Санкт–Петербургский государственный университет

Гусев Данила Андреевич

Выпускная квалификационная работа

Создание эффективных алгоритмов морской гидродинамики с использованием OpenGL.

Уровень образования: бакалавриат
Направление 02.03.02
«Программирование и информационные технологии»
Основная образовательная программа СВ.5002.2016
«Фундаментальная информатика и информационные технологии»
Профиль «Атоматизация научных исследований»

Научный руководитель:
профессор, кафедра компьютерного моделирования и
многопроцессорных систем, д.т.н. Дегтярев Александр Борисович

Рецензент:
эксперт, общество с ограниченной ответственностью
"Т-Системс Рус"Типикин Юрий Александрович

Санкт-Петербург
2020 г.

Содержание

Введение . . . 3

Постановка задачи . . . 5

Обзор литературы . . . 6

Глава 1. Описание OpenGL . . . 8

1.1. Графический конвеер OpenGL . . . 8

1.2. Дополнительные библиотеки . . . 10

1.3. Описание GLSL . . . 10

Глава 2. Реализация алгоритма на основе метода конечных объемов . . . 13

2.1. Описание алгоритма SIMPLE . . . 13

2.2. Численная схема для алгоритма SIMPLE . . . 14

2.3. Описание реализации . . . 16

Глава 3. Реализация алгоритма на основе метода сглаженных частиц . . . 19

3.1. Описание алгоритма SPH . . . 19

3.2. Численная схема для алгоритма SPH . . . 20

3.3. Описание реализации . . . 21

Выводы . . . 25

Заключение . . . 26

Список литературы . . . 27

Приложение А . . . 29

Приложение Б . . . 32

Введение

Производительность вычислительных устройств растет с каждым годом. Программное обеспечение тоже нуждается в регулярных обновлениях для повышения эффективности использования новых вычислительных ресурсов. В процессе разработки программист или исследователь должен иметь доступ к использованию тех технологий которые используются в приложении. Это приводит к тому что десктопные системы обладают достаточно высокими показателями вычислительной эффективности или хотя бы могут эмулировать необходимые интерфейсы.

При подготовке численного эксперимента важно иметь возможность отслеживать качество и корректность работы каждого составляющего элемента, алгоритма и компоненты. Определение правильности решения некоторых частей задачи может значительно упростится если при подготовке эксперимента была подготовлена или адаптирована одна из существующих систем визуализации, кроме того грамотная визуализация результатов эксперимента может увеличить количество и понятность полученных данных.

Одним из самых распространенных стандартов, которые используются в научной среде является OpenGl. На его основе создано огромное количество средств по представлению графики. Он поддерживаетя на linux, windows и macOS, также существует программная реализация данного стандарта Messa, что позволяет назвать его одним из самых кроссплатформенных в мире.

GPGPU на данный момент крайне популярная концепция. Для ее реализации созданы стандарты и фреймворки такие как CUDA или OзenCL. С версии 4.6 OpenGL тоже стал обладать механизмом для проведения расчетов общего назначения. Если к существующей реализации фреймворка по визуализации эксперимента добавить ускорение расчетов с использованием той же памяти видеокарты которая уже используется при отрисовки, то можно получить значительный прирост производительности.

Кроме того быстро развивается стандарт WebGL обеспечивающий поддержку механизмов OpenGL для web технологий. WebGL в данный момент уже поддерживает возможность запуска шейдеров общего назначения

на некоторых платформах. После реализации этого стандарта большинством современных браузеров, системы ускоренные с помощью вычислительных шейдеров общего назначения могут стать одними из самых легко распространяемых.

Постановка задачи

Целью данной работы является повысить эффективность расчетов при помощи современных методов OpenGL по созданию высокопроизводительных программ связанных с нахождением численных решений в задачах морской гидродинамики. Для решения поставленной задачи предлагается провести оптимизацию 2-х алгоритмов на основе различных подходов: метода сглаженных частиц и метода конечных объемов.

Решение данной задачи требует реализации следующих этапов:

1. Обзор способов проведения расчетов общего назначения с использованием OpenGL;
2. Реализация алгоритма на основе метода коначеного объема;
3. Реализация алгоритма на основе метода гладких частиц.

Обзор литературы

В данной работе рассматривается технология OpenGL [1] и возможность ускорения прямых численных экспериментов с ее помощью. До явного выделения методики GPGPU для использования графчических ускорителей приходилось встраивать процесс моделирования в графический конвеер [2]. OpenGL является достаточно старой технологией и обладает множеством дополнительных библиотек [3], [4], [5], [6]. Шейдеры являются аналогами ядра из других библиотек для GPGPU и на основе них будет происходить ускорени алгоритмов [7]. И начиная с версии OpenGL 4.6 появился удобный интерфейс для проведения рассчетов общего назначения запуская шейдеры независмо от графического конвеера. Более подробно об OpenGL расписано в главе 1.

В вычислительной гидродинамики часто применяются методы прямого численного моделирования. При этом возникает множество однотипных задач, которые можно эффективно решать с помощью паралельного программироавния или GPGPU. Также каждую итерацию алгоритма можно ассоциировать с кадром отрисовки, благодаря чему легко добавить визуализацию результатов в динамике. Алгоритмов и мтеодов моделирования существует ограмное количество, но в данной работе рассмотрено только 2.

Подход на базе метода конечных объемов подразумевает разбить область через которую идет поток жидкости на маленькие участки, предпалогая что внутри области характиристики жидкости одинаковы и она может взаимодйествовать только с областями имеющие общую грань [8], [9].

Другой подход заключается в представление потока в виде набора частиц каждая из которых описывает жидкость рядом с собой. Поток при таком представление является прмым чичленным моделированием поведения этих частиц. [10], [11]

Описанные выше подходы были выбраны в сиду того, что они являются базовыми для более сложных подходов или для них существует множество модификаций [12], [13], и при этом подзоды заложенные в них

сильно отличаются.

Глава 1. Описание OpenGL

OpenGL (open graphics library) - открытый стандарт программного интерфейса для эффективиной реализации программ связанных с компьютерной графикой. Для данного стандарта существует аппаратная поддержка у большинства современных представителей десктопных систем.[1] Данный интерфейс позволяет скрыть упростить работу с графическими ускорителями и скрыть детали реализации, предствив таким образом возможность меньше зависеть от вендора. С другой стороны производители оборудования при реализации этого стандарта в праве добавлять платформа зависимые функции(так например компания NVIDIA добавляет к подобным расширениям аббревиатуру NV как в функции glCombinerParameterfvNV()). Если расширение поддержано несколькими производителями оборудования то оно получает аббревиатуру EXT.

OpenGl с программной точки зрения низкорувнеый набор функций и работа с этой библиотекой представляет из себя последовательный вызов процедур в отличие от других библиотек позволяющих создавать пользовательский интерфейс таких как qt [14].

