7.
Resultados

A análise de desempenho, para melhor descrição dos resultados, se dividiu em cinco categorias:

- Testes de Simulação na *CPU*, que serão usados apenas para justificar o uso da *GPU*.
- Testes de Simulação na *GPU*
- Testes de Variação de Harmônicas de Gerstner
- Testes de Nível de Detalhe, que utilizarão dados pré-computados.
- Testes Totais, que correspondem à utilização de cada técnica de simulação de *GPU* (em sua melhor implementação, ou seja, usando *CUDA* ou *Shaders*) com cada técnica de nível de detalhe

O parâmetro de comparação de performance foi o tempo de computação, expresso em milissegundos.

Após a apresentação e análise dos testes, faz-se a análise de complexidade e viabilidade, que tem como objetivos, explicitar os pontos fortes e fracos de cada técnica, bem como, mostrar a viabilidade do uso de cada uma em cinco contextos de aplicação: Computadores Classe 1, Computadores Classe 2, *Notebooks*, *Netbooks* e *Smartphones*.

7.1. Metodologia de Teste de Desempenho

Para a realização dos testes foi desenvolvido um software auxiliar que utiliza o núcleo de simulação da *OceanEngine*. Nele, todos os testes utilizam a visualização em tela cheia (800x600) e a geração dos dados de análise inicia-se aos 15 s depois da inicialização do sistema (para evitar

que problemas de carga de dados interfiram no resultado) e persiste até que sejam obtidas, no mínimo, 300 amostras do tempo de computação. Sendo que alguns algoritmos são extremamente rápidos, chegando a gerar 150 amostras por segundo, limita-se o tempo de execução a, no mínimo, 60 segundos. A Figura 42 mostra o sistema de análise de simulação em funcionamento.

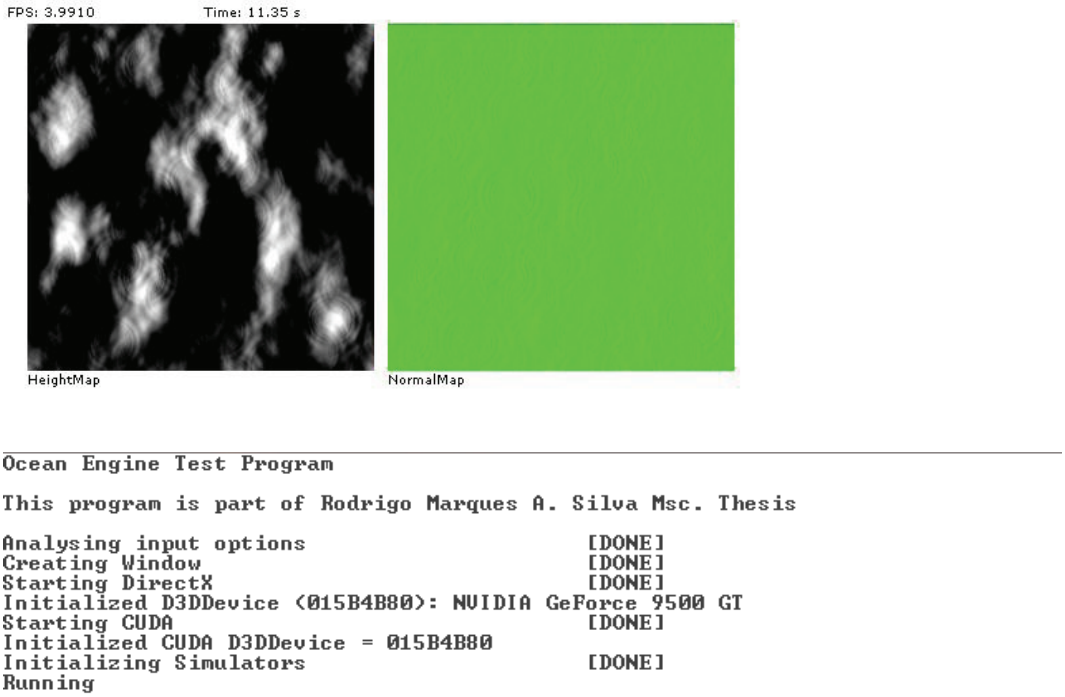


Figura 42: Sistema de teste de simulação

Esse sistema possui uma interface de entrada de linha de comando, a qual permite a criação de arquivos de lote (.bat) para a computação de vários testes. A Figura 43 mostra a interface de entrada desse sistema, disponível pelo comando `oceantest -h`.

```
Ocean Engine Test Program

This program is part of Rodrigo Marques A. Silva Msc. Thesis

Analysing input options

usage: oceantest [-opts?[=value]]

        -size=[integer],[integer]       Texture size(width, height)
        -cores=[integer]        Number of CPU cores
        -waves=[integer]        Number of Gerstner Waves
        -time=[integer] Time to simulate
        -mode=[string]  Silumation Mode: CUDA, CPU, GPGPU
        -heightmap=[string]     Heightmap synthesis technique: Gerstner, Perlin
FFT
        -normalmap=[string]     Normalmap synthesis technique: Gerstner, Perlin
FFT
        -output=[string]        Output filename
        -save   Save the images in DDS file format
        -h      Show this help
```

Figura 43: Opções de linha de comando do sistema de simulação

Na simulação de nível de detalhe, a câmera segue um caminho circular fixo, definido por um arquivo *XML*. Esse arquivo define uma série de pontos no espaço, os quais a câmera deve passar. Sendo assim, a direção da câmera é definida pelo versor entre dois pontos consecutivos e a posição da mesma é interpolada linearmente.

Após a geração dos dados, o sistema armazena-os no formato *CSV.* Devido à grande quantidade de dados a serem analisados, criou-se outro programa a fazer a análise estatística. Esse faz a análise de tempo de computação mínimo, máximo, médio, desvio padrão e intervalo de confiança.

A fim de reduzir espaço na tabela, o nome das técnicas está codificado no seguinte padrão:

<g,f,p>-<h,n>-<Texto>.

O conjunto <g,f,p> se refere à técnica Gerstner, *FFT* e *Perlin*, respectivamente. Já o conjunto <h,n> se refere ao estágio de geração do *Height Map* ou *Normal Map*, repectivamente. O termo texto é uma explicação.

O hardware utilizado para os testes foi um computador com 4 Gbytes de memória RAM, CPU Intel Core 2 Quad @ 2.4 GHz e placa de

vídeo de NVidia 9500 GT 128 bits com 256 Mbytes dedicados. Os testes foram realizados com o sistema operacional Microsoft Windows 7 recém instalado, a fim de reduzir ao máximo as trocas de contextos e intromissões de outros aplicativos. Além disso, o sistema de teste foi inicializado com prioridade de tempo real (a mais alta disponível aos usuários de sistema Microsoft Windows). O sincronismo vertical foi desligado para a correta computação dos intervalos de tempo.

Nos testes gerais foram usadas 10 harmônicas para o algoritmo de Gerstner e as mesmas possuem os parâmetros definidos aleatoriamente. Os testes que utilizam *CUDA* foram configurados para um número fixo de 256 *threads*. A placa de vídeo utilizada possui 32 unidades de processamento (*stream processors*). A técnica de Gerstner com *CUDA* utilizou a memória constante para armazenamento das informações das harmônicas, pois, essa memória proporcionou o melhor desempenho.

Por fim, cada teste foi repetido 3 vezes para avaliar a variação dos parâmetros. O último teste realizado foi o considerado na análise deste capítulo. Cada rodada de testes durou cerca de 5 horas e 30 minutos, sendo realizados cerca de 480 testes por rodada.

Outro software de teste de desempenho é o próprio NVidia Profiler, que é capaz de analisar dados de banda de transferência de dados. Contudo, por se tratar de um estudo comparativo de desempenho para técnicas usando shaders e CUDA , precebe-se que o NVidia Profiler adiciona um vício na análise, dessa forma, os resultados do mesmo não são apresentados neste trabalho.

7.2. Metodologia de Análise Estatística

A análise estatística realizada utiliza à média aritmética (Equação 34) como estimador de média, pois essa é não viciada (ou viezada) e consistente. O estimador de variância (desvio-padrão) escolhido foi a variância amostral (Equação 35), por ser não viciada e consistente.

$$\mu = \bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$$

Equação 34: Estimador de media

$$\sigma^2 = S^2 = \frac{1}{n-1}\sum_{i=1}^{n} (X_i - \bar{X})^2$$

Equação 35: Estimador de variância

Para a comprovação de validade de dados, foi utilizado o nível de significância de 5%, o qual é padrão utilizado pela indústria médica, portanto de alta confiabilidade. Para a composição do intervalo de confiança foi utilizada a curva *t-student* (Equação 36), devido à forma de estimação dos parâmetros μ e σ.

$$IC(\mu, (1-\alpha)\%) = \bar{X} \pm Z_{n,\frac{\alpha}{2}} \frac{S}{\sqrt{n}}$$

Equação 36: Cálculo do intervalo de confiança

Para evitar uma poluição visual nas tabelas apresentadas neste capítulo, optou-se por colocar as tabelas completas (com média, desvio-padrão e intervalo de confiança) no Anexo 1. Dessa maneira, as tabelas deste capítulo possuem apenas a média da informação observada.

