

```

1 #pragma once
2
3 //TODO: @REFACTOR file to just be memory_util
4
5
6 #ifdef _WIN32
7
8 #define W_ALIGN(x) __declspec( align (x) )
9 #define U_ALIGN(x) /*nothing*/
10 //This isn't specifically alignment.
11
12 #define alloca __alloca
13
14 #else
15
16 #define W_ALIGN(x) /*nothing*/
17 #define U_ALIGN(x) __attribute__(( aligned (x) ))
18
19 #endif
20 #pragma once
21 #include <alignment_util.h>
22 #include <stdbool.h>
23 #include <stdint.h>
24
25 typedef int ivec3[4]; //1 int padding
26 typedef float vec2[2];
27 typedef float vec3[4]; //1 float padding
28 typedef float vec4[4];
29 typedef float mat4[16];
30
31 *****/
32 /* Ray */
33 *****/
34 typedef struct ray
35 {
36     vec3 orig;
37     vec3 dir;
38     //float t_min, t_max;
39 } ray; //already aligned
40
41
42 *****/
43 /* Voxel/AABB */
44 *****/
45
46 typedef struct AABB
47 {
48     vec3 max;
49     vec3 min;
50 } AABB;
51
52 void AABB_divide(AABB, uint8_t, float, AABB*, AABB*);
53 void AABB_divide_world(AABB, uint8_t, float, AABB*, AABB*);
54 float AABB_surface_area(AABB);
55 void AABB_clip(AABB, AABB*, AABB* );
56 float AABB_lerp(AABB, uint8_t, float);
57 bool AABB_is_planar(AABB*, uint8_t);
58
59 void AABB_construct_from_vertices(AABB*, vec3*, unsigned int);
60 void AABB_construct_from_triangle(AABB*, ivec3*, vec3*);
61 *****/
62 /* Sphere */
63 *****/
64
65 //NOTE: Less memory efficient but aligns with opengl
66 typedef W_ALIGN(16) struct //sphere
67 {
68     vec4 pos; //GPU stores all vec3s as vec4s in memory so we need the padding.
69
70     float radius;
71     int material_index;
72
73 } U_ALIGN(16) sphere;
74
75
76 float does_collide_sphere(sphere, ray);
77
78
79 *****/
80 /* Plane */
81 *****/
82
83 typedef W_ALIGN(16) struct plane // bytes
84 {
85     vec4 pos; //12
86     //float test;
87
88     vec4 norm;
89     //float test2;
90
91 //32
92     int material_index;
93 } U_ALIGN(16) plane;
94 float does_collide_plane(plane, ray);
95
96 ray generate_ray(int x, int y, int width, int height, float fov);
97 float* matvec_mul(mat4 m, vec4 v);
98 *****/
99 /* NOTE: Irradiance Caching is Incomplete */
100 *****/
101
102 #pragma once
103 #include <stdint.h>
104 #include <alignment_util.h>

```

```

105
106 #define NUM_MIPMAPS 4 //NOTE: 1080/(2^4) != integer
107
108 typedef struct _rt_ctx raytracer_context;
109
110 //    0 = 000: x-, y-, z-
111 //    1 = 001: x-, y-, z+
112 //    2 = 010: x-, y+, z-
113 //    3 = 011: x-, y+, z+
114 //    4 = 100: x+, y-, z-
115 //    5 = 101: x+, y-, z+
116 //    6 = 110: x+, y+, z-
117 //    7 = 111: x+, y+, z+
118
119 typedef struct
120 {
121     vec3 point;
122     vec3 normal;
123
124     float rad;
125
126     vec3 col;
127
128     vec3 gpos;
129     vec3 gdir;
130 } ic_ir_value;
131
132 typedef struct _ic_octree_node ic_octree_node;
133
134 struct _ic_octree_node
135 {
136     bool leaf;
137     bool active;
138
139     union
140     {
141         struct
142         {
143             unsigned int buffer_offset;
144             unsigned int num_elems;
145         } leaf;
146         struct
147         {
148             ic_octree_node* children[8];
149         } branch;
150     } data;
151     vec3 min;
152     vec3 max;
153 };
154
155 typedef struct
156 {
157     ic_octree_node* root;
158     int node_count;
159     unsigned int width;
160     unsigned int max_depth;
161 } ic_octree;
162
163 typedef struct
164 {
165     //vec4* texture;
166     cl_mem cl_image_ref;
167     unsigned int width, height;
168 } ic_mipmap_gb;
169
170
171 typedef struct
172 {
173     //float* texture;
174     cl_mem cl_image_ref;
175     unsigned int width, height;
176 } ic_mipmap_f;
177
178 typedef struct
179 {
180     cl_image_format cl_standard_format;
181     cl_image_desc cl_standard_descriptor;
182     ic_octree octree;
183     ic_ir_value* ir_buf;
184     unsigned int ir_buf_size;
185     unsigned int ir_buf_current_offset;
186 } ic_context;
187
188 void ic_init(raytracer_context*);
189 void ic_screenspace(raytracer_context*);
190 void ic_octree_init_branch(ic_octree_node* );
191 void ic_octree_insert(ic_context*, vec3 point, vec3 normal);
192
193 #pragma once
194 #include <stdint.h>
195 #include <stdbool.h>
196
197 struct scene;
198 //struct AABB;
199 //TODO: make these variable from the ui, eventually
200 #define KDTREE_KT 1.0f //Cost for traversal
201 #define KDTREE_KI 1.5f //Cost for intersection
202
203 #define KDTREE_LEAF 1
204 #define KDTREE_NODE 2
205
206
207 //serializable kd traversal node
208 typedef struct W_ALIGN(16) _skd_tree_traversal_node
209 {

```

```

210     uint8_t type;
211     uint8_t k;
212     float b;
213
214     size_t left_ind; //NOTE: always going to be aligned by at least 8 (could multiply by 8 on gpu)
215     size_t right_ind;
216 } U_ALIGN(16) _skd_tree_traversal_node;
217
218
219 //serializable kd Leaf node
220 typedef struct W_ALIGN(16) _skd_tree_leaf_node
221 {
222     uint8_t type;
223     unsigned int num_triangles;
224     //uint tri 1
225     //uint tri 2
226     //uint etc...
227 } U_ALIGN(16) _skd_tree_leaf_node;
228
229 typedef struct kd_tree_triangle_buffer
230 {
231     unsigned int* triangle_buffer;
232     unsigned int num_triangles;
233 } kd_tree_triangle_buffer;
234
235 //NOTE: not using a vec3 for the floats because it would be a waste of space.
236 typedef struct kd_tree_collision_result
237 {
238     unsigned int triangle_index;
239     float t;
240     float u;
241     float v;
242 } kd_tree_collision_result;
243
244 //NOTE: should the depth be stored in here?
245 typedef struct kd_tree_node
246 {
247     uint8_t k; //Splitting Axis
248     float b; //World Split plane
249
250     struct kd_tree_node* left;
251     struct kd_tree_node* right;
252
253     kd_tree_triangle_buffer triangles;
254
255 } kd_tree_node;
256
257 typedef struct kd_tree
258 {
259     kd_tree_node* root;
260     unsigned int k; //Num dimensions, should always be three in this case
261
262
263     unsigned int num_nodes_total;
264     unsigned int num_tris_padded;
265     unsigned int num_traversal_nodes;
266     unsigned int num_leaves;
267     unsigned int num_indices_total;
268
269     unsigned int max_recurse;
270     unsigned int tri_for_leaf_threshold;
271
272     scene* s;
273     AABB bounds;
274
275     //Serialized form.
276     char* buffer;
277     unsigned int buffer_size;
278     cl_mem cl_kd_tree_buffer;
279
280     //AABB V; //Total bounding box
281
282 } kd_tree;
283
284
285 kd_tree* kd_tree_init();
286 kd_tree_node* kd_tree_node_init();
287
288 bool kd_tree_node_is_leaf(kd_tree_node* );
289 void kd_tree_construct(kd_tree* tree); //O(n Log^2 n) implementation
290 void kd_tree_generate_serialized(kd_tree* tree);
291
292 #pragma once
293 #include <scene.h>
294 #include <alignment_util.h>
295
296 scene* load_scene_json(char* data);
297 scene* load_scene_json_url(char* url);
298 #pragma once
299
300 typedef struct
301 {
302     void (*start_func)();
303     void (*loop_start_func)();
304     void (*update_func)();
305     void (*sleep_func)(int);
306     void (*draw_weird)();
307     void* (*get_bitmap_memory_func)();
308     int (*get_time_mili_func)();
309     int (*get_width_func)();
310     int (*get_height_func)();
311     void (*start_thread_func)(void (*func)(void*), void* data);
312 } os_abs;
313
314 void os_start(os_abs);

```

```

315 void os_loop_start(os_abs);
316 void os_update(os_abs);
317 void os_sleep(os_abs, int);
318 void os_draw_weird(os_abs abs);
319 void* os_get_bitmap_memory(os_abs);
320 int os_get_time_mili(os_abs);
321 int os_get_width(os_abs);
322 int os_get_height(os_abs);
323 void os_start_thread(os_abs, void (*func)(void*), void* data);
324 #pragma once
325 #include <time.h>
326 #include <os_abs.h>
327
328 os_abs init_osx_abs();
329
330 void osx_start();
331 void osx_loop_start();
332 void osx_enqueue_update();
333 void osx_sleep(int milliseconds);
334 void* osx_get_bitmap_memory();
335 int osx_get_time_mili();
336 int osx_get_width();
337 int osx_get_height();
338 void osx_start_thread(void (*func)(void*), void* data);
339 #pragma once
340 #include <alignment_util.h>
341
342 #include <CL/opencl.h>
343 #include <geom.h>
344
345 #define MACRO_GEN(n, t, v, i) \
346     char n[64]; \
347     sprintf(n, "#define " #t, v); \
348     i++; \
349
350
351 typedef struct _rt_ctx raytracer_context;
352
353 typedef struct
354 {
355     cl_platform_id platform_id;           // compute device id
356     cl_device_id device_id;               // compute context
357     cl_context context;                  // compute command queue
358     cl_command_queue commands;           // compute command queue
359
360     unsigned int simt_size;
361     unsigned int num_simt_per_multiprocessor;
362     unsigned int num_multiprocessors;
363     unsigned int num_cores;
364
365 } rcl_ctx;
366
367 typedef struct
368 {
369     cl_program program;
370     cl_kernel* raw_kernels; //NOTE: not a good solution
371     char* raw_data;
372
373 } rcl_program;
374
375 typedef struct rcl_img_buf
376 {
377     cl_mem buffer;
378     cl_mem image;
379     size_t size;
380 } rcl_img_buf;
381
382 void cl_info();
383 void create_context(rcl_ctx* context);
384 void load_program_raw(rcl_ctx* ctx, char* data, char** kernels, unsigned int num_kernels,
385                         rcl_program* program, char** macros, unsigned int num_macros);
386 void load_program_url(rcl_ctx* ctx, char* url, char** kernels, unsigned int num_kernels,
387                         rcl_program* program, char** macros, unsigned int num_macros);
388 void test_sphere_raytracer(rcl_ctx* ctx, rcl_program* program,
389                             sphere* spheres, int num_spheres,
390                             uint32_t* bitmap, int width, int height);
391 cl_mem gen_rgb_image(raytracer_context* rctx,
392                       const unsigned int width,
393                       const unsigned int height);
394 cl_mem gen_grayscale_buffer(raytracer_context* rctx,
395                            const unsigned int width,
396                            const unsigned int height);
397 cl_mem gen_id_image(raytracer_context* rctx, size_t t, void* ptr);
398 rcl_img_buf gen_id_image_buffer(raytracer_context* rctx, size_t t, void* ptr);
399 void retrieve_buf(raytracer_context* rctx, cl_mem g_buf, void* c_buf, size_t);
400
401 void zero_buffer_img(raytracer_context* rctx, cl_mem buf, size_t element,
402                       const unsigned int width,
403                       const unsigned int height);
404 void zero_buffer(raytracer_context* rctx, cl_mem buf, size_t size);
405 size_t get_workgroup_size(raytracer_context* rctx, cl_kernel kernel);
406 #pragma once
407
408 struct _rt_ctx;
409
410 typedef struct path_raytracer_context
411 {
412     struct _rt_ctx* rctx; //General Raytracer Context
413     bool up_to_date;
414
415     unsigned int num_samples;
416     unsigned int current_sample;
417     bool render_complete;
418     int start_time;
419