1.1 Графический конвеер OpenGL

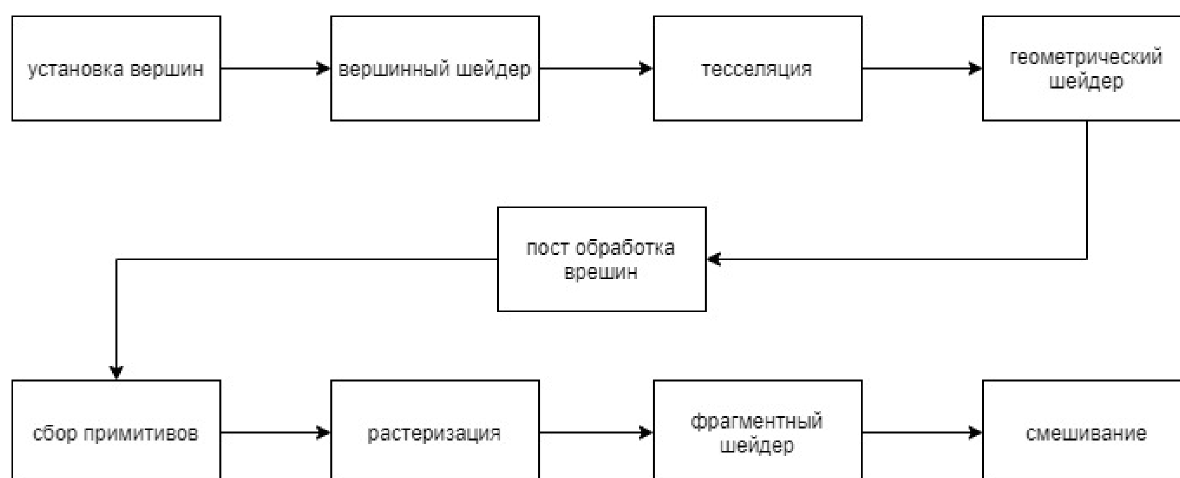

Рис. 1: Графический конвеер OpenGL

Вызов функции OpenGL либо добавляет набор вершин на сцену или

же модифицирует графический конвеер (см рис 1) - последовательность действий при построении одного кадра в процессе работы OpenGL.[2] Часть действий являются запусками различных шедеров - программ, которые выполнят специальную логику на графическом процессоре. Конвеер разделяют на следующие этапы:

1. Установка вершин - (обязательный) пользовательская программа подает список вершин для отрисовки;

2. Вершинный шейдер - (не обязательный) возможно модифицировать пришедшие вершины;

3. Теселяция - (не обязательный) могут быть добавлены дополнительные вершины для уточнения формы отрисовывющися объектов;

4. Геометрический шейдер - (не обязательный этап) возможность удалять вершины или разбивать их на несколько на основе геометрических примитивов;

5. Постобработка вершин - (обязательный) набор встроенных методов по потсобработке вершин (к примеру если вершина принадлежит к нескольким треугольникам ее разбивают на 2);

6. Сбор примитивов - (обязательный) после фиксации всех вершин обрабатываются примитивы(к примеру удаляются не видимые треугольники);

7. Растеризация - (обязательный) набор геометрических примитивов пробразуется в матрицу пикселей;

8. Фрагментный шейдер - (не обязательный) возможность модифичировать отдельно цвет каждого пикселя;

9. Смешивание - (обязательный) обновление экрана.

1.2 Дополнительные библиотеки

Помимо основного стандарта существует множество дополнительных библиотек. Рассмотрим некоторые из них:

1. GLEW (OpenGL Extension Wrangler Library) - кроссплатформенная библиотека написанная на базе с использованием языка C/C++. Ее применение позволяет упростить взаимодействие с расширениями которые представлены вендорами графического оборудования; [3]

2. GLM (OpenGL Mathematics) - библиотека для OpenGL реализованная на C/C++ состоящая только из заголовочных файлов. Данная библиотека сожержит структуры данных и функции использующиеся в OpenGL при проведении математических расчетов по обработке сцены; [4]

3. GLUT (OpenGL Utility Toolkit) - кросплатформенная библиотека утилит, содержащая функции для работы с окнами операционной системы и несколько дополнительных графических примитивов; [5]

4. GLFW - так же как и GLUT эта библиотека предоставляет возможность создавать и управлять окнами, но позволяет использовать не только OpenGl, но и Vulkan. [6]

1.3 Описание GLSL

GLSL (OpenGL shading language) - язык для программирования шейдеров использующейся в OpenGL. [7] По мимо описанных выше шейдеров есть отдельные тип предназначенный для вычислений.

Компиляция шейдеров происходит в 2 этапа:

1. Исходный код на GLSL с помощью компилятора glslangValidator переводится в промежуточный язык SPIR-V;

2. Программа на SPIR-V компилируется в исполняемый шейдер с учетом окружения в процессе инициализации зависимостей программы.

Запуск вычислительного шейдера происходит с помощью команды glDispatchCompute() - принимает количество групп по каждой из осей которых нужно запустить. Внутри группы будет запущено некоторое количество нитей. Точное их количество можно задать с помощью команды помощью команды "layout (local_size_x = 32) in;". Это определяется в процессе первого этапа компиляции и если оборудование не пооддерживает такое количество нитей с которым шейдер был переведен в SPIR-V, то он не будет скомпилирован на втором этапе.

Каждая группа нитей может выделить небольшую разделяемую память с помощью модификатора shared. Внутри группы существует ряд функций позволяющих упорядочить доступ к разделяемой памяти и обеспечивающий синхронизацию нитей.

При написании кода вычислительного шейдера доступен ряд переменных связанных с индкесом текущей нити:

1. gl_WorkGroupSize - заданный размер группы по каждой размерности;
2. gl_WorkGroupID - координата текщей группы;
3. gl_LocalInvocationID - координата нити внутри группы;
4. gl_GlobalInvocationID - уникальная координата нити, эквивалентна gl_WorkGroupID*gl_WorkGroupSize+gl_LocalInvocationID;
5. gl_LocalInvocationIndex - позволяет получить индекс нити, а не ее трех-мерную координату.

Передача данных на вычислительный шейдер происходит в несколько этапов:

1. Создать идентификатор для буффера памяти - glGenBuffers;
2. Прикрепить к идентификатору не инициализированный буффер - glBindBufferRange, при этом требуется указать его тип, необходимо отметить GL_SHADER_STORAGE_BUFFER;

3. Подготовить данные и скопировать их при инициализации буффера через void* - glBufferStorage;

4. Разметить буффер - glBindBufferRange, требуется указать начало и конец участка и индекс по которому этот участок будет доступен из шейдера.

