro lugar, um objeto representado por um sistema de partículas não é determinante, pois sua forma não é completamente especificada. Em vez disso são usados processos estocásticos para criar e mudar a forma e aparência do objeto [10].

Na computação gráfica, um sistema de partículas é um grupo de objetos que são usados para simular efeitos, como fumaça, fogo, explosões, e outros fenômenos semelhantes. Cada partícula é considerada como sendo um objeto, com um ponto de posição.

Os fenômenos que modelam o sistema de partículas não são fixos, por isso são difíceis de modelar nos métodos de polígonos tradicionais. Por esse motivo são modelados por nuvens de partículas primitivas, formando o volume do objeto. Cada partícula do sistema tem a sua dinâmica.

Os atributos que uma partícula podem incluir são:

- Posição
- Velocidade
- Cor
- Transparência
- Idade
- Tamanho

Podem haver outros tipos de atributos como forma, aceleração, posição anterior, rotação e assim por diante. O sistema de partículas pode ou não lidar com colisão. As partículas geralmente são pequenas e são afetadas por forças externas, como gravidade e vento, mas não são afetadas por iluminação ou sombras.

O ciclo de vida de uma partícula no sistema é:

1. Geração/Nascimento: inicializado em algum local, podendo estar dentro de um objeto ou não, mas com uma quantidade adicional de aleatoriedade.

2. A vida dinâmica: Os seus atributos variam ao longo do tempo. Muitas vezes, o atributo é definido por uma equação paramétrica usando o tempo como parâmetro.

3. Extinção: Quando a idade da partícula (que começa em 0 e tradicionalmente é medido em número de quadros) atinge um tempo de vida pré-definido; ou a partícula atinge o solo, outro limite, deixa o quadro de vista, ou alguma outra regra for atendida, a partícula é destruída.

Estes atributos foram levados em consideração para implementação do sistema de partículas, que é discutido no Capítulo 5.

5 Desenvolvimento da Aplicação

5.1 Introdução

A implementação do Sistema de Partículas foi feita utilizando WebGL. Este sistema poderá ser utilizado para reproduzir o comportamento de alguns elementos da natureza, ou efeitos especiais.

No sistema de partículas criado, estão implementados métodos genéricos, de modo que os efeitos e as formas dos efeitos podem ser modificados com o minimo de alterações no código, podendo posteriormente ser usado como base para demais simulações.

Neste Capítulo será trabalhada a montagem da aplicação, começando a citar os atributos utilizados até o *rendering* do sistema em si.

5.1.1 Apresentação

Inicialmente deve-se:

- Inicializar os *shaders* de vértice e de fragmento.
- Criar os *buffers* para a aplicação.
- Carregar a *textura*.

O sistema de partículas possui vários atributos, que irão influenciar no seu comportamento. O atributo *size* define o tamanho que as partículas

possuirão na etapa de *rendering*. O atributo *particlePosition* define a posição em que a partícula é gerada no ambiente 3D, utilizando as coordenadas x,y e z. O atributo *particleVelocities* fornece a direção da mesma. Já o atributo *particleAcceleration*, trabalha a aceleração que a partícula sofre durante sua vida.

O atributo *particleLifeTime*, define o tempo de vida que uma partícula terá. Para representar se a uma partícula está 'morta" ou "viva", foi necessário criar o atributo *particleStatus*, o qual armazena 1 (viva) ou 0 (morta). Foi criado também um atributo que define o tempo em que a partícula ficará "morta" (*particleRespawnTime*) até que renasça.

Para auxiliar no controle do renascimento, o atributo *respawnTime* guarda o tempo corrente da partícula "morta". Todos os atributos foram usados para dar uma maior realidade e flexibilidade ao sistema de partículas criado, sempre visando a reutilização para várias simulações.

Ainda, o sistema de partículas pode contar com uma textura. Esta pode ser uma imagem com textura única ou com várias texturas na mesma. Quando há duas ou mais textura na imagem, deverá ser feito a divisão para que cada textura se encaixe em sua partícula, não ocorrendo erros na renderização, como Figura 5.1 que mostra um exemplo de imagem que pode ser utilizada para criar as texturas das partículas.

