МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №4
по курсу «Параллельная обработка данных»

Моделирование и визуализация системы N взаимодействующих тел с использованием технологий OpenGL и CUDA.

Выполнил: А.С. Федоров
Группа: 8О-407Б
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2022

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”.
Вариант задания. 1.

Программное и аппаратное обеспечение

Графический процессор
Название: NVIDIA GeForce GTX 1050
Compute capability: 6.1
Объем графической памяти: 2147352576 байтов
Объем разделяемой памяти на блок: 49152 байтов
Объем регистров на блок: 65536
Размер варпа: 32
Максимальное количество потоков на блок: (1024, 1024, 64)
Максимальное число блоков: (2147483647, 65535, 65535)
Объем постоянной памяти: 65536 байтов
Число мультипроцессоров: 5

Процессор
Название: i5-8250U
Базовая тактовая частота процессора: 1,60 ГГц
Количество ядер: 4
Количество потоков: 8
Кеш L1: 256 Кб
Кеш L2: 1 Мб
Кеш L3: 6 Мб

Оперативная память
Тип: DDR4
Объем: 11.9 Гб
Частота: 2400 МГц

Програмное обеспечение
ОС: WSL2 (Windows 11)
IDE: Microsoft Visual Studio 2022 (аддон NVIDIA Nsight)
Компилятор: nvcc

Метод решения

Для быстрого обсчета позиций частиц и текстуры пола с напряженностью поля предполагается использовать GPU для параллельного вычисления результата. На каждо1й итерации данные текстуры и позиций частиц обновляются и затем, на их основе отрисовывается новый кадр. Также происходит обработка «снаряда», который имеет гораздо больший заряд, чем обычная частица, не подвержен внешнему влиянию и имеет постоянное направление и скорость.

Описание программы

Для пересчета позиций используется CUDA ядро, которое, по сути, реализует формулы пересчета, приведенные в методических материалах к лабораторной работе. На каждой итерации данные по позициях частиц загружаются в глобальную память GPU, обрабатываются и выгружаются обратно для дальнейшей отрисовки.

Код ядра пересчета позиций частиц:

```
__global__ void particle_kernel(particle* particles, uint num_particles,
particle cam_particle, particle projectile, float K, float W, float g, float
offset, float dt, float e) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    while (idx < num_particles){
        // замедление
            particles[idx].dx *= W;
            particles[idx].dy *= W;
            particles[idx].dz *= W;
```

```
        // Отталкивание от стен
            particles[idx].dx += 2*sqr(particles[idx].q)*K*(particles[idx].x-
offset)/(sqr3(fabs(particles[idx].x-offset))+e)*dt;
            particles[idx].dx +=
2*sqr(particles[idx].q)*K*(particles[idx].x+offset)/(sqr3(fabs(particles[idx]
.x+offset))+e)*dt;

            particles[idx].dy += 2*sqr(particles[idx].q)*K*(particles[idx].y-
offset)/(sqr3(fabs(particles[idx].y-offset))+e)*dt;
            particles[idx].dy +=
2*sqr(particles[idx].q)*K*(particles[idx].y+offset)/(sqr3(fabs(particles[idx]
.y+offset))+e)*dt;

            particles[idx].dz +=
2*particles[idx].q*particles[idx].q*K*(particles[idx].z-
2*offset)/(sqr3(fabs(particles[idx].z-2*offset))+e)*dt;
            particles[idx].dz +=
2*particles[idx].q*particles[idx].q*K*(particles[idx].z)/(sqr3(fabs(particles
[idx].z))+e)*dt;

        // Отталкивание от камеры
            float cam_r = sqrt(sqr(particles[idx].x-
cam_particle.x)+sqr(particles[idx].y-cam_particle.y)+sqr(particles[idx].z-
cam_particle.z));
            particles[idx].dx +=
cam_particle.q*particles[idx].q*K*(particles[idx].x-
cam_particle.x)/(sqr3(cam_r)+e)*dt;
            particles[idx].dy +=
cam_particle.q*particles[idx].q*K*(particles[idx].y-
cam_particle.y)/(sqr3(cam_r)+e)*dt;
            particles[idx].dz +=
cam_particle.q*particles[idx].q*K*(particles[idx].z-
cam_particle.z)/(sqr3(cam_r)+e)*dt;
```

```
        // отталкивание от снаряда
            float projectile_r = sqrt(sqr(particles[idx].x-
projectile.x)+sqr(particles[idx].y-projectile.y)+sqr(particles[idx].z-
projectile.z));
            particles[idx].dx +=
projectile.q*particles[idx].q*K*(particles[idx].x-
projectile.x)/(sqr3(projectile_r)+e)*dt;
            particles[idx].dy +=
projectile.q*particles[idx].q*K*(particles[idx].y-
projectile.y)/(sqr3(projectile_r)+e)*dt;
            particles[idx].dz +=
projectile.q*particles[idx].q*K*(particles[idx].z-
projectile.z)/(sqr3(projectile_r)+e)*dt;

        // отталкивание от остальных частиц
            for (uint i = 0; i < num_particles; ++i){
                if (idx == i)
                    continue;

                float r = sqrt(sqr(particles[idx].x-
particles[i].x)+sqr(particles[idx].y-particles[i].y)+sqr(particles[idx].z-
particles[i].z));
                particles[idx].dx +=
particles[i].q*particles[idx].q*K*(particles[idx].x-
particles[i].x)/(sqr3(r)+e)*dt;
                particles[idx].dy +=
particles[i].q*particles[idx].q*K*(particles[idx].y-
particles[i].y)/(sqr3(r)+e)*dt;
                particles[idx].dz +=
particles[i].q*particles[idx].q*K*(particles[idx].z-
particles[i].z)/(sqr3(r)+e)*dt;
            }

        // гравитация
            particles[idx].dz -= g*dt;

        // шаг по времени
            particles[idx].x += particles[idx].dx*dt;
            particles[idx].y += particles[idx].dy*dt;
            particles[idx].z += particles[idx].dz*dt;
            particles[idx].angle += particles[idx].spin*dt;
```

```
        // коллизия со стенами
            if (particles[idx].x < -offset+e)
                particles[idx].x = -offset+e;
            if (particles[idx].x > offset-e)
                particles[idx].x = offset-e;

            if (particles[idx].y < -offset+e)
                particles[idx].y = -offset+e;
            if (particles[idx].y > offset-e)
                particles[idx].y = offset-e;

            if (particles[idx].z < e)
                particles[idx].z = e;
            if (particles[idx].z > 2*offset-e)
                particles[idx].z = 2*offset-e;

        idx += blockDim.x * gridDim.x;
    }
}
```

