Визуализация водной поверхности. Быстрое преобразование Фурье на GPU

Автор:
Дмитрий Трифонов

Содержание

1. Введение
2. Плоская поверхность воды
   2.1 Рельефное текстурирование
   2.2 Добавление локальных отражений и преломлений
   2.3 Комбинирование отражений и преломлений
   2.4 Модель освещения
3. Быстрое преобразование Фурье на GPU
   3.1 Дискретное преобразование Фурье
   4.2 Быстрое преобразование Фурье
   4.3 Реализация с использованием шейдеров
   4.4 Производительность
4. Статистическая модель синтеза поверхности океана
   4.1 Основные положения. Примеры использования
   4.2 Спектр Филлипса
   4.3 Нормали и “острые” волны
5. Визуализация поверхности океана
   5.1 Проекционная сетка
   5.2 Пересечение с пирамидой видимости
   5.3 Распределение детализации
   5.4 Коррекция по краям
   5.5 Фильтрация
   5.6 Отражения и преломления
   5.7 Океанский шейдер
6. Заключение
7. Источники
8. Исходный код

1. Введение

Реалистичная визуализация водной поверхности один из самых эффективных способов сделать 3D приложение привлекательным. Но многие алгоритмы синтеза поверхности, как правило, сложны в реализации и требовательны к аппаратуре, поэтому к вопросу выбора алгоритма стоит подойти с параноидальной осторожностью.

Ранние методы визуализации воды в реальном времени, основывались на предположении о том, что водная поверхность плоская. Иллюзия волн создавалась за счет рельефного текстурирования с использованием заранее сгенерированных карт для нормалей и высот. Такой подход до сих пор часто используется. Он отлично подходит для визуализации небольших и спокойных поверхностей воды, например озер и луж. Кроме того, плоская поверхность воды сильно упрощает визуализацию отражений, преломлений, физическую модель, требует минимального колличества полигонов и позволяет легко добавлять спецэффекты, например, волны от объектов.

К сожалению такой подход очень плохо справляется с визуализаций больших открытых водоемов, а именно тех, где ожидаешь увидить хоть сколько-нибудь крупные волны. Как только у 3D ускорителей появилась возможность изменять геометрию на лету, без серьезного падения производительности, рельефное текстурирование было вытеснено анимацией вершин. Но получаемые поверхности все равно приходилось делать достаточно плоскими, потому что для реалистичного синтеза больших волн требуются более изощренные подходы, нежели анимация за счет деформирования исходных карт нормалей и высот. Таким подходом, например, является параметрическая модель "Gernster Waves", но хотя модель применялась на практике, широкого распространения не получила. Благо, существует гораздо более реалистичная статистическая модель, которой будет посвящена основная часть статьи.

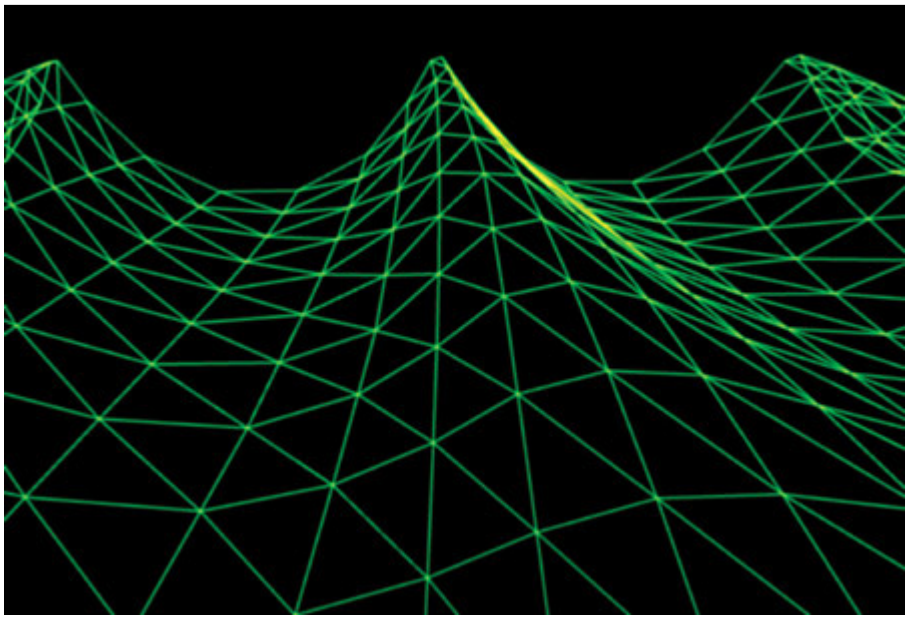

Рисунок 1. Поверхность, синтезированная моделью "Gernster Waves".

2. Плоская поверхность воды

2.1 Рельефное текстурирование

Самым простым методом визуализации водной поверхности, как уже было сказано, является рельефное текстурирование плоскости анимированными или деформируемыми картами нормалей. Обычно используется периодическая текстура с фрактальным шумом, возможно использование объемной текстуры.

При отрисовке текстурная координата зависит от положения точки поверхности и от времени. В качетсве функции иногда используют формулу турбулентности, но использование более простых подходов также дает приемлемый результат. Вычисленная нормаль используется как текстурная координата для выборки из кубической текстуры окружения и для подсчета освещения.

$$h(\boldsymbol{p},t) = \sum_{k=0}^{N} \frac{1}{2^k} |2 \cdot noise(2^k \boldsymbol{p}, 2^k t) - 1|$$

Формула 1. Функция турбулентности. P – позиция точки, t - время. Источник

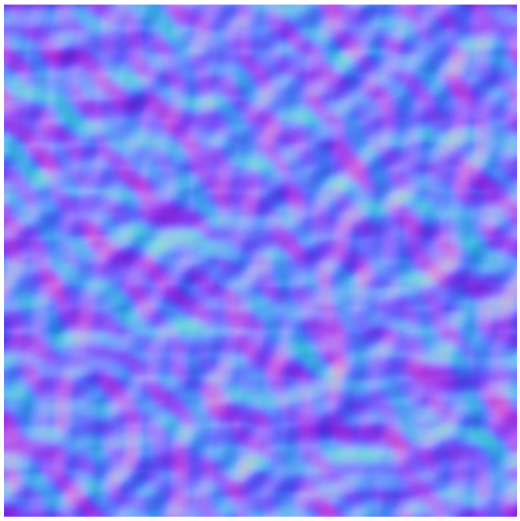

Рисунок 2. Статическая карта нормалей, используемая для анимации водной поверхности

2.2 Добавление локальных отражений и преломлений

В качестве кубической текстуры для задания отражений обычно используют только текстуру окружения. Рисование всей сцены в кубическую текстуру для создания динамических отражений невозможно, потому что кубом придется объять слишком большую область, занимаемую водоемом. Использование техники для визуализации отражений от плоского зеркала с использованием буфера маски тоже не годится, потому что с нельзя наложить рябь или каким-либо другим образом деформировать полученное отражение.

Обычно используют следующий алгоритм:

1. Отражаем камеру относительно поверхности воды:

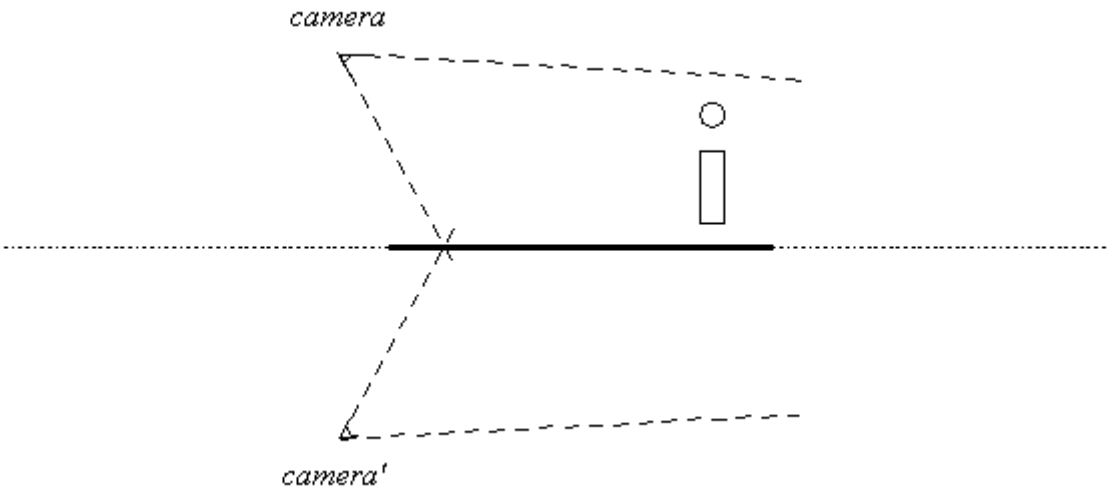

Рисунок 3. Отражение камеры для визуализации локальных отражений

2. Устанавливаем плоскость отсечения, чтобы подводные объекты или части объектов не отрисовывались.
3. Отображаем сцену в текстуру, используя отраженную камеру:

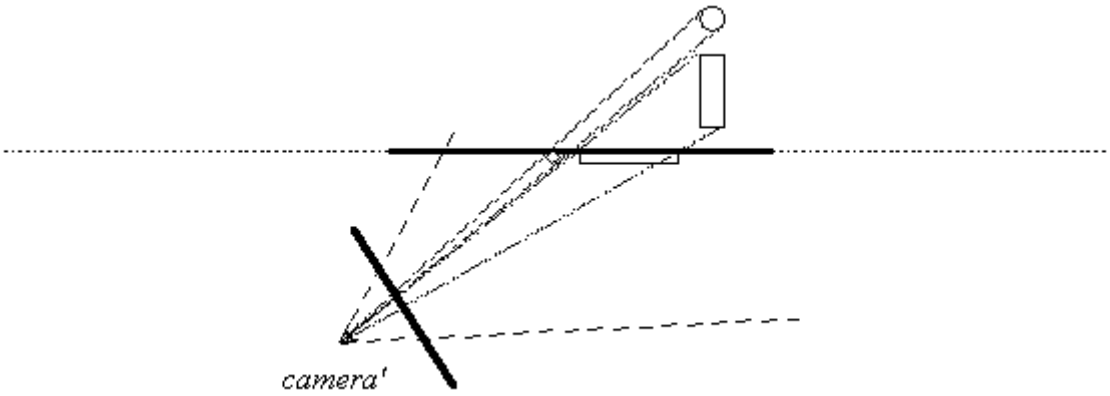

Рисунок 4. Проецирование сцены в карту отражений.

4. Устанавливаем специальную матрицу, используемую для получения текстурных координат в карте отражений по координате точки поверхности воды. С использованием OpenGL можно вычислить это матрицу следующим образом:

```
5.     glLoadIdentity ();
6.     glTranslatef   ( 0.5, 0.5, 0 );  // из квадрата [-1,1]x[-1,1]
7.     glScalef       ( 0.5, 0.5, 1 );  // в квадрат [0,1]x[0,1]
8.     glMultMatrixf  ( projMat );      // перспективная матрица камеры
   отражений
9.     glMultMatrixf  ( modelViewMat ); // видовая матрица камеры
   отражений
```

10. Рисуем поверхность воды, используя полученную карту с отражениями и матрицу из предыдущего этапа. Умножая матрицу на позицию точки поверхности, получаем проективную тексурную координату в карте отражений. Для наложения ряби можно сдвигать текстурную координату на величину нормали в точке водной поверхности. В некоторых случаях размытие стоит производить в экранной плоскости или ввести зависимость размытия от расстояния, чтобы поверхность расположенная далеко от наблюдателся не выглядела слишком плоской. Все вышесказанное эквивалентно командам GLSL:

```
11.      // Определение текстурных координат через матрицу отражений.
12.      vec4 projTexCoord = reflectionMatrix * waterVertex;
13.
14.      // Размытие не в экранной плоскоти
15.      projTexCoord.xy += distortConstant * normal.xy;
16.      vec4 texel       = texture2DProj(reflectMap, projTexCoord);
17.
18.      // Размытие в экранной плоскости
19.      vec2 texCoord = projTexCoord.xy / projTexCoord.w + distortConstant
   * normal.xy;
20.      vec4 texel    = texture2D(reflectMap, texCoord);
```

Замечение: Учитывая специфику модельно-видового преобразования, совпадение матриц проекции и то, что водная поверхность представляет собой плоскость, текстурную координату для выборки из текстуры отражений можно определить и не вводя дополнительной матрицы. По сути в качестве текстурной координаты просто используется относительная позиция пикселя на экране. Матрицу для отражений будем получать следующим образом:

```
// отражение точки относительно плоскости:
vec3 reflect(vec3 vec, vec4 plane) {
    return vec - 2 * (dot(vec, plane.xyz) - plane.w) * plane.xyz;
```

```
	}

	vec3 position  = reflect( masterCamera->getPosition().xyz(),
waterPlane );
	vec3 direction = reflect( masterCamera->getDirection(),
vec4(waterPlane.xyz, 0.0) );
	vec3 up        = reflect( masterCamera->getUp(),
vec4(waterPlane.xyz, 0.0) );

	// look_at – создает матрицу такую же как и gluLookAt.
	reflectCamera.setViewMatrix( look_at(position, position +
direction, up) );
```

Тогда текстурная координата в карте отражений будет находиться следующим образом(GLSL):

```
	vec4 projTexCoord = gl_ModelViewProjectionMatrix * waterVertex;

	// После перспективного деления получаются нормализованные
координаты,
	// они в квадрате [-1,1]x[-1,1]. Их надо перевести в текстурные,
	// которые в квадрате [0,1]x[0,1]. В этом и суть операций
glTranslatef, glScalef
	// для получения описанной матрицы для получения текстурных
координат отражений.
	vec2 texCoord = 0.5 * projTexCoord.xy / projTexCoord.w +
vec2(0.5);

	// В описанном алгоритме получения матрицы для камеры отражения
инвертируется
	// ось камеры OY (или up, это и нужно), но вместе с ней и OX (а
это уже не нужно),
	// поэтому надо инвертировать координату x.
	texCoord.x = 1.0 – texCoord.x;
```

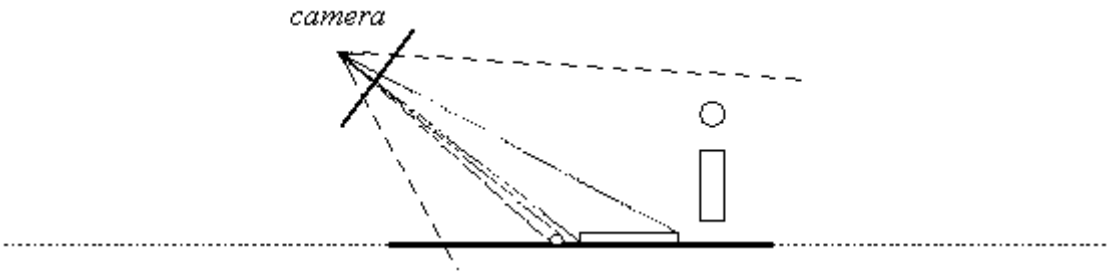

Рисунок 5. Восстановление отражений из текстуры.