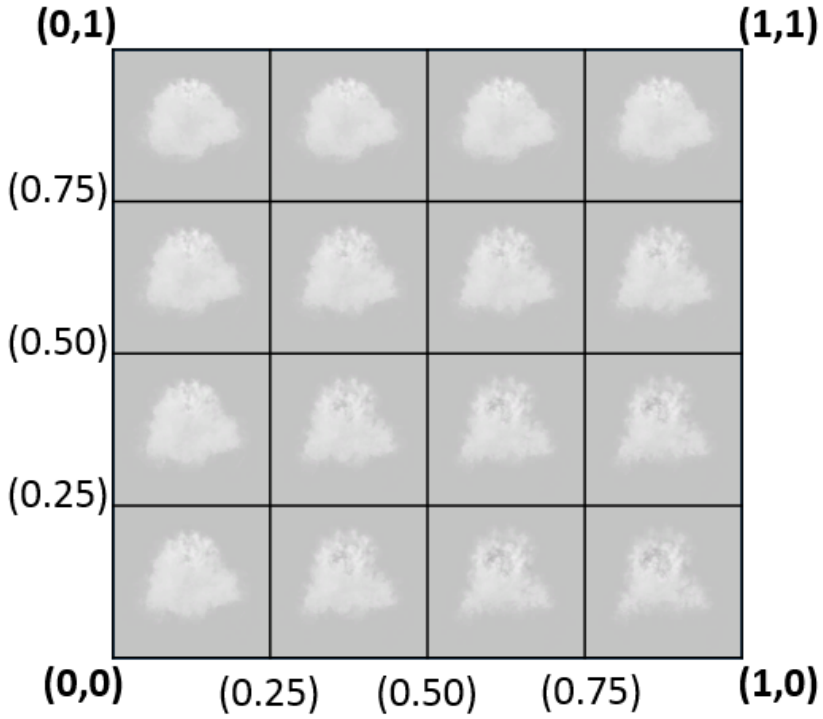

Figura 5.1: Imagem com várias texturas que podem ser aplicadas aleatoriamente às partículas. Retirado de http://www.tonysalvaggio.com/?page_id=22

.

O movimento das partículas será feito usando a fórmula 5.1 [14]:

$$P(t) = P + V * t + \frac{1}{2} * a * t^2 \tag{5.1}$$

onde, $P(t)$ é a posição atual da partícula a ser atualizada, P é a sua posição inicial, V sua velocidade, t o tempo decorrido desde seu nascimento e a sua aceleração.

Na inicialização do sistema são geradas as posições, direções, aceleração e texturas iniciais para cada partícula de acordo com o tipo de efeito a ser simulado(fogo, fumaça ou explosão), sempre utilizando a equação 5.1 para manter uma aceleração constante das partículas.

5.1.2 Atualização

Na atualização do sistema, a primeira etapa é atualizar o *timer* de cada partícula; este *timer* serve para verificar o tempo que a mesma ficará "viva" ou o tempo de espera para "renascer", caso o sistema esteja configurado para ter o renascimento de partículas.

Caso a partícula esteja viva, é então calculada a sua nova posição e verifica se o seu tempo "viva" excedeu seu tempo máximo de vida; Caso sim, ela passa então a ficar "morta".

Caso a partícula esteja morta, é verificado se o tempo que esteve "morta" já foi superior ao seu tempo de renascimento, e, em caso positivo, esta passa a ficar "viva".

