# 6.
# Desenvolvimento

Para tornar viável o desenvolvimento de várias técnicas de simulação para análise de desempenho, é necessário o projeto de uma arquitetura de software extremamente flexível e extensível. Para isso a utilização de vários padrões de projeto de software (Gamma et al., 1995) é crucial para efetivação de tais requisitos.

Além disso, devido às recentes inovações tecnológicas das placas gráficas, surge uma nova forma de programação para as mesmas. Portanto, para aproveitar melhor os recursos do hardware e realizar testes, para o desenvolvimento do sistema foi utilizada a API *CUDA* da NVidia e a *API DirectCompute* da Microsoft.

## 6.1. *CUDA*

*CUDA* (acrônimo de Compute Unified Device Architecture) é uma arquitetura de computação paralela desenvolvida pela NVidia (NVIDIA Corporation , 2009) e disponibilizada para uso em Fevereiro de 2007. *CUDA* é um motor de computação das placas de vídeo da NVidia, que permite acesso aos recursos do hardware para desenvolvedores de software sem a necessidade de utilizar das *APIs* gráficas. Dessa forma, programadores sem conhecimento avançado em computação gráfica podem usufruir dos benefícios da computação paralela em placas de vídeo.

Contudo, há, pelo menos, três pontos chave para que a utilização do *CUDA* traga eficiência para a aplicação. 1) O problema deve ser paralelizável. 2) Deve-se conhecer a arquitetura física da placa na qual se deseja programar, a fim de estabelecer os melhores parâmetros de configuração e alocação de recursos da placa. 3) A placa de vídeo deve

suportar *CUDA*, ou seja, nenhuma placa das concorrentes ATI e Intel servirão para o sistema. A análise detalhada do funcionamento das arquiteturas das placas de vídeo, dos modos de configuração da *API CUDA* e de modos de "*tunning*" são discutidos no documento público de melhores práticas da NVidia (2009).

Portanto, deve-se analisar com cuidado a introdução de *CUDA* em uma aplicação. As explicações de arquitetura e modos de operação da *API* são discutidas em NVidia (2009).

## 6.2. DirectCompute

Microsoft *DirectCompute* é uma *API* que permite realizar o chamado *GPGPU* (General-purpose computing on graphics processing units) sem a necessidade de utilizar uma *API* gráfica ativada. Desse modo, ela tem o mesmo propósito que a *API CUDA*, no entanto, ela tem como diferencial ser independente de fabricante de placa de vídeo, sendo assim, possível executar tanto em placas NVidia, quanto em ATI e Intel. DirectCompute é parte da coleção de *APIs* do Microsoft *DirectX*.

Apesar de muito mais simples de utilizar do que *CUDA*, pouquíssimas placas suportam suas funcionalidades, sendo assim, a maioria das funções é emulada por software, tornando-a lenta e utilizável, ainda, apenas para teste. Infelizmente, até a escrita deste trabalho não havia disponível uma placa para a realização dos testes nessa *API*, sendo assim, ela foi utilizada apenas para fins de estudo e não foram realizados os testes comparativos de desempenho dela com as demais *APIs*.

## 6.3. Arquitetura

Para o desenvolvimento geral do sistema, foi adotado o padrão Model-View-Controller (MVC) (Singh, 2002), que é um padrão de projeto de software bastante difundido para sistema de visualização e simulação.

Com o objetivo de desenvolver um sistema extensível para visualização e simulação de oceano, optou-se por criar um motor 3D.

Esse motor deve suportar os vários tipos de simulação de oceano e permitir a inserção de malhas 3D convencionais. A Figura 39 apresenta o diagrama de classes em alto nível do motor.