Нетрудно заметить, что точно таким же образом можно получить и карту для преломлений. Нужно только использовать не отраженную камеру, а обыкновенную и задавать плоскость отсечения таким образом, чтобы отсечь все объекты выше поверхности воды. В конце концов можно рисовать всю сцену в текстуру, как скорей всего и будет делаться, чтобы добавить постэффекты, и использовать её в качестве карты преломлений.

2.3 Комбинирование отражений и преломлений

Использование текстуры локальных отражений позволяет визуализировать отражения от объектов любых размеров вне зависимости от их расположения в достаточно хорошем качестве, но у этого метода так же есть и серьезный недостаток.

Алгоритм не позволяет физически точно определять куда попадет луч после отражения от водной поверхности. Для наложения ряби, текстурные координаты для выборок из текстуры отражений просто сдвигают на величину проекции нормали на плоскость поверхности воды или на величину какой-либо шумовой функции. Отличить отражения, "размытые" таким образом, от настоящих крайне трудно, но сама водная поверхность выглядит очень плоско. К счастью, этот недостаток можно скомпенсировать. Для этого делаются две выборки: из текстуры отражений и из текстуры окружения. Если первая выборка “пустая”, то нет объекта для локального отражения, - в качестве цвета берётся вторая выборка, из кубической текстуры окружения. На GLSL это может выглядеть более понятно:

```
vec4 localRefl = texture2D(reflectMap, reflectTexCoord.xy);  // Выборка из
текстуры локальных отражений
vec3 gobalRefl = texCUBE(skybox, reflectVec);                // Выборка из
текстуры окружения
vec3 color     = mix(gobalRefl, localRefl.rgb, localRefl.a); // Линейная
интерполяция между цветами
```

Стоит не забывать, что при использовании этого комбинированного подхода, во время рендеринга в текстуру отражений, надо отключить отрисовку скайбокса или других объектов глобального окружения. В противном случае теряется весь смысл совмещения глобальных и локальных отражений, потому что все они будут интерпретирваться как локальные.

Рисунок 6. Использование только локальных отражений

Рисунок 7. Использование комбинации локальных и глобальных отражений

Для определения доли отраженной и преломленной энергии служат коэффициенты Френеля. Они определяют какая часть энергии светового пучка отразилось от поверхности в точке раздела двух сред, а какая преломилась.

$$c = \cos\theta \cdot \frac{\eta_i}{\eta_t}$$

$$g = \sqrt{1 + c^2 - \left(\frac{\eta_i}{\eta_t}\right)^2}$$

$$R(\theta) = \frac{1}{2} \cdot \left(\frac{g-c}{g+c}\right)^2 \cdot \left[1 + \left[\frac{c(g+c) - \left(\eta_i/\eta_t\right)^2}{c(g-c) + \left(\eta_i/\eta_t\right)^2}\right]^2\right]$$

Формула 2. Коэффициенты Френеля. η_i,η_t - коэффициенты преломления сред. θ - угол между нормалью и падающим лучом.

Обычно точная формула не используется, ввиду её вычислительной сложности. Есть множество её приближений, используемых для графических вычислений в реальном времени.

$$R(\theta) = R(0) + \big(1 - R(0)\big)(1 - \cos\theta)^5$$
$$R(\theta) = 0.05 + 0.95 \cdot (1 - saturate(\cos\theta))^4$$
$$R(\theta) = clamp(0.1,0.9, -\cos\theta)$$
$$R(\theta) = \frac{1}{(1 + \cos\theta)^8}$$
$$R(\theta) = 1 - \cos\theta$$

Таблица 1. Различные формулы для приближенного вычисления коэффициентов Френеля. θ - угол между нормалью и падающим лучом.

Используя эти коэффициенты можно вычислить финальный цвет воды в точке поверхности, комбинируя цвета отражения и преломления(или же цвет воды).

2.4 Модель освещения

Для расчета освещения водной поверхности хорошо подходит модель Блинна-Фонга. При её использовании на поверхности возникают солнечные дорожки, а не овальные блики как при использовании модели Фонга.

Параметры для освещения обычно подбираются вручную в зависимости от конкретной сцены. Подбор этих параметров на основе характерных для водоемов физических величин практически невозможен из-за сложности физических явлений сопряженных с освещением воды. К примеру, модель освещения изменяется на протяжении суток, зависит от химического состава воды и погодных условий.

На практике, часто строят модель освещения исходя из следующих явлений:

1. В глубоком водоеме цвет водной поверхности меняется от зеленоватого до тёмно синего в соответствии с углом между направлением обзора на точку поверхности и нормалью в ней.
2. В зависимости от видимой толщи воды можно определять коэффициент для смешивания цвета воды в этой точке и выборки из текстуры преломлений. Для вычисления видимой толщи воды, достаточно буфера глубины, используемого при создании текстуры с преломлениями. Из значения в точке буфера нужно вычесть расстояние до водной поверхности.

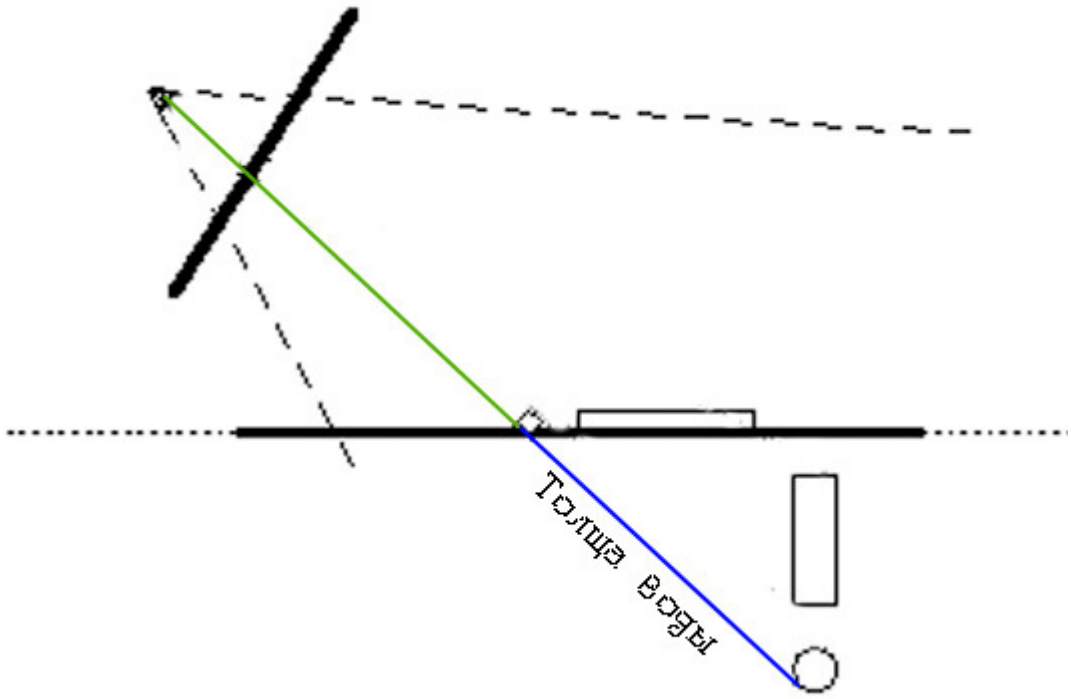

Рисунок 8. Определение видимой толщины воды.

Разумеется, такой подход зачастую может оказаться излишним. Редкая игра, в которой художники украшают дно изобилием скважин, рифов и обломков пиратских кораблей. Смешивание цвета из текстуры преломлений и цвета воды с фиксированным коэффициентом или коэффициентом из заранее изготовленной карты прозрачности может дать вполне приемлемый результат. В конце концов прозрачность на основе альфа смешивания тоже может сгодиться, но стоит учесть, что при таком подходе нельзя добавить искажения при преломлении.

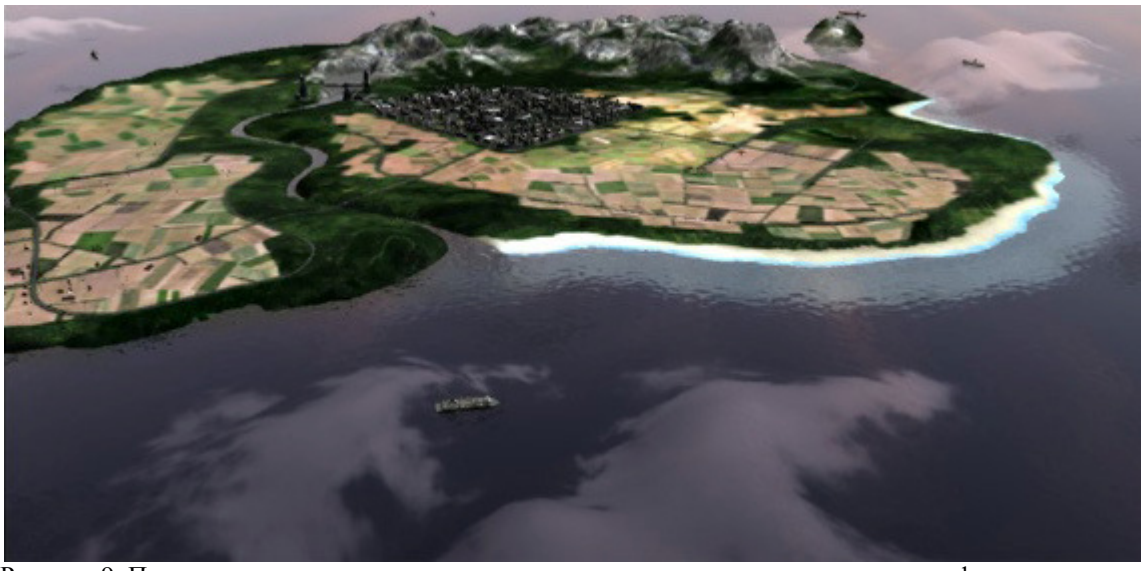

Рисунок 9. Пример визуализации океана с преломлениями на основе смешивания с фиксированным коэфициентом.

3. При визуализации крупных водоемов важно создать иллюзию бесконечности. Обычно, для таких целей добавляется рассчет тумана. За счет него поверхность воды плавно сливается с небом(тут важно, чтобы цвет именно сливался, вероятно придется добавить рассчет тумана так же и для неба).

Суммируя все вышеперечисленные допущения и факторы, наиболее полную модель освещения можно изобразить в виде схемы:

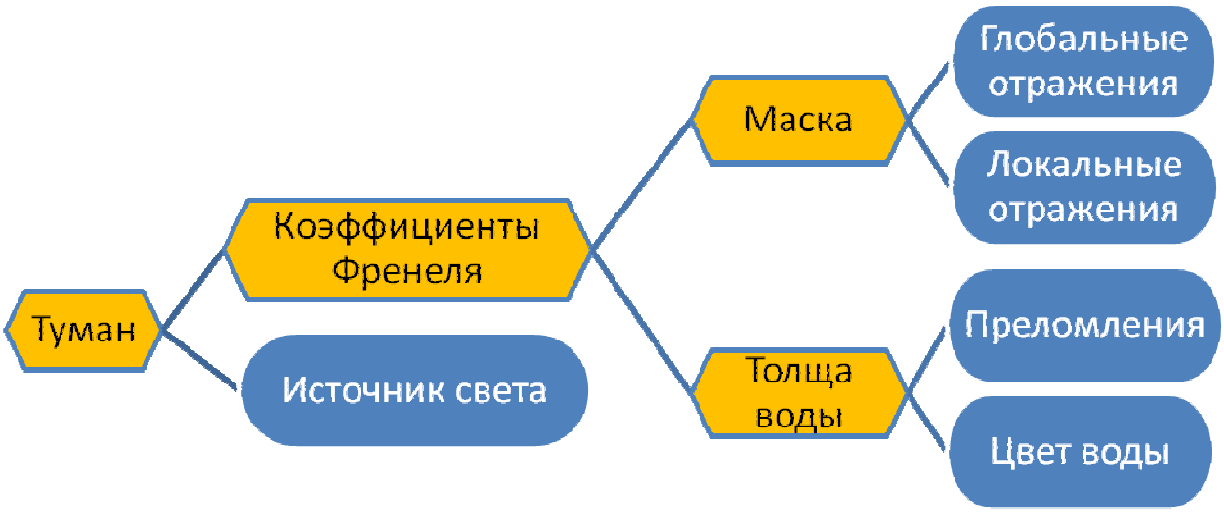

Рисунок 10. Модель освещения водной поверхности.

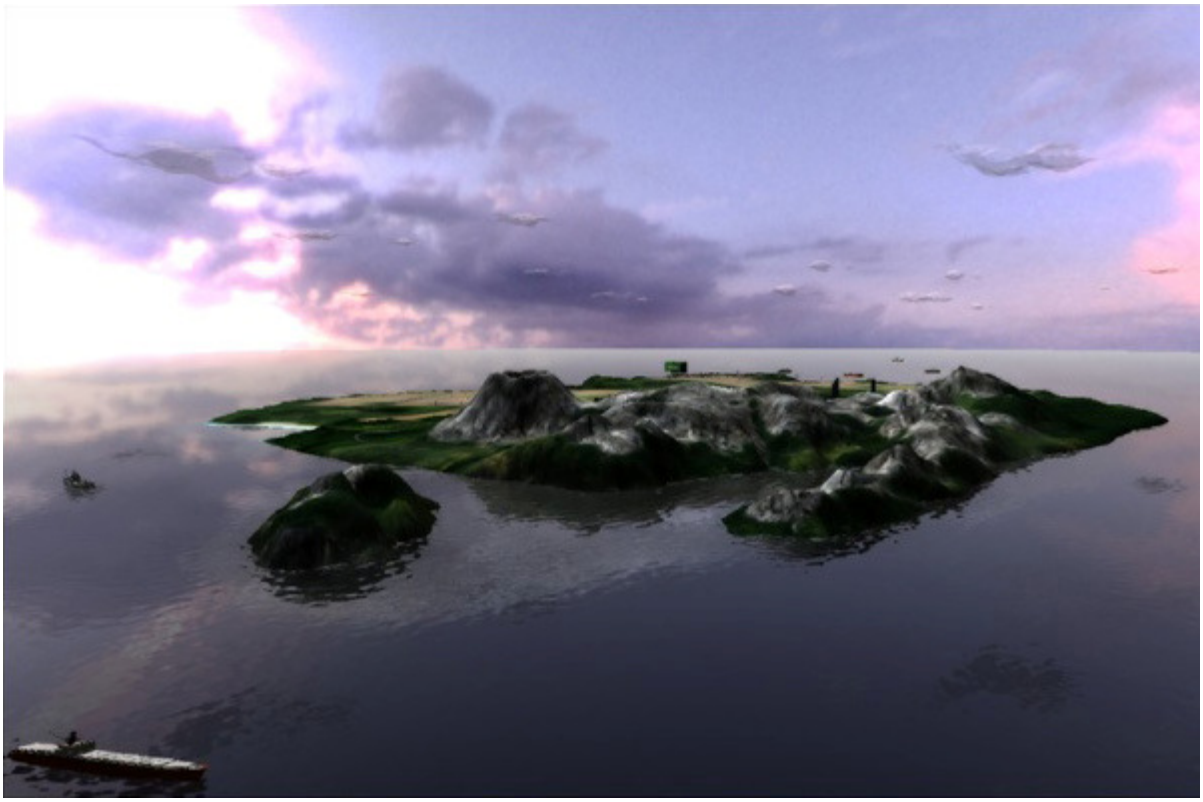

Рисунок 11. Пример визуализации плоской поверхности воды.
Замечение: Водная поверхность визуализирована с использованием описанных алгоритмов. Сами презентации разработаны в компании Eligovision, поэтому исходные коды этих приложений, разумеется, я не имею права предоставлять.

