

```

1  /*****
2  /* scene.h */
3  *****/
4
5  #pragma once
6  #include <vec.h>
7  //typedef struct{} sphere;
8  struct sphere;
9  struct plane;
10 typedef struct _rt_ctx raytracer_context;
11
12 typedef W_ALIGN(16) struct
13 {
14     vec4    colour;
15
16     float    reflectivity;
17
18     //TODO: add more.
19 } U_ALIGN(16) material;
20
21 /*typedef struct
22 {
23     vec3    max;
24     vec3    min;
25 } AABB;
26 */
27 typedef W_ALIGN(32) struct
28 {
29     mat4    model;
30
31     vec4    max;
32     vec4    min;
33
34     int     index_offset;
35     int     num_indices;
36
37     int     material_index;
38 } U_ALIGN(32) mesh;
39
40 typedef struct
41 {
42
43     mat4    camera_world_matrix;
44
45     //Materials
46     material* materials;
47     cl_mem  cl_material_buffer;
48     unsigned int num_materials;
49     bool    materials_changed;
50     //Primitives
51
52     //Spheres
53     sphere* spheres;
54     cl_mem  cl_sphere_buffer;
55     unsigned int num_spheres; //NOTE: must be constant.
56     bool    spheres_changed;
57     //Planes
58     plane* planes;
59     cl_mem  cl_plane_buffer;
60     unsigned int num_planes; //NOTE: must be constant.
61     bool    planes_changed;
62
63     //Meshes
64     mesh* meshes; //ALL vertex data is stored contiguously
65     cl_mem  cl_mesh_buffer;
66     unsigned int num_meshes;
67     bool    meshes_changed;
68
69
70     //NOTE: we could store vertices, normals, and texcoords contiguously as 1 buffer.
71     vec3* mesh_verts;
72     cl_mem  cl_mesh_vert_buffer;
73     unsigned int num_mesh_verts; //NOTE: must be constant.
74
75     vec3* mesh_nrmls;
76     cl_mem  cl_mesh_nrml_buffer;
77     unsigned int num_mesh_nrmls; //NOTE: must be constant.
78
79     vec2* mesh_texcoords;
80     cl_mem  cl_mesh_texcoord_buffer;
81     unsigned int num_mesh_texcoords; //NOTE: must be constant.
82
83     ivec3* mesh_indices;
84     cl_mem  cl_mesh_index_buffer;
85     unsigned int num_mesh_indices; //NOTE: must be constant.
86

```

```

87 } scene;
88
89
90 void scene_resource_push(raytracer_context*);
91 void scene_init_resources(raytracer_context*);
92
93
94 /*****/
95 /* win32.h */
96 /*****/
97
98 #pragma once
99 #include <windows.h>
100 #include <stdbool.h>
101 #include <os_abs.h>
102
103 typedef struct
104 {
105     HINSTANCE instance;
106     int nCmdShow;
107     WNDCLASSEX wc;
108     HWND win;
109
110     int width, height;
111
112     BITMAPINFO bitmap_info;
113     void* bitmap_memory;
114
115     // HDC render_device_context;
116
117     bool shouldRun;
118     //Bitbuffer
119 } win32_context;
120
121
122 os_abs init_win32_abs();
123
124 void win32_start_thread(void (*func)(void*), void* data);
125
126 //void create_win32_window();
127 void win32_start();
128 void win32_loop();
129
130 void win32_update();
131
132 void win32_sleep(int);
133
134 void* win32_get_bitmap_memory();
135
136 int win32_get_time_mili();
137
138 int win32_get_width();
139 int win32_get_height();
140
141 /*****/
142 /* startup.h */
143 /*****/
144
145 #pragma once
146
147 int startup();
148 void loop_exit();
149 void loop_pause();
150
151 /*****/
152 /* geom.h */
153 /*****/
154
155 #pragma once
156 #include <stdbool.h>
157
158 typedef int ivec3[4]; //1 int padding
159 typedef float vec2[2];
160 typedef float vec3[4]; //1 float padding
161 typedef float vec4[4];
162 typedef float mat4[16];
163
164 /*****/
165 /* Ray */
166 /*****/
167 typedef struct ray
168 {
169     vec3 orig;
170     vec3 dir;
171     //float t_min, t_max;
172 } ray;
173

```

```

174
175 /*****/
176 /* Sphere */
177 /*****/
178
179 //NOTE: Less memory efficient but aligns with opengl
180 typedef W_ALIGN(16) struct sphere
181 {
182     vec4 pos; //GPU stores all vec3s as vec4s in memory so we need the padding.
183
184     float radius;
185     int material_index;
186
187 } U_ALIGN(16) sphere;
188
189
190 float does_collide_sphere(sphere, ray);
191
192
193 /*****/
194 /* Plane */
195 /*****/
196
197 typedef W_ALIGN(16) struct plane // bytes
198 {
199     vec4 pos; //12
200     //float test;
201
202     vec4 norm;
203     //float test2;
204
205     //32
206     int material_index;
207 } U_ALIGN(16) plane;
208 float does_collide_plane(plane, ray);
209
210 ray generate_ray(int x, int y, int width, int height, float fov);
211 float* matvec_mul(mat4 m, vec4 v);
212
213 /*****/
214 /* irradiance_cache.h */
215 /*****/
216
217 /*****/
218 /* NOTE: Irradiance Caching is Incomplete */
219 /*****/
220
221 #pragma once
222 #include <stdint.h>
223
224 #define NUM_MIPMAPS 4 //NOTE: 1080/(2^4) != integer
225
226 typedef struct _rt_ctx raytracer_context;
227
228 // 0 = 000: x-, y-, z-
229 // 1 = 001: x-, y-, z+
230 // 2 = 010: x-, y+, z-
231 // 3 = 011: x-, y+, z+
232 // 4 = 100: x+, y-, z-
233 // 5 = 101: x+, y-, z+
234 // 6 = 110: x+, y+, z-
235 // 7 = 111: x+, y+, z+
236
237 typedef struct
238 {
239     vec3 point;
240     vec3 normal;
241
242     float rad;
243
244     vec3 col;
245
246     vec3 gpos;
247     vec3 gdir;
248 } ic_ir_value;
249
250 typedef struct _ic_octree_node ic_octree_node;
251
252 struct _ic_octree_node
253 {
254     bool leaf;
255     bool active;
256
257     union
258     {
259         struct
260         {

```

```

261         unsigned int buffer_offset;
262         unsigned int num_elems;
263     } leaf;
264     struct
265     {
266         ic_octree_node* children[8];
267     } branch;
268     } data;
269     vec3 min;
270     vec3 max;
271 };
272 };
273
274 typedef struct
275 {
276     ic_octree_node* root;
277     int node_count;
278     unsigned int width;
279     unsigned int max_depth;
280 } ic_octree;
281
282 typedef struct
283 {
284     //vec4* texture;
285     cl_mem cl_image_ref;
286     unsigned int width, height;
287 } ic_mipmap_gb;
288
289 typedef struct
290 {
291     //float* texture;
292     cl_mem cl_image_ref;
293     unsigned int width, height;
294 } ic_mipmap_f;
295
296 typedef struct
297 {
298
299     cl_image_format cl_standard_format;
300     cl_image_desc cl_standard_descriptor;
301     ic_octree octree;
302     ic_ir_value* ir_buf;
303     unsigned int ir_buf_size;
304     unsigned int ir_buf_current_offset;
305 } ic_context;
306
307 void ic_init(raytracer_context*);
308 void ic_screenspace(raytracer_context*);
309 void ic_octree_init_branch(ic_octree_node*);
310 void ic_octree_insert(ic_context*, vec3 point, vec3 normal);
311
312
313
314 /*****/
315 /* loader.h */
316 /*****/
317 #pragma once
318 #include <scene.h>
319
320 scene* load_scene_json(char* data);
321 scene* load_scene_json_url(char* url);
322
323
324
325 /*****/
326 /* os_abs.h */
327 /*****/
328
329 #pragma once
330
331 typedef struct
332 {
333     void (*start_func)();
334     void (*loop_start_func)();
335     void (*update_func)();
336     void (*sleep_func)(int);
337     void* (*get_bitmap_memory_func)();
338     int (*get_time_mili_func)();
339     int (*get_width_func)();
340     int (*get_height_func)();
341     void (*start_thread_func)(void (*func)(void*), void* data);
342 } os_abs;
343
344 void os_start(os_abs);
345 void os_loop_start(os_abs);
346 void os_update(os_abs);
347 void os_sleep(os_abs, int);

```

```

348 void* os_get_map_memory(os_abs);
349 int os_get_time_mili(os_abs);
350 int os_get_width(os_abs);
351 int os_get_height(os_abs);
352 void os_start_thread(os_abs, void (*func)(void*), void* data);
353
354
355
356 /*****
357  * parallel.h *
358  *****/
359 #pragma once
360 #include <CL/opencl.h>
361 #include <geom.h>
362 typedef struct _rt_ctx raytracer_context;
363
364 typedef struct
365 {
366     cl_platform_id platform_id;
367     cl_device_id device_id;           // compute device id
368     cl_context context;               // compute context
369     cl_command_queue commands;        // compute command queue
370 } rcl_ctx;
371
372
373 typedef struct
374 {
375     cl_program program;
376     cl_kernel* raw_kernels; //NOTE: not a good solution
377     char* raw_data;
378 } rcl_program;
379
380
381 void cl_info();
382 void create_context(rcl_ctx* context);
383 void load_program_raw(rcl_ctx* ctx, char* data, char** kernels, unsigned int num_kernels,
384                     rcl_program* program, char** macros, unsigned int num_macros);
385 void load_program_url(rcl_ctx* ctx, char* url, char** kernels, unsigned int num_kernels,
386                     rcl_program* program, char** macros, unsigned int num_macros);
387 void test_sphere_raytracer(rcl_ctx* ctx, rcl_program* program,
388                          sphere* spheres, int num_spheres,
389                          uint32_t* bitmap, int width, int height);
390 cl_mem gen_rgb_image(raytracer_context* rctx,
391                     const unsigned int width,
392                     const unsigned int height);
393 cl_mem gen_grayscale_buffer(raytracer_context* rctx,
394                             const unsigned int width,
395                             const unsigned int height);
396 cl_mem gen_ld_image(raytracer_context* rctx, size_t t, void* ptr);
397 void retrieve_buf(raytracer_context* rctx, cl_mem g_buf, void* c_buf, size_t);
398
399 void zero_buffer_img(raytracer_context* rctx, cl_mem buf, size_t element,
400                    const unsigned int width,
401                    const unsigned int height);
402 void zero_buffer(raytracer_context* rctx, cl_mem buf, size_t size);
403 size_t get_workgroup_size(raytracer_context* rctx, cl_kernel kernel);
404
405 /*****
406  * raytracer.h *
407  *****/
408 #pragma once
409 #include <stdint.h>
410 #include <parallel.h>
411 #include <CL/opencl.h>
412 #include <scene.h>
413 #include <irradiance_cache.h>
414
415 //Cheap, quick, and dirty way of managing kernels.
416 #define KERNELS {"cast_ray_test", "generate_rays", "path_trace", \
417                "buffer_average", "f_buffer_average", "f_buffer_to_byte_buffer", \
418                "ic_screen_textures", "generate_discontinuity", \
419                "float_average", "mip_single_upsample", "mip_upsample", \
420                "mip_upsample_scaled", "mip_single_upsample_scaled", "mip_reduce", \
421                "blit_float_to_output", "blit_float3_to_output"}
422 #define NUM_KERNELS 16
423 #define RAY_CAST_KRNL_INDX 0
424 #define RAY_BUFFER_KRNL_INDX 1
425 #define PATH_TRACE_KRNL_INDX 2
426 #define BUFFER_AVG_KRNL_INDX 3
427 #define F_BUFFER_AVG_KRNL_INDX 4
428 #define F_BUF_TO_BYTE_BUF_KRNL_INDX 5
429 #define IC_SCREEN_TEX_KRNL_INDX 6
430 #define IC_GEN_DISC_KRNL_INDX 7
431 #define IC_FLOAT_AVG_KRNL_INDX 8
432 #define IC_MIP_S_UPSAMPLE_KRNL_INDX 9
433 #define IC_MIP_UPSAMPLE_KRNL_INDX 10
434 #define IC_MIP_UPSAMPLE_SCALED_KRNL_INDX 11

```

```

435 #define IC_MIP_S_UPSAMPLE_SCALED_KRNL_INDX 12
436 #define IC_MIP_REDUCE_KRNL_INDX 13
437 #define BLIT_FLOAT_OUTPUT_INDX 14
438 #define BLIT_FLOAT3_OUTPUT_INDX 15
439
440
441 typedef struct _rt_ctx raytracer_context;
442
443 struct _rt_ctx
444 {
445     unsigned int width, height;
446
447     float* ray_buffer;
448     vec4* path_output_buffer;
449     uint32_t* output_buffer;
450     //uint32_t* fresh_frame_buffer;
451
452     scene* stat_scene;
453     ic_context* ic_ctx;
454
455
456     //TODO: seperate into contexts for each integrator.
457     //Path tracing only
458
459     unsigned int num_samples;
460     unsigned int current_sample;
461     bool render_complete;
462
463     //CL
464     rcl_ctx* rcl;
465     rcl_program* program;
466
467     cl_mem cl_ray_buffer;
468     cl_mem cl_output_buffer;
469     cl_mem cl_path_output_buffer;
470     cl_mem cl_path_fresh_frame_buffer; //Only exists on GPU
471
472 };
473
474 raytracer_context* raytracer_init(unsigned int width, unsigned int height,
475                                   uint32_t* output_buffer, rcl_ctx* ctx);
476 void raytracer_prepass(raytracer_context*);
477 void raytracer_render(raytracer_context*);
478 void raytracer_refined_render(raytracer_context*);
479 void _raytracer_gen_ray_buffer(raytracer_context*);
480 void _raytracer_path_trace(raytracer_context*, unsigned int);
481 void _raytracer_average_buffers(raytracer_context*, unsigned int); //NOTE: DEPRECATED
482 void _raytracer_push_path(raytracer_context*);
483 void _raytracer_cast_rays(raytracer_context*); //NOTE: DEPRECATED
484
485 /*****/
486 /* debug.c */
487 /*****/
488
489 #ifdef _MEM_DEBUG
490 void* _debug_memcpy(void* dest, void* from, size_t size, int line, const char *func)
491 {
492     printf("\n-");
493     memcpy(dest, from, size);
494     printf("- memcpy at %i, %s, %p[%li]\n\n", line, func, dest, size);
495     fflush(stdout);
496     return dest;
497 }
498 void* _debug_malloc(size_t size, int line, const char *func)
499 {
500     printf("\n-");
501     void* p = malloc(size);
502     printf("- Allocation at %i, %s, %p[%li]\n\n", line, func, p, size);
503     fflush(stdout);
504     return p;
505 }
506
507 void _debug_free(void* ptr, int line, const char *func)
508 {
509     printf("\n-");
510     free(ptr);
511     printf("- Free at %i, %s, %p\n\n", line, func, ptr);
512     fflush(stdout);
513 }
514
515
516 #define malloc(X) _debug_malloc( X, __LINE__, __FUNCTION__)
517 #define free(X) _debug_free( X, __LINE__, __FUNCTION__)
518 #define memcpy(X, Y, Z) _debug_memcpy( X, Y, Z, __LINE__, __FUNCTION__)
519
520 #endif
521

```

```

522 #ifdef WIN32
523 #define _FILE_SEP '\\'
524 #else
525 #define _FILE_SEP '/'
526 #endif
527
528 #define __FILENAME__ (strrchr(__FILE__, _FILE_SEP) ? strrchr(__FILE__, _FILE_SEP) + 1 : __FILE__)
529
530
531 //TODO: replace all errors with this.
532 #define ASRT_CL(m)
533 if(err!=CL_SUCCESS)
534 {
535     fprintf(stderr, "ERROR: %s. (code: %i, line: %i, file:%s)\nPRESS ENTER TO EXIT\n", \
536         m, err, __LINE__, __FILENAME__);
537     fflush(stderr);
538     while(1){char c; scanf("%c",&c); exit(1);}
539 }
540
541 /*****/
542 /* geom.c */
543 /*****/
544 #include <geom.h>
545 #define DEBUG_PRINT_VEC3(n, v) printf(n ": (%f, %f, %f)\n", v[0], v[1], v[2])
546
547
548 inline bool solve_quadratic(float *a, float *b, float *c, float *x0, float *x1)
549 {
550     float discr = (*b) * (*b) - 4 * (*a) * (*c);
551
552     if (discr < 0) return false;
553     else if (discr == 0) {
554         (*x0) = (*x1) = - 0.5 * (*b) / (*a);
555     }
556     else {
557         float q = (*b > 0) ?
558             -0.5 * (*b + sqrt(discr)) :
559             -0.5 * (*b - sqrt(discr));
560         *x0 = q / *a;
561         *x1 = *c / q;
562     }
563
564     return true;
565 }
566
567 float* matvec_mul(mat4 m, vec4 v)
568 {
569     float* out_float = (float*)malloc(sizeof(vec4));
570
571     out_float[0] = m[0+0*4]*v[0] + m[0+1*4]*v[1] + m[0+2*4]*v[2] + m[0+3*4]*v[3];
572     out_float[1] = m[1+0*4]*v[0] + m[1+1*4]*v[1] + m[1+2*4]*v[2] + m[1+3*4]*v[3];
573     out_float[2] = m[2+0*4]*v[0] + m[2+1*4]*v[1] + m[2+2*4]*v[2] + m[2+3*4]*v[3];
574     out_float[3] = m[3+0*4]*v[0] + m[3+1*4]*v[1] + m[3+2*4]*v[2] + m[3+3*4]*v[3];
575
576     return out_float;
577 }
578
579 void swap_float(float *f1, float *f2)
580 {
581     float temp = *f2;
582     *f2 = *f1;
583     *f1 = temp;
584 }
585
586 inline float does_collide_sphere(sphere s, ray r)
587 {
588     float t0, t1; // solutions for t if the ray intersects
589
590
591     vec3 L;
592     xv_sub(L, r.orig, s.pos, 3);
593
594
595     float a = 1.0f; //NOTE: we always normalize the direction vector.
596     float b = xv3_dot(r.dir, L) * 2.0f;
597     float c = xv3_dot(L, L) - (s.radius*s.radius); //NOTE: square can be optimized out.
598     if (!solve_quadratic(&a, &b, &c, &t0, &t1)) return -1.0f;
599
600     if (t0 > t1) swap_float(&t0, &t1);
601
602     if (t0 < 0) {
603         t0 = t1; // if t0 is negative, use t1 instead
604         if (t0 < 0) return -1.0f; // both t0 and t1 are negative
605     }
606
607     return t0;
608 }

