Laboverslag n-body probleem

Dries Kennes & Stijn Van Dessel

May 14, 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 geparallelliseerde code, om de resultaten vergelijkbaar te maken.

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.

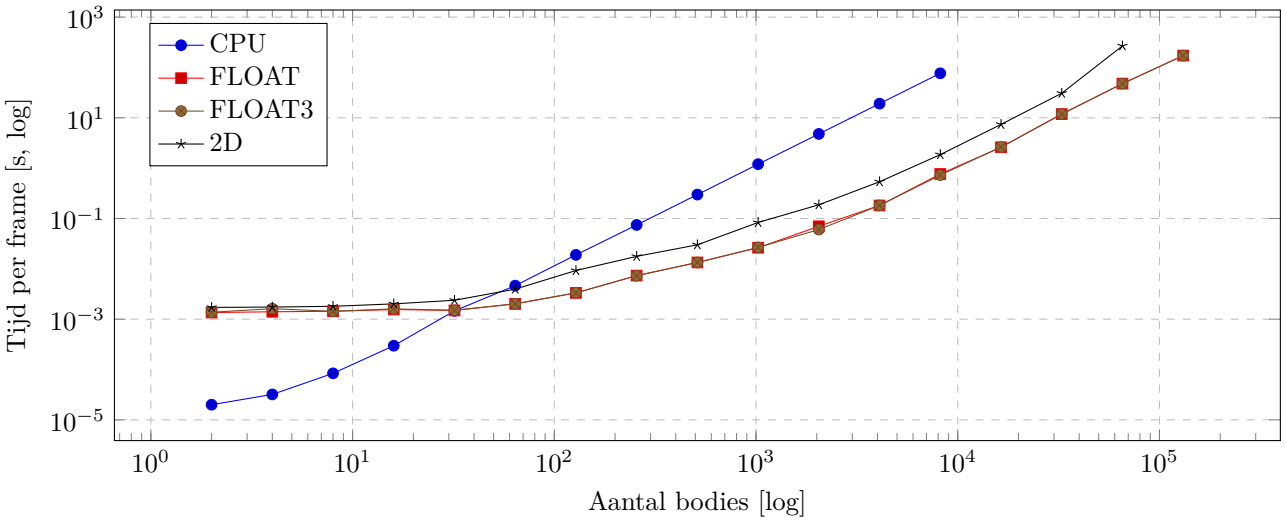
De code voor deze kernel is te vinden in de bijlage 3.1 (kernel.cl), regels 90 t.e.m. 122.

2 Resultaten

De tijden zijn bekomen op een Intel i5-4670K (4GHz) met 16GB RAM en een NVIDIA GeForce GTX 1060 6GB, Linux 4.16.8 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   * (c) Dries Kennes & Stijn Van Dessel 2018
3   *
4   * nBody problem (OpenCL kernels)
5   */
6  #ifdef FOEFELMAGIC
7  // CMakeLists.txt's 'add_definitions(-DFOEFELMAGIC)' makes syntax highlighting possible.
8  #define __global
9  #define __kernel
10 #define float3 struct {float x, y, z;}
11 #define get_global_id(dim) dim
12 #define sqrt(a) a
13 #define CLK_GLOBAL_MEM_FENCE 0
14 #define barrier(type) do {} while(0)
15 #endif
16
17 // Required for AtomicAdd
18 typedef union
19 {
20     float3 vec;
21     float arr[3];
22 } float3_;
23 // Based on: http://suhorukov.blogspot.be/2011/12/opencl-11-atomic-operations-on-floating.html
24 inline void AtomicAdd(volatile __global float *source, const float operand) {
25     union {
26         unsigned int intVal;
27         float floatVal;
28     } newVal;
29     union {
30         unsigned int intVal;
31         float floatVal;
32     } prevVal;
33     do {
34         prevVal.floatVal = *source;
35         newVal.floatVal = prevVal.floatVal + operand;
36     } while (atomic_cmpxchg((volatile __global unsigned int *)source, prevVal.intVal,
37                             newVal.intVal) != prevVal.intVal);
38 }
39
40 // Our data type (must be kept in sync with main program code)
41 typedef struct body
42 {
43     float3 pos;
44     float3_ speed;
45 } Body;
46
47 /** KERNEL FLOAT */
48 __kernel void kernel_step_float(unsigned long n, __global Body * data)
49 {
50     const float SCALE = 50;
51     const float TIME_DELTA = 1;
52     const float GRAV_CONST = 1;
53     const float MASS = 2;
54     const float MASS_GAV = (GRAV_CONST * MASS * MASS);
55
56     const int body = get_global_id(0);
57
58     if (body >= n) return;
59
60     __global Body* us = &data[body];
61
62     for (int i = 0; i < n; i++)
63     {
64         if (i == body) continue;
65
66         __global const Body* them = &data[i];
67
68         float dist_x = (us->pos.x - them->pos.x) * SCALE;
69         float dist_y = (us->pos.y - them->pos.y) * SCALE;
70         float dist_z = (us->pos.z - them->pos.z) * SCALE;
71
72         float dist = sqrt(dist_x * dist_x + dist_y * dist_y + dist_z * dist_z);
```

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

