Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Московский государственный технический университет имени Н.Э. Баумана

(национальный исследовательский университет)»

(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЕТ ПО ПРОИЗВОДСТВЕННОЙ ПРАКТИКЕ

Студент Топорков Павел

Группа ИУ7-43Б

Тип практики стационарная

Название предприятия МГТУ им. Н. Э. Баумана, каф. ИУ7

Студент: _____ Топорков П. А.

подпись, дата Фамилия, И.О.

Руководитель практики: _____ Куров А. В.

подпись, дата Фамилия, И.О.

Оценка _____

Москва — 2020 г.

Содержание

Введение **2**

1 Аналитическая часть **4**

1.1 Алгоритмы удаления невидимых линий и поверхностей . . . 4

1.2 Анализ моделей освещения 7

1.3 Алгоритмы генерации водной поверхности 9

1.4 Алгоритмы генерации морской волны 11

2 Конструкторская часть **16**

2.1 Подробный обзор волн Герстнера 16

2.2 Алгоритм получения высоты волны 16

2.3 Смещение волн по трём компонентам. 17

2.4 Амплитуда и направление волн. 18

2.5 Наложение волн. 18

3 Технологическая часть **20**

3.1 Выбор технических средствя 20

3.2 Листинг кода 20

4 Заключение **37**

Литература **38**

Введение

Симуляция прозрачной водной поверхности, которая способна преломлять и отражать свет, является одной из основных проблем при разработки игр или рендеринга видео. Одним из подходов построения изображения является реалистичное моделирование, нацеленное на максимальную схожесть с реальной морской поверхностью. Методы, позволяющие добиться данного эффекта, требуют огромное количество вычислительной мощности персональных компьютеров и используются в основном для кинопроизводства. Так же часто рассматривается вариант симуляции, при котором максимальная схожесть отходит на второй план, а в приоритете стоит производительность и вывод изображения в реальном времени. Данный способ применяется преимущественно в играх. В современное время специалисты стремятся добиться большей схожести изображения и анимации с реальным миром, при этом уменьшив сложность вычислений и время генерации.

В рамках практики была поставлена задача реализовать симуляцию большой, прозрачной водной поверхности, которая будет работать в реальном времени, на языке программирования c++. Решение поставленной задачи позволит людям внедрять модель океана в свои проекты, а также настраивать параметры симуляции по своему усмотрению.

Несмотря на продолжительное существование данной проблемы, проектов, предназначены решить данную задачу на языке программирования c++, ещё не было. В сети Интернет в открытом доступе выложены решения, подходящие только для игровых движков, но все они не предназначены для проектов создаваемых с нуля.

Для достижения поставленной цели в ходе работы требуется решить следующие задачи:

1. проанализировать существующие алгоритмы моделирования волн и

выбрать из них подходящие для выполнения проекта;

2. проанализировать существующие алгоритмы освещения, позволяющие реализовать прозрачность, и выбрать из них подходящие для выполнения проекта;

3. реализовать интерфейс программы.

1 Аналитическая часть

В этом разделе проводится анализ существующих алгоритмов построения трехмерного изображения, генерации текстур и освещения и выбираются наиболее подходящие алгоритмы для решения поставленных задач.

1.1 Алгоритмы удаления невидимых линий и поверхностей

Для того чтобы выбрать подходящий алгоритм построения изображения, необходимо провести обзор известных алгоритмов и осуществить выбор наиболее подходящих для решения поставленных задач.

Алгоритм Робертса

Алгоритм Робертса представляет собой первое известное решение задачи об удалении невидимых линий. Этот метод работает в объектном пространстве. Алгоритм прежде всего удаляет из каждого тела те ребра или грани, которые перекрываются самим телом. Затем все видимые рёбера каждого тела сравниваются с каждым из оставшихся тел для определения того, какая его часть или части, если таковые есть, перекрываются этими телами.

Преимущества данного алгоритма в том, что математические методы, используемые в нем просты, мощны и точны. Более поздние реализации алгоритма, например, использующие предварительную сортировку вдоль оси z, демонстрируют почти линейную зависимость от числа объектов. Минус этого алгоритма в том, что теоретически вычислительная трудоемкость алгоритма Робертса растет как квадрат числа объектов. Реализация оптимизированных алгоритмов весьма сложна.

Алгоритм Варнока

Алгоритм Варнока работает в пространстве изображений. В данном пространстве рассматривается окно и решается вопрос о том, пусто ли оно, или его содержимое достаточно просто для визуализации. Если это не так, то оно разбивается на фрагменты до тех пор, пока содержимое фрагмента не станет достаточно простым для визуализации или его размер не достигнет требуемого предела разрешения.

Для каждой подобласти (окна) определяются связанные с ней многоугольники и те из них, видимость которых определить "легко изображаются на экране. В противном же случае разбиение повторяется, и для каждой из вновь полученных подобластей рекурсивно применяется процедура принятия решения.

Предполагается, что с уменьшением размеров области ее перекрывает все меньшее и меньшее количество многоугольников. Считается, что в пределе будут получены области, содержащие не более одного многоугольника, и решение будет принято достаточно просто. Если же в процессе разбиения будут оставаться области, содержащие не один многоугольник, то следует продолжать процесс разбиения до тех пор, пока размер области не станет совпадать с одним пикселом. В этом случае для полученного пикселя необходимо вычислить глубину (значение координаты Z) каждого многоугольника и визуализировать тот из них, у которого максимальное значение этой координаты.

Алгоритм трассировки лучей

В этом методе для каждого пикселя картинной плоскости определяется ближайшая к нему грань, для чего через этот пиксель выпускается луч, находятся все его пересечения с гранями и среди них выбирается ближайшая.

К достоинствам данного алгоритма можно отнести возможность получения изображения гладких объектов без аппроксимации их примитивами (например, треугольниками). Вычислительная сложность метода линейно зависит от сложности сцены. Нетрудно реализовать наложение света и тени на объекты. Качество полученного изображения получается очень реалистичным, этот метод отлично подходит для создания фотореалистичных картин.

Серьёзным недостатком данного алгоритма является его производительность. Для получения изображения необходимо создавать большое количество лучей, проходящих через сцену и отражаемых от объекта. Это приводит к существенному снижению скорости работы программы.