```

```

600
610 inline float does_collide_plane(plane p, ray r)
611 {
612     float denom = xv_dot3(r.dir, p.norm);
613     if (denom > 1e-6)
614     {
615         vec3 l;
616         xv_sub(l, p.pos, r.orig, 3);
617         float t = xv_dot3(l, p.norm) / denom;
618         if (t >= 0)
619             return -1.0;
620         return t;
621     }
622     return -1.0;
623 }
624
625 ray generate_ray(int x, int y, int width, int height, float fov)
626 {
627     ray r;
628
629     //Simplified
630     /* float ndc_x = ((float)x+0.5)/width; */
631     /* float ndc_y = ((float)x+0.5)/height; */
632     /* float screen_x = 2 * ndc_x - 1; */
633     /* float screen_y = 1 - 2 * ndc_y; */
634     /* float aspect_ratio = width/height; */
635     /* float cam_x = (2*screen_x-1) * tan(fov / 2 * M_PI / 180) * aspect_ratio; */
636     /* float cam_y = (1-2*screen_y) * tan(fov / 2 * M_PI / 180); */
637
638     float aspect_ratio = width / (float)height; // assuming width > height
639     float cam_x = (2 * (((float)x + 0.5) / width) - 1) * tan(fov / 2 * M_PI / 180) * aspect_ratio;
640     float cam_y = (1 - 2 * (((float)y + 0.5) / height)) * tan(fov / 2 * M_PI / 180);
641
642
643     xv3_zero(r.orig);
644     vec3 v1 = {cam_x, cam_y, -1};
645     xv_sub(r.dir, v1, r.orig, 3);
646     xv_normeq(r.dir, 3);
647
648     return r;
649 }
650
651 /*****
652  * irradiance_cache.c
653  *****/
654
655 /*****
656  * NOTE: Irradiance Caching is Incomplete
657  *****/
658
659 #include <irradiance_cache.h>
660 #include <raytracer.h>
661 #include <parallel.h>
662
663 #ifdef WIN32
664 #define alloca _alloca
665 #endif
666 void ic_init(raytracer_context* rctx)
667 {
668     rctx->ic_ctx->cl_standard_format.image_channel_order = CL_RGBA;
669     rctx->ic_ctx->cl_standard_format.image_channel_data_type = CL_FLOAT;
670
671     rctx->ic_ctx->cl_standard_descriptor.image_type = CL_MEM_OBJECT_IMAGE2D;
672     rctx->ic_ctx->cl_standard_descriptor.image_width = rctx->width;
673     rctx->ic_ctx->cl_standard_descriptor.image_height = rctx->height;
674     rctx->ic_ctx->cl_standard_descriptor.image_depth = 0;
675     rctx->ic_ctx->cl_standard_descriptor.image_array_size = 0;
676     rctx->ic_ctx->cl_standard_descriptor.image_row_pitch = 0;
677     rctx->ic_ctx->cl_standard_descriptor.num_mip_levels = 0;
678     rctx->ic_ctx->cl_standard_descriptor.num_samples = 0;
679     rctx->ic_ctx->cl_standard_descriptor.buffer = NULL;
680
681     rctx->ic_ctx->octree.node_count = 1; //root
682     //TODO: add as parameter
683     rctx->ic_ctx->octree.max_depth = 8; //arbitrary
684     rctx->ic_ctx->octree.width = 15; //arbitrary
685
686     rctx->ic_ctx->octree.root = (ic_octree_node*) malloc(sizeof(ic_octree_node));
687     rctx->ic_ctx->octree.root->min[0] = (float)-rctx->ic_ctx->octree.width;
688     rctx->ic_ctx->octree.root->min[1] = (float)-rctx->ic_ctx->octree.width;
689     rctx->ic_ctx->octree.root->min[2] = (float)-rctx->ic_ctx->octree.width;
690     rctx->ic_ctx->octree.root->max[0] = (float) rctx->ic_ctx->octree.width;
691     rctx->ic_ctx->octree.root->max[1] = (float) rctx->ic_ctx->octree.width;
692     rctx->ic_ctx->octree.root->max[2] = (float) rctx->ic_ctx->octree.width;
693     rctx->ic_ctx->octree.root->leaf = false;
694     rctx->ic_ctx->octree.root->active = false;
695 }

```



```

697 void ic_octree_init_leaf(ic_octree_node* node, ic_octree_node* parent, unsigned int i)
698 {
699     float xhalf = (parent->max[0]-parent->min[0])/2;
700     float yhalf = (parent->max[1]-parent->min[1])/2;
701     float zhalf = (parent->max[2]-parent->min[2])/2;
702     node->active = false;
703
704     node->leaf = true;
705     for(int i = 0; i < 8; i++)
706         node->data.branch.children[i] = NULL;
707     node->min[0] = parent->min[0] + ( (i&4) ? xhalf : 0);
708     node->min[1] = parent->min[1] + ( (i&2) ? yhalf : 0);
709     node->min[2] = parent->min[2] + ( (i&1) ? zhalf : 0);
710     node->max[0] = parent->max[0] - (!(i&4) ? xhalf : 0);
711     node->max[1] = parent->max[1] - (!(i&2) ? yhalf : 0);
712     node->max[2] = parent->max[2] - (!(i&1) ? zhalf : 0);
713 }
714
715 void ic_octree_make_branch(ic_octree* tree, ic_octree_node* node)
716 {
717
718     node->leaf = false;
719     for(int i = 0; i < 8; i++)
720     {
721         node->data.branch.children[i] = malloc(sizeof(ic_octree_node));
722         ic_octree_init_leaf(node->data.branch.children[i], node, i);
723         tree->node_count++;
724     }
725 }
726
727 //TODO: test if points are the same
728 void _ic_octree_rec_resolve(ic_context* ictx, ic_octree_node* leaf, unsigned int node1, unsigned int node2,
729                             unsigned int depth)
730 {
731     if(depth > ictx->octree.max_depth)
732     {
733         //TODO: just group buffers together
734         printf("ERROR: octree reached max depth when trying to resolve collision. (INCOMPLETE)\n");
735         exit(1);
736     }
737     vec3 mid_point;
738     xv_sub(mid_point, leaf->max, leaf->min, 3);
739     xv_divieq(mid_point, 2, 3);
740     unsigned int i1 =
741         ((mid_point[0]<ictx->ir_buf[node1].point[0])<<2) |
742         ((mid_point[1]<ictx->ir_buf[node1].point[1])<<1) |
743         ((mid_point[2]<ictx->ir_buf[node1].point[2]));
744     unsigned int i2 =
745         ((mid_point[0]<ictx->ir_buf[node2].point[0])<<2) |
746         ((mid_point[1]<ictx->ir_buf[node2].point[1])<<1) |
747         ((mid_point[2]<ictx->ir_buf[node2].point[2]));
748     ic_octree_make_branch(&ictx->octree, leaf);
749     if(i1==i2)
750         _ic_octree_rec_resolve(ictx, leaf->data.branch.children[i1], node1, node2, depth+1);
751     else
752     { //happiness
753         leaf->data.branch.children[i1]->data.leaf.buffer_offset = node1;
754         leaf->data.branch.children[i1]->data.leaf.num_elems = 1;
755         leaf->data.branch.children[i2]->data.leaf.buffer_offset = node2;
756         leaf->data.branch.children[i2]->data.leaf.num_elems = 1;
757     }
758 }
759
760 void _ic_octree_rec_insert(ic_context* ictx, ic_octree_node* node, unsigned int v_ptr, unsigned int depth)
761 {
762     if(node->leaf && !node->active)
763     {
764         node->active = true;
765         node->data.leaf.buffer_offset = v_ptr;
766         node->data.leaf.num_elems = 1; //TODO: add suport for more than 1.
767         return;
768     }
769     else if(node->leaf)
770     {
771         //resolve
772         _ic_octree_rec_resolve(ictx, node, v_ptr, node->data.leaf.buffer_offset, depth+1);
773     }
774     else
775     {
776         ic_octree_node* new_node = node->data.branch.children[
777             ((ictx->ir_buf[node->data.leaf.buffer_offset].point[0]<ictx->ir_buf[v_ptr].point[0])<<2) |
778             ((ictx->ir_buf[node->data.leaf.buffer_offset].point[1]<ictx->ir_buf[v_ptr].point[1])<<1) |
779             ((ictx->ir_buf[node->data.leaf.buffer_offset].point[2]<ictx->ir_buf[v_ptr].point[2]));
780         _ic_octree_rec_insert(ictx, new_node, v_ptr, depth+1);
781     }
782 }

```

```

784 void ic_octree_insert(ic_context* ictx, vec3 point, vec3 normal)
785 {
786     if(ictx->ir_buf_current_offset==ictx->ir_buf_size) //TODO: dynamically resize or do something else
787     {
788         printf("ERROR: irradiance buffer is full!\n");
789         exit(1);
790     }
791     ic_ir_value irradiance_value; //TODO: EVALUATE THIS
792     ictx->ir_buf[ictx->ir_buf_current_offset++] = irradiance_value;
793     _ic_octree_rec_insert(ictx, ictx->octree.root, ictx->ir_buf_current_offset, 0);
794 }
795
796 //NOTE: outBuffer is only bools but using char for safety accross compilers.
797 // Also assuming that buf is grayscale
798 void dither(float* buf, const int width, const int height)
799 {
800     for(int y = 0; y < height; y++)
801     {
802         for(int x = 0; x < width; x++)
803         {
804             float oldpixel = buf[x+y*width];
805             float newpixel = oldpixel>0.5f ? 1 : 0;
806             buf[x+y*width] = newpixel;
807             float err = oldpixel - newpixel;
808
809             if( (x != (width-1)) && (x != 0) && (y != (height-1)) )
810             {
811                 buf[(x+1)+(y )*width] = buf[(x+1)+(y )*width] + err * (7.f / 16.f);
812                 buf[(x-1)+(y+1)*width] = buf[(x-1)+(y+1)*width] + err * (3.f / 16.f);
813                 buf[(x )+(y+1)*width] = buf[(x )+(y+1)*width] + err * (5.f / 16.f);
814                 buf[(x+1)+(y+1)*width] = buf[(x+1)+(y+1)*width] + err * (1.f / 16.f);
815             }
816         }
817     }
818 }
819
820
821 void get_geom_maps(raytracer_context* rctx, cl_mem positions, cl_mem normals)
822 {
823     int err;
824
825     cl_kernel kernel = rctx->program->raw_kernels[IC_SCREEN_TEX_KRNL_INDX];
826
827     float zeroed[] = {0., 0., 0., 1.};
828     float* result = matvec_mul(rctx->stat_scene->camera_world_matrix, zeroed);
829
830     //SO MANY ARGUEMENTS
831     clSetKernelArg(kernel, 0, sizeof(cl_mem), &positions);
832     clSetKernelArg(kernel, 1, sizeof(cl_mem), &normals);
833     clSetKernelArg(kernel, 2, sizeof(int), &rctx->width);
834     clSetKernelArg(kernel, 3, sizeof(int), &rctx->height);
835     clSetKernelArg(kernel, 4, sizeof(cl_mem), &rctx->cl_ray_buffer);
836     clSetKernelArg(kernel, 5, sizeof(vec4), result);
837     clSetKernelArg(kernel, 6, sizeof(cl_mem), &rctx->stat_scene->cl_material_buffer);
838     clSetKernelArg(kernel, 7, sizeof(cl_mem), &rctx->stat_scene->cl_sphere_buffer);
839     clSetKernelArg(kernel, 8, sizeof(cl_mem), &rctx->stat_scene->cl_plane_buffer);
840     clSetKernelArg(kernel, 9, sizeof(cl_mem), &rctx->stat_scene->cl_mesh_buffer);
841     clSetKernelArg(kernel, 10, sizeof(cl_mem), &rctx->stat_scene->cl_mesh_index_buffer);
842     clSetKernelArg(kernel, 11, sizeof(cl_mem), &rctx->stat_scene->cl_mesh_vert_buffer);
843     clSetKernelArg(kernel, 12, sizeof(cl_mem), &rctx->stat_scene->cl_mesh_nrml_buffer);
844
845     size_t global = rctx->width*rctx->height;
846     size_t local = 0;
847     err = clGetKernelWorkGroupInfo(kernel, rctx->rcl->device_id, CL_KERNEL_WORK_GROUP_SIZE,
848                                     sizeof(local), &local, NULL);
849     ASRT_CL("Failed to Retrieve Kernel Work Group Info");
850
851     err = clEnqueueNDRangeKernel(rctx->rcl->commands, kernel, 1, NULL, &global,
852                                  NULL, 0, NULL, NULL);
853     ASRT_CL("Failed to Enqueue kernel IC_SCREEN_TEX");
854
855     //Wait for completion
856     err = clFinish(rctx->rcl->commands);
857     ASRT_CL("Something happened while waiting for kernel to finish");
858 }
859
860 void gen_mipmap_chain_gb(raytracer_context* rctx, cl_mem texture,
861                          ic_mipmap_gb* mipmaps, int num_mipmaps)
862 {
863     int err;
864     unsigned int width = rctx->width;
865     unsigned int height = rctx->height;
866     cl_kernel kernel = rctx->program->raw_kernels[IC_MIP_REDUCE_KRNL_INDX];
867     for(int i = 0; i < num_mipmaps; i++)
868     {
869         mipmaps[i].width = width;

```

```

870 mipmaps[i].height = height;
871
872 if(i==0)
873 {
874     mipmaps[0].cl_image_ref = texture;
875
876     height /= 2;
877     width /= 2;
878     continue;
879 }
880
881 clSetKernelArg(kernel, 0, sizeof(cl_mem), &mipmaps[i-1].cl_image_ref);
882 clSetKernelArg(kernel, 1, sizeof(cl_mem), &mipmaps[i].cl_image_ref);
883 clSetKernelArg(kernel, 2, sizeof(int), &width);
884 clSetKernelArg(kernel, 3, sizeof(int), &height);
885
886 size_t global = width*height;
887 size_t local = get_workgroup_size(rctx, kernel);
888
889 err = clEnqueueNDRangeKernel(rctx->rcl->commands, kernel, 1,
890                             NULL, &global, NULL, 0, NULL, NULL);
891 ASRT_CL("Failed to Enqueue kernel IC_MIP_REDUCE");
892
893 height /= 2;
894 width /= 2;
895 //Wait for completion before doing next mip
896 err = clFinish(rctx->rcl->commands);
897 ASRT_CL("Something happened while waiting for kernel to finish");
898 }
899 }
900
901 void upsample_mipmaps_f(raytracer_context* rctx, cl_mem texture,
902                        ic_mipmap_f* mipmaps, int num_mipmaps)
903 {
904     int err;
905
906     cl_mem* full_maps = (cl_mem*) alloca(sizeof(cl_mem)*num_mipmaps);
907     for(int i = 1; i < num_mipmaps; i++)
908     {
909         full_maps[i] = gen_grayscale_buffer(rctx, 0, 0);
910     }
911     full_maps[0] = texture;
912     { //Upsample
913         for(int i = 0; i < num_mipmaps; i++) //First one is already at proper resolution
914         {
915             cl_kernel kernel = rctx->program->raw_kernels[IC_MIP_S_UPSAMPLE_SCALED_KRNL_INDX];
916
917             clSetKernelArg(kernel, 0, sizeof(cl_mem), &mipmaps[i].cl_image_ref);
918             clSetKernelArg(kernel, 1, sizeof(cl_mem), &full_maps[i]); //NOTE: need to generate this for the function
919             clSetKernelArg(kernel, 2, sizeof(int), &i);
920             clSetKernelArg(kernel, 3, sizeof(int), &rctx->width);
921             clSetKernelArg(kernel, 4, sizeof(int), &rctx->height);
922
923             size_t global = rctx->width*rctx->height;
924             size_t local = get_workgroup_size(rctx, kernel);
925
926             err = clEnqueueNDRangeKernel(rctx->rcl->commands, kernel, 1,
927                                         NULL, &global, NULL, 0, NULL, NULL);
928             ASRT_CL("Failed to Enqueue kernel IC_MIP_S_UPSAMPLE_SCALED");
929
930         }
931         err = clFinish(rctx->rcl->commands);
932         ASRT_CL("Something happened while waiting for kernel to finish");
933     }
934     printf("Upsampled Discontinuity Mipmaps\nAveraging Upsampled Discontinuity Mipmaps\n");
935
936     { //Average
937         int total = num_mipmaps;
938         for(int i = 0; i < num_mipmaps; i++) //First one is already at proper resolution
939         {
940             cl_kernel kernel = rctx->program->raw_kernels[IC_FLOAT_AVG_KRNL_INDX];
941
942             clSetKernelArg(kernel, 0, sizeof(cl_mem), &full_maps[i]);
943             clSetKernelArg(kernel, 1, sizeof(cl_mem), &texture);
944             clSetKernelArg(kernel, 2, sizeof(int), &rctx->width);
945             clSetKernelArg(kernel, 3, sizeof(int), &rctx->height);
946             clSetKernelArg(kernel, 4, sizeof(int), &total);
947
948             size_t global = rctx->width*rctx->height;
949             size_t local = 0;
950             err = clGetKernelWorkGroupInfo(kernel, rctx->rcl->device_id, CL_KERNEL_WORK_GROUP_SIZE,
951                                         sizeof(local), &local, NULL);
952             ASRT_CL("Failed to Retrieve Kernel Work Group Info");
953
954             err = clEnqueueNDRangeKernel(rctx->rcl->commands, kernel, 1,
955                                         NULL, &global, NULL, 0, NULL, NULL);
956             ASRT_CL("Failed to Enqueue kernel IC_FLOAT_AVG");

```

```

957     err = clFinish(rctx->rcl->commands);
958     ASRT_CL("Something happened while waiting for kernel to finish");
959 }
960 }
961 }
962 for(int i = 1; i < num_mipmaps; i++)
963 {
964     err = clReleaseMemObject(full_maps[i]);
965     ASRT_CL("Failed to cleanup fullsize mipmaps");
966 }
967 }
968
969 void gen_discontinuity_maps(raytracer_context* rctx, ic_mipmap_gb* pos_mipmaps,
970                             ic_mipmap_gb* nrm_mipmaps, ic_mipmap_f* disc_mipmaps,
971                             int num_mipmaps)
972 {
973     int err;
974     //TODO: tune k and intensity
975     const float k = 1.6f;
976     const float intensity = 0.02f;
977     for(int i = 0; i < num_mipmaps; i++)
978     {
979         cl_kernel kernel = rctx->program->raw_kernels[IC_GEN_DISC_KRNL_INDX];
980         disc_mipmaps[i].width = pos_mipmaps[i].width;
981         disc_mipmaps[i].height = pos_mipmaps[i].height;
982
983         clSetKernelArg(kernel, 0, sizeof(cl_mem), &pos_mipmaps[i].cl_image_ref);
984
985
986         clSetKernelArg(kernel, 1, sizeof(cl_mem), &nrm_mipmaps[i].cl_image_ref);
987         clSetKernelArg(kernel, 2, sizeof(cl_mem), &disc_mipmaps[i].cl_image_ref);
988         clSetKernelArg(kernel, 3, sizeof(float), &k);
989         clSetKernelArg(kernel, 4, sizeof(float), &intensity);
990         clSetKernelArg(kernel, 5, sizeof(int), &pos_mipmaps[i].width);
991         clSetKernelArg(kernel, 6, sizeof(int), &pos_mipmaps[i].height);
992
993         size_t global = pos_mipmaps[i].width*pos_mipmaps[i].height;
994         size_t local = get_workgroup_size(rctx, kernel);
995
996         err = clEnqueueNDRangeKernel(rctx->rcl->commands, kernel, 1,
997                                     NULL, &global, NULL, 0, NULL, NULL);
998         ASRT_CL("Failed to Enqueue kernel IC_GEN_DISC");
999     }
1000 }
1001 err = clFinish(rctx->rcl->commands);
1002 ASRT_CL("Something happened while waiting for kernel to finish");
1003 }
1004
1005 void ic_screenspace(raytracer_context* rctx)
1006 {
1007     int err;
1008
1009
1010     vec4* pos_tex = (vec4*) malloc(rctx->width*rctx->height*sizeof(vec4));
1011     vec4* nrm_tex = (vec4*) malloc(rctx->width*rctx->height*sizeof(vec4));
1012     float* c_fin_disc_map = (float*) malloc(rctx->width*rctx->height*sizeof(float));
1013
1014     ic_mipmap_gb pos_mipmaps [NUM_MIPMAPS]; //A lot of buffers
1015     ic_mipmap_gb nrm_mipmaps [NUM_MIPMAPS];
1016     ic_mipmap_f disc_mipmaps[ NUM_MIPMAPS];
1017     cl_mem fin_disc_map;
1018     //OpenCL
1019     cl_mem cl_pos_tex;
1020     cl_mem cl_nrm_tex;
1021     cl_image_desc cl_mipmap_descriptor = rctx->ic_ctx->cl_standard_descriptor;
1022
1023     { //OpenCL Init
1024         cl_pos_tex = gen_rgb_image(rctx, 0,0);
1025         cl_nrm_tex = gen_rgb_image(rctx, 0,0);
1026
1027         fin_disc_map = gen_grayscale_buffer(rctx, 0,0);
1028         zero_buffer_img(rctx, fin_disc_map, sizeof(float), 0, 0);
1029
1030
1031         unsigned int width = rctx->width,
1032                     height = rctx->height;
1033         for(int i = 0; i < NUM_MIPMAPS; i++)
1034         {
1035             if(i!=0)
1036             {
1037                 pos_mipmaps[i].cl_image_ref = gen_rgb_image(rctx, width, height);
1038                 nrm_mipmaps[i].cl_image_ref = gen_rgb_image(rctx, width, height);
1039             }
1040             disc_mipmaps[i].cl_image_ref = gen_grayscale_buffer(rctx, width, height);
1041
1042             width /= 2;
1043             height /= 2;