Este processo pode ser visto no Código 5.1.

```
this.updateParticleSystem = function(timeElapsed, gl){
  var i;
  for(i = 0; i < this.numParticles; i++)
  {
    this.particleTimer[i] += timeElapsed;
    if(this.particleStatus[i] == 1)
    {
      this.particlePositions[i * 3  ] += this.particleVelocities
    [i * 3]    * timeElapsed + ((1/2) * this.particleAcceleration
    [i] * (timeElapsed * timeElapsed))  ;
      this.particlePositions[i * 3+1] += this.particleVelocities
    [i * 3+1]  * timeElapsed + ((1/2) * this.particleAcceleration
    [i] * (timeElapsed * timeElapsed))  ;
      this.particlePositions[i * 3+2] += this.particleVelocities
    [i * 3+2]  * timeElapsed + ((1/2) * this.particleAcceleration
    [i] * (timeElapsed * timeElapsed))  ;
      if(this.particleTimer[i] > this.particleLifeTime[i])
      {
        this.particleStatus[i] = 0;
        this.particleTimer[i] = 0;
      }
    }
    else if(this.particleStatus[i] == 0 && this.particleRespawn)
    {
```

```
19          if(this.particleTimer[i] >= this.particleRespawnTime[i])
20          {
21            this.particleStatus[i] = 1;
22            this.particleTimer[i] = 0;
23            this.particlePositions[i * 3   ] = this.center[0];
24            this.particlePositions[i * 3+1] = this.center[1];
25            this.particlePositions[i * 3+2] = this.center[2];
26          }
27        }
28      }
29  };
```

Listagem 5.1: Código de atualização do sistema de partículas.

Após fazer a atualização no sistema de partículas, é feita a atualização também nos *buffers* da GPU, enviando assim as novas posições das partículas.

5.1.3 Rendering do Sistema de Partículas

Para o *rendering* das partículas, é necessário transformá-las em quadrados e aplicar texturas, isso porque cada partícula é um ponto e não aceitaria textura. Para cada partícula é gerado um quadro e as coordenadas de textura para este. Estes dados são enviados à GPU e então é feito o *rendering*. Técnicas mais avançadas poderiam ser utilizadas em versões mais atuais da GLSL (4.5) ou do GLSL ES (3.0), porém, até a data de escrita deste documento, estas ainda não são suportadas nos navegadores e estão em fase de implementação.

Na Figura 5.2 é possível observar como as partículas são geradas sem texturas, em sua forma mais simples.

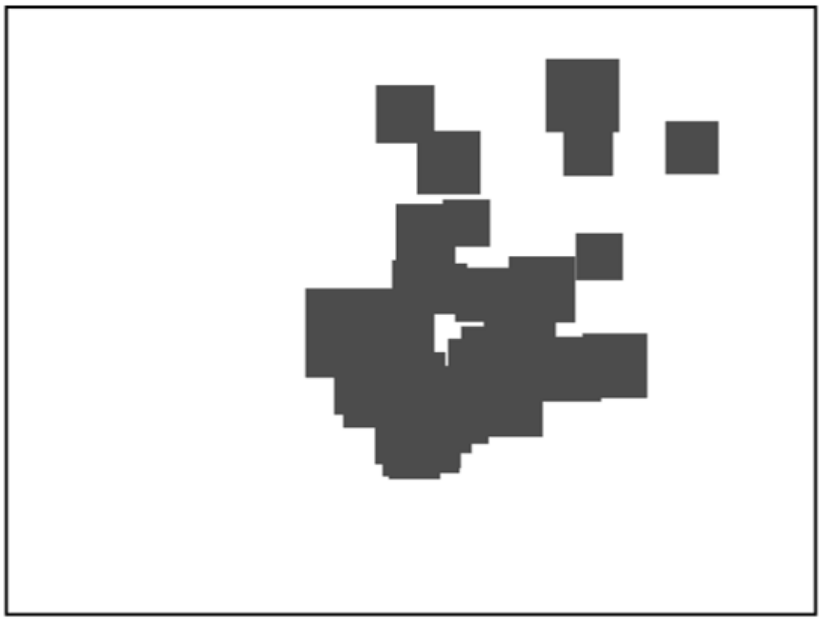

Figura 5.2: Transformação de partículas em quadrados.

