Оглавление

Введение

Симуляция 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 водной поверхности, которая будет работать в реальном времени и задействовать только вычислительные мощности cpu, 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 правдоподобную закраску с тенями, 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 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.

1.1.1 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.

1.1.2 Метод Лагранжа

Всю сложность движущихся потоков жидкости способна описать система уравнений Навье-Стокса (1). Это физический подход к моделированию водной поверхности. Он позволяет получать реалистичную модель поверхности жидкости, но требует большого количества вычислительных ресурсов.

Уавнений Навье-Стокса:

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

Одним из решений этого уравнения и является метод Лагранжа [1]. 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.

1.2 Алгоритмы генерации морской волны

1.2.1 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 [2] высот художники могут легко управлять параметрами водной анимации вплоть до отдельных волн (см. рисунок 1.1). 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.

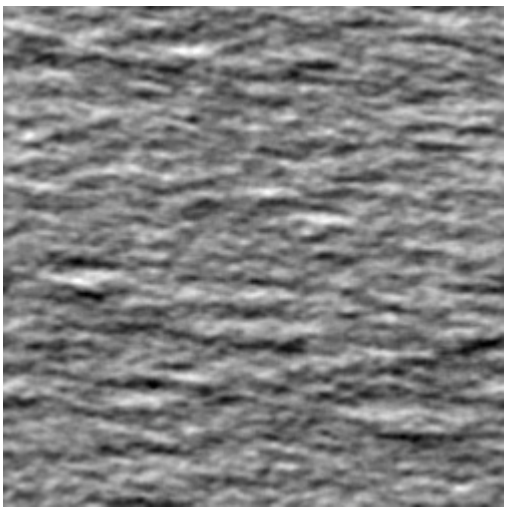

Рисунок 1.1 – пример карты высот

1.2.2 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 вол-

на распространяется в пространстве со скоростью, определяемой фазовой скоростью c.

1.2.3 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 [3].

Выделим набор параметров для определения каждой волны(рис. 1.2). 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.

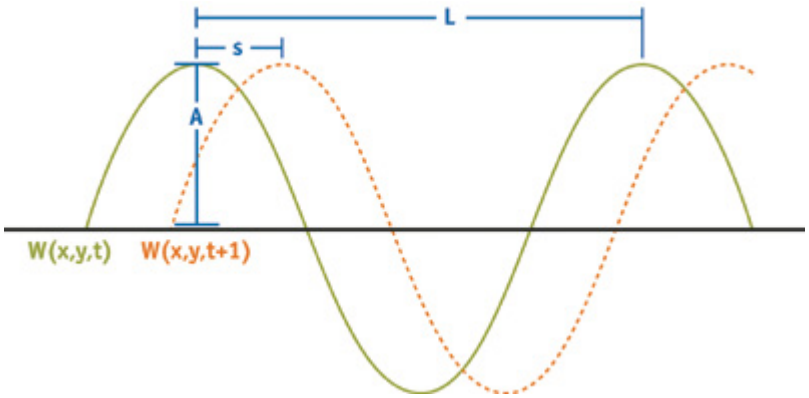

Рисунок 1.2 – пример гармонической волны и её параметров

Из данных параметров волны выделим функцию высоты волны:

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

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).

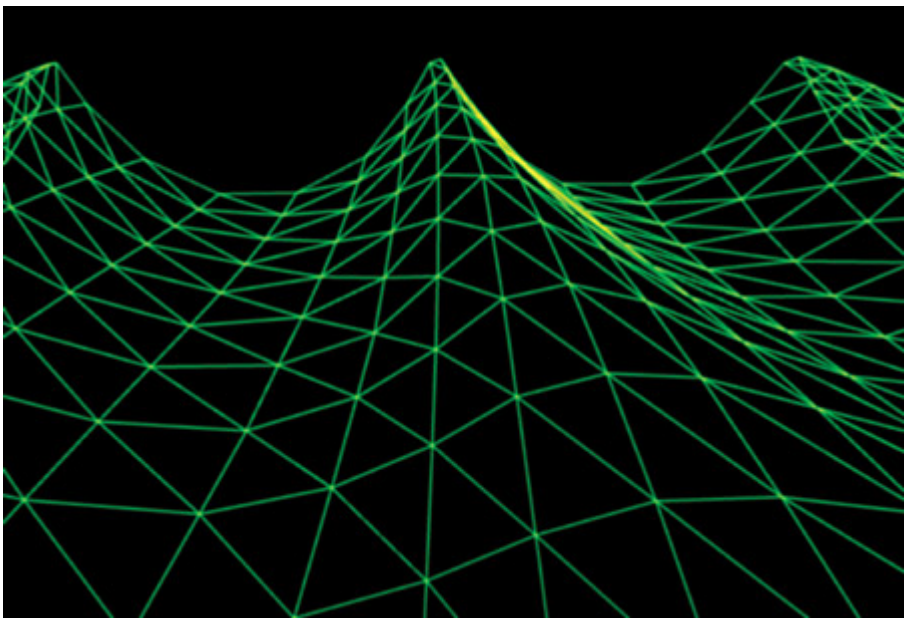

Рисунок 1.3 – пример волн, полученных алгоритмом Герстнера

1.3 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.

1.3.1 Алгоритм Робертса

Алгоритм Робертса [4] 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.

1.3.2 Алгоритм Варнока

Алгоритм Варнока [5] 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.

1.3.3 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.

1.3.4 Алгоритм, использующий z-буфер

Это один из простейших алгоритмов удаления невидимых поверхностей, который работает в пространстве изображения [6]. 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 сложные 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.

1.4 Анализ моделей освещения

1.4.1 Затенение по Гуро

Закраска по Гуро [7] 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. Данный алгоритм позволяет реализовать эфект мягких теней.

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 1.4)
3. Вычисление интенсивности в вершинах грани.
4. Интерполяция интенсивности вдоль ребер грани.
5. Линейная интерполяция интенсивности вдоль сканирующей строки.

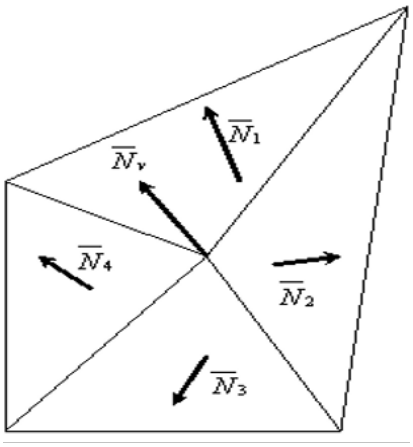

Рисунок 1.4 – вычисление векторов нормали к каждой вершине

1.4.2 Модель освещения Ламберта

Модель Ламберта [8] - 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 сравни-тельно гладких поверхностей. В отличии от нее модель Орен-Найара основана на предположении, что поверхность состоит из множества микро-граней, освещение каждой из которых описывается моделью Ламберта(см. рисунок 1.5).

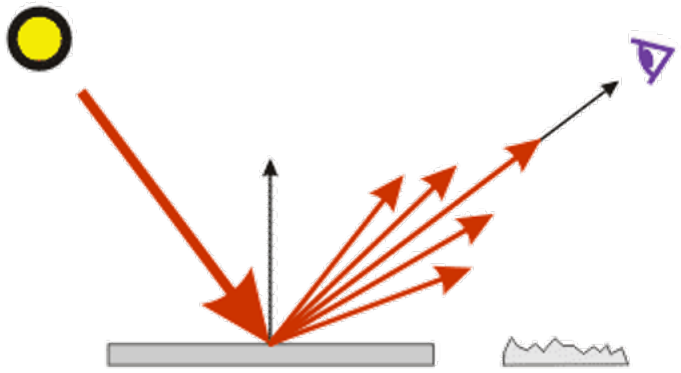

Рисунок 1.5 – падение и рассеивание света

1.4.3 Модель освещения Фонга

Модель Фонга [9] – 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.

1.4.4 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.

Вывод

Проведенный анализ существующих методов генерации водной поверхности позволяет сделать вывод, что для решения поставленной задачи наилучшим решением будет либо 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). Для получения более сглаженного изображения будет использован метод гуро, который также позволяет реализовать эфект теней, а также z-buffer для удаления невидимых линий.

2 Конструкторская часть

2.1 Подробный обзор волн Герстнера

В своей статье [3] 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):

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

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.

