Московский Авиационный Институт
(Национальный Исследовательский Университет)

Курсовая работа по
Компьютерной графике

Симуляция поверхности океана в реальном времени с использованием GPU

Студент:
Колесников Е. В.

Преподаватель:
Измайлов А. А.

21 февраля 2015 г.

Содержание

1 Введение

1.1 Исследуемая область

Исследуемой областью является одно из перспективных направлений в компьютерном моделировании – симуляция поверхности океана. Кажущееся интуитивным предположение, что поставленная задача эквивалентна задаче моделирования жидкости, на самом деле является неверным.

Моделирование жидкости – область компьютерной графики, использующая средства вычислительной гидродинамики для реалистичного моделирования, анимации и визуализации жидкостей, газов, взрывов и других связанных с этим явлений, в то время как задача симуляции поверхности жидкости является задачей моделирования волны, что сильно упрощает используемый математический аппарат.

1.2 Перспективность исследований

Активному развитию данной области способствуют кинематограф и игровая индустрия, которые используют разработанные методы в своих продуктах. Так например в данной работе частично реализована модель океанской поверхности, которая была использована в фильме "Титаник".

Хотя CPU с каждым годом увеличивают свою мощность, ее все равно еще не достаточно для использование реалистичных моделей поверхности жидкости в компьютерных играх. Решением данной проблемы может стать одно из перспективных направлений исследований в последнее время – распределенные вычисления на GPU. Именно этот подход используется в данной работе для увеличения скорости работы программы.

Если для кинематографа скорость не является ключевым фактором, то в игровой индустрии – наоборот. Поэтому, пока у пользователей нет возможности использовать необходимую мощность для просчета реалистичной модели воды, данная область исследований останется перспективной, так как с одной стороны необходимо улучшать внешний вид моделей и их физическое поведение, так с другой стороны – нельзя выходить за рамки возможного количества вычислений, т.е. необходимо разрабатывать новые модели и более оптимизированные алгоритмы.

1.3 Постановка задачи

В данной работе решается задача моделирования поверхности открытого океана в реальном времени с использованием GPU для вычисления поля высот.

1.4 Возможные методы решения

Существует много подходов к моделированию и анимации поверхности воды. Большая часть из них основана на аналитической модели суперпозиции волн, и решение здесь либо задается сразу в виде линейной комбинации тригонометрических функций со специально подобранными коэффициентами, либо получается в результате обратного преобразования Фурье со специально заданным спектром. Выбор спектра и определяет сложность, реалистичность и детализацию модели. Такие модели могут быть как очень сложными и дорогими с вычислительной точки зрения, так и довольно простыми.

Моделированием водной поверхности занималось достаточное количество разработчиков, наиболее успешным среди которых был Тессендорф (Tessendorf). Именно его модель частично реализована в данной работе.

1.5 Используемые технологии

- Программа написана на OpenGL 4.1 с использованием GLFW3 в качестве GUI.
- Для вычисления поля высот используется технология CUDA и библиотека CUFFT для вычисления обратного преобразования Фурье.
- Библиотека SDL2_image и Devil необходимы для загрузки текстур для шейдера.
- В качестве библиотеки линейной алгебры используется библиотека GLM, специально разработанная для использования с OpenGL.

2 Используемые математические модели

2.1 Модель волны

В данной работе рассморен статистический метод создания волны. Модели такого типа основаны на возможности разложения поля высоты волны в сумму синусоид и косинусоид с использование случайных величин для генерации амплитуд частот. Вычислительно более выгодно использовать БПФ для вычисления этих сумм.

Поле высот с использованием БПФ можно представить в виде сумм синусоид с сложными, изменяющимися со временем амплитудами:

$$h(\mathbf{x}, t) = \sum_{\mathbf{k}} \tilde{h}(\mathbf{k}, t) \exp(ik \cdot \mathbf{x}) \tag{1}$$

где $\mathbf{x} = (x_1, x_2)$ – положение точки на двумерной сетке, t – время, а $\mathbf{k} = (k_1, k_2), k_1 = \frac{2\pi n}{L_1}, k_2 = \frac{2\pi m}{L_2}, -\frac{N}{2} \leq n < \frac{N}{2}, -\frac{M}{2} \leq m < \frac{M}{2}$.

