Laboratório de Engenharia Informática

Mestrado em Engenharia Informática

Peter Vala A84556 and Pedro Fernandes A85853

Universidade do Minho, Braga, Portugal

Abstract. Keywords: *WebGL · Empirical Ocean Simulation · Graphical Computing*

1 Introdução

No âmbito da unidade curricular de LEI(Laboratório de Engenharia Informática), foi nos proposto pelo sr^{o} professor António Ramires a aplicação da simulação de oceano em WebGL. Para tal, foi-nos fornecida uma implementação em *GLSL* com recurso a compute shaders, e a partir desta implementação migra-la para o ambiente de WebGL.

2 Fundamento Teórico

Antes de começarmos a implementação, achamos que, uma lucidez à matemática "por detrás da coisa" seria uma boa prática, e ajudar também a perceber o funcionamento onde se baseia o mecanismo para produzir o efeito final.

Dessa forma através de dois artigos nos quais foram baseados a implementação base fornecida(empiracle directional wave spectra for computer graphics - Christopher J. Horvath e Simulating Ocean Water - Christopher J. Horvath), foi analisada de que forma as contas encaixavam no código. Pela analise dos artigos foi-nos possível entender que este é um tópico já investigado a muitos anos, e entender também de que forma e em que contexto cada autor dos algoritmos se baseou. Percebemos também que, Jerry Tessendorf juntou todos estes algoritmos e aplicou-os no código de forma a simular o oceano. Assim, o primeiro artigo, fala-nos sobretudo sobre a simulação do efeito das ondas e das diferentes formulações do espectro de onda do oceano e o segundo já nos explica melhor na forma como foi elaborada a parte de Computação Gráfica.

Além da curiosidade teórica, o fundamento teórico não produziu grande impacto na metodologia implementada, uma vez que, dado que o projecto fornecido já continha todos esses tópicos implementados no código.

3 Implementação NAU3D

A Nau3D é um motor gráfico desenvolvido *in-house* na Universidade do Minho. Este suporta OpenGL, Optix, scripts Lua e usa ImGui como interface com o utilizador. A Nau3D permite de maneira relativamente simples *pipelines* de *shaders* 3D em GLSL. Também suporta *kernels* escritos em CUDA para Optix7, uma funcionalidade que não usamos neste trabalho.

Suporta *shaders vertex*, *geometry*, *fragment* e *compute*, dos quais todos são usados, exceto o *geometry*, no projeto que nos foi fornecido como referência.

Esta implementação, praticamente completa, foi-nos fornecida devido à relativamente grande complexidade do problema em mãos. Ter uma implementação funcional, apesar de não ser diretamente traduzível para WebGL, foi uma grande ajuda durante o desenvolvimento do *software*, porque permitiu baixar uma das maiores barreiras a implementar algo assim: a análise teórica do problema, explicada brevemente acima.

3.1 Pipeline de shaders

De uma forma bastante simples, o pipeline de *shaders* do projeto Nau3D é constítuído por 4 *compute shaders*, e 2 conjuntos de *shaders vertex + fragment*, sendo que o segundo conjunto será ignorado pois apenas coloca uma bola a flutuar no oceano, e não o afeta em si.

Os *compute shaders* trabalham sobre uma textura de *floats*, da qual os *vertex + fragment shaders* usam os valores para calcular a aparência do oceano.

Os primeiros dois *shaders compute* calculam o espectro inicial e ao longo do tempo, respetivamente. Esta é a secção do código que "decidirá" efetivamente qual será o aspeto do oceano. No final da sua execução, a textura 2D de *floats* com valores no domínio das frequências.

Estes recebem como "argumento", se é que se pode chamar argumentos aos valores que são passados a um *shader* OpenGL, o tipo de espetro e respetivos parâmetros, um fator de escala, entre outros. Estes parâmetro podem ser alterados em tempo real pelo utilizador, com recurso a uma interface composta de *sliders* e *dropdowns*.