Для доступа к выделенному буфферу и получения результата вычислений необходимо воспользоваться функцией glGetBufferSubData.

Для синхронизации буферов данных между выполнениями различных шейдеров используется команда glMemoryBarrier с специальным параметром GL_SHADER_STORAGE_BARRIER_BIT.

Внутри шейдера можно получить доступ к блоку памяти следующим образом:

```
layout(std430, binding = индекс_буффера) buffer имя_блока
{
flaot value[ ];
};
```

Для демонстрации работы вычислительных шейдеров в данной работе рассмотрено 2 алгоритма моделирования жидкости: метод сглаженных частиц(SPH) и алгоритм с фиксированной расчетной сеткой(SIMPLE).

Глава 2. Реализация алгоритма на основе метода конечных объемов

2.1 Описание алгоритма SIMPLE

Для описания несжимающихся течений часто описываются системой уравнений Навье-Стокса [15]:

$$\begin{cases} \rho \bigtriangledown \cdot u = 0 \\ \frac{\partial u}{\partial t} = -(u \cdot \bigtriangledown)u - \frac{1}{\rho} \bigtriangledown p + \nu \bigtriangledown^2 u + f \end{cases} \tag{1}$$

Где u - вектор скорости течения, ρ - плотность, ν - коэффициент вязкости, f - внешняя сила. В нашем случае будем рассматривать 2-мерный случай данного уравнения без воздействия внешней силы. В таком случае уравнение 1 переписывается в виде 2.

$$\begin{cases} \rho \bigtriangledown \cdot u^{n+1} = 0 \\ \frac{u^{n+1} - u^n}{dt} = -(u^n \cdot \bigtriangledown)u^n - \frac{1}{\rho} \bigtriangledown p^{n+1} + \nu \bigtriangledown^2 u^n \end{cases} \tag{2}$$

На первом шаге алгоритма рассматривается уравнение количества движения из системы 2 в котором u^{n+1} заменяется на $u^\star$ и мы находим приближение скорости в момент времени $n+1$ 3.

$$\frac{u^\star - u^n}{dt} = -(u^n \cdot \bigtriangledown)u^n - \frac{1}{\rho} \bigtriangledown p^n + \nu \bigtriangledown^2 u^n \tag{3}$$

Вычетая из 2 уравнение 3 и положив $p' = p^{n+1} - p^n$ и $u' = u^{n+1} - u^\star$ получаем:

$$\frac{1}{dt} u' = -\frac{1}{\rho} \bigtriangledown p' \tag{4}$$

При этом подстановка уравнения 4 в уравнение не разрывности из неравенства 2:

$$\rho \bigtriangledown \cdot u^\star = \bigtriangledown \cdot (dt \bigtriangledown p') \tag{5}$$

Одна итерация алгоритма SIMPLE состоит из следующих шагов [16]:

1. Рассчитать приближение скорости $u^\star$ с помощью уравнения 3;

2. Определить приращение давления p' из уравнения 5;

3. Определить корректировку скорости u' из уравнения 4;

4. Положить $p^{n+1} = p^n + p'$ и $u^{n+1} = u^\star + u'$.

2.2 Численная схема для алгоритма SIMPLE

Разобьем область в которой проходит эксперимент на множество малых прямоугольников образующих матрицу, где dx - расстояние между ячейками по x и dy - расстояние между ячейками по y. Если В центре каждого объема определить значение скорости $u_{i,j}^n$ и давления $p_{i,j}^n$ то уравнение неразрывности будет иметь следующий вид:

$$((u_x)_{i+1,j}^n - (u_x)_{i-1,j}^n) * dy + ((u_y)_{i,j+1}^n - (u_y)_{i,j-1}^n) * dx = 0$$

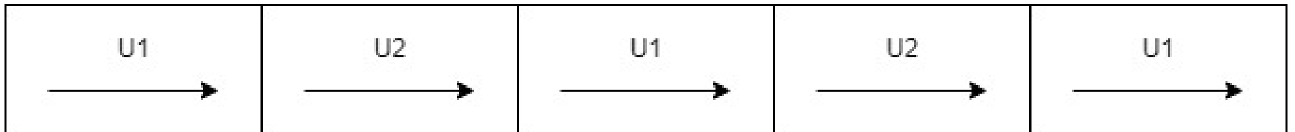

Рис. 2: Пример не правильного хранения скорости

Не сложно заметить что данное уравнение связывает ячейки в шахматном порядке из-за чего возможно что данная схема выполняется для полей которые не обладают свойством не разрывности (см рис 2). В следствии предлагается использовать следующее представление поля: давление по прежнему описывается в центре контрольного объема, а скорость хранить по компонентно: на границах перпендикулярных оси x ханить скорость вдоль оси x $w_{i,j}^n$, на границах перпендикулярных оси y хранить скорость вдоль оси y $v_{i,j}^n$(см рис 3). В таком случае уравнение не разрывности будет описано следующим образом:

$$(w_{i,j}^n - w_{i-1,j}^n) * dy + (v_{i,j}^n - v_{i,j-1}^n) * dx = 0$$

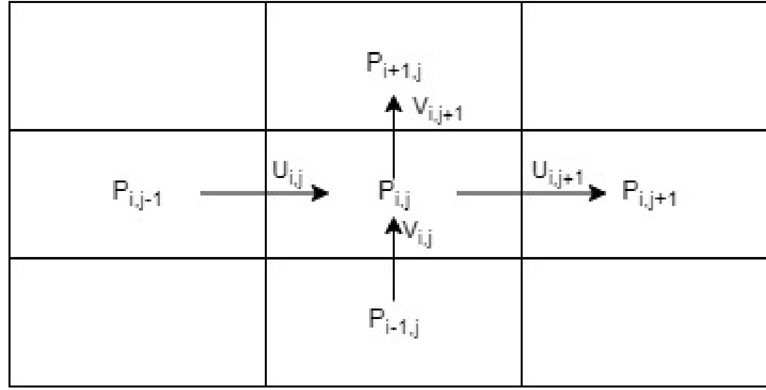

Рис. 3: Пример правильного хранения скорости

В следствии чего описанная выше проблема решена. [17] При такой схеме уравнение неразраыности решается относительно центров ячеек, однако уравнение количества движения необходимо решать независимо для координаты x и y относительно границ ячеек 6.

