Laboverslag n-body probleem

Dries Kennes & Stijn Van Dessel

May 13, 2018

Abstract

In dit laboverslag kan u de testresultaten van het n-body probleem voor het vak parallelle programmatie terugvinden. Dit verslag geeft een overzicht van de bekomen testresultaten. De testen werden uitgevoerd met een verschillend aantal body's en met verschillende soorten kernels op de GPU. Daarnaast werd de opgave ook getest op de CPU als referentie.

1 De geteste modi

Om dit verslag overzichtelijk te houden is alle code met regelnummers als bijlage toegevoegd. Alle testresultaten en conclusies zijn verzameld in sectie 2.

1.1 Referentie-implementatie CPU

Als benchmark wordt de niet geparallelliseerde code op CPU getest. Deze code is aangepast om dezelfde rendering code te gebruiken als de GPU-geaccelereerde code, om de resultaten vergelijkbaar te maken. Deze resultaten geven een goed inzicht in het verschil tussen GPU en GPU. De overhead van het kopiëren van data van en naar de GPU eerst zorgt voor een aanzienlijk verschil in het voordeel van de CPU.

De code voor deze modus is te vinden in de bijlage 3.2 (main.c), regels 94 t.e.m. 141.

1.2 Kernel 1: FLOAT

Voor de eerste GPU implementatie wordt enkel de buitenste for lust vervangen door een parallellisatie. We zien hier het concept data-afhankelijkheid voor het eerst opduiken. Aangezien we ze zeker van moeten zijn dat eerst alle snelheden zijn berekend voor er een van de posities wordt gewijzigd. Dit is belangrijk aangezien de positie nodig is om de snelheid te berekenen. Daarom wordt deze instructie toegevoegd:

```
1 barrier(CLK_GLOBAL_MEM_FENCE);
```

Hiermee geven we aan dat iedere kernel op dit punt moet wachten totdat al de andere kernels ook tot op dit punt zijn geraakt. Als we dit niet zouden doen zouden sommige kernels al een positie update kunnen doen met als gevolg dat kernels die nog niet klaar zijn met rekenen zouden kunnen gaan verder rekenen met deze verkeerde waarden. Dit doet zich voor als het aantal workitems groter wordt dan het aantal beschikbare CPU cores.

De code voor deze kernel is te vinden in de bijlage 3.1 (kernel.cl), regels 46 t.e.m. 88.

1.3 Kernel 2: FLOAT3

De volgende test is het vergelijken van het verschil tussen het gebruik van 3 floats of het ingebouwde float3 datatype. Omdat float3 een ingebouwd type van OpenCL is, heeft het allerlei hulpfuncties en wiskundige operatoren ingebouwd. Dit maakt het programmeren eenvoudiger en de code overzichtelijker.

De code voor deze kernel is te vinden in de bijlage 3.1 (kernel.cl), regels 124 t.e.m. 156.

1.4 Kernel 3: 2D

Kernel 1 en 2 bevatten nog steeds een for lus, en kunnen dus nogmaals geparallelliseerd worden. De workitems wordt dan een 2d array, vandaar de naam van deze kernel. De delen van de code die buiten de for lus liggen kunnen niet op GPU worden uitgevoerd, tenzij hiervoor nog een extra kerel wordt toegevoegd. Dit hebben wij niet onderzocht.

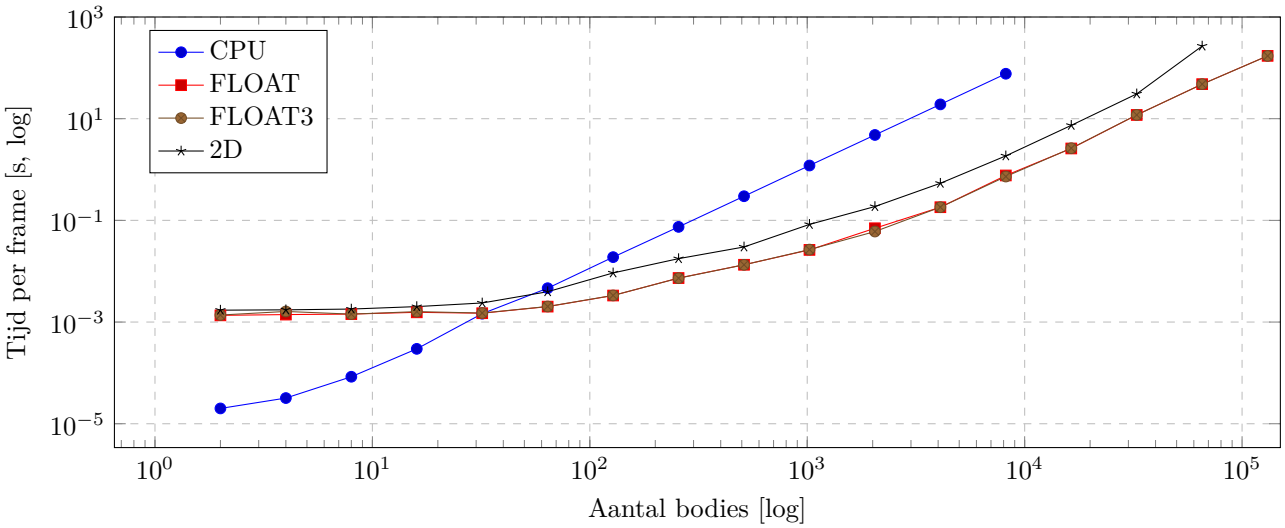
De code voor deze kernel is te vinden in de bijlage 3.1 (kernel.cl), regels 90 t.e.m. 122.