3. Быстрое преобразование Фурье на GPU

3.1. Дискретное преобразование Фурье

Рассмотрим дискретное одномерное комплексное преобразование Фурье, далее ДПФ:

$$F(k) = \sum_{n=0}^{N-1} f(n) W_N^{kn}, f(k) = \frac{1}{N} \sum_{n=0}^{N-1} F(n) W_N^{-kn}$$
$$W_N^{kn} = e^{-i\pi kn/N}, k = 0, \ldots, N-1$$

Формула 3. Прямое и обратное дискретное комплексное преобразование Фурье. Весовые коэффициенты.

Если вычислять преобразование Фурье напрямую, потребуется $O(N^2)$ операций. При реализации статистической модели синтеза поверхности океана будут использоваться комплексные сетки размера N=128x128(N^2 = 268 435 456) и больше, - такая сложность не приемлема для вычислений в реальном времени.

3.2 Быстрое преобразование Фурье

Алгоритм быстрого преобразование Фурье(Cooley and Tukey "Decimation In Time") – далее БПФ, основывается на стратегии "разделяй и властвуй". Он не самый эффективный по количеству операций, но легко распараллеливается и не требует сложных вспомогательных вычислений, реализация которых на GPU может оказаться затруднительной или даже невозможной.

Рассмотрим первый шаг алгоритма. Просуммируем отдельно слагаемые с четными и нечетными степенями весовых коэффициентов, полагая что N делится на два:

$$F(k) = \sum_{n=0}^{N-1} f(n) W_N^{kn} = \sum_{n=0}^{\frac{N}{2}-1} f(2n) W_N^{k2n} + \sum_{n=0}^{\frac{N}{2}-1} f(2n+1) W_N^{k(2n+1)}$$
$$= \sum_{n=0}^{\frac{N}{2}-1} f^e(n) W_N^{2kn} + W_N^k \sum_{n=0}^{\frac{N}{2}-1} f^o(n) W_N^{2kn}$$

Теперь, учитывая свойство весовых коэффициентов, преобразуем получившееся выражение:

$$W_N^2 = e^{-i\left(\frac{2\pi}{N}\right)2} = e^{-i\left(\frac{2\pi}{N/2}\right)} = W_{N/2}$$

$$F(k) = \sum_{n=0}^{\frac{N}{2}-1} f^e(n) W_N^{2kn} + W_N^k \sum_{n=0}^{\frac{N}{2}-1} f^o(2n) W_N^{2kn}$$

$$= \sum_{n=0}^{\frac{N}{2}-1} f^e(n) W_{N/2}^{kn} + W_N^k \sum_{n=0}^{\frac{N}{2}-1} f^o(n) W_{N/2}^{kn}$$

Рассмотрим ещё несколько вспомогательных свойств весовых коэффициентов:

$$W_N^{k+l\cdot N} = e^{-i\frac{2\pi(k+l\cdot N)}{N}} = e^{-i\frac{2\pi k}{N}} e^{-i\frac{2\pi l\cdot N}{N}} = e^{-i\frac{2\pi k}{N}} e^{i2\pi l} = e^{-i\frac{2\pi k}{N}} = W_N^k$$

$$W_N^{k-N/2} = e^{-i\frac{2\pi(k-N/2)}{N}} = e^{-i\frac{2\pi k}{N}} e^{-i\frac{2\pi(-N/2)}{N}} = e^{-i\frac{2\pi k}{N}} e^{i\pi} = -W_N^k$$

$$W_N^{n(k+l\cdot N)} = W_N^n W_N^{k+l\cdot N} = W_N^n W_N^k = W_N^{nk}$$

Используя последние равенства несложно свести задачу ДПФ от N коэф-фициентов к ДПФ от N/2 коэффициентов:

$$F(k) = \sum_{n=0}^{\frac{N}{2}-1} f^e(n) W_{N/2}^{kn} + W_N^k \sum_{n=0}^{\frac{N}{2}-1} f^o(n) W_{N/2}^{kn}$$

$$= \begin{cases} F^e(k) + W_N^k F^o(k) & k < N/2 \\ F^e(k - N/2) - W_N^{k-N/2} F^o(k - N/2) & k \geq N/2 \end{cases}$$

Формула 4. Разбиение ДПФ от N элементов на два.

Таким образом, вычисляя F(k) только для k=0..N/2 мы можем заполнить весь массив коэффициентов, уменьшив тем самым сложность вдвое.

Оба получившихся ДПФ можно разбить ещё на два, если N кратно четырем. Если N является степенью двойки, то можно свести искомое преобразование Фурье к ДПФ от одного элемента. Сложность БПФ в таком случае будет Θ(N log N).

Все вышеперечисленные рассуждения легко перенести на двумерный случай. Но лучше воспользоваться свойством многомерного преобразования Фурье: для вычисления многомерного ДПФ можно последовательно применять одномерные преобразования в каждом направлении. То есть, для двумерного случая сначала надо произвести

преобразование Фурье для всех строк, потом для всех столбцов. Такой подход существенно уменьшит вычислительную сложность.

3.3 Реализация с использованием шейдеров

Наивное исполнение алгоритма из предыдущего раздела предполагает рекурсивное вычисление ДПФ меньшего порядка. Это не самый эффективный способ, кроме того на GPU нет стека, а следовательно нет возможности выполнять какие-либо рекурсивные алгоритмы.

Вычисление БПФ без рекурсии похоже на алгоритм сортировки слияниями. Используется два массива: один для хранения исходных данных, другой для результата. На каждом шаге, производим операцию над первым массивом и записываем результат во второй, потом меняем их местами и переходим к следующей итерации.

На нулевом шаге алгоритма в исходном массиве N коэффициентов ДПФ, фактически N преобразований Фурье размера 1. На первом необходимо посчитать N/2 преобразований Фурье размера 2, на втором N/4 размера 4 и так далее. Итого, на i'м шаге алгоритм работает $N/2^i$ разбиениями исходной последовательности длины 2^i.

Для слияния двух ДПФ в одно, более высокого порядка, можно воспользо-ваться формулой 4, но необходимо ещё учитывать, что на каждом следующем шаге алгоритма будет меняться расположение коэффициентов преобразования. Классический метод индексации для БПФ называется bit-reversing. Суть его состоит в предварительной перестановке исходных элементов массива так, чтобы на каждой итерации коэффициенты для любого ДПФ были расположены последовательно.

Для реализации на GPU лучше избавиться от предварительной перестановки, используемой в этом методе. Есть способ определения индексов для итеративного вычисления БПФ без каких либо модификаций исходного массива.

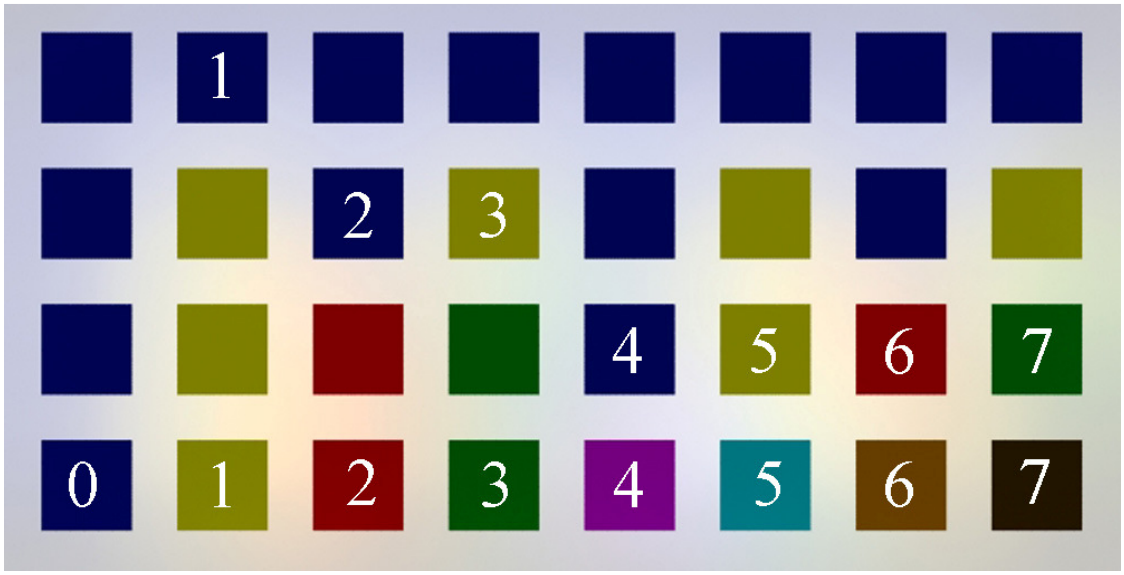

Рисунок 12. Разбиение ДПФ. Сначала поделили элементы на две группы: с четными индексами и с нечетными.
Затем каждую группу разбили ещё на две, уже смотря на четность или нечетность индекса внутри группы, и т. д.

Выше на рисунке изображено последовательное разбиение исходного массива: синяя группа разбивается на синюю и желтую, полученная синяя на синюю и красную, а желтая на желтую и зеленую и т. д. Если же идти снизу вверх, объединяя в группы ячейки простым суммированием, обратно тому как производилось их разбиение, то на любом уровне в каждой ячейке будет хранится сумма тех элементов, соответствующие ячейки которых имеют такой же цвет. Таким образом, сумма значений 2 и 3 ячейки на втором уровне – есть сумма всех элементов массива. На самом верхнем уровне, в каждой ячейке хранится сумма всех элементов массива.

Теперь необходимо, чтобы объединение ячеек производило объединение ДПФ. Это возможно сделать, если при суммировании на каждом этапе домножать слагаемые на необходимые весовые коэффициенты. Формула 5, как и формула 4, объединяет два преобразования Фурье, полученных на предыдущем этапе, но уже с учетом индексации в массиве.

$$A_i[n] = A_{i-1}\left[n + u\,N/2^i\right] + W_{2^i}^{u} A_{i-1}\left[n + u\,N/2^i + N/2^i\right]$$

$$u = n \operatorname{div} N/2^i$$

Формула 5. Определение индексов элементов для БПФ, i – номер итерации, N – размер ДПФ. Индексы берем по модулю N.

Строгое обоснование приведенной выше формулы и более подробное описание метода в целом можно посмотреть в статье “The FFT on a GPU” Kenneth Moreland & Edward Angel. Ниже приведен формальный пример вычисления второго элемента ДПФ, с использованием этой формулы, для N=8:

$$\begin{aligned} A_3[1] &= A_2[2] + A_2[3]W_8^1 = (A_1[4] + A_1[6]W_4^1) + (A_1[5] + A_1[7]W_4^1)W_8^1 \\ &= (A_0[0] + A_0[4]W_2^1) + (A_0[2] + A_0[6]W_2^1)W_4^1 \\ &+ ((A_0[1] + A_0[5]W_2^1) + (A_0[3] + A_0[7]W_2^1)W_4^1)W_8^1 \\ &= A_0[0] + A_0[1]W_8^1 + A_0[2]W_8^2 + A_0[3]W_8^3 + A_0[4]W_8^4 \\ &+ A_0[5]W_8^5 + A_0[6]W_8^6 + A_0[7]W_8^7 \end{aligned}$$

Алгоритм для вычисления двумерного БПФ средствами графического API выглядит следующим образом:

```
renderTarget->SetDrawBuffer(0, pingPongTexture[0]);
renderTarget->SetDrawBuffer(1, pingPongTexture[1]);
BindRenderTarget(renderTarget);

int numIterations = (int)(log( (double)N ) / log(2.0) + 0.5);
fftProgram->Bind();  // Шейдер, выполняющий формулу 5
sizeUniform->Set(N); // Размер преобразования

// БПФ по строкам
int curPing = 0;
directionUniform->Set(1.0f, 0.0f); // Направление БПФ
for(int i = 1; i <= numIterations; ++i)
{
    inputUniform->Set(0, pingPongTexture[curPing]); // Входной массив
    numIterationUniform->Set(i); // Номер итерации
    curPing = !curPing;
```

```
    renderTarget->SetDrawBuffer(curPing); // Меняем выходную текстуру
    gridQuad->Draw(QUADS);
}

// БПФ по столбцам
directionUniform->Set(0.0f, 1.0f); // Установить направление по столбцам
for(int i = 1; i <= numIterations; ++i)
{
    inputUniform->Set(0, pingPongTexture[curPing]);
    numIterationUniform ->Set(i);
    curPing = !curPing;
    renderTarget->SetDrawBuffer(curPing);
    gridQuad->Draw(QUADS);
}
```