Величины амплитуды однозначно задают все поле высот. Идея статистического метода заключается в создании случайных наборов амплитуд, удовлетворяющих эмпирическим законам океанографии.

Океанографические исследования показали, что уравнение 1 является достаточно точным представлением ветряных волн, возникающих в открытом океане.

$\tilde{h}(\mathbf{k}, t)$ можно представить в виде:

$$\tilde{h}(\mathbf{k}, t) = \tilde{h_0}(\mathbf{k}) \exp(i\omega(k)t) + \tilde{h_0^*}(-\mathbf{k}) \exp(-i\omega(k)t) \tag{2}$$

где $\tilde{h_0}(\mathbf{k})$ - амплитуды поля высоты в момент времени $t = 0$, которые задаются по формуле:

$$\tilde{h_0}(\mathbf{k}) = \frac{1}{\sqrt{2}}(\xi_r + i\xi_i)\sqrt{P_h(\mathbf{k})} \tag{3}$$

где $\xi_r, \xi_i \sim N(0, 1)$, а $P_h(\mathbf{k})$ - спектр Филлипса, который задается эмпирической формулой:

$$P_h(\mathbf{k}) = A\frac{\exp(-1/(kL)^2)}{k^4}|\hat{\mathbf{k}} \cdot \hat{\omega}|^2 \tag{4}$$

в которой $L = \frac{V^2}{g}$.

Именно спектр Филлипса является необходимым эмпирическим законом океанографии, благодаря которому волны становятся похожи на

настоящие. Данная формула была получена с помощью эксперементальных данных разного вида, собиравшихся в течение продолжительного количества времени.

2.2 Модель освещения

В качестве модели освещения используется модель Блинна-Фонга. Несмотря на то, что существую более точные модели освещения, она является стандартом в компьютерной графике.

Основная идея модели Блинна-Фонга заключается в предположении, что освещенность каждой точки тела разлагается на 3 компоненты:

1. фоновое освещение (ambient),
2. рассеянный свет (diffuse),
3. бликовая составляющая (specular).

Свойства источника определяют мощность излучения для каждой из этих компонент, а свойства материала поверхности определяют её способность воспринимать каждый вид освещения.

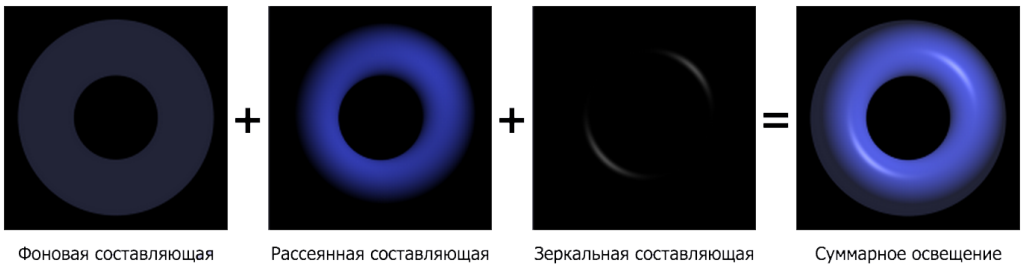

Рис. 1: Модель Блинна-Фонга

Фоновое освещение это постоянная в каждой точке величина надбавки к освещению. Вычисляется фоновая составляющая освещения как:

$$I_a = k_a i_a, \text{ где} \tag{5}$$

- I_a – фоновая составляющая освещенности в точке;
- k_a – свойство материала воспринимать фоновое освещение;
- i_a – мощность фонового освещения.

Рассеянный свет при попадании на поверхность рассеивается равномерно во все стороны. При расчете такого освещения учитывается только ориентация поверхности (нормаль) и направление на источник света. Рассеянная составляющая рассчитывается по закону косинусов (закон Ламберта):

$$I_d = k_d(\vec{L} \cdot \vec{N})i_d, \text{ где} \tag{6}$$

- I_d – рассеянная составляющая освещенности в точке,
- k_d – свойство материала воспринимать рассеянное освещение,
- $\vec{L}$ – направление из точки на источник,
- $\vec{N}$ – вектор нормали в точке,
- i_d – мощность рассеянного освещения.