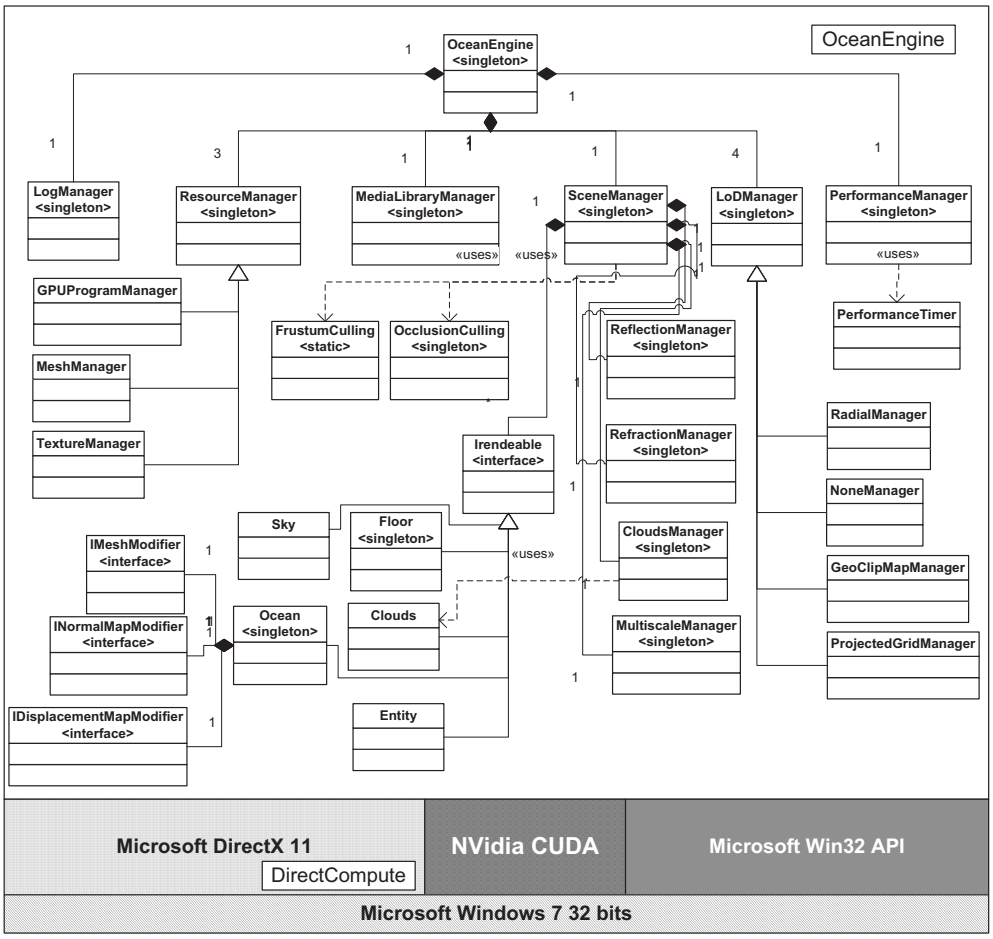


**Figura 39:** Arquitetura do Motor OceanEngine

O sistema foi desenvolvido sobre a *API* do Windows e utiliza o *framework* de trabalho do *DirectX*. Além disso, observa-se o uso massivo do padrão *Singleton*. Isso é necessário para garantir que os recursos não sejam duplicados, visto que, a arquitetura é *multi-thread* e, portanto, deve-se ter singletons *thread-safe*. Sendo assim, utiliza-se a técnica de *Double-checked locking* proposta por Schmidt et al. (2000), dessa forma, melhorando o acionamento do *singleton*. O *Mutex* (trava de sincronismo de exclusão mútua) utilizado é o oferecido pela função ***InterlockedExchange*** da *API* Win32.

Dividindo o sistema em subsistemas componentes, tem-se uma forma de entender o funcionamento do mesmo. Esses subsistemas são: Gerência de Recursos, Simulação, Visualização, Análise de Desempenho, *LoD*, *Log* e Catalogador de arquivos.

### 6.3.1. Subsistemas de gerência de recursos

A gerência dos recursos possui como classe *singleton* principal o *ResourceManager, o qual é* responsável por gerenciar a memória, tanto da *CPU* quanto a da *GPU.* No entanto, existem ainda gerenciadores específicos para os objetos mais usados como recursos 3D, como texturas, malhas, *buffers* e *shaders*.

Dessa forma, os gerenciadores *TextureManager*, *MeshManager* e *GPUProgramManager* possuem funções úteis para carga do dado específico, assim, o *TextureManager* é capaz de ler uma textura de um formato específico vinda do disco e carregá-la na memória da placa gráfica. O mesmo ocorre com o *MeshManager,* que carrega um objeto 3D, juntamente com suas texturas, e também com o *GPUProgramManager* que carrega um código de efeito em linguagem *HLSL*, compila e o inicializa na placa gráfica.

Os códigos desenvolvidos na *API CUDA* não são compilados em tempo real, eles são compilados separadamente pelo compilador da NVidia (nvcc) e depois é ligado ao programa final, dessa forma, não é possível mudá-lo em tempo de execução/carga. O DirectCompute pode ser usado com compilação em tempo de execução/carga ou usado estaticamente na ligação do código.

### 6.3.2. Subsistemas de simulação

A arquitetura do motor de simulação foi feita para permitir a combinação e a expansão das técnicas. Dessa forma, a arquitetura cria várias interfaces customizáveis para cada etapa de simulação. Essas interfaces são: *IDisplacementMapModifier*, *INormalMapModifier*, *IMeshModifier*, sendo,

respectivamente, responsáveis pela modificação do mapa de altura, mapa de normais e malha.

Cada algoritmo é desenvolvido como uma derivação (implementação) de uma ou mais dessas classes, de forma a permitir até mesmo a ligação em tempo de execução via uma DLL.

### 6.3.3. Subsistemas de Visualização

Nesse subsistema são desenvolvidos todos os recursos para a visualização do oceano. Em particular, as duas classes *ReflectionManager* e *RefractionManager* possuem um objetivo muito claro, elas acomodam as texturas e *RenderTargets* (classe para desenho direto na textura) da refração e reflexão, dessa forma, esses dados possuem um compartilhamento para todo o subsistema.

Nele, ainda, é realizado o desenho de todos os modelos 3D, fundo de oceano, nuvens e céu. Para permitir que outros objetos 3D sejam incorporados ao motor, tudo que venha a ser desenhado deve implementar a classe *IRenderable*, até mesmo o oceano. A classe *CloudManager* é um subgerenciador utilizado para ordenar a renderização das nuvens, visto que, as mesmas utilizam variação de intensidade do canal *alpha*, sendo assim, devem ser desenhadas de trás para frente, segundo o algoritmo do pintor.

