МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

**Лабораторная работа №5**

**по курсу «Параллельная обработка данных»**

***Моделирование и визуализация системы N взаимодействующих тел с***

***использованием технологий OpenGL и CUDA.***

Выполнил: *А.А. Почечура*

Группа: *8О-406Б*

Преподаватели: К.Г. Крашенинников,

А.Ю. Морозов

Москва, 2023

**Условие**

**Цель работы.** Использование GPU для моделирования и визуализации системы N взаимодействующих тел. Взаимодействие технологий CUDA и OpenGL: vbo + texture. Решение проблемы коллизий множества объектов. Создание простейшей “игры”.

**Задание.**

**Сцена.** Виртуальный куб, у которого отрисовывается только нижняя грань (пол) и

ребра. Внутри куба находятся N частиц (текстурированные сферы), которые отталкиваются друг от друга и от стенок куба. Условно предполагается что стенки куба и частицы одноименно заряжены. Для частиц учитывается ускорение свободного падения. Нижняя грань куба закрашивается в соответствии с напряженностью электрического поля, создаваемого частицами (строится карта напряженности).

**Игрок.** Камера может перемещаться по пространству без каких-либо ограничений. Управление кнопками и мышкой. При приближении камеры к частицам, они должны “убегать” от неё (предполагается наличии большого одноименного заряда у игрока). По нажатию кнопки мыши, игрок совершает “выстрел” сильно заряженной частицей в направлении взгляда. Эта частица движется равномерно и действует только на другие частицы, а на неё саму никто не влияет.

**Программное и аппаратное обеспечение**

**Графический процессор:**

Compute capability: 6.1

Name: NVIDIA GeForce GTX 1050

Total Global Memory: 4294967296 bytes

Shared memory per block: 49152

Registers per block: 32768

Warp size: 32

Max threads per block: (1024, 1024, 64)

Max block: (2147483647, 65535, 65535)

Total constant memory: 65536

Multiprocessors count: 40

**Процессор:**

vendor\_id : GenuineIntel

cpu family: 6

model: 158

model name: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz

stepping: 9

microcode : 0x42

cpu MHz: 1059.228

cache size : 6144 KB

physical id: 0

siblings: 8

core id: 0

cpu cores : 4

apicid: 0

initial apicid: 0

fpu: yes

fpu\_exception: yes

cpuid level: 22

wp : yes

bogomips : 5616.00

clflush size: 64

cache\_alignment : 64

address sizes: 39 bits physical, 48 bits virtual

**Оперативная память и жёсткий диск:**

*Оперативная память:*

Тип оперативной памяти: DDR4

Частота оперативной памяти: 2400 МГц

Размер оперативной памяти: 8 ГБ

Количество слотов под модули памяти: 2

Максимальный объем памяти: 32 Гб

*Жёсткий диск:*

Общий объём жестких дисков (HDD): 1000 ГБ

Общий объем твердотельных накопителей (SSD): 128 Гб

**Программное обеспечение:**

Операционная система: Windows 10

Оболочка: Visual Studio 2022

Компилятор: Nvidia CUDA Compiler C++ 2011

**Метод решения**

Задача делится на несколько этапов: отрисовка сцены (куба), создание шаров в качестве объектов и программирование траектории их движения, построение карты напряжённости для нижней части куба, реализация перемещения камеры и возможности “выстрела” сильно заряженной частицы.

Отрисовка куба – самая лёгкая часть, для её реализации нужно создать объект и задать грани нужного размера.

При создании шаров нужно будет воспользоваться заранее подготовленной текстурой, которая будет наложена на их модели. Расположение шаров нужно задавать каждый раз случайно, в пределах границ куба. Также шарам можно задать вращение вокруг своей оси.

Программирование траектории движения шаров – самая сложная часть задания. Для её выполнения нужно задать начальную скорость (направление) и заряд каждому шару, а затем обновлять эти данные, учитывая их взаимодействие друг с другом (как одноимённых зарядов), взаимодействие со стенами куба, силу тяжести и затухание их движения. Также нужно учесть, что шары не должны вылетать за пределы куба.

Далее, на ГПУ, происходит обновление изображения нижней части куба в соответствии с расположением шаров. Близость шаров к полу определяет размер и яркость пятен, которые буду отрисованы на полу в каждый момент времени (таким образом мы иллюстрируем влияние напряжённости на нижнюю сторону).