Перепишем уравнение 3 для описанной выше схемы описания поля значений:

$$\begin{cases} \frac{w^{\star}_{i,j}-w^n_{i,j}}{dt} = -\frac{\partial v^n_{i,j} * v^n_{i,j}}{\partial x} - \frac{\partial w^n_{i,j} * v^n_{i,j}}{\partial y} - \frac{1}{\rho}\frac{\partial p^n_{i,j}}{\partial x} + \nu(\frac{\partial^2 w^n_{i,j}}{\partial x^2} + \frac{\partial^2 w^n_{i,j}}{\partial y^2}) \\ \frac{v^{\star}_{i,j}-v^n_{i,j}}{dt} = -\frac{\partial v^n_{i,j} * w^n_{i,j}}{\partial x} - \frac{\partial v^n_{i,j} * v^n_{i,j}}{\partial y} - \frac{1}{\rho}\frac{\partial p^n_{i,j}}{\partial y} + \nu(\frac{\partial^2 v^n_{i,j}}{\partial x^2} + \frac{\partial^2 v^n_{i,j}}{\partial y^2}) \end{cases} \tag{6}$$

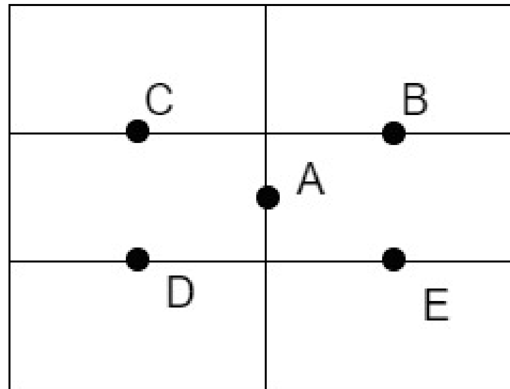

Рис. 4: Схема положения для значений скорости по компоненте y

Для решения уравнения 6 требуется определить значение скорости вдоль y в точке A смотри схему 4. Скорость в данной точке предполагается

равной среднему значению скорости в точках B, C, D и E 7. Аналогично для скорости вдоль оси x 8.

$$v_{i,j}^{mid} = \frac{v_{i,j}^n + v_{i+1,j}^n + v_{i,j+1}^n + v_{i+1,j+1}^n}{4} \tag{7}$$

$$w_{i,j}^{mid} = \frac{w_{i,j}^n + w_{i+1,j}^n + w_{i,j+1}^n + w_{i+1,j+1}^n}{4} \tag{8}$$

Перепишем уравнение 5 в следующем виде:

$$\frac{\rho}{dt}(\frac{(w_{i,j}^n - w_{i-1,j}^n)}{dx} + \frac{(v_{i,j}^n - v_{i,j-1}^n)}{dy}) = \bigtriangledown \cdot (\bigtriangledown p') = c_{i,j}$$

Это уравнение представляет из себя систему линейных алгебраических уравнений. Нахождение ее решения будет происходить с помощью метода Якоби. Выпишем переход для этого метода с переходом:

$$p_{i,j}^{\prime k+1} = \frac{(p_{i+1,j}^{\prime k} + p_{i-1,j}^{\prime k}) * dy + (p_{i,j+1}^{\prime k} + p_{i,j-1}^{\prime k}) * dx - c_{i,j} * dx * dy}{2(dx + dy)}$$

Корректировка сокрсоти находится следующим образом:

$$\begin{cases} w_{i,j}' = -\frac{dt}{\rho}(\frac{p_{i,j}' - p_{i-1,j}'}{dx}) \\ v_{i,j}' = -\frac{dt}{\rho}(\frac{p_{i,j}' - p_{i,j-1}'}{dy}) \end{cases}$$

2.3 Описание реализации

Анализ проводился на машине с видеокартой NVIDIA Geforce GTX 960m, процессором Inte(R) Core(TM) i7-6700HQ CPU @ 2.60GHz и опирационной системой windows 10. Исходный код шейдеров представлен в приложении А.

Для упрощения рассчетов предполагалось что смешение по x и по y равны, А расчеты происходят на квадратной сетке.

Так как компненты скорости храняться на границах сетки то для их хранения требуется массивы размерностью k на $k+1$ для x и $k+1$ на k для y. Для упрощения перидачи данных матрица описывающие поле скоростей

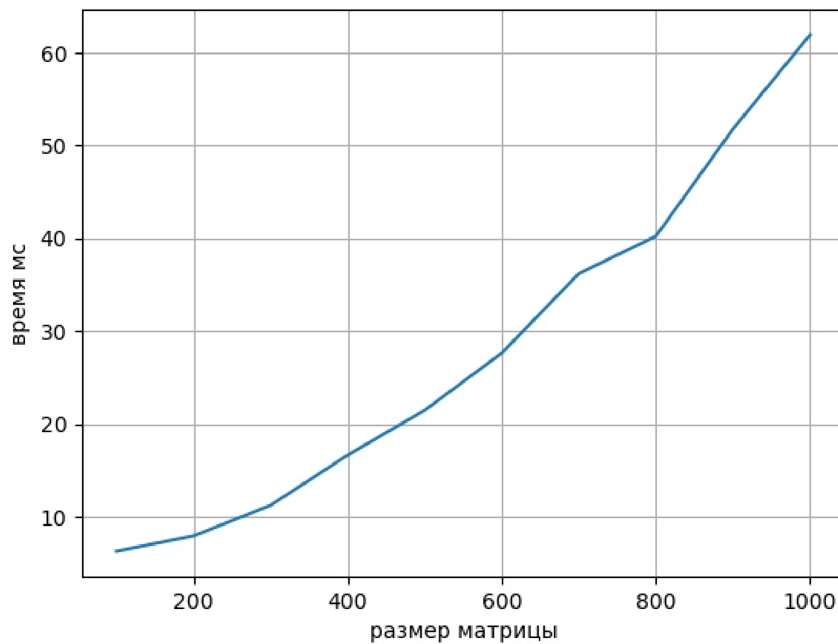

Рис. 5: Зависисмость времени работы алгоритма simple от частоты сетки

вдоль y транспонировалось и была объединена с матрицей описывающей поле скоростей вдоль компоненты x.

Для обеспечения граничных условий рассчет проводился только для внитрених элементов матрицы. Если элемен находился на расстоянии не менне 2 до границы то он не менялся и в расчетах выступал в виде граничного условия.