7.3. Testes de Simulação na *CPU*

A tabela a seguir mostra o desempenho obtido com as técnicas em *single core* e *multi core* na *CPU*, usando a biblioteca *OpenMP*, variando o tamanho da grade de simulação. Para a aplicação da *IFFT* foi utilizada a biblioteca *FFTW* (Frigo & Johnson, 2005) na versão 3.3alpha1 com suporte *multi-thread*. No total, realizou-se 90 testes para a *CPU*, gerando um conjunto mínimo de 27.000 amostras. Na Tabela 15 as linhas correspondem à técnica e o número de núcleos de *CPU* utilizados, as colunas ao tamanho da textura utilizada e o valor de cada célula é o tempo de processamento em milissegundos.

Tabela 15: Resultados da Simulação na *CPU*

Técnica	128	256	512	1024	2048
g-h–CPU-1 core	17.682	70.556	282.360	1132.367	4557.846
g-n–CPU-1 core	37.052	142.150	568.729	2278.132	9140.469
g-h–CPU-2 core	8.917	35.536	141.420	565.790	2271.970
g-n–CPU-2 core	17.874	71.297	284.847	1139.385	4566.923
g-h–CPU-4 core	4.684	19.486	73.207	289.497	1233.289
g-n–CPU-4 core	9.329	37.076	146.123	584.344	2341.325
f-h–CPU-1 core	11.162	44.883	180.676	743.688	3170.146
f-n–CPU-1 core	6.928	31.040	135.684	612.019	2861.218
f-h–CPU-2 core	8.589	35.213	143.043	594.510	2519.148
f-n–CPU-2 core	6.773	30.685	131.151	585.051	2780.462
f-h–CPU-4 core	7.685	30.843	126.599	527.194	2252.764
f-n–CPU-4 core	6.871	30.102	129.631	579.810	2771.730
p-h–CPU-1 core	1.337	4.173	16.369	66.223	344.217
p-n–CPU-1 core	8.190	31.660	125.270	493.840	2077.144
p-h–CPU-2 core	0.860	2.533	9.337	36.831	221.365
p-n–CPU-2 core	4.115	16.236	64.423	256.088	1117.289
p-h–CPU-4 core	0.629	1.676	5.939	22.599	163.268
p-n–CPU-4 core	2.524	9.728	38.843	152.427	685.652

Devido à grande variação de valores, produziu-se o gráfico da Figura 44 em escala logarítmica de base 10.

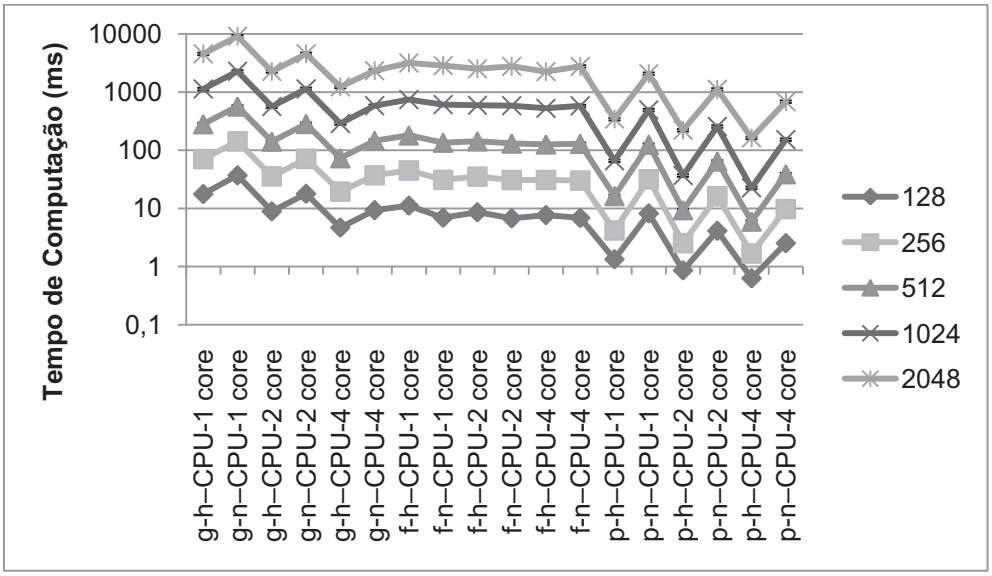


Figura 44: Gráfico de desempenho da Simulação na *CPU*

No gráfico da Figura 44 pode-se ressaltar várias observações dos testes. Como o intervalo de confiança é bastante estreito, as afirmações sobre os valores médios são suficientes para a prova da mesma (com nível de significância de 5%).

Primeiro, observa-se que o cálculo do mapa de normais é mais custoso que o de altura para o método de Gerstner (cerca de 2 vezes

mais), isso é evidente, pois, a computação da normal demanda o cálculo da posição. Se a mesma fosse passada como parâmetro para o gerador de normais, haveria uma redução considerável na computação das normais, no entanto, para manter a independência entre a síntese do mapa de normais e do mapa de altura, isso não pode ser aplicado. A mesma observação ocorre no método de Perlin. No entanto, isso não é observado no método de *FFT*, isso porque, o plano de simulação da biblioteca *FFTW* otimiza as chamadas sobre um mesmo plano, aproveitando a simulação realizada para o mapa de altura.

Pode-se também observar que, o aumento do número de núcleos reduz o tempo de processamento, o que, também é esperado, devido à mescla de computação e acesso à memória, principalmente, do método de Gerstner. Porém, a simulação de *FFT* não obteve um ganho significativo com o aumento do número de cores. A explicação desse fato passa por três aspectos: o primeiro é em relação à biblioteca *FFTW*, pois a versão utilizada ainda não é totalmente paralela, mesmo sendo descrita como paralela pela documentação.

Outro aspecto remete à análise de banda no barramento Memória – *CPU*, pois, sendo a computação da *IFFT* um processo que demanda muito acesso à memória, a banda pode limitar o desempenho, ainda mais se essa for forçada a transmitir dados para vários núcleos. A *CPU* utilizada possui dois canais de dados para a memória *RAM*, dessa forma, 2 núcleos podem comunicar paralelamente com a mesma, sem que haja interrupção pelo árbitro do barramento. Na mesma linha de análise, para o último aspecto, pode-se, em último caso, analisar os "*misses*" no cachê L2 do processador (funcionalidade disponível nas versões *Team Edition* do *Microsoft Visual Studio*).

Por fim, existe uma grande redução do desempenho com o aumento do tamanho da textura a ser sintetizada. Isso pode ser parcialmente explicado pelo acréscimo quadrático no número de células (ao se dobrar o tamanho o desempenho reduz-se pelo fator de quatro

vezes). Outros motivos são os relacionados ao acesso de memória e de banda de barramento.

7.4. Testes de Simulação na *GPU*

A tabela a seguir mostra o desempenho obtido com as técnicas usando a arquitetura de *shader* e a biblioteca *CUDA*, variando o tamanho da grade de simulação. Observa-se que no caso do *FFT* não foi desenvolvida a técnica usando *Shader*, conforme discutido previamente. Assim, realizou-se 50 testes, compondo um grupo mínimo de 15.000 amostras. Na Tabela 16 as linhas correspondem à técnica e modo de simulação utilizado (*CUDA* ou *Shader*), as colunas ao tamanho da textura utilizada e o valor de cada célula é o tempo de processamento em milissegundos.

Tabela 16: Resultados da Simulação na GPU

Técnica	128	256	512	1024	2048
g-h–GPU-CUDA	1.344	2.881	8.501	28.400	108.480
g-n–GPU-CUDA	1.018	3.266	12.033	46.454	182.245
g-h–GPU-Shader	0.237	0.460	1.715	5.914	23.085
g-n–GPU-Shader	0.321	0.468	1.910	6.729	25.044
f-h–GPU-CUDA	1.458	3.118	9.891	33.354	165.830
f-n–GPU-CUDA	0.774	2.115	7.611	28.296	112.307
p-h–GPU-Shader	0.234	0.571	1.718	5.913	22.048
p-n–GPU-Shader	0.327	0.647	1.770	5.953	22.023
p-h–GPU-CUDA	0.759	1.872	6.225	22.280	87.218
p-n–GPU-CUDA	0.763	1.871	6.215	22.273	86.597

Para uma melhor análise dos dados, utilizam-se os gráficos da Figura 45 e da Figura 46. Essa última é construída em escala logarítmica de base 10 para visualizar a variação, visto que, há uma grande discrepância de escala entre os métodos.