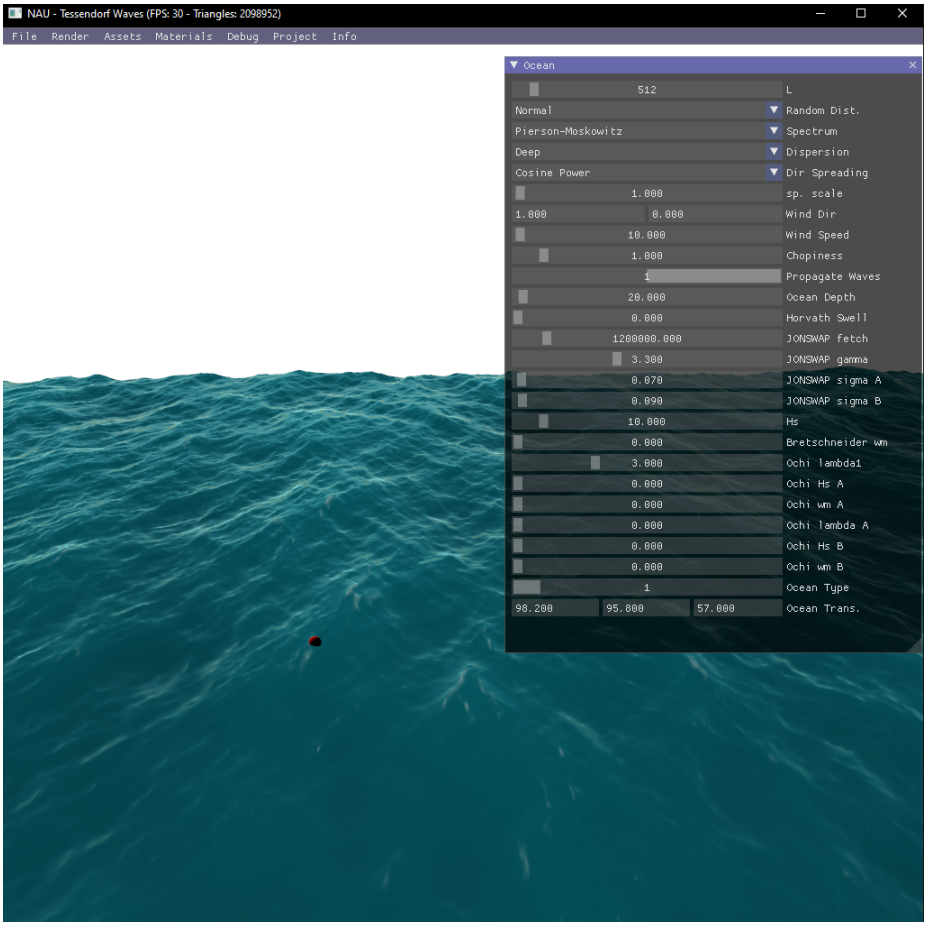

Fig. 1. Projeto Nau3D, com a respetiva interface

O terceiro e quarto *compute shaders* trabalham em conjuntos para computar a transformada de fourrier inversa da textura criada pelos 2 *shaders* anteriores, para assim recuperar-mos a função original.

Por fim, o conjunto de *vertex + fragment shaders* desenha a simulação do oceano no ecrâ, a partir de uma grelha espaçada uniformemente no plano XZ. Com base nos valores da textura obtida com os *shaders* anteriores, os pontos da grelha são deslocados para o local correto no *shader vertex*. Durante o *fragment*, a cor do oceano é calculada e aplicada. Em certos locais, é aplicada uma textura de espuma, para um efeito um pouco mais realista.

Claro que muitos detalhes estão a ser ignorados neste explicação, mas este não será o foco deste relatório.

4 Desenvolvimento WebGL

Tendo já analisado o código em OpenGL a nossa primeira preocupação foi entender até que ponto poderíamos migrar o código para WebGL. O WebGL usa como API gráfica o OpenGL ES (OpenGL for Embedded Systems), um *subset* do OpenGL desenhado para sistemas com pouca potência computacional.

Sendo esperado correr no browser, como seria natural, a código corrido em CPU é escrita em Javascript, e é muito verboso. Para nos ajudar com esta parte, usamos uma biblioteca muito usada chamada "TWGL" [2]. Ela não acrescenta qualquer nova funcionalidade. Simplesmente providencia *wrappers* para as funções já existente, podendo inclusive ser misturado livremente com código WebGL puro, sem qualquer problema. A utilização desta biblioteca simplesmente reduz muito o tamanho do código.

Devido à naruteza do WebGL, algumas das funcionalidades do OpenGL moderno não estão disponíveis, visto que o OpenGL ES é fortemente baseado no OpenGL 2.0, desatualizado nos dias de hoje. Este facto viria a revelar-se bastante problemático mais tarde.