На графике 5 показано завсисмость работы алгоритма от частоты сетки на которые была разбита рассматриваемая область. Ускорение алгоритма показано на графике 6. Фактическое значение будет несколько меньше так как алгоритм с которым было проведено сравнение реализавон не самым оптмальным образом. Этот результат сходится с аналогичным из работы [9]. Ускорение данного алгоритма не слишком высоко из-за того что на первой итерации алгоритма необходимо мног раз обращаться к далеко расположенным данным. Из-за чего их не возможно сохранить в кэш для быстрого доступа. Кроме того процесс определения схождения метода Якоби реализавон на стороне центральног процессора. Если вместо проверки насстояния между итерациями каждый раз выполнять фиксированное

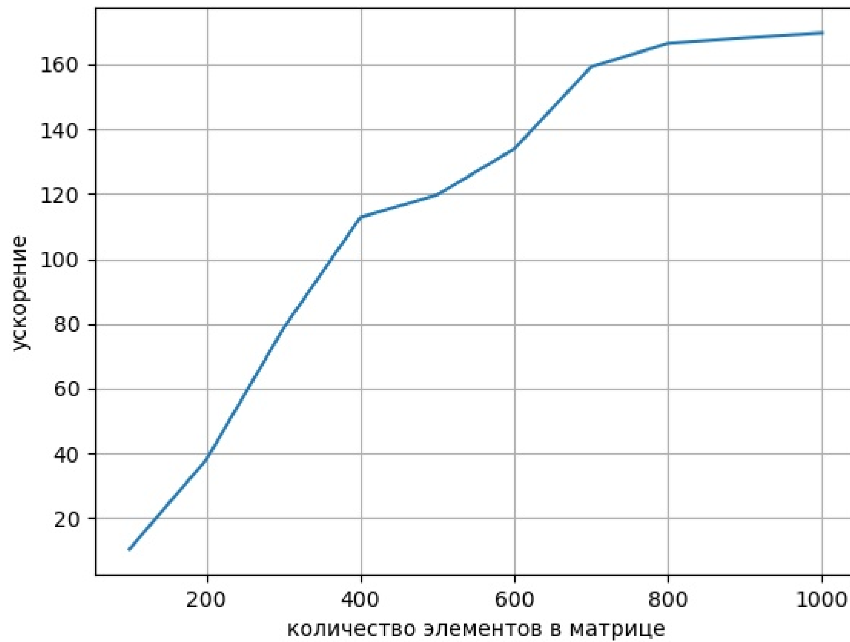

Рис. 6: Зависисмость ускорения алгоритма simple от частоты сетки

количество операций, то можно увеличить производительность почти в 2 раза.

Итого для алгоритма simple не получилось провести значительное ускорение, как одна из причин нет возможности эффективно определить сходимость метода Якоби при решении СЛАУ отнсительно приращения давления, которая является самой долго опреацией в данном алгоритме.

Глава 3. Реализация алгоритма на основе метода сглаженных частиц

3.1 Описание алгоритма SPH

Метод сглаженных частиц заключается в том что жидкость представляется в виде множества частиц. Данный алгоритм применим в случаях сложной или сильно меняющейся в ходе эксперимента геометрии. [11] Для того чтобы определить характеристики жидкости на всем объеме, когда известны их значения только в конкретных точках вводятся область $\Omega(r) = x : |r - x| < h (h$ - длина сглаживания) и значение данной величины аппроксимируются как:

$$f^{\star}(r) = \int_{\Omega(r)} f(x) W(|x - r|, h) dx \tag{9}$$

где W - функция-ядро. Данная функция должна удовлетворять следующим условиям:

$$\int_{\Omega(r)} W(|x - r|, h) dx = 1$$

$$\lim_{h \to 0} W(|x - r|, h) = \delta(|x - r|)$$

$$W(|x - r|, h) = 0, |x - r| > h$$

где $\delta(|x - r|)$ - дельта функция Дирака.

С каждой точкой, в которой нам известно значение параметра, соотнесен некоторый объем $V_i = \frac{m_i}{\rho_i}$. Это позволяет нам заменить интеграл в выражении 9 на сумму по всем частицам:

$$f^{\star}(r) = \sum_i \frac{m_i}{\rho_i} f(x_i) W(|x_i - r|, h) \tag{10}$$

Для определения параметров течения жидкости необходимо отслеживать положение и скорость каждой частицы. При моделеировании эти параметры можно получить путем интегрирования ускорения, которое в

свою очередь можно вывести зная действующую силу. Она представляется в виде суммы гравитации, силы давления, силы вязкости:

$$F_i = F_i^p + F_i^v + m_i * g \tag{11}$$

Сила давления определяется следующим образом [10]:

$$F_i^p = -m_i \sum_j \frac{p_i + p_j}{2\rho_j} \bigtriangledown_i W(r_i - r_j, h) \tag{12}$$

где

$$\bigtriangledown_i W(r_i - r_j, h) = \frac{\partial W(r, h)}{\partial} \frac{r_{ij}}{r}, r_{ij} = r_i - r_j, r = |r_{ij}|$$

Силу вязкости сожно выразить как [10]:

$$F_i^v = m_i \sum_j \mu(v_i + v_j) \bigtriangledown_i^2 W(r_i - r_j, h) \tag{13}$$

Если плотность подставить в выражение 10 то можно убедится что она определяется как:

$$\rho_i = \sum_j m_j W(r_i - r_j, h) \tag{14}$$

Давление можно выразить через плотсность [18]:

$$p_i = \max(0, c_0(\rho_i - \rho_o)) \tag{15}$$

где ρ - исходная плотсность, c_0 - коэффициент жесткости.

3.2 Численная схема для алгоритма SPH

Для запуска алогритма требутеся определить начальные положения частиц r_i^0 и начальные скорости v_i^0.

Одна итерация алгоритма разбивается на следующие последовательные этапы:

1. Вычисление плотности ρ_i^{n+1} с помощью уранения 14 с учетом поло-

женя r_i^n и скорости v_i^n;

2. Вычисление давления p_i^{n+1} на основании 15 используя ρ_i^{n+1};

3. Зная значение давления p_i^{n+1} и плотности ρ_i^{n+1} можно вычислить значение силы F_i^{n+1} используя уравнения 11, 12 и 13;

4. При помощи уравнений 16 итерация алгоритма заканчиватеся нахождением нового положения и скорости частицы.

$$\begin{cases} v_i^{n+1} = v_i^n + F_i^{n+1}/\rho_i^{n+1} * dt \\ r_i^{n+1} = r_i^n + v_i^{n+1} * dt \end{cases} \tag{16}$$

При интегрировании нужно проверить граничные условия, а именно не пересекает ли частица границу рассматриваемой области, при необходимости уменьшить значение и поменять направление скорости частицы.