```
149     float3 dist_all_sq = dist_all * dist_all;
150     float dist = sqrt(dist_all_sq.x + dist_all_sq.y + dist_all_sq.z);
151
152     float3 acc = (-MASS_GAV * dist_all / (dist * dist * dist)) / MASS;
153
154     AtomicAdd(&us->speed.arr[0], acc.x * TIME_DELTA);
155     AtomicAdd(&us->speed.arr[1], acc.y * TIME_DELTA);
156     AtomicAdd(&us->speed.arr[2], acc.z * TIME_DELTA);
157 }
```

3.2 main.c

```
1  /**
2   * (c) Dries Kennes & Stijn Van Dessel 2018
3   *
4   * nBody problem (Main program)
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(
76                      cd->bodies[i].pos.x + cd->bodies[i].speed.x / 2,
77                      cd->bodies[i].pos.y + cd->bodies[i].speed.y / 2,
78                      cd->bodies[i].pos.z + cd->bodies[i].speed.z / 2);
79          }
80          glEnd();
81      }
82  }
83
84  struct timespec diff(struct timespec start, struct timespec end)
85  {
86      struct timespec temp;
87      if ((end.tv_nsec-start.tv_nsec)<0) {
88          temp.tv_sec = end.tv_sec-start.tv_sec-1;
89          temp.tv_nsec = 1000000000+end.tv_nsec-start.tv_nsec;
90      } else {
91          temp.tv_sec = end.tv_sec-start.tv_sec;
92          temp.tv_nsec = end.tv_nsec-start.tv_nsec;
93      }
94      return temp;
95  }
96
97  void kernel_cpu(CallbackData* cd)
98  {
99      const float delta_time = 1.f;
100     const float grav_constant = 1;
101     const float mass_of_sun = 2;
102     const float mass_grav = grav_constant * mass_of_sun * mass_of_sun;
103     const float scale = 50;
104
105     for (int i = 0; i < cd->n; ++i)
106     {
107         for (int j = 0; j < cd->n; ++j)
108         {
109             if (i == j) continue;
110
111             cl_float3 pos_a = cd->bodies[i].pos;
112             cl_float3 pos_b = cd->bodies[j].pos;
113
114             float dist_x = (pos_a.s[0] - pos_b.s[0]) * scale;
115             float dist_y = (pos_a.s[1] - pos_b.s[1]) * scale;
116             float dist_z = (pos_a.s[2] - pos_b.s[2]) * scale;
117
118             float distance = sqrt(
119                     dist_x * dist_x +
120                     dist_y * dist_y +
121                     dist_z * dist_z);
122
123             float force_x = -mass_grav * dist_x / (distance * distance * distance);
124             float force_y = -mass_grav * dist_y / (distance * distance * distance);
125             float force_z = -mass_grav * dist_z / (distance * distance * distance);
126
127             float acc_x = force_x / mass_of_sun;
128             float acc_y = force_y / mass_of_sun;
129             float acc_z = force_z / mass_of_sun;
130
131             cd->bodies[i].speed.s[0] += acc_x * delta_time;
132             cd->bodies[i].speed.s[1] += acc_y * delta_time;
133             cd->bodies[i].speed.s[2] += acc_z * delta_time;
134         }
135     }
136
137     for (int i = 0; i < cd->n; ++i)
138     {
139         cd->bodies[i].pos.s[0] += (cd->bodies[i].speed.s[0] * delta_time) / scale;
140         cd->bodies[i].pos.s[1] += (cd->bodies[i].speed.s[1] * delta_time) / scale;
```

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