Os shaders utilizados para rendering do sistema são bem simples. Apenas calculam a posição final dos pontos, e, repassam as coordenadas das texturas para os vértices e fragmentos. Os Códigos 5.2 e 5.3 mostram os shaders utilizados. O shader de vértice recebe as coordenadas de textura e interpola estas, repassando aos fragmentos. Os fragmentos que possuem cor preta são descartados, assim, as texturas possuem fundo preto e somente o que for diferente de preto aparecerá na tela.

```
1  attribute vec3  particlePosition;
2  attribute vec2 textureCoord;
3  //matrizes de view e projecao
4  uniform mat4 uMVMatrix;
5  uniform mat4 uPMatrix;
6  uniform int useTexture;
7  //sera enviado ao fragment shader
8  varying highp vec2 vTextureCoord;
9  void main()
10 {
11   //posicao do vertice projetada
12   gl_Position = uPMatrix * uMVMatrix *
```

```
13      vec4(particlePosition, 1.0);
14    if(useTexture == 1)
15    {
16      //define a coordenada de textura
17      //enviada para o shader pela cpu
18      vTextureCoord = textureCoord;
19    }
20  }
```

Listagem 5.2: *Shader* de vértice.

```
1  //vem interpolado do vertex shader
2  varying highp vec2 vTextureCoord;
3  uniform sampler2D uSampler;
4
5  void main(void) {
6    //cor do fragmento a partir da textura
7    gl_FragColor = texture2D(uSampler, vec2(vTextureCoord.s,
       vTextureCoord.t));
8    //aplica transparencia
9    gl_FragColor.a = 0.45;
10   //se o fragmento tem a cor preta, e descartado
11   if(gl_FragColor.r == 0.0 && gl_FragColor.g == 0.0 &&
       gl_FragColor.b == 0.0)
12     discard;
13 }
```

Listagem 5.3: *Shader* de fragmento.

6 Testes

Neste capítulo serão apresentados alguns exemplos de sistemas de partículas criados e será analisado o comportamento dos atributos.

6.1 Sistemas de partículas de Fogo e Fumaça

Vários Testes foram realizados constantemente durante o andamento do trabalho, porém aqui são feitas algumas observações e será possível entender um pouco mais o lado prático do que foi discutido Capítulo 5.

Um dos elementos simulados foi o fogo. O fogo tem sua origem em uma posição que inicialmente é fixa, e depois pode se alastrar, assim mudando de posição. Aqui a atenção será voltada apenas para uma fonte única de fogo. Foi utilizada uma textura com várias labaredas de fogo e cada partícula é colorida com uma dessas. A Figura 6.1 mostra a textura de fogo utilizada e aa Figura 6.2 mostra o efeito gerado.