Шейдеры, выполняющие преобразование Фурье(выполняется два двумерных преобразования Фурье, записанных в одной текстуре формата RGBA32F или RGBA16F - каждый элемент занимает два числа, т.к. элементы комплексные):

```
/=============== Vertex Shader ===============/
#extension GL_EXT_gpu_shader4 : enable // используем, если есть

uniform vec2  viewport;
uniform vec2  direction; // направление: столбцы или строки
uniform float N;         // размер преобразования(одномерного)

#if GL_EXT_gpu_shader4
noperspective varying vec2  index;
noperspective varying float linearIndex;
#else
varying vec2    index;       // двумерный индекс
varying float   linearIndex; // одномерный индекс
#endif

void main()
{
    vec2 gridVertex = N * gl_Vertex.xy;

    linearIndex = gridVertex.x;
    index       = gridVertex.x * direction.xy + gridVertex.y * direction.yx;
    gl_Position = vec4( 2.0 * ( index.xy / N - vec2(0.5) ), 0.0, 1.0 );
}

/=============== Fragment Shader ===============/
#extension GL_EXT_gpu_shader4 : enable

uniform sampler2D fftInput;

uniform vec2  direction;
uniform float N;
uniform float numPartitions; // колличество выполняемых ДПФ. Равно N / 2^i, i
- номер прохода

#if GL_EXT_gpu_shader4
noperspective varying vec2  index;
noperspective varying float linearIndex;
#else
varying vec2    index;
varying float   linearIndex;
#endif

#define PI     3.141592653
```

```
#define PI_2   (2.0 * PI)

// Просто умножение комплексных чисел
vec2 complex_mult(vec2 a, vec2 b)
{
    return vec2(a.x*b.x - a.y*b.y, a.x*b.y + a.y*b.x);
}

// Рассчет весового коэффициента
vec2 W(float partitionSize, float X)
{
    float angle = PI_2 * X / partitionSize;
    return vec2( cos(angle), sin(angle) );
}

void main()
{
    float k = floor(linearIndex / numPartitions);

    // fftIndex = ( iIndex + k * n ) % N
    vec2 fftIndex = index + direction * k * numPartitions;
    fftIndex     -= floor(fftIndex / N) * N;

    // Выбираем коэффициенты из текстуры
#if GL_EXT_gpu_shader4
    vec4 He = texelFetch(fftInput, ivec2(fftIndex), 0);  // Выборка без
фильтрации
    vec4 Ho = texelFetch(fftInput, ivec2(fftIndex + direction *
numPartitions), 0);
#else
    vec4 He = texture2DLod(fftInput, fftIndex / N, 0.0); // Выборка без
фильтрации
    vec4 Ho = texture2DLod(fftInput, (fftIndex + direction * numPartitions) /
N, 0.0);
#endif

    // Преобразуем коэффициенты по формуле 5
    vec2 Wn     = W(N / numPartitions, k);
    vec4 WHo    = vec4( complex_mult(Wn, Ho.xy), complex_mult(Wn, Ho.zw) );
// vec4(H*W, H*W)
    vec4 result = He + WHo;

    /* Сдвигаем все преобразование на некоторую величину.
     * Данный фрагмент понадобится при реализации синтеза водной поверхности.
     * Просто для вычисления ДПФ он не нужен.
    if ( numPartitions == 1.0 )
    {
        Wn = W(N, -linearIndex * N / 2.0);
        result.xy = complex_mult(Wn, result.xy);
        result.zw = complex_mult(Wn, result.zw);
    }*/

    gl_FragColor = result; // H = He + W * Ho
}
```

3.4 Производительность

В следующей таблице приведено сравнение производительности различных реализаций БПФ на GeForce 8800 GTX(Cory Quammen April 11, 2007). Рассмотренная реализация была протестирована на GeForce 8800 GT, при этом она производила два БПФ

параллельно, а затраченное время было разделено на два. Видеокарты примерно одного уровня, поэтому результаты включены в ту же таблицу.

	1D 16,384 milliseconds	1D 8,388,608 milliseconds	2D 1024^2 milliseconds	2D 2048^2 milliseconds
GPUFFTW	169	198	200	288
CUDA	0.701	?	6.781	32
Mitchell2003	?	?	12.205	?
Moreland2003	Cg error	Cg error	Cg error	Cg error
Sumanaweera2005	pbuffer error	pbuffer error	pbuffer error	pbuffer error
Рассмотренная реализация	?	?	12	47

Таблица 2. Сравнение производительности различных реализаций БПФ на GeForce 8800 GTX(GT)

Хотя и реализация Moreland2003 оказалась не рабочей в тестовой конфигурации(по крайней мере, если верить Cory Quammen), можно предположить что производительность будет практически идентичной рассмотренной, потому что реализация крайне похожая.

Стоит обратить внимание, что узким местом является пропускная способность памяти. При изменении формата текстуры для вычислений с GL_RGBA32F на GL_RGBA16F скорость работы увеличивается практически ровно в два раза. А использование же возможностей GL_EXT_gpu_shader4 не дает никаких преимуществ по производительности.

Исходя из этого, напрашивается оптимизация: переупорядочить выполнение операций таким образом, чтобы повторяющиеся выборки из текстуры находились в кэше. Из [5] видно, что например, при n=0 и n=N/2i выборки будут повторяться. На самом деле ровно половина выборок из текстуры актуальна.

Вероятно, такая оптимизация была сделана в реализации на CUDA, что и привело к двукратному увеличению производительности, но без CUDA такую оптимизацию провести не так просто. Кроме того, при небольших размерах сетки(128x128, 256x256), которые обычно используются при визуализации водной поверхности в реальном времени, вычисление БПФ занимает совсем немного времени(десятки, в худшем случае сотни микросекунд на данной конфигурации).

4. Статистическая модель синтеза поверхности океана

4.1 Основные положения. Примеры использования

Как можно было предположить, в основе статистической модели синтеза поверхности океана лежит дискретное преобразование Фурье. Волна представляется как сумма большого числа гармоник (т.е. фактически осуществлялось разложение в ряд Фурье по пространственным переменным x и y - [6]).

Проводя ДПФ над комплексной сеткой можно получить значения амплитуд волн в её узлах, то есть, карту высот. Полученная с помощью преобразования Фурье карта высот периодична и её можно “натягивать” на сколь угодно большую поверхность. Соответственно, чем больше размер сетки для преобразования Фурье, тем больше

полученная карта высот, тем более изощренная поверхность и менее заметна периодичность волн, что сильно увеличивает реалистичность.

$$h(\boldsymbol{x},t) = \sum_{k} \tilde{h}(\boldsymbol{k},t)e^{i\boldsymbol{k}x}$$

$$\boldsymbol{k} = (k_x,k_z), k_x = 2n/L_x, k_z = 2m/L_z$$

$$-M/_2 < m < M/_2 \quad -N/_2 < n < N/_2$$

Формула 6. Статистическая модель синтеза океанской поверхности. $L(L_x,L_z)$ – размер фрагмента водной поверхности. (N,M) – размеры сетки для преобразования Фурье.

Впервые модель была описана Джерри Тессендорфом (Jerry Tessendorf. “Simulating Ocean Water”) и использовалась впоследствии почти во всех проектах, требовавших визуализации реалистичной поверхности воды. Один из наиболее полных обзоров реализации этой модели и разнообразных эффектов, таких как каустики и пена, был произведен в работе Lasse Staff Jensen and Robert Golias. “Deep-Water Animation and Rendering”.

Выше был описан алгоритм вычисления ДПФ полностью на GPU, поэтому реализацию описанной модели можно так же переложить на графический ускоритель. Первая реализация, полностью производимая на GPU, была предложена в 2005 году: Jason L. Mitchell. “Real-Time Synthesis and Rendering of Ocean Water”. Но в тот момент вычислительная мощность 3D ускорителей была ещё недостаточно велика, чтобы модель начала широко применяться в приложениях реального времени. На самом деле, таких приложений и сейчас немного.

Рисунок 13. Фильм "Титаник" и игра Crysis. Пример использования статистической модели для визуализации океана.

4.2 Спектр Филлипса

Чтобы синтезировать реалистичную водную поверхность необходимо подобрать способ выбора коэффициенты для преобразования Фурье. Также необходимо решить вопрос о плавной анимации водной поверхности со временем.

Обычно коэффициенты ДПФ выбираются исходя из спектра Филлипса - [7]. В работе Тессендорфа есть и другие примеры спектров.

$$P_h(\boldsymbol{k}) = \frac{A\, e^{-\frac{1}{(kl)^2}}}{k^4}(\hat{\boldsymbol{k}}, \hat{\boldsymbol{w}})^2$$
$$\boldsymbol{k} = (k_x, k_z), k_x = 2n/L_x, k_z = 2m/L_z$$
$$-M/2 < m < M/2 \quad -N/2 < n < N/2$$
$$l = V^2/g, V = length(\boldsymbol{w})$$

Формула 7. Спектр Филлипса. ξ_r,ξ_i - выборки из нормального распределения. **w** – направление и сила ветра. **L** – размер фрагмента водной поверхности. (N,M) – размер ДПФ. A - нормирующая константа.

Замечание: Жирным выделены векторы, обычным шрифтом – скалярные величины либо длинны соответствующих векторов, символы с крышечкой – нормализованные векторы. Такая мнемоника будет использована и в дальнейшем.

Синтезировать спектр Филлипса каждый кадр не нужно. Можно просто ”анимировать” спектр с фиксированным по времени периодом.

$$\tilde{h}(\boldsymbol{k}, t) = \tilde{h}_0(\boldsymbol{k}, t)e^{i\omega(k)t} + \tilde{h}^*{}_0(-\boldsymbol{k}, t)e^{-i\omega(k)t}$$
$$\omega(k) = \sqrt{gk}$$

Формула 8. Анимация спектра Филлипса во времени. g – гравитационная постоянная.

Замечание: $\tilde{h}^*{}_0$ - сопряжение $\tilde{h}_0$. $\tilde{h}(\boldsymbol{k}, t)$ – исходные коэффициенты для [6].

Можно заметить, что коэффициенты преобразования Фурье комплексные, хотя высота волны представляется вещественным числом. На самом деле, комплексная компонента не лишняя, она означает фазу волны. Формула [6] как раз и описывает сдвиг фазы волны. Таким образом, сдвиг фаз всех гармоник дает плавную, и в то же время, непредсказуемую, анимацию поверхности.

4.3 Нормали и "острые" волны

Для вычисления нормалей можно воспользоваться разностью значений амплитуд в соседних узлах преобразования Фурье (соседних точек в карте высот). Но согласно Тессендорфу этот метод может давать недостаточно точные результаты при небольшой высоте волн, поэтому лучше воспользоваться ещё одним преобразованием Фурье для вычисления градиента, которое получается простым почленным дифференцированием ряда Фурье для амплитуд волн.

$$\nabla h(\boldsymbol{x},t) = \sum_{k} i\boldsymbol{k}\tilde{h}(\boldsymbol{k},t)e^{i\boldsymbol{k}x}$$

Формула 9. Вычисление градиента для водной поверхности. Обозначения такие же, как и в [4].

Замечание: Для восстановление нормали по градиенту в данном случае можно воспользоваться формулой: normal = normalize(vec3(-slope.x, 1.0, -slope.y)). Выводится из утверждения о том, что нормаль в точке (x,y,z) для функции F(x,y,z) = 0 совпадает с градиентом в этой точке.

Из-за того, что преобразование Фурье представляет собой сумму гармоник, то есть достаточно гладких функций sin, cos, получить остроконечные штормовые волны трудно. Для этого придется использовать очень большие размеры сеток коэффициентов для преобразования Фурье, то есть очень большое число гармоник. К счастью есть "искусственный" способ придать волнам остроконечный профиль.

Суть метода состоит в сдвиге точек поверхности воды так, чтобы “сжать” волны. Величину сдвига можно определить из следующего преобразования Фурье, которое очень напоминает ДПФ для вычисления градиента.

$$D(\boldsymbol{x},t) = \sum_{k} -i\hat{\boldsymbol{k}}\tilde{h}(\boldsymbol{k},t)e^{i\boldsymbol{k}x}$$

Формула 10. Вычисление величины сдвига точек поверхности воды, для придания волнам остроконечного профиля.

Замечание: $\hat{\boldsymbol{k}}$ – нормализованный вектор **k**.

При этом использование ДПФ для вычисления градиента и нормалей уже не годится. Но большого смысла оно и не имеет, потому что остроконечные волны бывают только при достаточно сильном ветре. В таких условиях волны крупнее и вычисление нормали через разницу амплитуд в соседних точках дает достаточно точный результат. Это обстоятельство довольно удобно, потому что и при слабом и при сильном ветре разумно вычислять только два преобразования Фурье, которые укладываются в одну четырехкомпонентную текстуру.

5. Визуализация поверхности океана

5.1 Проекционная сетка

Визуализация огромной поверхности океана достаточно ресурсоемкая задача. Во-первых, из-за большого числа видимых пикселей и сложности фрагментного шейдера, используемого при отрисовке. Во-вторых, из-за выборок из текстуры в вершинном шейдере. Соответственно первая проблема остро встает при больших разрешениях, а вторая для более старых видеокарт.

В теории со второй проблемой бороться несколько проще. Во-первых, можно использовать отложенный рендеринг (deffered shading) и избавиться от визуализации всей поверхности каждый кадр. Во-вторых, нужно реализовать грамотную систему распределения уровня деталлизации и использовать как можно меньшее колличество вершин.

Одним из самых оптимальных способов уменьшения уровня детализации в зависимости от расстояния до наблюдателя является использование проекционной сетки.