```

```

420     cl_mem cl_path_output_buffer;
421     cl_mem cl_path_fresh_frame_buffer; //Only exists on GPU TODO: put in path tracer file.
422
423
424 } path_raytracer_context;
425
426 path_raytracer_context* init_path_raytracer_context(struct _rt_ctx* );
427
428 void path_raytracer_render(path_raytracer_context* );
429 void path_raytracer_prepass(path_raytracer_context* );
430 #pragma once
431 #include <alignment_util.h>
432
433 #include <stdint.h>
434 #include <parallel.h>
435 #include <CL/opencl.h>
436 #include <scene.h>
437 #include <irradiance_cache.h>
438
439 #define SS_RAYTRACER 0
440 #define PATH_RAYTRACER 1
441 #define SPLIT_PATH_RAYTRACER 2
442
443 //Cheap, quick, and dirty way of managing kernels.
444 #define KERNELS {"cast_ray_test", "generate_rays", "path_trace",      \
445     "buffer_average", "f_buffer_average",          \
446     "f_buffer_to_byte_buffer",          \
447     "ic_screen_textures", "generate_discontinuity",          \
448     "float_average", "mip_single_upsample", "mip_upsample", \
449     "mip_upsample_scaled", "mip_single_upsample_scaled", \
450     "mip_reduce", "blit_float_to_output",          \
451     "blit_float3_to_output", "kdtree_intersection",          \
452     "kdtree_test_draw", "segmented_path_trace",          \
453     "f_buffer_to_byte_buffer_avg", "segmented_path_trace_init", \
454     "kdtree_ray_draw", "xorshift_batch"} \
455
455 #define NUM_KERNELS 23
456 #define RAY_CAST_KRNL_INDX 0
457 #define RAY_BUFFER_KRNL_INDX 1
458 #define PATH_TRACE_KRNL_INDX 2
459 #define BUFFER_AVG_KRNL_INDX 3
460 #define F_BUFFER_AVG_KRNL_INDX 4
461 #define F_BUF_TO_BYTE_BUF_KRNL_INDX 5
462 #define IC_SCREEN_TEX_KRNL_INDX 6
463 #define IC_GEN_DISC_KRNL_INDX 7
464 #define IC_FLOAT_AVG_KRNL_INDX 8
465 #define IC_MIP_S_UPSAMPLE_KRNL_INDX 9
466 #define IC_MIP_UPSAMPLE_KRNL_INDX 10
467 #define IC_MIP_UPSAMPLE_SCALED_KRNL_INDX 11
468 #define IC_MIP_S_UPSAMPLE_SCALED_KRNL_INDX 12
469 #define IC_MIP_REDUCE_KRNL_INDX 13
470 #define BLIT_FLOAT_OUTPUT_INDX 14
471 #define BLIT_FLOAT3_OUTPUT_INDX 15
472 #define KDTREE_INTERSECTION_INDX 16
473 #define KDTREE_TEST_DRAW_INDX 17
474 #define SEGMENTED_PATH_TRACE_INDX 18
475 #define F_BUF_TO_BYTE_BUF_AVG_KRNL_INDX 19
476 #define SEGMENTED_PATH_TRACE_INIT_INDX 20
477 #define KDREE_RAY_DRAW_INDX 21
478 #define XORSHIFT_BATCH_INDX 22
479
480 typedef struct _rt_ctx raytracer_context;
481
482 typedef struct rt_vtable //NOTE: @REFACTOR not used anymore should delete
483 {
484     bool up_to_date;
485     void (*build)(void* );
486     void (*pre_pass)(void* );
487     void (*render_frame)(void* );
488 } rt_vtable;
489
490
491 struct _rt_ctx
492 {
493     unsigned int width, height;
494
495     float* ray_buffer;
496     vec4* path_output_buffer; //TODO: put in path tracer output
497     uint32_t* output_buffer;
498     //uint32_t* fresh_frame_buffer;
499
500     scene* stat_scene;
501     ic_context* ic_ctx;
502
503     unsigned int block_size_y;
504     unsigned int block_size_x;
505
506     unsigned int event_stack[32];
507     unsigned int event_position;
508
509     //TODO: seperate into contexts for each integrator.
510     //Path tracing only
511
512     unsigned int num_samples;    //TODO: put in path tracer file.
513     unsigned int current_sample; //TODO: put in path tracer file.
514     bool render_complete;
515
516     //CL
517     rcl_ctx* rcl;
518     rcl_program* program;
519
520     cl_mem cl_ray_buffer;
521     cl_mem cl_output_buffer;
522     cl_mem cl_path_output_buffer; //TODO: put in path tracer file
523     cl_mem cl_path_fresh_frame_buffer; //Only exists on GPU TODO: put in path tracer file.
524

```

```

525 };
526
527 raytracer_context* raytracer_init(unsigned int width, unsigned int height,
528                                     uint32_t* output_buffer, rcl_ctx* ctx);
529
530 void raytracer_build(raytracer_context* );
531 void raytracer_prepass(raytracer_context*); //NOTE: I wouldn't call it a prepass, its more like a build
532 void raytracer_render(raytracer_context* );
533 void raytracer_refined_render(raytracer_context* );
534 void _raytracer_gen_ray_buffer(raytracer_context* );
535 void _raytracer_path_trace(raytracer_context*, unsigned int);
536 void _raytracer_average_buffers(raytracer_context*, unsigned int); //NOTE: DEPRECATED
537 void _raytracer_push_path(raytracer_context* );
538 void _raytracer_cast_rays(raytracer_context*); //NOTE: DEPRECATED
539 #pragma once
540 #include <alignment_util.h>
541 #include <vec.h>
542 //typedef struct{} sphere;
543 //struct sphere;
544 //struct plane;
545 //struct kd_tree;
546
547 typedef struct _rt_ctx raytracer_context;
548
549 typedef W_ALIGN(16) struct
550 {
551     vec4 colour;
552
553     float reflectivity;
554
555     //TODO: add more.
556 } U_ALIGN(16) material;
557
558
559
560 typedef W_ALIGN(32) struct
561 {
562     mat4 model;
563
564     vec4 max;
565     vec4 min;
566
567     int index_offset;
568     int num_indices;
569
570     int material_index;
571 } U_ALIGN(32) mesh;
572
573 typedef struct
574 {
575
576     mat4 camera_world_matrix;
577
578     //Materials
579     material* materials;
580     cl_mem cl_material_buffer;
581     unsigned int num_materials;
582     bool materials_changed;
583     //Primitives
584
585     //Spheres
586     sphere* spheres;
587     cl_mem cl_sphere_buffer;
588     unsigned int num_spheres; //NOTE: must be constant.
589     bool spheres_changed;
590     //Planes
591     plane* planes;
592     cl_mem cl_plane_buffer;
593     unsigned int num_planes; //NOTE: must be constant.
594     bool planes_changed;
595
596     //Meshes
597     mesh* meshes; //ALL vertex data is stored contiguously
598     cl_mem cl_mesh_buffer;
599     unsigned int num_meshes;
600     bool meshes_changed;
601
602     //Trying to remember how I got all of the other structs to use typedefs...
603     //kd_tree
604     struct kd_tree* kdt;
605
606
607     //NOTE: we could store vertices, normals, and texcoords contiguously as 1 buffer.
608     vec3* mesh_verts;
609     rcl_img_buf cl_mesh_vert_buffer;
610     unsigned int num_mesh_verts; //NOTE: must be constant.
611
612     vec3* mesh_nrmls;
613     rcl_img_buf cl_mesh_nrml_buffer;
614     unsigned int num_mesh_nrmls; //NOTE: must be constant.
615
616     vec2* mesh_texcoords;
617     rcl_img_buf cl_mesh_texcoord_buffer;
618     unsigned int num_mesh_texcoords; //NOTE: must be constant.
619
620     ivec3* mesh_indices;
621     rcl_img_buf cl_mesh_index_buffer;
622     unsigned int num_mesh_indices; //NOTE: must be constant.
623
624 } scene;
625
626
627 void scene_resource_push(raytracer_context* );
628 void scene_init_resources(raytracer_context* );
629 void scene_generate_resources(raytracer_context* ); //k-d tree generation

```

```

630 #pragma once
631
632 struct _rt_ctx;
633
634
635 typedef struct spath_raytracer_context
636 {
637     struct _rt_ctx* rctx; //General Raytracer Context
638     bool up_to_date;
639
640     unsigned int num_iterations;
641     unsigned int current_iteration;
642     bool render_complete;
643
644     //unsigned int segment_width;
645     //unsigned int segment_offset;
646
647     unsigned int start_time;
648
649     unsigned int* random_buffer;
650
651     cl_mem cl_path_output_buffer;
652     cl_mem cl_path_ray_origin_buffer; //Only exists on GPU
653     cl_mem cl_path_collision_result_buffer; //Only exists on GPU
654     cl_mem cl_spath_progress_buffer; //Only exists on GPU
655     cl_mem cl_path_origin_collision_result_buffer; //Only exists on GPU
656
657     cl_mem cl_random_buffer; //Only exists on GPU
658
659
660     cl_mem cl_bad_api_design_buffer;
661
662
663 } spath_raytracer_context;
664
665 spath_raytracer_context* init_spath_raytracer_context(struct _rt_ctx* );
666
667 void spath_raytracer_render(spath_raytracer_context* );
668 //void ss_raytracer_build(ss_raytracer_context* );
669 void spath_raytracer_prepass(spath_raytracer_context* );
670 #pragma once
671
672 struct _rt_ctx;
673
674 typedef struct ss_raytracer_context
675 {
676     struct _rt_ctx* rctx; //General Raytracer Context
677     bool up_to_date;
678 } ss_raytracer_context;
679
680
681 //TODO: create function table;
682
683 rt_vtable get_ss_raytracer_vtable();
684
685 ss_raytracer_context* init_ss_raytracer_context(struct _rt_ctx* );
686
687 void ss_raytracer_render(ss_raytracer_context* );
688 //void ss_raytracer_build(ss_raytracer_context* );
689 void ss_raytracer_prepass(ss_raytracer_context* );
690 #pragma once
691
692 int startup();
693 void loop_exit();
694 void loop_pause();
695 #pragma once
696
697 struct _rt_ctx;
698
699
700 typedef struct ui_ctx
701 {
702     struct _rt_ctx* rctx; //General Raytracer Context
703
704 } ui_ctx;
705
706 void web_server_start(void* );
707 #pragma once
708 #include <windows.h>
709 #include <stdbool.h>
710 #include <os_abs.h>
711
712 typedef struct
713 {
714     HINSTANCE instance;
715     int nCmdShow;
716     WNDCLASSEX wc;
717     HWND win;
718
719     int width, height;
720
721     BITMAPINFO bitmap_info;
722     void* bitmap_memory;
723
724     // HDC render_device_context;
725
726     bool shouldRun;
727     //Bitbuffer
728 } win32_context;
729
730
731 os_abs init_win32_abs();
732
733 void win32_start_thread(void (*func)(void*), void* data);
734

```

```

735 //void create_win32_window();
736 void win32_start();
738 void win32_loop();
739 void win32_update();
740 void win32_sleep(int);
742 void* win32_get_bitmap_memory();
744 int win32_get_time_mili();
746 int win32_get_width();
748 int win32_get_height();
749 #define CL_TARGET_OPENCL_VERSION 120
750
751 #include <math.h>
752 #include <stdlib.h>
753
754 #define MMX_IMPLEMENTATION
755 #include <vec.h>
756 #undef MMX_IMPLEMENTATION
757 #define TINYOBJ_LOADER_C_IMPLEMENTATION
758 #include <tinyobj_loader_c.h>
759 #undef TINYOBJ_LOADER_C_IMPLEMENTATION
760
761
762 #include <mongoose.c>
763 #include <parson.c>
764
765 #ifdef _WIN32
766 #define WIN32 // I don't want to fix all of my accidents right now.
767 #endif
768
769
770
771 //REMOVE FOR PRESENTATION
772 #define DEV_MODE
773
774
775
776 #ifdef WIN32
777 #include <win32.c>
778 #else
779 #include <osx.m>
780 #endif
781
782 //#define _MEM_DEBUG //Enable verbose memory allocation, movement and freeing
783
784 #include <CL/opencl.h>
785
786 #include <debug.c>
787
788 #include <os_abs.c>
789 #include <startup.c>
790 #include <scene.c>
791 #include <geom.c>
792 #include <Loader.c>
793 #include <parallel.c>
794 #include <ui.c>
795 #include <irradiance_cache.c>
796 #include <raytracer.c>
797 #include <ss_raytracer.c>
798 #include <path_raytracer.c>
799 #include <spath_raytracer.c>
800 #include <kdtree.c>
801 #ifdef _MEM_DEBUG
802 void* _debug_memcpy(void* dest, void* from, size_t size, int line, const char *func)
803 {
804     printf("\n-");
805     memcpy(dest, from, size);
806     printf("- memcpy at %i, %s, %p[%li]\n\n", line, func, dest, size);
807     fflush(stdout);
808     return dest;
809 }
810 void* _debug_malloc(size_t size, int line, const char *func)
811 {
812     printf("\n-");
813     void *p = malloc(size);
814     printf("- Allocation at %i, %s, %p[%li]\n\n", line, func, p, size);
815     fflush(stdout);
816     return p;
817 }
818
819 void _debug_free(void* ptr, int line, const char *func)
820 {
821     printf("\n-");
822     free(ptr);
823     printf("- Free at %i, %s, %p\n\n", line, func, ptr);
824     fflush(stdout);
825 }
826
827 #define malloc(X) _debug_malloc( X, __LINE__, __FUNCTION__)
828 #define free(X) _debug_free( X, __LINE__, __FUNCTION__)
829 #define memcpy(X, Y, Z) _debug_memcpy( X, Y, Z, __LINE__, __FUNCTION__)
830
831
832 #endif
833
834 #ifdef WIN32
835 #define DEBUG_BREAK __debugbreak
836 #define _FILE_SEP '\\'
837 #else
838 #define DEBUG_BREAK
839 #define _FILE_SEP '/'