Além disso, ele fornece as classes responsáveis por maximizar o poder de computação do sistema, que visam reduzir o esforço necessário para desenhar todos os elementos. Essas classes utilizam técnicas para que somente elementos visíveis sejam enviados para desenho na placa de vídeo, dessa forma, objetiva-se descartar elementos que não estão no volume de visão e elementos totalmente oclusos por outros. Essas técnicas são chamadas de técnicas de corte ou descarte (*culling*).

Nesse sentido, duas técnicas de corte foram desenvolvidas, o *Frustum Cullling* (FC), essencial para qualquer aplicação 3D e o *Occlusion*

*Culling* (OC). A implementação do FC foi realizada seguindo os passos descritos por Sýkora & Jelínek (2002) no espaço do mundo usando *AABB's* (caixa alinhada com os eixos cartesianos).

O OC implementado baseia-se na técnica descrita por Sekulic (2004), na qual, a funcionalidade de *Query* (*IDirect3DQuery9*) da placa é otimizada, evitando esperas (*busy waiting*) em ambos lados. Além disso, optou-se por utilizar o *Small Feature Culling*, ou seja, um corte de elementos que gerem menos de 0,001% dos pixels totais, dessa forma, numa tela de 512x512, objetos que gerem menos de 3 pixels são eliminados do desenho de um quadro.

Por fim, existe ainda um módulo gerenciador para análise de dados em multi-escala, o *MultiscaleManager*, que, basicamente, regula os planos de visão (*Near* e *Far*) e a velocidade de navegação. A técnica utilizada é a do cubo de distâncias, descrita por McCrae et al. (2009).

### 6.3.4. Subsistemas de Análise de Desempenho

A análise de desempenho é um subsistema crítico, pois, é onde são coletados os dados. Dessa forma, necessita-se da maior precisão possível do temporizador. Para obter o melhor resultado possível, utilizou-se o temporizador de alta precisão da *API* do Windows, a função *QueryPerformanceFrequency*.

O sistema conta com um gerenciador que permite que vários temporizadores possam ser acionados simultaneamente, permitindo assim, numa rodada a análise de vários trechos do sistema.

### 6.3.5. Subsistemas de *LoD*

O subsistema de *LoD* é responsável pela gerência das malhas de desenho do oceano (e eventualmente de terreno). Nele, as classes *RadialManager*, *GeoClipMapManager*, *ProjectedGridManager* e *NoneManager* são , respectivamente, as derivações dos algoritmos de

*Radial LoD*, *GeoClipMap LoD*, *Projected Grid LoD* e o algoritmo sem nenhuma técnica de *LoD*.

### 6.3.6. Subsistemas de *Log*

Esse subsistema é o mais simples, pois trata-se de uma classe *singleton* para gerenciar os *Logs* do sistema (*LogManager*).

### 6.3.7. Subsistema catalogador de arquivos

Em motores comerciais para visualização de cenas 3D, é comum separar os arquivos em diversas pastas. Contudo, muitas vezes deseja-se acessar um arquivo de determinado nome, sem a preocupação da localização física do mesmo na estrutura de pastas do motor.

Dessa forma, o gerenciador *MediaLibraryManager* executa uma busca na árvore de pastas do motor (com a raiz sendo a pasta no qual o motor se localiza) adicionando cada arquivo de dado numa tabela hash (Cormen et al., 2001) para posterior consulta. Essa busca é realizada utilizando a *API* do *Windows* (***WIN32_FIND_DATA***), que conta com diversas funções para tal finalidade.

## 6.4. Efeitos adicionais

Para a geração de uma cena de oceano realística, é necessário muito mais do que apenas as técnicas de simulação e visualização de oceano. Isso é devido à própria ótica utilizada para a iluminação do mesmo. Dessa forma, pouco adianta iluminar o oceano se os elementos externos ao mesmo estiverem com baixa qualidade visual.

Sendo assim, viu-se necessário o estudo e desenvolvimento de técnicas para renderização de elementos como o céu e as nuvens. No entanto, a descrição dessas foge ao escopo desse trabalho, mas são cruciais para uma imagem realística. Portanto, para a computação de uma imagem foto-realística de oceano ainda é preciso desenvolver as metodologias de visualização de, no mímino, o céu.

Outra funcionalidade desenvolvida para estudo da iluminação, em especial os efeitos de refração e reflexão, foi o desenvolvimento das técnicas de visualização estereoscópica. Essa foi realizada com algoritmos próprios e utilizando a *API NVidia StereoAPI*. Isso foi essencial para a percepção dos problemas ocasionados com o uso dos mapas de reflexão e refração.

## 6.5. Utilização conjunta de *CUDA* e *Microsoft DirectX*

Para o funcionamento integrado dos elementos do *DirectX* com a *API CUDA*, é necessário realizar uma série de inicializações no motor *CUDA*. Isso devido aos problemas de paralelismo, pois, uma textura não pode ser utilizada simultaneamente pelo *DirectX* e pelo *CUDA*. Dessa forma, o motor *OceanEngine* possui uma série de funções que realizam o procedimento que chaveamente entre as *APIs*.