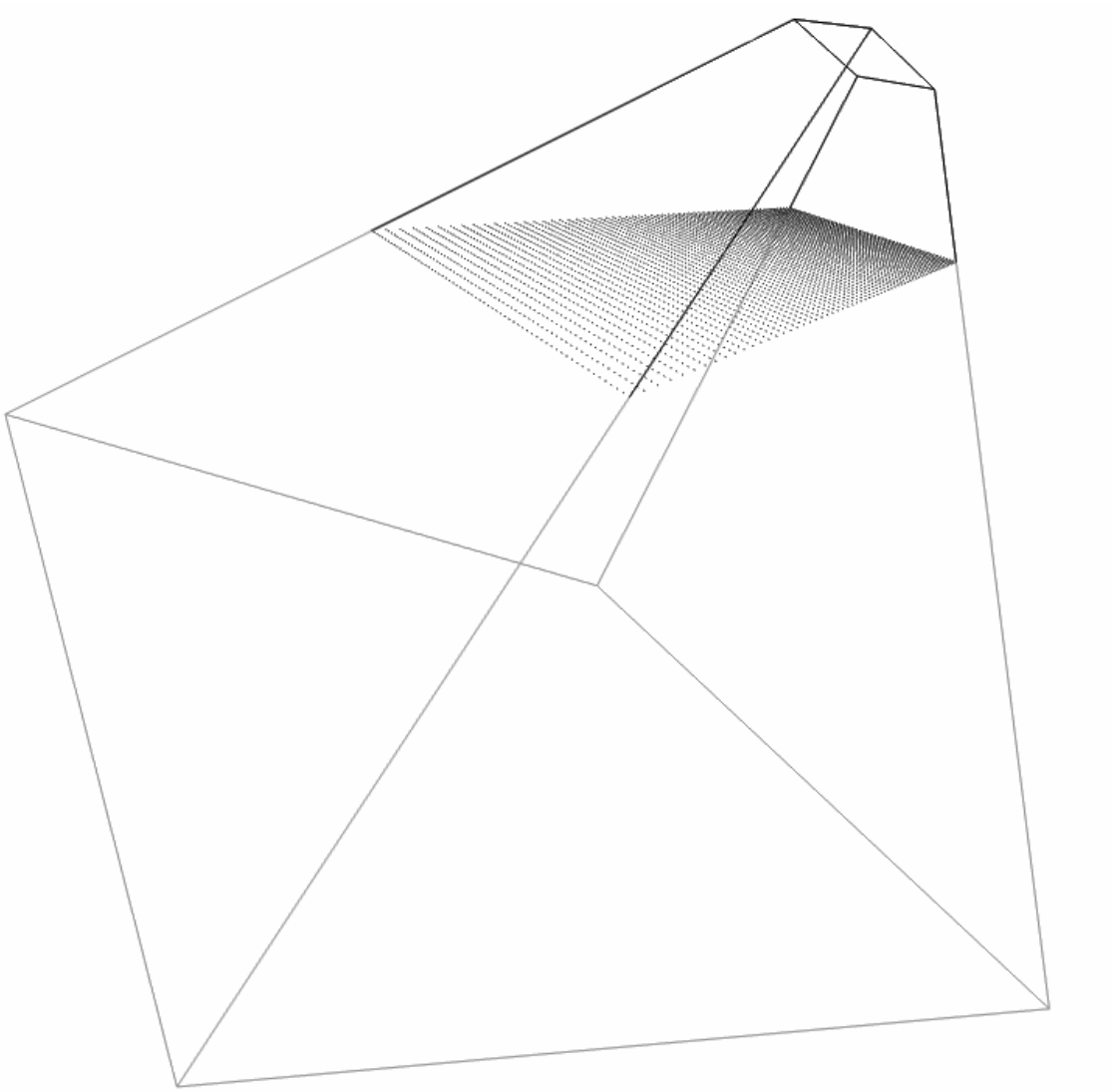

Рисунок 14. Проекционная сетка.

В классическом варианте алгоритма экран заполняется равномерной сеткой полигонов. Затем производится перевод сетки из экранных координат в видовые и проецирование на плоскость воды. На самом деле, алгоритм, не смотря на кажущуюся простоту, таит в себе огромное количество проблем. Так, например, разработчики CryEngine 2 отказались от его использования.

Вместо проецирования сетки на плоскость воды, можно её просто "прикрепить" к камере. Таким образом, камера будет двигаться вместе с фрагментом сетки, причем более деталлизированная часть будет ближе к наблюдателю. Распределение деталлизации в таком случае будет менее эффективным, зато такой подход значительно проще в реализации.

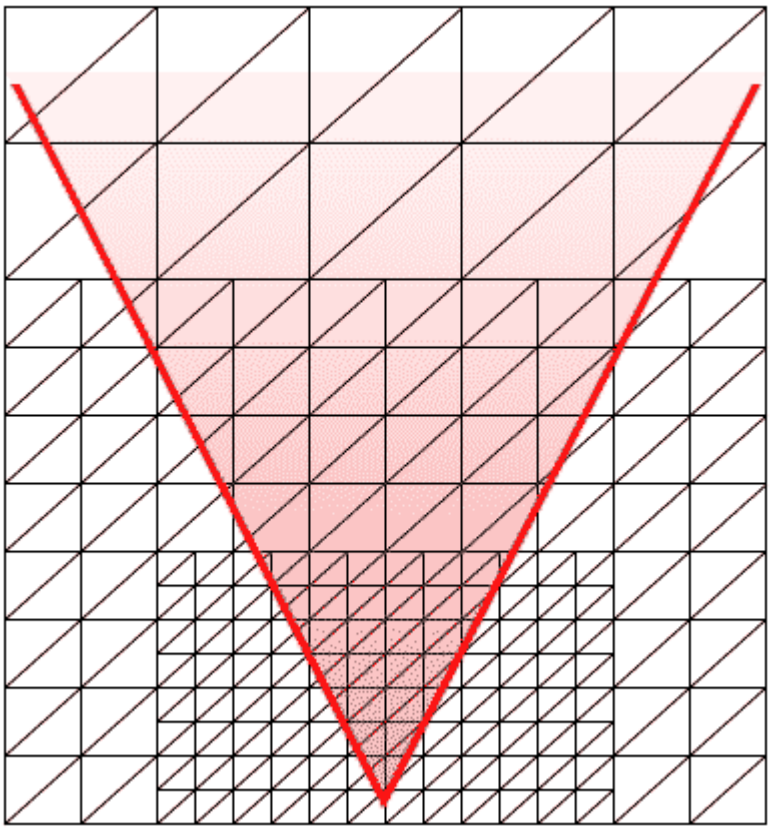

Рисунок 15. Сетка привязанная к камере.

5.2 Пересечение с пирамидой видимости

В некоторых случаях сетка не проецируется на плоскость воды или при её проецировании, полученная трапеция имеет слишком большую площадь. Второй случай плох, потому что сетка полигонов слишком растянется и будет иметь недостаточную плотность близи камеры. Будут видны артефакты.

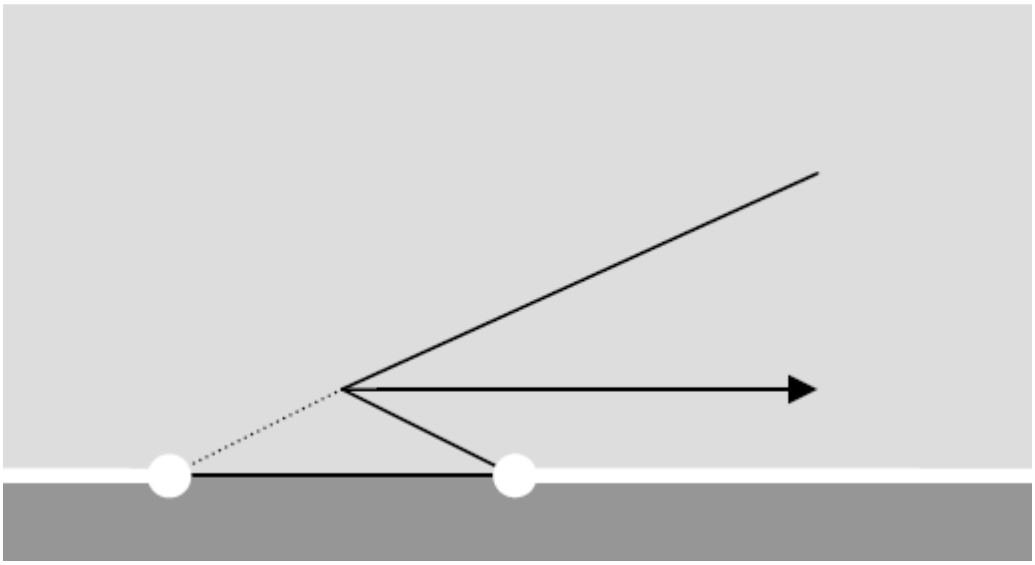

Рисунок 16. Случай, когда нет возможности правильно спроецировать экранную плоскость на плоскость воды.

С этим можно бороться, поворачивая камеру для проекции так, чтобы она смотрела под достаточно большим углом на плоскость воды, либо каким-то образом учитывать дальнюю плоскость отсечения.

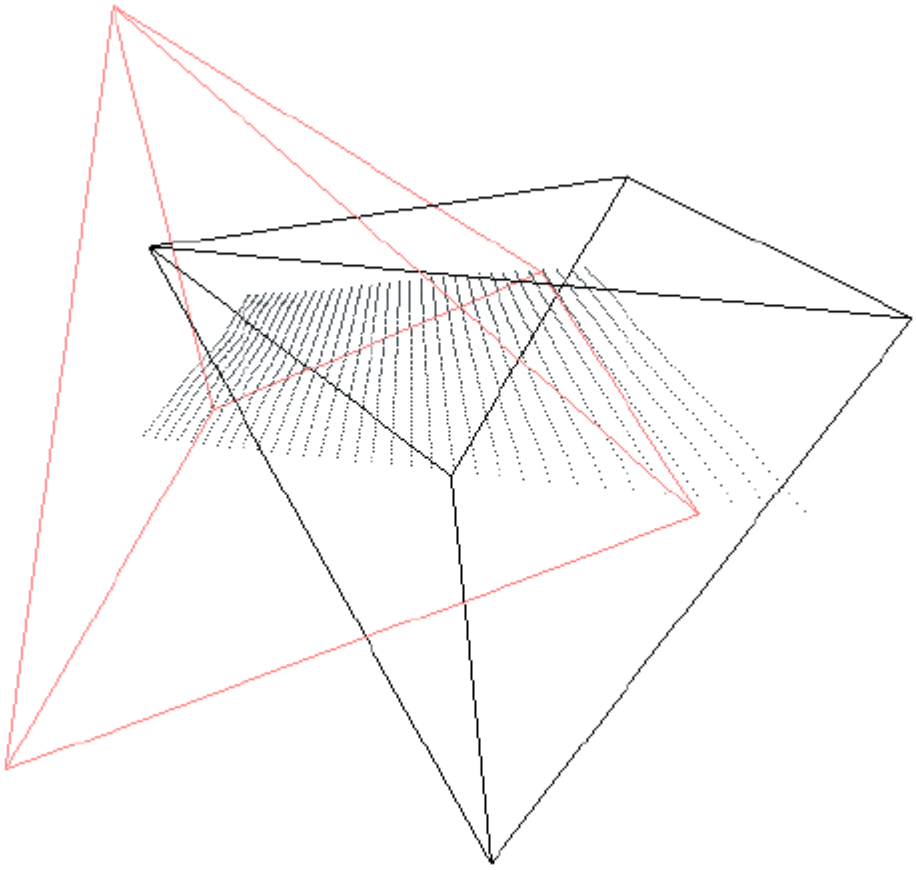

Рисунок 17. Поворот камеры, чтобы была возможность спроецировать сетку на плоскость.

С другой стороны, можно заменить проецирование определением пересечения пирамиды видимости камеры и плоскости воды. Уравнения для плоскостей пирамиды видимости можно вычислить из модельно видовой матрицы и матрицы проекции. Потом можно найти граничные векторы пирамиды и их точки пересечения с плоскостью.

Полученные точки можно передать в шейдер, выполняющий визуализацию водной поверхности. Вычислять интерполированную позицию точки в плоскости воды можно используя, например, барицентрические координаты.

Стоит заметить, что в некоторых случаях точки пересечения могут образовывать четырехугольник, а в некоторых пятиугольник. Нужно либо это учитывать при интерполяции вершин водной поверхности, либо приводить все к одной фигуре.

5.3 Распределение детализации

Несмотря на то, что при использовании перспективной проекции, вблизи камеры плотность точек водной поверхности будет значительно больше чем вдали, на практике, при небольшом размере сетки, этого недостаточно. Поэтому, чтобы не перегружать GPU огромным количеством полигонов, лучше распределять плотность точек неравномерно, а располагать большую часть точек у той стороны ограничивающего сетку прямоугольника, которая будет ближе к камере.

В предложенной реализации была использована следующая формула для распределения точек в проекционной сетке:

$$x = i, y = \alpha \cdot (e^{\beta \cdot j} - 1)$$
$$i \in [0,1], j \in [0,1]$$

Лучше подгонять параметры α,β и размеры сетки в зависимости от разрешения и разностью между ближней и дальней плоскостью отсечения. В целом, значения α = 5.0,β = 1.0 / (e^{α} - 1.0) при размерах сетки от 128x768 дают приемлемый результат.

5.4 Коррекция по краям

Ещё одной проблемой проекционной сетки является наличие артефактов по краям, возникающих при сдвиге вершин спроецированной сетки.

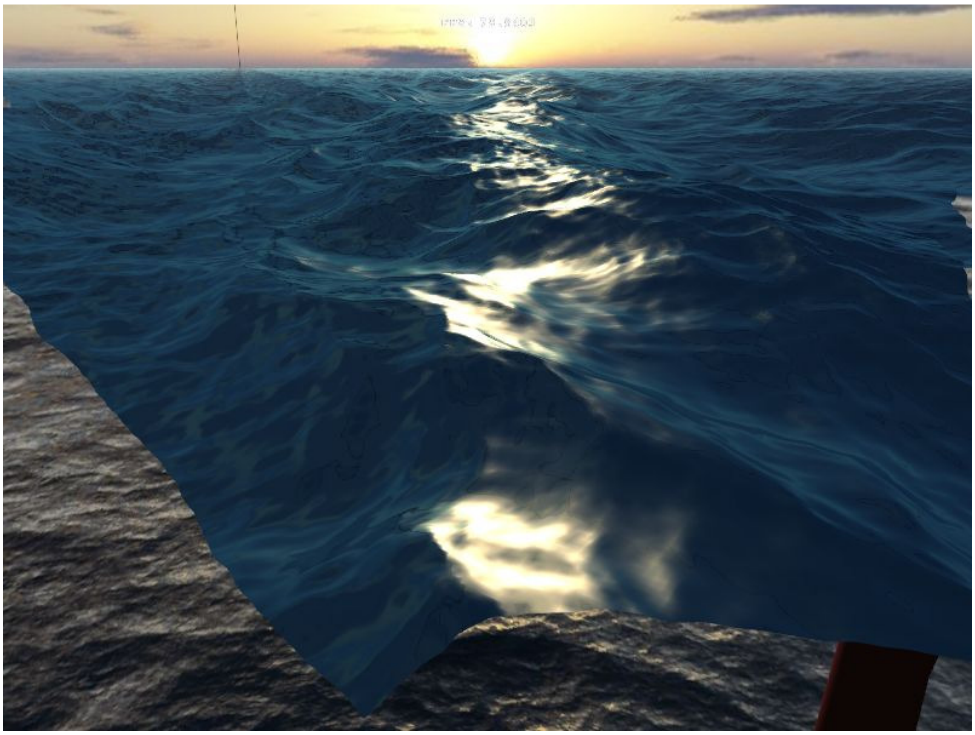

Рисунок 18. Артефакты, возникающие из-за сдвига вершин сетки поверхности воды.

Очевидным решением является расширение проекционной сетки, чтобы сдвинутые краевые вершины оставались за пределами экрана. Но сетку в таком случае придется либо очень сильно расширять, либо реализовывать достаточно сложную схему выбора величины сдвига в зависимости от положения наблюдателя относительно поверхности воды, амплитуды волн, расстояния до края сетки.