```
212             printf("- FLOAT\n");
213             printf("- FLOAT3\n");
214             printf("- 2D\n");
215             return -1;
216         }
217         errno = 0;
218         cl_ulong n = strtoul(argv[1], NULL, 0);
219         if (errno != 0)
220         {
221             printf("'%s' was not converted to a proper number. (%ld)\n", argv[1], n);
222             return -1;
223         }
224         enum MODE mode;
225         if (strcmp(argv[2], "CPU") == 0 || strcmp(argv[2], "cpu") == 0)
226         {
227             mode = MODE_CPU;
228         }
229         else if (strcmp(argv[2], "FLOAT3") == 0 || strcmp(argv[2], "float3") == 0)
230         {
231             mode = MODE_FLOAT3;
232         }
233         else if (strcmp(argv[2], "FLOAT") == 0 || strcmp(argv[2], "float") == 0)
234         {
235             mode = MODE_FLOAT;
236         }
237         else if (strcmp(argv[2], "2D") == 0 || strcmp(argv[2], "2d") == 0)
238         {
239             mode = MODE_2D;
240         }
241         else
242         {
243             printf("'%s' was not converted to a mode.\n", argv[1]);
244             return -1;
245         }
246
247         if (mode != MODE_CPU)
248         {
249             cl_platform_id platform = ocl_select_platform();
250             cl_device_id device = ocl_select_device(platform);
251             init_ocl(device);
252             create_program("kernel.cl", "");
253         }
254
255         char* title = malloc(1 + (size_t) snprintf(NULL, 0, "nBody problem - %ld bodies", n));
256         sprintf(title, "nBody problem - %ld bodies", n);
257         renderer_init(argc, argv, title);
258
259         srand((unsigned int) time(NULL));
260         Body* bodies = create_bodies((size_t) n);
261
262         // Create kernel
263         cl_int error = 0;
264         cl_kernel kernel = NULL;
265         cl_mem dev_bodies;
266         switch (mode)
267         {
268             case MODE_CPU:
269                 kernel = NULL;
270                 dev_bodies = NULL;
271                 break;
272             case MODE_FLOAT3:
273                 kernel = clCreateKernel(g_program, "kernel_step_float3", &error);
274                 break;
275             case MODE_FLOAT:
276                 kernel = clCreateKernel(g_program, "kernel_step_float", &error);
277                 break;
278             case MODE_2D:
279                 kernel = clCreateKernel(g_program, "kernel2_step_float3", &error);
280                 break;
281             default:
282                 abort();
283         }
284         if (kernel != NULL)
285         {
286             ocl_err(error);
287             ocl_err(clFinish(g_command_queue));
```