O funcionamento integrado é necessário devido ao fato de se utilizar texturas para o processo de amostragem e iluminação, dessa forma, não é possível alocar memória na placa de vídeo para simular, como é comumente feito com CUDA.

Sendo assim, uma sequência de passos necessária para o uso do CUDA integrado com o DirectX pode ser vista como:

- Inicializa-se o dispositivo DirectX no sistema do CUDA, antes de qualquer utilização de recursos.

```
cudaD3D9SetDirect3DDevice(_pd3dDevice);
```

- Para utilizar uma textura DirectX no contexto CUDA, deve-se registrá-la no CUDA (no caso uma textura 2D, mas isso pode ser feito para qualquer tipo de textura).

```
cudaD3D9RegisterResource(_pTexture2D.pTexture, cudaD3D9RegisterFlagsNone);
cutilSafeCallNoSync( cudaD3D9MapResources (1, (IDirect3DResource9 **)&
_pTexture2D.pTexture) );
void* data;
size_t size;
// Mapeia a textura _pTexture2D.pTexture para data.
cutilSafeCallNoSync ( cudaD3D9ResourceGetMappedPointer(&data,
_pTexture2D.pTexture, 0, 0) );
```

```
// Obtendo o tamanho
cutilSafeCallNoSync ( cudaD3D9ResourceGetMappedSize(&size,
_pTexture2D.pTexture, 0, 0) );
cutilSafeCallNoSync ( cudaD3D9UnmapResources (1, (IDirect3DResource9 **)&
_pTexture2D.pTexture) );
```

- Quando deseja-se acionar a função CUDA, é necessário travar a textura para o DirectX.

```
cudaD3D9MapResources(1, _pTexture2D.pTexture);
static float t = 0.0f;
void* pData;
size_t pitch;
cutilSafeCallNoSync ( cudaD3D9ResourceGetMappedPointer(&pData,
_pTexture2D.pTexture, 0, 0) );
cutilSafeCallNoSync ( cudaD3D9ResourceGetMappedPitch(&pitch, NULL,
_pTexture2D.pTexture, 0, 0) );
 cuda_gerstner_2d(pData, _pTexture2D.width, _pTexture2D.height, pitch, t);
```

- Logo após o uso deve-se destravar a textura.

```
cudaD3D9UnmapResources(1, _pTexture2D.pTexture);
```

- Por fim, pode-se liberar o registro da textura e do dispositivo DirectX

```
cudaD3D9UnregisterResource(_pTexture2D.pTexture);
cudaThreadExit();
```

Se fosse utilizado um buffer (*Vertex Buffer* ou *Index Buffer*) seria necessária realização de um procedimento muito similar também.

## 6.6. Pipeline de Simulação e Renderização

Os trabalhos que se propõem a simular e desenhar cenas oceânicas possuem a mesma metodologia de desenho, variando apenas pequenos detalhes relativos às técnicas empregadas. Desse modo, eles não permitem a composição de técnicas e nem a escolha das mesmas. A Figura 40 ilustra a sequência de atividades do processo de visualização.

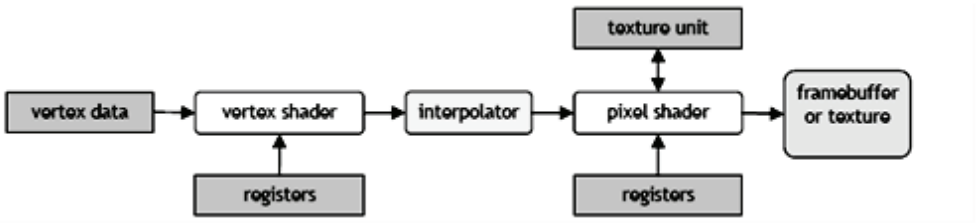


**Figura 40:** Pipeline de funcionamento das técnicas convencionais (Johanson, 2004)

Nota-se que não há separação entre o processo de simulação e de visualização, tudo ocorre na mesma linha de execução. Dessa forma, não é possível a troca de parte da simulação, pois, necessita-se modificar todo o processo.

A proposta deste trabalho utiliza o conceito do *MVC*, aplicado à programação em placa gráfica. Sendo assim, cria-se duas linhas de execução para a geração de um quadro, a linha de simulação e a linha de visualização. Um diagrama explicativo desse processo é ilustrado na Figura 41.