```

```

840 #endif
841
842 #define __FILENAME__ (strrchr(__FILE__, '_FILE_SEP') ? strrchr(__FILE__, '_FILE_SEP') + 1 : __FILE__)
843
844
845 //TODO: replace all errors with this.
846 #define ASRT_CL(m)
847     if(err!=CL_SUCCESS)
848     {
849         fprintf(stderr, "ERROR: %s. (code: %i, line: %i, file:%s)\nPRESS ENTER TO EXIT\n", \
850             m, err, __LINE__, __FILENAME__);
851         fflush(stderr);
852         while(1){char c; scanf("%c",&c); exit(1);}
853     }
854 //DEBUG_BREAK();
855 #include <geom.h>
856 #define DEBUG_PRINT_VEC3(n, v) printf(n " : (%f, %f, %f)\n", v[0], v[1], v[2])
857
858
859 bool solve_quadratic(float *a, float *b, float *c, float *x0, float *x1)
860 {
861     float discr = (*b) * (*b) - 4 * (*a) * (*c);
862
863     if (discr < 0) return false;
864     else if (discr == 0) {
865         (*x0) = (*x1) = - 0.5 * (*b) / (*a);
866     }
867     else {
868         float q = (*b > 0) ?
869             -0.5 * (*b + sqrt(discr)) :
870             -0.5 * (*b - sqrt(discr));
871         *x0 = q / *a;
872         *x1 = *c / q;
873     }
874
875     return true;
876 }
877
878 float* matvec_mul(mat4 m, vec4 v)
879 {
880     float* out_float = (float*)malloc(sizeof(vec4));
881
882     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];
883     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];
884     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];
885     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];
886
887     return out_float;
888 }
889
890 void swap_float(float *f1, float *f2)
891 {
892     float temp = *f2;
893     *f2 = *f1;
894     *f1 = temp;
895 }
896
897
898 inline void AABB_divide(AABB source, uint8_t k, float b, AABB* left, AABB* right)
899 {
900     vec3 new_min, new_max;
901     memcpy(new_min, source.min, sizeof(vec3));
902     memcpy(new_max, source.max, sizeof(vec3));
903
904     float wrld_split = source.min[k] + (source.max[k] - source.min[k]) * b;
905     new_min[k] = new_max[k] = wrld_split;
906
907     memcpy(left->min, source.min, sizeof(vec3));
908     memcpy(left->max, new_max, sizeof(vec3));
909     memcpy(right->min, new_min, sizeof(vec3));
910     memcpy(right->max, source.max, sizeof(vec3));
911 }
912
913
914 inline void AABB_divide_world(AABB source, uint8_t k, float world_b, AABB* left, AABB* right)
915 {
916     vec3 new_min, new_max;
917     memcpy(new_min, source.min, sizeof(vec3));
918     memcpy(new_max, source.max, sizeof(vec3));
919
920     new_min[k] = new_max[k] = world_b;
921
922     memcpy(left->min, source.min, sizeof(vec3));
923     memcpy(left->max, new_max, sizeof(vec3));
924     memcpy(right->min, new_min, sizeof(vec3));
925     memcpy(right->max, source.max, sizeof(vec3));
926 }
927
928
929 inline float AABB_surface_area(AABB source)
930 {
931     vec3 diff;
932
933     xv_sub(diff, source.max, source.min, 3);
934
935     return (diff[0]*diff[1]*2 +
936             diff[1]*diff[2]*2 +
937             diff[0]*diff[2]*2);
938 }
939
940 inline void AABB_clip(AABB* result, AABB* target, AABB* container)
941 {
942     memcpy(result, target, sizeof(AABB));
943
944     for (int i = 0; i < 3; i++)

```

```

945     {
946         if(result->min[i] < container->min[i])
947             result->min[i] = container->min[i];
948         if(result->max[i] > container->max[i])
949             result->max[i] = container->max[i];
950     }
951 }
952
953 inline void AABB_construct_from_triangle(AABB* result, ivec3* indices, vec3* vertices)
954 {
955     for(int k = 0; k < 3; k++)
956     {
957         result->min[k] = 1000000;
958         result->max[k] = -1000000;
959     }
960
961     for(int i = 0; i < 3; i++)
962     {
963         float* vertex = vertices[indices[i][0]];
964
965         for(int k = 0; k < 3; k++)
966         {
967             if(vertex[k] < result->min[k])
968                 result->min[k] = vertex[k];
969
970             if(vertex[k] > result->max[k])
971                 result->max[k] = vertex[k];
972         }
973     }
974 }
975
976 inline void AABB_construct_from_vertices(AABB* result, vec3* vertices,
977                                         unsigned int num_vertices)
978 {
979     for(int k = 0; k < 3; k++)
980     {
981         result->min[k] = 1000000;
982         result->max[k] = -1000000;
983     }
984     for(int i = 0; i < num_vertices; i++)
985     {
986         for(int k = 0; k < 3; k++)
987         {
988             if(vertices[i][k] < result->min[k])
989                 result->min[k] = vertices[i][k];
990
991             if(vertices[i][k] > result->max[k])
992                 result->max[k] = vertices[i][k];
993         }
994     }
995 }
996
997 inline bool AABB_is_planar(AABB* source, uint8_t k)
998 {
999     if(source->max[k]-source->min[k] == 0.0f) //TODO: use epsilon instead of 0
1000         return true;
1001     return false;
1002 }
1003
1004 inline float AABB_ilerp(AABB source, uint8_t k, float world_b)
1005 {
1006     return (world_b - source.min[k]) / (source.max[k] - source.min[k]);
1007 }
1008
1009 inline float does_collide_sphere(sphere s, ray r)
1010 {
1011     float t0, t1; // solutions for t if the ray intersects
1012
1013     vec3 L;
1014     xv_sub(L, r.orig, s.pos, 3);
1015
1016
1017     float a = 1.0f; //NOTE: we always normalize the direction vector.
1018     float b = xv3_dot(r.dir, L) * 2.0f;
1019     float c = xv3_dot(L, L) - (s.radius*s.radius); //NOTE: square can be optimized out.
1020     if (!solve_quadratic(&a, &b, &c, &t0, &t1)) return -1.0f;
1021
1022     if (t0 > t1) swap_float(&t0, &t1);
1023
1024     if (t0 < 0) {
1025         t0 = t1; // if t0 is negative, use t1 instead
1026         if (t0 < 0) return -1.0f; // both t0 and t1 are negative
1027     }
1028
1029     return t0;
1030 }
1031
1032
1033 inline float does_collide_plane(plane p, ray r)
1034 {
1035     float denom = xv3_dot(r.dir, p.norm);
1036     if (denom > 1e-6)
1037     {
1038         vec3 l;
1039         xv_sub(l, p.pos, r.orig, 3);
1040         float t = xv3_dot(l, p.norm) / denom;
1041         if (t >= 0)
1042             return -1.0;
1043         return t;
1044     }
1045     return -1.0;
1046 }
1047
1048 ray generate_ray(int x, int y, int width, int height, float fov)
1049 {

```

```

1050     ray r;
1051
1052     //Simplified
1053     /* float ndc_x =((float)x+0.5)/width; */
1054     /* float ndc_y =((float)x+0.5)/height; */
1055     /* float screen_x = 2 * ndc_x - 1; */
1056     /* float screen_y = 1 - 2 * ndc_y; */
1057     /* float aspect_ratio = width/height; */
1058     /* float cam_x =(2*screen_x-1) * tan(fov / 2 * M_PI / 180) * aspect_ratio; */
1059     /* float cam_y = (1-2*screen_y) * tan(fov / 2 * M_PI / 180); */
1060
1061     float aspect_ratio = width / (float)height; // assuming width > height
1062     float cam_x = (2 * ((float)x + 0.5) / width - 1) * tan(fov / 2 * M_PI / 180) * aspect_ratio;
1063     float cam_y = (1 - 2 * (((float)y + 0.5) / height)) * tan(fov / 2 * M_PI / 180);
1064
1065
1066     xv3_zero(r.orig);
1067     vec3 v1 = {cam_x, cam_y, -1};
1068     xv_sub(r.dir, v1, r.orig, 3);
1069     xv_normed(r.dir, 3);
1070
1071     return r;
1072 }
1073 /*********************************************************************
1074 /* NOTE: Irradiance Caching is Incomplete */
1075 /*****************************************************************/
1076
1077 #include <irradiance_cache.h>
1078 #include <raytracer.h>
1079 #include <parallel.h>
1080
1081
1082 void ic_init(raytracer_context* rctx)
1083 {
1084     rctx->ic_ctx->cl_standard_format.image_channel_order      = CL_RGBA;
1085     rctx->ic_ctx->cl_standard_format.image_channel_data_type = CL_FLOAT;
1086
1087     rctx->ic_ctx->cl_standard_descriptor.image_type = CL_MEM_OBJECT_IMAGE2D;
1088     rctx->ic_ctx->cl_standard_descriptor.image_width = rctx->width;
1089     rctx->ic_ctx->cl_standard_descriptor.image_height = rctx->height;
1090     rctx->ic_ctx->cl_standard_descriptor.image_depth  = 0;
1091     rctx->ic_ctx->cl_standard_descriptor.image_array_size = 0;
1092     rctx->ic_ctx->cl_standard_descriptor.image_row_pitch = 0;
1093     rctx->ic_ctx->cl_standard_descriptor.num_mip_levels = 0;
1094     rctx->ic_ctx->cl_standard_descriptor.num_samples = 0;
1095     rctx->ic_ctx->cl_standard_descriptor.buffer = NULL;
1096
1097     rctx->ic_ctx->octree.node_count = 1; //root
1098     //TODO: add as parameter
1099     rctx->ic_ctx->octree.max_depth = 8; //arbitrary
1100     rctx->ic_ctx->octree.width    = 15; //arbitrary
1101
1102     rctx->ic_ctx->octree.root = (ic_octree_node*) malloc(sizeof(ic_octree_node));
1103     rctx->ic_ctx->octree.root->min[0] = (float)-rctx->ic_ctx->octree.width;
1104     rctx->ic_ctx->octree.root->min[1] = (float)-rctx->ic_ctx->octree.width;
1105     rctx->ic_ctx->octree.root->min[2] = (float)-rctx->ic_ctx->octree.width;
1106     rctx->ic_ctx->octree.root->max[0] = (float) rctx->ic_ctx->octree.width;
1107     rctx->ic_ctx->octree.root->max[1] = (float) rctx->ic_ctx->octree.width;
1108     rctx->ic_ctx->octree.root->max[2] = (float) rctx->ic_ctx->octree.width;
1109     rctx->ic_ctx->octree.root->leaf = false;
1110     rctx->ic_ctx->octree.root->active = false;
1111 }
1112
1113 void ic_octree_init_leaf(ic_octree_node* node, ic_octree_node* parent, unsigned int i)
1114 {
1115     float xhalf = (parent->max[0]-parent->min[0])/2;
1116     float yhalf = (parent->max[1]-parent->min[1])/2;
1117     float zhalf = (parent->max[2]-parent->min[2])/2;
1118     node->active = false;
1119
1120     node->leaf = true;
1121     for(int i = 0; i < 8; i++)
1122         node->data.branch.children[i] = NULL;
1123     node->min[0] = parent->min[0] + ( (i&4) ? xhalf : 0 );
1124     node->min[1] = parent->min[1] + ( (i&2) ? yhalf : 0 );
1125     node->min[2] = parent->min[2] + ( (i&1) ? zhalf : 0 );
1126     node->max[0] = parent->max[0] - (!i&4) ? xhalf : 0;
1127     node->max[1] = parent->max[1] - (!i&2) ? yhalf : 0;
1128     node->max[2] = parent->max[2] - (!i&1) ? zhalf : 0;
1129 }
1130
1131 void ic_octree_make_branch(ic_octree* tree, ic_octree_node* node)
1132 {
1133
1134     node->leaf = false;
1135     for(int i = 0; i < 8; i++)
1136     {
1137         node->data.branch.children[i] = malloc(sizeof(ic_octree_node));
1138         ic_octree_init_leaf(node->data.branch.children[i], node, i);
1139         tree->node_count++;
1140     }
1141 }
1142
1143 //TODO: test if points are the same
1144 void _ic_octree_rec_resolve(ic_context* ictx, ic_octree_node* leaf, unsigned int node1, unsigned int node2,
1145                             unsigned int depth)
1146 {
1147     if(depth > ictx->octree.max_depth)
1148     {
1149         //TODO: just group buffers together
1150         printf("ERROR: octree reached max depth when trying to resolve collision. (INCOMPLETE)\n");
1151         exit(1);
1152     }
1153     vec3 mid_point;