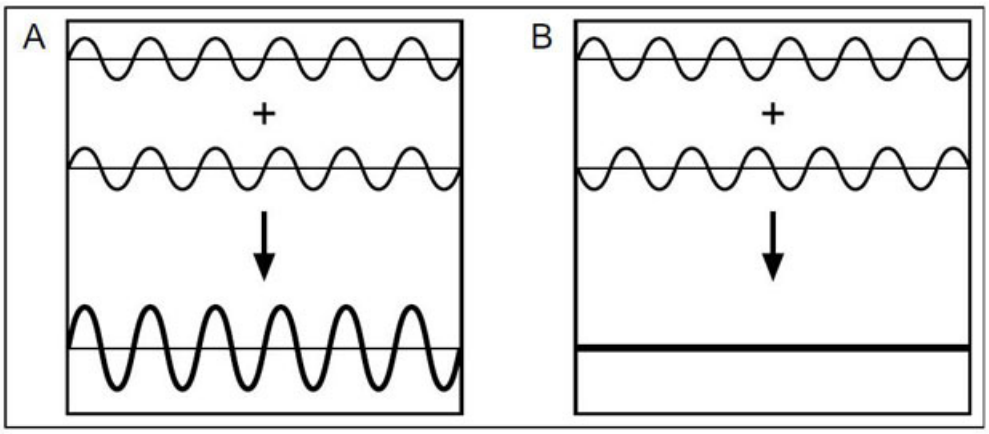

Рисунок 2.1 – Примеры наложения волн.

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 средств

В качестве языка программирования, на котором будет реализовано программное обеспечение, выбран язык программирования C++ [10]. Выбор языка обусловлен тем, что в нём есть поддержка объектно-ориентированной парадигмы, а также для этого языка существует фреймворк Qt [11], предоставляющая полный функционал, который нужен для создания пользовательского интерфейса программного обеспечения.

В качестве среды разработки выбран QtCreator [12], позволяющий легко работать с фраймворком Qt и содержащий большим количеством инструментов для различных языка программирования C++.

3.2 Листинг кода

В листингах 3.1 – 3.8 приведен исходный код реализации алгоритма симуляции водной поверхности.

Алгоритм разделён на модули: генерация волн Герстнера, обработчик волн, обработчик перемещения и поворотов, z-buffer с обработчиком освещения.

Листинг 3.1 – Заголовочный файл модуля обработчик волн.

```
1 #ifndef HANDLER_WAVES_H
2 #define HANDLER_WAVES_H
3
4 #include "waves.h"
5
6 class HandlerWaves {
7 public:
8     HandlerWaves(int width, int height);
9     shared_ptr<GerstWave> get_main_wave();
```

```
    void set_simulation(const double cur_time);
    void append_wave(shared_ptr<GerstWave> wave);
    void set_main_wave(shared_ptr<GerstWave> wave);
    Point2D get_point_start();
    Point2D get_point_fin();
    void set_points_start_fin(Point2D point_start, Point2D point_fin);
    void set_border(int width, int hieght);
    void set_center(shared_ptr<GerstWave> wave);
    void append_from_dsw(double amplitude, double len_wave, double speed, MathVector2D
        direction, double steepnes);
    void delete_wave(int index);
    void clear_vector_waves();
    vector<QString> get_disc();
    void reset_main_wave();

private:
    int f_width; int f_height;
    Point2D start_point, fin_point;
    shared_ptr<GerstWave> main_wave;
    shared_ptr<GerstWave> copy_wave;
    vector<shared_ptr<GerstWave>> vector_waves;
};

#endif // HANDLER_WAVES_H
```

Листинг 3.2 – Файл реализации модуля обработчик волн.

```
#include "handler_waves.h"


void HandlerWaves::set_center(shared_ptr<GerstWave> wave) {
    int h_width = f_width >> 1;
    int h_height = f_height >> 1;
    for (int i = 0; i < wave->get_len() + 1; i++) {
        for (int j = 0; j < wave->get_len() + 1; j++) {
            Point3D point(wave->get_point(i, j).get_x(),
                          wave->get_point(i, j).get_y(),
                          wave->get_point(i, j).get_z() - 0);
            wave->set_point(i, j, point);
        }
    }
}

void HandlerWaves::append_from_dsw(double amplitude, double len_wave, double speed,
    MathVector2D direction, double steepnes) {
    shared_ptr<GerstWave> new_wave(new GerstWave(start_point, fin_point,
        get_main_wave()->get_len()));
    new_wave->set_params(amplitude, len_wave, speed);
    new_wave->set_direction(direction);
```

```
21      new_wave->set_steepness(steepnes);
22      append_wave(new_wave);
23  }
24
25  void HandlerWaves::delete_wave(int index) {
26      vector_waves.erase(vector_waves.begin() + index);
27  }
28
29  void HandlerWaves::clear_vector_waves() {
30      vector_waves.clear();
31  }
32
33  vector<QString> HandlerWaves::get_disc() {
34      vector<QString> temp;
35      for (int i = 0; i < vector_waves.size(); ++i) {
36          temp.push_back(vector_waves[i]->get_string_disc());
37      }
38      return temp;
39  }
40
41  void HandlerWaves::reset_main_wave()
42  {
43      copy_wave = main_wave->copy();
44  }
45
46
47
48
49  HandlerWaves::HandlerWaves(int width, int height) : f_width(width), f_height(height) {
50
51      start_point = Point2D(0, 0);
52      fin_point = Point2D(600, 600);
53
54      main_wave = shared_ptr<GerstWave>(new GerstWave(Point2D(0, 0), Point2D(600, 600),
            10));
55      main_wave->set_params(30, 100, 20);
56      main_wave->set_direction(MathVector2D(1, 1));
57      main_wave->set_steepness(0.003);
58      copy_wave = main_wave->copy();
59      set_center(copy_wave);
60  }
61
62  shared_ptr<GerstWave> HandlerWaves::get_main_wave() {
63      return copy_wave;
64  }
65
66  void HandlerWaves::set_simulation(const double cur_time) {
67      copy_wave = main_wave->copy();
```

```
68      for (int i = 0; i < copy_wave->get_len() + 1; i++) {
69          for (int j = 0; j < copy_wave->get_len() + 1; j++) {
70              double cur_x = copy_wave->get_point(i, j).get_x() +
                    main_wave->get_offset_x(main_wave->get_point(i, j), cur_time);
71              double cur_y = copy_wave->get_point(i, j).get_y() +
                    main_wave->get_offset_y(main_wave->get_point(i, j), cur_time);
72              double cur_z = main_wave->get_height(main_wave->get_point(i, j), cur_time);
73              for (int ind = 0; ind < vector_waves.size(); ind++) {
74                  cur_x += vector_waves[ind]->get_offset_x(vector_waves[ind]->get_point(i,
                        j), cur_time);
75                  cur_y += vector_waves[ind]->get_offset_y(vector_waves[ind]->get_point(i,
                        j), cur_time);
76                  cur_z += vector_waves[ind]->get_height(vector_waves[ind]->get_point(i, j),
                        cur_time);
77              }
78              copy_wave->set_point(i, j, Point3D(cur_x, cur_y, cur_z));
79          }
80      }
81      set_center(copy_wave);
82  }
83
84  void HandlerWaves::append_wave(shared_ptr<GerstWave> wave) {
85      vector_waves.push_back(wave);
86  }
87
88  void HandlerWaves::set_main_wave(shared_ptr<GerstWave> wave) {
89      main_wave = wave;
90      copy_wave = main_wave->copy();
91      set_center(copy_wave);
92  }
93
94  Point2D HandlerWaves::get_point_start() {
95      return start_point;
96  }
97
98  Point2D HandlerWaves::get_point_fin() {
99      return fin_point;
100 }
101
102 void HandlerWaves::set_points_start_fin(Point2D point_start, Point2D point_fin) {
103     start_point = point_start;
104     fin_point = point_fin;
105 }
106
107 void HandlerWaves::set_border(int width, int hieght) {
108     f_width = width;
109     f_height = hieght;
110 }
```