Разработчики CryEngine 2 предлагают достаточно простой подход для решения этой проблемы: уменьшать амплитуду волн по краям. Это достаточно простой и эффективный способ, но лучше вдобавок к нему ещё немного растягивать спроецированную сетку, чтобы уменьшение амплитуды волн по краям было менее заметным.

5.5 Фильтрация

Вычисление ДПФ можно производить только с использованием вещественной текстуры, использование других форматов не дает достаточной точности. Тут возникает проблема: видеокарты уровня Direct3D 9, если и поддерживают фильтрацию вещественных текстур, то только fp16. С другой стороны, эти же видеокарты в вершинном шейдере могут выбирать только из текстур 32 битной точности (и то, как правило, не всех).

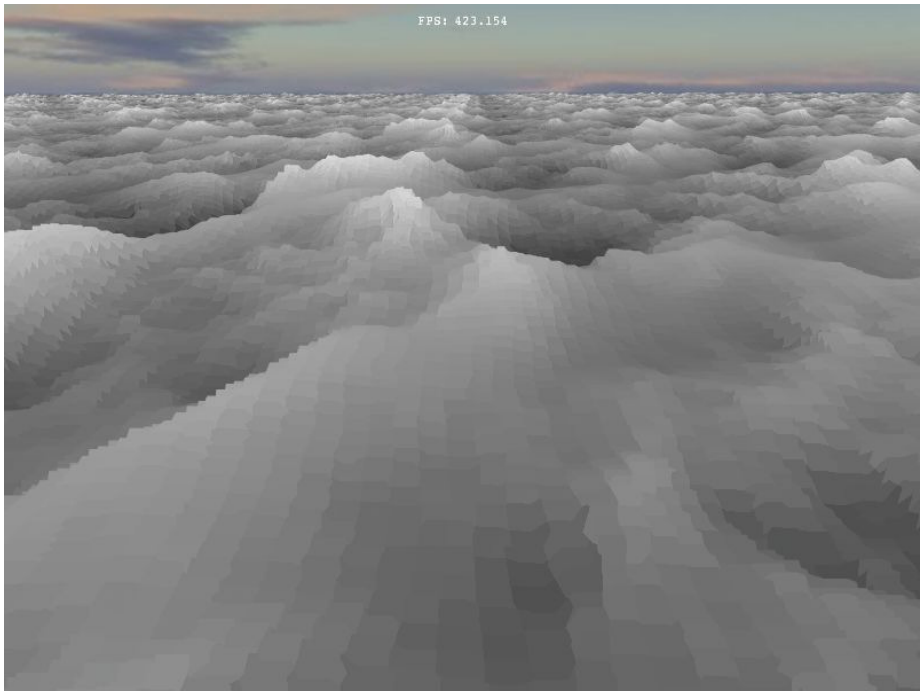

Рисунок 19. Поверхность воды без фильтрации.

Замечание: На рисунке приведен пример с слишком “растянутой” картой высот. В реальности такое соотношение разрешения текстуры и размера фрагмента поверхности использовать не стоит.

Некоторые решения данной проблемы:

- Уменьшение соотношения: разрешение текстуры / размера фрагмента водной поверхности. Д. Тессендорф в своей работе говорит, что хорошими являются соотношения 5-10см. реальной поверхности на тексель в карте высот.
- Выполнение фильтрации в шейдере. Стоит учитывать, что выборка из текстуры в вершинном шейдере - очень дорогая операция.

Стоит заметить, что после того как преобразование Фурье произведено, карту высот и нормалей можно перерисовать в обычные текстуры и успользовать уже их для визуализации поверхности. Это снизит суммарное колличество выбираемой памяти. Видеокарты уровная Direct3D 10 могу использовать любые форматы текстур в вершинном шейдере. Выборки же из карты нормалей можно делать в фрагментном шейдере.

5.6 Отражения и преломления

Деформированная поверхность воды усложняет использование описанных алгоритмов для генерации отражений и преломлений, потому что возникает проблема определения фрагментов попавших под воду(на воду), которые не нужно рисовать в карту отражений (соответсвенно преломлений), из-за этого могут быть видны артефакты стыковки отражений(преломлений) и видимых объектов в точке их соприкосновения с водой.

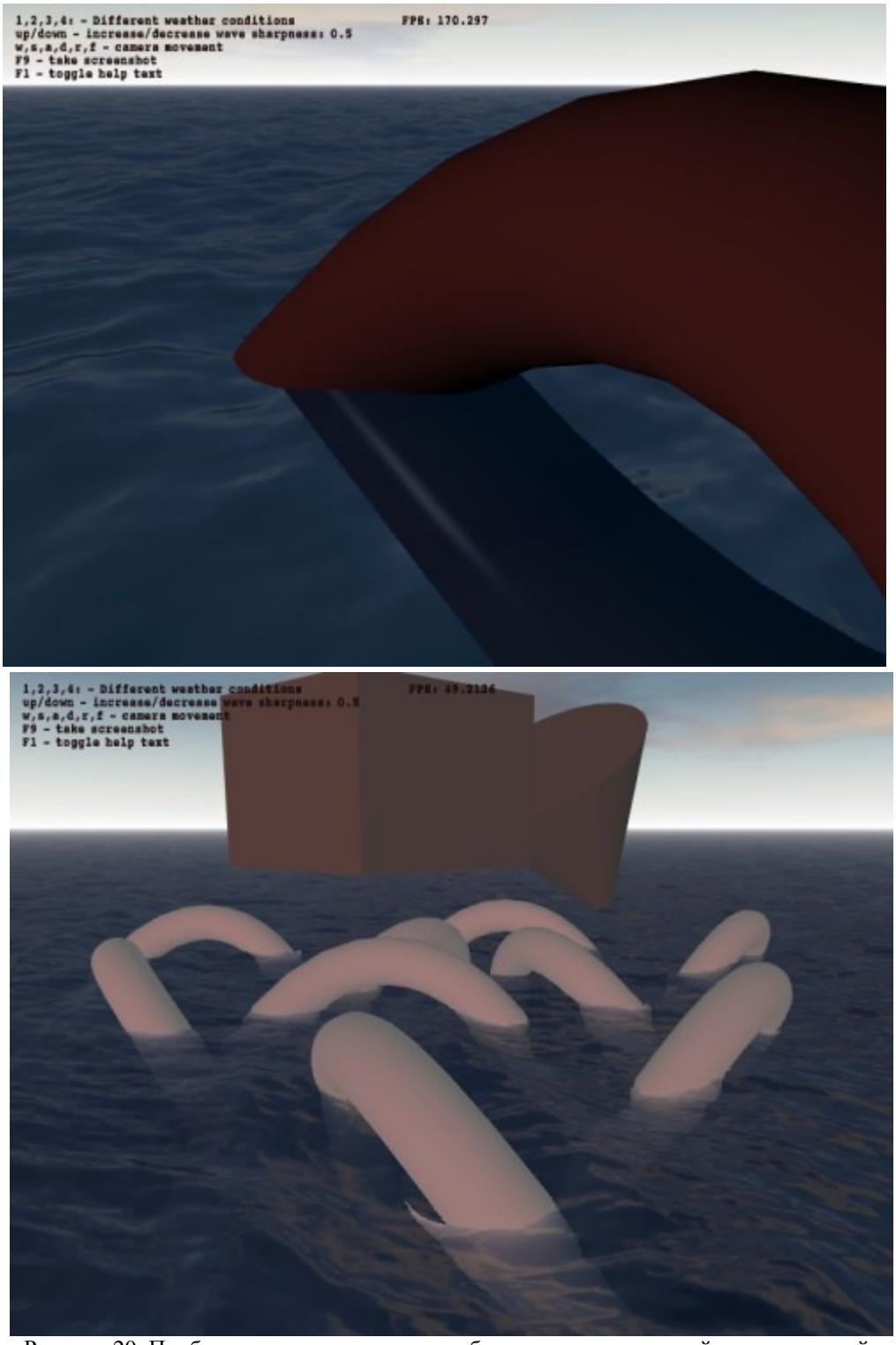

Рисунок 20. Проблемы стыковки видимых объектов и их отражений и преломлений.

Можно модифицировать шейдеры, используемые при рендеринге в текстуру отражений, чтобы они отсекали только то, что действительно под водой, но при деформации текстурной координаты нормалью, все равно можно получить "нереальное" отражение.

Без деформации нормалью тоже плохо, хотя и смещение точек водной поверхности дает изломанное отражение, выглядит оно не очень реалистично. Получается, что океан больше похож на покрывало колыхающееся на ветру.

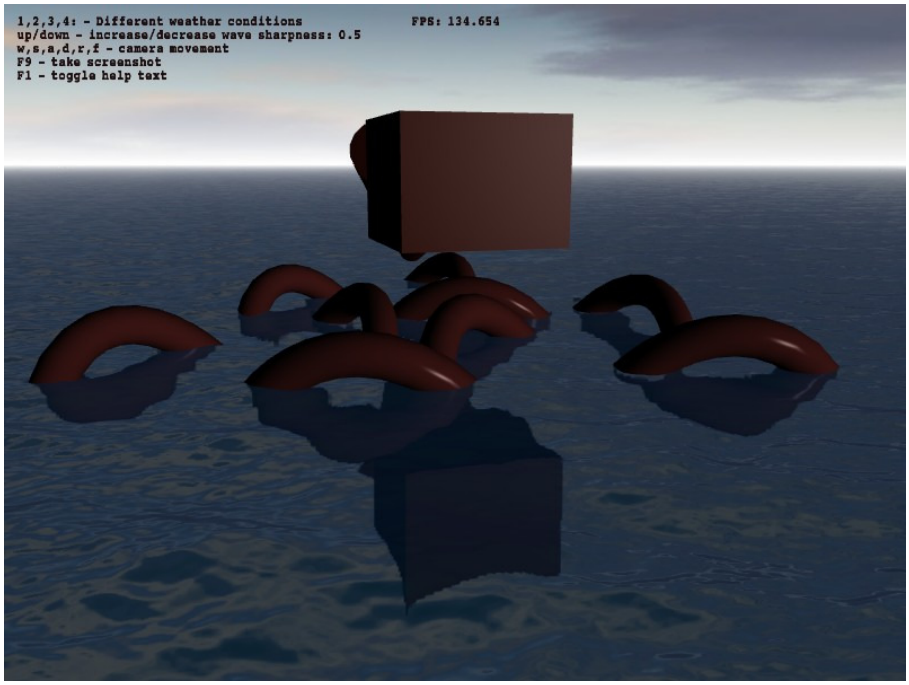

Рисунок 21. Отражения без деформации нормалью. Отсечение подводных объектов производятся плоскостью.

Во многих приложениях, проблемы с отражениями и преломлениями решаются уменьшением амплитуды волн и соответсвенно уменьшением силы деформации, при таких условиях проблема ограничивается, просто потому что баг труднозаметен.

Рисунок 22. Скриншоты из демо Unigine и игры Crysis.

Можно пойти и другим путем(если вам все-таки требуются большие волны и сильно искаженные отражения и преломления) - наоборот, деформировать и размыть всё сильнее, чтобы опять же, баг было труднее заметить. Конечно, оба решения косметические, но в конце концов, они дают приемлемый результат. Для пущего размытия можно вместо обычной выборки из текстуры отражений делать выборку из какого-нибудь её mipmap уровня(сделать это можно командой GLSL texture2DLod). Разумеется перед отрисовкой воды нужно сгенерировать mipmap уровни(в OpenGL функция glGenerateMipmapEXT):

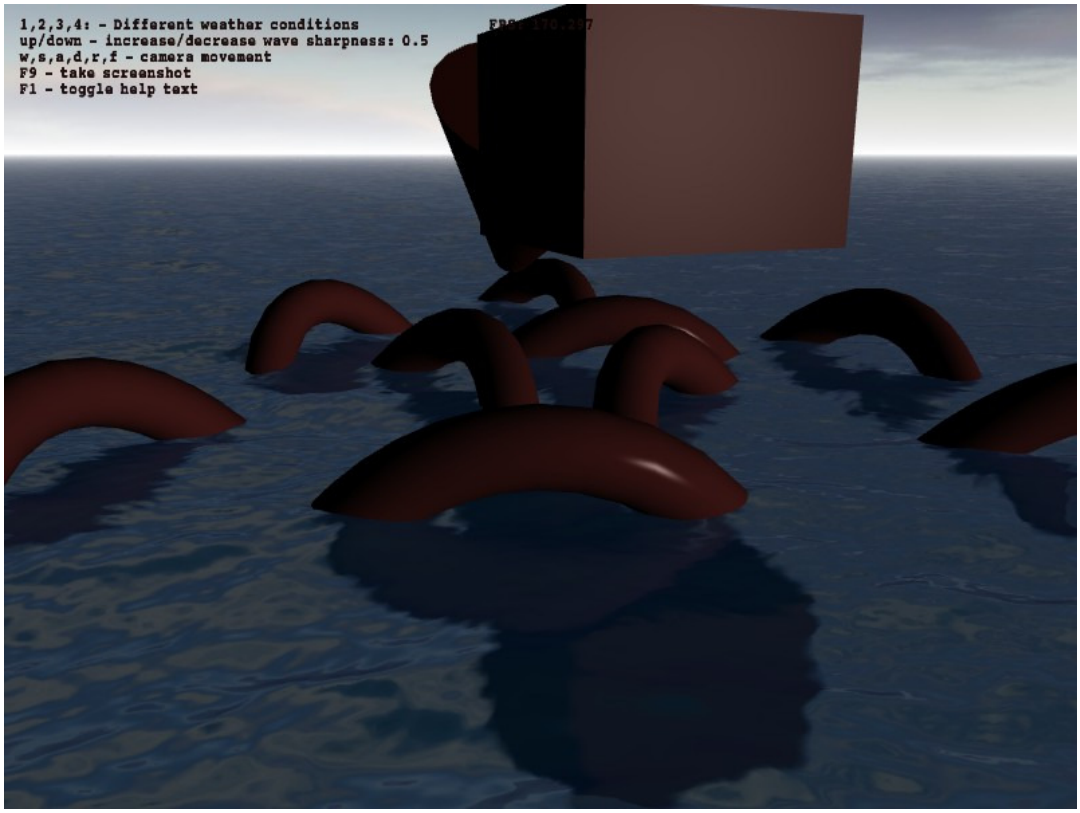

Рисунок 23. Сильное искажение и размытие отражений, чтобы спрятать артефакты у места соприкосновения объектов с водой.

Корректное отображение преломлений несколько более трудная задача. Во-первых, потому что вблизи объектов они видны лучше, по большому счету, вблизи берега отражений вообще почти не видно - просвечивает дно, и проблемы стыковки достаточно трудно увидеть. Во-вторых, преломления более четкие, поэтому решение проблемы размытием выглядит неестественно. Также, желательно учитывать толщу воды, ослабляющую видимость, поэтому будет использоваться карта глубин.