2 Resultaten

De tijden zijn bekomen op een Intel i5-4670K (4GHz) met 16GB RAM en een NVIDIA GeForce GTX 1060 6GB, Linux 4.18.8 kernel met de Zen Kernel patches. Na het bereiken van 60 seconden per frame wordt deze reeks kernel tests stop gezet, omdat de volgende nog veel langer zou duren. 60 seconden per frame over 100 frames (om een representatief gemiddelde te kunnen maken) duurt al 1h40.

Bodies	CPU	FLOAT	FLOAT3	2D
2	0.000020	0.001354	0.001374	0.001713
4	0.000032	0.001403	0.001616	0.001740
8	0.000084	0.001435	0.001432	0.001802
16	0.000297	0.001554	0.001604	0.002014
32	0.001462	0.001492	0.001504	0.002374
64	0.004634	0.002012	0.002025	0.003951
128	0.018927	0.003325	0.003342	0.009255
256	0.074447	0.007342	0.007291	0.017675
512	0.298560	0.013389	0.013361	0.030005
1024	1.199929	0.026264	0.026623	0.083283
2048	4.786847	0.070001	0.060309	0.187174
4096	19.115835	0.181524	0.181600	0,536761
8192	76.570487	0.764762	0.728157	1.856455
16384	na	2.595713	2.625760	7.400403
32768	na	11.846362	11.865299	30.745776
65536	na	47.805630	47.775654	268.237756
131072	na	172.297190	171.784650	na

De resultaten, dubbel logaritmisch



Hieruit is duidelijk te zien dat het verloop exponentieel is, wat verwacht is bij code die een dubbele for lus heeft. Echter heeft de CPU een voorsprong omdat er minder overhead is. De GPU kernels moeten namelijk worden opgestart met data die moet worden gekopieerd van en naar de het CPU geheugen. Daarom is de curve van de GPU modi eerst vlak, om daarna een substantieel voordeel op te leveren ten opzichte van de CPU.

Er is geen (merkbaar) verschil tussen de FLOAT en FLOAT3 kernels, dit kan eenvoudig worden verklaard met wat achtergrondinformatie: NVIDEA is een scalaire architectuur, en implementeert de vectoroperatie intern als een reeks scalaire operaties.

Er is echter wel een merkbaar verschil tussen de 1D en 2D kernels. De 2D kernel is trager door twee oorzaken. Ten eerste moet een deel van de berekening op CPU worden uitgevoerd, maar is het resultaat daarvan nodig bij de volgende iteratie. Daardoor moet het geheugen opnieuw van CPU naar GPU worden gekopieerd, wat bij de 1D kernels niet nodig is. Ten tweede moet de som operatie atomisch worden uitgevoerd. Dit wil zeggen dat er meermaals geprobeerd moet worden om de som uit te voeren tot ze correct is gebeurd (omdat er geen native atomic add is kan die zo worden geëmuleerd). Omdat alle kernels in een workgroup dezelfde instructies moeten uitvoeren moeten ze dus allemaal wachten tot elk van de som operaties geslaagd is. Dit voegt meer overhead toe dan de snelheidswinst die de uitbreiding naar 2D oplevert.

Onze conclusie is dat de FLOAT3 de beste kernel is. Op ons NVIDEA platform leverde dit geen voordeel op t.o.v. de FLOAT kernel, maar op een ander platform zou dit wel het geval kunnen zijn. Ook is de code duidelijker en eenvoudiger te schrijven.