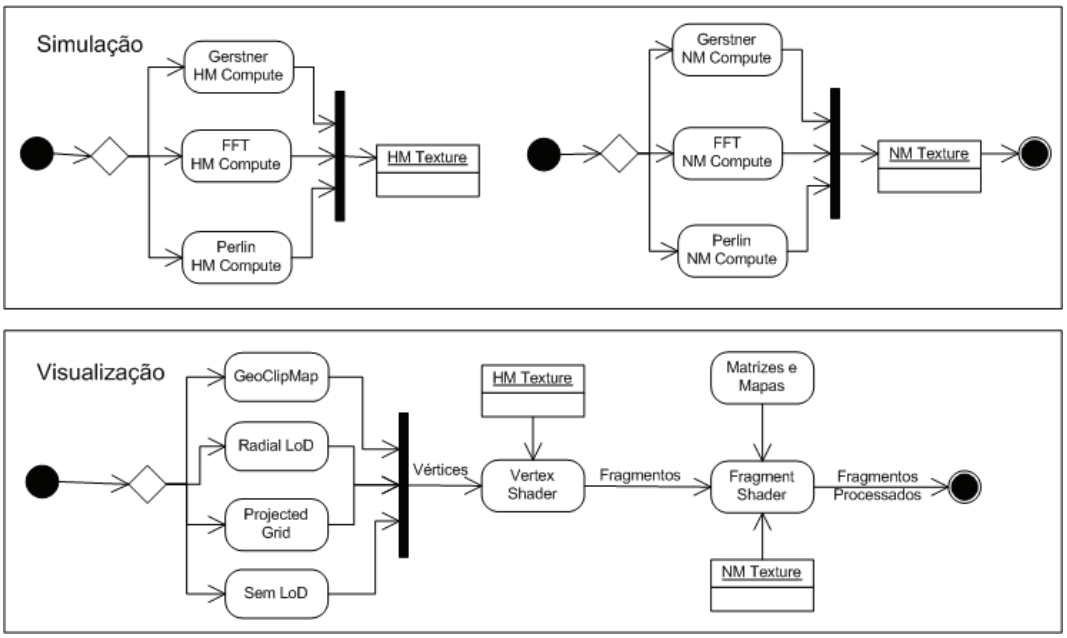


**Figura 41:** Pipeline de Simulação e Visualização

Outro benefício dessa metodologia é a possibilidade de extrair informações da malha do oceano por qualquer objeto de uma cena. Isso é muito importante para cenas que possuam vários objetos sobre a água, pois, para que os mesmos não sejam cobertos por ela, é necessário que esses sejam reposicionados em relação à altura da malha do oceano em um determinado ponto. Isso não é possível quando se realiza a simulação diretamente nos vértices da malha, pois, nenhum outro *Shader* poderá acessar a informação.

Tal fato não ocorre quando se utiliza de uma textura, pois a mesma pode ser transmitida como uma variável uniforme de textura para qualquer programa *shader*, permitindo que, por exemplo, um barco seja reposicionado para manter-se sobre a água. No caso da simulação de Gerstner, alguns softwares chegam a computar várias vezes a mesma simulação para reposicionar objetos. A alegação é a redução do consumo de memória, pois, não é necessário armazenar dados. No entanto isso tende a ser equivocado, pois, a quantidade de memória disponível nas placas aumenta cada vez mais.

Dessa forma, a linha de simulação mostra claramente a separação das etapas dos algoritmos. Nela, o primeiro losango de decisão escolhe qual será a técnica aplicada para a geração do mapa de altura, que é armazenado na textura *HM*. Após esse procedimento, a linha avança para a decisão do algoritmo de geração do mapa de normais, o qual é armazenado na textura *NM*. Nota-se que a simulação não se preocupa com nenhum detalhe da iluminação, apenas com dados de interface.

Poder-se-ia imaginar a execução paralela do *HM* e do *NM*, no entanto, há duas razões para a não ocorrência desse fato. Primeiramente, a geração do *HM* ocupa os recursos da *GPU*, impedindo que outro processo ocorra, e mesmo se fosse possível, isso traria uma grande perda de desempenho no sistema, pois, o gerenciamento de recursos perderia a otimização de localidade[7]. No entanto, um dos mapas poderia ser gerado em paralelo pela *CPU*, contudo isso ainda traria problemas, visto que haveria um grande tráfego de informações no barramento de dados, além da *CPU* poder se tornar um gargalo da simulação. O segundo motivo é o fato de que quando se serializa esse processo, pode-se utilizar dados já calculados no processo de geração do *HM* para acelerar o do *NM*, mesmo sem saber o que de fato será utilizado no *NM*.

[7] Todavia, isso seria possível se fosse utilizado um sistema com duas placas em paralelo.

Analisando o processo de visualização, percebe-se que não há variação no *Vertex Shader* e no *Pixel Shader*, dessa forma, pode-se mudar a forma de iluminação sem alterar o processo de simulação, o que é esperado pelo padrão *MVC*. Além disso, a modularização dessas frentes de execução é importante para depuração também, pois, blinda-se cada processo dos erros do outro, além de facilitar a localização de eventuais erros.

As técnicas de nível de detalhe servem de geradores de vértices para o *Shader*. Assim que o vértice entra no processo de rasterização, ele chega ao programa de vértice, o qual utilizará a textura de mapa de altura para acessar os deslocamentos horizontais e verticais do mesmo. Nota-se que as texturas de *HM* e *NM* assumem o papel do *Controller* do padrão *MVC*. Após a passagem pelo *Vertex Shader*, o vértice é transformado em fragmento não processado e então inicia-se a passagem pelo *Fragment Shader*, ou *Pixel Shader*, onde ele sofrerá o processo de iluminação. Por fim, o mesmo será composto pela *API* de renderização e, dependendo do corte do *Z-Buffer*, tornar-se-á um *pixel*.