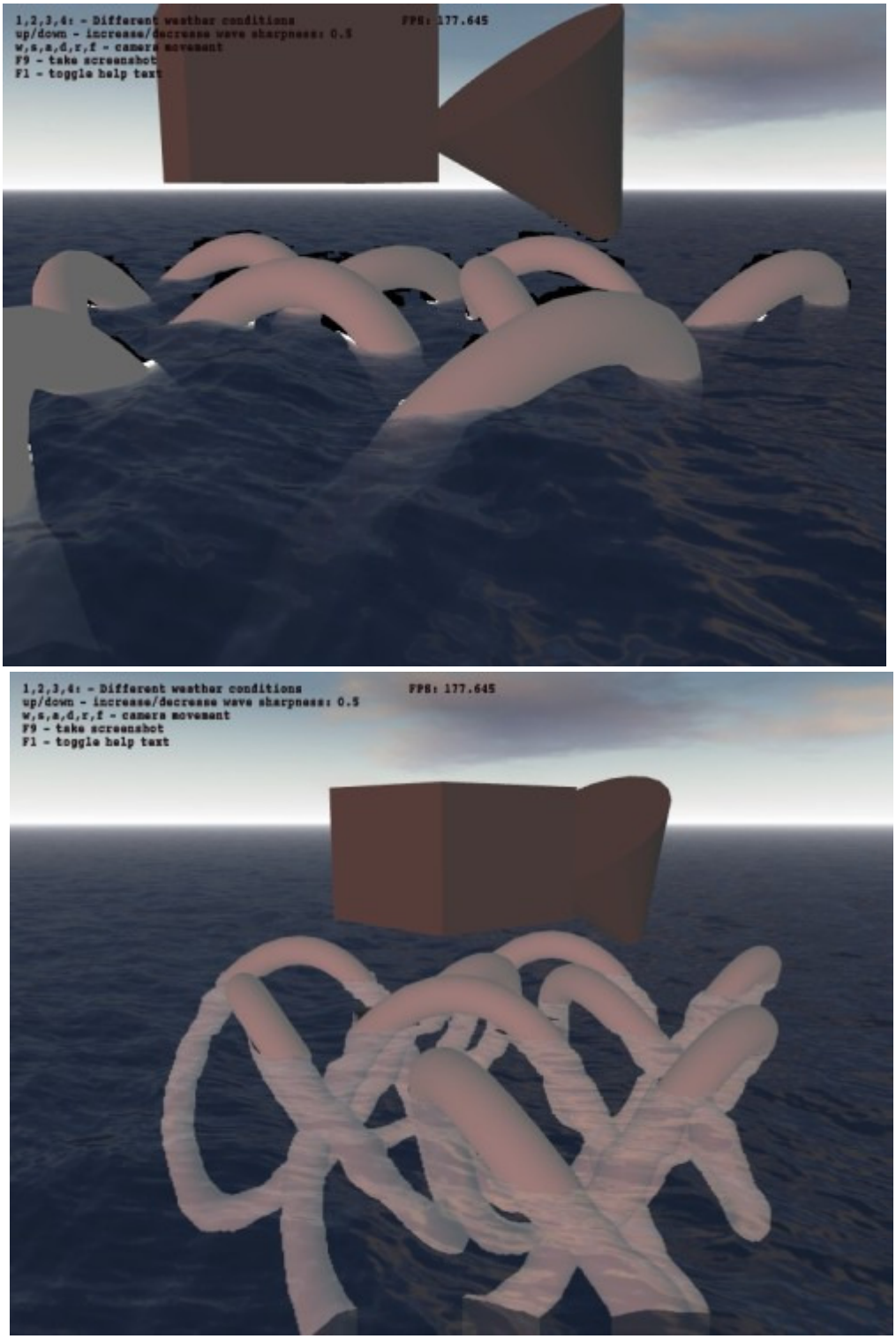

Рисунок 24. Преломления искажаются неправильно, потому что преломляются объекты находящиеся над водой. При сильных искажениях также могут возникнуть проблемы по краям.

В случае с отражениями, использовалась плоскость отсечения для отбраковки объектов, которые недолжны отражаться. Для преломлений можно предложить более точное решение. Нужно сравнивать расстоянием до объекта с расстоянием до поверхности воды. Если первое больше, то объект находится под водой и преломления учитывать нужно, в противном случае искажать преломления не надо. После такой коррекции ещё остаются проблемы как на рисунке 19 или 23 справа. Проблемы стыковки, как уже говорилось, обычно не исправляют, но ограничивают. А коррекцию по краям можно сделать уменьшая

силу искажения в зависимости от расстояния до рамки окна, примерно так как это делалось в случае с амплитудой волн при использовании проекционной сетки.

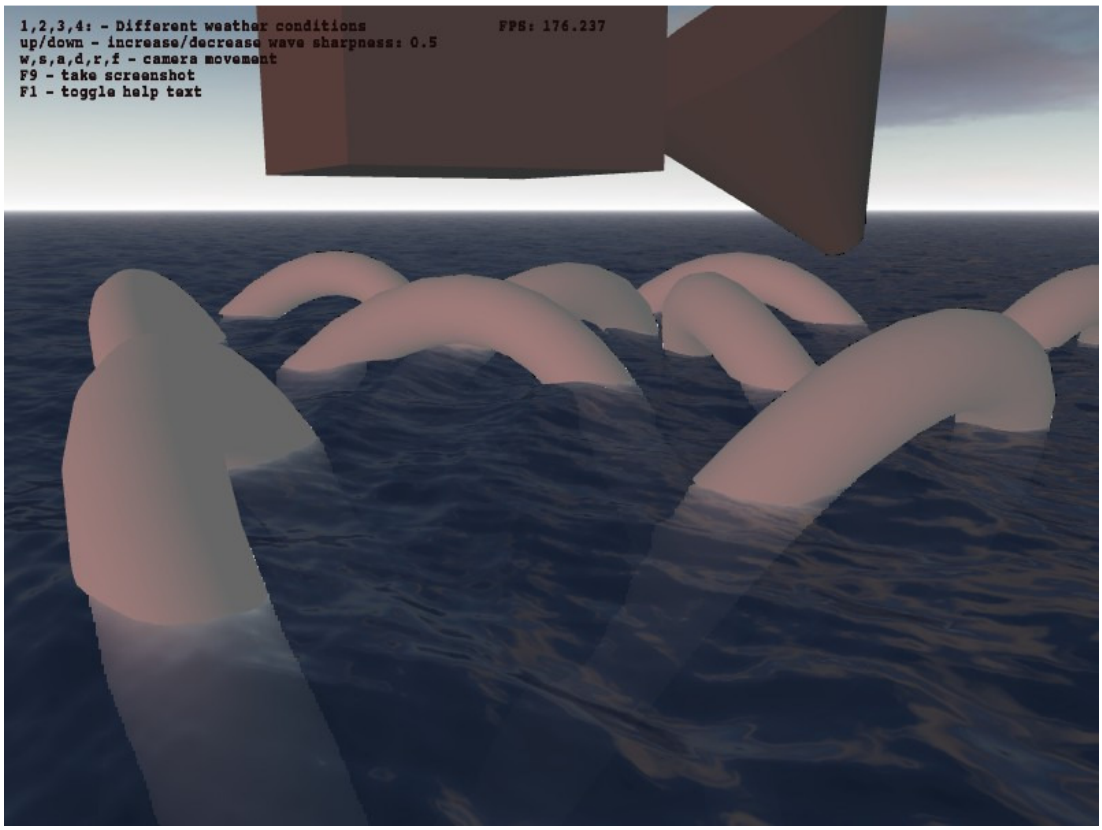

Рисунок 25. Корректировка преломлений.

5.7 Шейдер для океана

Далее рассмотрен шейдер для визуализации океана, работающий при следующих условиях, по крайней мере иногда:

- Поверхность воды лежит в плоскость XOZ, поэтому рассчет освещения производится в мировом пространстве, чтобы не заморачиваться с касательным.
- Используется проекционная сетка, при этом пересечение с пирамидой видимости приводится к четырёхугольнику и в шейдер передаются только его граничные точки.
- Вода визуализируется сеткой с точками в квадрате [0,1]x[0,1]. Они соответсвенно растягиваются, чтобы заполнить весь четырёхугльник, это уже производится в шейдере.
- Текстура с нормалями вычисляется отдельно, используя преобразование Фурье или же просто разницу амплитуд в соседних точках.
- Опционально можно включить отражения и преломления. При этом текстура с отражениями должна иметь mipmap уровни. Для учета глубины следует также передать в шейдер карту глубин, используемую при рендеринге в текстуру отражений.

```
/=============== Vertex Shader ===============/
// Карта полученная с помощью двух FFT: для вычисления аплитды волн, для
придания волнам острого профиля.
// Первые две компоненты – (высота, фаза), вторые две – смещение для придания
остроугольности.
```

```
uniform sampler2D heightMap;

uniform float sharpness;         // величина, характеризующая остроту волн
uniform vec2  surfaceSize;       // размер просчитанного сегмента поверхности
uniform vec3  surfaceCorners[4]; // четырехугльник, полученный при
пересечении с пирамидой видимости
uniform vec4  lightPosition;     // позиция источника света в мировых
координатах
uniform vec4  eyePosition;       // позиция наблюдателя в мировых координатах

uniform mat4  projectionMatrix;
uniform mat4  worldViewMatrix;
uniform mat4  reflectionMatrix;  // матрица для вычисления текстурных
координат в карте отражений

varying vec2  normalTexCoord;
varying vec2  reflectionTexCoord;
varying vec3  eyeDir;
varying vec3  lightDir;
varying vec3  fragmentPosition;

void main()
{
    // Вычисляем абсолютную позицию точки внутри четырехугольника
    vec3 a = (surfaceCorners[2] - surfaceCorners[3]) * gl_Vertex.y;
    vec3 b = (surfaceCorners[1] - surfaceCorners[0]) * gl_Vertex.y;
    vec3 c = mix(b, a, gl_Vertex.x);
    vec3 d = (surfaceCorners[3] - surfaceCorners[0]) * gl_Vertex.x;
    vec4 waterVertex = vec4(surfaceCorners[0] + c + d, 1.0);

    // Вычисляем текстурную координату для нормали и амплитуды
    normalTexCoord = waterVertex.xz / surfaceSize;

    // Сдвигаем вершину, используя карту высот и смещения
    vec4  texel       = texture2D(heightMap, normalTexCoord);
    float attenuation = min( pow(1.0 - gl_Vertex.y, 0.2), pow(1.0 -
abs(gl_Vertex.x - 0.5) * 2.0, 0.2) );
    waterVertex.y     = texel.x * attenuation;
    waterVertex.xz   -= sharpness * texel.zw * attenuation;

    // Вектор от наблюдателя(источника света) в точку океана
    eyeDir   = waterVertex.xyz - eyePosition.xyz;
    lightDir = waterVertex.xyz - lightPosition.xyz;

    // Вычисление текстурной координаты для отражений,
    // её деформацию производится в фрагментном шейдере, чтобы
    // не делать лишних выборок в вершинном.
    vec4 projTexCoord  = reflectionMatrix * waterVertex;
    reflectionTexCoord = projTexCoord.xy / projTexCoord.w;

    gl_Position      = projectionMatrix * worldViewMatrix * waterVertex;

    // Вычисляем нормализованную позиция фрагмента. В третьей компоненте
расстояние до точки.
    fragmentPosition = vec3(gl_Position.xy / gl_Position.w, gl_Position.w);
}

/=============== Fragment Shader =============/
uniform sampler2D   depthMap;       // Текстура глубины, используемая при
рендеринге преломлений
uniform sampler2D   normalMap;      // Предрассчитанная текстура с нормалями
для карты высот
uniform sampler2D   reflectMap;     // Текстура с отражениями
```

```
uniform sampler2D   refractMap;     // Текстура с преломлениями
uniform samplerCube environmentMap; // Текстура окражения(skybox)

uniform float distanceAttenuation;  // Величина характеризующая скорость
затуманивания в зависимоти от дальности
uniform float waterTransparency;    // Прозрачность воды
uniform vec4  fogColor;             // Цвет тумана
uniform vec4  lightSpecular;        // Увет источника света

// Нижний правый угол инвертированной песпективной матрицы.
// Матрица нужна для восстановления расстояния до точки по значению в
текстуре глдубины.
// Нужен только угол размера 2x2, потому что есть необходимость только в
последних
// двух компонентах для определения расстояния до точки.
uniform mat2  projectionMatrixInverse;

varying vec2  normalTexCoord;
varying vec2  reflectionTexCoord;
varying vec3  eyeDir;
varying vec3  lightDir;
varying vec3  fragmentPosition;

// Получить глубину точки по значению в карте глубин.
// fragmentPosition - позиция фрагмента в нормализованных экранных
координатах.
// texCoord - текстурная координата для выборки из текстуры глубин.
float get_fragment_depth(sampler2D depthMap, vec2 fragmentPosition, vec2
texCoord)
{
    float depth = 2.0 * texture2D(depthMap, texCoord).r - 1.0;
    vec2  vec   = projectionMatrixInverse * vec2(depth, 1.0);
    return -vec.x / vec.y;
}

void main()
{
    const vec3 waterMinColor = vec3(0.0, 0.05, 0.15);
    const vec3 waterMaxColor = vec3(0.0, 0.1,  0.15);

    // Нормализованное направление на точку и нормаль в мировых координатах.
    vec3 eyeDirNorm = normalize(eyeDir);
    vec3 normalNorm = 2.0 * ( texture2D(normalMap, normalTexCoord).rgb -
vec3(0.5) );

    // Нормаль будем выпрямлять вдали, чтобы поверхность выглядела менее
периодичной.
    float distVal = exp(-fragmentPosition.z * distanceAttenuation);
    normalNorm    = mix( vec3(0.0, 1.0, 0.0), normalNorm, pow(distVal, 10.0)
);

#ifdef ENABLE_LIGHTING
    // Рассчет освещения по модели Блинна-Фонга
    vec3 lightDirNorm = normalize(lightDir);
    vec3 halfVecNorm  = normalize(eyeDirNorm + lightDirNorm);
    vec4 light        = lightSpecular * pow( abs( dot(halfVecNorm,
normalNorm) ), 16.0);

    // Ослабляем освещенность вдали, так выглядит естественней
    gl_FragColor = light * pow(distVal, 10.0);
#else
    // Меняем цвет воды в зависимоти от угла обзора.
    float dotValue   = dot(eyeDirNorm, normalNorm);
```

```
    vec3  waterColor = mix( waterMinColor, waterMaxColor, abs(dotValue) );

    // Вычисляем текстурную координату, соответсвующую позиции фрагмента
    vec2  fragTexCoord = 0.5 * fragmentPosition.xy + vec2(0.5);

    #ifdef ENABLE_REFRACTIONS
    {
        // Деформируем текстурную координату для преломлений
        vec2  distortTexCoord = fragTexCoord + normalNorm.xz * 0.02;

        #ifdef ENABLE_DEPTH_MAP
        float nonDistortDepth = get_fragment_depth(depthMap,
fragmentPosition.xy, fragTexCoord);
        float distortDepth    = get_fragment_depth(depthMap,
fragmentPosition.xy, distortTexCoord);

        // Защищаем от деформации объекты над поверхностью воды
        float deltaDepth;
        if (distortDepth > fragmentPosition.z) {
            deltaDepth = distortDepth - fragmentPosition.z;
        }
        else
        {
            distortTexCoord = fragTexCoord;
            deltaDepth      = nonDistortDepth - fragmentPosition.z;
        }

        // Вычисляем ослабление в зависимости от глубины. Можно использовать
и экспоненту.
        float depthAttenuation = 1.0 / pow(1.0 + deltaDepth, 1.0 /
waterTransparency - 1.0);

        // Смешиваем преломления с цветом воды
        vec4 refractColor = texture2D(refractMap, distortTexCoord).rgba;
        waterColor        = mix(waterColor, refractColor.rgb,
depthAttenuation * refractColor.a);
        #else
        vec4 refractColor = texture2D(refractMap, distortTexCoord).rgba;
        waterColor        = mix(waterColor, refractColor.rgb,
waterTransparency * refractColor.a);
        #endif
    }
    #endif

    // Вычисляем коэффициенты Френеля
    float fresnel      = clamp( pow(1.0 + dotValue, 4.0), 0.05, 0.3 );
    vec3  reflectColor = textureCube( environmentMap, reflect(eyeDirNorm,
normalNorm) ).rgb;

    #ifdef ENABLE_REFLECTIONS
    {
        // Добавляем локальные отражения
        vec2 distortTexCoord   = reflectionTexCoord + normalNorm.xz * 0.05;
        vec4 localReflectColor = texture2DLod(reflectMap, distortTexCoord,
2.0).rgba;
        reflectColor           = mix(reflectColor, localReflectColor.rgb,
localReflectColor.a);
    }
    #endif

    // Смешиваем все
    waterColor = mix(waterColor, reflectColor, fresnel);
    waterColor = mix(fogColor.rgb, waterColor, distVal);
```

```
    gl_FragColor = vec4(waterColor, 1.0);
#endif
}
```