3 Bijlage

3.1 kernel.cl

```
1  /*
2   * nBody problem OpenCL kernel
3   */
4
5  #ifdef FOEFELMAGIC
6  // This makes CLion think this is a C file.
7  #define __global
8  #define __kernel
9  #define float3 struct {float x, y, z;}
10 #define get_global_id(dim) dim
11 #define sqrt(a) a
12 #define CLK_GLOBAL_MEM_FENCE 0
13 #define barrier(type) do {} while(0)
14 #endif
15
16 // Required for AtomicAdd
17 typedef union
18 {
19     float3 vec;
20     float arr[3];
21 } float3_;
22 // Based on: http://suhorukov.blogspot.be/2011/12/opencl-11-atomic-operations-on-floating.html
23 inline void AtomicAdd(volatile __global float *source, const float operand) {
24     union {
25         unsigned int intVal;
26         float floatVal;
27     } newVal;
28     union {
29         unsigned int intVal;
30         float floatVal;
31     } prevVal;
32     do {
33         prevVal.floatVal = *source;
34         newVal.floatVal = prevVal.floatVal + operand;
35     } while (atomic_cmpxchg((volatile __global unsigned int *)source, prevVal.intVal,
36                             newVal.intVal) != prevVal.intVal);
37 }
38
39 // Our data type (must be kept in sync with main program code)
40 typedef struct body
41 {
42     float3 pos;
43     float3_ speed;
44 } Body;
45
46 /** KERNEL FLOAT */
47 __kernel void kernel_step_float(unsigned long n, __global Body * data)
48 {
49     const float SCALE = 50;
50     const float TIME_DELTA = 1;
51     const float GRAV_CONST = 1;
52     const float MASS = 2;
53     const float MASS_GAV = (GRAV_CONST * MASS * MASS);
54
55     const int body = get_global_id(0);
56
57     if (body >= n) return;
58
59     __global Body* us = &data[body];
60
61     for (int i = 0; i < n; i++)
62     {
63         if (i == body) continue;
64
65         __global const Body* them = &data[i];
66
67         float dist_x = (us->pos.x - them->pos.x) * SCALE;
68         float dist_y = (us->pos.y - them->pos.y) * SCALE;
69         float dist_z = (us->pos.z - them->pos.z) * SCALE;
70
71         float dist = sqrt(dist_x * dist_x + dist_y * dist_y + dist_z * dist_z);
72
```

```
73          float acc_x = (-MASS_GAV * dist_x / (dist * dist * dist)) / MASS;
74          float acc_y = (-MASS_GAV * dist_y / (dist * dist * dist)) / MASS;
75          float acc_z = (-MASS_GAV * dist_z / (dist * dist * dist)) / MASS;
76
77
78          us->speed.vec.x += acc_x * TIME_DELTA;
79          us->speed.vec.y += acc_y * TIME_DELTA;
80          us->speed.vec.z += acc_z * TIME_DELTA;
81      }
82
83      barrier(CLK_GLOBAL_MEM_FENCE);
84
85      us->pos.x += us->speed.vec.x * (TIME_DELTA / SCALE);
86      us->pos.y += us->speed.vec.y * (TIME_DELTA / SCALE);
87      us->pos.z += us->speed.vec.z * (TIME_DELTA / SCALE);
88  }
89
90  /** KERNEL FLOAT3 */
91  __kernel void kernel_step_float3(unsigned long n, __global Body * data)
92  {
93      const float SCALE = 50;
94      const float TIME_DELTA = 1;
95      const float GRAV_CONST = 1;
96      const float MASS = 2;
97      const float MASS_GAV = (GRAV_CONST * MASS * MASS);
98
99      const int body = get_global_id(0);
100
101     if (body >= n) return;
102
103     __global Body* us = &data[body];
104
105     for (int i = 0; i < n; i++)
106     {
107         if (i == body) continue;
108
109         __global const Body* them = &data[i];
110
111         float3 dist_all = (us->pos - them->pos) * SCALE;
112         float3 dist_all_sq = dist_all * dist_all;
113         float dist = sqrt(dist_all_sq.x + dist_all_sq.y + dist_all_sq.z);
114
115         float3 acc = (-MASS_GAV * dist_all / (dist * dist * dist)) / MASS;
116         us->speed.vec += acc * TIME_DELTA;
117     }
118
119     barrier(CLK_GLOBAL_MEM_FENCE);
120
121     us->pos += us->speed.vec * (TIME_DELTA / SCALE);
122 }
123
124 /** KERNEL 2D */
125 __kernel void kernel2_step_float3(unsigned long n, __global Body * data)
126 {
127     const float SCALE = 50;
128     const float TIME_DELTA = 1;
129     const float GRAV_CONST = 1;
130     const float MASS = 2;
131     const float MASS_GAV = (GRAV_CONST * MASS * MASS);
132
133     const int body = get_global_id(0);
134
135     if (body >= n) return;
136
137     __global Body* us = &data[body];
138
139     const int i = get_global_id(1);
140
141     if (i >= n) return;
142
143     if (i == body) return;
144
145     __global const Body* them = &data[i];
146
147     float3 dist_all = (us->pos - them->pos) * SCALE;
148     float3 dist_all_sq = dist_all * dist_all;
```

```
149     float dist = sqrt(dist_all_sq.x + dist_all_sq.y + dist_all_sq.z);
150
151     float3 acc = (-MASS_GAV * dist_all / (dist * dist * dist)) / MASS;
152
153     AtomicAdd(&us->speed.arr[0], acc.x * TIME_DELTA);
154     AtomicAdd(&us->speed.arr[1], acc.y * TIME_DELTA);
155     AtomicAdd(&us->speed.arr[2], acc.z * TIME_DELTA);
156 }
```