```

```

1044 }
1045 }
1046 printf("Initialised Irradiance Cache Screenspace Buffers\nGetting Screenspace Geometry Data\n");
1047 get_geom_maps(rctx, cl_pos_tex, cl_nrm_tex);
1048 printf("Got Screenspace Geometry Data\nGenerating MipMaps\n");
1049 gen_mipmap_chain_gb(rctx, cl_pos_tex,
1050                    pos_mipmaps, NUM_MIPMAPS);
1051 gen_mipmap_chain_gb(rctx, cl_nrm_tex,
1052                    nrm_mipmaps, NUM_MIPMAPS);
1053 printf("Generated MipMaps\nGenerating Discontinuity Map for each Mip\n");
1054 gen_discontinuity_maps(rctx, pos_mipmaps, nrm_mipmaps, disc_mipmaps, NUM_MIPMAPS);
1055 printf("Generated Discontinuity Map for each Mip\nUpsampling Discontinuity Mipmaps\n");
1056 upsample_mipmaps_f(rctx, fin_disc_map, disc_mipmaps, NUM_MIPMAPS);
1057 printf("Averaged Upsampled Discontinuity Mipmaps\nRetrieving Discontinuity Data\n");
1058 retrieve_buf(rctx, fin_disc_map, c_fin_disc_map,
1059             rctx->width*rctx->height*sizeof(float));
1060 retrieve_image(rctx, cl_pos_tex, pos_tex, 0, 0);
1061 retrieve_image(rctx, cl_pos_tex, pos_tex, 0, 0);
1062
1063 printf("Retrieved Discontinuity Data\nDithering Discontinuity Map\n");
1064 //NOTE: read buffer is blocking so we don't need clFinish
1065 dither(c_fin_disc_map, rctx->width, rctx->height);
1066 err = clEnqueueWriteBuffer(rctx->rcl->commands, fin_disc_map,
1067                            CL_TRUE, 0,
1068                            rctx->width*rctx->height*sizeof(float),
1069                            c_fin_disc_map, 0, 0, NULL);
1070 ASRT_CL("Failed to write dithered discontinuity map");
1071
1072
1073 //INSERT
1074 cl_kernel kernel = rctx->program->raw_kernels[BLIT_FLOAT_OUTPUT_INDX];
1075
1076 clSetKernelArg(kernel, 0, sizeof(cl_mem), &rctx->cl_output_buffer);
1077 clSetKernelArg(kernel, 1, sizeof(cl_mem), &fin_disc_map);
1078 clSetKernelArg(kernel, 2, sizeof(int), &rctx->width);
1079 clSetKernelArg(kernel, 3, sizeof(int), &rctx->height);
1080
1081 size_t global = rctx->width*rctx->height;
1082 size_t local = 0;
1083 err = clGetKernelWorkGroupInfo(kernel, rctx->rcl->device_id, CL_KERNEL_WORK_GROUP_SIZE,
1084                                sizeof(local), &local, NULL);
1085 ASRT_CL("Failed to Retrieve Kernel Work Group Info");
1086
1087 err = clEnqueueNDRangeKernel(rctx->rcl->commands, kernel, 1,
1088                              NULL, &global, NULL, 0, NULL, NULL);
1089 ASRT_CL("Failed to Enqueue kernel BLIT_FLOAT_OUTPUT_INDX");
1090
1091 clFinish(rctx->rcl->commands);
1092
1093 err = clEnqueueReadBuffer(rctx->rcl->commands, rctx->cl_output_buffer, CL_TRUE, 0,
1094                           rctx->width*rctx->height*sizeof(int), rctx->output_buffer, 0, NULL, NULL );
1095 ASRT_CL("Failed to Read Output Buffer");
1096 printf("test!!\n");
1097
1098
1099 }
1100
1101
1102
1103
1104 /*****/
1105 /* Loader.c */
1106 /*****/
1107 #include <loader.h>
1108 #include <parson.h>
1109 #include <vec.h>
1110 #include <float.h>
1111 #include <tinyobj_loader_c.h>
1112 #include <assert.h>
1113
1114
1115
1116 #ifndef WIN32
1117 #include <libproc.h>
1118 #include <unistd.h>
1119
1120 #define FILE_SEP '/'
1121
1122 char* _get_os_pid_bin_path()
1123 {
1124     static bool initialised = false;
1125     static char path[PROC_PIDPATHINFO_MAXSIZE];
1126     if(!initialised)
1127     {
1128         int ret;
1129         pid_t pid;
1130         char path[PROC_PIDPATHINFO_MAXSIZE];

```

```

1131         pid = getpid();
1132         ret = proc_pidpath(pid, path, sizeof(path));
1133
1134         if(ret <= 0)
1135         {
1136             printf("Error: couldn't get bin path.\n");
1137             exit(1);
1138         }
1139     }
1140 }
1141 return path;
1142 }
1143 #else
1144 #include <windows.h>
1145 #define FILE_SEP '\\\\'
1146
1147 char* _get_os_pid_bin_path()
1148 {
1149     static bool initialised = false;
1150     static char path[260];
1151     if(!initialised)
1152     {
1153         HMODULE hModule = GetModuleHandleW(NULL);
1154
1155         WCHAR tpath[260];
1156         GetModuleFileNameW(hModule, tpath, 260);
1157
1158         char DefChar = ' ';
1159         WideCharToMultiByte(CP_ACP, 0, tpath, -1, path, 260, &DefChar, NULL);
1160
1161         *(strrchr(path, FILE_SEP)) = '\\0'; //get last occurence;
1162     }
1163     return path;
1164 }
1165 #endif
1166
1167 char* load_file(const char* url, long *ret_length)
1168 {
1169     char real_url[260];
1170     sprintf(real_url, "%s%cres%c%s", _get_os_pid_bin_path(), FILE_SEP, FILE_SEP, url);
1171
1172     char * buffer = 0;
1173     long length;
1174     FILE * f = fopen (real_url, "rb");
1175
1176     if (f)
1177     {
1178         fseek (f, 0, SEEK_END);
1179         length = ftell (f)+1;
1180         fseek (f, 0, SEEK_SET);
1181         buffer = malloc (length);
1182         if (buffer)
1183         {
1184             fread (buffer, 1, length, f);
1185         }
1186         fclose (f);
1187     }
1188     if (buffer)
1189     {
1190         buffer[length] = '\\0';
1191
1192         *ret_length = length;
1193         return buffer;
1194     }
1195     else
1196     {
1197         printf("Error: Couldn't load file '%s'.\n", real_url);
1198         exit(1);
1199     }
1200 }
1201 }
1202
1203 //Linked List for Mesh Loading
1204 struct obj_list_elem
1205 {
1206     struct obj_list_elem* next;
1207     tinyobj_attrib_t attrib;
1208     tinyobj_shape_t* shapes;
1209     size_t num_shapes;
1210     int mat_index;
1211     mat4 model_mat;
1212 };
1213
1214 void obj_pre_load(char* data, long data_len, struct obj_list_elem* elem,
1215                 int* num_meshes, unsigned int* num_indices, unsigned int* num_vertices,
1216                 unsigned int* num_normals, unsigned int* num_texcoords)

```

```

1218 {
1219
1220     tinyobj_material_t* materials = NULL; //NOTE: UNUSED
1221     size_t num_materials; //NOTE: UNUSED
1222
1223
1224     {
1225         unsigned int flags = TINYOBJ_FLAG_TRIANGULATE;
1226         int ret = tinyobj_parse_obj(&elem->attrib, &elem->shapes, &elem->num_shapes, &materials,
1227                                     &num_materials, data, data_len, flags);
1228         if (ret != TINYOBJ_SUCCESS) {
1229             printf("Error: Couldn't parse mesh.\n");
1230             exit(1);
1231         }
1232     }
1233
1234     *num_vertices += elem->attrib.num_vertices;
1235     *num_normals += elem->attrib.num_normals;
1236     *num_texcoords += elem->attrib.num_texcoords;
1237     *num_meshes += elem->num_shapes;
1238     //tinyobjloader has dumb variable names: attrib.num_faces = num_vertices+num_faces
1239     *num_indices += elem->attrib.num_faces;
1240 }
1241
1242
1243
1244 void load_obj(struct obj_list_elem elem, int* mesh_offset, int* vert_offset, int* nrml_offset,
1245              int* texcoord_offset, int* index_offset, scene* out_scene)
1246 {
1247     for(int i = 0; i < elem.num_shapes; i++)
1248     {
1249         tinyobj_shape_t shape = elem.shapes[i];
1250
1251         //Get mesh and increment offset.
1252         mesh* m = (out_scene->meshes) + (*mesh_offset)++;
1253
1254         m->min[0] = m->min[1] = m->min[2] = FLT_MAX;
1255         m->max[0] = m->max[1] = m->max[2] = -FLT_MAX;
1256
1257         memcpy(m->model, elem.model_mat, 4*4*sizeof(float));
1258
1259         m->index_offset = *index_offset;
1260         m->num_indices = shape.length*3;
1261         m->material_index = elem.mat_index;
1262
1263         for(int f = 0; f < shape.length; f++)
1264         {
1265             //TODO: don't do this error check for each iteration
1266             if(elem.attrib.face_num_verts[f+shape.face_offset]!=3)
1267             {
1268                 //This should never get called because the mesh gets triangulated when loaded.
1269                 printf("Error: the obj loader only supports triangulated meshes!\n");
1270                 exit(1);
1271             }
1272             for(int i = 0; i < 3; i++)
1273             {
1274                 tinyobj_vertex_index_t face_index = elem.attrib.faces[(f+shape.face_offset)*3+i];
1275
1276                 vec3 vertex;
1277                 vertex[0] = elem.attrib.vertices[3*face_index.v_idx+0];
1278                 vertex[1] = elem.attrib.vertices[3*face_index.v_idx+1];
1279                 vertex[2] = elem.attrib.vertices[3*face_index.v_idx+2];
1280
1281                 m->min[0] = vertex[0] < m->min[0] ? vertex[0] : m->min[0]; //X min
1282                 m->min[1] = vertex[1] < m->min[1] ? vertex[1] : m->min[1]; //Y min
1283                 m->min[2] = vertex[2] < m->min[2] ? vertex[2] : m->min[2]; //Z min
1284
1285                 m->max[0] = vertex[0] > m->max[0] ? vertex[0] : m->max[0]; //X max
1286                 m->max[1] = vertex[1] > m->max[1] ? vertex[1] : m->max[1]; //Y max
1287                 m->max[2] = vertex[2] > m->max[2] ? vertex[2] : m->max[2]; //Z max
1288
1289                 ivec3 index;
1290                 index[0] = (*vert_offset)+face_index.v_idx;
1291                 index[1] = (*nrml_offset)+face_index.vn_idx;
1292                 index[2] = (*texcoord_offset)+face_index.vt_idx;
1293                 out_scene->mesh_indices[(*index_offset)][0] = index[0];
1294                 out_scene->mesh_indices[(*index_offset)][1] = index[1];
1295                 out_scene->mesh_indices[(*index_offset)][2] = index[2];
1296
1297                 //xv3_cpy(out_scene->mesh_indices + (*index_offset), index);
1298                 (*index_offset)++;
1299             }
1300         }
1301     }
1302
1303     //GPU MEMORY ALIGNMENT FUN
1304     //NOTE: this is done because the gpu stores all vec3s 4 floats for memory alignment

```



```

1305 // and it is actually faster if they are aligned like this even
1306 // though it wastes more memory.
1307 for(int i = 0; i < elem.attrib.num_vertices; i++)
1308 {
1309
1310     memcpy(out_scene->mesh_verts + (*vert_offset),
1311            elem.attrib.vertices+3*i,
1312            sizeof(vec3));
1313     (*vert_offset) += 1;
1314 }
1315 for(int i = 0; i < elem.attrib.num_normals; i++)
1316 {
1317     memcpy(out_scene->mesh_nrmls + (*nrml_offset),
1318            elem.attrib.normals+3*i,
1319            sizeof(vec3));
1320     (*nrml_offset) += 1;
1321 }
1322 //NOTE: the texcoords are already aligned because they only have 2 elements.
1323 memcpy(out_scene->mesh_texcoords + (*texcoord_offset), elem.attrib.texcoords,
1324        elem.attrib.num_texcoords*sizeof(vec2));
1325 (*texcoord_offset) += elem.attrib.num_texcoords;
1326 }
1327
1328 scene* load_scene_json(char* json)
1329 {
1330     printf("Beginning scene loading...\n");
1331     scene* out_scene = (scene*) malloc(sizeof(scene));
1332     JSON_Value *root_value;
1333     JSON_Object *root_object;
1334     root_value = json_parse_string(json);
1335     root_object = json_value_get_object(root_value);
1336
1337
1338     //Name
1339     {
1340         const char* name = json_object_get_string(root_object, "name");
1341         printf("Scene name: %s\n", name);
1342     }
1343
1344     //Version
1345     { //TODO: do something with this.
1346         int major = (int)json_object_dotget_number(root_object, "version.major");
1347         int minor = (int)json_object_dotget_number(root_object, "version.minor");
1348         const char* type = json_object_dotget_string(root_object, "version.type");
1349     }
1350
1351     //Materials
1352     {
1353         JSON_Array* material_array = json_object_get_array(root_object, "materials");
1354         out_scene->num_materials = json_array_get_count(material_array);
1355         out_scene->materials = (material*) malloc(out_scene->num_materials*sizeof(material));
1356         assert(out_scene->num_materials>0);
1357         for(int i = 0; i < out_scene->num_materials; i++)
1358         {
1359             JSON_Object* mat = json_array_get_object(material_array, i);
1360             xv_x(out_scene->materials[i].colour) = json_object_get_number(mat, "r");
1361             xv_y(out_scene->materials[i].colour) = json_object_get_number(mat, "g");
1362             xv_z(out_scene->materials[i].colour) = json_object_get_number(mat, "b");
1363             out_scene->materials[i].reflectivity = json_object_get_number(mat, "reflectivity");
1364         }
1365         printf("Materials: %d\n", out_scene->num_materials);
1366     }
1367
1368     //Primitives
1369     {
1370
1371         JSON_Object* primitive_object = json_object_get_object(root_object, "primitives");
1372
1373         //Spheres
1374         {
1375             JSON_Array* sphere_array = json_object_get_array(primitive_object, "spheres");
1376             int num_spheres = json_array_get_count(sphere_array);
1377
1378             out_scene->spheres = malloc(sizeof(sphere)*num_spheres);
1379             out_scene->num_spheres = num_spheres;
1380
1381             for(int i = 0; i < num_spheres; i++)
1382             {
1383                 JSON_Object* sphere = json_array_get_object(sphere_array, i);
1384                 out_scene->spheres[i].pos[0] = json_object_get_number(sphere, "x");
1385                 out_scene->spheres[i].pos[1] = json_object_get_number(sphere, "y");
1386                 out_scene->spheres[i].pos[2] = json_object_get_number(sphere, "z");
1387                 out_scene->spheres[i].radius = json_object_get_number(sphere, "radius");
1388                 out_scene->spheres[i].material_index = json_object_get_number(sphere, "mat_index");
1389             }
1390             printf("Spheres: %d\n", out_scene->num_spheres);
1391         }
1392     }

```



```

1392 //Planes
1393 {
1394     JSON_Array* plane_array = json_object_get_array(primitive_object, "planes");
1395     int num_planes = json_array_get_count(plane_array);
1396
1397     out_scene->planes = malloc(sizeof(plane)*num_planes);
1398     out_scene->num_planes = num_planes;
1399
1400     for(int i = 0; i < num_planes; i++)
1401     {
1402         JSON_Object* plane = json_array_get_object(plane_array, i);
1403         out_scene->planes[i].pos[0] = json_object_get_number(plane, "x");
1404         out_scene->planes[i].pos[1] = json_object_get_number(plane, "y");
1405         out_scene->planes[i].pos[2] = json_object_get_number(plane, "z");
1406         out_scene->planes[i].norm[0] = json_object_get_number(plane, "nx");
1407         out_scene->planes[i].norm[1] = json_object_get_number(plane, "ny");
1408         out_scene->planes[i].norm[2] = json_object_get_number(plane, "nz");
1409
1410         out_scene->planes[i].material_index = json_object_get_number(plane, "mat_index");
1411     }
1412     printf("Planes: %d\n", out_scene->num_planes);
1413 }
1414 }
1415
1416 //Meshes
1417 {
1418     JSON_Array* mesh_array = json_object_get_array(root_object, "meshes");
1419
1420     int num_meshes = json_array_get_count(mesh_array);
1421
1422     out_scene->num_meshes = 0;
1423     out_scene->num_mesh_verts = 0;
1424     out_scene->num_mesh_nrmls = 0;
1425     out_scene->num_mesh_texcoords = 0;
1426     out_scene->num_mesh_indices = 0;
1427
1428     struct obj_list_elem* first = (struct obj_list_elem*) malloc(sizeof(struct obj_list_elem));
1429     struct obj_list_elem* current = first;
1430
1431     //Pre evaluation
1432     for(int i = 0; i < num_meshes; i++)
1433     {
1434         JSON_Object* mesh = json_array_get_object(mesh_array, i);
1435         const char* url = json_object_get_string(mesh, "url");
1436         long length;
1437         char* data = load_file(url, &length);
1438         obj_pre_load(data, length, current, &out_scene->num_meshes, &out_scene->num_mesh_indices,
1439                     &out_scene->num_mesh_verts, &out_scene->num_mesh_nrmls,
1440                     &out_scene->num_mesh_texcoords);
1441         current->mat_index = (int) json_object_get_number(mesh, "mat_index");
1442         //mat4 model_mat;
1443         {
1444             //xm4_identity(model_mat);
1445             mat4 translation_mat;
1446             xm4_translatev(translation_mat,
1447                           json_object_get_number(mesh, "px"),
1448                           json_object_get_number(mesh, "py"),
1449                           json_object_get_number(mesh, "pz"));
1450             mat4 scale_mat;
1451             xm4_scalev(scale_mat,
1452                       json_object_get_number(mesh, "sx"),
1453                       json_object_get_number(mesh, "sy"),
1454                       json_object_get_number(mesh, "sz"));
1455             //TODO: add rotation.
1456             xm4_mul(current->model_mat, translation_mat, scale_mat);
1457         }
1458         free(data);
1459
1460         if(i!=num_meshes-1) //messy but it works
1461         {
1462             current->next = (struct obj_list_elem*) malloc(sizeof(struct obj_list_elem));
1463             current = current->next;
1464         }
1465         current->next = NULL;
1466     }
1467 }
1468
1469 //Allocation
1470 out_scene->meshes = (mesh*) malloc(sizeof(mesh)*out_scene->num_meshes);
1471 out_scene->mesh_verts = (vec3*) malloc(sizeof(vec3)*out_scene->num_mesh_verts);
1472 out_scene->mesh_nrmls = (vec3*) malloc(sizeof(vec3)*out_scene->num_mesh_nrmls);
1473 out_scene->mesh_texcoords = (vec2*) malloc(sizeof(vec2)*out_scene->num_mesh_texcoords);
1474 out_scene->mesh_indices = (ivec3*) malloc(sizeof(ivec3)*out_scene->num_mesh_indices);