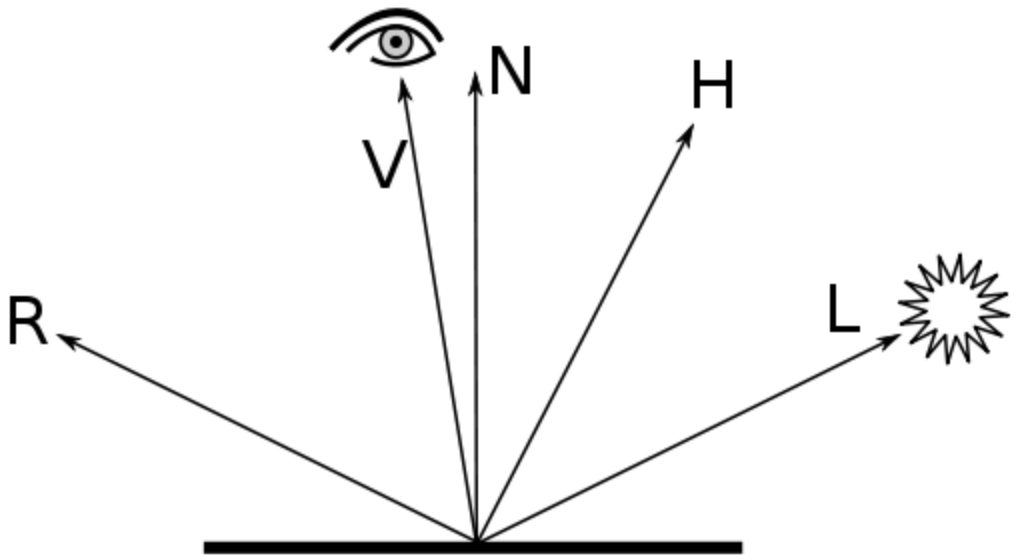

Рис. 2: Необходимые векторы для модели Блинна-Фонга

Зеркальный свет при попадании на поверхность подчиняется следующему закону: “Падающий и отраженный лучи лежат в одной плоскости с нормалью к отражающей поверхности в точке падения, и эта нормаль

делит угол между лучами на две равные части". Т.о. отраженная составляющая освещенности в точке зависит от того, насколько близки направления на наблюдателя и отраженного луча. Это можно выразить следующей формулой:

$$I_s = k_s(\vec{H} \cdot \vec{N})^{\beta} i_s, \text{ где} \tag{7}$$

- I_s – зеркальная составляющая освещенности в точке,
- k_s – коэффициент зеркального отражения,
- $\vec{H} = \dfrac{\vec{L} + \vec{V}}{|\vec{L} + \vec{V}|}$ – ориентация площадки, на которой будет максимальное отражение,
- $\vec{N}$ – вектор нормали в точке,
- i_s – мощность зеркального освещения,
- β – коэффициент блеска, свойство материала.

Именно зеркальное отражение представляет наибольший интерес, но в то же время его расчет требует больших вычислительных затрат. При фиксированном положении поверхности относительно источников света фоновая и рассеянные составляющие освещения могут быть просчитаны единожды для всей сцены, т.к. их значение не зависит от направления взгляда. С зеркальной составляющей этот фокус не сработает и придется пересчитывать её каждый раз, когда взгляд меняет свое направление.

Во всех вычислениях выше, для рассеянной и зеркальной компонен, если скалярное произведение в правой части меньше нуля, то соответствующая компонента освещенности полагается равной нулю.

3 Детали реализации

3.1 Скайбокс

Скайбокс – объект в трёхмерной графике, играющий роль неба и горизонта. Представляет собой несложную трёхмерную модель (как правило, куб), с внутренней стороны которой натянута текстура неба – "кубическая текстура".