Пересчет текстуры напряженностей аналогично используется отдельное CUDA ядро. О отличии от предыдущего ядра, данное ядро взаимодействует с текстурным буфером OpenGL. Этот буфер предварительно зарегистрирован для работы с CUDA. На каждой итерации данный буфер занимается CUDA, чтобы пересчитать значения цветов пикселей. Пересчет происходит по формулам, приведенным в методичке в лабораторной работе. Вычисления осуществляются на двумерной сетке потоков. После завершения работы ядра, буфер становится доступен для OpenGL и последующей отрисовки кадра.

Код ядра пересчета текстуры пола:

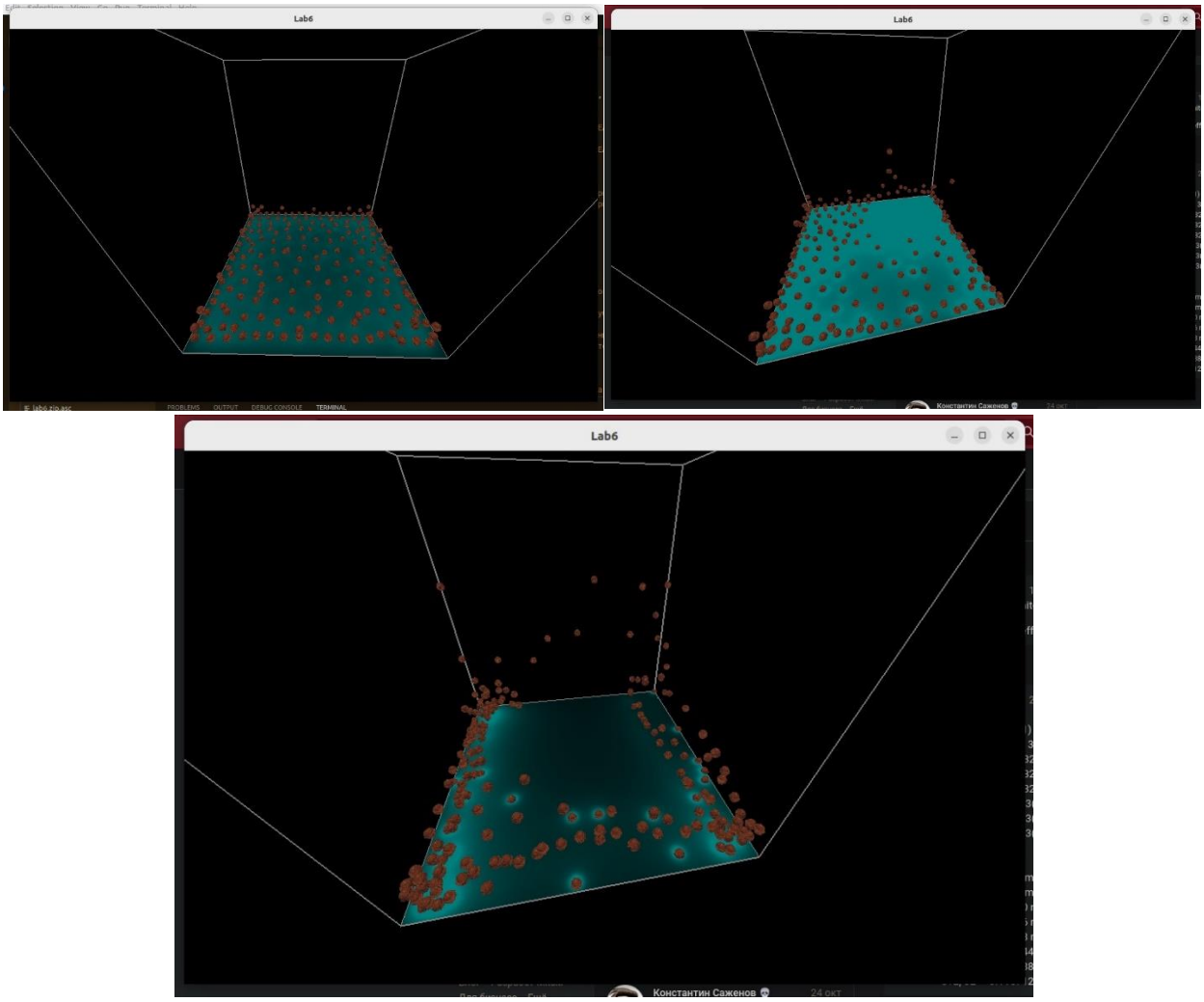
```
__global__ void floor_kernel(uchar4* dev_floor_data, uint floor_texture_size,
particle* particles, uint num_particles, particle projectile, float offset){
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int idy = blockIdx.y * blockDim.y + threadIdx.y;

    float x, y;
    for (uint i = idx; i < floor_texture_size; i += blockDim.x*gridDim.x){
        for (uint j = idy; j < floor_texture_size; j += blockDim.y*gridDim.y){
            x = (2.0*i/(floor_texture_size-1.0)-1.0)*offset;
            y = (2.0*j/(floor_texture_size-1.0)-1.0)*offset;

            float fg = 0.0;
            for (uint k = 0; k < num_particles; ++k){
                fg += 100.0*particles[k].q/(sqr(x-particles[k].x)+sqr(y-
particles[k].y)+sqr(particles[k].z));
            }
            fg += 100.0*projectile.q/(sqr(x-projectile.x)+sqr(y-
projectile.y)+sqr(projectile.z));

            fg = min(max(0.0f, fg), 255.0f);
            dev_floor_data[j*floor_texture_size+i] = make_uchar4(0, (uint)fg,
(uint)fg, 255);
        }
    }
}
```