3.2 main.c

```
1  /**
2   * (c) Dries Kennes & Stijn Van Dessel 2018
3   *
4   * nBody problem
5   */
6  #include "lib/ocl_utils.h"
7  #include "lib/renderer.h"
8
9  #define AVG_MAX 100
10
11 enum MODE {
12     MODE_CPU,
13     MODE_FLOAT3,
14     MODE_FLOAT,
15     MODE_2D,
16 };
17
18 typedef struct body
19 {
20     cl_float3 pos;
21     cl_float3 speed;
22 } Body;
23
24 typedef struct callback_data {
25     cl_kernel kernel;
26     size_t n;
27     Body* bodies;
28     cl_mem dev_bodies;
29     Vect4f point_color;
30     float point_size;
31     bool draw_lines;
32     Vect4f line_color;
33     enum MODE mode;
34 } CallbackData;
35
36 Body* create_bodies(size_t n)
37 {
38     Body* data = calloc(n, sizeof(Body));
39     if (data == NULL) abort();
40
41     for (int i = 0; i < n; i++)
42     {
43         float offset = (rand() < RAND_MAX / 2) ? -5 : 5;
44         data[i].pos.x = ((float)rand() / (float)RAND_MAX) * 2.f - 1.f + offset;
45         data[i].pos.y = ((float)rand() / (float)RAND_MAX) * 2.f - 1.f;
46         data[i].pos.z = ((float)rand() / (float)RAND_MAX) * 2.f - 1.f;
47
48         data[i].speed.x = ((float)rand() / (float)RAND_MAX) * 1.f - 0.5f;
49         data[i].speed.y = ((float)rand() / (float)RAND_MAX) * 1.f - 0.5f;
50         data[i].speed.z = ((float)rand() / (float)RAND_MAX) * 1.f - 0.5f;
51     }
52
53     return data;
54 }
55
56 void draw(void* data)
57 {
58     CallbackData* cd = data;
59
60     glColor4fv((const GLfloat *) &cd->point_color);
61     glPointSize(cd->point_size);
62     glBegin(GL_POINTS);
63     for (int i = 0; i < cd->n; i++)
64     {
65         glVertex3fv((float *) &cd->bodies[i].pos);
```

```
66      }
67      glEnd();
68      if (cd->draw_lines)
69      {
70          glColor4fv((float *) &cd->line_color);
71          glBegin(GL_LINES);
72          for (int i = 0; i < cd->n; i++)
73          {
74              glVertex3fv((float *) &cd->bodies[i].pos);
75              glVertex3f(cd->bodies[i].pos.x + cd->bodies[i].speed.x / 2, cd->bodies[i].pos.y +
                    cd->bodies[i].speed.y / 2, cd->bodies[i].pos.z + cd->bodies[i].speed.z / 2);
76          }
77          glEnd();
78      }
79  }
80
81  struct timespec diff(struct timespec start, struct timespec end)
82  {
83      struct timespec temp;
84      if ((end.tv_nsec-start.tv_nsec)<0) {
85          temp.tv_sec = end.tv_sec-start.tv_sec-1;
86          temp.tv_nsec = 1000000000+end.tv_nsec-start.tv_nsec;
87      } else {
88          temp.tv_sec = end.tv_sec-start.tv_sec;
89          temp.tv_nsec = end.tv_nsec-start.tv_nsec;
90      }
91      return temp;
92  }
93
94  void kernel_cpu(CallbackData* cd)
95  {
96      const float delta_time = 1.f;
97      // const float grav_constant = 6.67428e-11;
98      const float grav_constant = 1;
99      const float mass_of_sun = 2;
100     const float mass_grav = grav_constant * mass_of_sun * mass_of_sun;
101     const float distance_to_nearest_star = 50;
102
103     for (int i = 0; i < cd->n; ++i)
104     {
105         for (int j = 0; j < cd->n; ++j)
106         {
107             if (i == j) continue;
108
109             cl_float3 pos_a = cd->bodies[i].pos;
110             cl_float3 pos_b = cd->bodies[j].pos;
111
112             float dist_x = (pos_a.s[0] - pos_b.s[0]) * distance_to_nearest_star;
113             float dist_y = (pos_a.s[1] - pos_b.s[1]) * distance_to_nearest_star;
114             float dist_z = (pos_a.s[2] - pos_b.s[2]) * distance_to_nearest_star;
115
116             float distance = sqrt(
117                     dist_x * dist_x +
118                     dist_y * dist_y +
119                     dist_z * dist_z);
120
121             float force_x = -mass_grav * dist_x / (distance * distance * distance);
122             float force_y = -mass_grav * dist_y / (distance * distance * distance);