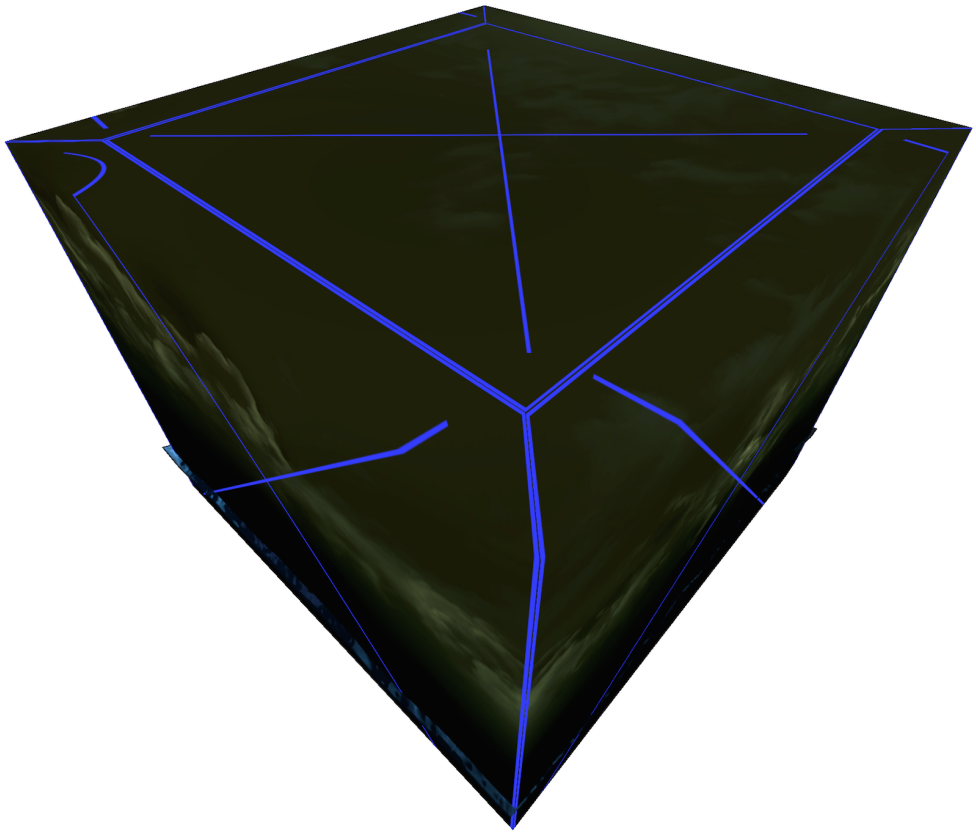

Рис. 3: Реализованный скайбокс

Обработка трёхмерной графики требует много вычислительной работы, поэтому "честно"просчитывать объекты, находящиеся на горизонте, было бы расточительством. К тому же, трёхмерное аппаратное обеспечение имеет Z-буферы, которые из-за ограниченной разрядности отбрасывают всё, что находится далеко от камеры.

Поэтому удалённые объекты изображаются крайне примитивно: в виде куба, шесть граней которого — текстуры неба и горизонта. Если отобразить этот куб так, чтобы камера находилась точно в центре, будет казаться, что через камеру действительно видны небо и горизонт.

В данной компьютерной симуляции скайбокс реализован в виде куба с 6-ю различными текстурами, которые накладываются на каждую из граней.

Фрагментный шейдер, используемый для текстурирования имеет тривиальный вид:

```
1 #version 410 core
2
3 in vec3 texCoord;
4 out vec4 fColor;
5 uniform samplerCube cubemap;
6
7 void main (void) {
8     fColor = texture(cubemap, texCoord);
9 }
```

3.2 Поле высот

Для того, чтобы эффективно реализовать симуляцию, используя распределенные вычисления на GPU, необходимо принимать во внимание следующие 2 факта:

1. Копирование данных со стороны CPU на сторону GPU является очень затратной операцией,

2. GPU имеет ограниченный объем памяти.

Вторая проблема решится на уровне постановки задачи – предполагается, что вся необходимая для симуляции информация, полностью помещается в памяти GPU.

Для того, чтобы решить первую проблему, необходимо написать программу таким образом, чтобы значения, вычисленные на GPU не передавались обратно на сторону CPU, а сразу копировались в OpenGL буффер. Для того, чтобы реализовать такое поведение, CUDA предоставляет 4 функции:

- cudaGraphicsGLRegisterBuffer – регистрирует вспомогательную CUDA структуру, которая может обращаться к OpenGL буферу,

- cudaGraphicsMapResources – соединяет вспомогательную структуру с OpenGL буфером,

- cudaGraphicsResourceGetMappedPointer – возвращает указатель, с помощью которого можно скопировать данные в OpenGL буфер напрямую,
- cudaGraphicsUnmapResources – закрывает соединение вспомогательной структуры с OpenGL буфером.