```

```

1154 xv_sub(mid_point, leaf->max, leaf->min, 3);
1155 xv_divieq(mid_point, 2, 3);
1156 unsigned int i1 =
1157   ((mid_point[0]<ictx->ir_buf[node1].point[0])<<2) |
1158   ((mid_point[1]<ictx->ir_buf[node1].point[1])<<1) |
1159   ((mid_point[2]<ictx->ir_buf[node1].point[2]));
1160 unsigned int i2 =
1161   ((mid_point[0]<ictx->ir_buf[node2].point[0])<<2) |
1162   ((mid_point[1]<ictx->ir_buf[node2].point[1])<<1) |
1163   ((mid_point[2]<ictx->ir_buf[node2].point[2]));
1164 ic_octree_make_branch(&ictx->octree, leaf);
1165 if(i1==i2)
1166   _ic_octree_rec_resolve(ictx, leaf->data.branch.children[i1], node1, node2, depth+1);
1167 else
1168 { //happiness
1169   leaf->data.branch.children[i1]->data.leaf.buffer_offset = node1;
1170   leaf->data.branch.children[i1]->data.leaf.num_elems = 1;
1171   leaf->data.branch.children[i2]->data.leaf.buffer_offset = node2;
1172   leaf->data.branch.children[i2]->data.leaf.num_elems = 1;
1173 }
1174 }
1175
1176 void _ic_octree_rec_insert(ic_context* ictx, ic_octree_node* node, unsigned int v_ptr, unsigned int depth)
1177 {
1178   if(node->leaf && !node->active)
1179   {
1180     node->active = true;
1181     node->data.leaf.buffer_offset = v_ptr;
1182     node->data.leaf.num_elems = 1; //TODO: add suport for more than 1.
1183     return;
1184   }
1185   else if(node->leaf)
1186   {
1187     //resolve
1188     _ic_octree_rec_resolve(ictx, node, v_ptr, node->data.leaf.buffer_offset, depth+1);
1189   }
1190   else
1191   {
1192     ic_octree_node* new_node = node->data.branch.children[
1193       ((ictx->ir_buf[node->data.leaf.buffer_offset].point[0]<ictx->ir_buf[v_ptr].point[0])<<2) |
1194       ((ictx->ir_buf[node->data.leaf.buffer_offset].point[1]<ictx->ir_buf[v_ptr].point[1])<<1) |
1195       ((ictx->ir_buf[node->data.leaf.buffer_offset].point[2]<ictx->ir_buf[v_ptr].point[2]));
1196     _ic_octree_rec_insert(ictx, new_node, v_ptr, depth+1);
1197   }
1198 }
1199
1200 void ic_octree_insert(ic_context* ictx, vec3 point, vec3 normal)
1201 {
1202   if(ictx->ir_buf_current_offset==ictx->ir_buf_size) //TODO: dynamically resize or do something else
1203   {
1204     printf("ERROR: irradiance buffer is full!\n");
1205     exit(1);
1206   }
1207   ic_ir_value irradiance_value; //TODO: EVALUATE THIS
1208   irradiance_value.rad = 0.f; //Gets rid of error, this doesn't work anyways so its good enough.
1209   ictx->ir_buf[ictx->ir_buf_current_offset++] = irradiance_value;
1210   _ic_octree_rec_insert(ictx, ictx->octree.root, ictx->ir_buf_current_offset, 0);
1211 }
1212
1213 //NOTE: outBuffer is only bools but using char for safety accross compilers.
1214 // Also assuming that buf is grayscale
1215 void dither(float* buf, const int width, const int height)
1216 {
1217   for(int y = 0; y < height; y++)
1218   {
1219     for(int x = 0; x < width; x++)
1220     {
1221       float oldpixel = buf[x+y*width];
1222       float newpixel = oldpixel>0.5f ? 1 : 0;
1223       buf[x+y*width] = newpixel;
1224       float err = oldpixel - newpixel;
1225
1226       if( (x != (width-1)) && (x != 0) && (y != (height-1)) )
1227       {
1228         buf[(x+1)+(y )*width] = buf[(x+1)+(y )*width] + err * (7.f / 16.f);
1229         buf[(x+1)+(y+1)*width] = buf[(x+1)+(y+1)*width] + err * (3.f / 16.f);
1230         buf[(x )+(y+1)*width] = buf[(x )+(y+1)*width] + err * (5.f / 16.f);
1231         buf[(x+1)+(y+1)*width] = buf[(x+1)+(y+1)*width] + err * (1.f / 16.f);
1232       }
1233     }
1234   }
1235 }
1236
1237
1238 void get_geom_maps(raytracer_context* rctx, cl_mem positions, cl_mem normals)
1239 {
1240   int err;
1241
1242   cl_kernel kernel = rctx->program->raw_kernels[IC_SCREEN_TEX_KRNL_INDX];
1243
1244   float zeroed[] = {0., 0., 0., 1.};
1245   float* result = matvec_mul(rctx->stat_scene->camera_world_matrix, zeroed);
1246
1247   //SO MANY ARGUMENTS
1248   clSetKernelArg(kernel, 0, sizeof(cl_mem), &positions);
1249   clSetKernelArg(kernel, 1, sizeof(cl_mem), &normals);
1250   clSetKernelArg(kernel, 2, sizeof(int), &rctx->width);
1251   clSetKernelArg(kernel, 3, sizeof(int), &rctx->height);
1252   clSetKernelArg(kernel, 4, sizeof(cl_mem), &rctx->cl_ray_buffer);
1253   clSetKernelArg(kernel, 5, sizeof(vec4), result);
1254   clSetKernelArg(kernel, 6, sizeof(cl_mem), &rctx->stat_scene->cl_material_buffer);
1255   clSetKernelArg(kernel, 7, sizeof(cl_mem), &rctx->stat_scene->cl_sphere_buffer);
1256   clSetKernelArg(kernel, 8, sizeof(cl_mem), &rctx->stat_scene->cl_plane_buffer);
1257   clSetKernelArg(kernel, 9, sizeof(cl_mem), &rctx->stat_scene->cl_mesh_buffer);
1258   clSetKernelArg(kernel, 10, sizeof(cl_mem), &rctx->stat_scene->cl_mesh_index_buffer);

```

```

1259 clSetKernelArg(kernel, 11, sizeof(cl_mem), &rctx->stat_scene->cl_mesh_vert_buffer);
1260 clSetKernelArg(kernel, 12, sizeof(cl_mem), &rctx->stat_scene->cl_mesh_nrm1_buffer);
1261
1262 size_t global = rctx->width*rctx->height;
1263 size_t local = 0;
1264 err = clGetKernelWorkGroupInfo(kernel, rctx->rcl->device_id, CL_KERNEL_WORK_GROUP_SIZE,
1265                               sizeof(local), &local, NULL);
1266 ASRT_CL("Failed to Retrieve Kernel Work Group Info");
1267
1268 err = clEnqueueNDRangeKernel(rctx->rcl->commands, kernel, 1, NULL, &global,
1269                             NULL, 0, NULL, NULL);
1270 ASRT_CL("Failed to Enqueue kernel IC_SCREEN_TEX");
1271
1272 //Wait for completion
1273 err = clFinish(rctx->rcl->commands);
1274 ASRT_CL("Something happened while waiting for kernel to finish");
1275 }
1276
1277 void gen_mipmap_chain_gb(raytracer_context* rctx, cl_mem texture,
1278                           ic_mipmap_gb* mipmaps, int num_mipmaps)
1279 {
1280     int err;
1281     unsigned int width = rctx->width;
1282     unsigned int height = rctx->height;
1283     cl_kernel kernel = rctx->program->raw_kernels[IC_MIP_REDUCE_KRNL_INDX];
1284     for(int i = 0; i < num_mipmaps; i++)
1285     {
1286         mipmaps[i].width = width;
1287         mipmaps[i].height = height;
1288
1289         if(i==0)
1290         {
1291             mipmaps[0].cl_image_ref = texture;
1292
1293             height /= 2;
1294             width /= 2;
1295             continue;
1296         }
1297
1298         clSetKernelArg(kernel, 0, sizeof(cl_mem), &mipmaps[i-1].cl_image_ref);
1299         clSetKernelArg(kernel, 1, sizeof(cl_mem), &mipmaps[i].cl_image_ref);
1300         clSetKernelArg(kernel, 2, sizeof(int), &width);
1301         clSetKernelArg(kernel, 3, sizeof(int), &height);
1302
1303         size_t global = width*height;
1304         size_t local = get_workgroup_size(rctx, kernel);
1305
1306         err = clEnqueueNDRangeKernel(rctx->rcl->commands, kernel, 1,
1307                                       NULL, &global, NULL, 0, NULL, NULL);
1308         ASRT_CL("Failed to Enqueue kernel IC_MIP_REDUCE");
1309
1310         height /= 2;
1311         width /= 2;
1312         //Wait for completion before doing next mip
1313         err = clFinish(rctx->rcl->commands);
1314         ASRT_CL("Something happened while waiting for kernel to finish");
1315     }
1316 }
1317
1318 void upsample_mipmaps_f(raytracer_context* rctx, cl_mem texture,
1319                           ic_mipmap_f* mipmaps, int num_mipmaps)
1320 {
1321     int err;
1322
1323     cl_mem* full_maps = (cl_mem*) alloca(sizeof(cl_mem)*num_mipmaps);
1324     for(int i = 1; i < num_mipmaps; i++)
1325     {
1326         full_maps[i] = gen_grayscale_buffer(rctx, 0, 0);
1327     }
1328     full_maps[0] = texture;
1329     { //Upsample
1330         for(int i = 0; i < num_mipmaps; i++) //First one is already at proper resolution
1331         {
1332             cl_kernel kernel = rctx->program->raw_kernels[IC_MIP_S_UPSAMPLE_SCALED_KRNL_INDX];
1333
1334             clSetKernelArg(kernel, 0, sizeof(cl_mem), &mipmaps[i].cl_image_ref);
1335             clSetKernelArg(kernel, 1, sizeof(cl_mem), &full_maps[i]); //NOTE: need to generate this for the function
1336             clSetKernelArg(kernel, 2, sizeof(int), &i);
1337             clSetKernelArg(kernel, 3, sizeof(int), &rctx->width);
1338             clSetKernelArg(kernel, 4, sizeof(int), &rctx->height);
1339
1340             size_t global = rctx->width*rctx->height;
1341             size_t local = get_workgroup_size(rctx, kernel);
1342
1343             err = clEnqueueNDRangeKernel(rctx->rcl->commands, kernel, 1,
1344                                           NULL, &global, NULL, 0, NULL, NULL);
1345             ASRT_CL("Failed to Enqueue kernel IC_MIP_S_UPSAMPLE_SCALED");
1346
1347         }
1348         err = clFinish(rctx->rcl->commands);
1349         ASRT_CL("Something happened while waiting for kernel to finish");
1350     }
1351     printf("Upsampled Discontinuity Mipmaps\nAveraging Upsampled Discontinuity Mipmaps\n");
1352
1353     { //Average
1354         int total = num_mipmaps;
1355         for(int i = 0; i < num_mipmaps; i++) //First one is already at proper resolution
1356         {
1357             cl_kernel kernel = rctx->program->raw_kernels[IC_FLOAT_AVG_KRNL_INDX];
1358
1359             clSetKernelArg(kernel, 0, sizeof(cl_mem), &full_maps[i]);
1360             clSetKernelArg(kernel, 1, sizeof(cl_mem), &texture);
1361             clSetKernelArg(kernel, 2, sizeof(int), &rctx->width);
1362             clSetKernelArg(kernel, 3, sizeof(int), &rctx->height);
1363             clSetKernelArg(kernel, 4, sizeof(int), &total);

```

```

1364     size_t global = rctx->width*rctx->height;
1365     size_t local = 0;
1366     err = clGetKernelWorkGroupInfo(kernel, rctx->rcl->device_id, CL_KERNEL_WORK_GROUP_SIZE,
1367                                     sizeof(local), &local, NULL);
1368     ASRT_CL("Failed to Retrieve Kernel Work Group Info");
1369
1370     err = clEnqueueNDRangeKernel(rctx->rcl->commands, kernel, 1,
1371                                 NULL, &global, NULL, 0, NULL, NULL);
1372     ASRT_CL("Failed to Enqueue kernel IC_FLOAT_AVG");
1373
1374     err = clFinish(rctx->rcl->commands);
1375     ASRT_CL("Something happened while waiting for kernel to finish");
1376 }
1377
1378 for(int i = 1; i < num_mipmaps; i++)
1379 {
1380     err = clReleaseMemObject(full_maps[i]);
1381     ASRT_CL("Failed to cleanup fullsize mipmaps");
1382 }
1383
1384 }
1385
1386 void gen_discontinuity_maps(raytracer_context* rctx, ic_mipmap_gb* pos_mipmaps,
1387                             ic_mipmap_gb* nrm_mipmaps, ic_mipmap_f* disc_mipmaps,
1388                             int num_mipmaps)
1389 {
1390     int err;
1391     //TODO: tune k and intensity
1392     const float k = 1.6f;
1393     const float intensity = 0.02f;
1394     for(int i = 0; i < num_mipmaps; i++)
1395     {
1396         cl_kernel kernel = rctx->program->raw_kernels[IC_GEN_DISC_KRNL_INDX];
1397         disc_mipmaps[i].width = pos_mipmaps[i].width;
1398         disc_mipmaps[i].height = pos_mipmaps[i].height;
1399
1400         clSetKernelArg(kernel, 0, sizeof(cl_mem), &pos_mipmaps[i].cl_image_ref);
1401
1402         clSetKernelArg(kernel, 1, sizeof(cl_mem), &nrm_mipmaps[i].cl_image_ref);
1403         clSetKernelArg(kernel, 2, sizeof(cl_mem), &disc_mipmaps[i].cl_image_ref);
1404         clSetKernelArg(kernel, 3, sizeof(float), &k);
1405         clSetKernelArg(kernel, 4, sizeof(float), &intensity);
1406         clSetKernelArg(kernel, 5, sizeof(int), &pos_mipmaps[i].width);
1407         clSetKernelArg(kernel, 6, sizeof(int), &pos_mipmaps[i].height);
1408
1409         size_t global = pos_mipmaps[i].width*pos_mipmaps[i].height;
1410         size_t local = get_workgroup_size(rctx, kernel);
1411
1412         err = clEnqueueNDRangeKernel(rctx->rcl->commands, kernel, 1,
1413                                     NULL, &global, NULL, 0, NULL, NULL);
1414         ASRT_CL("Failed to Enqueue kernel IC_GEN_DISC");
1415
1416     }
1417     err = clFinish(rctx->rcl->commands);
1418     ASRT_CL("Something happened while waiting for kernel to finish");
1419 }
1420
1421 void ic_screenspace(raytracer_context* rctx)
1422 {
1423     int err;
1424
1425     vec4* pos_tex = (vec4*) malloc(rctx->width*rctx->height*sizeof(vec4));
1426     vec4* nrm_tex = (vec4*) malloc(rctx->width*rctx->height*sizeof(vec4));
1427     float* c_fin_disc_map = (float*) malloc(rctx->width*rctx->height*sizeof(float));
1428
1429     ic_mipmap_gb pos_mipmaps [NUM_MIPMAPS]; //A lot of buffers
1430     ic_mipmap_gb nrm_mipmaps [NUM_MIPMAPS];
1431     ic_mipmap_f disc_mipmaps[NUM_MIPMAPS];
1432     cl_mem fin_disc_map;
1433
1434     //OpenCL
1435     cl_mem cl_pos_tex;
1436     cl_mem cl_nrm_tex;
1437     cl_image_desc cl_mipmap_descriptor = rctx->ic_ctx->cl_standard_descriptor;
1438
1439     { //OpenCL Init
1440         cl_pos_tex = gen_rgb_image(rctx, 0,0);
1441         cl_nrm_tex = gen_rgb_image(rctx, 0,0);
1442
1443         fin_disc_map = gen_grayscale_buffer(rctx, 0,0);
1444         zero_buffer_img(rctx, fin_disc_map, sizeof(float), 0, 0);
1445
1446
1447         unsigned int width = rctx->width,
1448                     height = rctx->height;
1449         for(int i = 0; i < NUM_MIPMAPS; i++)
1450         {
1451             if(i!=0)
1452             {
1453                 pos_mipmaps[i].cl_image_ref = gen_rgb_image(rctx, width, height);
1454                 nrm_mipmaps[i].cl_image_ref = gen_rgb_image(rctx, width, height);
1455             }
1456             disc_mipmaps[i].cl_image_ref = gen_grayscale_buffer(rctx, width, height);
1457
1458             width /= 2;
1459             height /= 2;
1460         }
1461     }
1462
1463     printf("Initialised Irradiance Cache Screenspace Buffers\nGetting Screenspace Geometry Data\n");
1464     get_geom_maps(rctx, cl_pos_tex, cl_nrm_tex);
1465     printf("Got Screenspace Geometry Data\nGenerating MipMaps\n");
1466     gen_mipmap_chain_gb(rctx, cl_pos_tex,
1467                         pos_mipmaps, NUM_MIPMAPS);
1468     gen_mipmap_chain_gb(rctx, cl_nrm_tex,