Перемещение камеры, а также переносимый ею заряд, реализуется примерно также, как и движения шаров. Камера (игрок) влияет на перемещение шаров как переносчик одноимённого заряда, но сама влиянию с их стороны не подвергается. Также на камеру не действует гравитация, на нижнюю часть куба значение её заряда не действует.

“Выстрел” частицы означает создание (либо обновление положения) нового объекта, который также будет влиять на положение других шаров из-за наличия собственного заряда. На его траекторию не влияют другие шары, стены (может проходить сквозь них) и игрок. При повторном нажатии на левую кнопку мыши положение частицы вновь оказывается возле игрока, а направление движения аналогично направлению камеры в момент нажатия кнопки. На нижней грани куба также стоит учесть положение данной части для отрисовки карты напряжённости.

**Описание программы**

Сначала нужно произвести инициализацию glut, выделить память под массивы, в которых будут храниться данные о положении и направлении шаров (на ЦПУ и ГПУ), выполнить активацию всех функций обновления движения объектов и камеры, нажатия кнопок и движения мыши, создать и загрузить все необходимые текстуры и задать начальные данные всем шарам. Пройдёмся по объектам и функциям, которые использовались в работе:

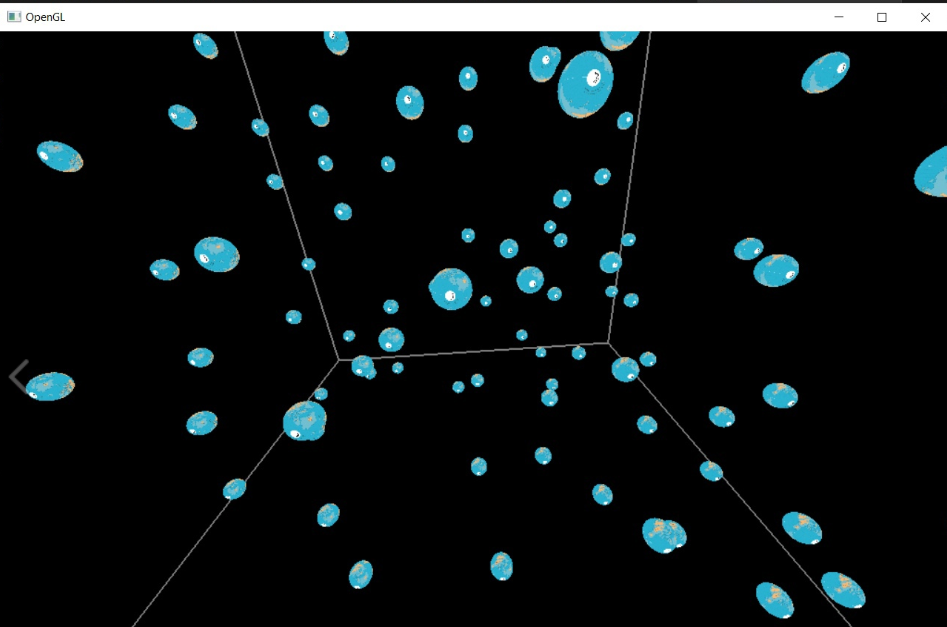
* *struct t\_item* – структура, которая была использована для хранения данных о положении, движении и заряде каждого шара, а также заряженной частицы.
* *t\_item\* balls* – массив шаров, которые изображаются на сцене.
* *t\_item ball\_shot* – переменная для хранения данных о заряженной частице.
* *void display()* – функция перерисовки графики. В ней происходит отображение квадрата, шаров, заряженной частицы, а также отрисовка на экране той части сцены, которую охватывает камера.
* *\_\_global\_\_ void kernel* – единственное ядро в программе, которое используется для обновления карты напряжённости на нижней стороне куба. Принимает в качестве аргументов массив *balls* и переменную *ball\_shot*, чтобы для каждого пикселя актуализировать информацию о действии полей.
* *void update()* – в данной функции происходит обновление параметров положения и направления (ускорение свободного падения, отталкивание, замедление и т.д.) камеры, шаров и частицы, а также запуск ядра.
* *void keys* – функция для обработки нажатия кнопок перемещения камеры и завершения работы программы.
* *void mouse* – функция для обработки движения камеры (с помощью мыши) вокруг своей оси.
* *void mouse\_shot* – функция для изменения состояния заряженной частицы при нажатии на ЛКМ (появление возле “игрока”, статус наличия частицы на сцене становится “истина”).
* *void reshape –* функция для обработки изменения размеров экрана.