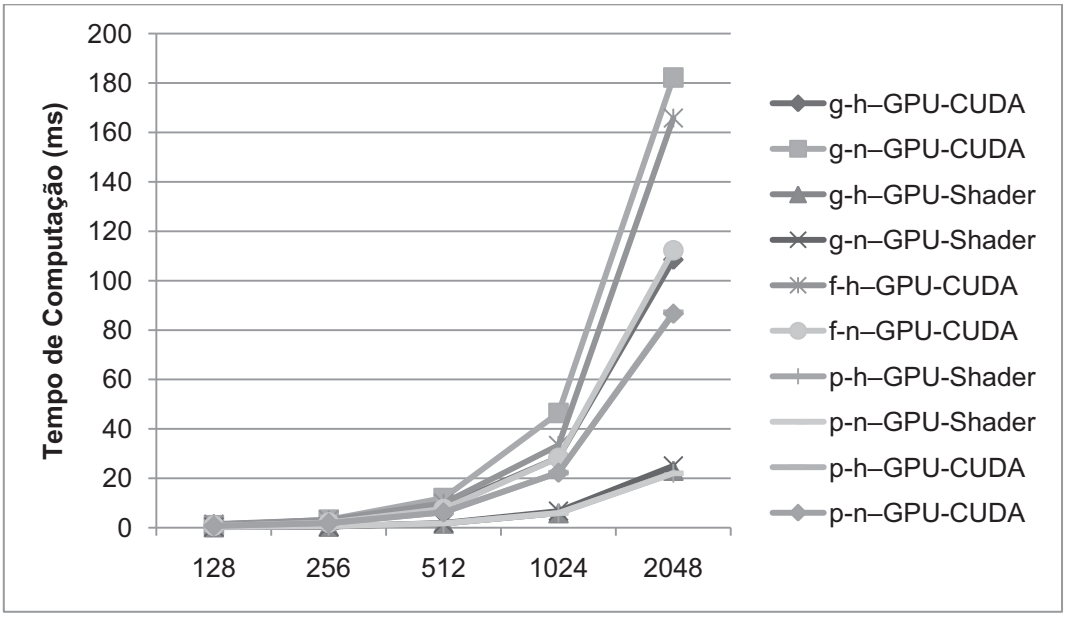


Figura 45: Gráfico de desempenho da Simulação na GPU

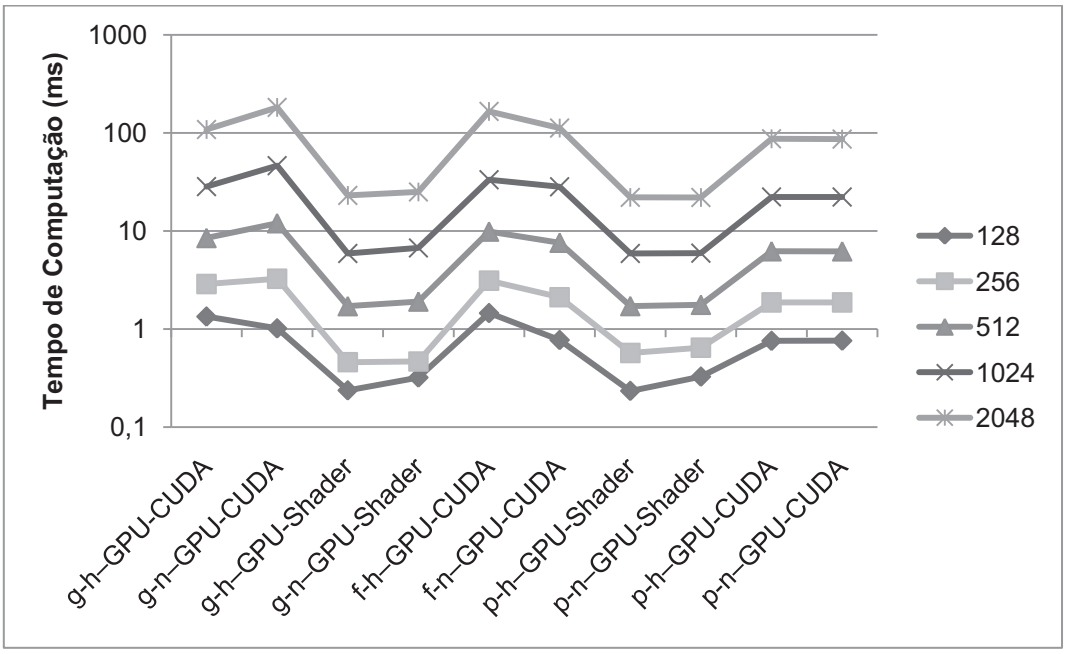


Figura 46: Gráfico de desempenho da Simulação na GPU em escala logarítmica

Percebe-se facilmente que o algoritmo de Gerstner utilizando *CUDA* possui o pior desempenho, a discussão desse fato é realizada no item 7.5, por se tratar da análise mais aprofundada da técnica de Gerstner.

Outro ponto interessante é a análise do algoritmo de *Perlin Noise*, visto que a técnica de *Shader* e a usando *CUDA* utilizam a mesma arquitetura de simulação e de memória, pois, ambos usam a memória da textura para gerar a simulação. Sendo assim, é esperado que ambos possuam o desempenho muito similar, entretanto, a técnica usando *shaders* é cerca de 4 vezes mais eficiente (em relação à técnica usando 4 núcleos de *CPU*, o ganho chega a 126 vezes para um textura de 2048 x 2048), e tendo em vista a significância de 5%, percebe-se que há uma vantagem significativa no uso dos *shaders*.

A explicação para tal situação decorre do fato que a placa de vídeo é otimizada para a renderização, dessa forma, as *APIs* gráficas tendem a melhor escalonar os recursos do hardware disponíveis. Além disso, o pixel shader possui a restrição de só permitir a gravação de dados em apenas uma posição específica da memória, podendo acessar dados de qualquer outra, esse conceito é chamado de *Gather*. Isso permite uma série de aprimoramentos para ganho de desempenho, dessa forma, *CUDA* sai perdendo, pois, permite ler e escrever em qualquer posição de memória, sendo portanto um *Gather* e um *Scatter* (grava em qualquer posição).

Além disso, o desempenho do algoritmo de *FFT* com o uso de *CUDA* é muito superior ao da *CPU*, mesmo com 4 núcleos, pois, o ganho é de quase 17 vezes para uma textura de 2048 x 2048. Isso se deve à dois fatores: primeiramente à grande robutez e otimização do algoritmo de *IFFT* na biblioteca *CUFFT* da NVidia e, ao baixo nível de paralelismo da biblioteca *FFTW*.

Por fim, notoriamente a técnicas de Gerstner e *Perlin Noise* usando *Shaders* são superiores, em termos de desempenho, à de *FFT*, tendo um ganho de, aproximadamente, 6 e 7 vezes, respectivamente. O algoritmo de *Perlin Noise* é cerca de 1.10 vezes mais rápido que o de Gerstner com 10 harmônicas, mesmo considerando o intervalo de confiança.

7.5. Testes de Variação de Harmônicas de Gerstner

A tabela a seguir mostra o desempenho obtido com a variação do número de harmônicas da técnica de Gerstner com um mapa de 256 x 256. Nesse caso, o tempo total de computação entre a geração do mapa de altura e do mapa de normais é utilizado. Desse modo, realizou-se 200 testes (*heightmap* e *normalmap*), compondo um grupo mínimo de 60.000 amostras. Na Tabela 17 as linhas correspondem ao número de harmônicas, as colunas ao modo de simulação utilizada e o valor de cada célula é o tempo de processamento em milissegundos.