Алгоритм, использующий z-буфер

Это один из простейших алгоритмов удаления невидимых поверхностей, который работает в пространстве изображения. Здесь обобщается идея о буфере кадра. Буфер кадра используется для заполнения атрибутов (интенсивности) каждого пикселя в пространстве изображения. Так же вводится Z-буфер, представляющий собой специальный буфер глубины, в котором запоминаются координаты Z (глубина) каждого видимого пикселя. В процессе работы глубина каждого нового пикселя, который надо занести в буфер кадра, сравнивается с глубиной того пикселя, который уже занесен в Z-буфер. Если это сравнение показывает, что новый пиксель расположен ближе к наблюдателю, чем пиксел, уже находящийся в буфере кадра, то уже новый пиксель заносится в буфер кадра. Помимо этого, производится корректировка Z-буфера: в него заносится глубина нового пикселя. Если же глубина нового пикселя меньше, чем хранящегося в буфере, то никаких действий производить не надо. В сущности, алгоритм для каждой точки (x,y) находит наибольшее значение функции Z(x,y).

Несмотря на свою простоту, этот алгоритм позволяет удалять слож-

ные поверхности и визуализировать пересечения таких поверхностей. Сцены могут быть произвольной сложности, а поскольку размеры изображения ограничены размером экрана дисплея, то трудоемкость алгоритма имеет линейную зависимость от числа рассматриваемых поверхностей. Элементы сцены заносятся в буфер кадра в произвольном порядке, поэтому в данном алгоритме не тратится время на сортировки, которые необходимы в других алгоритмах.

1.2 Анализ моделей освещения

Затенение по Гуро

Закраска по Гуро хорошо сочетается с диффузным отражением. Данный метод интерполяции обеспечивает лишь непрерывность значений интенсивности вдоль границ многоугольников, но не обеспечивает непрерывность изменения интенсивности, следовательно, возможно появление полос Маха.

Метод Гуро является одним из способов устранения дискретности интенсивностей закрашивания.

Данный алгоритм предполагает следующие шаги:

1. Вычисление векторов нормалей к каждой грани.
2. Вычисление векторов нормали к каждой вершине грани путем усреднения нормалей к граням (см. рисунок 1)
3. Вычисление интенсивности в вершинах грани.
4. Интерполяция интенсивности вдоль ребер грани.
5. Линейная интерполяция интенсивности вдоль сканирующей строки.

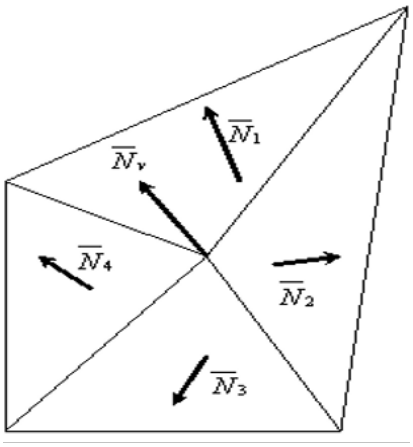

Рисунок 1 – вычисление векторов нормали к каждой вершине

Модель освещения Ламберта

Модель Ламберта - простейшая модель освещения, моделирующая идеально диффузное освещение. Считается, что свет, падающий в точку, одинаково рассеивается по всем направлениям полупространства. Таким образом, освещенность в точке определяется только плотностью света в точке поверхности, которая в свою очередь линейно зависит от косинуса угла падения.

Модель освещения Ламберта хорошо работает только для сравнительно гладких поверхностей. В отличии от нее модель Орен-Найара основана на предположении, что поверхность состоит из множества микрограней, освещение каждой из которых описывается моделью Ламберта(см. рисунок 2).

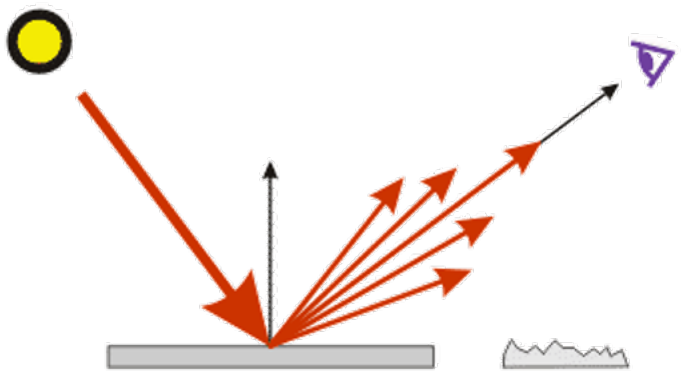

Рисунок 2 – падение и рассеивание света

Модель освещения Фонга

Модель Фонга – модель освещения, состоящая из диффузной компоненты (модель Ламберта) и зеркальной компоненты. Помимо равномерного освещения, модель реализует блики, появляющиеся на модели. Отраженная составляющая освещенности в точке зависит от того, насколько близки направления вектора, направленного на наблюдателя, и отраженного в сторону наблюдателя луча. В модели учитываются интенсивности фоновой и рассеянной компонент освещения, а также глянцевые блики.

Модель освещения Уиттеда

Модель освещения Уиттеда предназначена для того, чтобы рассчитать интенсивность отраженного к наблюдателю света в каждом пикселе изображения, которая в свою очередь может быть локальной или глобальной. В первом случае во внимание принимается только свет, падающий от источника, и ориентация поверхности. Во втором учитывается свет, отраженный от других объектов сцены или пропущенный сквозь них.

Уиттед пользуется моделью с такими же членами рассеянного и ламбертовского диффузного отражения, а также зеркального отражения Фонга, как и в локальной модели освещения. Модель Уиттеда учитывает эффекты преломления и отражения, зеркальности.

1.3 Алгоритмы генерации водной поверхности

Для водной анимации и рендеринга разработан ряд методов. Наиболее реалистичными являются те, которые основаны на гидродинамике и быстром преобразовании Фурье (например, Tessendorf 2001). Эти методы обеспечивают очень реалистичные результаты, но, к сожалению, они требуют значительного объема вычислений, что делает их непригодными для большой части интерактивных приложений.

С другой стороны, большинство проектов в настоящее время используют очень простые модели воды, большинство из которых используют карты нормалей или сеточный метод. К сожалению, эти подходы не могут обеспечить достаточного реализма и не точно воспроизводят волны на поверхности.

Сеточный метод

Сеточный метод предполагает дискретизацию потока жидкости с помощью введения двухмерной сетки и моделирование поверхности жидкости как движение её между узлами сетки. Зачастую данный способ называют разбиением на полигоны.