4.1 Compute Shaders

O problema mais imediato é o facto de que, tal como no OpenGL 2.0, no WebGL não há suporte para *compute shaders*. Tendo em conta que 80% dos shaders que queríamos implementar eram *compute shaders*, isto representa obviamente um problema.

A solução para este problema foi transformar cada *compute shader* num conjunto de *vertex + fragment shader*. Iterando sobre toda a textura de dados com recurso a simples *quad* no vertex shader, é possível realizar as contas provenientes do *compute shader* no *fragment shader*.

4.2 Escrita para texturas

Aquando desta tradução, debatemo-nos com outro problema. Em WebGL é possível amostrar qualquer posição de uma textura. No entanto, a única maneira de escrever para uma textura é usando um *renderTarget*. Isto significa que cada vez que o *fragment shader* é executado, ele só pode escrever no fragmento, ou no caso de uma textura de dados, na posição da textura de output que lhe foi "atribuída". Isto não resultou num problema nos primeiros dois *shaders*, visto que esses escrevem exatamente apenas na mesma posição sobre a qual foram chamados.

No entanto, os *compute shaders* correspondentes à transformada de fourrier implementados no projeto da Nau3D requerem escritas arbitrárias na textura de *output*, algo que é simplesmente impossível em WebGL. A única maneira de contornar este problema era encontrar outra formulação de WebGL. A única solução viável que conseguimos encontrar foi a formulação de StockHam, usada numa outra implementação mais simples de uma simulação de oceanos [1]. No entanto, nunca conseguimos obter os mesmos resultados que o nosso projeto de

referências, apesar de inúmeras horas de *debug*. Iremos discutir isto com mais detalhe mais à frente.

4.3 Dificuldade no Debug

Devido à natureza do OpenGl, é relativamente difícil dar *debug* às aplicações que o usam. Não existe, por exemplo, uma maneira direta de verificar o conteúdo de uma variável num certo ponto de um shader. Por este motivo, cerca de 50% da duração do desenvolvimento do trabalho foi realizado praticamente sem *feedback*. O que isto quer dizer é que até todos os *shaders* estarem implementados, não haveria qualquer qualquer resultado discernível no ecrã.

Algo que podíamos fazer era comparar o resultado da textura de dados depois de cada *shader*. No entanto, para o podermos fazer tínhamos de reduzir o tamanho da textura de dados para algo muito pequeno, por exemplo, 4 por 4, algo que nem sempre escalava para quando a textura tinha o seu tamanho normal de 512 por 512.

4.4 Pequenos problemas

A tradução dos *shaders* trouxe outros problemas, facilmente resolvidos. O *casting* automático do OpenGL ES é sem dúvida muito menos avançado do que o atual. Por exemplo, ele não aceita que se multiplique um *float* por um inteiro. No entanto, resolver esse problema é tão simples como acrescentar um ".0" no fim dos inteiros.

Para além disso, existem algumas pequenas diferenças de sintaxe, por exemplo, na declaração de um *layout* para escrita numa textura. Sendo apenas problemas de sintaxe, são facilmente corrigíveis.

4.5 Estado final da simulação do oceano

Aquando da implementação de todos os *shaders*, e da *correção* dos *bugs* que invariavelmente existem em qualquer projeto, aterramos num ponto bastante estranho. A nossa simulação produzia um oceano relativamente realista, mas não igual ao projeto de referência. Depois de alguns testes, reduzimos o problema ao cálculo do IFFT. Infelizmente, mesmo após do que estimamos ser cerca de 15 horas a olhar para apenas 50 linhas de código, não conseguimos com que a simulação ficasse 100% correta.

O problema que de facto nos assombrou todo este tempo foi a dificuldade em localizar exatamente o problema, dado que ele aparentemente nascia do nada. Testando cada conta individual do IFFT o resultado de cada uma parecia correto, mas o resultado final não o estava. Chegamos a comparar o resultado do nosso projeto de referência com o projeto do David Li [1], e estes davam valores diferentes. No entanto, ambos dão o que aparente ser resultados corretos nas suas aplicações específicas.

Para além disso, em casos de teste, a nossa aplicação e a aplicação do David Li [1] davam o mesmo resultado.