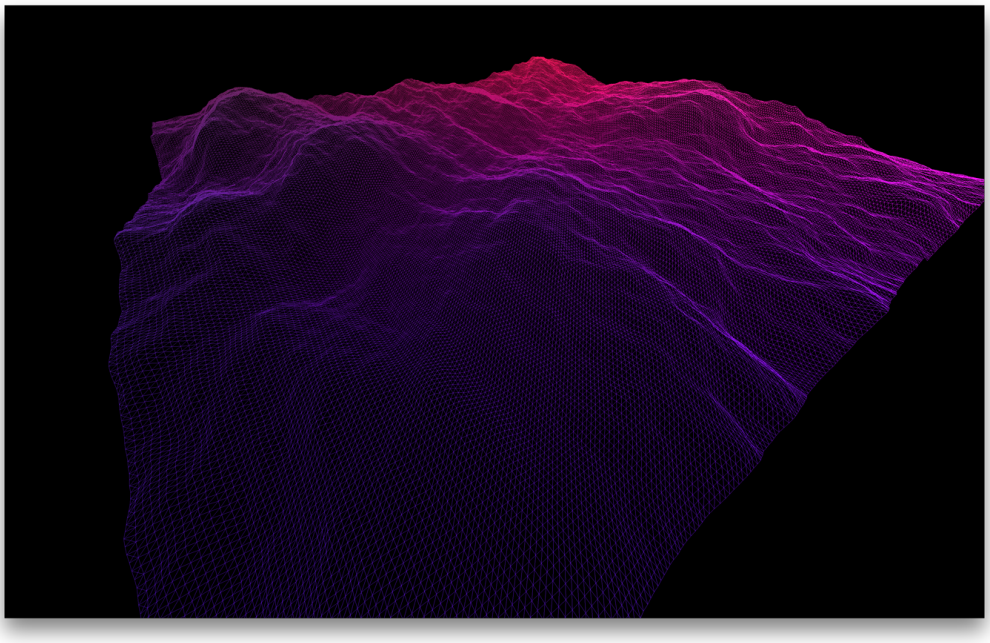

Рис. 4: Пример сгенерированного поля высот

Вычисление поля высот происходит каждый раз, когда рендерится изображение.

```
1 // generate wave spectrum in frequency domain
2 cudaGenerateSpectrumKernel(d_h0, d_ht, spectrum, meshSize,
      meshSize, curTime, patchSize);
3
4 // execute inverse FFT to convert to spatial domain
5 checkCudaErrors(cufftExecC2C(fftPlan, d_ht, d_ht,
      CUFFT_INVERSE));
6
7 // update heightmap values in vertex buffer
8 checkCudaErrors(cudaGraphicsMapResources(1, &
      cuda_heightVB_resource, 0));
9 checkCudaErrors(cudaGraphicsResourceGetMappedPointer((void **)
      &g_hptr, &num_bytes, cuda_heightVB_resource));
```

```
10 cudaUpdateHeightmapKernel(g_hptr, d_ht, meshSize, meshSize);
11 checkCudaErrors(cudaGraphicsUnmapResources(1, &
       cuda_heightVB_resource, 0));
```

В данном коде функции cudaGenerateSpectrumKernel, cufftExecC2C, cudaUpdateHeightmapKernel выполняются на GPU. Функция cufftExecC2C является библиотечной функцией, которая производит в данном случае обратное преобразование Фурье, а оставшиеся функции написаны вручную и имеют следующую реализацию:

```
1  // generate wave heightfield at time t based on initial
       heightfield and dispersion relationship
2  __global__ void generateSpectrumKernel(float2 *h0, float2 *ht,
3                  unsigned int in_width, unsigned int out_width,
4                  unsigned int out_height, float t, float
                       patchSize)
5  {
6    unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
7    unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;
8    unsigned int in_index = y*in_width+x;
9    unsigned int in_mindex = (out_height - y)*in_width + (
       out_width - x); // mirrored
10   unsigned int out_index = y*out_width+x;
11
12   float2 k;
13   k.x = (-(int)out_width / 2.0f + x) * (2.0f * CUDART_PI_F /
       patchSize);
14   k.y = (-(int)out_width / 2.0f + y) * (2.0f * CUDART_PI_F /
       patchSize);
15
16   float k_len = sqrtf(k.x*k.x + k.y*k.y);
17   float w = sqrtf(9.81f * k_len);
18
19   if((x < out_width) && (y < out_height)) {
20     float2 h0_k = h0[in_index];
21     float2 h0_mk = h0[in_mindex];
22     ht[out_index] = complex_add(complex_mult(h0_k, complex_exp
         (w * t)),
23                          complex_mult(conjugate(h0_mk),
                                complex_exp(-w * t)));
24   }
25 }
26
27 // update height map values based on output of FFT
28 __global__ void updateHeightmapKernel(float  *heightMap,
29                  float2 *ht, unsigned int width)
30 {
31   unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
32   unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;
```

```
33    unsigned int i = y * width  +x;
34
35    float sign_correction = ((x + y) & 0x01) ? -1.0f : 1.0f;
36
37    heightMap[i] = ht[i].x * sign_correction;
38  }
```