```

```

1479     assert(out_scene->meshes!=NULL);
1480     assert(out_scene->mesh_verts!=NULL);
1481     assert(out_scene->mesh_nrmls!=NULL);
1482     assert(out_scene->mesh_texcoords!=NULL);
1483     assert(out_scene->mesh_indices!=NULL);
1484
1485     //Parsing and Assignment
1486     int mesh_offset = 0;
1487     int vert_offset = 0;
1488     int nrml_offset = 0;
1489     int texcoord_offset = 0;
1490     int index_offset = 0;
1491
1492
1493     current = first;
1494     while(current != NULL && num_meshes)
1495     {
1496
1497         load_obj(*current, &mesh_offset, &vert_offset, &nrml_offset, &texcoord_offset,
1498             &index_offset, out_scene);
1499
1500         current = current->next;
1501     }
1502     printf("%i and %i\n", vert_offset, out_scene->num_mesh_verts);
1503     assert(mesh_offset==out_scene->num_meshes);
1504     assert(vert_offset==out_scene->num_mesh_verts);
1505     assert(nrml_offset==out_scene->num_mesh_nrmls);
1506     assert(texcoord_offset==out_scene->num_mesh_texcoords);
1507
1508     assert(index_offset==out_scene->num_mesh_indices);
1509
1510     printf("Meshes: %d\nVertices: %d\nIndices: %d\n",
1511         out_scene->num_meshes, out_scene->num_mesh_verts, out_scene->num_mesh_indices);
1512
1513 }
1514
1515 out_scene->materials_changed = true;
1516 out_scene->spheres_changed = true;
1517 out_scene->planes_changed = true;
1518 out_scene->meshes_changed = true;
1519
1520
1521 printf("Finshed scene loading.\n\n");
1522
1523 json_value_free(root_value);
1524 return out_scene;
1525 }
1526
1527
1528 scene* load_scene_json_url(char* url)
1529 {
1530     long variable_doesnt_matter;
1531
1532     return load_scene_json( load_file(url, &variable_doesnt_matter) ); //TODO: put data
1533 }
1534
1535
1536 /*****
1537  * os_abs.c *
1538  *****/
1539 #include <os_abs.h>
1540
1541 void os_start(os_abs abs)
1542 {
1543     (*abs.start_func)();
1544 }
1545
1546 void os_loop_start(os_abs abs)
1547 {
1548     (*abs.loop_start_func)();
1549 }
1550
1551 void os_update(os_abs abs)
1552 {
1553     (*abs.update_func)();
1554 }
1555
1556 void os_sleep(os_abs abs, int num)
1557 {
1558     (*abs.sleep_func)(num);
1559 }
1560
1561 void* os_get_bitmap_memory(os_abs abs)
1562 {
1563     return (*abs.get_bitmap_memory_func)();
1564 }
1565

```

```

1566 int os_get_time_mili(os_abs abs)
1567 {
1568     return (*abs.get_time_mili_func)();
1569 }
1570
1571 int os_get_width(os_abs abs)
1572 {
1573     return (*abs.get_width_func)();
1574 }
1575
1576 int os_get_height(os_abs abs)
1577 {
1578     return (*abs.get_height_func)();
1579 }
1580
1581 void os_start_thread(os_abs abs, void (*func)(void*), void* data)
1582 {
1583     (*abs.start_thread_func)(func, data);
1584 }
1585
1586 /*****
1587  * parallel.c *
1588  *****/
1589 #include <CL/opencl.h>
1590 #include <raytracer.h>
1591 //Parallel util.
1592
1593 void cl_info()
1594 {
1595
1596     int i, j;
1597     char* value;
1598     size_t valueSize;
1599     cl_uint platformCount;
1600     cl_platform_id* platforms;
1601     cl_uint deviceCount;
1602     cl_device_id* devices;
1603     cl_uint maxComputeUnits;
1604
1605     // get all platforms
1606     clGetPlatformIDs(0, NULL, &platformCount);
1607     platforms = (cl_platform_id*) malloc(sizeof(cl_platform_id) * platformCount);
1608     clGetPlatformIDs(platformCount, platforms, NULL);
1609
1610     for (i = 0; i < platformCount; i++) {
1611
1612         // get all devices
1613         clGetDeviceIDs(platforms[i], CL_DEVICE_TYPE_ALL, 0, NULL, &deviceCount);
1614         devices = (cl_device_id*) malloc(sizeof(cl_device_id) * deviceCount);
1615         clGetDeviceIDs(platforms[i], CL_DEVICE_TYPE_ALL, deviceCount, devices, NULL);
1616
1617         // for each device print critical attributes
1618         for (j = 0; j < deviceCount; j++) {
1619
1620             // print device name
1621             clGetDeviceInfo(devices[j], CL_DEVICE_NAME, 0, NULL, &valueSize);
1622             value = (char*) malloc(valueSize);
1623             clGetDeviceInfo(devices[j], CL_DEVICE_NAME, valueSize, value, NULL);
1624             printf("%i.%d. Device: %s\n", i, j+1, value);
1625             free(value);
1626
1627             // print hardware device version
1628             clGetDeviceInfo(devices[j], CL_DEVICE_VERSION, 0, NULL, &valueSize);
1629             value = (char*) malloc(valueSize);
1630             clGetDeviceInfo(devices[j], CL_DEVICE_VERSION, valueSize, value, NULL);
1631             printf(" %i.%d.%d Hardware version: %s\n", i, j+1, 1, value);
1632             free(value);
1633
1634             // print software driver version
1635             clGetDeviceInfo(devices[j], CL_DRIVER_VERSION, 0, NULL, &valueSize);
1636             value = (char*) malloc(valueSize);
1637             clGetDeviceInfo(devices[j], CL_DRIVER_VERSION, valueSize, value, NULL);
1638             printf(" %i.%d.%d Software version: %s\n", i, j+1, 2, value);
1639             free(value);
1640
1641             // print c version supported by compiler for device
1642             clGetDeviceInfo(devices[j], CL_DEVICE_OPENCL_C_VERSION, 0, NULL, &valueSize);
1643             value = (char*) malloc(valueSize);
1644             clGetDeviceInfo(devices[j], CL_DEVICE_OPENCL_C_VERSION, valueSize, value, NULL);
1645             printf(" %i.%d.%d OpenCL C version: %s\n", i, j+1, 3, value);
1646             free(value);
1647
1648             // print parallel compute units
1649             clGetDeviceInfo(devices[j], CL_DEVICE_MAX_COMPUTE_UNITS,
1650                 sizeof(maxComputeUnits), &maxComputeUnits, NULL);
1651             printf(" %i.%d.%d Parallel compute units: %d\n", i, j+1, 4, maxComputeUnits);
1652

```

```

1653     }
1654
1655     free(devices);
1656
1657 }
1658 printf("\n");
1659 free(platforms);
1660 return;
1661 }
1662 void pfn_notify (
1663     const char *errinfo,
1664     const void *private_info,
1665     size_t cb,
1666     void *user_data)
1667 {
1668     fprintf(stderr, "\n--\nOpenCL ERROR: %s\n--\n", errinfo);
1669     fflush(stderr);
1670 }
1671 void create_context(rcl_ctx* ctx)
1672 {
1673     int err = CL_SUCCESS;
1674
1675
1676     int num_of_platforms;
1677
1678     if (clGetPlatformIDs(0, NULL, &num_of_platforms) != CL_SUCCESS)
1679     {
1680         printf("Error: Unable to get platform_id\n");
1681         exit(1);
1682     }
1683     cl_platform_id *platform_ids = malloc(num_of_platforms*sizeof(cl_platform_id));
1684     if (clGetPlatformIDs(num_of_platforms, platform_ids, NULL) != CL_SUCCESS)
1685     {
1686         printf("Error: Unable to get platform_id\n");
1687         exit(1);
1688     }
1689     bool found = false;
1690     for(int i=0; i<num_of_platforms; i++)
1691         if(clGetDeviceIDs(platform_ids[i], CL_DEVICE_TYPE_GPU, 1, &ctx->device_id, NULL) == CL_SUCCESS)
1692         {
1693             found = true;
1694             ctx->platform_id = platform_ids[i];
1695
1696             break;
1697         }
1698     if(!found){
1699         printf("Error: Unable to get a GPU device_id\n");
1700         exit(1);
1701     }
1702
1703
1704     // Create a compute context
1705     //
1706     ctx->context = clCreateContext(0, 1, &ctx->device_id, &pfn_notify, NULL, &err);
1707     if (!ctx->context)
1708     {
1709         printf("Error: Failed to create a compute context!\n");
1710         exit(1);
1711     }
1712
1713     // Create a command commands
1714     //
1715     ctx->commands = clCreateCommandQueue(ctx->context, ctx->device_id, 0, &err);
1716     if (!ctx->commands)
1717     {
1718         printf("Error: Failed to create a command commands!\n");
1719         return;
1720     }
1721     ASRT_CL("Failed to Initialise OpenCL");
1722
1723 }
1724 }
1725
1726 cl_mem gen_rgb_image(raytracer_context* rctx,
1727     const unsigned int width,
1728     const unsigned int height)
1729 {
1730     cl_image_desc cl_standard_descriptor;
1731     cl_image_format cl_standard_format;
1732     cl_standard_format.image_channel_order = CL_RGBA;
1733     cl_standard_format.image_channel_data_type = CL_FLOAT;
1734
1735     cl_standard_descriptor.image_type = CL_MEM_OBJECT_IMAGE2D;
1736     cl_standard_descriptor.image_width = width==0 ? rctx->width : width;
1737     cl_standard_descriptor.image_height = height==0 ? rctx->height : height;
1738     cl_standard_descriptor.image_depth = 0;
1739     cl_standard_descriptor.image_array_size = 0;

```

```

1740 cl_standard_descriptor.image_row_pitch = 0;
1741 cl_standard_descriptor.num_mip_levels = 0;
1742 cl_standard_descriptor.num_samples = 0;
1743 cl_standard_descriptor.buffer = NULL;
1744
1745 int err;
1746
1747 cl_mem img = clCreateImage(rctx->rcl->context,
1748                             CL_MEM_READ_WRITE,
1749                             &cl_standard_format,
1750                             &cl_standard_descriptor,
1751                             NULL,
1752                             &err);
1753 ASRT_CL("Couldn't Create OpenCL Texture");
1754 return img;
1755 }
1756 cl_mem gen_1d_image(raytracer_context* rctx, size_t t, void* ptr)
1757 {
1758
1759     cl_image_desc cl_standard_descriptor;
1760     cl_image_format cl_standard_format;
1761     cl_standard_format.image_channel_order = CL_RGBA;
1762     cl_standard_format.image_channel_data_type = CL_FLOAT;
1763
1764     cl_standard_descriptor.image_type = CL_MEM_OBJECT_IMAGE1D;
1765     cl_standard_descriptor.image_width = t/4 == 0 ? 1 : t/4;
1766     cl_standard_descriptor.image_height = 0;
1767     cl_standard_descriptor.image_depth = 0;
1768     cl_standard_descriptor.image_array_size = 0;
1769     cl_standard_descriptor.image_row_pitch = 0;
1770     cl_standard_descriptor.num_mip_levels = 0;
1771     cl_standard_descriptor.num_samples = 0;
1772     cl_standard_descriptor.buffer = NULL;
1773
1774     int err;
1775
1776
1777     cl_mem img = clCreateImage(rctx->rcl->context,
1778                             CL_MEM_READ_WRITE | (/*ptr == NULL ? 0 :*/ CL_MEM_COPY_HOST_PTR),
1779                             &cl_standard_format,
1780                             &cl_standard_descriptor,
1781                             ptr,
1782                             &err);
1783     ASRT_CL("Couldn't Create OpenCL Texture");
1784     return img;
1785 }
1786 cl_mem gen_grayscale_buffer(raytracer_context* rctx,
1787                             const unsigned int width,
1788                             const unsigned int height)
1789 {
1790     int err;
1791
1792     cl_mem buf = clCreateBuffer(rctx->rcl->context, CL_MEM_READ_WRITE,
1793                                (width==0 ? rctx->width : width)*
1794                                (height==0 ? rctx->height : height)*
1795                                sizeof(float),
1796                                NULL, &err);
1797     ASRT_CL("Couldn't Create OpenCL Float Buffer Image");
1798     return buf;
1799 }
1800
1801 void retrieve_image(raytracer_context* rctx, cl_mem g_buf, void* c_buf,
1802                   const unsigned int width,
1803                   const unsigned int height)
1804 {
1805     int err;
1806     size_t origin[3] = {0,0,0};
1807     size_t region[3] = {(width==0 ? rctx->width : width),
1808                        (height==0 ? rctx->height : height),
1809                        1};
1810     err = clEnqueueReadImage (rctx->rcl->commands,
1811                              g_buf,
1812                              CL_TRUE,
1813                              origin,
1814                              region,
1815                              0,
1816                              0,
1817                              c_buf,
1818                              0,
1819                              0,
1820                              NULL);
1821     ASRT_CL("Failed to retrieve Opencl Image");
1822 }
1823
1824 void retrieve_buf(raytracer_context* rctx, cl_mem g_buf, void* c_buf, size_t size)
1825 {
1826     int err;

```

```

1827     err = clEnqueueReadBuffer(rctx->rcl->commands, g_buf, CL_TRUE, 0,
1828                               size, c_buf,
1829                               0, NULL, NULL);
1830     ASRT_CL("Failed to retrieve OpenCL Buffer");
1831 }
1832
1833 void zero_buffer(raytracer_context* rctx, cl_mem buf, size_t size)
1834 {
1835     int err;
1836     char pattern = 0;
1837     err = clEnqueueFillBuffer (rctx->rcl->commands,
1838                               buf,
1839                               &pattern, 1, 0,
1840                               size,
1841                               0, NULL, NULL);
1842     ASRT_CL("Couldn't Zero OpenCL Buffer");
1843 }
1844 void zero_buffer_img(raytracer_context* rctx, cl_mem buf, size_t element,
1845                      const unsigned int width,
1846                      const unsigned int height)
1847 {
1848     int err;
1849
1850     char pattern = 0;
1851     err = clEnqueueFillBuffer (rctx->rcl->commands,
1852                               buf,
1853                               &pattern, 1, 0,
1854                               (width==0 ? rctx->width : width)*
1855                               (height==0 ? rctx->height : height)*
1856                               element,
1857                               0, NULL, NULL);
1858     ASRT_CL("Couldn't Zero OpenCL Buffer");
1859 }
1860 size_t get_workgroup_size(raytracer_context* rctx, cl_kernel kernel)
1861 {
1862     int err;
1863     size_t local = 0;
1864     err = clGetKernelWorkGroupInfo(kernel, rctx->rcl->device_id,
1865                                     CL_KERNEL_WORK_GROUP_SIZE,
1866                                     sizeof(local), &local, NULL);
1867     ASRT_CL("Failed to Retrieve Kernel Work Group Info");
1868     return local;
1869 }
1870
1871
1872 void load_program_raw(rcl_ctx* ctx, char* data,
1873                      char** kernels, unsigned int num_kernels,
1874                      rcl_program* program, char** macros, unsigned int num_macros)
1875 {
1876     int err;
1877
1878     char* fin_data = (char*) malloc(strlen(data));
1879     strcpy(fin_data, data);
1880
1881     for(int i = 0; i < num_macros; i++)
1882     {
1883         int length = strlen(macros[i]);
1884         char* buf = (char*) malloc(length+strlen(fin_data)+3);
1885         sprintf(buf, "%s\n%s\0", macros[i], fin_data);
1886         free(fin_data);
1887         fin_data = buf;
1888     }
1889
1890     program->program = clCreateProgramWithSource(ctx->context, 1, (const char **) &fin_data, NULL, &err);
1891     if (!program->program)
1892     {
1893         printf("Error: Failed to create compute program!\n");
1894         exit(1);
1895     }
1896
1897     // Build the program executable
1898     //
1899     err = clBuildProgram(program->program, 0, NULL, NULL, NULL, NULL);
1900     if (err != CL_SUCCESS)
1901     {
1902         size_t len;
1903         char buffer[2048*256];
1904         buffer[0] = '!';
1905         buffer[1] = '\0';
1906
1907         printf("Error: Failed to build program executable!\n");
1908         printf("KERNEL:\n %s\nprogram done\n", fin_data);
1909         int n_err = clGetProgramBuildInfo(program->program, ctx->device_id, CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &len);
1910         if(n_err != CL_SUCCESS)
1911         {
1912             printf("The error had an error, I hate this. err:%i\n", n_err);

```

```

1914     }
1915     printf("err code:%i\n %s\n", err, buffer);
1916     exit(1);
1917 }
1918
1919 program->raw_kernels = malloc(sizeof(cl_kernel)*num_kernels);
1920 for(int i = 0; i < num_kernels; i++)
1921 {
1922     // Create the compute kernel in the program we wish to run
1923     //
1924
1925     program->raw_kernels[i] = clCreateKernel(program->program, kernels[i], &err);
1926     if (!program->raw_kernels[i] || err != CL_SUCCESS)
1927     {
1928         printf("Error: Failed to create compute kernel! %s\n", kernels[i]);
1929         exit(1);
1930     }
1931 }
1932
1933 program->raw_data = fin_data;
1934
1935 }
1936 }
1937
1938 void load_program_url(rcl_ctx* ctx, char* url,
1939                     char** kernels, unsigned int num_kernels,
1940                     rcl_program* program, char** macros, unsigned int num_macros)
1941 {
1942     char * buffer = 0;
1943     long length;
1944     FILE * f = fopen (url, "rb");
1945
1946     if (f)
1947     {
1948         fseek (f, 0, SEEK_END);
1949         length = ftell (f);
1950         fseek (f, 0, SEEK_SET);
1951         buffer = malloc (length+2);
1952         if (buffer)
1953         {
1954             fread (buffer, 1, length, f);
1955         }
1956         fclose (f);
1957     }
1958     if (buffer)
1959     {
1960         buffer[length] = '\0';
1961
1962         load_program_raw(ctx, buffer, kernels, num_kernels, program,
1963                         macros, num_macros);
1964     }
1965 }
1966 }
1967
1968 //NOTE: old
1969 void test_sphere_raytracer(rcl_ctx* ctx, rcl_program* program,
1970                           sphere* spheres, int num_spheres,
1971                           uint32_t* bitmap, int width, int height)
1972 {
1973     int err;
1974
1975     static cl_mem tex;
1976     static cl_mem s_buf;
1977     static bool init = false; //temporary
1978
1979     if(!init)
1980     {
1981         //New Texture
1982         tex = clCreateBuffer(ctx->context, CL_MEM_WRITE_ONLY,
1983                             width*height*4, NULL, &err);
1984
1985         //Spheres
1986         s_buf = clCreateBuffer(ctx->context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
1987                               sizeof(float)*4*num_spheres, spheres, &err);
1988         if (err != CL_SUCCESS)
1989         {
1990             printf("Error: Failed to create Sphere Buffer! %d\n", err);
1991             return;
1992         }
1993         init = true;
1994     }
1995     else
1996     {
1997         clEnqueueWriteBuffer ( ctx->commands,
1998                               s_buf,
1999                               CL_TRUE,
2000                               0,