Листинг 3.3 – Заголовочный файл обработчика перемещения и поворотов.

```
#ifndef HANDLER_TRANSFORM_H
#define HANDLER_TRANSFORM_H

#include <memory>

#include "waves.h"

using namespace std;

enum RotateType {
    ROTATE_OX,
    ROTATE_OY,
    ROTATE_OZ
};

enum MoveType {
    MOVE_OX,
    MOVE_OY,
    MOVE_OZ
};

class HandlerTransform {
public:
    HandlerTransform();
    double rotate_wave(shared_ptr<GerstWave> wave, const RotateType type, const double
        alpha=0);
    double move_wave(shared_ptr<GerstWave> wave, const MoveType type, const double dt=0);
    void set_data_move_dialog(const double _dx_move, const double _dy_move, const double
        _dz_move);
    void set_data_rotate_dialog(const double _dx_rotate, const double _dy_rotate, const
        double _dz_rotate);

private:
    double dx_move, dy_move, dz_move;
    double dx_rotate, dy_rotate, dz_rotate;
    double get_need_value(const double new_param, const double std_param);
    double get_radians(int degree);
    void rotate_dx(shared_ptr<GerstWave> wave, const double angle);
    void rotate_dy(shared_ptr<GerstWave> wave, const double angle);
    void rotate_dz(shared_ptr<GerstWave> wave, const double angle);
    void move_dx(shared_ptr<GerstWave> wave, const double dx);
    void move_dy(shared_ptr<GerstWave> wave, const double dy);
    void move_dz(shared_ptr<GerstWave> wave, const double dz);
    Point3D mult_matr(vector<vector<double>> coords, vector<vector<double>>
        transfer_matrix, double offset_x, double offset_y);
    double get_offset_x(shared_ptr<GerstWave> wave);
    double get_offset_y(shared_ptr<GerstWave> wave);
```

```
44
45 private slots:
46     void get_data_dialog();
47 };
48
49 #endif // HANDLER_TRANSFORM_H
```

Листинг 3.4 – Файл реализации обработчика перемещения и поворотов.

```
1  #include "handler_transform.h"
2
3
4  HandlerTransform::HandlerTransform() {
5      dx_move = 5; dy_move = 5; dz_move = 5;
6      dx_rotate = 5; dy_rotate = 5; dz_rotate = 5;
7  }
8
9
10 double HandlerTransform::rotate_wave(shared_ptr<GerstWave> wave, const RotateType type,
       const double alpha) {
11     double angle = 0;
12     switch (type) {
13     case ROTATE_OX:
14         angle = get_need_value(alpha, dx_rotate * M_PI / 180);
15         rotate_dx(wave, angle);
16         break;
17     case ROTATE_OY:
18         angle = get_need_value(alpha, dy_rotate * M_PI / 180);
19         rotate_dy(wave, angle);
20         break;
21     case ROTATE_OZ:
22         angle = get_need_value(alpha, dz_rotate * M_PI / 180);
23         rotate_dz(wave, angle);
24         break;
25     }
26
27     return angle;
28 }
29
30
31 double HandlerTransform::move_wave(shared_ptr<GerstWave> wave, const MoveType type,
       const double dt) {
32     double d = 0;
33     switch (type) {
34     case MOVE_OX:
35         d = get_need_value(dt, dx_move);
36         move_dx(wave, d);
37         break;
38     case MOVE_OY:
```

```
39          d = get_need_value(dt, dy_move);
40          move_dy(wave, d);
41          break;
42      case MOVE_OZ:
43          d = get_need_value(dt, dz_move);
44          move_dz(wave, d);
45          break;
46      }
47
48      return d;
49  }
50
51
52  void HandlerTransform::set_data_move_dialog(const double _dx_move, const double
        _dy_move, const double _dz_move) {
53      dx_move = _dx_move;
54      dy_move = _dy_move;
55      dz_move = _dz_move;
56  }
57
58
59  void HandlerTransform::set_data_rotate_dialog(const double _dx_rotate, const double
        _dy_rotate, const double _dz_rotate) {
60      dx_rotate = _dx_rotate;
61      dy_rotate = _dy_rotate;
62      dz_rotate = _dz_rotate;
63  }
64
65
66  double HandlerTransform::get_need_value(const double new_param, const double std_param) {
67      if (new_param) {
68          return new_param;
69      }
70
71      return std_param;
72  }
73
74
75  void HandlerTransform::rotate_dx(shared_ptr<GerstWave> wave, const double angle) {
76      double offset_x = get_offset_x(wave), offset_y = get_offset_y(wave);
77
78      vector<vector<double>> transfer_matrix {
79          {1, 0, 0, 0},
80          {0, cos(angle), -sin(angle), 0},
81          {0, sin(angle), cos(angle), 0},
82          {0, 0, 0, 1}
83      };
84
```

```
85      for (int i = 0; i < wave->get_len() + 1; i++) {
86          for (int j = 0; j < wave->get_len() + 1; j++) {
87              vector<vector<double>> coords{
88                  {
89                      wave->get_point(i, j).get_x() - offset_x,
90                      wave->get_point(i, j).get_y() - offset_y,
91                      wave->get_point(i, j).get_z(),
92                      1
93                  }
94              };
95              wave->set_point(i, j, mult_matr(coords, transfer_matrix, offset_x, offset_y));
96          }
97      }
98  }
99
100
101 void HandlerTransform::rotate_dy(shared_ptr<GerstWave> wave, const double angle) {
102     double offset_x = get_offset_x(wave), offset_y = get_offset_y(wave);
103
104     vector<vector<double>> transfer_matrix {
105         {cos(-angle), 0, -sin(-angle), 0},
106         {0, 1, 0, 0},
107         {sin(-angle), 0, cos(-angle), 0},
108         {0, 0, 0, 1}
109     };
110
111     for (int i = 0; i < wave->get_len() + 1; i++) {
112         for (int j = 0; j < wave->get_len() + 1; j++) {
113             vector<vector<double>> coords{
114                 {
115                     wave->get_point(i, j).get_x() - offset_x,
116                     wave->get_point(i, j).get_y() - offset_y,
117                     wave->get_point(i, j).get_z(),
118                     1
119                 }
120             };
121             wave->set_point(i, j, mult_matr(coords, transfer_matrix, offset_x, offset_y));
122         }
123     }
124 }
125
126
127 void HandlerTransform::rotate_dz(shared_ptr<GerstWave> wave, const double angle) {
128     double offset_x = get_offset_x(wave), offset_y = get_offset_y(wave);
129
130     vector<vector<double>> transfer_matrix {
131         {cos(angle), -sin(angle), 0, 0},
132         {sin(angle), cos(angle), 0, 0},
```

```
        {0, 0, 1, 0},
        {0, 0, 0, 1}
    };

    for (int i = 0; i < wave->get_len() + 1; i++) {
        for (int j = 0; j < wave->get_len() + 1; j++) {
            vector<vector<double>> coords{
                {
                    wave->get_point(i, j).get_x() - offset_x,
                    wave->get_point(i, j).get_y() - offset_y,
                    wave->get_point(i, j).get_z(),
                    1
                }
            };
            wave->set_point(i, j, mult_matr(coords, transfer_matrix, offset_x, offset_y));
        }
    }
}


void HandlerTransform::move_dx(shared_ptr<GerstWave> wave, const double dx) {
    for (int i = 0; i < wave->get_len() + 1; i++) {
        for (int j = 0; j < wave->get_len() + 1; j++) {
            Point3D new_point(wave->get_point(i, j).get_x() + dx,
                              wave->get_point(i, j).get_y(),
                              wave->get_point(i, j).get_z());
            wave->set_point(i, j, new_point);
        }
    }
}


void HandlerTransform::move_dy(shared_ptr<GerstWave> wave, const double dy) {
    for (int i = 0; i < wave->get_len() + 1; i++) {
        for (int j = 0; j < wave->get_len() + 1; j++) {
            Point3D new_point(wave->get_point(i, j).get_x(),
                              wave->get_point(i, j).get_y() + dy,
                              wave->get_point(i, j).get_z());
            wave->set_point(i, j, new_point);
        }
    }
}


void HandlerTransform::move_dz(shared_ptr<GerstWave> wave, const double dz) {
}


```

```
181 Point3D HandlerTransform::mult_matr(vector<vector<double> > coords,
        vector<vector<double> > transfer_matrix, double offset_x, double offset_y)
182 {
183     vector<vector<double>> coords_point{{0, 0, 0, 0}};
184 
185     for(int i = 0; i < coords.size(); i++) {
186         for(int j = 0; j < transfer_matrix[0].size(); j++)
187         {
188             coords_point[i][j] = 0;
189             for(int k = 0; k < transfer_matrix.size(); k++)
190                 coords_point[i][j] += coords[i][k] * transfer_matrix[k][j];
191         }
192     }
193 
194     return Point3D(coords_point[0][0] + offset_x, coords_point[0][1] + offset_y,
            coords_point[0][2]);
195 }
196 
197 
198 double HandlerTransform::get_offset_x(shared_ptr<GerstWave> wave) {
199     double max_x = -100000, min_x = 10000000;
200     double temp = 0;
201 
202     for (int i = 0; i < wave->get_len() + 1; i++) {
203         for (int j = 0; j < wave->get_len() + 1; j++) {
204             temp = wave->get_point(i, j).get_x();
205             if (temp > max_x) {
206                 max_x = temp;
207             }
208             if (temp < min_x) {
209                 min_x = temp;
210             }
211         }
212     }
213 
214     return min_x + ((max_x - min_x) / 2);
215 }
216 
217 
218 double HandlerTransform::get_offset_y(shared_ptr<GerstWave> wave) {
219     double max_y = -100000, min_y = 10000000;
220     double temp = 0;
221 
222     for (int i = 0; i < wave->get_len() + 1; i++) {
223         for (int j = 0; j < wave->get_len() + 1; j++) {
224             temp = wave->get_point(i, j).get_y();
225             if (temp > max_y) {
226                 max_y = temp;
```

```
            }
            if (temp < min_y) {
                min_y = temp;
            }
        }
    }

    return min_y + ((max_y - min_y) / 2);
}
```