Сеточные методы более точны, чем методы взаимодействия частиц, т.к. вычисления легче проводить с регулярной сеткой, чем с неупорядоченным набором частиц. Более того, использование регулярной сетки позволяет упростить геометрию моделируемой водной поверхности, получить более качественное изображение гладкой поверхности воды, к тому же, к координатам каждой вершины сетки легко получить доступ через двухмерный массив вершин. Хоть данный метод и является достаточно удобным для динамической симуляции жидкости, но он не является сильно выигрышным по скорости, т.к. вычисления будут производиться несколько дольше, чем с использованием, например, метода Лагранжа. С помощью сеточных методов невозможно моделировать некоторые трёхмерные свойства жидкости, например, разбивающиеся волны океана.

Данный метод подходит для симуляции обычных волн в реальном времени.

Метод Лагранжа

Всю сложность движущихся потоков жидкости способна описать система уравнений Навье-Стокса(1). Это физический подход к моделирова-

нию водной поверхности. Он позволяет получать реалистичную модель поверхности жидкости, но требует большого количества вычислительных ресурсов.

Уавнений Навье-Стокса:

$$\begin{cases} \frac{\partial \vec{u}}{\partial t} + \vec{u} + \frac{1}{\rho}\nabla p = \vec{F} + v\nabla * \nabla \vec{u} \\ \nabla * u = 0 \end{cases} \tag{1}$$

Одним из решений этого уравнения и является метод Лагранжа. В нём жидкость рассматривается как набор частиц, подчиняющихся физическим законам. Каждая частица имеет набор параметров, таких, как масса, скорость и др. и может оказывать влияние на соседние частицы. Но у данного метода есть недостаток, для детализированной визуализации большого объема воды требуется огромное число частиц, что приводит к высоким требованиям к вычислительным ресурсам.

1.4 Алгоритмы генерации морской волны

Наложение карт высот

Данная модель водной поверхности основана на наложении нескольких карт высот, составленных как в пространстве, так и во времени. Каждая текстура представляет одну «гармонику» или «октаву» спектра, и текстуры складываются вместе. Эти текстуры называются картами высот, потому что каждое значение представляет высоту соответствующей точки над горизонтальной плоскостью.

С помощью карт высот художники могут легко управлять параметрами водной анимации вплоть до отдельных волн (см. рисунок 3). Комбинируя несколько карт высот с разными пространственными и временными масштабами, мы можем добиться сложной анимации. Этого достаточно для моделирования движущихся поверхностей океана на масштабах от 10 см до 40 км.

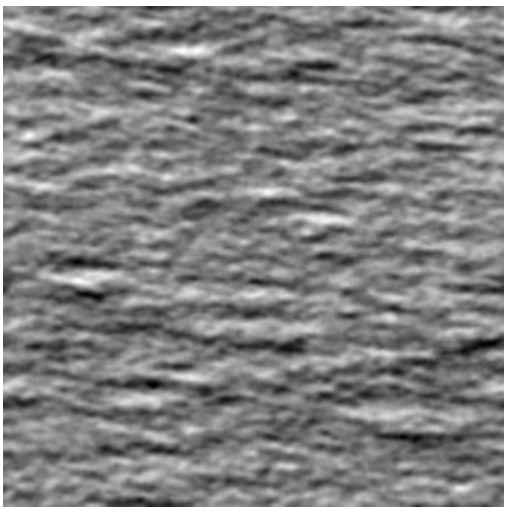

Рисунок 3 – пример карты высот

Линеаризованные волны на воде

Теория волн Эйри - это линейная модель, которая аппроксимирует движение поверхностных волн на водоеме суммой синусоидальных функций (2):

$$\eta(\vec{x}, t) = \eta_0 + \sum_{i=1}^{N} a_i sin(\omega_i \cdot (\phi_i(\vec{x}) + t)) \tag{2}$$

где η - высота воды, η_0 - произвольное постоянное смещение, a_i - амплитуда волны i, ω_i - угловая частота, t - текущее время, а ϕ_i - тщательно выбранная фазовая функция.

Каждая волна имеет угловую частоту ω, которая описывает, насколько быстро она колеблется во времени. Для поверхностных водных волн угловая частота определяется дисперсионным соотношением:

$$\omega = \sqrt{(gk + \frac{\sigma}{\rho}k^3)tanh(kh)} \tag{3}$$

где g - сила тяжести, σ - поверхностное натяжение, ρ - плотность воды, h - глубина воды в данном месте, а k - волновое число. Волновое число обратно связано с длиной волны λ соотношением $k = 2\pi/\lambda$. Каждая волна распространяется в пространстве со скоростью, определяемой фазовой скоростью c.

Волны Герстнера

Для эффективного моделирования нам необходимо контролировать крутизну наших волн. Синусоидальные волны имеют округлый вид, что может быть именно тем, что нам нужно для спокойного пасторального пруда. Но для бурного моря или океана нам нужно формировать более острые пики и более широкие впадины. Именно для этого и подходят волны Герстнера.

Выделим набор параметров для определения каждой волны(рис. 4). Длина волны (L): расстояние от гребня до гребня между волнами в мировом пространстве. Длина волны L относится к частоте w как $w = 2/L$. Амплитуда (A): высота от водной плоскости до гребня волны. Скорость (S): расстояние, на которое гребень перемещается вперед в секунду. Скорость удобно выразить как фазовую постоянную ϕ, где $\phi = Sx2/L$. Направление (D) : горизонтальный вектор, перпендикулярный фронту волны, по которой движется гребень.

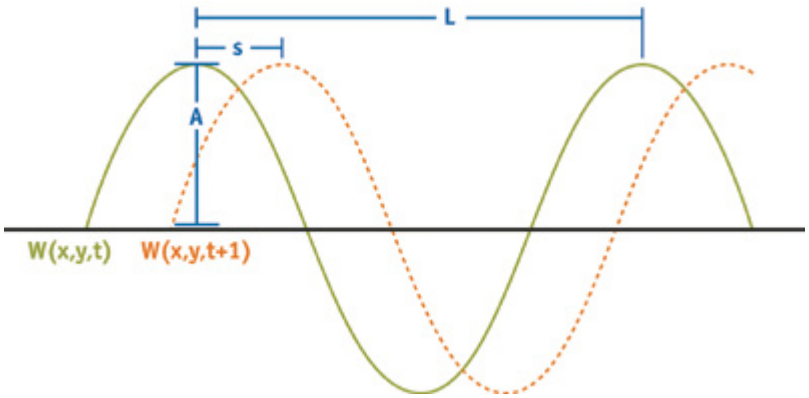

Рисунок 4 – пример гармонической волны и её параметров

Из данных параметров волны выделим функцию высоты волны:

$$H(x, y, t) = \sum (A_i \times sin(D_i \cdot (x, y) \times \omega_i + t \times \varphi_i)) \tag{4}$$