## 6.7. Desenvolvimento e Execução

Inicialmente, desenvolveu-se o motor utilizando a linguagem C#.NET, por se tratar de uma linguagem extremamente extensível e produtiva. Apesar de não ser observada uma perda significativa de desempenho com a utilização da mesma, optou-se por utilizar a linguagem C++ para produção do sistema final. Assim, garante-se o menor *overhead* (custo operacional) possível.

O desenvolvimento da parte 3D da aplicação foi realizado com o auxílio da *API Microsoft DirectX*, e dessa forma, optou-se por utilizar a linguagem *HLSL* para codificação de *Shaders*. A *API CUDA* foi utilizada como solução alternativa ao uso de *Shaders*. Conforme discutido, apesar da arquitetura utilizar o *DirectCompute*, não foi possível realizar testes com o mesmo.

O código *CUDA* é criado em um arquivo separado dos de projeto com a extensão "***cuda***". Nesse, utiliza-se a linguagem C para desenvolver os algoritmos, contudo, algumas palavras-chave e variáveis internas possuem um significado especial. Na Tabela 9 mostra o código para a computação de um mapa de altura e deslocamento usando Gerstner.

**Tabela 9:** Exemplo do programa CUDA para simulação de Gertstner

```
01 //#define SHARED_MEMORY
02 //#define SHARED_MEMORY_CONST
03 #define CONST_MEMORY
04
05 __device__ __constant__ float ger_data[147];
06
07 __device__
08 float dotf(float3 a,float bx, float by, float bz)
09 {
10     return a.x*bx + a.y*by + a.z*bz;
11 }
12
13 __global__ void KernelGerstnerCUDA_H(unsigned char* surface, int nwaves, float
*data, int width, int height, size_t pitch, float t)
14 {
15     int x = blockIdx.x*blockDim.x + threadIdx.x;
16     int y = blockIdx.y*blockDim.y + threadIdx.y;
17     float* pixel;
18     if (x >= width || y >= height) return;
29 #ifdef SHARED_MEMORY
20     int cnt = max(1,7 * nwaves / (blockDim.x* blockDim.y)+1);
21     __shared__ float local_data[147];
22     for(int i = 0; i < cnt; ++i)
23     {
24         int p = min(7 * nwaves-2, (threadIdx.x + threadIdx.y*blockDim.x)*cnt);
25 #ifdef SHARED_MEMORY_CONST
26         local_data[p+i] = ger_data[p+i];
27 #else
28         local_data[p+i] = data[p+i];
29 #endif
30     }
31     __syncthreads();
32 #endif
33
34     pixel = (float*)(surface + y*pitch) + 4*x;
35
36     float3 P0 = make_float3(x,0,y);
37     float3 Position = make_float3(x,0,y);
38
39     for(int i = 0; i< nwaves; ++i)
40     {
41 #ifdef CONST_MEMORY
42         float Q = ger_data[i*7];
43         float FREQ = ger_data[i*7+1];
44         float PHASE = ger_data[i*7+2];
45         float AMP = ger_data[i*7+3];
46         float DX = ger_data[i*7+4];
47         float DY = ger_data[i*7+5];
48         float DZ = ger_data[i*7+6];
49 #elif defined(SHARED_MEMORY)
50         float Q = local_data[i*7];
51         float FREQ = local_data[i*7+1];
52         float PHASE = local_data[i*7+2];
53         float AMP = local_data[i*7+3];
54         float DX = local_data[i*7+4];
55         float DY = local_data[i*7+5];
56         float DZ = local_data[i*7+6];
57 #else
58         float Q = data[i*7];
```

```
59          float FREQ = data[i*7+1];
60          float PHASE = data[i*7+2];
61          float AMP = data[i*7+3];
62          float DX = data[i*7+4];
63          float DY = data[i*7+5];
64          float DZ = data[i*7+6];
65  #endif
66
67          float angle = FREQ * dotf(P0,DX,DY,DZ) + PHASE * t;
68
69          float Si = sin(angle);
70          float C = cos(angle);
71
72          Position.x += Q * AMP * DX * C;
73          Position.z += Q * AMP * DZ * C;
74          Position.y += AMP * Si;
75      }
76
77      pixel[0] = Position.x;
78      pixel[1] = Position.y;
79      pixel[2] = Position.z;
80      pixel[3] = 1;
81  }
82
83  extern "C"
84  void GerstnerCUDA_H(void* surface, int nwaves, float *data, int width, int
height, size_t pitch, float t)
85  {
86      cudaError_t error = cudaSuccess;
87
88      dim3 Db = dim3( 16, 16 ); // block dimensions are fixed to be 256 threads
89      dim3 Dg = dim3( (width+Db.x-1)/Db.x, (height+Db.y-1)/Db.y );
90
91      KernelGerstnerCUDA_H<<<DG,DB>>>( (unsigned char*)surface, nwaves, data,
width, height, pitch, t );
92
93      error = cudaGetLastError();
94      if (error != cudaSuccess) {
95          printf("GerstnerCUDA_H() failed to launch error = %d\n", error);
96      }
97  }
```

A *API CUDA* dispõe de vários tipos de memória para o armazenamento e acesso de dados, dentre esses, destacam-se a memória Global, Constante (***__constant__***), Local (***__shared__***) e de Textura. O código da Tabela 9 foi desenvolvido para testar o desempenho obtido com a memória Global (que permite leitura e escrita pelo kernel *CUDA*), Constante (que só permite leitura pelo *Kernel*, sendo a escrita feita pela *CPU* na carga de dados) e Local (que é apenas para um bloco de threads *CUDA*). Observa-se na linha 31 o uso da função `__syncthreads`, que serve como barreira para um grupo de *threads* de um bloco de execução *CUDA*. Isso é muito útil para transposição de dados na memória global para a local, pois, asegura que todas as *threads* possuam os dados globais.