Листинг 3.5 – Заголовочный файл генерации волн.

```
#ifndef WAVES_H
#define WAVES_H


#include "surface.h"
#include <memory>


/*
Class for harmonic wave
*/
class HarmonicWave: public Surface {
private:
    // params
    double amplitude = -1, wave_len = -1, speed = -1, omega = -1, phi = -1;
    MathVector2D direction = MathVector2D(0, 0);
public:
    Point2D _start_point, _fin_point;
    HarmonicWave(const Point2D start_point, const Point2D fin_point, const int
        parts_ver);
    shared_ptr<HarmonicWave> copy(HarmonicWave &wave);
    void set_params(const double new_amplitude = 0, const double new_wave_len = 0, const
        double new_speed = 0);
    void set_direction(const MathVector2D direction);
    MathVector2D get_direction() const;
    double get_amplitude() const;
    double get_wave_len() const;
    double get_speed() const;
    double get_omega() const;
    double get_phi() const;
    // set value x, y, z for cur times in all point matrix
    virtual void set_simulation(const double cur_time);
    void setup_base_wave();
    double get_height(const Point2D point, const double cur_time) const;

protected:
    void check_params(const double amplitude, const double wave_len, const double speed)
```

```
        const;
36
37 };
38
39 /*
40 Class for Gerstner wave
41 */
42 class GerstWave: public HarmonicWave {
43 private:
44     double steepness = -1;
45 public:
46     double get_offset_x(const Point3D cur_point, const double cur_time) const;
47     double get_offset_y(const Point3D cur_point, const double cur_time) const;
48     Point2D _start_point, _fin_point;
49     GerstWave(const Point2D start_point, const Point2D fin_point, const int parts_ver);
50     shared_ptr<GerstWave> copy();
51     void set_steepness(const double steepness);
52     double get_steepness();
53     // set value x, y, z for cur times in all point matrix
54     virtual void set_simulation(const double cur_time);
55     QString get_string_disc();
56 protected:
57     void check_steepness() const;
58 };
59
60
61 #endif // WAVES_H
```

Листинг 3.6 – Файл реализации генерации волн.

```
1  #include "waves.h"
2  #include "exception.h"
3  #include "math_vectors_operation.h"
4  #include <math.h>
5  #include <QDebug>
6
7  #ifdef _DEBUG_
8      void print_data_HarmonicWave(HarmonicWave test_wave) {
9          cout << "debug␣print:␣" << endl << endl;
10         for (size_t i = 0; i < test_wave.get_len(); ++i) {
11             for (size_t j = 0; j < test_wave.get_len(); ++j) {
12                 cout << test_wave.get_point(i, j) << "␣";
13             }
14             cout << endl;
15         }
16     }
17 #endif
18
19
```

```
20 HarmonicWave::HarmonicWave(const Point2D start_point, const Point2D fin_point, const int
       parts_ver): Surface(start_point, fin_point, parts_ver), _start_point(start_point),
       _fin_point(fin_point) {
21 }
22
23
24 shared_ptr<HarmonicWave> HarmonicWave::copy(HarmonicWave &wave)
25 {
26     shared_ptr<HarmonicWave> new_wave(new HarmonicWave(wave._start_point,
           wave._fin_point, wave.get_len()));
27     return new_wave;
28 }
29
30
31 void HarmonicWave::check_params(const double amplitude, const double wave_len, const
       double speed) const {
32     if (amplitude < 0 || wave_len < 0 || speed < 0) {
33         throw NegativeValueException("All␣params(amplitude,␣wave_len,␣speed)␣must␣not␣be␣
               less␣than␣0");
34     }
35 }
36
37
38 void HarmonicWave::set_params(const double new_amplitude, const double new_wave_len,
       const double new_speed) {
39     check_params(new_amplitude, new_wave_len, new_speed);
40
41     amplitude = new_amplitude;
42     wave_len = new_wave_len;
43     speed = new_speed;
44
45     omega = 2 / wave_len;
46     phi = speed * (2 / wave_len);
47 }
48
49
50 void HarmonicWave::set_direction(const MathVector2D direction) {
51     this->direction = direction;
52 }
53
54
55 double HarmonicWave::get_height(const Point2D cur_point, const double cur_time) const {
56     check_params(amplitude, wave_len, speed);
57
58     MathVector2D position(cur_point.get_x(), cur_point.get_y());
59     double rez = amplitude * sin(get_scal_mult(direction, position) * omega + cur_time *
           phi);
60     return rez;
```

```
61 }
62
63
64
65
66 void HarmonicWave::set_simulation(const double cur_time) {
67     check_params(amplitude, wave_len, speed);
68
69     for (int i = 0; i < get_len(); ++i) {
70         for (int j = 0; j < get_len(); ++j) {
71             double cur_x = get_point(i, j).get_x();
72             double cur_y = get_point(i, j).get_y();
73             double cur_z = get_height(get_point(i, j), cur_time);
74             set_point(i, j, Point3D(cur_x, cur_y, cur_z));
75         }
76     }
77 }
78
79
80 void HarmonicWave::setup_base_wave() {
81     init_matrix_point();
82 }
83
84
85 double HarmonicWave::get_amplitude() const {
86     return amplitude;
87 }
88
89
90 double HarmonicWave::get_wave_len() const {
91     return wave_len;
92 }
93
94
95 double HarmonicWave::get_speed() const {
96     return speed;
97 }
98
99
100 double HarmonicWave::get_omega() const {
101     return omega;
102 }
103
104
105 double HarmonicWave::get_phi() const {
106     return phi;
107 }
108
```

```
109
110 MathVector2D HarmonicWave::get_direction() const {
111     return direction;
112 }
113
114
115 GerstWave::GerstWave(const Point2D start_point, const Point2D fin_point, const int
        parts_ver): HarmonicWave(start_point, fin_point, parts_ver),
        _start_point(start_point), _fin_point(fin_point) {
116
117 }
118
119
120 shared_ptr<GerstWave> GerstWave::copy()
121 {
122     shared_ptr<GerstWave> new_wave(new GerstWave(this->_start_point, this->_fin_point,
            this->get_len()));
123     new_wave->set_params(this->get_amplitude(), this->get_wave_len(), this->get_speed());
124     new_wave->set_steepness(this->get_steepness());
125     new_wave->set_direction(this->get_direction());
126     for (int i = 0; i < new_wave->get_len() + 1; i++) {
127         for (int j = 0; j < new_wave->get_len() + 1; j++) {
128             new_wave->set_point(i, j, this->get_point(i, j));
129         }
130     }
131     return new_wave;
132 }
133
134
135 void GerstWave::set_steepness(const double steepness) {
136     this->steepness = steepness;
137     check_steepness();
138 }
139
140 double GerstWave::get_steepness() {
141     return steepness;
142 }
143
144
145 void GerstWave::check_steepness() const {
146     if (steepness < 0 || steepness > 1 / (get_amplitude() * get_omega())) {
147         throw OutOfRangeException("Steepness must be in [0; 1/(omega * amplitude)");
148     }
149 }
150
151
152 double GerstWave::get_offset_x(const Point3D cur_point, const double cur_time) const {
153     MathVector2D position(cur_point.get_x(), cur_point.get_y());
```

```
154     return steepness * get_amplitude() * get_direction().get_x_coord()
155             * cos(get_omega() * get_scal_mult(get_direction(), position) +
156             get_phi() * cur_time);
157 }
158
159
160 double GerstWave::get_offset_y(const Point3D cur_point, const double cur_time) const {
161     MathVector2D position(cur_point.get_x(), cur_point.get_y());
162     return steepness * get_amplitude() * get_direction().get_y_coord()
163             * cos(get_omega() * get_scal_mult(get_direction(), position) +
164             get_phi() * cur_time);
165 }
166
167
168 void GerstWave::set_simulation(const double cur_time) {
169     check_steepness();
170     check_params(get_amplitude(), get_wave_len(), get_speed());
171
172     for (int i = 0; i < get_len() + 1; ++i) {
173         for (int j = 0; j < get_len() + 1; ++j) {
174             double cur_x = get_point(i, j).get_x() + get_offset_x(get_point(i, j),
                    cur_time);
175             double cur_y = get_point(i, j).get_y() + get_offset_y(get_point(i, j),
                    cur_time);
176             double cur_z = get_height(get_point(i, j), cur_time);
177             set_point(i, j, Point3D(cur_x, cur_y, cur_z));
178         }
179     }
180 }
181
182 QString GerstWave::get_string_disc() {
183     QString valueAsString = "ampl:" + QString::number(get_amplitude()) +
184             "␣len␣wave:" + QString::number(get_wave_len()) + "␣speed␣wave:" +
                    QString::number(get_speed())
185             + "␣dir␣x:" + QString::number(get_direction().get_x_coord(),'f', 2) + "␣dir␣
                    y:" + QString::number(get_direction().get_y_coord(),'f', 2)
186             + "␣steep:" + QString::number(get_steepness(),'f', 6);
187     return valueAsString;
188 }
```