123             float force_z = -mass_grav * dist_z / (distance * distance * distance);
124
125             float acc_x = force_x / mass_of_sun;
126             float acc_y = force_y / mass_of_sun;
127             float acc_z = force_z / mass_of_sun;
128
129             cd->bodies[i].speed.s[0] += acc_x * delta_time;
130             cd->bodies[i].speed.s[1] += acc_y * delta_time;
131             cd->bodies[i].speed.s[2] += acc_z * delta_time;
132         }
133     }
134
135     for (int i = 0; i < cd->n; ++i)
136     {
137         cd->bodies[i].pos.s[0] += (cd->bodies[i].speed.s[0] * delta_time) /
                distance_to_nearest_star;
138         cd->bodies[i].pos.s[1] += (cd->bodies[i].speed.s[1] * delta_time) /
                distance_to_nearest_star;
```

```
139         cd->bodies[i].pos.s[2] += (cd->bodies[i].speed.s[2] * delta_time) /
                distance_to_nearest_star;
140     }
141 }
142
143 void step(void* data)
144 {
145     static struct timespec avg[AVG_MAX];
146     static int time_index = 0;
147
148     struct timespec time1, time2;
149     clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &time1);
150
151     CallbackData* cd = data;
152
153     switch (cd->mode)
154     {
155         case MODE_CPU:
156             kernel_cpu(cd);
157         break;
158         case MODE_FLOAT3:
159         case MODE_FLOAT:
160             ocl_err(clEnqueueNDRangeKernel(g_command_queue, cd->kernel, 1, NULL, &cd->n, NULL,
                    0, NULL, NULL)); //1D
161             ocl_err(clFinish(g_command_queue));
162             ocl_err(clEnqueueReadBuffer(g_command_queue, cd->dev_bodies, CL_TRUE, 0, sizeof(
                    Body) * cd->n, cd->bodies, 0, NULL, NULL));
163             ocl_err(clFinish(g_command_queue));
164             break;
165         case MODE_2D:
166         {
167             size_t size[] = {cd->n, cd->n};
168             ocl_err(clEnqueueWriteBuffer(g_command_queue, cd->dev_bodies, CL_TRUE, 0, sizeof(
                    Body) * cd->n, cd->bodies, 0, NULL, NULL));
169             ocl_err(clFinish(g_command_queue));
170             ocl_err(clEnqueueNDRangeKernel(g_command_queue, cd->kernel, 2, NULL, size, NULL,
                    0, NULL, NULL)); //2D
171             ocl_err(clFinish(g_command_queue));
172             ocl_err(clEnqueueReadBuffer(g_command_queue, cd->dev_bodies, CL_TRUE, 0, sizeof(
                    Body) * cd->n, cd->bodies, 0, NULL, NULL));
173             ocl_err(clFinish(g_command_queue));
174             const float SCALE = 50;
175             const float TIME_DELTA = 1;
176             for (int i = 0; i < cd->n; ++i)
177             {
178                 cd->bodies[i].pos.s[0] += (cd->bodies[i].speed.s[0] * TIME_DELTA) / SCALE;
179                 cd->bodies[i].pos.s[1] += (cd->bodies[i].speed.s[1] * TIME_DELTA) / SCALE;
180                 cd->bodies[i].pos.s[2] += (cd->bodies[i].speed.s[2] * TIME_DELTA) / SCALE;
181             }
182         }
183             break;
184         default:
185             abort();
186     }
187
188     clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &time2);
189     avg[time_index] = diff(time1, time2);
190     time_index ++;
191     if (time_index == AVG_MAX)
192     {
193         time_index = 0;
194         double sum = 0;
195         for (int i = 0; i < AVG_MAX; i++) {
196             sum += avg[i].tv_sec + avg[i].tv_nsec / 1000000000.0;
197         }
198         sum /= AVG_MAX;
199         printf("AVG: %lf\n", sum);
200     }
201 }
202
203 int main(int argc, char ** argv)
204 {
205     if (argc <= 2)
206     {
207         printf("%s <number of bodies> <mode>\n", argv[0]);
208         printf("Mode must be one of:\n");
```

```
209             printf("- CPU\n");