Tabela 17: Resultados da Variação do Número de Harmônicas de Gerstner

	CPU (1 core)	CPU (2 core)	CPU (4 core)	CUDA	Shader
1	25.658	11.929	6.326	3.395	1.185
2	44.844	22.463	11.864	3.553	1.185
3	65.720	33.436	17.270	3.772	1.193
4	86.603	44.028	22.917	4.058	2.684
5	107.736	54.905	28.583	4.376	3.102
6	128.668	64.661	33.837	4.749	3.574
7	149.836	75.177	39.373	5.074	3.893
8	180.718	86.608	44.815	5.460	4.273
9	191.936	96.248	50.335	5.773	4.757
10	212.680	106.736	55.908	6.150	5.136
11	233.901	117.343	62.454	6.483	5.604
12	254.682	131.893	66.676	6.849	6.008
13	275.897	138.689	71.965	7.204	6.463
14	296.578	152.398	78.622	7.580	6.932
15	317.940	159.841	82.954	7.835	7.369
16	339.217	174.960	88.427	8.202	7.510
17	359.803	180.999	93.979	8.541	8.219
18	380.876	196.670	99.335	8.917	12.158
19	401.749	201.996	104.874	9.233	9.129
20	422.811	212.678	110.525	9.583	9.359

De forma similar, utiliza-se os gráficos da Figura 47 e da Figura 48 para uma melhor análise dos dados, sendo que, a última é construída em escala logarítmica de base 10 para destacar as variações de performance dos métodos.

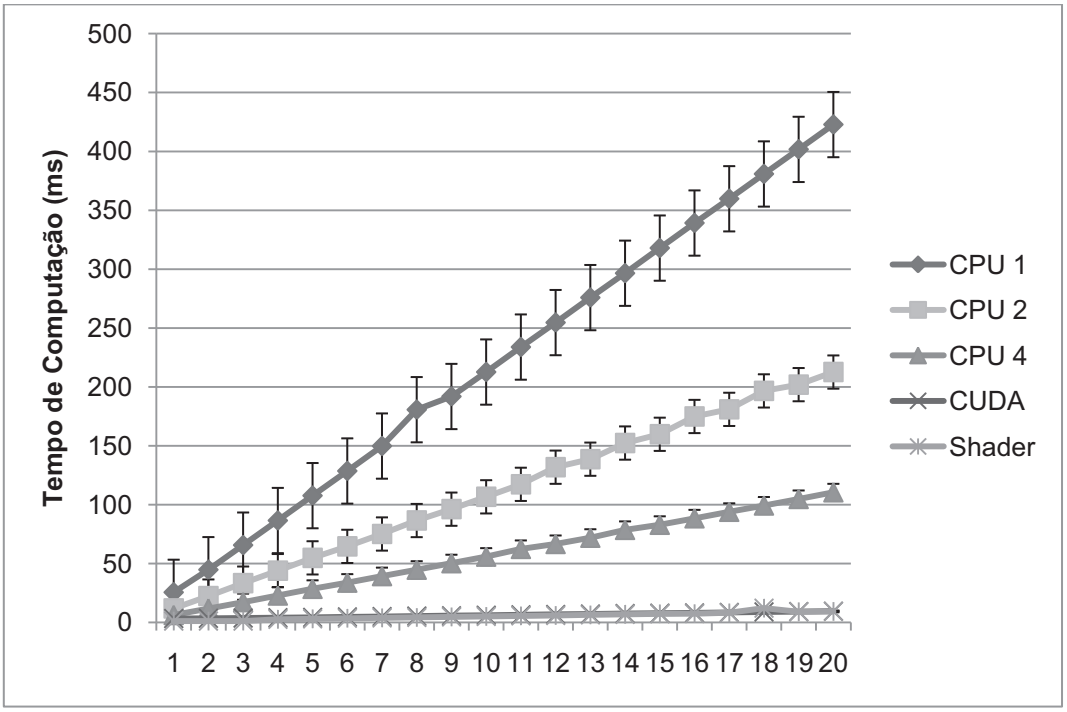


Figura 47: Gráfico de desempenho da Variação do Número de Harmônicas de Gerstner

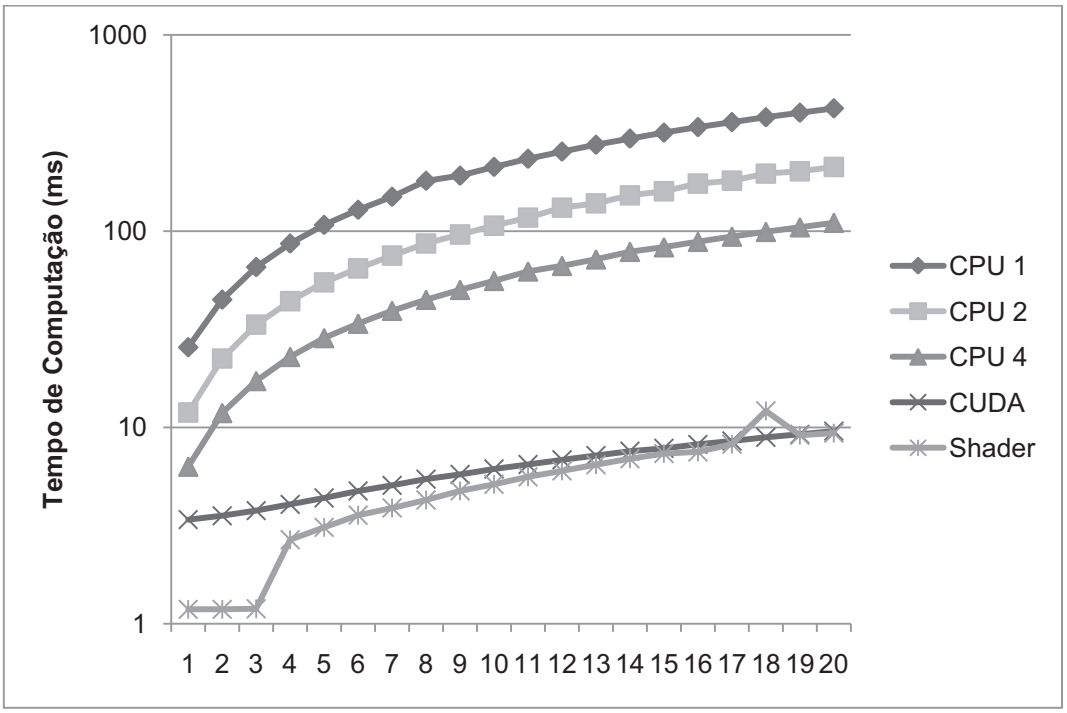


Figura 48: Gráfico de desempenho da Variação do Número de Harmônicas de Gerstner em escala logarítmica

A Figura 47 e a Figura 48 mostram o decaimento da performance com o aumento do número de harmônicas. De forma similar às análises

anteriores, devido à pequena variação no intervalo de confiança, pode-se fazer uma afirmação com base apenas na média (esperança estatística) do dado.

A Figura 48 deixa absolutamente claro que a técnica de Gerstner usando *Shaders* possui o maior desempenho, contudo, a mesma técnica utilizando *CUDA*, tende a apresentar o mesmo desempenho com o aumento do número de harmônica para grade de 256 x 256. A diferença inicial se dá, pois, as funções de mapeamento e ativação dos recursos *CUDA* consomem certa quantidade de tempo, tornando-se irrelevantes no tempo total para valores maiores de harmônicas.

No entanto, de acordo com o gráfico da Figura 45 a técnica usando *CUDA* perde consideravelemente o poder de processamento com o aumento da grade de simulação, isso se deve ao fato de que quanto maior a grade a ser simulada, é necessária a criação de mais blocos e *threads CUDA*, reduzindo assim a eficiência do acesso de memória constante. Além disso, observa-se um aumento abrupto no tempo de computação com *shader* na interface entre 3 e 4 harmônicas, isso pode ser originado por uma otimização de *hardware* quanto ao número de intruções do *shader*. Observa-se ainda um pequeno crescimento com o aumento do número de harmônicas. As técnicas usando a *CPU* logo apresentam um rendimento baixo.

Para analisar o comportanento do algoritmo de Gerstner com outros tipos de memória *CUDA*, realizou-se mais 120 testes, no quais, fez-se a armazenagem das informações das harmônicas na memória Global, Local copiando da Global e Local copiando da Constante. Os resultados são apresentados na Tabela 18.

Tabela 18: Resultados de Gerstner com a troca de memória na *GPU CUDA*

	Global	**Constante**	**Local - Global**	**Local - Constante**
1	8.290	3.395	4.690	4.059
2	13.081	3.553	5.971	4.650
3	17.675	3.772	7.332	5.316

4	22.296	4.058	8.525	6.009
5	27.150	4.376	9.825	6.687
6	31.533	4.749	11.089	7.396
7	36.117	5.074	12.272	7.909
8	40.731	5.460	13.487	8.664
9	45.350	5.773	14.887	9.237
10	46.554	6.150	16.142	9.866
11	50.435	6.483	17.360	10.530
12	54.906	6.849	18.607	11.153
13	59.426	7.204	19.855	11.813
14	63.905	7.580	21.110	12.379
15	68.384	7.835	22.336	13.002
16	72.884	8.202	23.571	13.643
17	77.436	8.541	24.835	14.361
18	81.895	8.917	26.081	15.030
19	86.960	9.233	27.260	15.741
20	90.790	9.583	28.488	16.139

Novamente, para se analisar o comportamento dessas opções e comparar com o melhor resultado em *CPU* (4 núcleos) utiliza-se os gráficos da Figura 51 e Figura 52.