Теперь обратимся волновой функции Герстнера:

$$P(x,y,t) = \begin{pmatrix} x + \sum(Q_i A_i \times D_i x \times cos(\omega_i D_i \cdot (x,y) + \varphi_i t)), \\ y + \sum(Q_i A_i \times D_i y \times cos(\omega_i D_i \cdot (x,y) + \varphi_i t)), \\ \sum(A_i sin(\omega_i D_i \cdot (x,y) + \varphi_i t)) \end{pmatrix} \quad (5)$$

Q_i - параметр, контролирующий крутизну волн. Для одной волны i , Q_i от 0 дает обычную синусоидальную волну, и $Q_i = 1/(w_i A_i)$ дает острый гребень. Следует избегать больших значений Q_i, поскольку они вызовут образование петель над гребнями волн. Фактически, мы можем оставить спецификацию Q в качестве параметра «крутизны» для художника-постановщика, разрешив диапазон от 0 до 1 и используя $Q_i = Q/(w_i A_i \times numWaves)$, чтобы варьироваться от абсолютно гладких волн до самых резких, которые мы можем произвести.

Именно данная функция(5) способна моделировать волны с сильными гребнями(см рис. 5).

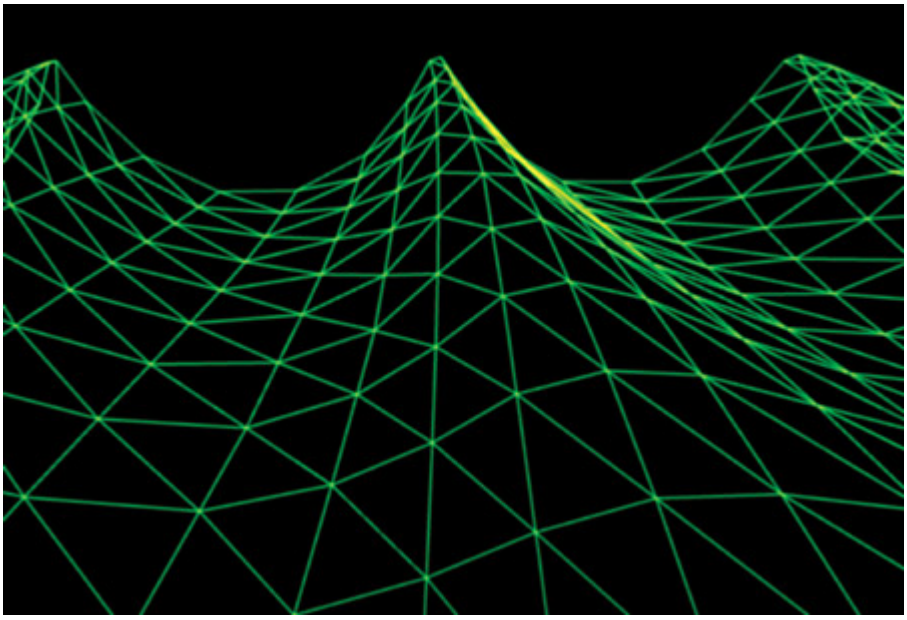

Рисунок 5 – пример волн, полученных алгоритмом Герстнера

Вывод

Проведенный анализ существующих методов генерации водной поверхности позволяет сделать вывод, что для решения поставленной задачи наилучшим решением будет либо шум метод Лагранжа, либо сеточный метод. Несмотря на то, что метод Лагранжа выигрывает у сеточного в реалистичности, но мой выбор пал на метод сеточный метод, т.к. он выигрывает в вычислительной сложности. В качестве метода генерации морской волны был выбран алгоритм Герстнера, он наиболее подходит для моделирования большой водной поверхности (океан, море).

2 Конструкторская часть

2.1 Подробный обзор волн Герстнера

В своей статье [1] компания Nvidia выделяет преимущество волн Герстнера, которое заключается в том, что данные волны, по сравнению с обычными синусоидальными, дают более острые пики и более широкие впадины. Это достигается благодаря смещению не только но z компоненту, но и по x, y. Так же для большей реалистичности используют наложение волн друг на друга.

2.2 Алгоритм получения высоты волны

Функция (4) позволяет получить нам высоту одной конкретной синусоидальной волны. Рассмотрим её подробнее:

- A — амплитуда
- D — Двумерный вектор направления волны
- x, y — координаты вершины
- ω – частота волны
- t — время
- φ — фазовый угол

Выделим два новых параметра:

- L – длина волны
- V – скорость волны

Выразим через них ω и φ:

$$\omega = \frac{2}{L} \tag{6}$$

$$\varphi = \frac{2V}{L} \tag{7}$$

Подставим формулы (6) и (7) в (4):

$$H(x, y, t) = \sum (A_i \times sin(D_i \cdot (x, y) \times \frac{2}{L_i} + t \times \frac{2V_i}{L_i})) \tag{8}$$

Остальные переменные, кроме времени и координат вершины, можно задавать, как произвольные константы.

2.3 Смещение волн по трём компонентам.

Как уже отмечалось ранее, волны Герстнера определяют смещения по компонентам x, y, z. Это помогает достичь реалистичного результата моделирования.

Рассмотрим волновую функцию Герстнера (5). Можно заметить, что компоненты получаемого вектора отвечают за смещения по каждой из координат. Так же компонента z (высота гребня волны) данного вектора уже известна - это формула (8).

Подставим (8) в (5):

$$P(x, y, t) = \begin{pmatrix} x + \sum (Q_i A_i \times D_i x \times cos(\omega_i D_i \cdot (x, y) + \varphi_i t)), \\ y + \sum (Q_i A_i \times D_i y \times cos(\omega_i D_i \cdot (x, y) + \varphi_i t)), \\ \sum (A_i \times sin(D_i \cdot (x, y) \times \frac{2}{L_i} + t \times \frac{2V_i}{L_i})) \end{pmatrix} \tag{9}$$

Получена конечная волновая функция Герстнера.

2.4 Амплитуда и направление волн.

Важным вопросом в моделировании волн являются определение амплитуды и направления.

Зависимость амплитуды волны от длины и текущих погодных условий, чаще всего задаются в сценарии во время разработки. Обычно вместе со средней длиной волны указывается медианная амплитуда, а для волны любого размера отношение её амплитуды к длине соответствует отношению средней амплитуды к средней длине волы.

Направление, по которому движется волна, полностью не зависит от других параметров. Следовательно выбор направления волны может происходить на основе любых критериев. Это всё также можно задать во время разработки.

2.5 Наложение волн.