210             printf("- FLOAT\n");
211             printf("- FLOAT3\n");
212             printf("- 2D\n");
213             return -1;
214         }
215         errno = 0;
216         cl_ulong n = strtoul(argv[1], NULL, 0);
217         if (errno != 0)
218         {
219             printf("'%s' was not converted to a proper number. (%ld)\n", argv[1], n);
220             return -1;
221         }
222         enum MODE mode;
223         if (strcmp(argv[2], "CPU") == 0 || strcmp(argv[2], "cpu") == 0) mode = MODE_CPU;
224         else if (strcmp(argv[2], "FLOAT3") == 0 || strcmp(argv[2], "float3") == 0) mode =
                MODE_FLOAT3;
225         else if (strcmp(argv[2], "FLOAT") == 0 || strcmp(argv[2], "float") == 0) mode = MODE_FLOAT
                ;
226         else if (strcmp(argv[2], "2D") == 0 || strcmp(argv[2], "2d") == 0) mode = MODE_2D;
227         else
228         {
229             printf("'%s' was not converted to a mode.\n", argv[1]);
230             return -1;
231         }
232
233         if (mode != MODE_CPU)
234         {
235             cl_platform_id platform = ocl_select_platform();
236             cl_device_id device = ocl_select_device(platform);
237             init_ocl(device);
238             create_program("kernel.cl", "");
239         }
240
241         char* title = malloc(1 + (size_t) snprintf(NULL, 0, "nBody problem - %ld bodies", n));
242         sprintf(title, "nBody problem - %ld bodies", n);
243         renderer_init(argc, argv, title);
244
245         srand((unsigned int) time(NULL));
246         Body* bodies = create_bodies((size_t) n);
247
248         // Create kernel
249         cl_int error;
250         cl_kernel kernel;
251         cl_mem dev_bodies;
252         switch (mode)
253         {
254             case MODE_CPU:
255                 kernel = NULL;
256                 dev_bodies = NULL;
257                 break;
258             case MODE_FLOAT3:
259                 kernel = clCreateKernel(g_program, "kernel_step_float3", &error);
260                 break;
261             case MODE_FLOAT:
262                 kernel = clCreateKernel(g_program, "kernel_step_float", &error);
263                 break;
264             case MODE_2D:
265                 kernel = clCreateKernel(g_program, "kernel2_step_float3", &error);
266                 break;
267             default:
268                 abort();
269         }
270         if (kernel != NULL)
271         {
272             ocl_err(error);
273             ocl_err(clFinish(g_command_queue));
274             // Data buffer on GPU (RW)
275             dev_bodies = clCreateBuffer(g_context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
                    sizeof(Body) * n, bodies, &error);
276             ocl_err(error);
277             ocl_err(clFinish(g_command_queue));
278
279             // Set kernel arguments
280             cl_uint arg_num = 0;
281             ocl_err(clSetKernelArg(kernel, arg_num++, sizeof(n), &n));
```

```
282         ocl_err(clSetKernelArg(kernel, arg_num++, sizeof(dev_bodies), &dev_bodies));
283         ocl_err(clFinish(g_command_queue));
284     }
285
286     CallbackData cd = {
287             kernel,                 /* cl_kernel kernel */
288             (size_t) n,             /* size_t n */
289             bodies,                 /* Body* bodies */
290             dev_bodies,             /* cl_mem dev_bodies */
291             {0.7, 0.7, 1, 1},       /* Vect4f point_color */
292             2,                      /* float point_size */
293             true,                   /* bool draw_lines */
294             {0.7, 0.7, 0.7, 0.2},   /* Vect4f line_color */
295             mode,                   /* Mode */
296     };
297
298     renderer_start(&cd, step, draw);
299
300     return 0;
301 }
```