3.3 Описание реализации

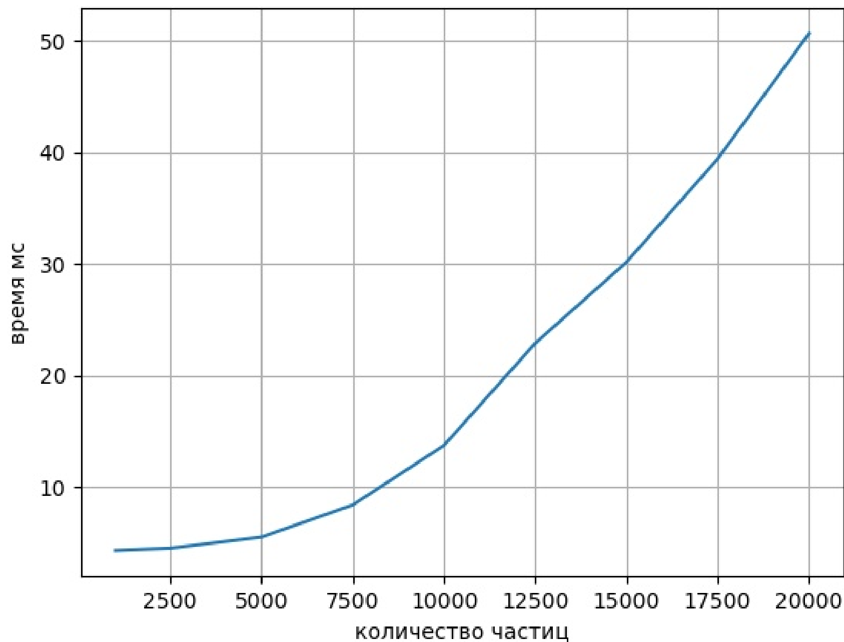

Рис. 7: Зависисмость времени работы алгоритма sph от количества частиц

Анализ проводился на машине с видеокартой NVIDIA Geforce GTX 960m, процессором Inte(R) Core(TM) i7-6700HQ CPU @ 2.60GHz и опирационной системой windows 10. Исходный код шейдеров представлне в приложении Б.

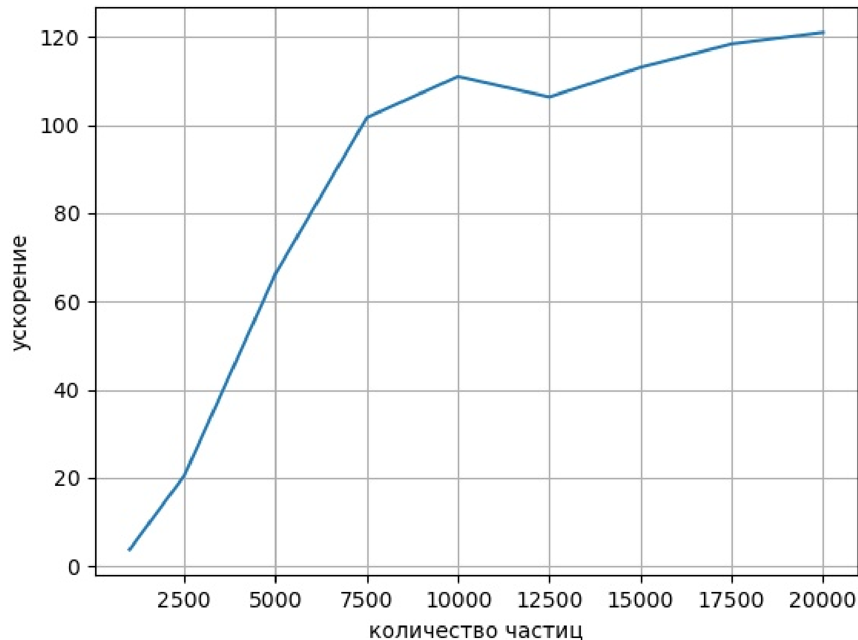

Рис. 8: Ускорение по сравнению с однопоточной реализацией

При эксперментах была стпользована следующая функция-ядро:

$$W(r,h) = \frac{315(h^2 - r^2)^3}{64\pi h^9}$$

На графике 7 показана среднее время на обработку одной итерации. На графике показана ускорение посравнению с однопоточным вариантом. Для каждого измерения проводилось 100 итераций алгоритма, после чего время усреднялось. Как видно из графика 8 при количестве вершин свыше 7500 ускорение алгоритма приблизилось к максимальному значению.

На графике 9 показана зависимость времени работы от размера группы потоков внутри шейдера, при этом все измерения проводились при 20000 верншин. Максимальная производительность алгоритма достигается при размерах группы 96, 224, 288. Для данного слкчая оптимальным яв-

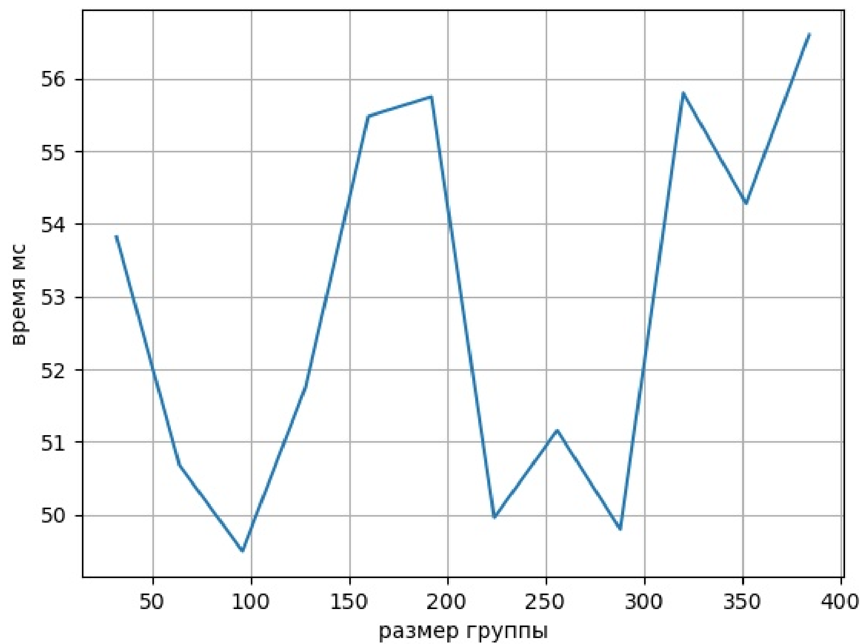

Рис. 9: Время работы алгоритма в зависисмости от размера группы потоков

ляется 96 так как группы меньшкго размера могут работать на большем количестве моделий графических ускорителей.

Самой времязатратной операцией при рассчетах является суммирование по всем объектам. [10] Для ускорение этого места было предположено что если для каждой частицы построить список тех с которыми она может взаимодействовать. Этот список обнавляется раз в несколько итераций. И предполагалась хранить для каждой вершины список смежных, расстояние между которыми в ближайщие несколько итераций может стать достаточно мало для их взаиможействия.

Оптыным путем было выяснено что для пожжрежания актуальной информации о соседях при количестве хранимых ссылок равгым 0.1 от всего количества частиц требуется обнавлять информацию не реже чем раз в 4-5 итераций.