При моделировании большого водного пространства(океан или море) нам необходимы большие, острые пики. Для достижения большей реалистичности можно также использовать наложение волн. Чаще всего совмещают 4-12 волн (9-12 на океан или море, 4-8 на озеро). Т.к. нам необходима симуляция большой водной поверхности, следовательно необходимо выбирать 9-12 совмещений волн.

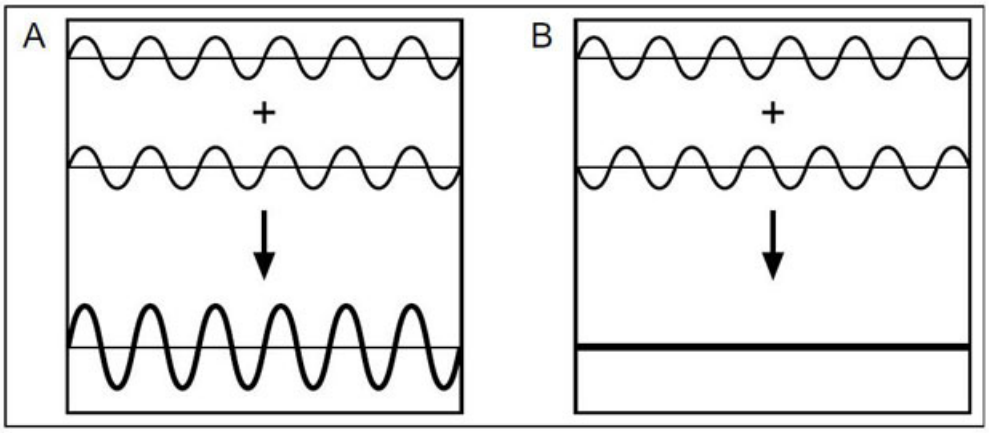

Рисунок 6 – Примеры наложения волн.

Вывод.

Изучив научный материал по теме, было принято решение выражать частоту и фазовый угол волны через её скорость и длину. Так же количество волн для наложении было выбрано от 9 до 12 для моделирования больших водных гребней. Что касается амплитуды, направления, длины и скорости, то их значения будут задаваться константами внутри ПО.

3 Технологическая часть

3.1 Выбор технических средствя

В качестве языка разработки выбран язык программирования C++ [10]. Данный выбор обусловлен тем, что он поддерживает объектно-ориентированн парадигму программирования, что позволяет естественным образом декомпозировать задачу и легко модифицировать программу. Так же данный язык содержит встроенные библиотеки для работы с многопоточностью, что позволяет использовать больше ресурсов компьютера. Редактор кода - VS Code [11].

3.2 Листинг кода

В ходе практики, проект не был выполнен окончательно, но был выполненный некоторый этюд - генератор простейших волн Герстнера (см. листинги).

Листинг 1 – Заголовочный файл классов Point2D и Poin3D

```
0  #ifndef POINT_H
1  #define POINT_H
2
3  #include <iostream>
4  #include <vector>
5  #include <array>
6
7  using namespace std;
8
9
10 /*
11 Class point in 2D
12 */
13 class Point2D {
14 private:
15     double x, y;
16 public:
17     Point2D(const double in_x, const double in_y);
```

```
18      void set_coords(const double x, const double y);
19      double get_x() const;
20      double get_y() const;
21      friend std::ostream& operator<< (std::ostream &out, const Point2D &
        point);
22      Point2D& operator= (const Point2D &point);
23  };
24
25  std::ostream& operator<< (std::ostream &out, const Point2D &point);
26
27
28  /*
29  Class point in 3D
30  */
31  class Point3D: public Point2D {
32  private:
33      double z;
34  public:
35      Point3D(const double in_x, const double in_y, const double in_z);
36      void set_coords(const double x, const double y, const double z);
37      double get_z() const;
38      friend std::ostream& operator<< (std::ostream &out, const Point3D &
        point3D);
39      Point3D& operator= (const Point3D &point);
40  };
41
42  std::ostream& operator<< (std::ostream &out, const Point3D &point3D);
43
44  #endif // POINT_H
```

Листинг 2 – Реализация классов Point2D и Poin3D

```
0   #include "../include/point.h"
1
2
3   Point2D::Point2D(const double in_x, const double in_y) : x(in_x), y(in_y)
        {}
4
5
6   void Point2D::set_coords(const double x, const double y) {
7       this->x = x;
```

```
8       this->y = y;
9   }
10
11
12  double Point2D::get_x() const {
13      return x;
14  }
15
16
17  double Point2D::get_y() const {
18      return y;
19  }
20
21
22  std::ostream& operator<< (std::ostream &out, const Point2D &point) {
23      out << "Point(" << point.x << ",␣" << point.y << ")";
24      return out;
25  }
26
27
28  Point2D& Point2D::operator= (const Point2D &point) {
29      x = point.x;
30      y = point.y;
31      return *this;
32  }
33
34
35  Point3D::Point3D(const double in_x, const double in_y, const double in_z)
        : Point2D(in_x, in_y), z(in_z) {}
36
37
38  void Point3D::set_coords(const double x, const double y, const double z) {
39      Point2D::set_coords(x, y);
40      this->z = z;
41  }
42
43
44  double Point3D::get_z() const {
45      return z;
46  }
```

```
47
48
49 std::ostream& operator<< (std::ostream &out, const Point3D &point3D) {
50     out << "Point(" << point3D.get_x() << ", " << point3D.get_y() << ", "
    << point3D.get_z() << ")";
51     return out;
52 }
53
54
55 Point3D& Point3D::operator= (const Point3D &point) {
56     set_coords(point.get_x(), point.get_y(), point.get_z());
57     return *this;
58 }
```

Листинг 3 – Заголовочный файл классов MathVector2D и MathVector3D

```
0  #ifndef MATH_VECTORS_OPERATION_H
1  #define MATH_VECTORS_OPERATION_H
2
3  #include "point.h"
4
5
6  /*
7  Math vector class in 2D
8  */
9  class MathVector2D {
10 private:
11     // x and y coords
12     double x, y;
13     Point2D start_point = Point2D(0, 0);
14     Point2D fin_point = Point2D(0, 0);
15 public:
16     MathVector2D(const double coord_x, const double coord_y);
17     MathVector2D(const Point2D start_point, const Point2D fin_point);
18     double get_x_coord() const;
19     double get_y_coord() const;
20     virtual double get_len() const;
21     friend std::ostream& operator<< (std::ostream &out, const MathVector2D
     &point);
22     MathVector2D& operator= (const MathVector2D &vect);
23     bool is_exist_start_point() const;
```

```
protected:
    bool _is_exist_start_point = false;
};

std::ostream& operator<< (std::ostream &out, const MathVector2D &vect);


/*
Math vector class in 3D
*/
class MathVector3D: public MathVector2D {
private:
    // z coords
    double z, start_z, fin_z;
public:
    MathVector3D(const double coord_x, const double coord_y, const double
    coord_z);
    MathVector3D(const Point3D start_point, const Point3D fin_point);
    double get_z_coord() const;
    virtual double get_len() const;
};

double get_scal_mult(const MathVector2D vect1, const MathVector2D vect2);

#endif // MATH_VECTORS_OPERATION_H
```