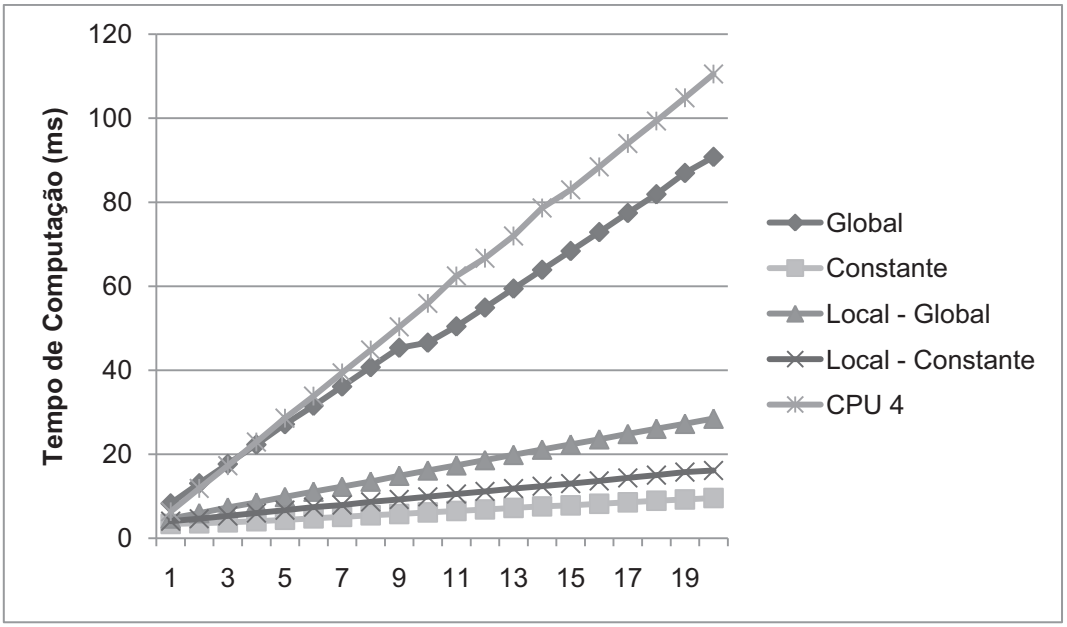


Figura 49: Gráfico de desempenho de Gerstner com a troca de memória na *GPU CUDA*

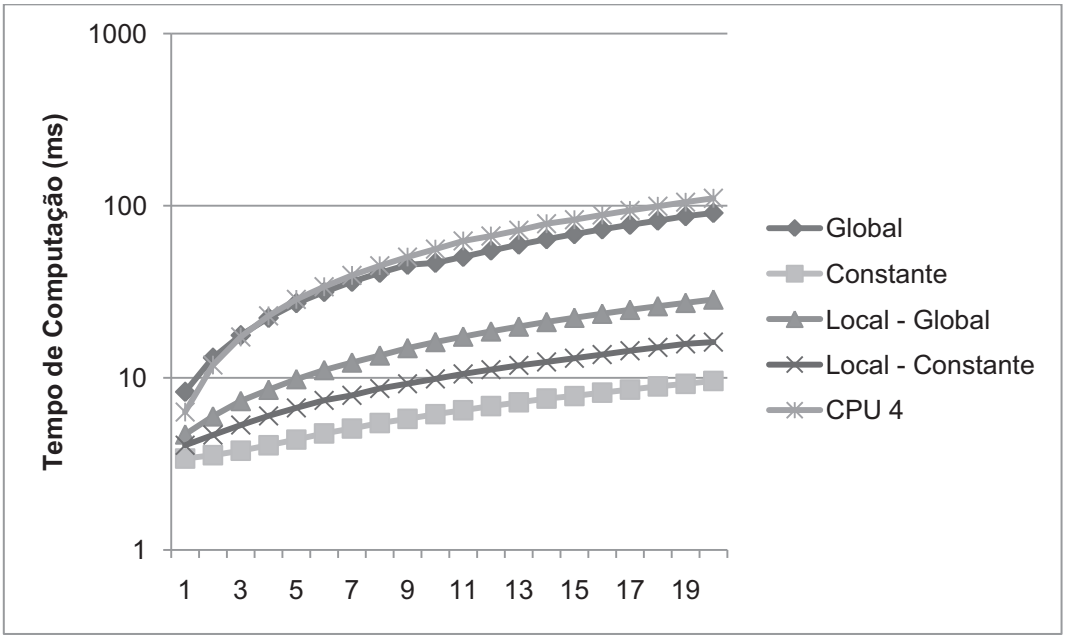


Figura 50: Gráfico de desempenho de Gerstner com a troca de memória na *GPU CUDA* em escala logarítmica

Os gráficos mostram que a memória global é muito mais lenta que as demais. Isso é evidente, pois, essa memória permite escrita e leitura de qualquer parte da placa, ao contrário da constante, que apesar de ter acesso global, não permite escrita. A memória constante ainda possui uma otimização de ser compartilhada dentro de um bloco de processamento, dessa forma, reduzindo a latência.

A memória local também é rápida, mesmo com o acesso de cópia sendo global. No entanto, essa perde para a constante devido à necessidade de sincronismo de dados e cópia local. Tal processo é otimizado em hardware para a memória constante. Observa-se que o código da Tabela 9 impõe que cada *thread* leia e armazene na memória local parte do vetor de informações. A quantidade é dependente do tamanho do bloco e do número de harmônicas utilizadas. A memória constante chega a ter um ganho de eficência de cerca de 9 vezes o da memória global. O problema da memória constante é o limite de tamanho de armazenamento de dados.

Um aspecto curioso é o observado pelas curvas da técnica com o uso de 4 núcleos de *CPU* e com o uso de *CUDA* com memória Global. A técnica usando *CUDA* com memória Global só ultrapassa a *CPU* com 4 núcleos quando a simulação utiliza 4 harmônicas. No entanto, observa-se que a técnica usando *CUDA* com memória Global não se afasta consideravelmente da curva da *CPU* com 4 núcleos. A explicação para esse fato se deve ao demasiado acesso à matriz de dados `data`, que é parâmetro da função *CUDA* para a simulação e, está armazenada na memória global *CUDA*, que é a mais lenta em termos de latência de acesso.

Assim, deve-se sempre analisar alternativas para reduzir o número de acessos, utilizar uma memória mais rápida (ou mais adequada), p.e. memória local, constante ou de textura, ou até mesmo mudar a posição de acessos no código. A própria *NVidia* (2009) mostra que o desempenho é fortemente influenciado pelo chaveamento entre tempo de computação e tempo de acesso a dados.

Uma observação importante em relação à técnica de Gerstner, é que quando a mesma é usada numa simulação para uma textura, pode-se limitar o tamanho da textura de acordo com as frequências utilizadas, pois, é possível calcular o tamanho de textura necessária para acomodar 1 período de simulação, assim, reduzindo o consumo de memória. Depois, pode-se utilizar o sistema de repetição de textura para o resto da dimensão do oceano. Desse modo, o uso do tamanho 256 x 256 é meramente para fixar a quantidade de fragmentos e elementos a serem simulados.

Por fim, da mesma forma como explicado em 7.4, a solução usando *Shader* é superior à usando *CUDA*.

7.6. Testes de Nível de Detalhe

A tabelas a seguir mostram o desempenho obtido com as técnicas de nível de detalhe usando dados pré-computados, variando o tamanho da

grade de visualização. Na Tabela 20 as linhas correspondem à técnica empregada, as colunas ao tamanho da grade utilizada e o valor de cada célula é o tempo de processamento em milissegundos.

Tabela 19: Resultados das Técnicas de nível de detalhe

Técnica	256	512	1024	2048
GeoClipMap	7.821	11.242	27.521	67.153
Radial LoD	6.256	8.139	14.083	47.611
Projected Grid	7.489	10.024	15.469	49.048

Observando os dados da Tabela 19 pode-se obter o seguinte o gráfico da Figura 51.