Листинг 3.7 – Заголовочный файл z-buffer с обработчиком освещения.

```
1 #ifndef Z_BUFFER_H
2 #define Z_BUFFER_H
3
4 #include <QPixmap>
5 #include <QGraphicsPixmapItem>
6 #include <QGraphicsScene>
7
```

```
#include "math_vectors_operation.h"
#include "waves.h"
#include "structurs.h"
#include "light.h"


class ZBuffer {
public:
    ZBuffer();
    void fill_scene(QImage *image);
    void start_fill(shared_ptr<GerstWave> wave, QImage *image);
    void set_polygons(shared_ptr<GerstWave> wave);
    void set_border(int width, int hieght);
    void set_color_wave(QColor new_color);
    void set_light(Light new_light);
    QColor get_color_light();
    QColor get_color_wave();
private:
    QColor color_wave;
    double *m_buffer;
    Light m_light;
    unsigned int z_buffer_size;
    int f_width, f_height;
    vector<Triangle> triangle_buffer;
    MathVector3D get_normal(const Point3D point1, const Point3D point2, const Point3D
        point3);
    void set_coords_points(double *const x, double *const y, double *const z, Triangle&
        tempSurface);
    void set_color(int * const rc, int * const gc, int * const bc, double *x, double *y,
        double *z_a, double *cosinus, MathVector3D l, MathVector3D norm);
    void resize_buffer(unsigned int cur_z_buffer_size);
    void restart_buffer();
};

#endif // Z_BUFFER_H
```