```

```

1469         nrm_mipmaps, NUM_MIPMAPS);
1470 printf("Generated MipMaps\nGenerating Discontinuity Map for each Mip\n");
1471 gen_discontinuity_maps(rctx, pos_mipmaps, nrm_mipmaps, disc_mipmaps, NUM_MIPMAPS);
1472 printf("Generated Discontinuity Map for each Mip\nUpsampling Discontinuity Mipmaps\n");
1473 upsample_mipmaps_f(rctx, fin_disc_map, disc_mipmaps, NUM_MIPMAPS);
1474 printf("Averaged Upsampled Discontinuity Mipmaps\nRetrieving Discontinuity Data\n");
1475 retrieve_buf(rctx, fin_disc_map, c_fin_disc_map,
1476             rctx->width*rctx->height*sizeof(float));
1477 retrieve_image(rctx, cl_pos_tex, pos_tex, 0, 0);
1478 retrieve_image(rctx, cl_pos_tex, pos_tex, 0, 0);
1479
1480 printf("Retrieved Discontinuity Data\nDithering Discontinuity Map\n");
1481 //NOTE: read buffer is blocking so we don't need clFinish
1482 dither(c_fin_disc_map, rctx->width, rctx->height);
1483 err = clEnqueueWriteBuffer(rctx->rcl->commands, fin_disc_map,
1484                            CL_TRUE, 0,
1485                            rctx->width*rctx->height*sizeof(float),
1486                            c_fin_disc_map, 0, 0, NULL);
1487 ASRT_CL("Failed to write dithered discontinuity map");
1488
1489 //INSERT
1490 cl_kernel kernel = rctx->program->raw_kernels[BLIT_FLOAT_OUTPUT_INDX];
1491
1492 clSetKernelArg(kernel, 0, sizeof(cl_mem), &rctx->cl_output_buffer);
1493 clSetKernelArg(kernel, 1, sizeof(cl_mem), &fin_disc_map);
1494 clSetKernelArg(kernel, 2, sizeof(int), &rctx->width);
1495 clSetKernelArg(kernel, 3, sizeof(int), &rctx->height);
1496
1497 size_t global = rctx->width*rctx->height;
1498 size_t local = 0;
1499 err = clGetKernelWorkGroupInfo(kernel, rctx->rcl->device_id, CL_KERNEL_WORK_GROUP_SIZE,
1500                                sizeof(local), &local, NULL);
1501 ASRT_CL("Failed to Retrieve Kernel Work Group Info");
1502
1503 err = clEnqueueNDRangeKernel(rctx->rcl->commands, kernel, 1,
1504                               NULL, &global, NULL, 0, NULL, NULL);
1505 ASRT_CL("Failed to Enqueue kernel BLIT_FLOAT_OUTPUT_INDX");
1506
1507 clFinish(rctx->rcl->commands);
1508
1509 err = clEnqueueReadBuffer(rctx->rcl->commands, rctx->cl_output_buffer, CL_TRUE, 0,
1510                           rctx->width*rctx->height*sizeof(int), rctx->output_buffer, 0, NULL, NULL );
1511 ASRT_CL("Failed to Read Output Buffer");
1512 printf("test!!\n");
1513
1514
1515
1516 }
1517 #include <kdtree.h>
1518 #include <scene.h>
1519
1520 #define KDTREE_EPSILON 0.001f
1521
1522 #define KDTREE_BOTH 0
1523 #define KDTREE_LEFT 1
1524 #define KDTREE_RIGHT 2
1525
1526 #define KDTREE_END 0
1527 #define KDTREE_PLANAR 1
1528 #define KDTREE_START 2
1529
1530 //Literally an index buffer to the index buffer
1531 typedef struct kd_tree_event
1532 {
1533     unsigned int tri_index_offset;
1534     float b;
1535     uint8_t k;
1536     uint8_t type;
1537 } kd_tree_event;
1538
1539 typedef struct kd_tree_sah_results
1540 {
1541     float cost;
1542     uint8_t side; //1 left, 2 right
1543 } kd_tree_sah_results;
1544
1545 inline kd_tree_sah_results kd_tree_sah_results_c(float cost, uint8_t side)
1546 {
1547     kd_tree_sah_results r;
1548     r.cost = cost;
1549     r.side = side;
1550     return r;
1551 }
1552
1553 typedef struct kd_tree_find_plane_results
1554 {
1555     kd_tree_event p;
1556     unsigned int NL;
1557     unsigned int NR;
1558     unsigned int NP;
1559     uint8_t side;
1560     float cost;
1561 } kd_tree_find_plane_results;
1562
1563
1564
1565 inline bool kd_tree_event_lt(kd_tree_event* left, kd_tree_event* right)
1566 {
1567     return
1568         (left->b < right->b) ||
1569         (left->b == right->b && left->type < right->type) ||
1570         (left->k > right->k);
1571 }
1572
1573 typedef struct kd_tree_event_buffer

```

```

1574 {
1575     kd_tree_event* events;
1576     unsigned int num_events;
1577
1578 } kd_tree_event_buffer;
1579
1580
1581
1582 //Optional Lambda
1583 float kd_tree_lambda(int NL, int NR, float PL, float PR)
1584 {
1585     if( (NL == 0 || NR == 0) && !(PL == 1.0f || PR == 1.0f) ) //TODO: be less exact for pl pr check, add epsilon
1586         return 0.8f;
1587     return 1.0f;
1588 }
1589
1590 //Cost function
1591 float kd_tree_C(float PL, float PR, uint32_t NL, uint32_t NR)
1592 {
1593     return kd_tree_lambda(NL, NR, PL, PR) *(KDTREE_KT + KDTREE_KI*(PL*NL + PR*NR));
1594 }
1595
1596 kd_tree_sah_results kd_tree_SAH(uint8_t k, float b, AABB V, int NL, int NR, int NP)
1597 {
1598     AABB VL;
1599     AABB VR;
1600     AABB_divide(V, k, b, &VL, &VR);
1601     float PL = AABB_surface_area(VL) / AABB_surface_area(V);
1602     float PR = AABB_surface_area(VR) / AABB_surface_area(V);
1603
1604     if (PL >= 1-KDTREE_EPSILON || PR >= 1-KDTREE_EPSILON) //NOTE: doesn't look like it but potential source of issues
1605         return kd_tree_sah_results_c(1000000000.0f, 0);
1606
1607     float CPL = kd_tree_C(PL, PR, NL+NP, NR);
1608     float CPR = kd_tree_C(PL, PR, NL, NR+NP);
1609
1610
1611     if(CPL < CPR)
1612         return kd_tree_sah_results_c(CPL, KDTREE_LEFT);
1613     else
1614         return kd_tree_sah_results_c(CPR, KDTREE_RIGHT);
1615 }
1616
1617
1618 kd_tree_event_buffer kd_tree_merge_event_buffers(kd_tree_event_buffer buf1, kd_tree_event_buffer buf2)
1619 {
1620     //buffer 1 is guaranteed to be to the direct left of buffer 2
1621     kd_tree_event_buffer event_out;
1622     event_out.num_events = buf1.num_events + buf2.num_events;
1623
1624     event_out.events = (kd_tree_event*) malloc(sizeof(kd_tree_event) * event_out.num_events);
1625
1626
1627     uint32_t buf1_i, buf2_i, eo_i;
1628     buf1_i = buf2_i = eo_i = 0;
1629
1630     while(buf1_i != buf1.num_events || buf2_i != buf2.num_events)
1631     {
1632         if(buf1_i == buf1.num_events)
1633         {
1634             event_out.events[eo_i++] = buf2.events[buf2_i++];
1635             continue;
1636         }
1637
1638         if(buf2_i == buf2.num_events)
1639         {
1640             event_out.events[eo_i++] = buf1.events[buf1_i++];
1641             continue;
1642         }
1643
1644         if( kd_tree_event_lt(buf1.events+buf1_i, buf2.events+buf2_i) )
1645             event_out.events[eo_i++] = buf1.events[buf1_i++];
1646         else
1647             event_out.events[eo_i++] = buf2.events[buf2_i++];
1648     }
1649     assert(eo_i == event_out.num_events);
1650     memcpy(buf1.events, event_out.events, sizeof(kd_tree_event) * event_out.num_events);
1651     free(event_out.events);
1652     event_out.events = buf1.events;
1653
1654     return event_out;
1655 }
1656
1657 kd_tree_event_buffer kd_tree_mergesort_event_buffer(kd_tree_event_buffer buf)
1658 {
1659     if(buf.num_events == 1)
1660         return buf;
1661
1662
1663     int firstHalf = (int)ceil( (float)buf.num_events / 2.f);
1664
1665
1666     kd_tree_event_buffer buf1 = {buf.events, firstHalf, };
1667     kd_tree_event_buffer buf2 = {buf.events+firstHalf, buf.num_events-firstHalf};
1668
1669
1670     buf1 = kd_tree_mergesort_event_buffer(buf1);
1671     buf2 = kd_tree_mergesort_event_buffer(buf2);
1672
1673
1674     return kd_tree_merge_event_buffers(buf1, buf2);
1675 }
1676
1677
1678 }
```

```

1679 kd_tree* kd_tree_init()
1680 {
1681     kd_tree* tree = malloc(sizeof(kd_tree));
1682     tree->root = NULL;
1683     //Defaults
1684     tree->k = 3;
1685     tree->max_recurse = 50;
1686     tree->tri_for_leaf_threshold = 2;
1687     tree->num_nodes_total = 0;
1688     tree->num_tris_padded = 0;
1689     tree->num_traversal_nodes = 0;
1690     tree->num_leaves = 0;
1691     tree->num_indices_total = 0;
1692     tree->buffer_size = 0;
1693     tree->buffer = NULL;
1694     tree->cl_kd_tree_buffer = NULL;
1695     xv3_zero(tree->bounds.min);
1696     xv3_zero(tree->bounds.max);
1697     return tree;
1698 }
1699
1700 kd_tree_node* kd_tree_node_init()
1701 {
1702     kd_tree_node* node = malloc(sizeof(kd_tree_node));
1703     node->k = 0;
1704     node->b = 0.5f; //generic default, shouldn't matter with SAH anyways
1705
1706     node->left = NULL;
1707     node->right = NULL;
1708
1709     return node;
1710 }
1711
1712 bool kd_tree_node_is_leaf(kd_tree_node* node)
1713 {
1714     if(node->left == NULL || node->right == NULL)
1715     {
1716         assert(node->left == NULL && node->right == NULL);
1717         return true;
1718     }
1719
1720     return false;
1721 }
1722
1723
1724
1725 kd_tree_find_plane_results kd_tree_find_plane(kd_tree* tree, AABB V,
1726                                              kd_tree_triangle_buffer tri_buf)
1727 {
1728     float best_cost = INFINITY;
1729     kd_tree_find_plane_results result;
1730
1731
1732     for(int k = 0; k < tree->k; k++)
1733     {
1734         kd_tree_event_buffer event_buf = {NULL, 0}; //gets rid of an initialization warning I guess?
1735         // Generate events
1736         //Divide by three because we only want tris
1737         event_buf.num_events = tri_buf.num_triangles*2;
1738
1739         event_buf.events = malloc(sizeof(kd_tree_event)*event_buf.num_events);
1740         unsigned int j = 0;
1741         for (int i = 0; i < tri_buf.num_triangles; i++)
1742         {
1743             AABB tv, B;
1744             AABB_construct_from_triangle(&tv, tree->s->mesh_indices+tri_buf.triangle_buffer[i],
1745                                         tree->s->mesh_verts);
1746             AABB_clip(&B, &tv, &V);
1747             if(AABB_is_planar(&B, k))
1748             {
1749                 event_buf.events[j++] = (kd_tree_event) {i*3, B.min[k], k, KDTREE_PLANAR};
1750             }
1751             else
1752             {
1753                 event_buf.events[j++] = (kd_tree_event) {i*3, B.min[k], k, KDTREE_START};
1754                 event_buf.events[j++] = (kd_tree_event) {i*3, B.max[k], k, KDTREE_END};
1755             }
1756         }
1757         event_buf.num_events = j;
1758
1759         int last_num_events = event_buf.num_events;
1760         event_buf = kd_tree_mergesort_event_buffer(event_buf);
1761         assert(event_buf.num_events == last_num_events);
1762     }
1763
1764     int NL, NP, NR;
1765     NL = NP = 0;
1766     NR = tri_buf.num_triangles;
1767     for (int i = 0; i < event_buf.num_events;)
1768     {
1769         kd_tree_event p = event_buf.events[i];
1770         int Ps, Pe, Pp;
1771         Ps = Pe = Pp = 0;
1772         while(i < event_buf.num_events && event_buf.events[i].b == p.b && event_buf.events[i].type == KDTREE_END)
1773         {
1774             Pe += 1;
1775             i++;
1776         }
1777         while(i < event_buf.num_events && event_buf.events[i].b == p.b && event_buf.events[i].type == KDTREE_PLANAR)
1778         {
1779             Pp += 1;
1780             i++;
1781         }
1782         while(i < event_buf.num_events && event_buf.events[i].b == p.b && event_buf.events[i].type == KDTREE_START)
1783         {