```

```

2000         sizeof(float)*4*num_spheres,
2001         spheres,
2002         0,
2003         NULL,
2004         NULL);
2005     }
2006 }
2007
2008
2009
2010     cl_kernel kernel = program->raw_kernels[0]; //just use the first one
2011
2012     clSetKernelArg(kernel, 0, sizeof(cl_mem), &tex);
2013     clSetKernelArg(kernel, 1, sizeof(cl_mem), &s_buf);
2014     clSetKernelArg(kernel, 2, sizeof(unsigned int), &width);
2015     clSetKernelArg(kernel, 3, sizeof(unsigned int), &height);
2016
2017
2018     size_t global;
2019     size_t local = 0;
2020
2021     err = clGetKernelWorkGroupInfo(kernel, ctx->device_id, CL_KERNEL_WORK_GROUP_SIZE,
2022         sizeof(local), &local, NULL);
2023     if (err != CL_SUCCESS)
2024     {
2025         printf("Error: Failed to retrieve kernel work group info! %d\n", err);
2026         return;
2027     }
2028
2029     // Execute the kernel over the entire range of our 1d input data set
2030     // using the maximum number of work group items for this device
2031     //
2032     //printf("STARTING\n");
2033     global = width*height;
2034     err = clEnqueueNDRangeKernel(ctx->commands, kernel, 1, NULL, &global, NULL, 0, NULL, NULL);
2035     if (err)
2036     {
2037         printf("Error: Failed to execute kernel! %i\n",err);
2038         return;
2039     }
2040
2041
2042     clFinish(ctx->commands);
2043     //printf("STOPPING\n");
2044
2045     err = clEnqueueReadBuffer(ctx->commands, tex, CL_TRUE, 0, width*height*4, bitmap, 0, NULL, NULL );
2046     if (err != CL_SUCCESS)
2047     {
2048         printf("Error: Failed to read output array! %d\n", err);
2049         exit(1);
2050     }
2051 }
2052
2053 *****
2054 /* raytracer.c */
2055 *****
2056
2057 #include <raytracer.h>
2058 #include <parallel.h>
2059
2060 //binary resources
2061 #include <test.cl.h> //test kernel
2062
2063
2064
2065 //NOTE: we are assuming the output buffer will be the right size
2066 raytracer_context* raytracer_init(unsigned int width, unsigned int height,
2067     uint32_t* output_buffer, rcl_ctx* rcl)
2068 {
2069     raytracer_context* rctx = (raytracer_context*) malloc(sizeof(raytracer_context));
2070     rctx->width = width;
2071     rctx->height = height;
2072     rctx->ray_buffer = (float*) malloc(width * height * sizeof(float)*3);
2073     rctx->output_buffer = output_buffer;
2074     //rctx->fresh_buffer = (uint32_t*) malloc(width * height * sizeof(uint32_t));
2075     rctx->rcl = rcl;
2076     rctx->program = (rcl_program*) malloc(sizeof(rcl_program));
2077     rctx->ic_ctx = (ic_context*) malloc(sizeof(ic_context));
2078     //ic_init(rctx);
2079     rctx->render_complete = false;
2080     rctx->num_samples = 64; //NOTE: arbitrary default
2081     rctx->current_sample = 0;
2082
2083     return rctx;
2084 }
2085
2086 void raytracer_cl_prepass(raytracer_context* rctx)
2087 {

```



```

2088 //CL init
2089 printf("Building Scene Kernels...\n");
2090
2091 int err = CL_SUCCESS;
2092
2093 //Kernels
2094 char* kernels[] = KERNELS;
2095
2096 //Macros
2097 char sphere_macro[64];
2098 sprintf(sphere_macro, "#define SCENE_NUM_SPHERES %i", rctx->stat_scene->num_spheres);
2099 char plane_macro[64];
2100 sprintf(plane_macro, "#define SCENE_NUM_PLANES %i", rctx->stat_scene->num_planes);
2101 char index_macro[64];
2102 sprintf(index_macro, "#define SCENE_NUM_INDICES %i", rctx->stat_scene->num_mesh_indices);
2103 char mesh_macro[64];
2104 sprintf(mesh_macro, "#define SCENE_NUM_MESHES %i", rctx->stat_scene->num_meshes);
2105 char material_macro[64];
2106 sprintf(material_macro, "#define SCENE_NUM_MATERIALS %i", rctx->stat_scene->num_materials);
2107 char* macros[] = {sphere_macro, plane_macro, mesh_macro, index_macro, material_macro};
2108
2109 {
2110     load_program_raw(rctx->rcl,
2111                     all_kernels_cl, //NOTE: Binary resource
2112                     kernels, NUM_KERNELS, rctx->program,
2113                     macros, 5);
2114 }
2115
2116 //Buffers
2117 rctx->cl_ray_buffer = clCreateBuffer(rctx->rcl->context,
2118                                     CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
2119                                     rctx->width*rctx->height*sizeof(float)*3,
2120                                     rctx->ray_buffer, &err);
2121 ASRT_CL("Error Creating OpenCL Ray Buffer.");
2122 rctx->cl_path_output_buffer = clCreateBuffer(rctx->rcl->context,
2123                                             CL_MEM_READ_WRITE,
2124                                             rctx->width*rctx->height*sizeof(vec4),
2125                                             NULL, &err);
2126 ASRT_CL("Error Creating OpenCL Path Tracer Output Buffer.");
2127
2128 rctx->cl_output_buffer = clCreateBuffer(rctx->rcl->context, CL_MEM_READ_WRITE,
2129                                         rctx->width*rctx->height*4, NULL, &err);
2130 ASRT_CL("Error Creating OpenCL Output Buffer.");
2131
2132 //TODO: all output buffers and frame buffers should be images.
2133 rctx->cl_path_fresh_frame_buffer = clCreateBuffer(rctx->rcl->context, CL_MEM_READ_WRITE,
2134                                                  rctx->width*rctx->height*sizeof(vec4), NULL, &err);
2135 ASRT_CL("Error Creating OpenCL Fresh Frame Buffer.");
2136
2137 printf("Pushing Scene Resources.\n");
2138 scene_init_resources(rctx);
2139
2140 printf("Built Scene Kernels.\n");
2141 }
2142
2143 void raytracer_prepass(raytracer_context* rctx)
2144 {
2145     printf("Starting Raytracer Prepass.\n");
2146
2147     raytracer_cl_prepass(rctx);
2148
2149     printf("Finished Raytracer Prepass.\n");
2150 }
2151
2152 //TODO: implement
2153 void raytracer_render(raytracer_context* rctx)
2154 {
2155     _raytracer_gen_ray_buffer(rctx);
2156
2157     _raytracer_cast_rays(rctx);
2158 }
2159
2160 //define JANK_SAMPLES 32
2161 void raytracer_refined_render(raytracer_context* rctx)
2162 {
2163     rctx->current_sample++;
2164     if(rctx->current_sample>rctx->num_samples)
2165     {
2166         rctx->render_complete = true;
2167         return;
2168     }
2169     _raytracer_gen_ray_buffer(rctx);
2170
2171     _raytracer_path_trace(rctx, rctx->current_sample);
2172
2173     if(rctx->current_sample==1) //really terrible place for path tracer initialization...

```

```

2175 {
2176     int err;
2177     char pattern = 0;
2178     err = clEnqueueCopyBuffer (    rctx->rcl->commands,
2179                                   rctx->cl_path_fresh_frame_buffer,
2180                                   rctx->cl_path_output_buffer,
2181                                   0,
2182                                   0,
2183                                   rctx->width*rctx->height*sizeof(vec4),
2184                                   0,
2185                                   0,
2186                                   NULL);
2187     ASRT_CL("Error copying OpenCL Output Buffer");
2188
2189     err = clFinish(rctx->rcl->commands);
2190     ASRT_CL("Something happened while waiting for copy to finish");
2191 }
2192
2193 _raytracer_average_buffers(rctx, rctx->current_sample);
2194 _raytracer_push_path(rctx);
2195 }
2196 }
2197
2198 void _raytracer_gen_ray_buffer(raytracer_context* rctx)
2199 {
2200     int err;
2201
2202     cl_kernel kernel = rctx->program->raw_kernels[RAY_BUFFER_KRNL_INDX];
2203     clSetKernelArg(kernel, 0, sizeof(cl_mem), &rctx->cl_ray_buffer);
2204     clSetKernelArg(kernel, 1, sizeof(unsigned int), &rctx->width);
2205     clSetKernelArg(kernel, 2, sizeof(unsigned int), &rctx->height);
2206     clSetKernelArg(kernel, 3, sizeof(mat4), rctx->stat_scene->camera_world_matrix);
2207
2208
2209     size_t global;
2210
2211
2212     global = rctx->width*rctx->height;
2213     err = clEnqueueNDRangeKernel(rctx->rcl->commands, kernel, 1, NULL, &global, NULL, 0, NULL, NULL);
2214     ASRT_CL("Failed to execute kernel");
2215
2216
2217     //Wait for completion
2218     err = clFinish(rctx->rcl->commands);
2219     ASRT_CL("Something happened while waiting for kernel raybuf to finish");
2220
2221 }
2222 }
2223 void _raytracer_average_buffers(raytracer_context* rctx, unsigned int sample_num)
2224 {
2225     int err;
2226
2227     cl_kernel kernel = rctx->program->raw_kernels[F_BUFFER_AVG_KRNL_INDX];
2228     clSetKernelArg(kernel, 0, sizeof(cl_mem), &rctx->cl_path_output_buffer);
2229     clSetKernelArg(kernel, 1, sizeof(cl_mem), &rctx->cl_path_fresh_frame_buffer);
2230     clSetKernelArg(kernel, 2, sizeof(unsigned int), &rctx->width);
2231     clSetKernelArg(kernel, 3, sizeof(unsigned int), &rctx->height);
2232     clSetKernelArg(kernel, 4, sizeof(unsigned int), &rctx->num_samples);
2233     clSetKernelArg(kernel, 5, sizeof(unsigned int), &sample_num);
2234
2235
2236     size_t global;
2237     size_t local = get_workgroup_size(rctx, kernel);
2238
2239     // Execute the kernel over the entire range of our 1d input data set
2240     // using the maximum number of work group items for this device
2241     //
2242     global = rctx->width*rctx->height;
2243     err = clEnqueueNDRangeKernel(rctx->rcl->commands, kernel, 1, NULL, &global, NULL, 0, NULL, NULL);
2244     ASRT_CL("Failed to execute kernel")
2245     err = clFinish(rctx->rcl->commands);
2246     ASRT_CL("Something happened while waiting for kernel to finish");
2247
2248 }
2249 }
2250
2251 void _raytracer_push_path(raytracer_context* rctx)
2252 {
2253     int err;
2254
2255     cl_kernel kernel = rctx->program->raw_kernels[F_BUF_TO_BYTE_BUF_KRNL_INDX];
2256     clSetKernelArg(kernel, 0, sizeof(cl_mem), &rctx->cl_output_buffer);
2257     clSetKernelArg(kernel, 1, sizeof(cl_mem), &rctx->cl_path_output_buffer);
2258     clSetKernelArg(kernel, 2, sizeof(unsigned int), &rctx->width);
2259     clSetKernelArg(kernel, 3, sizeof(unsigned int), &rctx->height);
2260
2261 }

```

```

2262     size_t global;
2263     size_t local = get_workgroup_size(rctx, kernel);
2264
2265     // Execute the kernel over the entire range of our 1d input data set
2266     // using the maximum number of work group items for this device
2267     //
2268     global = rctx->width*rctx->height;
2269     err = clEnqueueNDRangeKernel(rctx->rcl->commands, kernel, 1, NULL, &global, NULL, 0, NULL, NULL);
2270     ASRT_CL("Failed to execute kernel");
2271
2272
2273
2274     err = clFinish(rctx->rcl->commands);
2275     ASRT_CL("Something happened while waiting for kernel to finish");
2276
2277     err = clEnqueueReadBuffer(rctx->rcl->commands, rctx->cl_output_buffer, CL_TRUE, 0,
2278                             rctx->width*rctx->height*sizeof(int), rctx->output_buffer,
2279                             0, NULL, NULL );
2280     ASRT_CL("Failed to read output array");
2281
2282 }
2283
2284 //NOTE: the more divisions the slower.
2285 #define WATCHDOG_DIVISIONS_X 2
2286 #define WATCHDOG_DIVISIONS_Y 2
2287 void _raytracer_path_trace(raytracer_context* rctx, unsigned int sample_num)
2288 {
2289     int err;
2290
2291     const unsigned x_div = rctx->width/WATCHDOG_DIVISIONS_X;
2292     const unsigned y_div = rctx->height/WATCHDOG_DIVISIONS_Y;
2293
2294     //scene_resource_push(rctx); //Update Scene buffers if necessary.
2295
2296     cl_kernel kernel = rctx->program->raw_kernels[PATH_TRACE_KRNL_INDX]; //just use the first one
2297
2298     float zeroed[] = {0., 0., 0., 1.};
2299     float* result = matvec_mul(rctx->stat_scene->camera_world_matrix, zeroed);
2300
2301     clSetKernelArg(kernel, 0, sizeof(cl_mem), &rctx->cl_path_fresh_frame_buffer);
2302     clSetKernelArg(kernel, 1, sizeof(cl_mem), &rctx->cl_ray_buffer);
2303     clSetKernelArg(kernel, 2, sizeof(cl_mem), &rctx->stat_scene->cl_material_buffer);
2304     clSetKernelArg(kernel, 3, sizeof(cl_mem), &rctx->stat_scene->cl_sphere_buffer);
2305     clSetKernelArg(kernel, 4, sizeof(cl_mem), &rctx->stat_scene->cl_plane_buffer);
2306     clSetKernelArg(kernel, 5, sizeof(cl_mem), &rctx->stat_scene->cl_mesh_buffer);
2307     clSetKernelArg(kernel, 6, sizeof(cl_mem), &rctx->stat_scene->cl_mesh_index_buffer);
2308     clSetKernelArg(kernel, 7, sizeof(cl_mem), &rctx->stat_scene->cl_mesh_vert_buffer);
2309     clSetKernelArg(kernel, 8, sizeof(cl_mem), &rctx->stat_scene->cl_mesh_nrml_buffer);
2310
2311     clSetKernelArg(kernel, 9, sizeof(int), &rctx->width);
2312     clSetKernelArg(kernel, 10, sizeof(vec4), result);
2313     clSetKernelArg(kernel, 11, sizeof(int), &sample_num); //NOTE: I don't think this is used
2314
2315     size_t global[2] = {x_div, y_div};
2316
2317     //NOTE: tripping watchdog timer
2318     if(global[0]*WATCHDOG_DIVISIONS_X*global[1]*WATCHDOG_DIVISIONS_Y!=rctx->width*rctx->height)
2319     {
2320         printf("Watchdog divisions are incorrect!\n");
2321         exit(1);
2322     }
2323
2324     size_t offset[2];
2325
2326     for(int x = 0; x < WATCHDOG_DIVISIONS_X; x++)
2327     {
2328         for(int y = 0; y < WATCHDOG_DIVISIONS_Y; y++)
2329         {
2330             offset[0] = x_div*x;
2331             offset[1] = y_div*y;
2332             err = clEnqueueNDRangeKernel(rctx->rcl->commands, kernel, 2,
2333                                         offset, global, NULL, 0, NULL, NULL);
2334             ASRT_CL("Failed to execute path trace kernel");
2335         }
2336     }
2337
2338     err = clFinish(rctx->rcl->commands);
2339     ASRT_CL("Something happened while executing path trace kernel");
2340 }
2341
2342
2343 void _raytracer_cast_rays(raytracer_context* rctx) //TODO: do more path tracing stuff here
2344 {
2345     int err;
2346
2347
2348

```



```

2436 rctx->stat_scene->cl_mesh_nrml_buffer =
2437     gen_id_image(rctx, rctx->stat_scene->num_mesh_nrmls==0 ? 1 :
2438                 sizeof(vec3)*rctx->stat_scene->num_mesh_nrmls,
2439                 rctx->stat_scene->mesh_nrmls);
2440
2441 rctx->stat_scene->cl_mesh_index_buffer =
2442     gen_id_image(rctx, rctx->stat_scene->num_mesh_indices==0 ? 1 :
2443                 sizeof(int)*
2444                 rctx->stat_scene->num_mesh_indices, //maybe
2445                 rctx->stat_scene->mesh_indices);
2446 }
2447
2448
2449
2450 void scene_resource_push(raytracer_context* rctx)
2451 {
2452     int err;
2453
2454     if(rctx->stat_scene->meshes_changed)
2455     {
2456         clEnqueueWriteBuffer (    rctx->rcl->commands,
2457                                 rctx->stat_scene->cl_mesh_buffer,
2458                                 CL_TRUE,
2459                                 0,
2460                                 sizeof(mesh)*rctx->stat_scene->num_meshes,
2461                                 rctx->stat_scene->meshes,
2462                                 0,
2463                                 NULL,
2464                                 NULL);
2465     }
2466
2467     if(rctx->stat_scene->spheres_changed)
2468     {
2469         clEnqueueWriteBuffer (    rctx->rcl->commands,
2470                                 rctx->stat_scene->cl_sphere_buffer,
2471                                 CL_TRUE,
2472                                 0,
2473                                 sizeof(sphere)*rctx->stat_scene->num_spheres,
2474                                 rctx->stat_scene->spheres,
2475                                 0,
2476                                 NULL,
2477                                 NULL);
2478     }
2479
2480     if(rctx->stat_scene->planes_changed)
2481     {
2482         clEnqueueWriteBuffer (    rctx->rcl->commands,
2483                                 rctx->stat_scene->cl_plane_buffer,
2484                                 CL_TRUE,
2485                                 0,
2486                                 sizeof(plane)*rctx->stat_scene->num_planes,
2487                                 rctx->stat_scene->planes,
2488                                 0,
2489                                 NULL,
2490                                 NULL);
2491     }
2492
2493
2494     if(rctx->stat_scene->materials_changed)
2495     {
2496         clEnqueueWriteBuffer (    rctx->rcl->commands,
2497                                 rctx->stat_scene->cl_material_buffer,
2498                                 CL_TRUE,
2499                                 0,
2500                                 sizeof(material)*rctx->stat_scene->num_materials,
2501                                 rctx->stat_scene->materials,
2502                                 0,
2503                                 NULL,
2504                                 NULL);
2505     }
2506 }
2507
2508 /*****/
2509 /* startup.c */
2510 /*****/
2511 #include <os_abs.h>
2512 #include <stdint.h>
2513 #include <startup.h>
2514 #include <stdio.h>
2515 #include <raytracer.h>
2516
2517
2518
2519
2520 #ifdef WIN32
2521 #include <win32.h>
2522 #endif