Листинг 3.8 – Файл реализации z-buffer с обработчиком освещения.

```
#include "z_buffer.h"
#include "debuger.h"


ZBuffer::ZBuffer() : m_light(Light(Point3D(50, 10, -500), QColor(220, 219, 146, 255))) {
    f_width = 800; f_height = 800;
    z_buffer_size = f_width * f_height;
    m_buffer = new double[z_buffer_size];
    for(unsigned int i = 0; i < z_buffer_size; i++) {
        m_buffer[i] = -1000000;
    }
```

```
12
13      set_color_wave(QColor(6, 190, 210, 255));
14  }
15
16
17  void ZBuffer::fill_scene(QImage *image) {
18      rgba64 *bits = reinterpret_cast<rgba64 *>(image->bits());
19      double xScanLine1, xScanLine2;
20      double x1 = 0, x2 = 0, tc = 0, z1 = 0, z2 = 0, z = 0;
21      double x[3], y[3], z_a[3];
22      bool bordersFound;
23      int ymax, ymin, yScanLine, e1;
24      MathVector3D view = MathVector3D(0, 0, 0);
25      Triangle tempSurface;
26      Normal normal;
27
28      for(unsigned int j = 0; j < triangle_buffer.size(); ++j) {
29          tempSurface = triangle_buffer.at(j);
30          normal = tempSurface.get_normal();
31          view = MathVector3D(
32              m_light.get_direction().get_x_coord() - tempSurface.get_p1().get_x(),
33              m_light.get_direction().get_y_coord() - tempSurface.get_p1().get_y(),
34              m_light.get_direction().get_z_coord() - tempSurface.get_p1().get_z()
35          );
36          set_coords_points(&x[0], &y[0], &z_a[0], tempSurface);
37          ymin = std::min(std::min(y[0], y[1]),y[2]);
38          ymax = std::max(std::max(y[0], y[1]),y[2]);
39
40          MathVector3D l = m_light.get_direction();
41          MathVector3D norm = normal.get_normalize();
42          double cosinus[3];
43          int rc[3], bc[3], gc[3];
44
45          set_color(
46              &rc[0], &gc[0], &bc[0],
47              &x[0], &y[0], &z_a[0],
48              &cosinus[0], l, norm
49          );
50
51          int i1 = 0, i2 = 0;
52          for (yScanLine = round(ymin); yScanLine < round(ymax); yScanLine++) {
53              bordersFound = false;
54              for (int e = 0; e < 3; e++) {
55                  e1 = e + 1;
56                  if (e1 == 3) { e1 = 0; }
57
58                  if (y[e] < y[e1] && (y[e1] <= yScanLine || yScanLine < y[e])) {
59                      continue;
```

```
60              } else if (y[e] > y[e1] && (y[e1] > yScanLine || yScanLine >= y[e])) {
61                  continue;
62              } else if (y[e] == y[e1]) {
63                  continue;
64              }
65
66              tc = (y[e] - yScanLine) / (y[e] - y[e1]);
67              if (bordersFound) {
68                  x2 = x[e] + ( tc * (x[e1] - x[e]));
69                  z2 = z_a[e] + tc * (z_a[e1] - z_a[e]);
70                  i2 = qRgba(
71                      Light::check_rgb_color((int)(rc[e] + tc*(rc[e1] - rc[e]))),
72                      Light::check_rgb_color((int)(gc[e] + tc*(gc[e1] - gc[e]))),
73                      Light::check_rgb_color((int)(bc[e] + tc*(bc[e1] - bc[e]))),
74                      180
75                  );
76              } else {
77                  x1 = x[e] + (tc * (x[e1] - x[e]));
78                  z1 = z_a[e] + tc * (z_a[e1] - z_a[e]);
79                  i1 = qRgba(Light::check_rgb_color((int)(rc[e] + tc*(rc[e1] - rc[e]))),
80                            Light::check_rgb_color((int)(gc[e] + tc*(gc[e1] - gc[e]))),
81                            Light::check_rgb_color((int)(bc[e] + tc*(bc[e1] - bc[e]))),
82                            180);
83                  bordersFound = true;
84              }
85          }
86
87          if (x2 < x1) {
88              std::swap(x2, x1);
89              std::swap(z2, z1);
90              std::swap(i2, i1);
91          }
92
93          xScanLine1 = (x1 < -1*std::numeric_limits<float>::max()) ?
                -1*std::numeric_limits<float>::max() : x1;
94          xScanLine2 = (x2 < f_width) ? x2 : f_width;
95          for (int xScanLine = round(xScanLine1); xScanLine < round(xScanLine2);
                xScanLine++) {
96              float tc = (x1 - xScanLine) / (x1 - x2);
97              z = z1 + tc * (z2 - z1);
98
99
100             int index = (int)round((xScanLine) +(yScanLine) * f_width);
101             if (z > m_buffer[index]) {
102                 int clr = qRgba(Light::check_rgb_color((int)(qRed(i1) + tc*(qRed(i2) -
                        qRed(i1)))),
103                                 Light::check_rgb_color((int)(qGreen(i1) + tc*(qGreen(i2)
                                    - qGreen(i1)))),
```

```
104                                 Light::check_rgb_color((int)(qBlue(i1) + tc*(qBlue(i2) -
                                        qBlue(i1)))), 100);
105                     QRgb l_color = qRgba(get_color_light().red(),
                            get_color_light().green(), get_color_light().green(),
                            get_color_light().alpha());
106
107                     QRgb color = qRgba(qRed(clr)*0.70 +
                            0.30*qRed(l_color)*qAlpha(l_color)/255,
108                                 qGreen(clr)*0.70 + 0.30*qGreen(l_color)*qAlpha(l_color)/255,
109                                 qBlue(clr)*0.70 + 0.30*qBlue(l_color)*qAlpha(l_color)/255,
                                        100);
110                     m_buffer[index] = z;
111
112                     uint16_t red_c = (qRed(color) / 255.0) * 65365;
113                     uint16_t green_c = (qGreen(color) / 255.0) * 65365;
114                     uint16_t blue_c = (qBlue(color) / 255.0) * 65365;
115
116                     bits[index].red = red_c;
117                     bits[index].green = green_c;
118                     bits[index].blue = blue_c;
119                 }
120             }
121         }
122     }
123 }
124
125
126 void ZBuffer::start_fill(shared_ptr<GerstWave> wave, QImage *image) {
127     set_polygons(wave);
128     restart_buffer();
129     fill_scene(image);
130 }
131
132
133 void ZBuffer::set_polygons(shared_ptr<GerstWave> wave) {
134     triangle_buffer.clear();
135     Triangle temp;
136     MathVector3D math_vector_normal(0, 0, 0);
137     for (int i = 1; i < wave->get_len(); i+=2) {
138         for (int j = 1; j < wave->get_len(); j+=2) {
139             math_vector_normal = get_normal(wave->get_point(i, j), wave->get_point(i - 1,
                    j - 1), wave->get_point(i + 1, j - 1));
140             temp = Triangle(wave->get_point(i - 1, j - 1), wave->get_point(i + 1, j - 1),
                    wave->get_point(i, j)); // 1
141             temp.set_normal(
142                 Normal(
143                     math_vector_normal.get_x_coord(),
144                     math_vector_normal.get_y_coord(),
```

```
145                     math_vector_normal.get_z_coord()
146                 )
147             );
148             triangle_buffer.push_back(temp);
149 
150 
151             math_vector_normal = get_normal(wave->get_point(i, j), wave->get_point(i + 1,
                    j - 1), wave->get_point(i + 1, j + 1));
152             temp = Triangle(wave->get_point(i + 1, j - 1), wave->get_point(i + 1, j + 1),
                    wave->get_point(i, j)); // d
153             temp.set_normal(
154                 Normal(
155                     math_vector_normal.get_x_coord(),
156                     math_vector_normal.get_y_coord(),
157                     math_vector_normal.get_z_coord()
158                 )
159             );
160             triangle_buffer.push_back(temp);
161 
162 
163             math_vector_normal = get_normal(wave->get_point(i, j), wave->get_point(i + 1,
                    j + 1), wave->get_point(i - 1, j + 1));
164             temp = Triangle(wave->get_point(i + 1, j + 1), wave->get_point(i - 1, j + 1),
                    wave->get_point(i, j)); // r
165             temp.set_normal(
166                 Normal(
167                     math_vector_normal.get_x_coord(),
168                     math_vector_normal.get_y_coord(),
169                     math_vector_normal.get_z_coord()
170                 )
171             );
172             triangle_buffer.push_back(temp);
173 
174             math_vector_normal = get_normal(wave->get_point(i, j), wave->get_point(i - 1,
                    j + 1), wave->get_point(i - 1, j - 1));
175             temp = Triangle(wave->get_point(i - 1, j + 1), wave->get_point(i - 1, j - 1),
                    wave->get_point(i, j)); // u
176             temp.set_normal(
177                 Normal(
178                     math_vector_normal.get_x_coord(),
179                     math_vector_normal.get_y_coord(),
180                     math_vector_normal.get_z_coord()
181                 )
182             );
183             triangle_buffer.push_back(temp);
184         }
185     }
186 }
```

```
187
188
189 void ZBuffer::set_border(int width, int hieght) {
190     f_width = width;
191     f_height = hieght;
192     z_buffer_size = f_width * f_height;
193     resize_buffer(z_buffer_size);
194 }
195
196
197 void ZBuffer::set_color_wave(QColor new_color) {
198     color_wave.setRed(new_color.red());
199     color_wave.setGreen(new_color.green());
200     color_wave.setBlue(new_color.blue());
201     color_wave.setAlpha(new_color.alpha());
202 }
203
204
205 void ZBuffer::set_light(Light new_light) {
206     m_light = new_light;
207 }
208
209
210 QColor ZBuffer::get_color_light() {
211     return m_light.get_color();
212 }
213
214 QColor ZBuffer::get_color_wave() {
215     return color_wave;
216 }
217
218
219 MathVector3D ZBuffer::get_normal(const Point3D point1, const Point3D point2, const
        Point3D point3) {
220     MathVector3D mv1 = MathVector3D(
221         point2.get_x() - point1.get_x(),
222         point2.get_y() - point1.get_y(),
223         point2.get_z() - point1.get_z()
224     );
225
226     MathVector3D mv2 = MathVector3D(
227         point3.get_x() - point1.get_x(),
228         point3.get_y() - point1.get_y(),
229         point3.get_z() - point1.get_z()
230     );
231
232      MathVector3D mv = MathVector3D::crossProduct(mv1, mv2);
233
```

```
234     return mv;
235 }
236
237
238 void ZBuffer::set_coords_points(double * const x, double * const y, double * const z,
        Triangle &tempSurface) {
239     x[0] = round(tempSurface.get_p1().get_x()); y[0] =
            round(tempSurface.get_p1().get_y()); z[0] = tempSurface.get_p1().get_z();
240     x[1] = round(tempSurface.get_p2().get_x()); y[1] =
            round(tempSurface.get_p2().get_y()); z[1] = tempSurface.get_p2().get_z();
241     x[2] = round(tempSurface.get_p3().get_x()); y[2] =
            round(tempSurface.get_p3().get_y()); z[2] = tempSurface.get_p3().get_z();
242 }
243
244
245 void ZBuffer::set_color(int * const rc, int * const gc, int * const bc, double *x,
        double *y, double *z_a, double *cosinus, MathVector3D l, MathVector3D norm) {
246     QRgb color = qRgba(color_wave.red(), color_wave.green(), color_wave.blue(), 255);
247     for (int i = 0; i < 3; ++i) {
248         MathVector3D lt(l.get_x_coord()- x[i], l.get_y_coord() - y[i], l.get_z_coord() -
                z_a[i]);
249         cosinus[i] = (norm.get_x_coord()*lt.get_x_coord() +
                norm.get_y_coord()*lt.get_y_coord() + norm.get_z_coord()*lt.get_z_coord()) /
250                 (norm.get_len() * lt.get_len());
251
252         rc[i] = Light::check_rgb_color((int)(180*cosinus[i] + qRed(color)));
253         gc[i] = Light::check_rgb_color((int)(180*cosinus[i] + qGreen(color)));
254         bc[i] = Light::check_rgb_color((int)(180*cosinus[i] + qBlue(color)));
255     }
256 }
257
258
259 void ZBuffer::resize_buffer(unsigned int cur_z_buffer_size) {
260     delete[] m_buffer;
261     m_buffer = new double[cur_z_buffer_size];
262     for(unsigned int i = 0; i < cur_z_buffer_size; i++) {
263         m_buffer[i] = -1000000;
264     }
265 }
266
267
268 void ZBuffer::restart_buffer()
269 {
270     for(unsigned int i = 0; i < z_buffer_size; i++) {
271         m_buffer[i] = -1000000;
272     }
273 }
```

Вывод

В данном разделе были рассмотренны средства реализации программного обеспечения и листинги исходных кодов программного обеспечения.

4 Исследовательская часть

В данном разделе будут приведены результаты работы разработанного программного обеспечения и поставлен эксперимент по сравнению времени, затраченного на рендеринг одного кадра во время симуляции водной поверхности при различных размерах исходной сетки, разном количестве полигонов и накладываемых волн.

4.1 Результаты работы программного обеспечения

На рисунке 4.1 приведён кадр симуляции волны по сетке размером 600x600, с амплитудой равной 40, скоростью 20, длиной волны 50, кривизной 0.02, вектором направления (1,1), количеством полигонов 10000, а также с двумя наложенными волнами, имеющими направление (1,0) и (0,1), а также продемонстрирован интерфейс программы.