```



```

1889 kd_tree_node* node = kd_tree_node_init();
1890
1891 tree->num_nodes_total++;
1892 if(kd_tree_should_terminate(tree, tri_buf.num_triangles, V, depth))
1893 {
1894     node->triangles = tri_buf;
1895     tree->num_leaves++;
1896     tree->num_indices_total += tri_buf.num_triangles;
1897     tree->num_tris_padded += tri_buf.num_triangles % 8;
1898     return node;
1899 }
1900
1901 kd_tree_find_plane_results res = kd_tree_find_plane(tree, V, tri_buf);
1902
1903 if(res.cost > KDTREE_KI*(float)tri_buf.num_triangles)
1904 {
1905     node->triangles = tri_buf;
1906     tree->num_leaves++;
1907     tree->num_indices_total += tri_buf.num_triangles;
1908     tree->num_tris_padded += tri_buf.num_triangles % 8;
1909
1910     return node;
1911 }
1912
1913 tree->num_traversal_nodes++;
1914
1915
1916 uint8_t k = res.p.k;
1917 float world_b = res.p.b;
1918
1919 node->k = k;
1920 node->b = world_b; //Local b is honestly useless
1921
1922 assert(node->b != V.min[k]);
1923 assert(node->b != V.max[k]);
1924
1925 AABB VL;
1926 AABB VR;
1927 AABB_divide_world(V, k, world_b, &VL, &VR);
1928
1929 kd_tree_triangle_buffer TL, TR;
1930 kd_tree_classify(tree, tri_buf, res, &TL, &TR);
1931
1932 node->left = kd_tree_construct_rec(tree, VL, TL, depth+1);
1933 node->right = kd_tree_construct_rec(tree, VR, TR, depth+1);
1934
1935
1936 return node;
1937 }
1938
1939 kd_tree_triangle_buffer kd_tree_gen_initial_tri_buf(kd_tree* tree)
1940 {
1941     assert(tree->s->num_mesh_indices % 3 == 0);
1942     kd_tree_triangle_buffer buf;
1943     buf.num_triangles = tree->s->num_mesh_indices/3;
1944     buf.triangle_buffer = (unsigned int*) malloc(sizeof(unsigned int) * buf.num_triangles);
1945
1946     for (int i = 0; i < buf.num_triangles; i++)
1947         buf.triangle_buffer[i] = i * 3;
1948
1949     return buf;
1950 }
1951
1952 void kd_tree_construct(kd_tree* tree) //O(n Log^2 n) implementation
1953 {
1954     assert(tree->s != NULL);
1955
1956     if(tree->s->num_mesh_indices == 0)
1957     {
1958         printf("WARNING: Skipping k-d tree Construction, num_mesh_indices is 0.\n");
1959         return;
1960     }
1961
1962     AABB V;
1963     AABB_construct_from_vertices(&V, tree->s->mesh_verts, tree->s->num_mesh_verts); //works
1964     printf("DBG: kd-tree volume: (%f, %f, %f) (%f, %f, %f)\n", V.min[0], V.min[1], V.min[2], V.max[0], V.max[1], V.max[2]);
1965
1966     tree->bounds = V;
1967
1968     tree->root = kd_tree_construct_rec(tree, V, kd_tree_gen_initial_tri_buf(tree), 0);
1969 }
1970
1971 inline unsigned int _kd_tree_write_buf(char* buffer, unsigned int offset,
1972                                         void* data, size_t size)
1973 {
1974     memcpy(buffer+offset, data, size);
1975     return offset + size;
1976 }
1977
1978 //returns finishing offset
1979 unsigned int kd_tree_generate_serialized_buf_rec(kd_tree* tree, kd_tree_node* node, unsigned int offset)
1980 {
1981     //NOTE: this could really just be two functions
1982     if(kd_tree_node_is_leaf(node)) // Leaf
1983     {
1984
1985         { //Leaf body
1986             _skd_tree_leaf_node l;
1987             l.type = KDTREE_LEAF;
1988             l.num_triangles = node->triangles.num_triangles;
1989             //printf("TEST %u\n", l.num_triangles);
1990             //assert(l.num_triangles != 0);
1991             offset = _kd_tree_write_buf(tree->buffer, offset, &l, sizeof(_skd_tree_leaf_node));
1992         }
1993     }

```

```

1994     for(int i = 0; i < node->triangles.num_triangles; i++) //triangle indices
1995     {
1996         offset = _kd_tree_write_buf(tree->buffer, offset,
1997                                     node->triangles.triangle_buffer+i, sizeof(unsigned int));
1998     }
1999     if(node->triangles.num_triangles % 2)
2000         offset += 4;//if it isn't aligned with a Long add 4 bytes (8 byte alignment)
2001
2002     return offset;
2003 }
2004 else // traversal node
2005 {
2006     _skd_tree_traversal_node n;
2007     n.type = KDREE_NODE;
2008     n.k = node->k;
2009     n.b = node->b;
2010     unsigned int struct_start_offset = offset;
2011     offset += sizeof(_skd_tree_traversal_node);
2012
2013     unsigned int left_offset = kd_tree_generate_serialized_buf_rec(tree, node->left, offset);
2014     //this goes after the left node
2015     unsigned int right_offset = kd_tree_generate_serialized_buf_rec(tree, node->right, left_offset);
2016
2017     n.left_ind = offset;
2018     n.right_ind = left_offset;
2019
2020     memcpy(tree->buffer+struct_start_offset, &n, sizeof(_skd_tree_traversal_node));
2021
2022     return right_offset;
2023 }
2024 }
2025
2026 void kd_tree_generate_serialized(kd_tree* tree)
2027 {
2028     if(tree->s->num_mesh_indices == 0)
2029     {
2030         printf("WARNING: Skipping k-d tree Serialization, num_mesh_indices is 0.\n");
2031         tree->buffer_size = 0;
2032         tree->buffer = malloc(1);
2033         return;
2034     }
2035
2036     unsigned int mem_needed = 0;
2037
2038     mem_needed += tree->num_traversal_nodes * sizeof(_skd_tree_traversal_node); //traversal nodes
2039     mem_needed += tree->num_leaves * sizeof(_skd_tree_leaf_node); //leaf nodes
2040     mem_needed += (tree->num_indices_total+tree->num_tris_padded) * sizeof(unsigned int); //triangle indices
2041
2042     tree->buffer_size = mem_needed;
2043     printf("k-d tree is %d bytes long...", mem_needed);
2044
2045     tree->buffer = malloc(mem_needed);
2046     kd_tree_generate_serialized_buf_rec(tree, tree->root, 0);
2047
2048 }
2049 #include <Loader.h>
2050 #include <parson.h>
2051 #include <vec.h>
2052 #include <float.h>
2053 #include <tinyobj_loader_c.h>
2054 #include <assert.h>
2055
2056
2057
2058 #ifndef WIN32
2059 #include <libproc.h>
2060 #include <unistd.h>
2061
2062 #define FILE_SEP '/'
2063
2064 char* _get_os_pid_bin_path()
2065 {
2066     static bool initialised = false;
2067     static char path[PROC_PIDPATHINFO_MAXSIZE];
2068     if(!initialised)
2069     {
2070         int ret;
2071         pid_t pid;
2072         //char path[PROC_PIDPATHINFO_MAXSIZE];
2073
2074         pid = getpid();
2075         ret = proc_pidpath(pid, path, sizeof(path));
2076
2077         if(ret <= 0)
2078         {
2079             printf("Error: couldn't get bin path.\n");
2080             exit(1);
2081         }
2082         *strrchr(path, FILE_SEP) = '\0';
2083     }
2084     printf("TEST: %s !\n", path);
2085     return path;
2086 }
2087 #else
2088 #include <windows.h>
2089 #define FILE_SEP '\\'
2090
2091 char* _get_os_pid_bin_path()
2092 {
2093     static bool initialised = false;
2094     static char path[260];
2095     if(!initialised)
2096     {
2097         HMODULE hModule = GetModuleHandleW(NULL);
2098

```

```

2099     WCHAR tpath[260];
2100     GetModuleFileNameW(hModule, tpath, 260);
2101
2102     char DefChar = ' ';
2103     WideCharToMultiByte(CP_ACP, 0, tpath, -1, path, 260, &DefChar, NULL);
2104
2105     *(strrchr(path, FILE_SEP)) = '\0'; //get last occurrence;
2106
2107 }
2108
2109 }
2110 #endif
2111
2112 char* load_file(const char* url, long *ret_length)
2113 {
2114     char real_url[260];
2115     sprintf(real_url, "%s%cres%c%s", _get_os_pid_bin_path(), FILE_SEP, FILE_SEP, url);
2116
2117     char * buffer = 0;
2118     long length;
2119     FILE * f = fopen (real_url, "rb");
2120
2121     if (f)
2122     {
2123         fseek (f, 0, SEEK_END);
2124         length = ftell (f)+1;
2125         fseek (f, 0, SEEK_SET);
2126         buffer = malloc (length);
2127         if (buffer)
2128         {
2129             fread (buffer, 1, length, f);
2130         }
2131         fclose (f);
2132     }
2133     if (buffer)
2134     {
2135         buffer[length-1] = '\0';
2136
2137         *ret_length = length;
2138         return buffer;
2139     }
2140     else
2141     {
2142         printf("Error: Couldn't load file '%s'.\n", real_url);
2143         exit(1);
2144     }
2145 }
2146
2147
2148 //Linked List for Mesh Loading
2149 struct obj_list_elem
2150 {
2151     struct obj_list_elem* next;
2152     tinyobj_attrib_t attrib;
2153     tinyobj_shape_t* shapes;
2154     size_t num_shapes;
2155     int mat_index;
2156     mat4 model_mat;
2157 };
2158
2159 void obj_pre_load(char* data, long data_len, struct obj_list_elem* elem,
2160                     int* num_meshes, unsigned int* num_indices, unsigned int* num_vertices,
2161                     unsigned int* num_normals, unsigned int* num_texcoords)
2162 {
2163
2164     tinyobj_material_t* materials = NULL; //NOTE: UNUSED
2165     size_t num_materials; //NOTE: UNUSED
2166
2167
2168     {
2169         unsigned int flags = TINYOBJ_FLAG_TRIANGULATE;
2170         int ret = tinyobj_parse_obj(&elem->attrib, &elem->shapes, &elem->num_shapes, &materials,
2171                                     &num_materials, data, data_len, flags);
2172         if (ret != TINYOBJ_SUCCESS) {
2173             printf("Error: Couldn't parse mesh.\n");
2174             exit(1);
2175         }
2176     }
2177
2178     *num_vertices += elem->attrib.num_vertices;
2179     *num_normals += elem->attrib.num_normals;
2180     *num_texcoords += elem->attrib.num_texcoords;
2181     *num_meshes += elem->num_shapes;
2182     //tinyobjloader has dumb variable names: attrib.num_faces = num_vertices+num_faces
2183     *num_indices += elem->attrib.num_faces;
2184 }
2185
2186
2187
2188 void load_obj(struct obj_list_elem elem, int* mesh_offset, int* vert_offset, int* nrml_offset,
2189                 int* texcoord_offset, int* index_offset, scene* out_scene)
2190 {
2191     for(int i = 0; i < elem.num_shapes; i++)
2192     {
2193         tinyobj_shape_t shape = elem.shapes[i];
2194
2195         //Get mesh and increment offset.
2196         mesh* m = (out_scene->meshes) + (*mesh_offset)++;
2197
2198         m->min[0] = m->min[1] = m->min[2] = FLT_MAX;
2199         m->max[0] = m->max[1] = m->max[2] = -FLT_MAX;
2200
2201         memcpy(m->model, elem.model_mat, 4*4*sizeof(float));
2202
2203         m->index_offset = *index_offset;