Fique também registado que os testes que fizemos ao último par de *shaders* foi muito limitado. Ficamos tão presos na questão do IFFT que praticamente não o testamos. Sendo este a tradução mais direta da Nau3D para o nosso projeto, visto que já estava no formato correto de *vertex + fragment*, também achamos que era o sítio mais improvável para ter um erro. No entanto, nunca se sabe.

Tendo em conta todos os teste que fizemos, acreditamos que o problema é uma peculiaridade muito pequena que sinceramente acredito nunca iremos encontrar. Talvez um par fresco de olhos consiga encontrar o problema, e convidamos qualquer pessoa interessada a analisar o código, e a corrigi-lo, se assim o entender.

Acabamos então assim com um resultado que parece correto para alguém que não tenha outro ponto de referência, mas é claramente diferente do esperado, se comparado lado a lado, especialmente em termos de cor.

4.6 Interface

Podemos dividir a interface em dois componentes: a interface propriamente dita e o o "display" de um gráfico que representa o espetro que está a ser atualmente usado para a simulação do oceano.

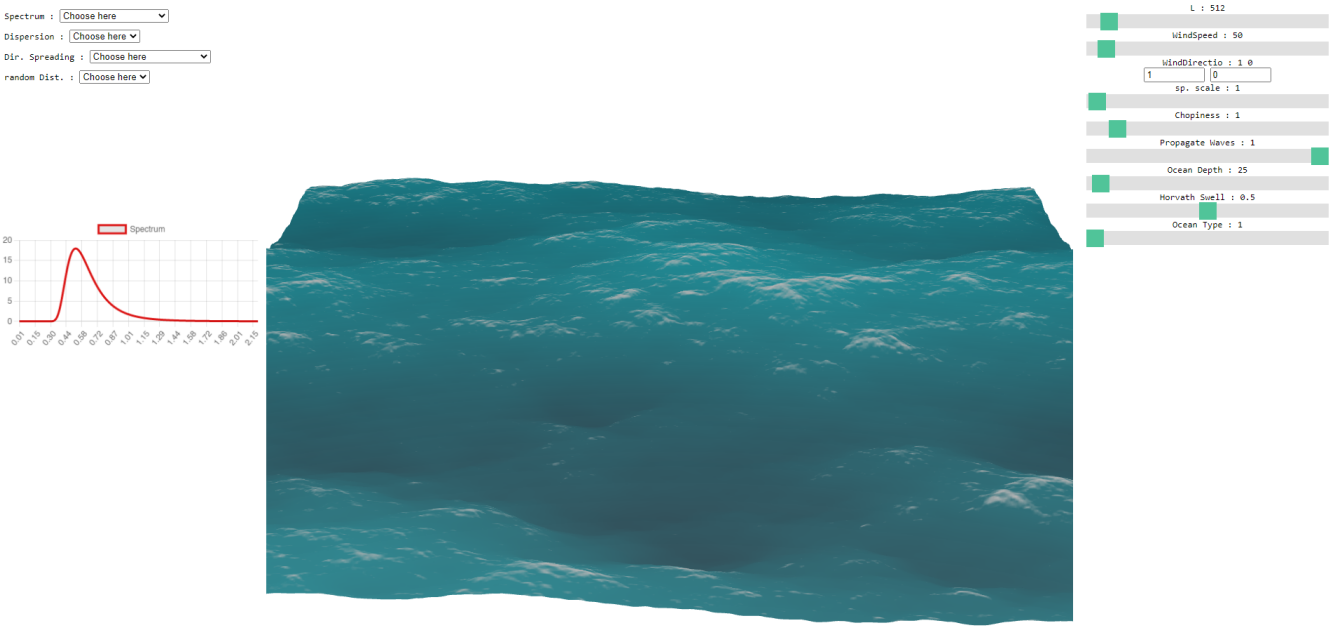

Fig. 2. Interface do programa, com a simulação no meio

4.7 Interface

No topo superior esquerdo e à direita da simulação, podemos ver os elementos que nos permitem interagir com o programa. Estes permitem-nos mudar o tipo de espetro a ser usado, e todas os parâmetros associados a cada um. São simples elementos de HTML, dos quais são extraídos os valores para atualizar as variáveis de JavaScript associadas.

5 Conclusão

References

1. David Li. *Ocean Wave Simulation*
   `http://david.li/waves/`
2. `http://twgljs.org/`