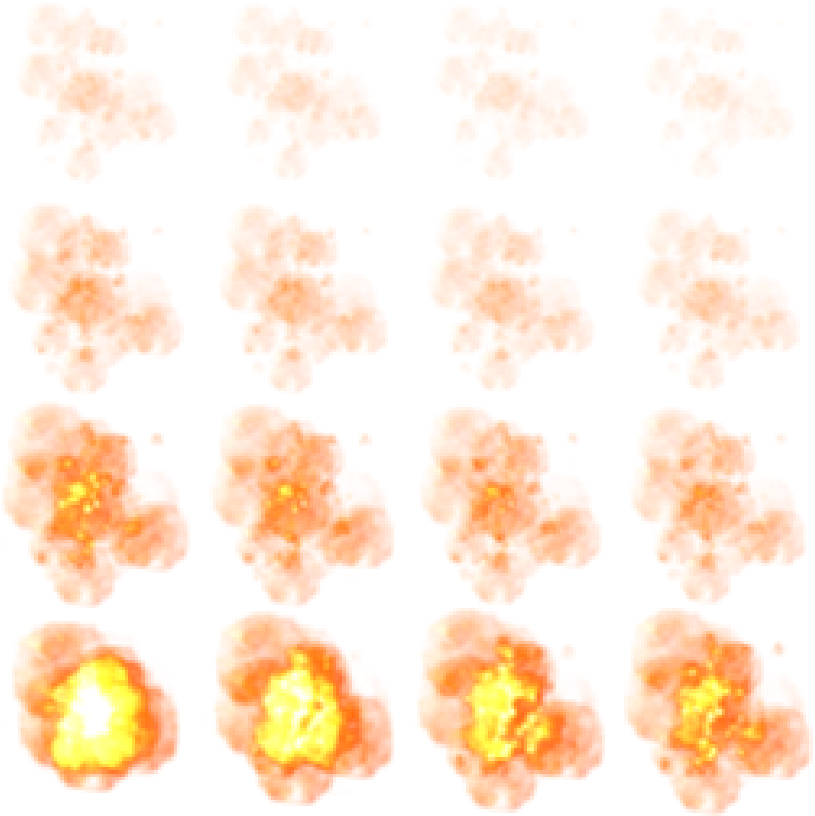

Figura 6.1: Textura utilizada para gerar o efeito de fogo. Fonte: http://www.nordenfelt-thegame.com/blog/wp-content/uploads/2011/11/explosion_opaque.png

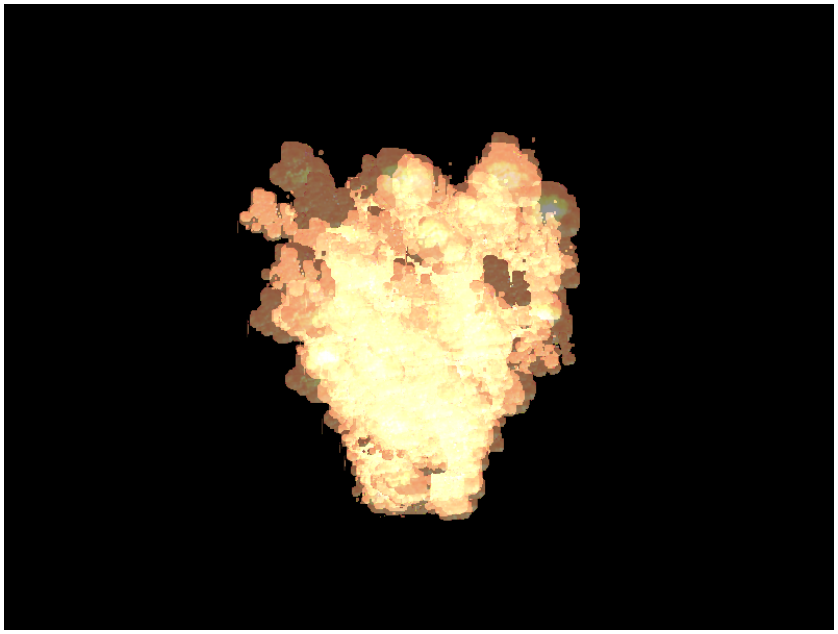

Figura 6.2: Exemplo de Simulador de Fogo

Características do fogo no sistema:

- Volume geralmente baixo em comparação a uma fumaça.
- Velocidade alta de geração de partículas.
- Tempo de vida curto das partículas.

Pode acontecer do fogo ser influenciado pelo vento, resultando na Figura 6.3.

Outro teste é a fumaça. Esta possui algumas características diferentes do fogo. As suas características são:

- Tempo de vida das Partículas relativamente longa.
- Volume grande.
- Aceleração não muito alta.
- Origem é opcional no sistema.
- Tem maior transparência.