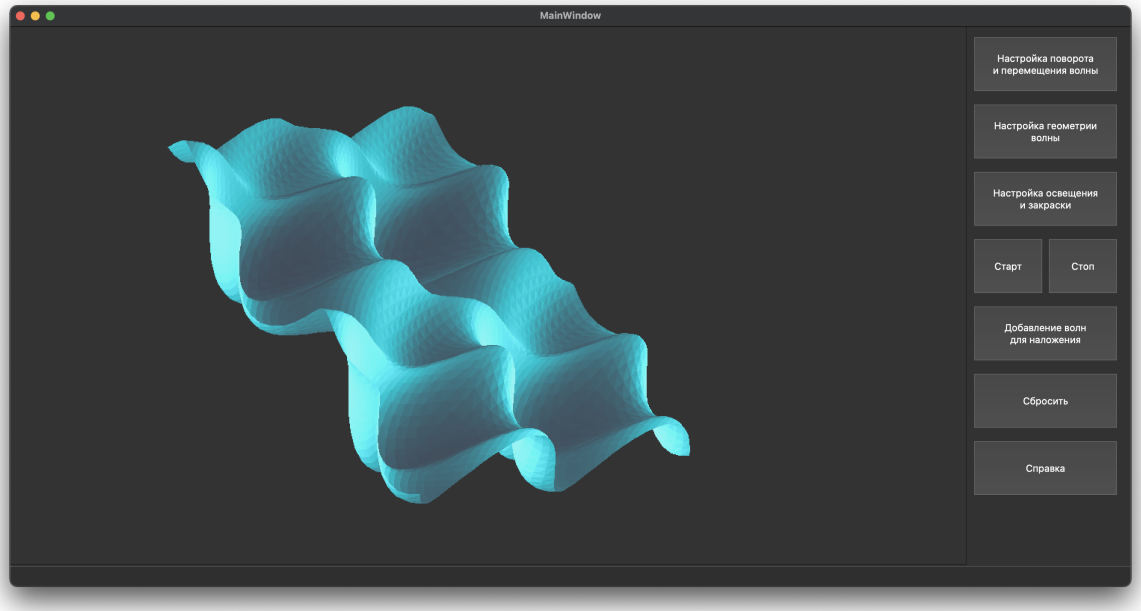

Рисунок 4.1

На рисунке 4.2 приведён кадр симуляции волны по сетке размером 300x300, с амплитудой равной 35, скоростью 25, длиной волны 100, кривизной 0.003, вектором направления (1,1), количеством полигонов 10000, наложение волн отсуствует.

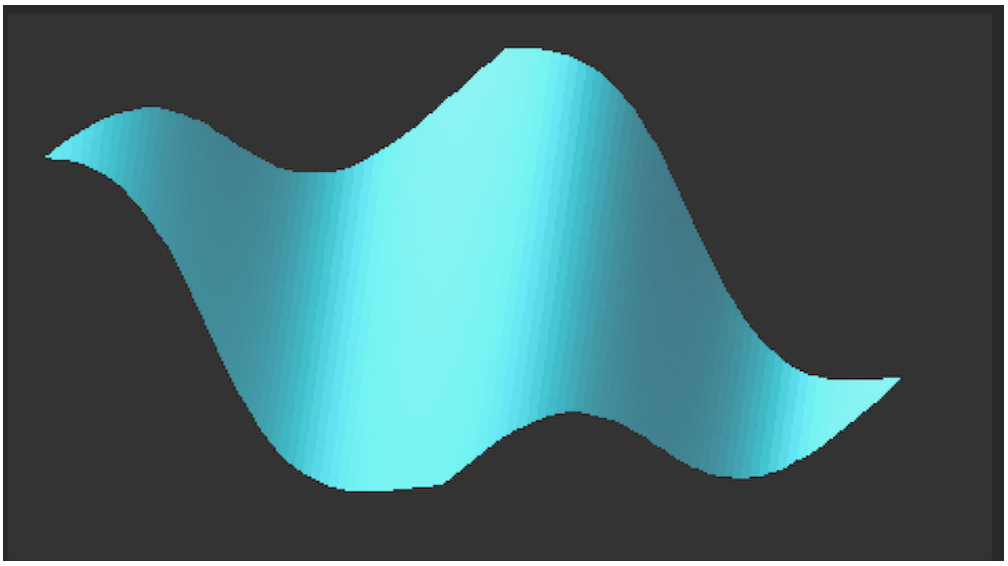

Рисунок 4.2

На рисунке 4.3 приведён кадр симуляции волны по сетке размером 300x300, с амплитудой равной 40, скоростью 8, длиной волны 100, кривизной 0.8, вектором направления (1,1), количеством полигонов 10000, а также с двумя наложенными волнами, имеющими направление (1,0) и (-1,2).

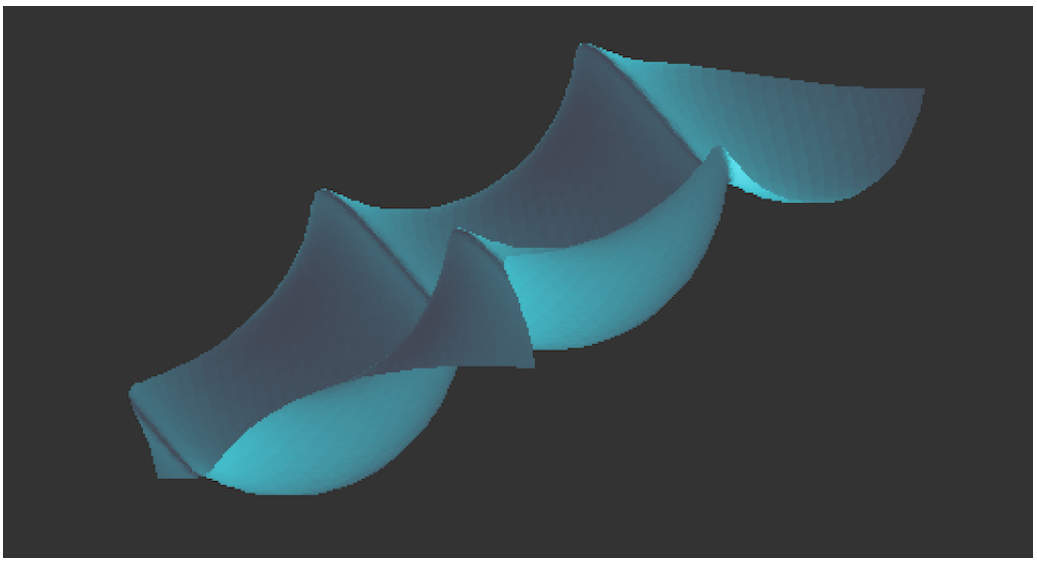

Рисунок 4.3

На рисунке 4.4 приведён кадр симуляции волны по сетке размером 300x300, с амплитудой равной 35, скоростью 18, длиной волны 100, кривизной 0.28, вектором направления (1, 1), количеством полигонов 10000, а также с четыремя наложенными волнами, имеющими направление (1, 1.2), (1, 1.5), (1, 0) и (1, 0.8).

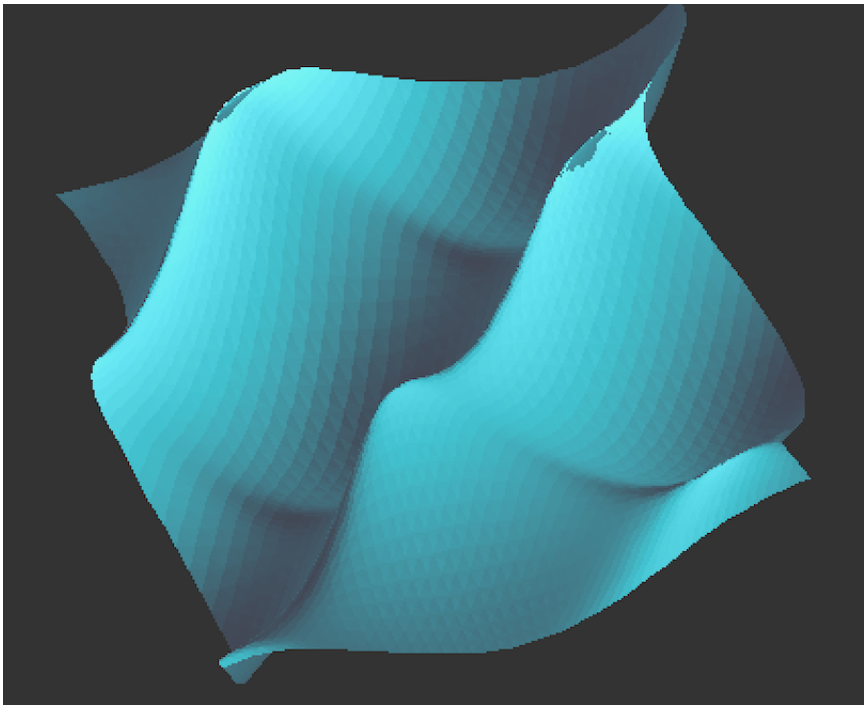

Рисунок 4.4

На рисунке 4.5 приведён кадр симуляции волны по сетке размером 600x600, с амплитудой равной 15, скоростью 20, длиной волны 50, кривизной 0.38, вектором направления (1, 1), количеством полигонов 4900, а также с девятью наложенными волнами, имеющими направление (1, 1.2), (1, 1.5), (0, 1), (-1, 2), (-1, -1), (1, 1), (1, 1), (1, 1), (-1, 1.2)б а также разные длины волн и скорости.