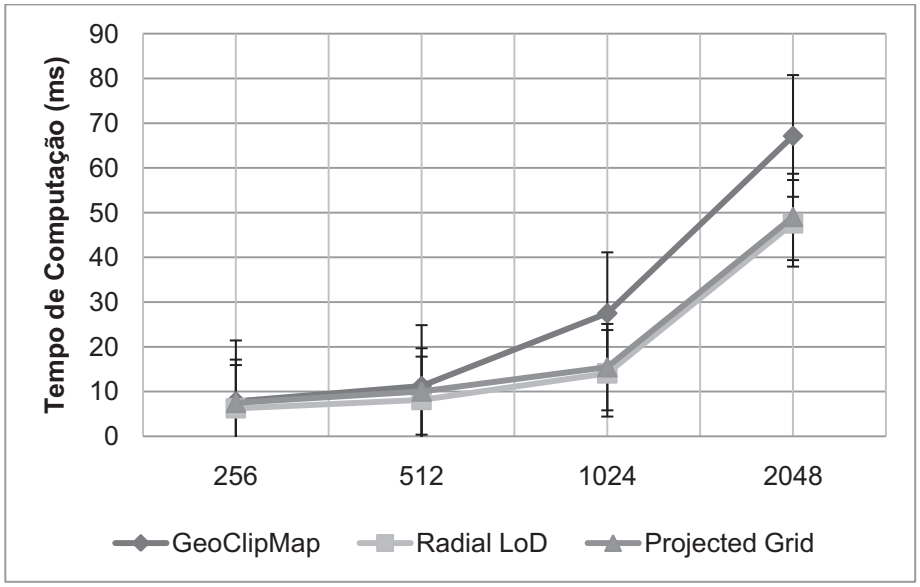


Figura 51: Gráfico de desempenho dos algoritmos de *LoD*

O gráfico mostra a clara perda de desempenho com o aumento do tamanho da grade de vértices para visualização. Tal fato é totalmente esperado, visto que, o aumento do número de vértices aumenta o consumo dos processadores alocados para o programa de vértice. No entanto, o algoritmo de *GeoClipMap* apresenta o pior desempenho, pois, com a significância de 5%, pode-se afirmar que isso ocorre, devido ao distanciamento do intervalo de confiança. O algoritmo de *GeoClipMap* apresenta o maior desvio-padrão, isso se deve às técnicas de *culling*, que reduzem abruptamente o número de vértices a serem processados.

Os algoritmos de *Radial LoD* e *Projected Grid* possuem uma eficiência muito similar. A proximidade da eficiência se deve à similaridade com que os vértices são espalhados, pois, ambos distribuem os vértices desde o *Near Plane* até o *Far Plane*, assim produzindo um número muito próximo de fragmentos. Dessa maneira, a diferença está apenas na forma com que os vértices são reposicionados.

7.7. Testes Totais

A tabela a seguir mostra o desempenho obtido com a combinação das técnicas de simulação, iluminação e nível de detalhe. Para as técnicas de simulação, selecionou-se a implementação com melhor desempenho. Na Tabela 20 as linhas correspondem à combinação de técnicas utilizada, as colunas ao tamanho da textura simulação e grade de visualização empregada e o valor de cada célula é o tempo de processamento em milissegundos.

Tabela 20: Resultados dos Testes Totais

Técnica	256	512	1024	2048
GeoClipMap & g-h-Shader	10.0844	15.3741	42.5898	121.2740
GeoClipMap & f-h-CUDA	15.5197	31.6850	92.2242	345.6890
GeoClipMap & p-h-Shader	9.6188	15.9353	40.7100	113.4721
Radial LoD & g-h-Shader	9.4468	13.1619	27.2464	100.6234
Radial LoD & f-h-CUDA	12.4901	28.6454	81.1687	331.8028
Radial LoD & p-h-Shader	7.7671	13.1704	29.7498	96.4543
Projected Grid & g-h-Shader	9.5551	13.7497	33.4389	98.2274
Projected Grid & f-h-CUDA	13.0011	29.1340	79.9805	333.8876
Projected Grid & p-h-Shader	11.0204	17.2300	33.4867	98.2851

A fim de prover uma melhor interpretação dos resultados, utiliza-se o gráfico da Figura 52. Nele, as colunas são agrupadas por tamaho de textura e grade. A ordem das colunas é a mesma da legenda.

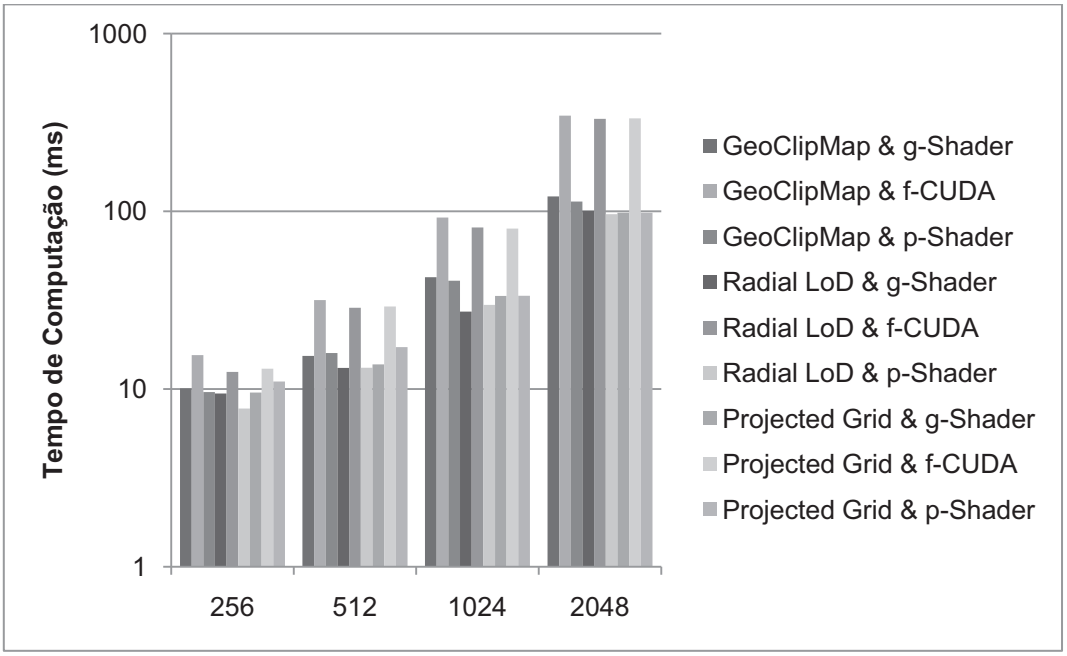


Figura 52: Gráfico de desempenho da combinação dos algoritmos

O resultado foi como o esperado, as técnicas de Perlin e Gerstner mantiveram-se a frente da técnica de *FFT*, na comparação para uma mesma técnica de *LoD*. A pior combinação em termos de desempenho é de *FFT* com *GeoClipMap*, como esperado devido à eficiência combinada dos dois algortimos. Em contrapartida, a melhor combinação é Perlin *Noise* com *Radial LoD*, tendo um eficiência de aproximadamente 3.5 vezes a obtida pela pior combinação.

Numa análise despercebida, pode-se concluir que os ganhos são pequenos, da ordem de 1.3 a 3.6 vezes. No entanto, isso não é pouco, pois o ganho é feito em relação às técnicas implementdas em GPU, dessa forma, também altamente paralelas e otimizadas.

Tendo em vista que o teste total possui uma característica de ser um somador de tempos de resposta, as análises feitas *a priori* são também válidas na composição. No entanto, nota-se que o tempo total de processamento, nesse caso de 1 quadro de imagem, não é apenas a soma aritimética do tempos de cada técnica, isso pois, existe ainda um tempo para a geração da imagem e *overheads* da própria *API* gráfica.

7.8. Análise de Complexidade e Viabilidade

Para analisar a complexidade e a viabilidade de cada uma das técnicas estudadas é necessário saber quais as funcionalidades de hardware demandadas pela técnica, quais delas estão disponíveis nos dispositivos alvos e quais as restrições do hardware dos mesmos. Dessa forma, se estabeleceu os seguintes dispositivos alvos.

- Computadores Classe 1: corresponde a um computador de baixo custo típico.
  Processador: 1 Core @ > 1.6 GHz
  Memória: 1 GBytes
  Placa de Vídeo: On-Board. Shader Model 2.0 com 128 MBytes de memória de textura/dados compartilhada (não possui CUDA).