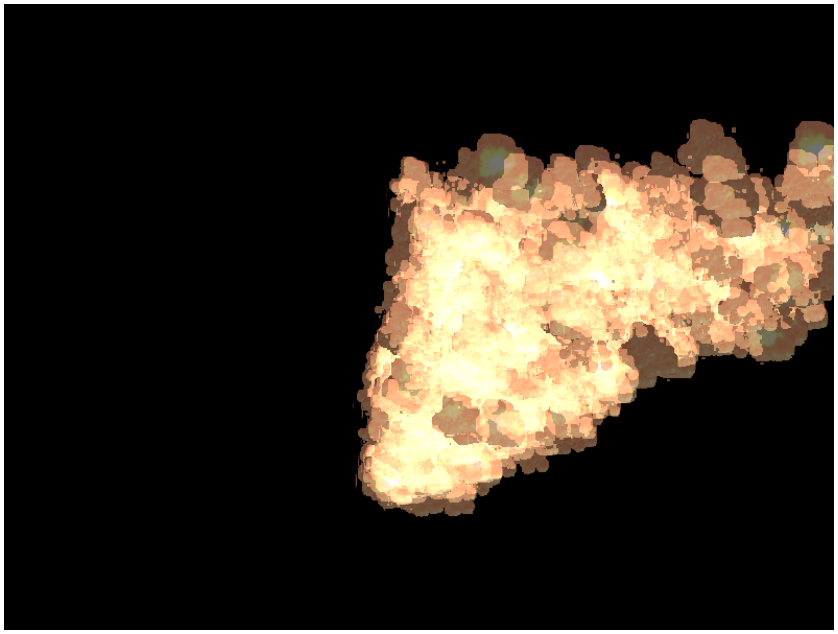

Figura 6.3: Exemplo de Simulador de Fogo, efeitos do vento.

A Figura 6.4 mostra o exemplo de um sistema de partículas para simular fumaça.

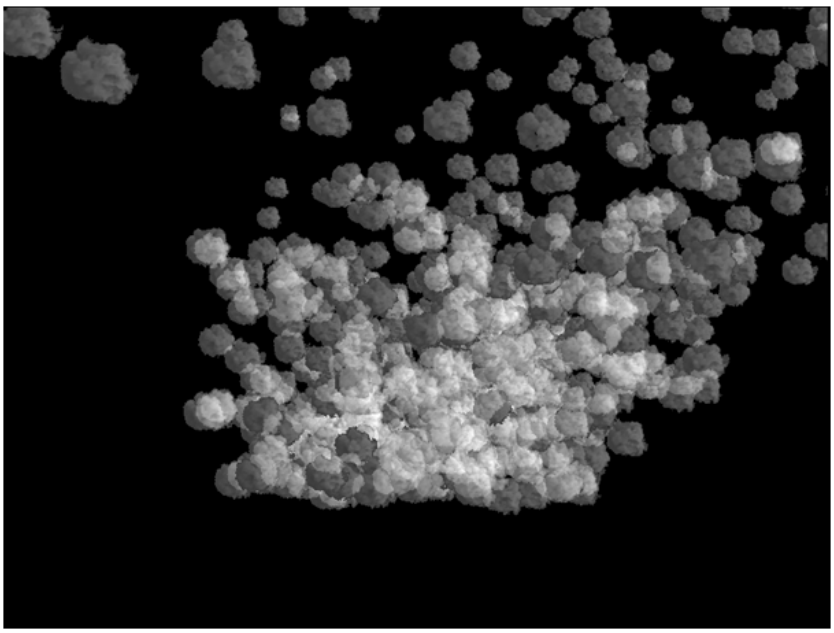

Figura 6.4: Exemplo de Simulador de Fumaça.

Na Figura 6.5, foi colocado para geração 10000 partículas, afim de mostrar o comportamento do *fps*, e com esse número de partículas ocorre uma redução no *fps*, pois quanto maior o numero de partículas a renderizar, mais pesado a aplicação ficará.

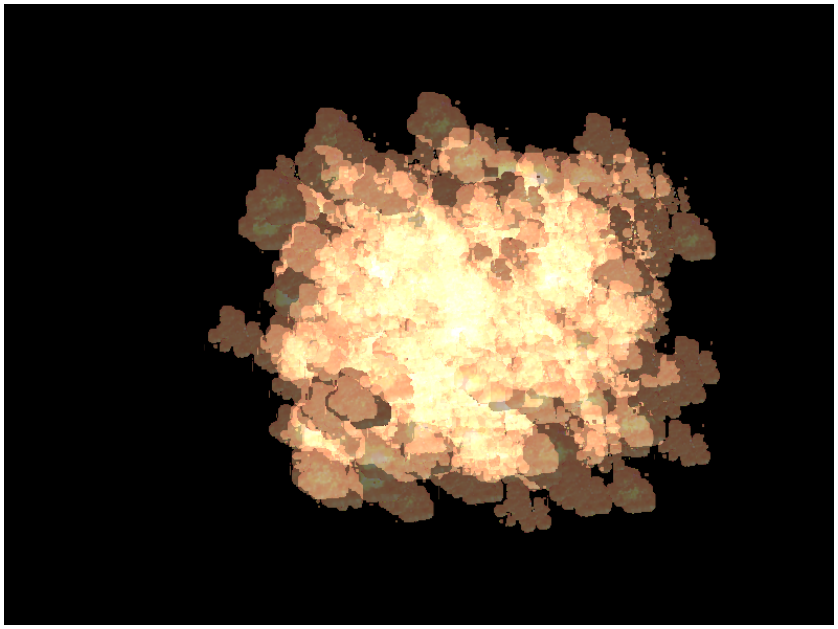

Figura 6.5: Exemplo de Simulador de Explosão.