Функция generateSpectrumKernel генерирует поле высот из начального поля высот и пройденного времени, а updateHeightmapKernel – вспомогательная функция, которая реализует смещение точек после обратного преобразования Фурье.

Не менее интересной является реализация функции, которая генерирует начальное поле высот:

```
1   float Waves::phillips(float Kx, float Ky)
2   {
3     float k_squared = Kx * Kx + Ky * Ky;
4
5     if (k_squared == 0.0f) {
6       return 0.0f;
7     }
8
9     float L = windSpeed * windSpeed / g;
10    float k_x = Kx / sqrtf(k_squared);
11    float k_y = Ky / sqrtf(k_squared);
12    float w_dot_k = k_x * windDir.x + k_y * windDir.y;
13    float phillips = A * expf(-1.0f / (k_squared * L * L))
14                         / (k_squared * k_squared) * w_dot_k *
                               w_dot_k;
15
16    // filter out waves moving opposite to wind
17    if (w_dot_k < 0.0f) {
18      phillips *= dirDepend; // dir_depend;
19    }
20
21    return phillips;
22  }
23
24  void Waves::generateH0()
25  {
26    for (unsigned int y = 0; y < spectrum; ++y) {
27      for (unsigned int x = 0; x < spectrum; ++x) {
28        float kx = (-(int)meshSize / 2.0f + x) * (2.0f *
              CUDART_PI_F / patchSize);
29        float ky = (-(int)meshSize / 2.0f + y) * (2.0f *
              CUDART_PI_F / patchSize);
30
31        float P = sqrtf(phillips(kx, ky));
32
```

```
33          float Er = gauss();
34          float Ei = gauss();
35
36          float h0_re = Er * P * CUDART_SQRT_HALF_F;
37          float h0_im = Ei * P * CUDART_SQRT_HALF_F;
38
39          int i = y * spectrum + x;
40          h_h0[i].x = h0_re;
41          h_h0[i].y = h0_im;
42      }
43    }
44  }
```

3.3 Освещение

Реализация модели освещения Блинна-Фонга имеет стандартный вид:

```
1  /* vertex shader */
2  #version 410 core
3
4  layout(location = 0) in vec4 meshPos;
5  layout(location = 1) in float height;
6  layout(location = 2) in vec2 slope;
7
8  uniform mat4 PVM;
9  uniform vec3 lightPos;
10 uniform vec3 eyePos;
11
12 out vec3 l;
13 out vec3 h;
14 out vec3 n;
15 out vec3 r;
16
17 out vec4 pos;
18
19 void main() {
20   vec3 lp = abs(lightPos);
21   vec3 p = vec3(meshPos.x, 1e+2 * height, meshPos.z);
22   gl_Position = PVM * vec4(p, 1.0);
23   p.x = p.x - 1000; p.z = p.z - 1000;
24   p.y = p.y - 500;
25   pos = vec4(p, 1.0);
26   l = normalize(lp - p);
27   vec3 v = normalize(eyePos - p);
28   h = normalize((v + l) / length(v + l));
29   n = normalize(cross( vec3(0.0, slope.y, 1.0 / 256), vec3(1.0
         / 256, slope.x, 0.0)));
30   r = reflect(-l, n);
```

```
}
```

```
/* fragment shader */
#version 410 core

in vec4 pos;
out vec4 fColor;

uniform vec3 sourceColor;
uniform vec3 diffColor;
uniform vec3 specColor;
uniform vec3 lightPos;
uniform vec3 eyePos;

in vec3 l;
in vec3 h;
in vec3 n;
in vec3 r;

uniform vec3 Ka;
uniform vec3 Kd;
uniform vec3 Ks;
uniform float alpha;

vec3 BlinnPhongModel()
{
  return  Ka * sourceColor +
          Kd * max(dot(n, -l), 0.0) * diffColor +
          Ks * max(pow(dot(n, h), alpha), 0.0) * specColor;
}


void main (void) {
  vec3 BlinnPhong = exp(-0.8 + 1.2*abs(pos.x/3000+pos.z/3000))
      * BlinnPhongModel();
  fColor = vec4(BlinnPhong, 0.9);
}
```