3.3 lib/renderer.h

```
1  #ifndef _RENDERER_H_
2  #define _RENDERER_H_
3
4  #include <stdlib.h>
5  #include <stdio.h>
6  #include <stdbool.h>
7  #include <string.h>
8  #include <errno.h>
9  #include <math.h>
10 #include <time.h>
11 #include <GL/glut.h>
12
13 /* Types */
14 typedef struct {
15     int x, y;
16 } Vect2i;
17
18 typedef struct {
19     union {
20         struct {double x, y, z;};
21         struct {double r, g, b;};
22     };
23 } Vect3d;
24
25 typedef struct {
26     union {
27         struct {float x, y, z;};
28         struct {float r, g, b;};
29     };
30 } Vect3f;
31
32 typedef struct {
33     union {
34         struct {double x, y, z, w;};
35         struct {double r, g, b, a;};
36     };
37 } Vect4d;
38
39 typedef struct {
40     union {
41         struct {float x, y, z, w;};
42         struct {float r, g, b, a;};
43     };
44 } Vect4f;
45
46 typedef struct {
47     Vect3d pos;
48     double yaw, pitch;
49 } Camera;
50
51 extern Camera camera;
52
53 void renderer_init(int argc, char **pString, char* title);
```

```
54 void renderer_start(void *data, void (*callback_step)(void *), void (*callback_draw)(void *));
55
56 #endif
```