```
288         // Data buffer on GPU (RW)
289         dev_bodies = clCreateBuffer(
290                 g_context,
291                 CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
292                 sizeof(Body) * n,
293                 bodies,
294                 &error);
295         ocl_err(error);
296         ocl_err(clFinish(g_command_queue));
297 
298         // Set kernel arguments
299         cl_uint arg_num = 0;
300         ocl_err(clSetKernelArg(kernel, arg_num++, sizeof(n), &n));
301         ocl_err(clSetKernelArg(kernel, arg_num++, sizeof(dev_bodies), &dev_bodies));
302         ocl_err(clFinish(g_command_queue));
303     }
304 
305     CallbackData cd = {
306             kernel,                 /* cl_kernel kernel */
307             (size_t) n,             /* size_t n */
308             bodies,                 /* Body* bodies */
309             dev_bodies,             /* cl_mem dev_bodies */
310             {0.7, 0.7, 1, 1},       /* Vect4f point_color */
311             2,                      /* float point_size */
312             true,                   /* bool draw_lines */
313             {0.7, 0.7, 0.7, 0.2},   /* Vect4f line_color */
314             mode,                   /* Mode mode */
315     };
316 
317     renderer_start(&cd, step, draw);
318 
319     return 0;
320 }
```

3.3 lib/renderer.h

```
1  #ifndef _RENDERER_H_
2  #define _RENDERER_H_
3  /**
4   * (c) Dries Kennes 2018
5   *
6   * Rendering helper. Largely a copy of my personal OpenGL project.
7   */
8  /* Handy includes for everything */
9  #include <stdlib.h>
10 #include <stdio.h>
11 #include <stdbool.h>
12 #include <string.h>
13 #include <errno.h>
14 #include <math.h>
15 #include <time.h>
16 #include <GL/glut.h>
17 
18 /* Types */
19 typedef struct {
20     int x, y;
21 } Vect2i;
22 
23 typedef struct {
24     union {
25         struct {double x, y, z;};
26         struct {double r, g, b;};
27     };
28 } Vect3d;
29 
30 typedef struct {
31     union {
32         struct {float x, y, z, w;};
33         struct {float r, g, b, a;};
34     };
35 } Vect4f;
36 
37 typedef struct {
38     Vect3d pos;
39     double yaw, pitch;
40 } Camera;
```

```
41
42 /* Public global variables */
43 extern Camera camera;
44
45 /* Public 'API' */
46 void renderer_init(int argc, char **pString, char* title);
47 void renderer_start(void *data, void (*callback_step)(void *), void (*callback_draw)(void *));
48
49 #endif
```

3.4 lib/renderer.c

```
1  #include "renderer.h"
2  /**
3   * (c) Dries Kennes 2018
4   *
5   * Rendering helper. Largely a copy of my personal OpenGL project.
6   */
7  /* Public global variables */
8  Camera camera = {
9          {15, 15, 15},
10         -45, -30
11 };
12 /* Private (global) variables */
13 static void* callback_data;
14 static void (*callback_step_func)(void *);
15 static void (*callback_draw_func)(void *);
16 static bool mouseLeftDown = false;
17 static bool mouseRightDown = false;
18 static bool mouseMiddleDown = false;
19 static double mouseZoomDiv = 10;
20 static double mouseRotateDiv = 2.5;
21 static double mousePanDiv = 10;
22 static Vect2i prevMouse = {0, 0};
23
24 /* Private functions */
25 static void reshape(int w, int h);
26 static void display(void);
27 static void idle();
28 static void moveCamera(double forwards, double strafe, double yaw, double pitch, double x,
       double y, double z);
29 static void mouse(int button, int state, int x, int y);
30 static void motion(int x, int y);
31 static void keyboard(unsigned char key, int x, int y);
32
33 /* Private functions */
34 static void reshape(int w, int h)
35 {
36     glMatrixMode(GL_PROJECTION);
37     glLoadIdentity();
38     gluPerspective(90, (double) w / h, 0.1, 1000);
39     glViewport(0, 0, w, h);
40 }
41
42 static void display(void)
43 {
44     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); /* Nuke everything */
45     glMatrixMode(GL_MODELVIEW);
46     glLoadIdentity();
47
48     glRotated(-camera.pitch, 1, 0, 0);
49     glRotated(camera.yaw, 0, 1, 0);
50     glTranslated(-camera.pos.x, -camera.pos.y, -camera.pos.z);
51
52     callback_draw_func(callback_data);
53
54     glutSwapBuffers();
55 }
56
57 static void idle()
58 {
59     callback_step_func(callback_data);
60     glutPostRedisplay();
61 }
62
63 static void moveCamera(double forwards, double strafe, double yaw, double pitch, double x,
```