В фрагментном шейдере используется α-канал не равный единице, в данной реализации $\alpha = 0.9$, чтобы была возможность сквозь воду просматривать дно.

4 Подведение итогов

4.1 Полученные результаты

Результат работы программы виден на следующем скриншоте:

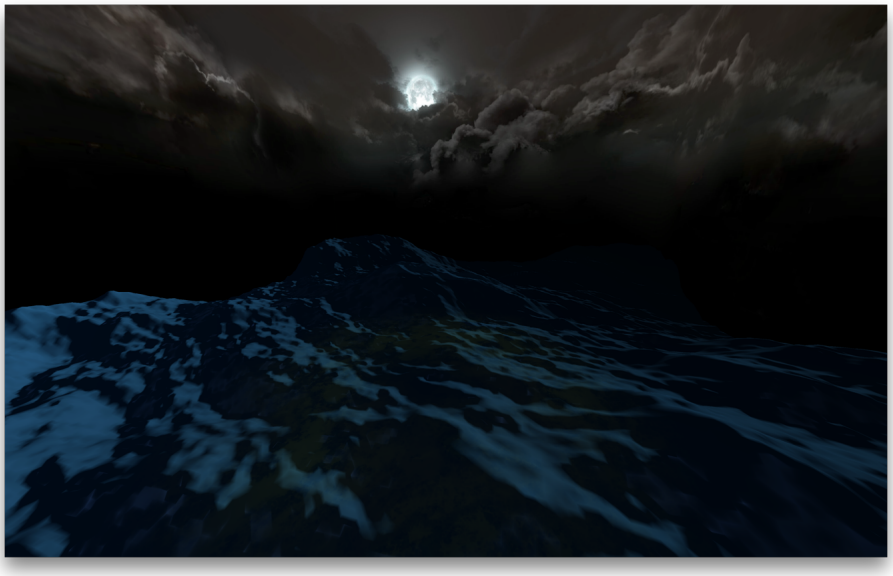

Рис. 5: Скриншот работы программы

Полученный результат напоминает жидкость, но океан имеет более сложную текстуру. Можно было бы продолжить исследовать проблему симуляции поверхности океана и добавить такие эффекты, как порывистые волны, брызги, пену, более подходящую для океана модель освещения, интерференцию волн, отражение мира на поверхности воды, каустический эффект и многое другое, которые бы улучшили внешний вид воды, но тема слишком сложная для любительского ознакомления.

4.2 Вывод

Симуляция поверхности океана - очень интересный, важный и активно развивающийся раздел моделирования. С помощью распределенных вычислений на GPU можно добиться вычисления очень большой площади поверхности в режиме реального времени с неплохой точностью. В данной работе была реализована самая простая статистическая модель волны, однако даже эта модель позволяет просчитать поведение волн с хорошей точностью.

Список литературы

[1] Tessendorf, J. 2001. Simulating Ocean Water. ACM SIGGRAPH.

[2] NVIDIA. 2011. Ocean Surface Simulation. NVIDIA Graphics SDK 11 Direct3D.

[3] Chin-Chih Wang, Jia-Xiang Wu, Chao-En Yen, Pangfeng Liu, Chuen-Liang Chen. Ocean Wave Simulation in Real-time using GPU.

[4] Farber, R. 2011, CUDA Application Design and Development, Applications of GPU computing, Elsevier Science

[5] David B. Kirk, Wen-mei W. Hwu. 2012, Programming Massively Parallel Processors: A Hands-on Approach, Newnes