3.4 lib/renderer.c

```
1  #include "renderer.h"
2
3  Camera camera = {
4          {15, 15, 15},
5          -45, -30
6  };
7  static void* callback_data;
8  static void (*callback_step_func)(void *);
9  static void (*callback_draw_func)(void *);
10 static bool mouseLeftDown = false;
11 static bool mouseRightDown = false;
12 static bool mouseMiddleDown = false;
13 static double mouseZoomDiv = 10;
14 static double mouseRotateDiv = 2.5;
15 static double mousePanDiv = 10;
16 static Vect2i prevMouse = {0, 0};
17
18 static void reshape(int w, int h);
19 static void drawAxis(double size);
20 static void display(void);
21 static void idle();
22 static void moveCamera(double forwards, double strafe, double yaw, double pitch, double x,
       double y, double z);
23 static void mouse(int button, int state, int x, int y);
24 static void motion(int x, int y);
25 static void keyboard(unsigned char key, int x, int y);
26
27 static void reshape(int w, int h)
28 {
29     glMatrixMode(GL_PROJECTION);
30     glLoadIdentity();
31     gluPerspective(90, (double) w / h, 0.1, 1000);
32     glViewport(0, 0, w, h);
33 }
34
35 static void drawAxis(double size)
36 {
37     glPushMatrix();
38     glPushAttrib(GL_LINE_STIPPLE);
39     glPushAttrib(GL_LINE_STIPPLE_PATTERN);
40     glPushAttrib(GL_LINE_STIPPLE_REPEAT);
41
42     glBegin(GL_LINES);
43     glColor3f(1, 0, 0); glVertex3d(0, 0, 0); glVertex3d(size, 0, 0);
44     glColor3f(0, 1, 0); glVertex3d(0, 0, 0); glVertex3d(0, size, 0);
45     glColor3f(0, 0, 1); glVertex3d(0, 0, 0); glVertex3d(0, 0, size);
46     glEnd();
47
48     glLineStipple((int) (size / 100), 0xAAAA);
49     glEnable(GL_LINE_STIPPLE);
50     glBegin(GL_LINES);
51     glColor3f(1, 0, 0); glVertex3d(0, 0, 0); glVertex3d(-size, 0, 0);
52     glColor3f(0, 1, 0); glVertex3d(0, 0, 0); glVertex3d(0, -size, 0);
53     glColor3f(0, 0, 1); glVertex3d(0, 0, 0); glVertex3d(0, 0, -size);
54     glEnd();
55
56     glPopAttrib();/* GL_LINE_STIPPLE_REPEAT */
57     glPopAttrib();/* GL_LINE_STIPPLE_PATTERN */
58     glPopAttrib();/* GL_LINE_STIPPLE */
59     glPopMatrix();
60 }
61
62 static void display(void)
63 {
64     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); /* Nuke everything */
65     glMatrixMode(GL_MODELVIEW);
66     glLoadIdentity();
67
68     glRotated(-camera.pitch, 1, 0, 0);
69     glRotated(camera.yaw, 0, 1, 0);
```

```
70      glTranslated(-camera.pos.x, -camera.pos.y, -camera.pos.z);
71
72  //      drawAxis(1000);
73  //      drawCheckers(1, 100);
74
75      callback_draw_func(callback_data);
76
77      glutSwapBuffers();
78  }
79
80  static void idle()
81  {
82      callback_step_func(callback_data);
83      glutPostRedisplay();
84  }
85
86  static void moveCamera(double forwards, double strafe, double yaw, double pitch, double x,
        double y, double z)
87  {
88      if (yaw != 0)
89      {
90          camera.yaw += yaw;
91          if (camera.yaw > 180) camera.yaw -= 360;
92          if (camera.yaw < -180) camera.yaw += 360;
93      }
94      if (pitch != 0)
95      {
96          camera.pitch += pitch;
97          if (camera.pitch > 90) camera.pitch = 90;
98          if (camera.pitch < -90) camera.pitch = -90;
99      }
100     if (x != 0 || y != 0 || z != 0)
101     {
102         camera.pos.x += x;
103         camera.pos.y += y;
104         camera.pos.z += z;
105     }
106     if (forwards != 0)
107     {
108         double dz = -cos(camera.yaw*M_PI/180.0);
109         double dy = sin(camera.pitch*M_PI/180.0);
110         double dx = sin(camera.yaw*M_PI/180.0);
111         camera.pos.x += dx * forwards;
112         camera.pos.y += dy * forwards;
113         camera.pos.z += dz * forwards;
114     }
115     if (strafe != 0)
116     {
117         double dz = sin(camera.yaw*M_PI/180.0);
118         double dx = cos(camera.yaw*M_PI/180.0);
119         camera.pos.x += dx * strafe;
120         camera.pos.z += dz * strafe;
121     }
122     glutPostRedisplay();
123 }
124
125 static void mouse(int button, int state, int x, int y)
126 {
127     prevMouse.x = x;
128     prevMouse.y = y;
129
130     switch (button)
131     {
132         default: return;
133         case GLUT_LEFT_BUTTON: mouseLeftDown = state == GLUT_DOWN; break;
134         case GLUT_RIGHT_BUTTON: mouseRightDown = state == GLUT_DOWN; break;
135         case GLUT_MIDDLE_BUTTON: mouseMiddleDown = state == GLUT_DOWN; break;
136         case 3:
137         case 4:
138             // Scroll wheel
139             moveCamera(button == 3 ? 1 : -1, 0, 0, 0, 0, 0, 0); // zoom
140     }
141 }
142
143 static void motion(int x, int y)
144 {
```

```
145     double dx = prevMouse.x - x;
146     double dy = prevMouse.y - y;
147     prevMouse.x = x;
148     prevMouse.y = y;
149 
150     if (mouseLeftDown) moveCamera(0, dx/mouseZoomDiv, 0, 0, 0, dy/mousePanDiv, 0); // Y & strafe
151     if (mouseRightDown) moveCamera(0, 0, -dx/mouseRotateDiv, dy/mouseRotateDiv, 0, 0, 0); // yaw & pitch
152     if (mouseMiddleDown) moveCamera(0, 0, 0, 0, dx/mousePanDiv, 0, dy/mousePanDiv); // pan X Z
153 }
154 
155 static void keyboard(unsigned char key, int x, int y)
156 {
157     switch (key)
158     {
159         default: return;
160         case 27: exit(0);
161     }
162 }
163 
164 void renderer_init(int argc, char **argv, char* title)
165 {
166     glutInit(&argc, argv);
167     glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH | GLUT_ALPHA);
168     glutInitWindowSize(900 * 16/9, 900);
169     glutCreateWindow(title);
170 
171     glClearColor(0, 0, 0, 0);
172     glClearDepth(1.0);
173     glEnable(GL_DEPTH_TEST);
174     glEnable(GL_BLEND);
175     glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
176 
177     glutReshapeFunc(reshape);
178     glutDisplayFunc(display);
179     glutKeyboardFunc(keyboard);
180 
181     glutMouseFunc(mouse);
182     glutMotionFunc(motion);
183     glutSetCursor(GLUT_CURSOR_INFO);
184 }
185 
186 void renderer_start(void *data, void (*callback_step)(void *), void (*callback_draw)(void *))
187 {
188     callback_data = data;
189     callback_step_func = callback_step;
190     callback_draw_func = callback_draw;
191 
192     glutIdleFunc(idle);
193 
194     glutMainLoop();
195 }
```