```

```

2523
2524 //include <time.h>
2525 #define _USE_MATH_DEFINES
2526 #include <math.h>
2527 #include <geom.h>
2528 #include <parallel.h>
2529 #include <loader.h>
2530 #define NUM_SPHERES 5
2531 #define NUM_PLANES 1
2532
2533 #define STRFY(x) #x
2534 #define DBL_STRFY(x) STRFY(x)
2535
2536
2537
2538
2539
2540 os_abs abst;
2541
2542 void cast_rays(int width, int height, uint32_t* bmap)
2543 {
2544
2545
2546     // unsigned width = 640, height = 480;
2547     // Vec3f *image = new Vec3f[width * height], *pixel = image;
2548     //float invWidth = 1 / (float)width, invHeight = 1 / (float)height;
2549     //float fov = 30, aspectratio = width / (float)height;
2550     //float angle = tan(M_PI * 0.5 * fov / 180.);
2551
2552
2553     static float dist = 5.0f;
2554
2555     sphere s;
2556     xv_x(s.pos) = 0.0f;
2557     xv_y(s.pos) = 0.0f;
2558     xv_z(s.pos) = -dist;
2559     s.radius = 1.0f;
2560
2561     if(dist<2.0f)
2562         dist = 10.0f;
2563     dist -= 0.05f;
2564
2565
2566     int last_time = os_get_time_mili(abst);
2567
2568     const pitch = width*4;
2569
2570     int y = 0;
2571     int x = 0;
2572     for(y = 0; y < height; y++)
2573     {
2574         uint32_t* pixel = (uint32_t*)bmap;
2575         for(x = 0; x < width; x++)
2576         {
2577             ray out_ray = generate_ray(x, y, width, height, 90);
2578             float dist = does_collide_sphere(s, out_ray);
2579             *pixel = dist != -1.0f ? 0x00ffffff & (int) dist*100 : 0x00000000;
2580             /*pixel = 0x000000ff | ((uint32_t)((uint8_t)(y)))<<16;
2581             pixel++;
2582         }
2583         bmap += width;
2584     }
2585     /* float stest = 0.0f; */
2586
2587     /* // compute 1e8 times either Sqrt(x) or its emulation as Pow(x, 0.5) */
2588     /* for (float d = 0; d < width*height*2; d += 1) */
2589     /*     // s += Math.Sqrt(d); // <- uncomment it to test Sqrt */
2590     /*     stest += sqrt(d*d); // <- uncomment it to test Pow */
2591
2592
2593     printf("frame took: %i ms\n", os_get_time_mili(abst)-last_time);
2594
2595 }
2596
2597 bool should_run = true;
2598 bool should_pause = false;
2599 void loop_exit()
2600 {
2601     should_run = false;
2602 }
2603 void loop_pause()
2604 {
2605     should_pause = !should_pause;
2606 }
2607
2608 void run(void* unused_rn)
2609 {

```

```

2610 char isMeme = 'y';
2611 //scanf("%c", &isMeme);
2612
2613
2614 if(isMeme=='y')
2615 {
2616     const int width = os_get_width(abst);
2617     const int height = os_get_height(abst);
2618
2619     const int pitch = width *4;
2620     uint32_t* row = (uint32_t*)os_get_bitmap_memory(abst);
2621
2622     cl_info();
2623
2624     rcl_ctx* rcl = (rcl_ctx*) malloc(sizeof(rcl_ctx));
2625     create_context(rcl);
2626
2627     raytracer_context* rctx = raytracer_init((unsigned int)width, (unsigned int)height,
2628                                             row, rcl);
2629     //scene* rscene = (scene*) malloc(sizeof(scene));
2630     scene* rscene = load_scene_json_url("scenes/path_test.rsc");
2631
2632     rctx->stat_scene = rscene;
2633     rctx->num_samples = 32;
2634
2635     raytracer_prepass(rctx);
2636
2637     xm4_identity(rctx->stat_scene->camera_world_matrix);
2638
2639     float dist = 0.f;
2640
2641
2642     int _timer_store = 0;
2643     int _timer_counter = 0;
2644     float _timer_average = 0.0f;
2645     printf("Rendering:\n\n");
2646
2647     /* static float t = 0.0f; */
2648     /* t += 0.0005f; */
2649     /* dist = sin(t)+1; */
2650     /* //mat4 temp; */
2651     /* xm4_translatev(rctx->stat_scene->camera_world_matrix, 0, dist, 0); */
2652     int real_start = os_get_time_mili(abst);
2653     while(should_run)
2654     {
2655
2656         if(should_pause)
2657             continue;
2658         int last_time = os_get_time_mili(abst);
2659
2660         if(kbhit())
2661         {
2662             switch (getch())
2663             {
2664                 case 'c':
2665                     exit(1);
2666                     break;
2667                 case 27: //ESCAPE
2668                     exit(1);
2669                     break;
2670                 default:
2671                     break;
2672             }
2673         }
2674
2675         raytracer_refined_render(rctx);
2676         if(rctx->render_complete)
2677         {
2678             printf("\n\nRender took: %02i ms\n\n", os_get_time_mili(abst)-real_start);
2679             break;
2680         }
2681
2682
2683         int mili = os_get_time_mili(abst)-last_time;
2684         _timer_store += mili;
2685         _timer_counter++;
2686         printf("\rFrame took: %02i ms, average per 20 frames: %0.2f, avg fps: %03.2f", mili, _timer_average, 1000.0f/_timer_average);
2687
2688         if(_timer_counter>20)
2689         {
2690             _timer_counter = 0;
2691             _timer_average = (float)(_timer_store)/20.f;
2692             _timer_store = 0;
2693         }
2694         os_update(abst);
2695     }
2696

```

```

2697     }
2698 }
2699
2700 }
2701
2702
2703 int startup() //main function called from win32 abstraction
2704 {
2705     #ifdef WIN32
2706         abst = init_win32_abs();
2707     #endif
2708     os_start(abst);
2709     os_start_thread(abst, run, NULL);
2710     //win32_start_thread(run, NULL);
2711
2712     os_loop_start(abst);
2713     return 0;
2714     /*
2715     printf("Hello World\n");
2716     testWin32();
2717     return 0;*/
2718 }
2719
2720 *****
2721 /* win32.c */
2722 *****
2723
2724 #include <win32.h>
2725 #include <startup.h>
2726 #include <windows.h>
2727 #include <math.h>
2728 #include <stdio.h>
2729 #include <stdint.h>
2730 #include <assert.h>
2731 #include <stdlib.h>
2732 #include <io.h>
2733 #include <fcntl.h>
2734 const char CLASS_NAME[] = "Raytracer";
2735
2736
2737 static win32_context* ctx;
2738
2739
2740 os_abs init_win32_abs()
2741 {
2742     os_abs abstraction;
2743     abstraction.start_func = &win32_start;
2744     abstraction.loop_start_func = &win32_loop;
2745     abstraction.update_func = &win32_update;
2746     abstraction.sleep_func = &win32_sleep;
2747     abstraction.get_bitmap_memory_func = &win32_get_bitmap_memory;
2748     abstraction.get_time_mili_func = &win32_get_time_mili;
2749     abstraction.get_width_func = &win32_get_width;
2750     abstraction.get_height_func = &win32_get_height;
2751     abstraction.start_thread_func = &win32_start_thread;
2752     return abstraction;
2753 }
2754
2755 void* get_bitmap_memory()
2756 {
2757     return ctx->bitmap_memory;
2758 }
2759
2760 void win32_draw_meme()
2761 {
2762     int width = ctx->width;
2763     int height = ctx->height;
2764
2765     int pitch = width*4;
2766     uint8_t* row = (uint8_t*)ctx->bitmap_memory;
2767
2768     for(int y = 0; y < height; y++)
2769     {
2770         uint8_t* pixel = (uint8_t*)row;
2771         for(int x = 0; x < width; x++)
2772         {
2773             *pixel = sin(((float)x)/150)*255;
2774             ++pixel;
2775
2776             *pixel = cos(((float)x)/10)*100;
2777             ++pixel;
2778
2779             *pixel = cos(((float)y)/50)*255;
2780             ++pixel;
2781
2782             *pixel = 0;
2783             ++pixel;

```



```

2784     /* ((char*)ctx->bitmap_memory)[(x+y*width)*4] = (y%2) ? 0xff : 0x00; */
2785     /* ((char*)ctx->bitmap_memory)[(x*4+y*width)+1] = 0x00; */
2786     /* ((char*)ctx->bitmap_memory)[(x*4+y*width)+2] = (y%2) ? 0xff : 0x00; */
2787     /* ((char*)ctx->bitmap_memory)[(x*4+y*width)+3] = 0x00; */
2788 }
2789     row += pitch;
2790 }
2791 }
2792
2793 void win32_sleep(int mili)
2794 {
2795     Sleep(mili);
2796 }
2797
2798 void win32_resize_dib_section(int width, int height)
2799 {
2800     if(ctx->bitmap_memory)
2801         VirtualFree(ctx->bitmap_memory, 0, MEM_RELEASE);
2802
2803     ctx->width = width;
2804     ctx->height = height;
2805
2806     ctx->bitmap_info.bmiHeader.biSize = sizeof(ctx->bitmap_info.bmiHeader);
2807     ctx->bitmap_info.bmiHeader.biWidth = width;
2808     ctx->bitmap_info.bmiHeader.biHeight = -height;
2809     ctx->bitmap_info.bmiHeader.biPlanes = 1;
2810     ctx->bitmap_info.bmiHeader.biBitCount = 32; //8 bits of paddingll
2811     ctx->bitmap_info.bmiHeader.biCompression = BI_RGB;
2812     ctx->bitmap_info.bmiHeader.biSizeImage = 0;
2813     ctx->bitmap_info.bmiHeader.biXPelsPerMeter = 0;
2814     ctx->bitmap_info.bmiHeader.biYPelsPerMeter = 0;
2815     ctx->bitmap_info.bmiHeader.biClrUsed = 0;
2816     ctx->bitmap_info.bmiHeader.biClrImportant = 0;
2817
2818     //I could use BitBlit if it would increase speed.
2819     int bytes_per_pixel = 4;
2820     int bitmap_memory_size = (width*height)*bytes_per_pixel;
2821     ctx->bitmap_memory = VirtualAlloc(0, bitmap_memory_size, MEM_COMMIT, PAGE_READWRITE);
2822 }
2823 }
2824
2825 void win32_update_window(HDC device_context, HWND win, int width, int height)
2826 {
2827
2828     int window_height = height; //window_rect.bottom - window_rect.top;
2829     int window_width = width; //window_rect.right - window_rect.left;
2830
2831
2832     //TODO: Replace with BitBlt this is way too slow... (we don't even need the scaling);
2833     StretchDIBits(device_context,
2834         /* x, y, width, height, */
2835         /* x, y, width, height, */
2836         0, 0, ctx->width, ctx->height,
2837         0, 0, window_width, window_height,
2838
2839         ctx->bitmap_memory,
2840         &ctx->bitmap_info,
2841         DIB_RGB_COLORS, SRCCOPY);
2842 }
2843
2844
2845 LRESULT CALLBACK WndProc(HWND win, UINT msg, WPARAM wParam, LPARAM lParam)
2846 {
2847     switch(msg)
2848     {
2849     case WM_KEYDOWN:
2850         switch (wParam)
2851         {
2852         case VK_ESCAPE:
2853             loop_exit();
2854             ctx->shouldRun = false;
2855             break;
2856
2857         case VK_SPACE:
2858             loop_pause();
2859             break;
2860         default:
2861             break;
2862         }
2863         break;
2864     case WM_SIZE:
2865     {
2866         RECT drawable_rect;
2867         GetClientRect(win, &drawable_rect);
2868
2869         int height = drawable_rect.bottom - drawable_rect.top;
2870         int width = drawable_rect.right - drawable_rect.left;

```

```

2871     win32_resize_dib_section(width, height);
2872
2873     win32_draw_meme();
2874 } break;
2875 case WM_CLOSE:
2876     ctx->shouldRun = false;
2877     break;
2878 case WM_DESTROY:
2879     ctx->shouldRun = false;
2880     break;
2881 case WM_ACTIVATEAPP:
2882     OutputDebugStringA("WM_ACTIVATEAPP\n");
2883     break;
2884 case WM_PAINT:
2885 {
2886     PAINTSTRUCT paint;
2887     HDC device_context = BeginPaint(win, &paint);
2888     EndPaint(win, &paint);
2889
2890     /*int x = paint.rcPaint.Left;
2891     int y = paint.rcPaint.top;
2892     int height = paint.rcPaint.bottom - paint.rcPaint.top;
2893     int width = paint.rcPaint.right - paint.rcPaint.left;*/
2894     //PatBlt(device_context, x, y, width, height, BLACKNESS);
2895
2896     RECT drawable_rect;
2897     GetClientRect(win, &drawable_rect);
2898
2899     int height = drawable_rect.bottom - drawable_rect.top;
2900     int width = drawable_rect.right - drawable_rect.left;
2901
2902     GetClientRect(win, &drawable_rect);
2903     win32_update_window(device_context,
2904                         win, width, height);
2905 } break;
2906 default:
2907     return DefWindowProc(win, msg, wParam, lParam);
2908 }
2909 return 0;
2910 }
2911 }
2912
2913
2914
2915 int _WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
2916             LPSTR lpCmdLine, int nCmdShow)
2917 {
2918
2919     ctx = (win32_context*) malloc(sizeof(win32_context));
2920
2921     ctx->instance = hInstance;
2922     ctx->nCmdShow = nCmdShow;
2923     ctx->wc.cbSize = sizeof(WNDCLASSEX);
2924     ctx->wc.style = CS_OWNDC|CS_HREDRAW|CS_VREDRAW;
2925     ctx->wc.lpfnWndProc = WndProc;
2926     ctx->wc.cbClsExtra = 0;
2927     ctx->wc.cbWndExtra = 0;
2928     ctx->wc.hInstance = hInstance;
2929     ctx->wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
2930     ctx->wc.hCursor = LoadCursor(NULL, IDC_ARROW);
2931     ctx->wc.hbrBackground = 0; //(HBRUSH)(COLOR_WINDOW+1);
2932     ctx->wc.lpszMenuName = NULL;
2933     ctx->wc.lpszClassName = CLASS_NAME;
2934     ctx->wc.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
2935
2936     if(!SetPriorityClass(
2937         GetCurrentProcess(),
2938         HIGH_PRIORITY_CLASS
2939     ))
2940     {
2941         printf("FUCKKKK!!!\n");
2942     }
2943
2944     startup();
2945
2946     return 0;
2947 }
2948
2949 int main()
2950 {
2951     //printf("JANKY WINMAIN OVERRIDE\n");
2952     return _WinMain(GetModuleHandle(NULL), NULL, GetCommandLine(), SW_SHOWNORMAL);
2953 }
2954
2955 //Should Block the Win32 Update Loop.
2956 #define WIN32_SHOULD_BLOCK_LOOP
2957

```

```

2958 void win32_loop()
2959 {
2960     printf("Starting WIN32 Window Loop\n");
2961     MSG msg;
2962     ctx->shouldRun = true;
2963     while(ctx->shouldRun)
2964     {
2965 #ifdef WIN32_SHOULD_BLOCK_LOOP
2966
2967         if(GetMessage(&msg, 0, 0, 0) > 0)
2968         {
2969             TranslateMessage(&msg);
2970             DispatchMessage(&msg);
2971         }
2972     }
2973 #else
2974     while(PeekMessage(&msg, 0, 0, 0, PM_REMOVE))
2975     {
2976         if(msg.message == WM_QUIT)
2977         {
2978             ctx->shouldRun = false;
2979         }
2980         TranslateMessage(&msg);
2981         DispatchMessage(&msg);
2982     }
2983 #endif
2984 //win32_draw_meme();
2985 //win32_update_window();
2986 }
2987 }
2988 }
2989
2990 void create_win32_window()
2991 {
2992     printf("Creating WIN32 Window\n");
2993
2994     ctx->win = CreateWindowEx(
2995         0,
2996         CLASS_NAME,
2997         CLASS_NAME,
2998         /* WS_OVERLAPPEDWINDOW, */
2999         (WS_POPUP | WS_SYSMENU | WS_MAXIMIZEBOX | WS_MINIMIZEBOX),
3000         CW_USEDEFAULT, CW_USEDEFAULT, 1920, 1080,
3001         NULL, NULL, ctx->instance, NULL);
3002
3003     if(ctx->win == NULL)
3004     {
3005         MessageBox(NULL, "Window Creation Failed!", "Error!",
3006             MB_ICONEXCLAMATION | MB_OK);
3007         return;
3008     }
3009
3010     ShowWindow(ctx->win, ctx->nCmdShow);
3011     UpdateWindow(ctx->win);
3012 }
3013
3014 //NOTE: Should the start func start the loop
3015 //define WIN32_SHOULD_START_LOOP_ON_START
3016 void win32_start()
3017 {
3018     if(!RegisterClassEx(&ctx->wc))
3019     {
3020         MessageBox(NULL, "Window Registration Failed!", "Error!",
3021             MB_ICONEXCLAMATION | MB_OK);
3022         return;
3023     }
3024     create_win32_window();
3025 #ifdef WIN32_SHOULD_START_LOOP_ON_START
3026     win32_loop();
3027 #endif
3028 }
3029
3030 int win32_get_time_mili()
3031 {
3032     SYSTEMTIME st;
3033     GetSystemTime(&st);
3034     return (int) st.wMilliseconds+(st.wSecond*1000)+(st.wMinute*1000*60);
3035 }
3036
3037 void win32_update()
3038 {
3039     //RECT win_rect;
3040     //GetClientRect(ctx->win, &win_rect);

```

```

3045     HDC dc = GetDC(ctx->win);
3046     win32_update_window(dc, ctx->win, ctx->width, ctx->height);
3047     ReleaseDC(ctx->win, dc);
3048 }
3049 }
3050
3051
3052 int win32_get_width()
3053 {
3054     return ctx->width;
3055 }
3056
3057 int win32_get_height()
3058 {
3059     return ctx->height;
3060 }
3061
3062 void* win32_get_bitmap_memory()
3063 {
3064     return ctx->bitmap_memory;
3065 }
3066
3067
3068 typedef struct
3069 {
3070     void* data;
3071     void (*func)(void*);
3072 } thread_func_meta;
3073
3074 DWORD WINAPI thread_func(void* data)
3075 {
3076     if(!SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_HIGHEST))
3077     {
3078         DWORD dwError;
3079         dwError = GetLastError();
3080         printf(TEXT("Failed to change thread priority (%d)\n"), dwError);
3081     }
3082
3083     thread_func_meta* meta = (thread_func_meta*) data;
3084     (meta->func)(meta->data); //confusing syntax: call the passed function with the passed data
3085     free(meta);
3086     return 0;
3087 }
3088
3089 void win32_start_thread(void (*func)(void*), void* data)
3090 {
3091     thread_func_meta* meta = (thread_func_meta*) malloc(sizeof(thread_func_meta));
3092     meta->data = data;
3093     meta->func = func;
3094     HANDLE t = CreateThread(NULL, 0, thread_func, meta, 0, NULL);
3095     //if(SetThreadPriority(t, THREAD_PRIORITY_HIGHEST)==0)
3096     //    assert(false);
3097 }
3098 }
3099
3100 /******
3101 /* _compiler_sources.c */
3102 /******
3103 #include <math.h>
3104 #include <stdlib.h>
3105
3106 #define MMX_IMPLEMENTATION
3107 #include <vec.h>
3108 #undef MMX_IMPLEMENTATION
3109 #define TINYOBJ_LOADER_C_IMPLEMENTATION
3110 #include <tinyobj_loader_c.h>
3111 #undef TINYOBJ_LOADER_C_IMPLEMENTATION
3112
3113
3114
3115 #include <parson.c>
3116
3117 #ifdef WIN32
3118 #define WIN32 // I guess CL doesn't add this macro by default...
3119 #endif
3120
3121 #ifdef WIN32
3122 #include <win32.c>
3123 #endif
3124
3125 //TODO: should put in a header
3126 #ifdef WIN32
3127 #define W_ALIGN(x) __declspec( align (x) )
3128 #define U_ALIGN(x) /*nothing*/
3129 #else
3130 #define W_ALIGN(x) /*nothing*/
3131 #define U_ALIGN(x) __attribute__ ((aligned (x)));