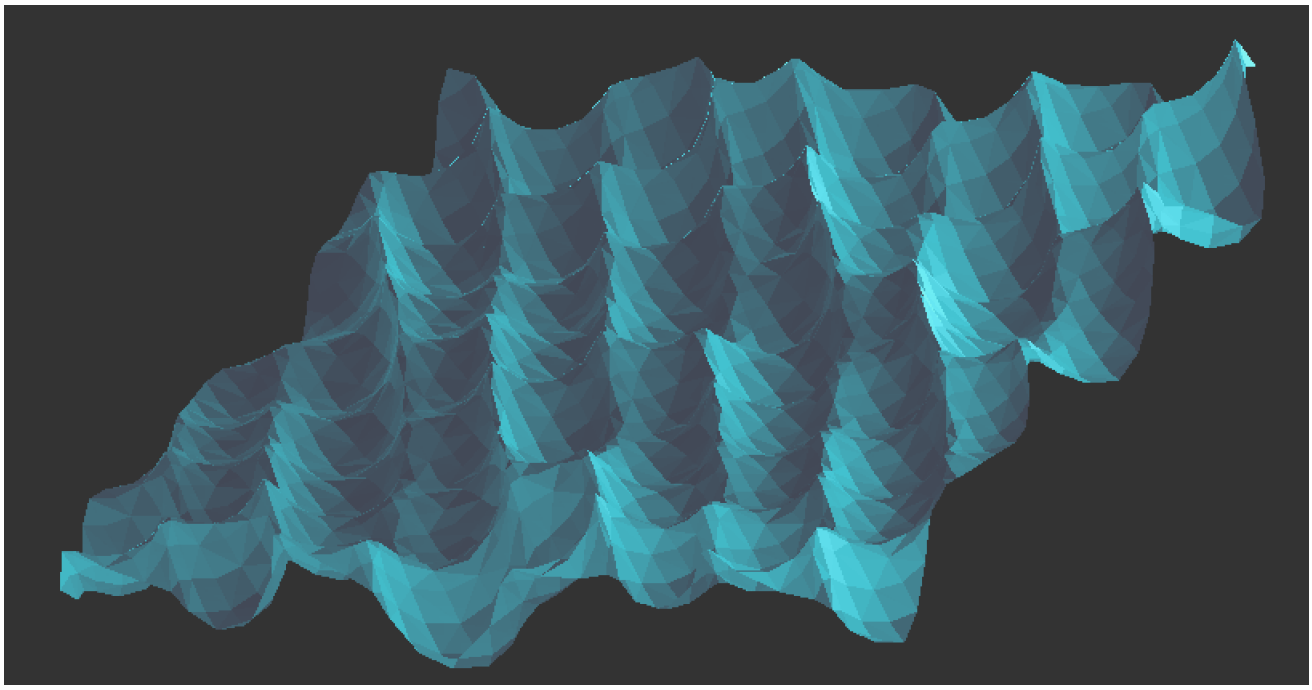

Рисунок 4.5

4.2 Постановка эксперимента

4.2.1 Цель эксперимента

Целью эксперимента является проведение трех независимых сравнений результатов работы программного обеспечения:

- сравнение времени, затраченного на рендеринг одного кадра при разных размерах исходной сетки;
- сравнение времени, затраченного на рендеринг одного кадра при разном количесве полигонов;
- сравнение времени, затраченного на рендеринг одного кадра при разном количесве накладываемых волн;

4.2.2 Сравнение времени, при разных размерах исходной сетки

Сравнить время рендеринга при разом размере базовой сетки можно, если не изменять количество полигонов, на которое она разбивается, а также количество накладываемых волн. Взятое количество полигонов равно 10000 и количество накладываемых волн 3.

Результаты сравнения времени рендеринга кадра для изображения водной поверхности при разных размерах начальной сетки приведены в таблице 4.1.

Таблица 4.1

Размер сетки	Среднее время рендеринга кадра(в тиках)
100x100	42782
200x200	43284
300x300	45594
400x400	47417
500x500	48903
600x600	50219
700x700	53101
800x800	56732

4.2.3 Сравнение времени, при разном количестве полигонов

Сравнить время рендеринга при разом количестве полигонов можно, если не изменять размер базовой сетки, а также количество накладываемых волн. Взятый размер стеки равен 400x400, количество накладываемых волн 3.

Результаты сравнения времени рендеринга кадра для изображения водной поверхности при разном количестве полигонов приведены в таблице 4.2.

Таблица 4.2

Количество полигонов	Среднее время рендеринга кадра(в тиках)
100	14775
400	15180
900	17946
1200	20605
1600	21182
2500	23760
3600	25367
4900	28089
6400	34606
8100	39220
10000	47151
12100	55233
14400	65147
16000	74444
19600	86878
22500	95676

4.2.4 Сравнение времени, при разном количестве накладываемых волн

Сравнить время рендеринга при разом количестве накладываемых волн можно, если не изменять размер базовой сетки, а также количество полигонов. Взятый размер стеки равен 400x400, количество полигонов 10000.

Результаты сравнения времени рендеринга кадра для изображения водной поверхности при разном количестве накладываемых волн приведены в таблице 4.3.

Таблица 4.3

Количество накладываемых волн	Среднее время рендеринга кадра(в тиках)
0	39753
1	41244
2	41453
3	44749
4	47229
5	47883
6	50893
7	52283
8	55608
9	57292
10	60908
11	61400
12	65104

Вывод

В данном разделе были рассмотрены примеры работы программного обеспечения и было сравнено время рендеринга при разном количестве полигонов, размере базовой сетки, количестве накладываемых волн. В результате сравнения были получены следующие результаты:

- время рендеринга увеличивается с увелечением размера базовой секи
- время рендеринга увеличиваетсяс увелечение количества полигонов
- время рендеринга увеличивается с увелечением количества накладываемых волн

Как и ожидалось, с увелечением размера поверхности, количства полигонов и с усложнением волн(их накладывании) увеличивается время отрисовки.

Заключение

Во время выполнения курсового проекта было реализованно программное обеспечение, которое позволяет симулировать поведение водной поверхности морей и океанов в реальном времени с использованием мощностей одного CPU.

Были проанализированы и рассмотренны существующие алгоритмы генерации водной поверхности. В качестве подходящего алгоритма, был выбран алгоритм волн Герстнера с импользованием сеточной модели построения водной поверхности. Для реалистичного построения изображения был выбран алгоритм удаления невидемых линий z-buffer, а также закраска методом Гуро. Данный выбор был сделан на основе анализа основных принципов построения реалистичного изображеия водной поверхности, которые позволили выбрать наиболее подходящее решение для поставленной задачи.

В процессе выполения данной работы были выполнены следующие задачи:

- проанализированы существующие алгоритмы моделирования волн и выбраны подходящие для выполнения проекта
- проанализированы существующие алгоритмы освещения, позволяющие реализовать правдоподобную закраску с тенями, и выбраны подходящие для выполнения проекта
- реализован интерфейс программы

В процессе исследовательской работы было выяснено, что повышение детализации волн повышает вычислительную сложность и время отрисовки одного кадра.

Литература

[1] A. M. Balk, A Lagrangian for water waves. 1996.

[2] GPU Gems. Chapter 18. Using Vertex Texture Displacement for Realistic Water Rendering [Электронный ресурс]. 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).

[3] GPU Gems. Chapter 1. Effective Water Simulation from Physical Models [Электронный ресурс]. Режим доступа: https://developer.download.nvidia.com/books/HTML/gpugems/gpugems_ch01.html (дата обращения: 13.08.2020).

[4] D. F. Rogers. Procedural Elements for Computer Graphics. 2nd ed. стр. 250-290. 1998.

[5] D. F. Rogers. Procedural Elements for Computer Graphics. 2nd ed. стр. 290-315. 1998.

[6] D. F. Rogers. Procedural Elements for Computer Graphics. 2nd ed. стр. 321-329. 1998.

[7] D. F. Rogers. Procedural Elements for Computer Graphics. 2nd ed. стр. 391. 1998.

[8] Lambertian Model [Электронный ресурс]. Режим доступа: https://www.sciencedirect.com/topics/computer-science/lambertian-model (дата обращения: 25.01.2021).

[9] D. F. Rogers. Procedural Elements for Computer Graphics. 2nd ed. стр. 394. 1998.

[10] ISO. ISO/IEC 14882:2020 Programming languages — C++.

[11] WHY Qt? [электронный ресурс]. Режим доступа: https://www.qt.io/why-qt (дата обращения: 05.02.2021).

[12] Qt Creator [электронный ресурс]. Режим доступа: https://www.qt.io/product/development-tools (дата обращения: 05.02.2021).