**Результаты**

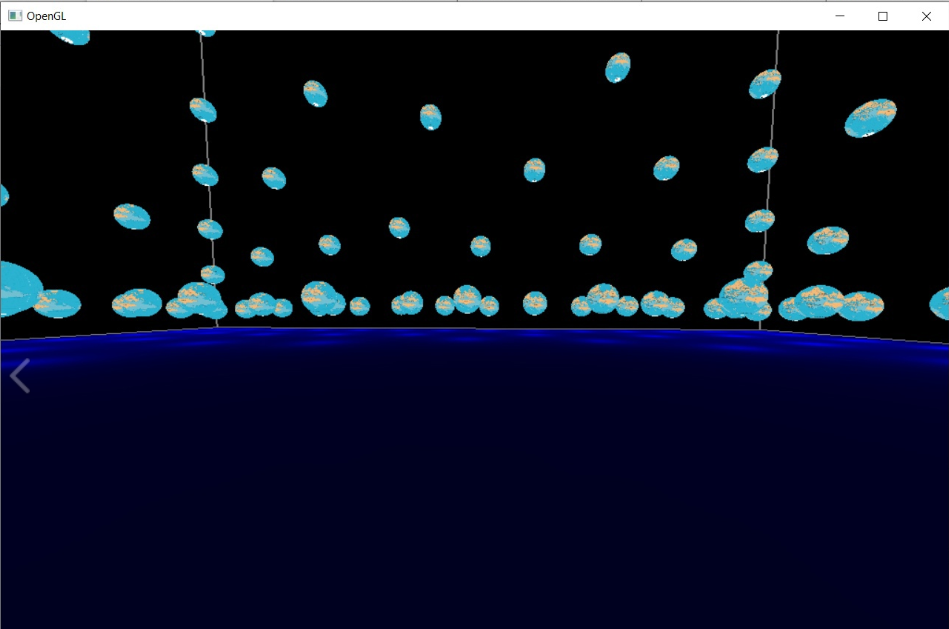
Тестирование работы производилось при количестве потоков, равном << <dim3(32, 32), dim3(32, 8) >> >. При значительном отклонении от этих значений (например: << <dim3(4, 4), dim3(4, 4) >> >) изображение начинает генерироваться отрывисто, карта напряжённости показывает информацию о действии полей сильно хуже.

**<< <dim3(32, 32), dim3(32, 8) >> >**

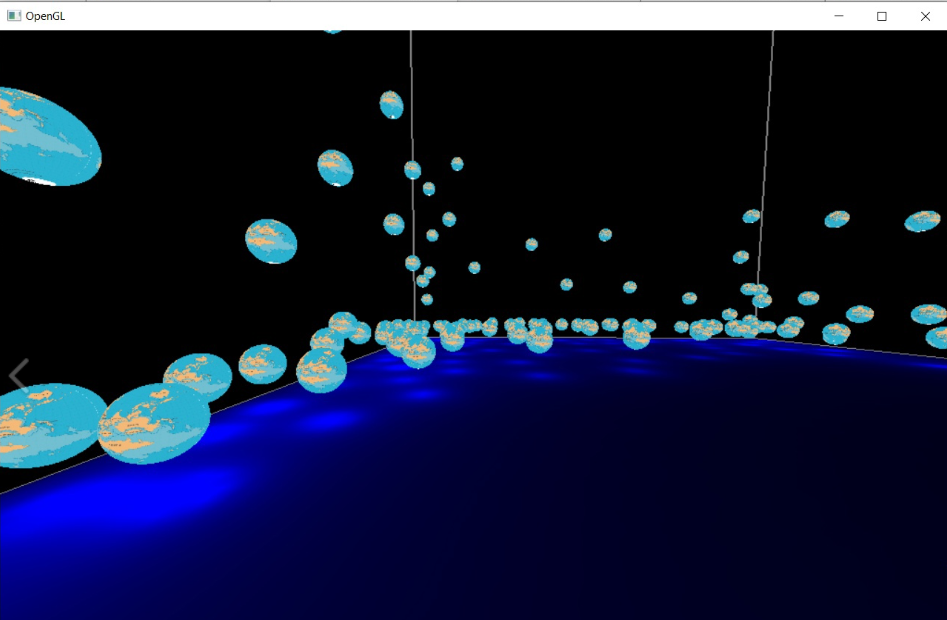
Начальное положение шаров внутри куба:



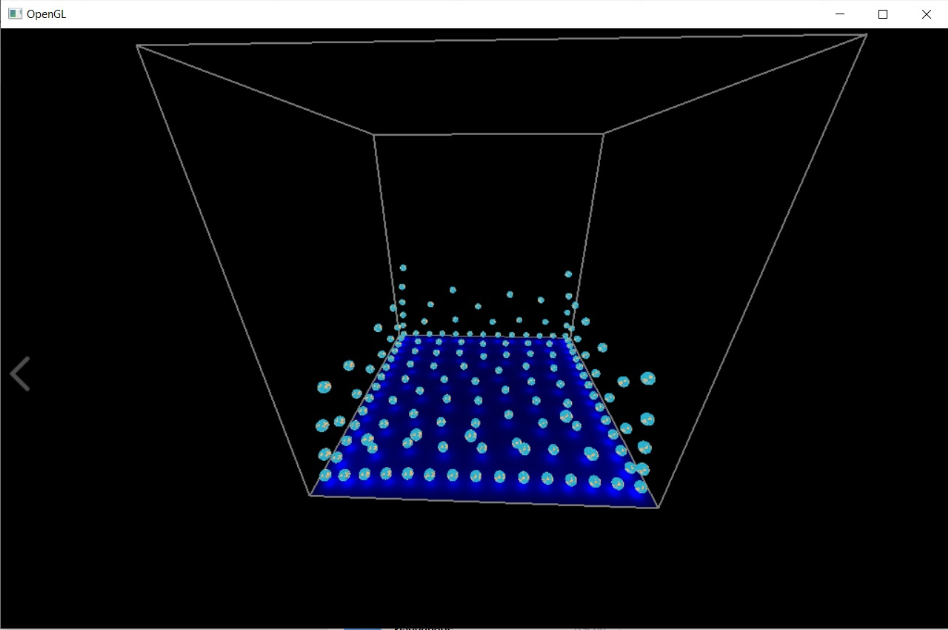
Приземлившись, шары отталкиваются от “игрока” и фиксируются в таком положении:



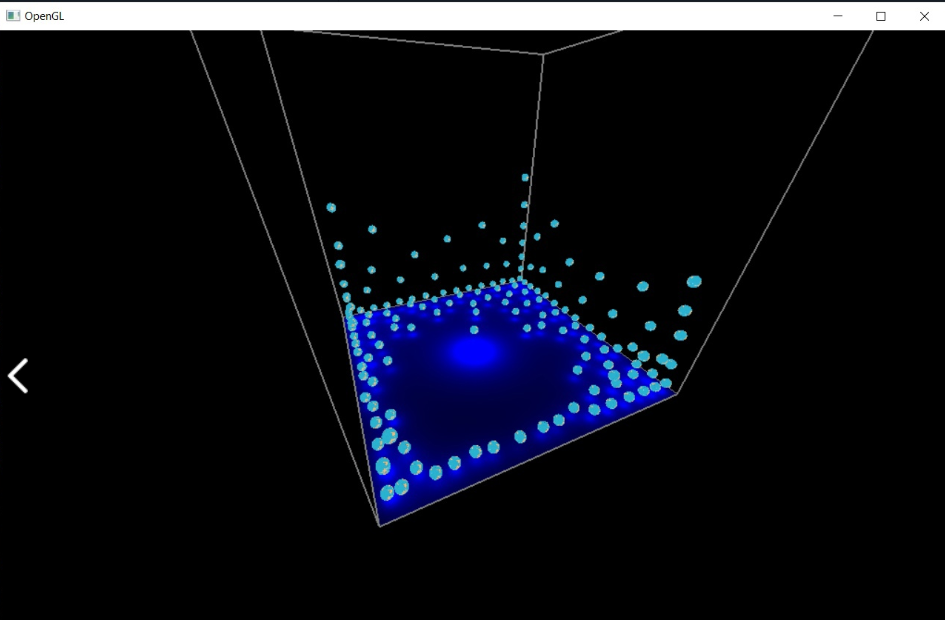
При движении камеры шары начинают отталкиваться от неё:



Если вылететь далеко за грани куба, то шары зафиксируются в одном положении:

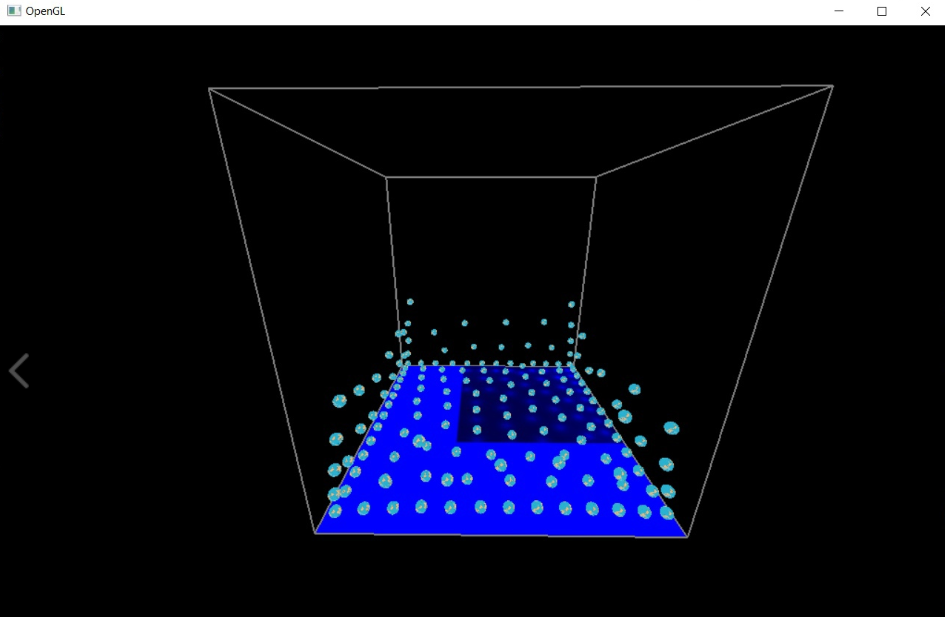


При “выстреле” заряженной частицы, все шары расступаются, а на карте напряжённости появляется характерный рисунок:

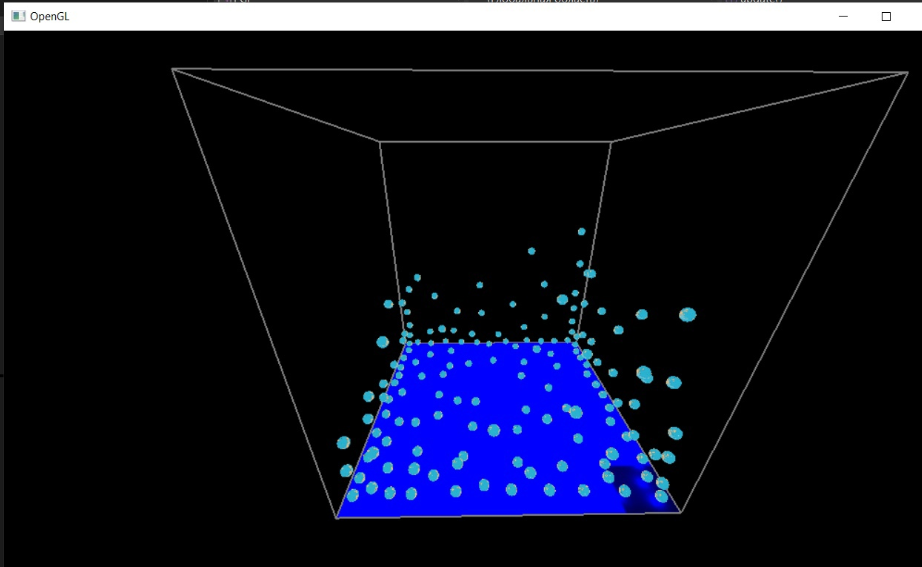


Для сравнения, приведу результаты тестирования при другом количестве потоков:

**<< <dim3(8, 8), dim3(8, 8) >> >**

****

**<< <dim3(4, 4), dim3(4, 4) >> >**



Видим, что карта напряжённости отрисовывает информацию при таких значениях намного хуже. Скорость работы программы также значительно снижается.

**Выводы**

В данной лабораторной работе я научился взаимодействовать с технологиями OpenGL для визуализации изображения с помощью ресурсов ГПУ. Лабораторная работа была безумно интересной: она позволила мне представить, как происходит обновление положения и траектории объектов, как осуществляется изменение значения пикселей на графическом процессоре в реальном времени, а самое главное – как происходит визуализация этих процессов. Возможно, в будущем я вернусь к этой работе, чтобы её усовершенствовать путём добавления новых процессов, объектов и различных событий.