```
    double y, double z)
{
    if (yaw != 0)
    {
        camera.yaw += yaw;
        if (camera.yaw > 180) camera.yaw -= 360;
        if (camera.yaw < -180) camera.yaw += 360;
    }
    if (pitch != 0)
    {
        camera.pitch += pitch;
        if (camera.pitch > 90) camera.pitch = 90;
        if (camera.pitch < -90) camera.pitch = -90;
    }
    if (x != 0 || y != 0 || z != 0)
    {
        camera.pos.x += x;
        camera.pos.y += y;
        camera.pos.z += z;
    }
    if (forwards != 0)
    {
        double dz = -cos(camera.yaw*M_PI/180.0);
        double dy = sin(camera.pitch*M_PI/180.0);
        double dx = sin(camera.yaw*M_PI/180.0);
        camera.pos.x += dx * forwards;
        camera.pos.y += dy * forwards;
        camera.pos.z += dz * forwards;
    }
    if (strafe != 0)
    {
        double dz = sin(camera.yaw*M_PI/180.0);
        double dx = cos(camera.yaw*M_PI/180.0);
        camera.pos.x += dx * strafe;
        camera.pos.z += dz * strafe;
    }
    glutPostRedisplay();
}

static void mouse(int button, int state, int x, int y)
{
    prevMouse.x = x;
    prevMouse.y = y;

    switch (button)
    {
        default: return;
        case GLUT_LEFT_BUTTON: mouseLeftDown = state == GLUT_DOWN; break;
        case GLUT_RIGHT_BUTTON: mouseRightDown = state == GLUT_DOWN; break;
        case GLUT_MIDDLE_BUTTON: mouseMiddleDown = state == GLUT_DOWN; break;
        case 3:
        case 4:
            // Scroll wheel
            moveCamera(button == 3 ? 1 : -1, 0, 0, 0, 0, 0, 0); // zoom
    }
}

static void motion(int x, int y)
{
    double dx = prevMouse.x - x;
    double dy = prevMouse.y - y;
    prevMouse.x = x;
    prevMouse.y = y;

    if (mouseLeftDown)
    {
        moveCamera(0, dx/mouseZoomDiv, 0, 0, 0, dy/mousePanDiv, 0); // Y & strafe
    }
    if (mouseRightDown)
    {
        moveCamera(0, 0, -dx/mouseRotateDiv, dy/mouseRotateDiv, 0, 0, 0); // yaw & pitch
    }
    if (mouseMiddleDown)
    {
        moveCamera(0, 0, 0, 0, dx/mousePanDiv, 0, dy/mousePanDiv); // pan X Z
    }
```

```
139 }
140
141 static void keyboard(unsigned char key, int x, int y)
142 {
143     switch (key)
144     {
145         default: return;
146         case 27: exit(0);
147     }
148 }
149
150 /* Public 'API' */
151 void renderer_init(int argc, char **argv, char* title)
152 {
153     glutInit(&argc, argv);
154     glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH | GLUT_ALPHA);
155     glutInitWindowSize(900 * 16/9, 900);
156     glutCreateWindow(title);
157
158     glClearColor(0, 0, 0, 0);
159     glClearDepth(1.0);
160     glEnable(GL_DEPTH_TEST);
161     glEnable(GL_BLEND);
162     glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
163
164     glutReshapeFunc(reshape);
165     glutDisplayFunc(display);
166     glutKeyboardFunc(keyboard);
167
168     glutMouseFunc(mouse);
169     glutMotionFunc(motion);
170     glutSetCursor(GLUT_CURSOR_INFO);
171 }
172
173 void renderer_start(void *data, void (*callback_step)(void *), void (*callback_draw)(void *))
174 {
175     callback_data = data;
176     callback_step_func = callback_step;
177     callback_draw_func = callback_draw;
178
179     glutIdleFunc(idle);
180
181     glutMainLoop();
182 }
```