```

```

3132 #endif
3133
3134 // #define _MEM_DEBUG // Enable verbose memory allocation, movement and freeing
3135 #include <debug.c>
3136
3137 #include <os_abs.c>
3138 #include <startup.c>
3139 #include <scene.c>
3140 #include <geom.c>
3141 #include <loader.c>
3142 #include <parallel.c>
3143 #include <irradiance_cache.c>
3144 #include <raytracer.c>
3145
3146
3147 /*****
3148  * collision.cl *
3149  *****/
3150
3151 /*****/
3152 /* Types */
3153 /*****/
3154
3155 #define MESH_SCENE_DATA_PARAM imageId_t indices, imageId_t vertices, imageId_t normals
3156 #define MESH_SCENE_DATA      indices, vertices, normals
3157
3158 typedef struct //16 bytes
3159 {
3160     vec3 colour;
3161
3162     float reflectivity;
3163 } __attribute__((aligned(16))) material;
3164
3165 typedef struct
3166 {
3167     vec3 orig;
3168     vec3 dir;
3169 } ray;
3170
3171 typedef struct
3172 {
3173     bool did_hit;
3174     vec3 normal;
3175     vec3 point;
3176     float dist;
3177     material mat;
3178 } collision_result;
3179
3180 typedef struct //32 bytes (one word)
3181 {
3182     vec3 pos;
3183     //4 bytes padding
3184     float radius;
3185     int material_index;
3186     //8 bytes padding
3187 } __attribute__((aligned(16))) sphere;
3188
3189 typedef struct plane
3190 {
3191     vec3 pos;
3192     vec3 normal;
3193
3194     int material_index;
3195 } __attribute__((aligned(16))) plane;
3196
3197 typedef struct
3198 {
3199
3200     mat4 model;
3201
3202     vec3 max;
3203     vec3 min;
3204
3205     int index_offset;
3206     int num_indices;
3207
3208
3209     int material_index;
3210 } __attribute__((aligned(32))) mesh; //TODO: align with cpu NOTE: I don't think we need 32
3211
3212 typedef struct
3213 {
3214     const __global material* material_buffer;
3215     const __global sphere* spheres;
3216     const __global plane* planes;
3217     //Mesh
3218     const __global mesh* meshes;

```

```

3219 } scene;
3220
3221
3222
3223 bool hitBoundingBox(vec3 vmin, vec3 vmax,
3224                     ray r)
3225 {
3226     vec3 tmin = (vmin - r.orig) / r.dir;
3227     vec3 tmax = (vmax - r.orig) / r.dir;
3228
3229     vec3 real_min = min(tmin, tmax);
3230     vec3 real_max = max(tmin, tmax);
3231
3232     vec3 minmax = min(min(real_max.x, real_max.y), real_max.z);
3233     vec3 maxmin = max(max(real_min.x, real_min.y), real_min.z);
3234
3235     if (dot(minmax, minmax) >= dot(maxmin, maxmin))
3236     { return (dot(maxmin, maxmin) > 0.001f ? true : false); }
3237     else return false;
3238 }
3239
3240
3241
3242 /*****/
3243 /*          */
3244 /*   Primitives   */
3245 /*          */
3246 /*****/
3247
3248 /*****/
3249 /* Triangle */
3250 /*****/
3251
3252 //Moller-Trumbore
3253 //t u v = x y z
3254 bool does_collide_triangle(vec3 tri[4], vec3* hit_coords, ray r) //tri has extra for padding
3255 {
3256     vec3 ab = tri[1] - tri[0];
3257     vec3 ac = tri[2] - tri[0];
3258
3259     vec3 pvec = cross(r.dir, ac); //Triple product
3260     float det = dot(ab, pvec);
3261
3262     if (det < EPSILON) // Behind or close to parallel.
3263         return false;
3264
3265     float invDet = 1.f / det;
3266     vec3 tvec = r.orig - tri[0];
3267
3268     //u
3269     hit_coords->y = dot(tvec, pvec) * invDet;
3270     if(hit_coords->y < 0 || hit_coords->y > 1)
3271         return false;
3272
3273     //v
3274     vec3 qvec = cross(tvec, ab);
3275     hit_coords->z = dot(r.dir, qvec) * invDet;
3276     if (hit_coords->z < 0 || hit_coords->y + hit_coords->z > 1)
3277         return false;
3278
3279     //t
3280     hit_coords->x = dot(ac, qvec) * invDet;
3281
3282     return true; //goose
3283 }
3284
3285
3286 /*****/
3287 /* Sphere */
3288 /*****/
3289
3290 bool does_collide_sphere(sphere s, ray r, float *dist)
3291 {
3292     float t0, t1; // solutions for t if the ray intersects
3293
3294     // analytic solution
3295     vec3 L = s.pos - r.orig;
3296     float b = dot(r.dir, L) ;// 2.0f;
3297     float c = dot(L, L) - (s.radius*s.radius); //NOTE: you can optimize out the square.
3298
3299     float disc = b * b - c;//a*/; /* discriminant of quadratic formula */
3300
3301     /* solve for t (distance to hitpoint along ray) */
3302     float t = false;
3303
3304     if (disc < 0.0f) return false;
3305     else t = b - sqrt(disc);

```

```

3306     if (t < 0.0f)
3307     {
3308         t = b + sqrt(disc);
3309         if (t < 0.0f) return false;
3310     }
3311     *dist = t;
3312     return true;
3313 }
3314 }
3315
3316
3317
3318 /*****/
3319 /* PPlane */
3320 /*****/
3321
3322 bool does_collide_plane(plane p, ray r, float *dist)
3323 {
3324     float denom = dot(r.dir, p.normal);
3325     if (denom < EPSILON) //Counter intuitive.
3326     {
3327         vec3 l = p.pos - r.orig;
3328         float t = dot(l, p.normal) / denom;
3329         if (t >= 0)
3330         {
3331             *dist = t;
3332             return true;
3333         }
3334     }
3335     return false;
3336 }
3337 }
3338
3339
3340 /*****/
3341 /*
3342 /* Meshes
3343 /*
3344 /*****/
3345
3346
3347 bool does_collide_with_mesh(mesh collider, ray r, vec3* normal, float* dist, scene s,
3348                             MESH_SCENE_DATA_PARAM)
3349 {
3350     //TODO: k-d trees
3351     *dist = FAR_PLANE;
3352     float min_t = FAR_PLANE;
3353     vec3 hit_coord; //NOTE: currently unused
3354     ray r2 = r;
3355     if(!hitBoundingBox(collider.min, collider.max, r))
3356     {
3357         return false;
3358     }
3359
3360     for(int i = 0; i < collider.num_indices/3; i++) // each ivec3
3361     {
3362         vec3 tri[4];
3363
3364         //get vertex (first element of each index)
3365
3366         int4 idx_0 = read_imagei(indices, i*3+collider.index_offset+0);
3367         int4 idx_1 = read_imagei(indices, i*3+collider.index_offset+1);
3368         int4 idx_2 = read_imagei(indices, i*3+collider.index_offset+2);
3369
3370         tri[0] = read_imagef(vertices, idx_0.x).xyz;
3371         tri[1] = read_imagef(vertices, idx_1.x).xyz;
3372         tri[2] = read_imagef(vertices, idx_2.x).xyz;
3373
3374
3375
3376         vec3 bc_hit_coords = (vec3)(0.f); //t u v = x y z
3377         if(does_collide_triangle(tri, &bc_hit_coords, r) &&
3378            bc_hit_coords.x<min_t && bc_hit_coords.x>0)
3379         {
3380             min_t = bc_hit_coords.x; //t (distance along direction)
3381             *normal =
3382                 read_imagef(normals, idx_0.y).xyz*(1-bc_hit_coords.y-bc_hit_coords.z)+
3383                 read_imagef(normals, idx_1.y).xyz*bc_hit_coords.y+
3384                 read_imagef(normals, idx_2.y).xyz*bc_hit_coords.z;
3385             //break; //convex optimization
3386         }
3387     }
3388 }
3389
3390
3391 *dist = min_t;
3392 return min_t != FAR_PLANE;

```

```

3393 }
3394 }
3395
3396 bool does_collide_with_mesh_alt(mesh collider, ray r, vec3* normal, float* dist, scene s,
3397                                MESH_SCENE_DATA_PARAM)
3398 {
3399     *dist = FAR_PLANE;
3400     float min_t = FAR_PLANE;
3401     vec3 hit_coord; //NOTE: currently unused
3402     ray r2 = r;
3403
3404     for(int i = 0; i < SCENE_NUM_INDICES/3; i++)
3405     {
3406         vec3 tri[4];
3407
3408         //get vertex (first element of each index)
3409
3410         int4 idx_0 = read_imagei(indices, i*3+collider.index_offset+0);
3411         int4 idx_1 = read_imagei(indices, i*3+collider.index_offset+1);
3412         int4 idx_2 = read_imagei(indices, i*3+collider.index_offset+2);
3413
3414         tri[0] = read_imagef(vertices, idx_0.x).xyz;
3415         tri[1] = read_imagef(vertices, idx_1.x).xyz;
3416         tri[2] = read_imagef(vertices, idx_2.x).xyz;
3417
3418
3419         vec3 bc_hit_coords = (vec3)(0.f); //t u v = x y z
3420         if(does_collide_triangle(tri, &bc_hit_coords, r) &&
3421            bc_hit_coords.x<min_t && bc_hit_coords.x>0)
3422         {
3423             min_t = bc_hit_coords.x; //t (distance along direction)
3424             *normal =
3425                 read_imagef(normals, idx_0.y).xyz*(1-bc_hit_coords.y-bc_hit_coords.z)+
3426                 read_imagef(normals, idx_1.y).xyz*bc_hit_coords.y+
3427                 read_imagef(normals, idx_2.y).xyz*bc_hit_coords.z;
3428         }
3429     }
3430 }
3431
3432
3433 *dist = min_t;
3434 return min_t != FAR_PLANE;
3435 }
3436 }
3437
3438
3439
3440 *****
3441 /* High Level Collision */
3442 *****
3443
3444
3445 bool collide_meshes(ray r, collision_result* result, scene s, MESH_SCENE_DATA_PARAM)
3446 {
3447
3448     float dist = FAR_PLANE;
3449     result->did_hit = false;
3450     result->dist = FAR_PLANE;
3451
3452     for(int i = 0; i < SCENE_NUM_MESHES; i++)
3453     {
3454         mesh current_mesh = s.meshes[i];
3455         float local_dist = FAR_PLANE;
3456         vec3 normal;
3457         if(does_collide_with_mesh(current_mesh, r, &normal, &local_dist, s, MESH_SCENE_DATA))
3458         {
3459
3460             if(local_dist<dist)
3461             {
3462                 dist = local_dist;
3463                 result->dist = dist;
3464                 result->normal = normal;
3465                 result->point = (r.dir*dist)+r.orig;
3466                 result->mat = s.material_buffer[current_mesh.material_index];
3467                 result->did_hit = true;
3468             }
3469         }
3470     }
3471     return result->did_hit;
3472 }
3473
3474 bool collide_primitives(ray r, collision_result* result, scene s)
3475 {
3476
3477     float dist = FAR_PLANE;
3478     result->did_hit = false;
3479     result->dist = FAR_PLANE;

```



```

4480 for(int i = 0; i < SCENE_NUM_SPHERES; i++)
4481 {
4482     sphere current_sphere = s.spheres[i]; //get_sphere(spheres, i);
4483     float local_dist = FAR_PLANE;
4484     if(does_collide_sphere(current_sphere, r, &local_dist))
4485     {
4486         if(local_dist < dist)
4487         {
4488             dist = local_dist;
4489             result->did_hit = true;
4490             result->dist = dist;
4491             result->point = r.dir*dist+r.orig;
4492             result->normal = normalize(result->point - current_sphere.pos);
4493             result->mat = s.material_buffer[current_sphere.material_index];
4494         }
4495     }
4496 }
4497
4498 for(int i = 0; i < SCENE_NUM_PLANES; i++)
4499 {
4500     plane current_plane = s.planes[i]; //get_plane(planes, i);
4501     float local_dist = FAR_PLANE;
4502     if(does_collide_plane(current_plane, r, &local_dist))
4503     {
4504         if(local_dist < dist)
4505         {
4506             dist = local_dist;
4507             result->did_hit = true;
4508             result->dist = dist;
4509             result->point = r.dir*dist+r.orig;
4510             result->normal = current_plane.normal;
4511             result->mat = s.material_buffer[current_plane.material_index];
4512         }
4513     }
4514 }
4515
4516 return dist != FAR_PLANE;
4517 }
4518
4519 bool collide_all(ray r, collision_result* result, scene s, MESH_SCENE_DATA_PARAM)
4520 {
4521     float dist = FAR_PLANE;
4522     if(collide_primitives(r, result, s))
4523         dist = result->dist;
4524
4525     collision_result m_result;
4526     if(collide_mesher(r, &m_result, s, MESH_SCENE_DATA))
4527         if(m_result.dist < dist)
4528             *result = m_result;
4529
4530     return result->did_hit;
4531 }
4532
4533
4534
4535 /*****/
4536 /* irradiance_cache.cl */
4537 /*****/
4538 /*****/
4539 /* NOTE: Irradiance Caching is Incomplete */
4540 /*****/
4541
4542 /*****/
4543 /* Irradiance Caching */
4544 /*****/
4545
4546 __kernel void ic_hemisphere_sample(
4547     )
4548 {
4549 {
4550
4551
4552
4553 }
4554
4555 __kernel void ic_screen_textures(
4556     __write_only image2d_t pos_tex,
4557     __write_only image2d_t nrm_tex,
4558     const unsigned int width,
4559     const unsigned int height,
4560     const __global float* ray_buffer,
4561     const vec4 pos,
4562     const __global material* material_buffer,
4563     const __global sphere* spheres,
4564     const __global plane* planes,
4565     const __global mesh* meshes,
4566     image1d_t indices,

```

```

3567     imageId_t vertices,
3568     imageId_t normals)
3569 {
3570     scene s;
3571     s.material_buffer = material_buffer;
3572     s.spheres         = spheres;
3573     s.planes          = planes;
3574     s.meshes          = meshes;
3575
3576
3577     int id = get_global_id(0);
3578     int x  = id%width;
3579     int y  = id/width;
3580     int offset = x*y*width;
3581     int ray_offset = offset*3;
3582
3583     ray r;
3584     r.orig = pos.xyz; //NOTE: slow unaligned memory access.
3585     r.dir.x = ray_buffer[ray_offset];
3586     r.dir.y = ray_buffer[ray_offset+1];
3587     r.dir.z = ray_buffer[ray_offset+2];
3588
3589     collision_result result;
3590     if(!collide_all(r, &result, s, MESH_SCENE_DATA))
3591     {
3592         write_imagef(pos_tex, (int2)(x,y), (vec4)(0));
3593         write_imagef(nrm_tex, (int2)(x,y), (vec4)(0));
3594         return;
3595     }
3596
3597     write_imagef(pos_tex, (int2)(x,y), (vec4)(result.point,0)); //Maybe ???
3598     write_imagef(nrm_tex, (int2)(x,y), (vec4)(result.normal,0));
3599
3600     /* pos_tex[offset] = (vec4)(result.point,0); */
3601     /* nrm_tex[offset] = (vec4)(result.normal,0); */
3602 }
3603
3604
3605
3606 __kernel void generate_discontinuity(
3607     image2d_t pos_tex,
3608     image2d_t nrm_tex,
3609     __global float* out_tex,
3610     const float k,
3611     const float intensity,
3612     const unsigned int width,
3613     const unsigned int height)
3614 {
3615     int id = get_global_id(0);
3616     int x  = id%width;
3617     int y  = id/width;
3618     int offset = x*y*width;
3619
3620     //NOTE: this is fine for edges because the sampler is clamped
3621
3622     //Positions
3623     vec4 pm = read_imagef(pos_tex, sampler, (int2)(x,y));
3624     vec4 pu = read_imagef(pos_tex, sampler, (int2)(x,y+1));
3625     vec4 pd = read_imagef(pos_tex, sampler, (int2)(x,y-1));
3626     vec4 pr = read_imagef(pos_tex, sampler, (int2)(x+1,y));
3627     vec4 pl = read_imagef(pos_tex, sampler, (int2)(x-1,y));
3628
3629     //NOTE: slow doing this many distance calculations
3630     float posDiff = max(distance(pu,pm),
3631                         max(distance(pd,pm),
3632                             max(distance(pr,pm),
3633                                 distance(pl,pm))));
3634     posDiff = clamp(posDiff, 0.f, 1.f);
3635     posDiff *= intensity;
3636
3637     //Normals
3638     vec4 nm = read_imagef(nrm_tex, sampler, (int2)(x,y));
3639
3640     vec4 nu = read_imagef(nrm_tex, sampler, (int2)(x,y+1));
3641     vec4 nd = read_imagef(nrm_tex, sampler, (int2)(x,y-1));
3642     vec4 nr = read_imagef(nrm_tex, sampler, (int2)(x+1,y));
3643     vec4 nl = read_imagef(nrm_tex, sampler, (int2)(x-1,y));
3644     //NOTE: slow doing this many distance calculations
3645     float nrmDiff = max(distance(nu,nm),
3646                         max(distance(nd,nm),
3647                             max(distance(nr,nm),
3648                                 distance(nl,nm))));
3649     nrmDiff = clamp(nrmDiff, 0.f, 1.f);
3650     nrmDiff *= intensity;
3651
3652     out_tex[offset] = k*nrmDiff+posDiff;
3653 }

```

```

3654
3655 __kernel void float_average(
3656     __global float* in_tex,
3657     __global float* out_tex,
3658     const unsigned int width,
3659     const unsigned int height,
3660     const int total)
3661 {
3662     int id = get_global_id(0);
3663     int x = id%width;
3664     int y = id/width;
3665     int offset = x+y*width;
3666
3667     out_tex[offset] += in_tex[offset]/(float)total;
3668 }
3669
3670
3671
3672 __kernel void mip_single_upsample( //nearest neighbour upsample.
3673     __global float* in_tex,
3674     __global float* out_tex,
3675     const unsigned int width, //Of upsampled
3676     const unsigned int height)//Of upsampled
3677 {
3678     int id = get_global_id(0);
3679     int x = id%width;
3680     int y = id/width;
3681     int offset = x+y*width;
3682
3683     out_tex[offset] = in_tex[(x+y*width)/2]; //truncated
3684 }
3685
3686 __kernel void mip_upsample( //nearest neighbour upsample.
3687     image2d_t in_tex,
3688     __write_only image2d_t out_tex, //NOTE: not having __write_only caused it to crash without err
3689     const unsigned int width, //Of upsampled
3690     const unsigned int height)//Of upsampled
3691 {
3692     int id = get_global_id(0);
3693     int x = id%width;
3694     int y = id/width;
3695
3696     write_imagef(out_tex, (int2)(x,y),
3697         read_imagef(in_tex, sampler, (float2)((float)x/2.f, (float)y/2.f)));
3698 }
3699
3700 __kernel void mip_upsample_scaled( //nearest neighbour upsample.
3701     image2d_t in_tex,
3702     __write_only image2d_t out_tex,
3703     const int s,
3704     const unsigned int width, //Of upsampled
3705     const unsigned int height)//Of upsampled
3706 {
3707     int id = get_global_id(0);
3708     int x = id%width;
3709     int y = id/width;
3710     float factor = pow(2.f, (float)s);
3711     write_imagef(out_tex, (int2)(x,y),
3712         read_imagef(in_tex, sampler, (float2)((float)x/factor, (float)y/factor)));
3713 }
3714
3715 __kernel void mip_single_upsample_scaled( //nearest neighbour upsample.
3716     __global float* in_tex,
3717     __global float* out_tex,
3718     const unsigned int s,
3719     const unsigned int width, //Of upsampled
3720     const unsigned int height)//Of upsampled
3721 {
3722     int id = get_global_id(0);
3723     int x = id%width;
3724     int y = id/width;
3725     int factor = (int) pow(2.f, (float)s);
3726     int offset = x+y*width;
3727     int fwidth = width/factor;
3728     int fheight = height/factor;
3729
3730     out_tex[offset] = in_tex[(x/factor)+(y/factor)*(width/factor)]; //truncated
3731 }
3732 //NOTE: not used
3733 __kernel void mip_reduce( //not the best
3734     image2d_t in_tex,
3735     __write_only image2d_t out_tex,
3736     const unsigned int width, //Of reduced
3737     const unsigned int height)//Of reduced
3738 {
3739     int id = get_global_id(0);
3740     int x = id%width;