- Computadores Classe 2: corresponde a um computador de um game player típico.
  Processador: 2 a 4 Core @ > 2.0 GHz
  Memória: 3 a 4 GBytes
  Placa de Vídeo: Off-Board. Shader Model 4.0 com 512 MBytes de memória de textura/dados dedicada (pode ou não ter CUDA).

- Notebook
  Processador: 2 Core @ > 1.6 GHz
  Memória: 3 a 4 GBytes
  Placa de Vídeo: On-Board. Shader Model 3.0 com 128 MBytes de memória de textura/dados dedicada (não possui CUDA).

- Netbooks
  Processador: 1 Core @ > 800 Mhz
  Memória: 512 MByte a 1 GByte
  Placa de Vídeo: On-Board. Shader Model 2.0 com 128 MBytes de memória de textura/dados compartilhada (não possui CUDA).

- Smartphones
  Processador: 1 Core @ > 500 Mhz
  Memória: 64 a 256 MBytes
  Placa de Vídeo: On-Board. Shader Model 1.1 com 32 MBytes de memória de textura/dados compartilhada (não possui CUDA).

Analisando as necessidades de hardware de cada técnica, pode-se, especificar as seguintes restrições:

Técnicas de Simulação

- Gerstner: Baixa Complexidade
  Melhor caso: Shader Model 3.0 ou maior, usado como vertex texture. Memória Mínima: 4 Mbytes (supondo textura de 512 x 512).

  Caso Médio: Shader Model 2.0, usado no vertex program. Não há necessidade de textura. Restrição à poucas harmônicas e malha pequena.

  Pior Caso: Shader Model 1.1, somente via CPU, atualizando a malha diretamente. Restrição à, no máximo, duas harmônicas e malha de no máximo 32 x 32 vértices com processador de clock maior que 500 MHz.

- *FFT*: Alta Complexidade
  Melhor caso: Shader Model 3.0 ou maior com *CUDA*, usado como *vertex texture*. Memória Mínima: 8 Mbytes (supondo textura de 512 x 512).

  Caso Médio: Shader Model 3.0 ou maior sem *CUDA*, usado como *vertex texture*. Memória Mínima: 8 Mbytes (supondo textura de 512

x 512). Necessidade de implementação de *FFT* no *fragment shader*, elevando ao grau de altíssima complexidade, e portanto, com grande chance de erros.

Pior Caso: Shader Model 2.0 ou inferior, somente via CPU, atualizando a malha diretamente. Restrição à, no máximo, 64 x 64 de região de simulação com processador com clock maior que 1.6 GHz.

- Perlin: Média Complexidade
  Melhor caso: Shader Model 3.0 ou maior com CUDA, usado como vertex texture. Memória Mínima: 8 Mbytes (supondo textura de 512 x 512).

  Caso Médio: Shader Model 3.0 ou maior sem CUDA, usado como vertex texture. Memória Mínima: 8 Mbytes (supondo textura de 512 x 512).

  Pior Caso: Shader Model 2.0 ou inferior, somente via CPU, atualizando a malha diretamente. Restrição à, no máximo, 128 x 128 de região de simulação com processador com clock maior que 1.0 GHz.

Técnicas de Nível de Detalhe

- GeoClipMap: Média Complexidade
  Melhor caso: Shader Model 3.0 ou maior. Memória Mínima: 32 Mbytes (supondo grade inicial de 256 x 256).

  Pior Caso: Shader Model 2.0 ou inferior, somente via CPU. Inviável.

- Radial LoD: Baixa Complexidade

Melhor caso: Shader Model 2.0 ou maior. Memória Mínima: 8 Mbytes (supondo grade inicial de 256 x 256). Pode ser retecelado na GPU.

Pior Caso: Shader Model 1.1 ou inferior, somente via CPU. Memória Mínima: 8 Mbytes (supondo grade inicial de 256 x 256).

- Projected Grid: Alta Complexidade
  Melhor caso: Shader Model 2.0 ou maior. Memória Mínima: 8 Mbytes (supondo grade inicial de 256 x 256).

  Pior Caso: Shader Model 1.1 ou inferior, somente via CPU. Memória Mínima: 8 Mbytes (supondo grade inicial de 256 x 256) com processador com clock maior que 800 MHz.

Tendo os dispositivos e as restrições definidas, gera-se a Tabela 21 com a nomenclatura realizada da seguinte maneira: <best,med,worst,none>, sendo, respectivamente, melhor caso, caso médio, pior caso e inviável. A célula cinza representa a melhor técnica para o dispositivo, baseado nas restrições e benefícios de cada técnica.

Tabela 21: Tabela de Viabilidade das Técnicas

	Comp. 1	**Comp. 2**	**Notebook**	**Netbook**	**Smartphone**
Gerstner	med	best	best	med	worst
FFT	worst	best	med	worst	none
Perlin	worst	best	med	worst	worst
GeoClipMap	none	best	best	none	none
Radial LoD	best	best	best	best	worst
Proj. Grid	best	best	best	best	worst

Observando a tabela e as análises, percebe-se que a técnica *FFT* é muito restrita, sendo aconselhada apenas para hardwares robustos. A técnica Perlin pode ser mais explorada, mesmo sem suporte a acesso de textura no *vertex shader*, isso permite que a mesma possa ser empregada

em qualquer contexto. Contudo, aconselha-se a não utilizá-la com dispositivos de baixa capacidade de processamento como *Netbooks* e *Smartphones*. Por fim, a técnica de Gerstner pode ser usada em todos os contextos, contudo, não gera resultados visuais tão bons quanto às demais. Ela é a solução mais adequada para *smartphones* e sistemas nos quais o controle rígido de cada parâmetro da onda é necessário.

A técnica de *GeoClipMap* é fatalmente dependente da funcionalidade de acesso a texturas no *vertex shader*, dessa forma, somente sistemas com essa funcionalidade bem implementada podem usufruir dela, sendo assim, ela é restrita a sistemas mais robustos. A técnica de *Radial LoD* é a mais adequado a todos os sistemas pois é de baixa complexidade, baixo custo de memória e compatível com qualquer hardware, pois, pode ser pré-processada (somente a rotação da malha). Já a técnica de *Projected Grid* é adequada a qualquer hardware, no entanto, é muito complexa e sujeita a diversos erros de implementação, além de um custo maior de processamento. Uma forma alternativa de comparação dos métodos pode ser observada através da Figura 53.

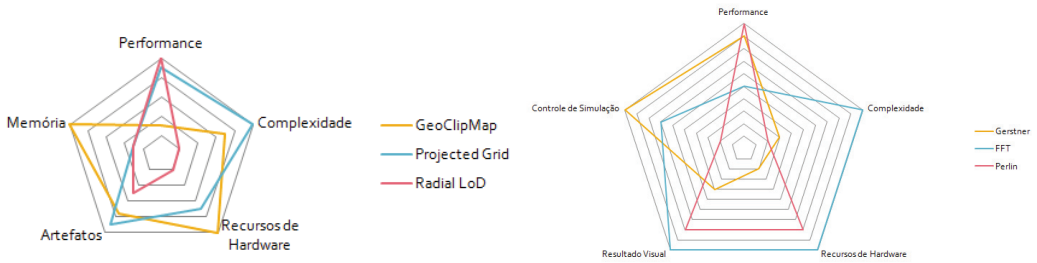


Figura 53: Gráfico dos pontos de análise dos algoritmos

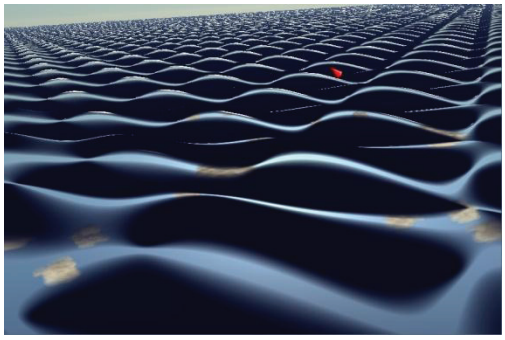
7.9. Imagens das técnicas

A utilização de cada técnica possui benefícios e problemas, dessa forma, a possibilidade de combinação das mesmas faz com que seja possível unir os benefícios. Um exemplo claro é utilizar o mapa de altura da técnica de Gerstner e as normais geradas pelo modo de *Perlin noise*, isso

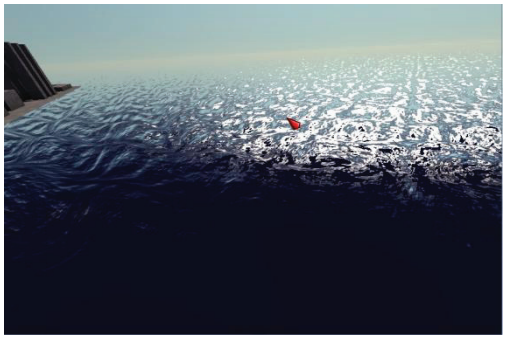
dará ao usuário o poder de variar os diversos parâmetros das ondas e trará um bom realismo de iluminação devido às normais de Perlin.