```

```

2204     m->num_indices = shape.length*3;
2205     m->material_index = elem.mat_index;
2206
2207     for(int f = 0; f < shape.length; f++)
2208     {
2209         //TODO: don't do this error check for each iteration
2210         if(elem.attrib.face_num_verts[f+shape.face_offset]!=3)
2211         {
2212             //This should never get called because the mesh gets triangulated when loaded.
2213             printf("Error: the obj loader only supports triangulated meshes!\n");
2214             exit(1);
2215         }
2216         for(int j = 0; j < 3; j++)
2217         {
2218             tinyobj_vertex_index_t face_index = elem.attrib.faces[(f+shape.face_offset)*3+j];
2219
2220             vec3 vertex;
2221             vertex[0] = elem.attrib.vertices[3*face_index.v_idx+0];
2222             vertex[1] = elem.attrib.vertices[3*face_index.v_idx+1];
2223             vertex[2] = elem.attrib.vertices[3*face_index.v_idx+2];
2224
2225             m->min[0] = vertex[0] < m->min[0] ? vertex[0] : m->min[0]; //X min
2226             m->min[1] = vertex[1] < m->min[1] ? vertex[1] : m->min[1]; //Y min
2227             m->min[2] = vertex[2] < m->min[2] ? vertex[2] : m->min[2]; //Z min
2228
2229             m->max[0] = vertex[0] > m->max[0] ? vertex[0] : m->max[0]; //X max
2230             m->max[1] = vertex[1] > m->max[1] ? vertex[1] : m->max[1]; //Y max
2231             m->max[2] = vertex[2] > m->max[2] ? vertex[2] : m->max[2]; //Z max
2232
2233             ivec3 index;
2234             index[0] = (*vert_offset)+face_index.v_idx;
2235             index[1] = (*nrm_offset)+face_index.vn_idx;
2236             index[2] = (*texcoord_offset)-face_index.vt_idx;
2237             out_scene->mesh_indices[(*index_offset)][0] = index[0];
2238             out_scene->mesh_indices[(*index_offset)][1] = index[1];
2239             out_scene->mesh_indices[(*index_offset)][2] = index[2];
2240             //Sorry to anyone reading this line...
2241             *((int*)out_scene->mesh_indices[(*index_offset)])+3 = (*mesh_offset)-1; //current mesh
2242
2243             //xv3_cpy(out_scene->mesh_indices + (*index_offset), index);
2244             (*index_offset)++;
2245         }
2246     }
2247 }
2248
2249 //__debugbreak();
2250
2251
2252 //GPU MEMORY ALIGNMENT FUN
2253 //NOTE: this is done because the gpu stores all vec3s 4 floats for memory alignment
2254 //      and it is actually faster if they are aligned like this even
2255 //      though it wastes more memory.
2256 for(int i = 0; i < elem.attrib.num_vertices; i++)
2257 {
2258
2259     memcpy(out_scene->mesh_verts + (*vert_offset),
2260           elem.attrib.vertices+3*i,
2261           sizeof(float)*3); //even though our buffer is aligned theres is
2262     (*vert_offset) += 1;
2263 }
2264 for(int i = 0; i < elem.attrib.num_normals; i++)
2265 {
2266     memcpy(out_scene->mesh_nrm1s + (*nrm_offset),
2267           elem.attrib.normals+3*i,
2268           sizeof(float)*3);
2269     (*nrm_offset) += 1;
2270 }
2271 //NOTE: the texcoords are already aligned because they only have 2 elements.
2272 memcpy(out_scene->mesh_texcoords + (*texcoord_offset), elem.attrib.texcoords,
2273         elem.attrib.num_texcoords*sizeof(vec2));
2274 (*texcoord_offset) += elem.attrib.num_texcoords;
2275 }
2276
2277 scene* load_scene_json(char* json)
2278 {
2279     printf("Beginning scene loading...\n");
2280     scene* out_scene = (scene*) malloc(sizeof(scene));
2281     JSON_Value *root_value;
2282     JSON_Object *root_object;
2283     root_value = json_parse_string(json);
2284     root_object = json_value_get_object(root_value);
2285
2286
2287     //Name
2288     {
2289         const char* name = json_object_get_string(root_object, "name");
2290         printf("Scene name: %s\n", name);
2291     }
2292
2293     //Version
2294     //{
2295         int major = (int)json_object_dotget_number(root_object, "version.major");
2296         int minor = (int)json_object_dotget_number(root_object, "version.major");
2297         const char* type = json_object_dotget_string(root_object, "version.type");
2298     }
2299
2300     //Materials
2301     {
2302         JSON_Array* material_array = json_object_get_array(root_object, "materials");
2303         out_scene->num_materials = json_array_get_count(material_array);
2304         out_scene->materials = (material*) malloc(out_scene->num_materials*sizeof(material));
2305         assert(out_scene->num_materials>0);
2306         for(int i = 0; i < out_scene->num_materials; i++)
2307         {
2308             JSON_Object* mat = json_array_get_object(material_array, i);

```

```

2309     xv_x(out_scene->materials[i].colour) = json_object_get_number(mat, "r");
2310     xv_y(out_scene->materials[i].colour) = json_object_get_number(mat, "g");
2311     xv_z(out_scene->materials[i].colour) = json_object_get_number(mat, "b");
2312     out_scene->materials[i].reflectivity = json_object_get_number(mat, "reflectivity");
2313 }
2314 printf("Materials: %d\n", out_scene->num_materials);
2315 }
2316
2317 //Primitives
2318 {
2319
2320     JSON_Object* primitive_object = json_object_get_object(root_object, "primitives");
2321
2322     //Spheres
2323     {
2324         JSON_Array* sphere_array = json_object_get_array(primitive_object, "spheres");
2325         int num_spheres = json_array_get_count(sphere_array);
2326
2327         out_scene->spheres = malloc(sizeof(sphere)*num_spheres);
2328         out_scene->num_spheres = num_spheres;
2329
2330         for(int i = 0; i < num_spheres; i++)
2331         {
2332             JSON_Object* sphere = json_array_get_object(sphere_array, i);
2333             out_scene->spheres[i].pos[0] = json_object_get_number(sphere, "x");
2334             out_scene->spheres[i].pos[1] = json_object_get_number(sphere, "y");
2335             out_scene->spheres[i].pos[2] = json_object_get_number(sphere, "z");
2336             out_scene->spheres[i].radius = json_object_get_number(sphere, "radius");
2337             out_scene->spheres[i].material_index = json_object_get_number(sphere, "mat_index");
2338         }
2339         printf("Spheres: %d\n", out_scene->num_spheres);
2340     }
2341
2342     //Planes
2343     {
2344         JSON_Array* plane_array = json_object_get_array(primitive_object, "planes");
2345         int num_planes = json_array_get_count(plane_array);
2346
2347         out_scene->planes = malloc(sizeof(plane)*num_planes);
2348         out_scene->num_planes = num_planes;
2349
2350         for(int i = 0; i < num_planes; i++)
2351         {
2352             JSON_Object* plane = json_array_get_object(plane_array, i);
2353             out_scene->planes[i].pos[0] = json_object_get_number(plane, "x");
2354             out_scene->planes[i].pos[1] = json_object_get_number(plane, "y");
2355             out_scene->planes[i].pos[2] = json_object_get_number(plane, "z");
2356             out_scene->planes[i].norm[0] = json_object_get_number(plane, "nx");
2357             out_scene->planes[i].norm[1] = json_object_get_number(plane, "ny");
2358             out_scene->planes[i].norm[2] = json_object_get_number(plane, "nz");
2359
2360             out_scene->planes[i].material_index = json_object_get_number(plane, "mat_index");
2361         }
2362         printf("Planes: %d\n", out_scene->num_planes);
2363     }
2364 }
2365
2366
2367
2368 //Meshes
2369 {
2370     JSON_Array* mesh_array = json_object_get_array(root_object, "meshes");
2371
2372     int num_meshes = json_array_get_count(mesh_array);
2373
2374     out_scene->num_meshes = 0;
2375     out_scene->num_mesh_verts = 0;
2376     out_scene->num_mesh_nrmls = 0;
2377     out_scene->num_mesh_texcoords = 0;
2378     out_scene->num_mesh_indices = 0;
2379
2380
2381     struct obj_list_elem* first = (struct obj_list_elem*) malloc(sizeof(struct obj_list_elem));
2382     struct obj_list_elem* current = first;
2383
2384     //Pre evaluation
2385     for(int i = 0; i < num_meshes; i++)
2386     {
2387         JSON_Object* mesh = json_array_get_object(mesh_array, i);
2388         const char* url = json_object_get_string(mesh, "url");
2389         long length;
2390         char* data = load_file(url, &length);
2391         obj_pre_load(data, length, current, &out_scene->num_meshes, &out_scene->num_mesh_indices,
2392                     &out_scene->num_mesh_verts, &out_scene->num_mesh_nrmls,
2393                     &out_scene->num_mesh_texcoords);
2394         current->mat_index = (int) json_object_get_number(mesh, "mat_index");
2395         //mat4 model_mat;
2396         {
2397             //xm4_identity(model_mat);
2398             mat4 translation_mat;
2399             xm4_translatev(translation_mat,
2400                           json_object_get_number(mesh, "px"),
2401                           json_object_get_number(mesh, "py"),
2402                           json_object_get_number(mesh, "pz"));
2403             mat4 scale_mat;
2404             xm4_scalev(scale_mat,
2405                         json_object_get_number(mesh, "sx"),
2406                         json_object_get_number(mesh, "sy"),
2407                         json_object_get_number(mesh, "sz"));
2408             //TODO: add rotation.
2409             xm4_mul(current->model_mat, translation_mat, scale_mat);
2410         }
2411         free(data);
2412
2413     if(i!=num_meshes-1) //messy but it works

```

```

2414     {
2415         current->next = (struct obj_list_elem*) malloc(sizeof(struct obj_list_elem));
2416         current = current->next;
2417     }
2418     current->next = NULL;
2419 }
2420
2421 //Allocation
2422 out_scene->meshes      = (mesh*) malloc(sizeof(mesh)*out_scene->num_meshes);
2423 out_scene->mesh_verts   = (vec3*) malloc(sizeof(vec3)*out_scene->num_mesh_verts);
2424 out_scene->mesh_nrmls   = (vec3*) malloc(sizeof(vec3)*out_scene->num_mesh_nrmls);
2425 out_scene->mesh_texcoords = (vec2*) malloc(sizeof(vec2)*out_scene->num_mesh_texcoords);
2426 out_scene->mesh_indices  = (ivec3*) malloc(sizeof(ivec3)*out_scene->num_mesh_indices);
2427
2428 assert(out_scene->meshes!=NULL);
2429 assert(out_scene->mesh_verts!=NULL);
2430 assert(out_scene->mesh_nrmls!=NULL);
2431 assert(out_scene->mesh_texcoords!=NULL);
2432 assert(out_scene->mesh_indices!=NULL);
2433
2434 //Parsing and Assignment
2435 int mesh_offset = 0;
2436 int vert_offset = 0;
2437 int nrml_offset = 0;
2438 int texcoord_offset = 0;
2439 int index_offset = 0;
2440
2441 current = first;
2442 while(current != NULL && num_meshes)
2443 {
2444
2445     load_obj(*current, &mesh_offset, &vert_offset, &nrml_offset, &texcoord_offset,
2446             &index_offset, out_scene);
2447
2448     current = current->next;
2449 }
2450 printf("%i and %i\n", vert_offset, out_scene->num_mesh_verts);
2451 assert(mesh_offset==out_scene->num_meshes);
2452 assert(vert_offset==out_scene->num_mesh_verts);
2453 assert(nrml_offset==out_scene->num_mesh_nrmls);
2454 assert(texcoord_offset==out_scene->num_mesh_texcoords);
2455
2456 assert(index_offset==out_scene->num_mesh_indices);
2457
2458 printf("Meshes: %d\nVertices: %d\nIndices: %d\n",
2459        out_scene->num_meshes, out_scene->num_mesh_verts, out_scene->num_mesh_indices);
2460
2461 }
2462
2463 out_scene->materials_changed = true;
2464 out_scene->spheres_changed = true;
2465 out_scene->planes_changed = true;
2466 out_scene->meshes_changed = true;
2467
2468
2469 printf("Finshed scene loading.\n\n");
2470
2471 json_value_free(root_value);
2472 return out_scene;
2473 }
2474 }
2475
2476
2477 scene* load_scene_json_url(char* url)
2478 {
2479     long variable_doesnt_matter;
2480
2481     return load_scene_json( load_file(url, &variable_doesnt_matter) ); //TODO: put data
2482 }
2483 #include <os_abs.h>
2484
2485 void os_start(os_abs abs)
2486 {
2487     (*abs.start_func)();
2488 }
2489
2490 void os_loop_start(os_abs abs)
2491 {
2492     (*abs.loop_start_func)();
2493 }
2494
2495 void os_update(os_abs abs)
2496 {
2497     (*abs.update_func)();
2498 }
2499
2500 void os_sleep(os_abs abs, int num)
2501 {
2502     (*abs.sleep_func)(num);
2503 }
2504
2505 void* os_get_bitmap_memory(os_abs abs)
2506 {
2507     return (*abs.get_bitmap_memory_func)();
2508 }
2509
2510 void os_draw_weird(os_abs abs)
2511 {
2512     (*abs.draw_weird)();
2513 }
2514
2515 int os_get_time_mili(os_abs abs)
2516 {
2517     return (*abs.get_time_mili_func)();
2518 }

```

```

2519
2520 int os_get_width(os_abs abs)
2521 {
2522     return (*abs.get_width_func)();
2523 }
2524
2525 int os_get_height(os_abs abs)
2526 {
2527     return (*abs.get_height_func)();
2528 }
2529
2530 void os_start_thread(os_abs abs, void (*func)(void*, void* data)
2531 {
2532     (*abs.start_thread_func)(func, data);
2533 }
2534 #include <CL/opencl.h>
2535 #include <raytracer.h>
2536 //Parallel util.
2537
2538 void cl_info()
2539 {
2540
2541     int i, j;
2542     char* value;
2543     size_t valueSize;
2544     cl_uint platformCount;
2545     cl_platform_id* platforms;
2546     cl_uint deviceCount;
2547     cl_device_id* devices;
2548     cl_uint maxComputeUnits;
2549     cl_uint recommendedWorkgroupSize = 0;
2550
2551     // get all platforms
2552     clGetPlatformIDs(0, NULL, &platformCount);
2553     platforms = (cl_platform_id*) malloc(sizeof(cl_platform_id) * platformCount);
2554     clGetPlatformIDs(platformCount, platforms, NULL);
2555
2556     for (i = 0; i < platformCount; i++) {
2557
2558         // get all devices
2559         clGetDeviceIDs(platforms[i], CL_DEVICE_TYPE_ALL, 0, NULL, &deviceCount);
2560         devices = (cl_device_id*) malloc(sizeof(cl_device_id) * deviceCount);
2561         clGetDeviceIDs(platforms[i], CL_DEVICE_TYPE_ALL, deviceCount, devices, NULL);
2562
2563         // for each device print critical attributes
2564         for (j = 0; j < deviceCount; j++) {
2565
2566             // print device name
2567             clGetDeviceInfo(devices[j], CL_DEVICE_NAME, 0, NULL, &valueSize);
2568             value = (char*) malloc(valueSize);
2569             clGetDeviceInfo(devices[j], CL_DEVICE_NAME, valueSize, value, NULL);
2570             printf("%i.%d. Device: %s\n", i, j+1, value);
2571             free(value);
2572
2573             // print hardware device version
2574             clGetDeviceInfo(devices[j], CL_DEVICE_VERSION, 0, NULL, &valueSize);
2575             value = (char*) malloc(valueSize);
2576             clGetDeviceInfo(devices[j], CL_DEVICE_VERSION, valueSize, value, NULL);
2577             printf(" %i.%d.%d Hardware version: %s\n", i, j+1, 1, value);
2578             free(value);
2579
2580             // print software driver version
2581             clGetDeviceInfo(devices[j], CL_DRIVER_VERSION, 0, NULL, &valueSize);
2582             value = (char*) malloc(valueSize);
2583             clGetDeviceInfo(devices[j], CL_DRIVER_VERSION, valueSize, value, NULL);
2584             printf(" %.1%.2%.3d Software version: %s\n", i, j+1, 2, value);
2585             free(value);
2586
2587             // print c version supported by compiler for device
2588             clGetDeviceInfo(devices[j], CL_DEVICE_OPENCL_C_VERSION, 0, NULL, &valueSize);
2589             value = (char*) malloc(valueSize);
2590             clGetDeviceInfo(devices[j], CL_DEVICE_OPENCL_C_VERSION, valueSize, value, NULL);
2591             printf(" %.1%.2%.3d OpenCL C version: %s\n", i, j+1, 3, value);
2592             free(value);
2593
2594             // print parallel compute units
2595             clGetDeviceInfo(devices[j], CL_DEVICE_MAX_COMPUTE_UNITS,
2596                             sizeof(maxComputeUnits), &maxComputeUnits, NULL);
2596             printf(" %.1%.2%.3d Parallel compute units: %d\n", i, j+1, 4, maxComputeUnits);
2597
2598             size_t max_work_group_size;
2599             clGetDeviceInfo(devices[j], CL_DEVICE_MAX_WORK_GROUP_SIZE,
2600                           sizeof(max_work_group_size), &max_work_group_size, NULL); //NOTE: just reuse var
2601             printf(" %.1%.2%.3d Max work group size: %zu\n", i, j+1, 4, max_work_group_size);
2602
2603             //clGetDeviceInfo(devices[j], CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE,
2604             //sizeof(recommendedWorkgroupSize), &recommendedWorkgroupSize, NULL);
2605             //printf(" %.1%.2%.3d Recommended work group size: %d\n", i, j+1, 4, recommendedWorkgroupSize);
2606
2607     }
2608
2609     free(devices);
2610
2611 }
2612
2613 printf("\n");
2614 free(platforms);
2615 return;
2616 }
2617 void pfn_notify (
2618 const char *errinfo,
2619 const void *private_info,
2620 size_t cb,
2621 void *user_data)
2622 {
2623     fprintf(stderr, "\n--\nOpenCL ERROR: %s\n--\n", errinfo);
2624     fflush(stderr);