Листинг 4 – Реализация классов MathVector2D и MathVector3D

```
#include "../include/math_vectors_operation.h"
#include <math.h>


MathVector2D::MathVector2D(const double coord_x, const double coord_y): x(
    coord_x), y(coord_y), _is_exist_start_point(false) {}


MathVector2D::MathVector2D(const Point2D start_point, const Point2D
    fin_point) {
    this->start_point = start_point;
    this->fin_point = fin_point;
    _is_exist_start_point = true;
```

```
    x = fin_point.get_x() - start_point.get_x();
    y = fin_point.get_y() - start_point.get_y();
}


double MathVector2D::get_len() const {
    return sqrt(pow(x, 2) + pow(y, 2));
}


bool MathVector2D::is_exist_start_point() const {
    return _is_exist_start_point;
}


double MathVector2D::get_x_coord() const {
    return x;
}


double MathVector2D::get_y_coord() const {
    return y;
}


double get_scal_mult(const MathVector2D vect1, const MathVector2D vect2) {
    return (
        vect1.get_x_coord() * vect2.get_x_coord() +
        vect1.get_y_coord() * vect2.get_y_coord()
    );
}


std::ostream& operator<< (std::ostream &out, const MathVector2D &vect) {
    out << "MathVector2D(x: " << vect.x << ", y: " << vect.y;
    if (vect._is_exist_start_point) {
        out << ", 1st_point: " << vect.start_point << ", 2nd_point: " <<
   vect.fin_point;
    }
    out << ")";
```

```
50
51      return out;
52  }
53
54
55  MathVector2D& MathVector2D::operator= (const MathVector2D &vect) {
56      x = vect.x;
57      y = vect.y;
58      _is_exist_start_point = vect._is_exist_start_point;
59      start_point = vect.start_point;
60      fin_point = vect.fin_point;
61      return *this;
62  }
63
64
65  MathVector3D::MathVector3D(const double coord_x, const double coord_y,
        const double coord_z): MathVector2D(coord_x, coord_y), z(coord_z) {}
66
67
68  MathVector3D::MathVector3D(const Point3D start_point, const Point3D
        fin_point):
69              MathVector2D(Point2D(start_point.get_x(), start_point.get_y())
        , Point2D(fin_point.get_x(), fin_point.get_y())) {
70      start_z = start_point.get_z();
71      fin_z = fin_point.get_z();
72      z = fin_point.get_z() - start_point.get_z();
73  }
74
75
76  double MathVector3D::get_z_coord() const {
77      return z;
78  }
79
80
81  double MathVector3D::get_len() const {
82      return sqrt(pow(get_x_coord(), 2) + pow(get_y_coord(), 2) + pow(z, 2))
        ;
83  }
```

Листинг 5 – Заголовочный файл класса Surface

```
#ifndef SURFACE_H
#define SURFACE_H

#include <vector>

#include "point.h"
#include "math_vectors_operation.h"

// typedef for matrix of points
using MatrixPoints = vector<vector<Point3D>>;


/*
Class for Surface
*/
class Surface {
private:
    MatrixPoints matrix_points;
    Point2D _start_point;
    Point2D _fin_point;
    int len = 0;
public:
    Surface(const Point2D start_point, const Point2D fin_point, const int
    parts_ver);
    Point3D get_point(const size_t index_row, const size_t index_column)
    const;
    void set_point(const size_t index_row, const size_t index_column,
    Point3D point);
    int get_len() const;
    virtual ~Surface() = default;
protected:
    void init_matrix_point();
};

#endif // SURFACE_H
```

Листинг 6 – Реализация класса Surface

```
#include "../include/surface.h"
#include <math.h>
#include "../include/exception.h"
```

```
3
4
5  Surface::Surface(const Point2D start_point, const Point2D fin_point, const
       int parts_ver):
6                                  _start_point(start_point), _fin_point(
       fin_point), len(parts_ver)
7  {
8      init_matrix_point();
9  };
10
11
12 void Surface::init_matrix_point() {
13     double dx = fabs(_fin_point.get_x() - _start_point.get_x()) / len;
14     double dy = fabs(_fin_point.get_y() - _start_point.get_y()) / len;
15     double start_x = _start_point.get_x(), start_y = _start_point.get_y();
16
17     for (int i = 0; i < len; ++i) {
18         vector<Point3D> temp_vector;
19         double cur_x = start_x + dx * i;
20         for (int j = 0; j < len; ++j) {
21             double cur_y = start_y + dy * j;
22             temp_vector.push_back(Point3D(cur_x, cur_y, 0.0));
23         }
24         matrix_points.push_back(temp_vector);
25     }
26 }
27
28
29 Point3D Surface::get_point(const size_t index_row, const size_t
       index_column) const {
30     if (index_row < 0 || index_row > len - 1 || index_column < 0 ||
       index_column > len - 1) {
31         throw IndexException("Invalid␣index␣in␣Surface::get_point");
32     }
33     return matrix_points[index_row][index_column];
34 }
35
36
37 void Surface::set_point(const size_t index_row, const size_t index_column,
       Point3D point) {
```

```
38      if (index_row < 0 || index_row > len - 1 || index_column < 0 ||
        index_column > len - 1) {
39          throw IndexException("Invalid index in Surface::set_point");
40      }
41      matrix_points[index_row][index_column] = point;
42  }
43
44
45  int Surface::get_len() const {
46      return len;
47  }
```