На графике 10 показана время работы такой модификации алгоритма. Как видно время работы не улучшилось. Это связано с тем что при стандартной реализации обращения у объектам идут последовательно что ускоряет к ним доступ, в то время как в ускоренной они перемешаны. Воз-

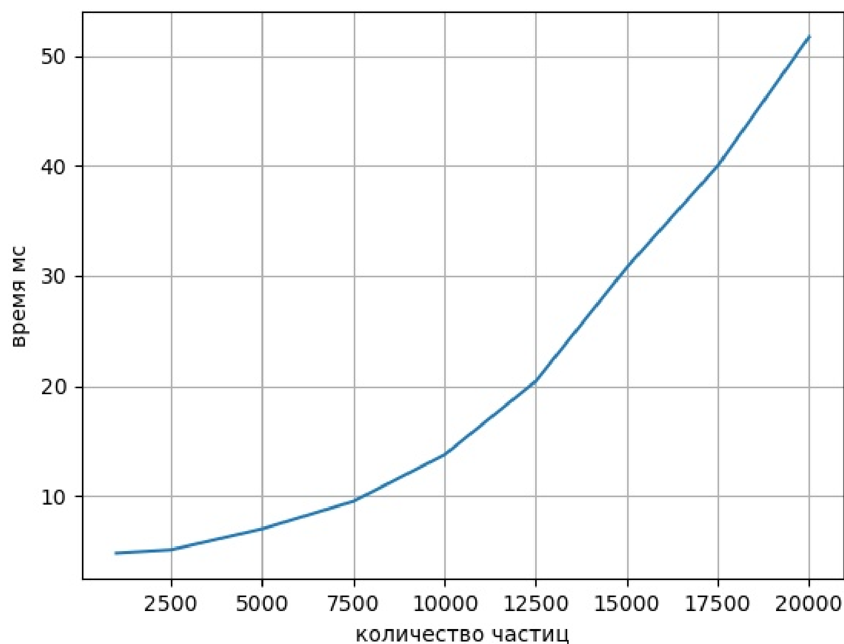

Рис. 10: Зависисмость времени работы ускоренного алгоритма sph от количества частиц

можно ускорение покажет лучшие результаты при большем количестве частиц, но данная модификация значительно увеличивает объемы памяти необходимые для работы алгоритма.

Итого алгоритм SPH получилось значительно ускорить, однако его суммарная вычислительная сложность $O(n^2)$, что превышает достижимую минимальную сложность для однопоточной версии. В следствии вего при значительном увеличении количества частиц она будет уступать верии алгоритма для центрального процессора.

Выводы

Основной целью данной работы было повысить эффективность расчетов при помощи современных методов OpenGL для создания эффективных алгоритмов гидродинамики.

Для достижения этих целей был изучен механизм работы OpenGL и вычислительных шейдеров. Способы создания таких шейдеров и особенности их запуска и механизм управления со стороны родительского процесса.

В процессе изучения работы был реализован алгоритм SIMPLE и SPH. Для этих алгоритмов было проведено сравнение с их однопоточными реализациями. Показавшее схожий коэффициент ускорения с реализациями на базе технологии CUDA и OpenCL представленных в литературе.

Заключение

В ходе работы:

1. Изучен механизм OpenGl по предоставлению возмодности проводить вычисления на видеокарте,
2. Реализован алгоритм SIMPLE,
3. Реализован алгоритм SPH,

Дальнейшие направления развития:

1. Сравнение ускорения на базе технологии OpenGl с CUDA и OpenCL
2. Создание усправляющего хост процесса на базе технолгии WebGL и проверка портируемости полученного решения.
3. Релизовать алгоритм с использованием мтаричных опреаций размером не более чем 4 элемента и рассмотреть возможность ускорения. Данные опреации являются базовыми в терминах компьютерной графики.

Список литературы

[1] Guha S. Computer graphics through OpenGL from theory to experiments. 3 изд. Boca Raton: CRC Press, 2019.

[2] Rendering Pipeline Overview // The Industry's Foundation for High Performance Graphics URL: https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview (дата обращения: 31.05.2020).

[3] The OpenGL Extension Wrangler Library URL: http://glew.sourceforge.net/ (дата обращения: 31.05.2020).

[4] OpenGL Mathematics URL: https://glm.g-truc.net/0.9.9/index.html (дата обращения: 31.05.2020).

[5] The freeglut project URL: http://freeglut.sourceforge.net/ (дата обращения: 31.05.2020).

[6] GLFW - An OpenGL library URL: https://www.glfw.org/ (дата обращения: 31.05.2020).

[7] The OpenGL® Shading Language URL: https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.40.pdf (дата обращения: 31.05.2020).

[8] Moukalled F., Darwish M.A. A unified formulation of the segregated class of algorithms for fluid flow at all speeds // Numer. Heat Tr. B-Fund. – 2000. – Vol. 37, no. 1. – P. 103-139.

[9] Д.В. Деги, А.В. Старченко Численное решение уравнений Навье Стокса на компьютерах с параллельной архитектурой // Вестник Томского государственного университета. Математика и механика. 2012. №18. С. 68-74.

[10] Суравикин А.Ю. Реализация метода SPH на CUDA для моделирования несжимаемых жидкостей // Наука и образование: научное издание МГТУ им. Н.Э. Баумана. 2012. №7.

[11] Паршиков А. Н. Численный метод SPH, использующий соотношения распада разрывов, и его применение в механике деформируемых гетерогенных сред: автореф. дис. ... канд. физ. мат. наук: 01.02.04. М., 2013.

[12] Simple and Fast Fluids. // HAL URL: https://hal.inria.fr/inria-00596050/document (дата обращения: 31.05.2020).

[13] Стояновская О.П., Снытников Н.В., Снытников В.Н. Алгоритм для решения нестационарных задач гравиатационной газовой динамики: комбинация метода SPH и сеточного метода вычисления гравитационного потенциала // вычислительные методы и программирование: новые вычислительные технологии. 2015. №16. С. 52-60.

[14] Qt | Cross=platform software development for embedded & destop URL: https://www.qt.io/ (дата обращения: 31.05.2020).

[15] Stam, J. Stable Fluids // ACM SIGGRAPH 99. 2001.

[16] Лашкин С. В., Козелков А. С., Ялозо А. В., Герасимов В. Ю., Зеленский Д. К. Исследование эффективности параллельной реализации алгоритма simple на многопроцессорных ЭВМ // Вычислительная механика сплошных сред. 2016. №3 . С. 298-315.

[17] ПАТАНКАР С. Численные методы решения задач теплообмена и динамики жидкости. М.: ЭНЕРГОАТОМИЗДАТ, 1984.

[18] Diego M., Colagrossi A. A simple procedure to improve the pressure evaluation in hydrodynamic context using the SPH // Computer Physics Communications.. 2009.

Приложение А

Шейдеры алгоритма sph. До кода объявлены необходимые окнстанты и получен доступ к буфферам: velocity_star, velocity, pressure_tmp1, pressure_tmp2, pressure_div, pressure.