Outros exemplos de partículas são mostrados nas Figuras 6.6, 6.7 e 6.8.

Figura 6.6: Efeito de Folhas.

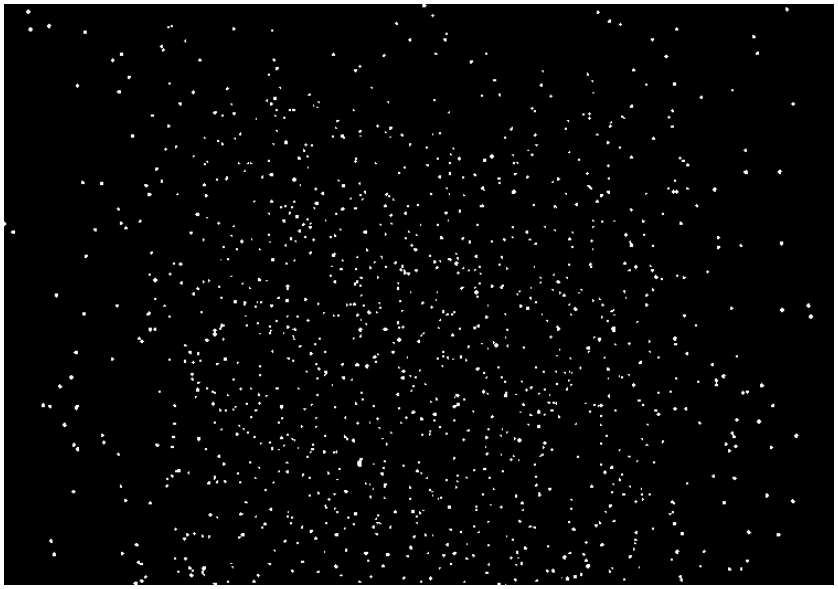

Figura 6.7: Efeito de Neve.

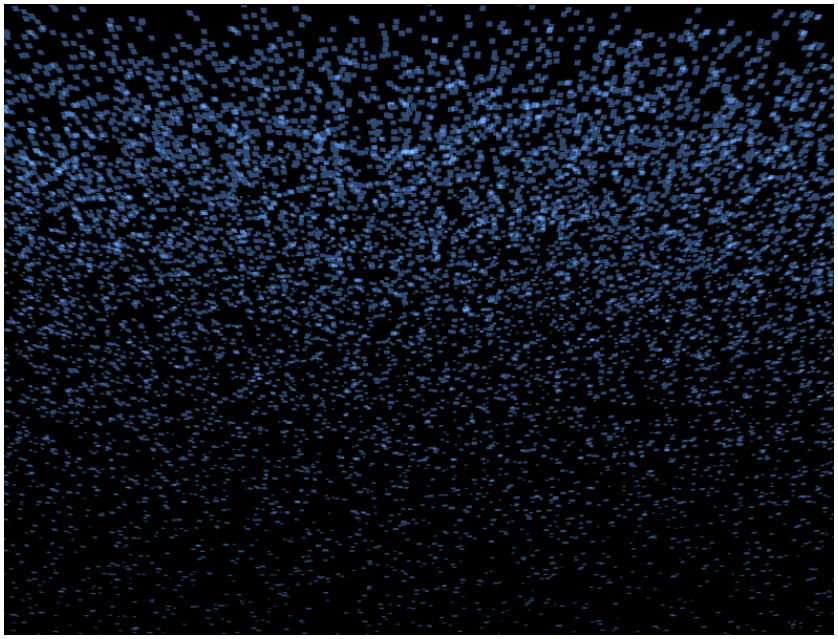

Figura 6.8: Efeito de Chuva.