HarmonicWave

Листинг 7 – Заголовочный файл классов HarmonicWave и GerstWave

```
0   #include "surface.h"
1
2
3   /*
4   Class for harmonic wave
5   */
6   class HarmonicWave: public Surface {
7   private:
8       // params
9       double amplitude = -1, wave_len = -1, speed = -1, omega = -1, phi =
        -1;
10      MathVector2D direction = MathVector2D(0, 0);
11  public:
12      HarmonicWave(const Point2D start_point, const Point2D fin_point, const
        int parts_ver);
13      void set_params(const double new_amplitude = 0, const double
        new_wave_len = 0, const double new_speed = 0);
14      void set_direction(const MathVector2D direction);
15      MathVector2D get_direction() const;
16      double get_amplitude() const;
17      double get_wave_len() const;
18      double get_speed() const;
19      double get_omega() const;
20      double get_phi() const;
21      // set value x, y, z for cur times in all point matrix
22      virtual void set_simulation(const double cur_time);
```

```
23      void setup_base_wave();
24  protected:
25      void check_params(const double amplitude, const double wave_len, const
        double speed) const;
26      double get_height(const Point2D point, const double cur_time) const;
27  };
28
29  /*
30  Class for Gerstner wave
31  */
32  class GerstWave: public HarmonicWave {
33  private:
34      double steepness = -1;
35      double get_offset_x(const Point3D cur_point, const double cur_time)
        const;
36      double get_offset_y(const Point3D cur_point, const double cur_time)
        const;
37  public:
38      GerstWave(const Point2D start_point, const Point2D fin_point, const
        int parts_ver);
39      void set_steepness(const double steepness);
40      // set value x, y, z for cur times in all point matrix
41      virtual void set_simulation(const double cur_time);
42  protected:
43      void check_steepness() const;
44  };
```

Листинг 8 – Реализация классов HarmonicWave и GerstWave

```
0  #include "../include/waves.h"
1  #include "../include/exception.h"
2  #include "../include/math_vectors_operation.h"
3  #include <math.h>
4
5  #ifdef _DEBUG_
6      void print_data_HarmonicWave(HarmonicWave test_wave) {
7          cout << "debug␣print:␣" << endl << endl;
8          for (size_t i = 0; i < test_wave.get_len(); ++i) {
9              for (size_t j = 0; j < test_wave.get_len(); ++j) {
10                 cout << test_wave.get_point(i, j) << "␣";
11             }
```

```
            cout << endl;
        }
    }
#endif

HarmonicWave::HarmonicWave(const Point2D start_point, const Point2D
    fin_point, const int parts_ver): Surface(start_point, fin_point,
    parts_ver) {}


void HarmonicWave::check_params(const double amplitude, const double
    wave_len, const double speed) const {
    if (amplitude < 0 || wave_len < 0 || speed < 0) {
        throw NegativeValueException("All params(amplitude, wave_len,
    speed) must not be less than 0");
    }
}


void HarmonicWave::set_params(const double new_amplitude, const double
    new_wave_len, const double new_speed) {
    check_params(new_amplitude, new_wave_len, new_speed);

    amplitude = new_amplitude;
    wave_len = new_wave_len;
    speed = new_speed;

    omega = 2 / wave_len;
    phi = speed * (2 / wave_len);
}


void HarmonicWave::set_direction(const MathVector2D direction) {
    this->direction = direction;
}


double HarmonicWave::get_height(const Point2D cur_point, const double
    cur_time) const {
    check_params(amplitude, wave_len, speed);
```

```
46
47      MathVector2D position(cur_point.get_x(), cur_point.get_y());
48      return amplitude * sin(get_scal_mult(direction, position)) * omega +
        cur_time * phi;
49  }
50
51
52  void HarmonicWave::set_simulation(const double cur_time) {
53      check_params(amplitude, wave_len, speed);
54
55      for (int i = 0; i < get_len(); ++i) {
56          for (int j = 0; j < get_len(); ++j) {
57              double cur_x = get_point(i, j).get_x();
58              double cur_y = get_point(i, j).get_y();
59              double cur_z = get_height(get_point(i, j), cur_time);
60              set_point(i, j, Point3D(cur_x, cur_y, cur_z));
61          }
62      }
63  }
64
65
66  void HarmonicWave::setup_base_wave() {
67      init_matrix_point();
68  }
69
70
71  double HarmonicWave::get_amplitude() const {
72      return amplitude;
73  }
74
75
76  double HarmonicWave::get_wave_len() const {
77      return wave_len;
78  }
79
80
81  double HarmonicWave::get_speed() const {
82      return speed;
83  }
84
```

```
85
86  double HarmonicWave::get_omega() const {
87      return omega;
88  }
89
90
91  double HarmonicWave::get_phi() const {
92      return phi;
93  }
94
95
96  MathVector2D HarmonicWave::get_direction() const {
97      return direction;
98  }
99
100
101 GerstWave::GerstWave(const Point2D start_point, const Point2D fin_point,
        const int parts_ver): HarmonicWave(start_point, fin_point, parts_ver)
        {}
102
103
104 void GerstWave::set_steepness(const double steepness) {
105     this->steepness = steepness;
106 }
107
108
109 void GerstWave::check_steepness() const {
110     if (steepness < 0 || steepness > 1 / (get_amplitude() * get_omega()))
        {
111         throw NegativeValueException("Steepness must be in [0; 1/(omega *
        amplitude)");
112     }
113 }
114
115
116 double GerstWave::get_offset_x(const Point3D cur_point, const double
        cur_time) const {
117     MathVector2D position(cur_point.get_x(), cur_point.get_y());
118     return steepness * get_amplitude() * get_direction().get_x_coord()
119             * cos(get_omega() * get_scal_mult(get_direction(), position) +
```

```
            get_phi() * cur_time);
}


double GerstWave::get_offset_y(const Point3D cur_point, const double
    cur_time) const {
    MathVector2D position(cur_point.get_x(), cur_point.get_y());
    return steepness * get_amplitude() * get_direction().get_y_coord()
            * cos(get_omega() * get_scal_mult(get_direction(), position) +
            get_phi() * cur_time);
}


void GerstWave::set_simulation(const double cur_time) {
    check_steepness();
    check_params(get_amplitude(), get_wave_len(), get_speed());

    for (int i = 0; i < get_len(); ++i) {
        for (int j = 0; j < get_len(); ++j) {
            double cur_x = get_offset_x(get_point(i, j), cur_time);
            double cur_y = get_offset_y(get_point(i, j), cur_time);
            double cur_z = get_height(get_point(i, j), cur_time);
            set_point(i, j, Point3D(cur_x, cur_y, cur_z));
        }
    }
}
```

Листинг 9 – Заголовочный файл классов IndexException и NegativeValueException

```
#include <iostream>
#include <string>

/*
Exclusion class when selecting a nonexistent array or matrix element
*/
class IndexException: public std::exception {
private:
    std::string m_error;
public:
    IndexException(std::string error);
```

```
11      const char* what() const noexcept;
12  };
13
14
15  /*
16  Exception class when dealing with negative elements
17  */
18  class NegativeValueException: public std::exception {
19  private:
20      std::string m_error;
21  public:
22      NegativeValueException(std::string error);
23      const char* what() const noexcept;
24  };
```