As figuras abaixo ilustram as imagens geradas de cada uma dessas técnicas e os resultados da combinação delas.

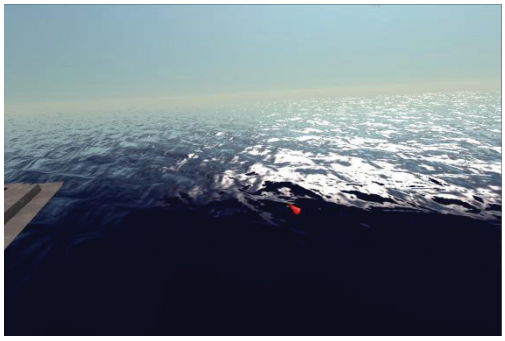
Tabela 22: Imagens das técnicas



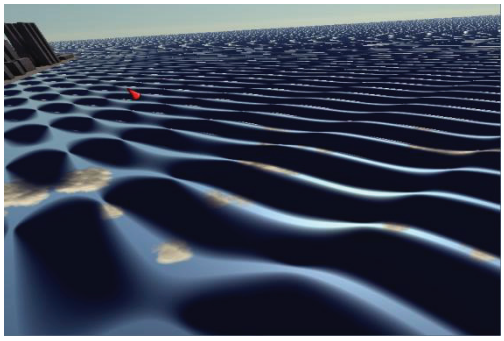
Gerstner + Gerstner



Gerstner + Perlin

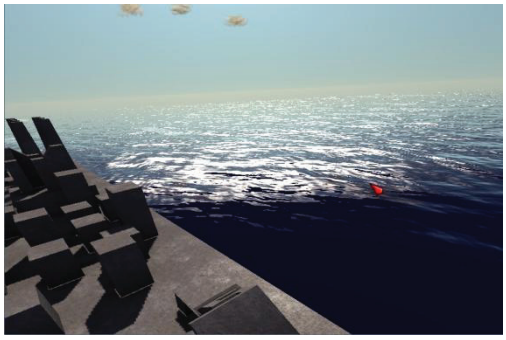


Gerstner + *FFT*

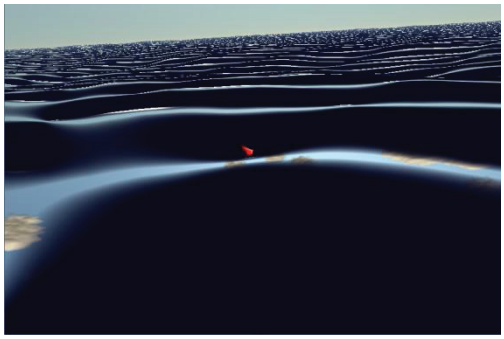


Perlin + Gerstner

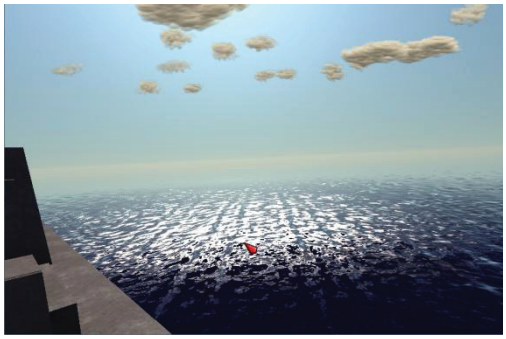
Perlin + Perlin



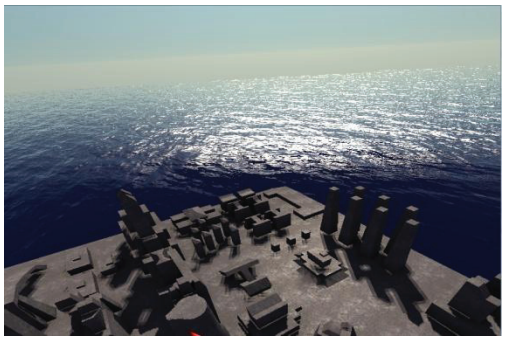
Perlin + *FFT*



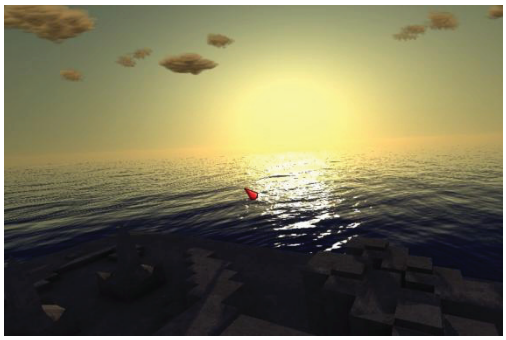
FFT + Gerstner



FFT + Perlin

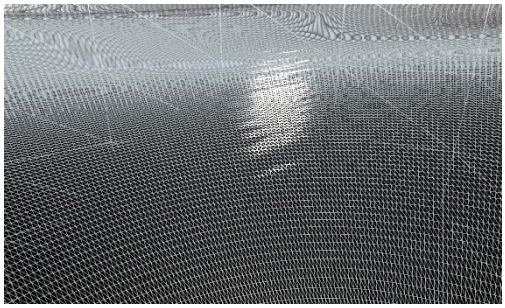


FFT + *FFT*

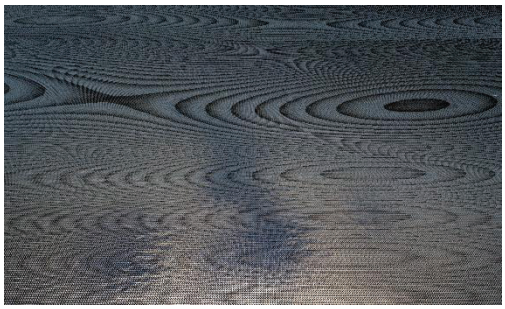


Visualização Completa[8]

Técnica de *GeoClipMap*



Técnica de *Radial LoD*



Técnica de *Projected Grid*

Imagem Estéreo Anaglifo

[8] Visualização utilizando céu realístico, nuvens de billboard. As sombras utilizam o algoritmo Perspective Parallel Split Shadow Map (Kozlov, 2004).

As imagens da Tabela 22 mostram o poder da combinação das técnicas sugerida pelo presente trabalho. A técnica puramente utilizando Gerstner apresenta um aspecto muito sintético e suave da água, sendo notoriamente não realístico. No entanto, a combinação do mesmo com *Perlin Noise* ou *FFT*, na etapa de geração de normais aumenta, consideravelmente, o realismo da cena, e, além disso, proporciona um ótimo controle da simulação na malha. Desse modo, sendo muito útil em situações nas quais desejam-se controlar o movimento da malha, mas tendo uma boa iluminação (p.e. um gota caindo em um lago parado).

Porém, deve-se ressaltar que a geração de normais de Gerstner não é adequada para a iluminação final de uma cena. Ela deve ser usada quando se deseja aplicar a técnica de *bump* ou normal *map* de forma exata.

A técnica usando *Perlin Noise* gera imagens bastante realísticas, utilizando baixo consumo de processamento. No entanto, apenas se tem controle do nível de agitação do oceano. A combinação com *FFT* pouco acrescenta no realismo, conforme observado nas imagens da Tabela 22, mas pode ser usada para a geração de normais com influência do vento. O mesmo ocorre com as combinações com *FFT* e Perlin (*heightmap* e *normalmap*), contudo, nesse caso, há um melhor controle das ondas, com uma geração de normais com baixíssimo ruído de repetição, o que não acontece com pequenas simulações de *FFT*. A imagem anaglifo apresentada ilustra o problema de desconforto visual provocado pelos mapas de textura discutidos em 5.2 e 5.3.

Além disso, as imagens das técnicas de *LoD* mostram a forma de tecelagem de malha que cada uma realiza. Essas imagens foram geradas em modo *Wireframe* com alto brilho para se notar o processo.

Por fim, a combinação das técnicas de mapa de altura e mapa de normais amplia as possibilidades de aplicação do oceano em sistemas computacionais, gerando imagens realísticas. Pode-se destacar 4

combinações úteis, Gerstner + Perlin, Gerstner + *FFT*, Perlin + *FFT* e *FFT* + Perlin.

Tendo os resultados e as análises realizadas, pode-se iniciar a conclusão final deste trabalho.