A Tabela 6.1 mostra o número de *frames* por segundo obtidos através de exemplos utilizados no sistema de partículas. O navegador limita o número de *frames* em 60 *fps*. Com 1000 partículas o navegador consegue mostrar todo o sistema mantendo esta taxa de *fps*.

Tabela 6.1: Número de *frames* por segundo.

Número de Partículas	**Número médio de *Frames* por segundo**
200	60
2000	60
20000	56
200000	6

7 Conclusão

Trabalhar com WebGL não é tão simples, sendo necessário muita dedicação, mas os resultados são recompensadores. Esta API tende a explorar o avanço das GPUs, pois estas estão a cada dia se tornando mais poderosas. A ferramenta torna possível criar aplicações no navegador que anteriormente era algo somente para desktop.

Conforme a aplicação estava em andamento, os olhos se abriam quanto a imensidão que um sistema de partículas abrange. Alguns exemplos foram colocados no capítulo anterior. WebGL pode trazer ao navegador e instantaneamente as páginas *web* efeitos incríveis, podendo conter interatividade com o usuário das mesmas.

A WebGL permite criar sistemas de partículas, porém, grande parte do cálculo tem que ser feito na CPU (atualizar as partículas) e estes dados devem então serem copiados para GPU. Na GLSL ES 3.0+ já é possível usar *shaders* para fazer cálculos não apenas de *rendering* (como a atualização das partículas), porém, isso não está implementado ainda nos navegadores atuais (chrome, firefox). Uma nova versão do webgl está em desenvolvimento e logo permitirá realizar cálculos mais rapidamente.

Bibliografia

[1] Brian Danchilla. *Beginnig WebGL for HTML5*. Apress, 2012. ISBN 9781430239963.

[2] John Kessenich Dave Shreiner, Graham Sellers and Bill Licea-Kane. *OpenGL Programming Guide Eighth Edition*. Pearson Education, Inc., 2013. ISBN 9780321773036.

[3] Diego Eis e Elcio Ferreira. *HTML5 e CSS3 com farinha e pimenta*. Tableless, 2012. ISBN 9781105096358.

[4] Steve Fulton and Jeff Fulton. *HTML5 Canvas, Second Edition*. O?Reilly Media, Inc, 2013. ISBN 9781449334987.

[5] Khronos Group. The industry's foundation for high performance graphics, Acessado em 12 de julho de 2016. URL https://www.khronos.org/opengl.

[6] Khronos Group. Webgl specification, Acessado em 21 de julho de 2016. URL https://www.khronos.org/registry/webgl/specs/1.0/.

[7] Rodger Lea Kouichi Matsuda. *WebGL Programming Guide: Interactive 3D Graphics Programming with WebGL*. Pearson Education, Inc., 2013. ISBN 9780321902924.

[8] Muhammad M. Movania. *OpenGL Development Cookbook: Over 40 recipes to help you learn, understand, and implement modern OpenGL in your applications*. Packt Publishing Ltd., 2013. ISBN 9781849695046.

[9] OpenGL.org. Rendering pipeline overview, Acessado em 12 de juho de 2016. URL https://www.opengl.org/wiki/Rendering_Pipeline_Overview.

[10] Willian T. Reeves. Particle systems: A technique for modeling a class of fuzzy objects. *ACM Transactions on Graphics*, 2(2):91–108, 1983.

[11] Maurício Samy Silva. *JavaScript: Guia do Programador*. Novatec Editora Ltda., 2010. ISBN 9788575222485.

[12] Maurício Samy Silva. *HTML 5: A linguagem de Marcaćão que Revolucionou a Web*. Novatec Editora Ltda., 2011. ISBN 9788575222614.

[13] Maurício Samy Silva. *HTML5: A linguagem de marcação que revolucionou a web*. Novatec Editora Ltda, 2014. ISBN 9788575224038.

[14] David Wolff. *OpenGL 4 Shading Language Cookbook Second Edition*. Packt Publishing, 2013. ISBN 9781782167020.