Листинг 10 – Реализация классов IndexException и NegativeValueException

```
0   #include "../include/exception.h"
1
2
3   IndexException::IndexException(std::string error): m_error(error) {}
4
5
6   const char* IndexException::what() const noexcept {
7       return m_error.c_str();
8   }
9
10
11  NegativeValueException::NegativeValueException(std::string error): m_error
        (error) {}
12
13
14  const char* NegativeValueException::what() const noexcept {
15      return m_error.c_str();
16  }
```

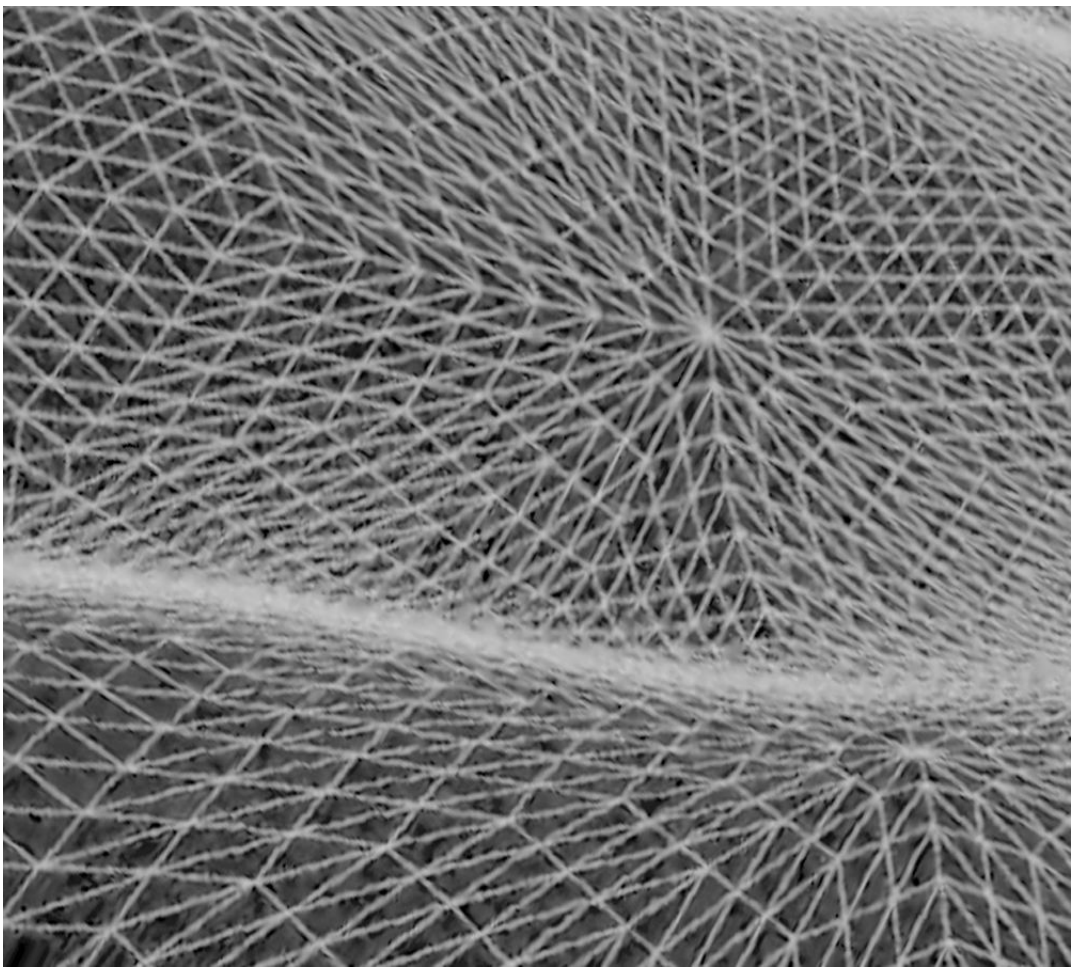

Рисунок 7 – Сгенерированные волны Герстнера (A=50; Speed=100; Q=0.3; len=200)

Вывод

В ходе практики был реализован алгоритм генирации волн Гестнера и получены сгенерированные изображения в реальном времени. В ходе курсового проекта еще предстоит решить поставленную задачу: реализовать прозрачность волн.

4 Заключение

Изучив предметную область, было принято решение, что для решения поставленной задачи из множества разнообразных методов генерации водной поверхности подойдёт сеточный метод с использованием волн Герстнера. В ходе практики программный продукт подготовлен не был. Была реализована подготовительная программа, в которой был продеманстрирован алгоритм моделирования волн Герстнера в реальном времени. Несмотря на то, что итоговая программа еще не реализована, было проведено исследование способов улучшения алгоритма и адаптации его для задачи проекта.

Список литературы

[1] В. И. Левенштейн. Двоичные коды с исправлением выпадений, вставок и замещений символов. Доклады Академий Наук СССР, 1965. 163.4:845-848.

[2] GPU Gems. Chapter 1. Effective Water Simulation from Physical Models.Режим доступа:

$https://developer.nvidia.com/gpugems/gpugems/part-i-natural-effects/chapter-1-effective-water-simulation-physical-models$

Дата обращения: 13.08.2020

[3] GPU Gems. Chapter 18. Using Vertex Texture Displacement for Realistic Water Rendering.Режим доступа:

$https://developer.nvidia.com/gpugems/gpugems2/part-ii-shading-lighting-and-shadows/chapter-18-using-vertex-texture-displacement$

Дата обращения: 18.08.2020

[4] Генерация Изображения Морской Поверхности В Реальном Времени.Режим доступа:

$https://gc2011.graphicon.ru/html/2000/Modelling/Yelyko_Belago_Lavrentyev$

Дата обращения: 9.08.2020

[5] GPU Gems. Chapter 30. Real-Time Simulation and Rendering of 3D Fluids.Режим доступа:

$https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-30-real-time-simulation-and-rendering-3d-fluids$

Дата обращения: 19.08.2020

[6] Редактор кода VS Code. Режим доступа:

https : *//code.visualstudio.com*

Дата обращения: 13.09.2020.

[7] ШлееМ. Qt 5.10. Профессиональное программирование на C++. - СПб.: БХВ-Петербург, 2018. - 1072 с.: ил. - (В подлиннике)

[8] D. F. Rogers. Procedural Elements for Computer Graphics. 2nd ed., 1998 – p. 300-517

[9] Е. А. Снижко. Компьютерная геометрия и графика [Текст], 2005.-17с.