Скриншоты работы программы:



Тест быстродействия (в миллисекундах)

Текстура (128 на 128)		Частицы (200)	
GPU (CUDA) <dim3(1, 1), dim3(1, 1)>	1761.061	**GPU (CUDA) <1, 1>**	22.219
GPU (CUDA) <dim3(1, 1), dim3(32, 32)>	7.621	**GPU (CUDA) <1, 32>**	0.646
GPU (CUDA) <dim3(16, 16), dim3(32, 32)>	**1.713**	**GPU (CUDA) <16, 32>**	0.117
GPU (CUDA) <dim3(32, 32), dim3(32, 32)>	**1.713**	**GPU (CUDA) <32, 32>**	0.106
GPU (CUDA) <dim3(64, 64), dim3(32, 32)>	**1.713**	**GPU (CUDA) <64, 32>**	**0.114**
GPU (CUDA) <dim3(128, 128), dim3(32, 32)>	2.507	**GPU (CUDA) <128, 32>**	0.108
GPU (CUDA) <dim3(256, 256), dim3(32, 32)>	4.773	**GPU (CUDA) <256, 32>**	**0.114**
GPU (CUDA) <dim3(512, 512), dim3(32, 32)>	16.333	**GPU (CUDA) <512, 32>**	0.115
CPU	-	**CPU**	2.277

Выводы

Параллельная обработка симуляций вычислительными ресурсами GPU имеет очень широкое применение: от создания визуальных эффектов и симуляции физики в движках реального времени, до обсчета уравнений в частных производных для задач проектирования. Реализация алгоритма пересчета позиций частиц и значений пикселей текстуры оказалась относительно несложной, так как вся необходимая математика была изложена в методических материалах к лабораторной работе.

Исходя из результатов замера быстродействия видно, что увеличение числа потоков для обработки на GPU положительно сказывается на скорости отрисовки. Однако, при достижении порога, когда каждой частице (каждому пикселю) соответствует хотя бы один поток, прироста в быстродействии более не происходит. Наоборот, из-за большого количества потоков, обработка которых требует ресурсов время на выполнение итерации начинает возрастать.