A linha 13 revela mais uma palavra especial chamada ***__global__***. Essa palavra aplicada em função possui a semântica de que todo o código dela será compilado e funcionará na *GPU*, contudo, esse pode ser chamado pela *CPU*. Outra palavra importante é a ***__device__*** (linha 5), que especifica uma função, ou memória, que funciona na placa de vídeo e só pode ser chamada em funções que estejam na *GPU* também. Na linha 70 é chamada a função cosseno, a qual é do tipo ***__device__***. Além disso, existem tipos especiais de variáveis, como o da linha 8 (**float3**), que corresponde ao mesmo tipo do *shader HLSL*. O parâmetro *data* armazena as informações da onda.

As variáveis internas (*blockIdx*, *blockDim* e *threadIdx*) das linhas 15 e 16 são informações de relativas à distribuição de carga da *GPU*. O trecho de código das linhas 83 a 97 é relativo à parte de chamada de função *CUDA* por parte da *CPU*. Para a implementação do método de *FFT* é necessária uma função na *GPU* que compute a transformada inversa de Fourier. Sendo esse cálculo de suma importância para diversas áreas de estudo, a *NVidia* criou a biblioteca *CUFFT*, que disponibiliza as funções necessárias para a computação das transformadas inversas e diretas.

O modelo de programação da *CUFFT* é muito simular ao da FFTW (Frigo & Johnson, 2005), que é uma biblioteca muito utilizada para esse fim. O desenvolvimento dessa técnica divide-se em três partes, a primeira é a inicialização do $\tilde{h}_0(k)$, que pode ser realizada pela *CPU* na inicialização do sistema. A segunda parte consiste em atualizar $\tilde{h}(k,t)$, o que é feito pela função *CUDA* da Tabela 10.

**Tabela 10:** Atualização de $\tilde{h}(k,t)$

```
01 __global__ void spectrum_philips(float2* h0, float2 *ht, unsigned int width,
unsigned int height, float t, float patchSize)
02 {
03     unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
04     unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;
05     unsigned int i = y*width+x;
06
07     float2 k;
08     k.x = CUDART_PI_F * x / (float) patchSize;
09     k.y = 2.0f * CUDART_PI_F * y / (float) patchSize;
10
```

```
11      // dispersção w(k)
12      float k_len = sqrtf(k.x*k.x + k.y*k.y);
13      float w = sqrtf(GRAVITY * k_len);
14
15      float2 h0_k = h0[i];
16      float2 h0_mk = h0[(((height-1)-y)*width)+x];
17
18      float2 h_tilda=complex_add(complex_mult(h0_k,complex_exp(w*t)),
19                              complex_mult(conjugate(h0_mk),complex_exp(-w * t)) );
20
21      if ((x < width) && (y < height)) {
22          ht[i] = h_tilda;
23      }
24  }
```

A terceira parte é a computação da transformada inversa de Fourier, que é realizada pela função *CUDA* `cufftExecC2R`. Essa função recebe como parâmetros o plano de simulação, *buffer* do dado de entrada no espaço da frequência e o *buffer* do dado de saída no espaço do tempo. O plano de simulação é um conceito do *FFTW*, que consiste na configuração da forma de simulação. A criação é feita pela função `cufftPlan2d`, onde se especifica o tamanho do plano de simulação (no caso 2D é a largura e altura) e o que será realizado, ou seja, transformar-se-á do espaço da frequência para o tempo ou o inverso. A Tabela 11 exemplifica essas chamadas.

**Tabela 11:** Código para computação do Método da *FFT*

```
cufftHandle fftPlan;
CUFFT_SAFE_CALL(cufftPlan2d(&fftPlan, width, height, CUFFT_C2R) );

01 ...
02  spectrum_philips(d_h0, d_ht, fftInputW, fftInputH, t, patchSize);
03
04  void* g_hptr;
05  size_t pitch;
06  cutilSafeCallNoSync(cudaD3D9ResourceGetMappedPointer(&g_hptr,heightmap,0,0));
07  cutilSafeCallNoSync(cudaD3D9ResourceGetMappedPitch(&pitch,NULL,heightmap,0,0));
08
09  cufftSafeCall( cufftExecC2R(fftPlan,(cufftComplex *) d_ht, (float*)g_hptr) );
10
11  cudaD3D9UnmapResources(1, ppResources);
12  cutilCheckMsg("cudaD3D9UnmapResources(3) failed");
13 ...
```

Nota-se que os dados devem ser alocados na memória da placa gráfica usando a função `cudaMalloc` e desalocada usando `cudaFree`. A cópia do mapa $\tilde{h}_0(k)$ da *CPU* para a *GPU* é feita pela função `cudaMemcpy`. Essa sequência é a mesma discutida em 4.2.1.4. Após isso, pode-se utilizar mais *IFFTs* para a geração dos mapas de normais, ou pode ser

usado o cálculo com diferenças, sabendo dos problemas que o mesmo possui relativo à amostragem. A Tabela 12 mostra esse código.