Вычисление $u^{\star}$:

```
void main()
{
uint x = gl_GlobalInvocationID.x;
uint y = gl_GlobalInvocationID.y;
vec2 vel_up = (velocity[cord(y+2, x)] + velocity[cord(y+1, x)] +
 velocity[cord(y+2, x-1)] + velocity[cord(y+1, x-1)])*0.25;
vec2 vel_down = (velocity[cord(y, x)] + velocity[cord(y-1, x)] +
 velocity[cord(y, x-1)] + velocity[cord(y-1, x-1)])*0.25;
float dx_uu = (velocity[cord(y, x+1)].x*velocity[cord(y, x+1)].x -
 velocity[cord(y, x-1)].x*velocity[cord(y, x-1)].x)*0.5/distx;
float dy_vu = (velocity[cord(y+1, x)].x*vel_up.y -
velocity[cord(y-1, x)].x*vel_down.y)*0.5/disty;
float dx_vu = (vel_up.x*velocity[cord_t(y, x+1)].y -
 vel_down.x*velocity[cord_t(y, x-1)].y)*0.5/distx;
float dy_vv =
(velocity[cord_t(y+1, x)].y*velocity[cord_t(y+1, x)].y -
 velocity[cord_t(y-1, x)].y*velocity[cord_t(y-1, x)].y)*0.5/disty;
float dx_p = (pressure[cord_p(y, x)] -
pressure[cord_p(y, x-1)])/distx;
float dy_p = (pressure[cord_p(y, x)] -
pressure[cord_p(y-1, x)])/disty;
vec2 d2x_vel = (velocity[cord(y, x+1)] + velocity[cord(y, x-1)] -
 (velocity[cord(y, x)]*2))*(1./distx/distx);
vec2 d2y_vel = (velocity[cord(y+1, x)] + velocity[cord(y-1, x)] -
 (velocity[cord(y, x)]*2))*(1./disty/disty);
vec2 dt_u;
dt_u.x = mu/ro*(d2x_vel.x + d2y_vel.x) -
```

```
(dx_uu + dy_vu + dx_p/ro);
dt_u.y = mu/ro*(d2x_vel.y + d2y_vel.y) -
(dx_vu + dy_vv + dy_p/ro);
velocity_star[cord(y, x)].x = velocity[cord(y, x)].x + dt_u.x*dt;
velocity_star[cord_t(y, x)].y =
velocity[cord_t(y, x)].y + dt_u.y*dt;
}
```

Вычисление p':

```
void main()
{
uint x = gl_GlobalInvocationID.x;
uint y = gl_GlobalInvocationID.y;
pressure_div[cord_p(y, x)] = ro*(velocity_star[cord_t(y+1, x)].y -
velocity_star[cord_t(y, x)].y + velocity_star[cord(y, x+1)].x -
velocity_star[cord(y, x)].x)/dt;
pressure_tmp1[cord_p(y, x)] = 0;
pressure_tmp2[cord_p(y, x)] = 0;
}

void main()
{
uint x = gl_GlobalInvocationID.x;
uint y = gl_GlobalInvocationID.y;
pressure_tmp2[cord_p(y, x)] = (pressure_tmp1[cord_p(y+1, x)] +
pressure_tmp1[cord_p(y-1, x)] + pressure_tmp1[cord_p(y, x+1)] +
pressure_tmp1[cord_p(y, x-1)] -
pressure_div[cord_p(y, x)]*distx)*0.25;
}


void main()
```

```
{
uint x = gl_GlobalInvocationID.x;
uint y = gl_GlobalInvocationID.y;
pressure_tmp1[cord_p(y, x)] = (pressure_tmp2[cord_p(y+1, x)] +
pressure_tmp2[cord_p(y-1, x)] + pressure_tmp2[cord_p(y, x+1)] +
pressure_tmp2[cord_p(y, x-1)] -
pressure_div[cord_p(y, x)]*distx)*0.25;
}
```

Обновление значения скорости и давления:

```
void main()
{
uint x = gl_GlobalInvocationID.x;
uint y = gl_GlobalInvocationID.y;
pressure[cord_p(y, x)] = pressure[cord_p(y, x)] +
pressure_tmp2[cord_p(y, x)];
vec2 vel1;
vel1.x = - (pressure_tmp2[cord_p(y, x)] -
pressure_tmp2[cord_p(y, x-1)])/distx/ro*dt;
vel1.y = - (pressure_tmp2[cord_p(y, x)] -
pressure_tmp2[cord_p(y-1, x)])/disty/ro*dt;

velocity[cord(y, x)].x = velocity_star[cord(y, x)].x + vel1.x;
velocity[cord_t(y, x)].y = velocity_star[cord_t(y, x)].y + vel1.y;
}
```

Приложение Б

Шейдеры алгоритма sph. До кода объявлены необходимые окнстанты и получен доступ к буфферам: position, velocity, force, density, pressure.

Рассчет плотности:

```
void main()
{
uint i = gl_GlobalInvocationID.x;
float density_sum = 0.f;
for (int j = 0; j < NUM_PARTS; j++)
{
vec2 delta = position[i] - position[j];
float r = length(delta);
if (r < SMOOTH_LEN)
{
density_sum += PARTS_MASS * 315.f *
pow(SMOOTH_LEN * SMOOTH_LEN - r * r, 3) /
(64.f * PI_FLOAT * pow(SMOOTH_LEN, 9));
}
}
density[i] = density_sum;
pressure[i] = max(PARTS_STIFFNESS *
(density_sum - PARTS_DENSITY0), 0.f);
}
```

Рассчет силы и интегрирование:

```
void main()
{
uint i = gl_GlobalInvocationID.x;
vec2 pressure_force = vec2(0, 0);
vec2 viscosity_force = vec2(0, 0);
for (uint j = 0; j < NUM_PARTS; j++)
```

```
{
if (i == j)
{
continue;
}
vec2 delta = position[i] - position[j];
float r = length(delta);
if (r < SMOOTH_LEN)
{
pressure_force -= PARTS_MASS * (pressure[i] + pressure[j]) /
(2.f * density[j]) * (-45.f) /
(PI_FLOAT * pow(SMOOTH_LEN, 6)) *
 pow(SMOOTH_LEN - r, 2) * normalize(delta);
viscosity_force += PARTS_MASS * (velocity[j] - velocity[i]) /
density[j] * 45.f / (PI_FLOAT * pow(SMOOTH_LEN, 6)) *
(SMOOTH_LEN - r);
}
}
viscosity_force *= PARTS_VISCOSITY;
vec2 external_force = density[i] * GRAVITY_FORCE;
force[i] = pressure_force + viscosity_force + external_force;
vec2 acceleration = force[i] / density[i];
vec2 new_velocity = velocity[i] + TIME_STEP * acceleration;
vec2 new_position = position[i] + TIME_STEP * new_velocity;
// boundary conditions
...
velocity[i] = new_velocity;
position[i] = new_position;
}
```