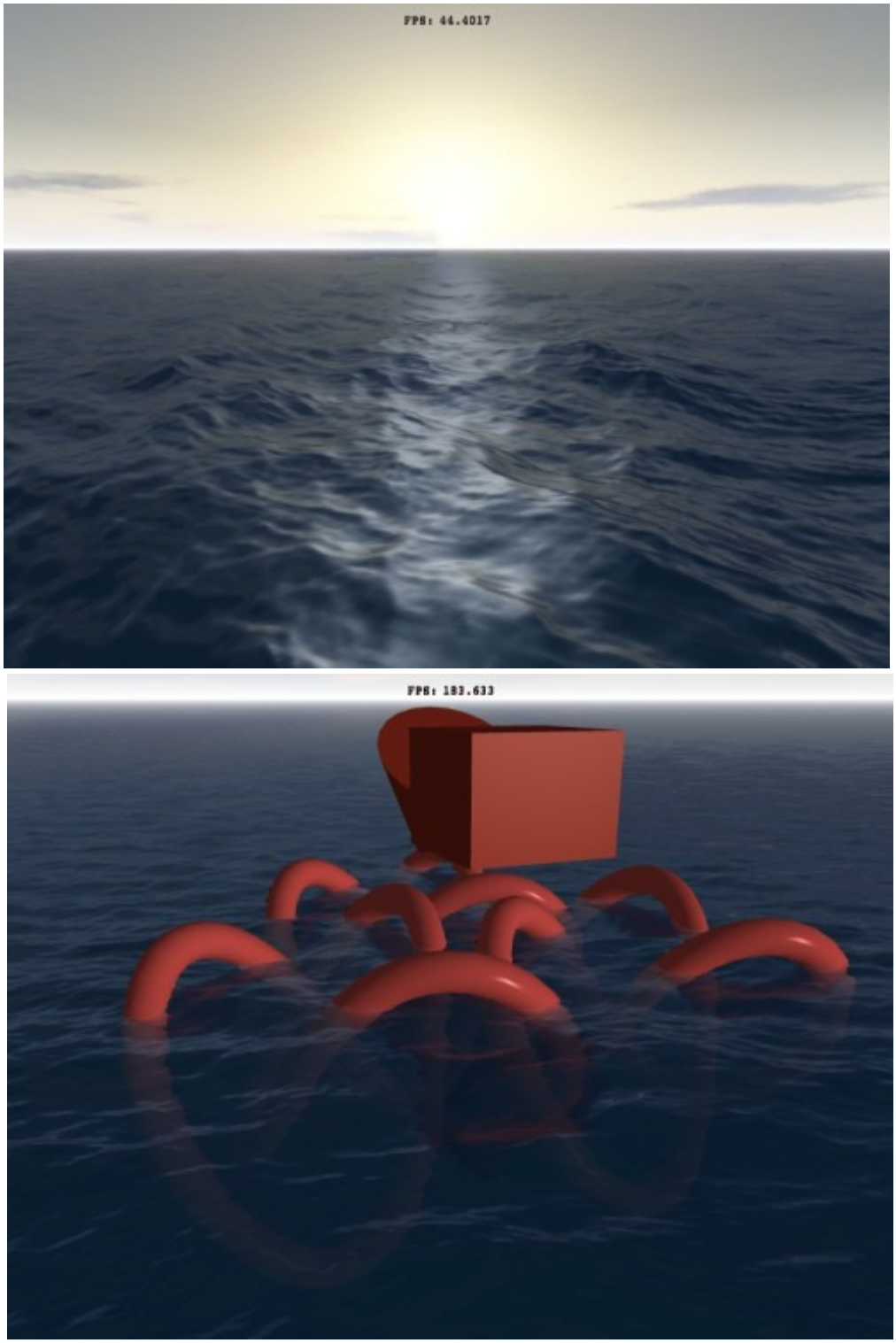

Рисунок 26. Океан, отрендеренный с помощью шейдера, рассмотренного выше. Размер преобразования Фурье 256x256.

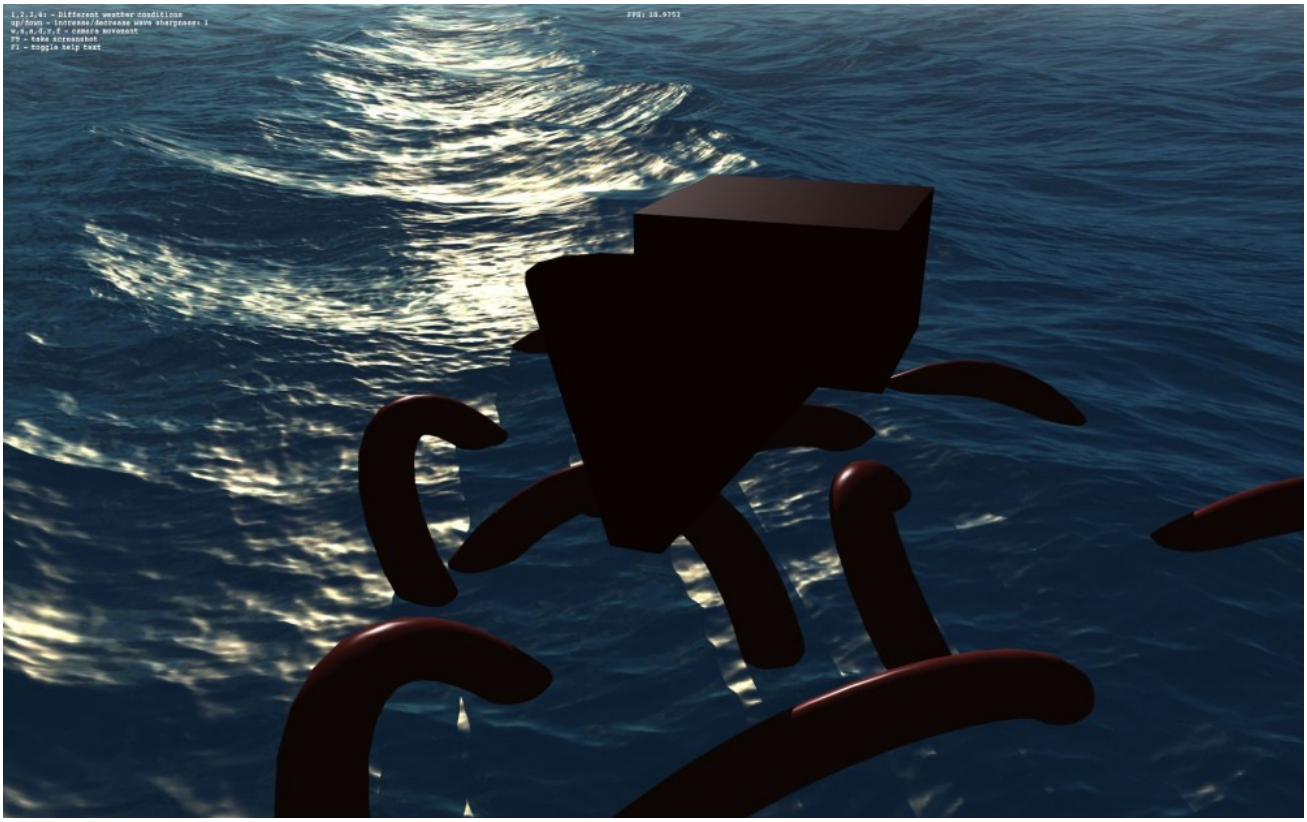

Рисунок 27. Океан отрендеренный без отражений и преломлений. Размер преобразования Фурье 1024x1024.

6. Заключение

В статье были рассмотрены лишь некоторые алгоритмы синтеза водной поверхности из их великого множества. У всех этих алгоритмов совершенно разные достоинства и недостатки, а соответсвенно и сферы применения.

Например, совершенно не имеет смысла реализовывать статистическую модель, хотя она и самая продвинутая, если в сцене присутсвуют только маленькие водоемы: озера, лужи, кружки с чаем. Поверхность, синтезируемая этой моделью характерна только для морей, океанов, супер-океанов, в которых волны возникают под действием сильного ветра. По большому счету, можно руководствоваться таким принципом: чем меньше водоем, тем меньше волны, тем проще модель и меньше проблем с её встраиванием в графический движок.

Разумеется, огромное влияние оказывает и целевая аудитория. Так например, если будет использоваться алгоритм, в котором необходимо производить выборки из текстуры или же рендерить в вершинный буфер, пользователи с видеокартами уровня ниже GeForce 6 серии вообще не смогут запустить приложение. На самом деле, учитывая производительность подобной операции, класс видеокарт, которые нормально справятся с нормальным приложением, использующем её, будет ещё меньше.

Последним фактором при выборе того или иного алгоритма оказывается сложность реализации (не смотря на то, что любой руководитель проекта поставил бы его первым пунктом). Реализовать статистическую модель синтеза поверхности океана и БПФ на GPU, определенно, сложно. Есть некоторые библиотеки, которые это делают, но вполне вероятно, что их использование окажется неприемлемым (например, приложение не

должно использовать CUDA или библиотеку невозможно нормально встроить в движок). При этом сложность модели только вершина айсберга. Как было видно из статьи, крупные волны усложняют практически все алгоритмы смежные с визуализацией водной поверхности: управление уровнем детализации, добавление отражений, физика взаимодействия с водой и т. д.

К сожалению, в статье не рассмотрены алгоритмы визуализации каустиков, пены, просвечивающих сквозь толщу воды лучей солнца, но всё это можно найти в работе Lasse Staff Jensen & Robert Golias "Deep-Water Animation and Rendering". Дополнительную информацию по статистической модели можно получить из статьи Jerry Tessendorf “Simulating Ocean Water”. В работе Tarjei Kvamme Loset “Real-Time Simulation and Visualization of Large Sea Surfaces” кроме вопросов реализации статистической модели рассматривается физическое моделирование водной поверхности как карты высот. И пожалуй, ещё стоит почитать про vertex texture fetch и R2VB, чтобы учеть нюансы.

7. Источники

Интернет источники:

- Быстрое преобразование Фурье: http://psi-logic.shadanakar.org/fft/fft.htm
- Модель Блинна-Фонга: http://en.wikipedia.org/wiki/Blinn-Phong_shading_model
- Рендеринг воды: http://steps3d.narod.ru/tutorials/water-tutorial.html
- Рендеринг отражений в текстуру: http://steps3d.narod.ru/tutorials/render-reflection-tutorial.html
- Вычисление коэффициентов Френеля: http://steps3d.narod.ru/tutorials/fresnel-tutorial.html

Статьи источники:

- **Jerry Tessendorf. “Simulating Ocean Water”.**
  SIGGRAPH course notes, 2004. paper
- **Lasse Staff Jensen & Robert Golias. "Deep-Water Animation and Rendering".**
  Gamasutra article, September 2001. article
- **Jason L. Mitchell. “Real-Time Synthesis and Rendering of Ocean Water”.**
  ATI Research Technical Report, April 2005. paper
- **Philipp Gerasimov, Simon Green, Randima (Randy) Fernando. “Shader Model 3.0. Using Vertex Textures”.**
  NVIDIA Corporation, June 2004. paper
- **Claes Johanson. “Real-time water rendering. Introducing the projected grid concept”.**
  Lund University, March 2004. paper
- **Martin Mittring. “Finding Next Gen – CryEngine 2”.**
  Crytek GmbH, SIGGRAPH course notes, 2007. paper
- **Tarjei Kvamme Loset. “Real-Time Simulation and Visualization of Large Sea Surfaces”.**
  Norwegian University of Science and Technology Department of Computer and Information Science, June 2007. paper
- **Kenneth Moreland & Edward Angel. “The FFT on a GPU”.**
  Sandia National Laboratories, Albuquerque, 2003. paper

- **Thilaka Sumanaweera & Donald Liu. "Medical Image Reconstruction with the FFT".**
  Siemens Medical Solutions USA. GPU Gems 2 chapter, April 2005. article
- **Mark Finch & Cyan Worlds. "Effective Water Simulation from Physical Models".**
  GPU Gems chapter, April 2005. article