```

```

3741     int y  = id/width;
3742
3743
3744
3745     vec4 p00 = read_imagef(in_tex, sampler, (int2)(x*2,   y*2  ));
3746
3747     vec4 p01 = read_imagef(in_tex, sampler, (int2)(x*2+1, y*2  ));
3748
3749     vec4 p10 = read_imagef(in_tex, sampler, (int2)(x*2,   y*2+1));
3750
3751     vec4 p11 = read_imagef(in_tex, sampler, (int2)(x*2+1, y*2+1));
3752
3753     write_imagef(out_tex, (int2)(x,y), p00+p01+p10+p11/4.f);
3754 }
3755
3756
3757 /*****/
3758 /* path.cl */
3759 /*****/
3760
3761 vec3 uniformSampleHemisphere(const float r1, const float r2)
3762 {
3763     float sinTheta = sqrt(1 - r1 * r1);
3764     float phi = 2 * M_PI * r2;
3765     float x = sinTheta * cos(phi);
3766     float z = sinTheta * sin(phi);
3767     return (vec3)(x, r1, z);
3768 }
3769 vec3 cosineSampleHemisphere(float u1, float u2, vec3 normal)
3770 {
3771     const float r = sqrt(u1);
3772     const float theta = 2 * M_PI * u2;
3773
3774     vec3 w = normal;
3775     vec3 axis = fabs(w.x) > 0.1f ? (vec3)(0.0f, 1.0f, 0.0f) : (vec3)(1.0f, 0.0f, 0.0f);
3776     vec3 u = normalize(cross(axis, w));
3777     vec3 v = cross(w, u);
3778
3779     /* use the coordinte frame and random numbers to compute the next ray direction */
3780     return normalize(u * cos(theta)*r + v*sin(theta)*r + w*sqrt(1.0f - u1));
3781 }
3782
3783
3784 #define NUM_BOUNCES 8
3785 #define NUM_SAMPLES 64
3786 __kernel void path_trace(
3787     __global vec4* out_tex,
3788     const __global float* ray_buffer,
3789     const __global material* material_buffer,
3790     const __global sphere* spheres,
3791     const __global plane* planes,
3792     //Mesh
3793     const __global mesh* meshes,
3794     image1d_t indices,
3795     image1d_t vertices,
3796     image1d_t normals,
3797     /* const __global vec2* texcoords, */
3798     const unsigned int width,
3799     const vec4 pos,
3800     unsigned int magic)
3801 {
3802     scene s;
3803     s.material_buffer = material_buffer;
3804     s.spheres         = spheres;
3805     s.planes          = planes;
3806     s.meshes          = meshes;
3807
3808
3809     const vec4 sky = (vec4) (0.16, 0.2, 0.2, 0);
3810     //return;
3811     int x = get_global_id(0);
3812     int y = get_global_id(1);
3813     //int x  = id*width+ get_global_offset(0)%total_width;
3814     //int y  = id/width/* + get_global_offset(0)/total_width*/;
3815     int offset = (x+y*width);
3816     int ray_offset = offset*3;
3817
3818     ray r;
3819     r.orig = pos.xyz;
3820     r.dir.x = ray_buffer[ray_offset]; //NOTE: unoptimized memory access.
3821     r.dir.y = ray_buffer[ray_offset+1];
3822     r.dir.z = ray_buffer[ray_offset+2];
3823
3824
3825
3826     union {
3827         float f;

```

```

3828     unsigned int ui;
3829 } res;
3830
3831 res.f = (float)magic*M_PI+x; //fill up the mantissa.
3832 unsigned int seed1 = res.ui + (int)(sin((float)x)*7.f);
3833
3834 res.f = (float)magic*M_PI+y;
3835 unsigned int seed2 = y + (int)(sin((float)res.ui)*7.f);
3836
3837 collision_result initial_result;
3838 if(!collide_all(r, &initial_result, s, MESH_SCENE_DATA))
3839 {
3840     out_tex[x+y*width] = sky;
3841     return;
3842 }
3843
3844 vec3 fin_colour = (vec3)(0.0f, 0.0f, 0.0f);
3845 for(int i = 0; i < NUM_SAMPLES; i++)
3846 {
3847     vec3 accum_color = (vec3)(0.0f, 0.0f, 0.0f);
3848     vec3 mask        = (vec3)(1.0f, 1.0f, 1.0f);
3849     ray sr;
3850     float rand1 = get_random(&seed1, &seed2);
3851     float rand2 = get_random(&seed1, &seed2);
3852
3853     vec3 sample_dir = cosineSampleHemisphere(rand1, rand2, initial_result.normal);
3854     sr.orig = initial_result.point + initial_result.normal * 0.0001f; //sweet spot for epsilon
3855     sr.dir = sample_dir;
3856     mask *= initial_result.mat.colour;
3857     for(int bounces = 0; bounces < NUM_BOUNCES; bounces++)
3858     {
3859         collision_result result;
3860         if(!collide_all(sr, &result, s, MESH_SCENE_DATA))
3861         {
3862             accum_color += mask * sky.xyz;
3863             break;
3864         }
3865
3866         rand1 = get_random(&seed1, &seed2);
3867         rand2 = get_random(&seed1, &seed2);
3868
3869         sample_dir = cosineSampleHemisphere(rand1, rand2, result.normal);
3870
3871         sr.orig = result.point + result.normal * 0.0001f; //sweet spot for epsilon
3872         sr.dir = sample_dir;
3873
3874         //NOTE: janky emission, if reflectivity is 1 emission is 2 (only for tests)
3875         accum_color += mask * (float)(result.mat.reflectivity==1.)*2; //NOTE: EMISSION
3876
3877         mask *= result.mat.colour;
3878
3879         mask *= dot(sample_dir, result.normal);
3880     }
3881     accum_color = clamp(accum_color, 0.f, 1.f);
3882
3883     fin_colour += accum_color * (1.f/NUM_SAMPLES);
3884 }
3885
3886 out_tex[offset] = (vec4)(fin_colour, 0);
3887
3888 }
3889
3890 }
3891
3892
3893 __kernel void buffer_average(
3894     __global uchar4* out_tex,
3895     __global uchar4* fresh_frame_tex,
3896     const unsigned int width,
3897     const unsigned int height,
3898     const unsigned int sample
3899     /*const unsigned int num_samples*/)
3900 {
3901     int id = get_global_id(0);
3902     int x = id%width;
3903     int y = id/width;
3904     int offset = (x + y * width);
3905
3906
3907     float4 temp = mix((float4)(
3908         (float)fresh_frame_tex[offset].x,
3909         (float)fresh_frame_tex[offset].y,
3910         (float)fresh_frame_tex[offset].z,
3911         (float)fresh_frame_tex[offset].w),
3912         (float4)(
3913             (float)out_tex[offset].x,
3914             (float)out_tex[offset].y,

```

```

3915         (float)out_tex[offset].z,
3916         (float)out_tex[offset].w), (float)sample/24.f);
3917     /*vec4 temp = (float)(
3918         (float)fresh_frame_tex[offset].x,
3919         (float)fresh_frame_tex[offset].y,
3920         (float)fresh_frame_tex[offset].z,
3921         (float)fresh_frame_tex[offset].w)/12.f;*/
3922     out_tex[offset] = (uchar4) ((unsigned char)temp.x,
3923         (unsigned char)temp.y,
3924         (unsigned char)temp.z,
3925         (unsigned char)temp.w);
3926     /*
3927         fresh_frame_tex[offset]/(unsigned char)(1.f/(1-(float)sample/255))
3928         + out_tex[offset]/(unsigned char)(1.f/((float)sample/255));*/
3929 }
3930
3931 __kernel void f_buffer_average(
3932     __global vec4* out_tex,
3933     __global vec4* fresh_frame_tex,
3934     const unsigned int width,
3935     const unsigned int height,
3936     const unsigned int num_samples,
3937     const unsigned int sample)
3938 {
3939     int id = get_global_id(0);
3940     int x = id%width;
3941     int y = id/width;
3942     int offset = (x + y * width);
3943     out_tex[offset] = mix(fresh_frame_tex[offset], out_tex[offset],
3944         ((float)sample)/(float)num_samples);
3945 }
3946
3947 __kernel void f_buffer_to_byte_buffer(
3948     __global unsigned int* out_tex,
3949     __global vec4* fresh_frame_tex,
3950     const unsigned int width,
3951     const unsigned int height)
3952 {
3953     int id = get_global_id(0);
3954     int x = id%width;
3955     int y = id/width;
3956     int offset = (x + y * width);
3957     out_tex[offset] = get_colour(fresh_frame_tex[offset]);
3958 }
3959
3960
3961 /*****/
3962 /* general_ray.cl */
3963 /*****/
3964
3965 vec4 shade(collision_result result, scene s, MESH_SCENE_DATA_PARAM)
3966 {
3967     const vec3 light_pos = (vec3)(1,2, 0);
3968     vec3 nspace_light_dir = normalize(light_pos-result.point);
3969     vec4 test_lighting = (vec4) (clamp((float)dot(result.normal, nspace_light_dir), 0.0f, 1.0f));
3970     ray r;
3971     r.dir = nspace_light_dir;
3972     r.orig = result.point + nspace_light_dir*0.01f;
3973     collision_result _cr;
3974     bool visible = !collide_all(r, &_cr, s, MESH_SCENE_DATA);
3975     //test_lighting *= (vec4)(result.mat.colour, 1.0f);
3976     return visible*test_lighting/2;
3977 }
3978
3979
3980 __kernel void cast_ray_test(
3981     __global unsigned int* out_tex,
3982     const __global float* ray_buffer,
3983     const __global material* material_buffer,
3984     const __global sphere* spheres,
3985     const __global plane* planes,
3986     //Mesh
3987     const __global mesh* meshes,
3988     image1d_t indices,
3989     image1d_t vertices,
3990     image1d_t normals,
3991     /* const __global vec2* texcoords, */
3992     /* , */
3993
3994
3995     const unsigned int width,
3996     const unsigned int height,
3997     const vec4 pos)
3998 {
3999     scene s;
4000     s.material_buffer = material_buffer;
4001     s.spheres = spheres;

```

```

4002 s.planes          = planes;
4003 s.meshes          = meshes;
4004
4005 const vec4 sky = (vec4) (0.2, 0.8, 0.5, 0);
4006 //return;
4007 int id = get_global_id(0);
4008 int x  = id%width;
4009 int y  = id/width;
4010 int offset = x+y*width;
4011 int ray_offset = offset*3;
4012
4013
4014 ray r;
4015 r.orig = pos.xyz; //NOTE: unoptimized unaligned memory access.
4016 r.dir.x = ray_buffer[ray_offset];
4017 r.dir.y = ray_buffer[ray_offset+1];
4018 r.dir.z = ray_buffer[ray_offset+2];
4019
4020 //r.dir = (vec3)(0,0,-1);
4021
4022 //out_tex[x+y*width] = get_colour_signed((vec4)(r.dir,0));
4023 //out_tex[x+y*width] = get_colour_signed((vec4)(1,1,0,0));
4024 collision_result result;
4025 if(!collide_all(r, &result, s, MESH_SCENE_DATA))
4026 {
4027     out_tex[x+y*width] = get_colour( sky );
4028     return;
4029 }
4030 vec4 colour = shade(result, s, MESH_SCENE_DATA);
4031
4032
4033 #define NUM_REFLECTIONS 2
4034 ray rays[NUM_REFLECTIONS];
4035 collision_result results[NUM_REFLECTIONS];
4036 vec4 colours[NUM_REFLECTIONS];
4037 int early_exit_num = NUM_REFLECTIONS;
4038 for(int i = 0; i < NUM_REFLECTIONS; i++)
4039 {
4040     if(i==0)
4041     {
4042         rays[i].orig = result.point + result.normal * 0.0001f; //NOTE: BIAS
4043         rays[i].dir = reflect(r.dir, result.normal);
4044     }
4045     else
4046     {
4047         rays[i].orig = results[i-1].point + results[i-1].normal * 0.0001f; //NOTE: BIAS
4048         rays[i].dir = reflect(rays[i-1].dir, results[i-1].normal);
4049     }
4050     if(collide_all(rays[i], results+i, s, MESH_SCENE_DATA))
4051     {
4052         colours[i] = shade(results[i], s, MESH_SCENE_DATA);
4053     }
4054     else
4055     {
4056         colours[i] = sky;
4057         early_exit_num = i;
4058         break;
4059     }
4060 }
4061 for(int i = early_exit_num-1; i > -1; i--)
4062 {
4063     if(i==NUM_REFLECTIONS-1)
4064         colours[i] = mix(colours[i], sky, results[i].mat.reflectivity);
4065
4066     else
4067         colours[i] = mix(colours[i], colours[i+1], results[i].mat.reflectivity);
4068 }
4069
4070 colour = mix(colour, colours[0], result.mat.reflectivity);
4071
4072 out_tex[offset] = get_colour( colour );
4073 }
4074 }
4075
4076
4077 //NOTE: it might be faster to make the ray buffer a multiple of 4 just to align with words...
4078 __kernel void generate_rays(
4079     __global float* out_tex,
4080     const unsigned int width,
4081     const unsigned int height,
4082     const t_mat4 wcm)
4083 {
4084     int id = get_global_id(0);
4085     int x  = id%width;
4086     int y  = id/width;
4087     int offset = (x + y * width) * 3;
4088

```

```

4089 ray r;
4090
4091 float aspect_ratio = width / (float)height; // assuming width > height
4092 float cam_x = (2 * (((float)x + 0.5) / width) - 1) * tan(FOV / 2 * M_PI / 180) * aspect_ratio;
4093 float cam_y = (1 - 2 * (((float)y + 0.5) / height)) * tan(FOV / 2 * M_PI / 180);
4094
4095 //r.orig = matvec((float*)&wcm, (vec4)(0.0, 0.0, 0.0, 1.0)).xyz;
4096 //r.dir = matvec((float*)&wcm, (vec4)(cam_x, cam_y, -1.0f, 1)).xyz - r.orig;
4097
4098 r.orig = (vec3)(0, 0, 0);
4099 r.dir = (vec3)(cam_x, cam_y, -1.0f) - r.orig;
4100
4101 r.dir = normalize(r.dir);
4102
4103 out_tex[offset] = r.dir.x;
4104 out_tex[offset+1] = r.dir.y;
4105 out_tex[offset+2] = r.dir.z;
4106 }
4107
4108 /*****/
4109 /* util.cl */
4110 /*****/
4111 #define FOV 80.0f
4112
4113 #define vec3 float3
4114 #define vec4 float4
4115
4116 #define EPSILON 0.0000001f
4117 #define FAR_PLANE 100000000
4118
4119 typedef float mat4[16];
4120
4121
4122
4123 /*****/
4124 /* Util */
4125 /*****/
4126
4127
4128 __constant sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE |
4129 CLK_ADDRESS_CLAMP_TO_EDGE |
4130 CLK_FILTER_NEAREST;
4131
4132 typedef struct
4133 {
4134     vec4 x;
4135     vec4 y;
4136     vec4 z;
4137     vec4 w;
4138 } __attribute__((aligned (16))) t_mat4;
4139
4140 void swap_float(float *f1, float *f2)
4141 {
4142     float temp = *f2;
4143     *f2 = *f1;
4144     *f1 = temp;
4145 }
4146
4147 vec4 matvec(float* m, vec4 v)
4148 {
4149     return (vec4) (
4150         m[0+0*4]*v.x + m[1+0*4]*v.y + m[2+0*4]*v.z + m[3+0*4]*v.w,
4151         m[0+1*4]*v.x + m[1+1*4]*v.y + m[2+1*4]*v.z + m[3+1*4]*v.w,
4152         m[0+2*4]*v.x + m[1+2*4]*v.y + m[2+2*4]*v.z + m[3+2*4]*v.w,
4153         m[0+3*4]*v.x + m[1+3*4]*v.y + m[2+3*4]*v.z + m[3+3*4]*v.w );
4154 }
4155
4156 unsigned int get_colour(vec4 col)
4157 {
4158     unsigned int outCol = 0;
4159
4160     col = clamp(col, 0.0f, 1.0f);
4161
4162     outCol |= 0xff000000 & (unsigned int)(col.w*255)<<24;
4163     outCol |= 0x00ff0000 & (unsigned int)(col.x*255)<<16;
4164     outCol |= 0x0000ff00 & (unsigned int)(col.y*255)<<8;
4165     outCol |= 0x000000ff & (unsigned int)(col.z*255);
4166
4167
4168     /* outCol |= 0xff000000 & min((unsigned int)(col.w*255), (unsigned int)255)<<24; */
4169     /* outCol |= 0x00ff0000 & min((unsigned int)(col.x*255), (unsigned int)255)<<16; */
4170     /* outCol |= 0x0000ff00 & min((unsigned int)(col.y*255), (unsigned int)255)<<8; */
4171     /* outCol |= 0x000000ff & min((unsigned int)(col.z*255), (unsigned int)255); */
4172     return outCol;
4173 }
4174
4175 static float get_random(unsigned int *seed0, unsigned int *seed1)

```



```

4176 {
4177     /* hash the seeds using bitwise AND operations and bitshifts */
4178     *seed0 = 36969 * ((*seed0) & 65535) + ((*seed0) >> 16);
4179     *seed1 = 18000 * ((*seed1) & 65535) + ((*seed1) >> 16);
4180     unsigned int ires = ((*seed0) << 16) + (*seed1);
4181     /* use union struct to convert int to float */
4182     union {
4183         float f;
4184         unsigned int ui;
4185     } res;
4186
4187     res.ui = (ires & 0x007fffff) | 0x40000000; /* bitwise AND, bitwise OR */
4188     return (res.f - 2.0f) / 2.0f;
4189 }
4190
4191 vec3 reflect(vec3 incidentVec, vec3 normal)
4192 {
4193     return incidentVec - 2.f * dot(incidentVec, normal) * normal;
4194 }
4195
4196 __kernel void blit_float_to_output(
4197     __global unsigned int* out_tex,
4198     __global float* in_flts,
4199     const unsigned int width,
4200     const unsigned int height)
4201 {
4202     int id = get_global_id(0);
4203     int x = id%width;
4204     int y = id/width;
4205     int offset = x+y*width;
4206     out_tex[offset] = get_colour((vec4)(in_flts[offset]));
4207 }
4208
4209 __kernel void blit_float3_to_output(
4210     __global unsigned int* out_tex,
4211     image2d_t in_flts,
4212     const unsigned int width,
4213     const unsigned int height)
4214 {
4215     int id = get_global_id(0);
4216     int x = id%width;
4217     int y = id/width;
4218     int offset = x+y*width;
4219     out_tex[offset] = get_colour(read_imagef(in_flts, sampler, (float2)(x, y)));
4220 }

```