**Tabela 12:** Código da geração de normais via diferenças

```
01 __global__ void normalmap_fft_diff(float* h, float2 *normalmap, unsigned int
width, unsigned int height)
02 {
03     unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
04     unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;
05     unsigned int i = y*width+x;
06
07     float2 normal;
08     if ((x > 0) && (y > 0) && (x < width-1) && (y < height-1)) {
09         normal.x = h[i+1] - h[i-1];
10         normal.y = h[i+width] - h[i-width];
11     } else {
12         normal = make_float2(0.0f, 0.0f);
13     }
14     normalmap[i] = normal;
15 }
```

No código percebe-se que não é realizada a computação efetiva da normal, apenas a diferença das alturas. Isso é feito para reduzir o consumo de memória. A técnica de *Perlin Noise* segue um funcionamento similar à de *FFT*, no entanto, não utiliza a biblioteca *CUFFT*. No caso da implementação em *CUDA*, aloca-se uma textura com números randômicos e cria-se um código *CUDA* para atualizar outra textura de saída. A Tabela 13 apresenta a versão da técnica de *Perlin Noise* usando *Shaders*, no caso, em linguagem *HLSL*.

**Tabela 13:** Código HLSL da técnica Perlin Noise

```
01 texture  g_txTexture;
02 float Overcast;
03 float Time;
04
05 sampler2D  g_samTexture =
06 sampler_state
07 {
08     Texture = <g_txTexture>;
09     MinFilter = Linear;
10     MagFilter = Linear;
11     MipFilter = Linear;
12     AddressU  = mirror;
13     AddressV  = mirror;
14 };
15
16 void PerlinVS( in  float4 iPos      : POSITION,
17                in  float2 iTexCoord : TEXCOORD0,
18                out float4 oPos      : POSITION,
19                out float2 oTexCoord : TEXCOORD0 )
20 {
21     oTexCoord = iTexCoord;
22     oPos = iPos;
23 }
24
25 void PerlinFS( in  float3 iTexCoord : TEXCOORD0,
26                out float4 oColor      : COLOR0 )
27 {
```

```
28
29      float2 move = float2(0.0,1.0); // Direção
30
31      float4 perlin = tex2D(Texture, (iTexCoord.xy)+Time*move)/2.0;
32      perlin += tex2D(Texture, (iTexCoord.xy)*2.0+Time*move)/4.0;
33      perlin += tex2D(Texture, (iTexCoord.xy)*4.0+Time*move)/8.0;
34      perlin += tex2D(Texture, (iTexCoord.xy)*8.0+Time*move)/16.0;
35      perlin += tex2D(Texture, (iTexCoord.xy)*16.0+Time*move)/32.0;
36      perlin += tex2D(Texture, (iTexCoord.xy)*32.0+Time*move)/32.0;
37
38      oColor.rgb = 1.0 - pow(perlin.r, Overcast) * 2.0;
39      oColor.a = 1.0;
40 }
41
42 technique OceanPerlin
43 {
44     pass P0
45     {
46         VertexShader = compile vs_1_1 PerlinVS();
47         PixelShader = compile ps_2_0 PerlinFS();
48     }
49 }
```

Assim, pode-se especificar um fluxo de execução geral do motor, em pseudo-linguagem, como o apresentado pela Tabela 14. Nele é especificado o procedimento passo a passo do funcionamento do sistema.

**Tabela 14:** Fluxograma de Execução

```
01 Inicializar Contexto DirectX
02   Criar Textura para a reflexão
03   Criar Textura para a refração
04   Criar Framebuffer para renderização offscreen
05 Loop() {
06   Atualizar Câmera
07   Gerar Mapa de Reflexão
08     Refletir a posição e alvo da câmera
09     Habilitar o plano de corte
10     Habilitar Framebuffer
11     Atualizar framebuffer para textura de reflexão
12     Renderizar cena
13     Desabilitar Framebuffer
14    Salva Matriz de View
15    Gerar Mapa de Refração
16     Habilitar o plano de corte
17     Habilitar Framebuffer
18     Atualizar framebuffer para textura de refração
19     Renderizar cena
20     Desabilitar Framebuffer
21   Posicionar Câmera e atualizar Frustum
22   Habilitar Programa de Céu
23   Desenhar Céu
24   Habilitar Programa de Objetos
25   Desenhar Objetos
26   Executar Simulação de Ondas (se utilizar CUDA, Inicializar CUDA)
27   Habilitar Programa de Oceano  (Iluminação)
28   Atualizar variáveis e texturas
```

```
29     Atualiza matrizes de reflexão, refração e shadow
30   Desenhar Oceano
31 }
```

Um *shader* para iluminação dos fragmentos está disponível no Anexo 2. É importante notar que para a visualização realística ainda é necessária a iluminação correta do céu.

Assim, finaliza-se a análise do desenvolvimento do motor utilizado para os testes e inicia-se a especificação e análise dos resultados.