```

```

2624 }
2625 void create_context(rcl_ctx* ctx)
2626 {
2627     int err = CL_SUCCESS;
2628
2629     unsigned int num_of_platforms;
2630
2631     if (clGetPlatformIDs(0, NULL, &num_of_platforms) != CL_SUCCESS)
2632     {
2633         printf("Error: Unable to get platform_id\n");
2634         exit(1);
2635     }
2636     cl_platform_id *platform_ids = malloc(num_of_platforms*sizeof(cl_platform_id));
2637     if (clGetPlatformIDs(num_of_platforms, platform_ids, NULL) != CL_SUCCESS)
2638     {
2639         printf("Error: Unable to get platform_id\n");
2640         exit(1);
2641     }
2642     bool found = false;
2643     for(int i=0; i<num_of_platforms; i++)
2644     {
2645         cl_device_id device_ids[8];
2646         unsigned int num_devices = 0;
2647
2648         //arbitrarily choosing 8 as the max gpus on a platform. TODO: ADD ERROR IF NUM DEVICES EXCEEDS 8
2649         if(clGetDeviceIDs(platform_ids[i], CL_DEVICE_TYPE_GPU, 8, device_ids, &num_devices) == CL_SUCCESS)
2650         {
2651             for(int j = 0; j < num_devices; j++)
2652             {
2653                 char* value;
2654                 size_t valueSize;
2655                 clDeviceInfo(device_ids[j], CL_DEVICE_NAME, 0, NULL, &valueSize);
2656                 value = (char*) malloc(valueSize);
2657                 clGetDeviceInfo(device_ids[j], CL_DEVICE_NAME, valueSize, value, NULL);
2658                 if(value[0]=='H'&&value[1]=='D') //janky but whatever
2659                 {
2660                     printf("WARNING: Skipping over '%s' during device selection\n", value);
2661                     free(value);
2662                     continue;
2663                 }
2664                 free(value);
2665
2666                 found = true;
2667                 ctx->platform_id = platform_ids[i];
2668                 ctx->device_id = device_ids[j];
2669                 break;
2670             }
2671         }
2672     }
2673     if(found)
2674         break;
2675 }
2676 if(!found){
2677     printf("Error: Unable to get a GPU device_id\n");
2678     exit(1);
2679 }
2680
2681 // Create a compute context
2682 //
2683 ctx->context = clCreateContext(0, 1, &ctx->device_id, &pfn_notify, NULL, &err);
2684 if (!ctx->context)
2685 {
2686     printf("Error: Failed to create a compute context!\n");
2687     exit(1);
2688 }
2689
2690 // Create a command commands
2691 //
2692 ctx->commands = clCreateCommandQueue(ctx->context, ctx->device_id, 0, &err);
2693 if (!ctx->commands)
2694 {
2695     printf("Error: Failed to create a command commands!\n");
2696     return;
2697 }
2698 ASRT_CL("Failed to Initialise OpenCL");
2699
2700 { // num compute cores
2701     unsigned int id;
2702     clGetDeviceInfo(ctx->device_id, CL_DEVICE_VENDOR_ID, sizeof(unsigned int), &id, NULL);
2703     switch(id)
2704     {
2705         case(0x10DE): //NVIDIA
2706         {
2707             unsigned int warp_size;
2708             unsigned int compute_capability;
2709             unsigned int num_sm;
2710             unsigned int warps_per_sm;
2711             clGetDeviceInfo(ctx->device_id, CL_DEVICE_WARP_SIZE_NV, //warp size
2712                             sizeof(unsigned int), &warp_size, NULL);
2713             clGetDeviceInfo(ctx->device_id, CL_DEVICE_COMPUTE_CAPABILITY_MAJOR_NV, //compute capability
2714                             sizeof(unsigned int), &compute_capability, NULL);
2715             clGetDeviceInfo(ctx->device_id, CL_DEVICE_MAX_COMPUTE_UNITS, //number of stream multiprocessors
2716                             sizeof(unsigned int), &num_sm, NULL);
2717
2718             switch(compute_capability)
2719             { //nvidia skipped 4 btw
2720                 case 2: warps_per_sm = 1; break; //FERMI (GK104/GK110)
2721                 case 3: warps_per_sm = 6; break; //KEPLER (GK104/GK110) NOTE: ONLY 4 WARP SCHEDULERS THOUGH!
2722                 case 5: warps_per_sm = 4; break; //Maxwell
2723                 case 6: warps_per_sm = 4; break; //Pascal is confusing because the sms vary a lot. GP100 is 2, but GP104 and GP106 have 4
2724                 case 7: warps_per_sm = 2; break; //Volta/Turing Might not be correct(NOTE: 16 FP32 PER CORE? what about warps?)
2725             }
2726             printf("NVIDIA INFO: SM: %d, WARP SIZE: %d, COMPUTE CAPABILITY: %d, WARPS PER SM: %d, TOTAL STREAM PROCESSORS: %d\n\n",
2727

```

```

2729         num_sm, warp_size, compute_capability, warps_per_sm, warps_per_sm*warp_size*num_sm);
2730     ctx->simt_size = warp_size;
2731     ctx->num_simt_per_multiprocessor = warps_per_sm;
2732     ctx->num_multiprocessors = num_sm;
2733     ctx->num_cores = warps_per_sm*warp_size*num_sm;
2734     break;
2735 }
2736 case(0x1002): //AMD
2737 {
2738     printf("AMD GPU INFO NOT SUPPORTED YET!\n");
2739     break;
2740 }
2741 case(0x8086): //INTEL
2742 {
2743     printf("INTEL INFO NOT SUPPORTED YET!\n");
2744     break;
2745 }
2746 }
2747
2748 }
2749
2750 }
2751
2752 cl_mem gen_rgb_image(raytracer_context* rctx,
2753                       const unsigned int width,
2754                       const unsigned int height)
2755 {
2756     cl_image_desc cl_standard_descriptor;
2757     cl_image_format cl_standard_format;
2758     cl_standard_format.image_channel_order = CL_RGBA;
2759     cl_standard_format.image_channel_data_type = CL_FLOAT;
2760
2761     cl_standard_descriptor.image_type = CL_MEM_OBJECT_IMAGE2D;
2762     cl_standard_descriptor.image_width = width==0 ? rctx->width : width;
2763     cl_standard_descriptor.image_height = height==0 ? rctx->height : height;
2764     cl_standard_descriptor.image_depth = 0;
2765     cl_standard_descriptor.image_array_size = 0;
2766     cl_standard_descriptor.image_row_pitch = 0;
2767     cl_standard_descriptor.num_mip_levels = 0;
2768     cl_standard_descriptor.num_samples = 0;
2769     cl_standard_descriptor.buffer = NULL;
2770
2771     int err;
2772
2773     cl_mem img = clCreateImage(rctx->rcl->context,
2774                               CL_MEM_READ_WRITE,
2775                               &cl_standard_format,
2776                               &cl_standard_descriptor,
2777                               NULL,
2778                               &err);
2779     ASRT_CL("Couldn't Create OpenCL Texture");
2780     return img;
2781 }
2782
2783 rcl_img_buf gen_1d_image_buffer(raytracer_context* rctx, size_t t, void* ptr)
2784 {
2785     int err = CL_SUCCESS;
2786
2787     rcl_img_buf ib;
2788     ib.size = t;
2789
2790     ib.buffer = clCreateBuffer(rctx->rcl->context,
2791                               CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
2792                               t,
2793                               ptr,
2794                               &err);
2795     ASRT_CL("Error Creating OpenCL ImageBuffer Buffer");
2796
2797     cl_image_desc cl_standard_descriptor;
2798     cl_image_format cl_standard_format;
2799     cl_standard_format.image_channel_order = CL_RGBA;
2800     cl_standard_format.image_channel_data_type = CL_FLOAT; //prob should be float
2801
2802     cl_standard_descriptor.image_type = CL_MEM_OBJECT_IMAGE1D_BUFFER;
2803     cl_standard_descriptor.image_width = t/4 == 0 ? 1 : t/sizeof(float)/4;
2804     cl_standard_descriptor.image_height = 0;
2805     cl_standard_descriptor.image_depth = 0;
2806     cl_standard_descriptor.image_array_size = 0;
2807     cl_standard_descriptor.image_row_pitch = 0;
2808     cl_standard_descriptor.image_slice_pitch = 0;
2809     cl_standard_descriptor.num_mip_levels = 0;
2810     cl_standard_descriptor.num_samples = 0;
2811     cl_standard_descriptor.buffer = ib.buffer;
2812
2813     ib.image = clCreateImage(rctx->rcl->context,
2814                             CL_MEM_READ_WRITE,
2815                             &cl_standard_format,
2816                             &cl_standard_descriptor,
2817                             NULL, //ptr,
2818                             &err);
2819     ASRT_CL("Error Creating OpenCL ImageBuffer Image");
2820
2821     return ib;
2822 }
2823
2824
2825 }
2826 cl_mem gen_1d_image(raytracer_context* rctx, size_t t, void* ptr)
2827 {
2828     cl_image_desc cl_standard_descriptor;
2829     cl_image_format cl_standard_format;
2830     cl_standard_format.image_channel_order = CL_RGBA;
2831     cl_standard_format.image_channel_data_type = CL_FLOAT; //prob should be float
2832
2833 }
```



```

2939 ASRT_CL("Failed to Retrieve Kernel Work Group Info");
2940     return local;
2941 }
2942
2943
2944 void load_program_raw(rcl_ctx* ctx, char* data,
2945                         char** kernels, unsigned int num_kernels,
2946                         rcl_program* program, char** macros, unsigned int num_macros)
2947 {
2948     int err;
2949
2950     char* fin_data = (char*) malloc(strlen(data)+1);
2951     strcpy(fin_data, data);
2952
2953     for(int i = 0; i < num_macros; i++) //TODO: make more efficient, don't copy all kernel code
2954     {
2955         int length = strlen(macros[i]);
2956         char* buf = (char*) malloc(length+strlen(fin_data)+3);
2957         sprintf(buf, "%s\n%s", macros[i], fin_data);
2958         free(fin_data);
2959         fin_data = buf;
2960     }
2961
2962     program->program = clCreateProgramWithSource(ctx->context, 1, (const char **) &fin_data, NULL, &err);
2963     if (!program->program)
2964     {
2965         printf("Error: Failed to create compute program!\n");
2966         exit(1);
2967     }
2968
2969     // Build the program executable
2970     //
2971     err = clBuildProgram(program->program, 0, NULL, NULL, NULL, NULL);
2972     if (err != CL_SUCCESS)
2973     {
2974         size_t len;
2975         char buffer[2048*25];
2976         buffer[0] = '!';
2977         buffer[1] = '\0';
2978
2979
2980         printf("Error: Failed to build program executable!\n");
2981         printf("KERNEL:\n %s\nprogram done\n", fin_data);
2982         int n_err = clGetProgramBuildInfo(program->program, ctx->device_id, CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &len);
2983         if(n_err != CL_SUCCESS)
2984         {
2985             printf("The error had an error, I hate this. err:%i\n", n_err);
2986         }
2987         printf("err code:%i\n %s\n", err, buffer);
2988         exit(1);
2989     }
2990     else
2991     {
2992         size_t len;
2993         char buffer[2048 * 25];
2994         buffer[0] = '!';
2995         buffer[1] = '\0';
2996         int n_err = clGetProgramBuildInfo(program->program, ctx->device_id, CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &len);
2997         if (n_err != CL_SUCCESS)
2998         {
2999             printf("The error had an error, I hate this. err:%i\n", n_err);
3000         }
3001         printf("Build info: %s\n", buffer);
3002     }
3003
3004     program->raw_kernels = malloc(sizeof(cl_kernel)*num_kernels);
3005     for(int i = 0; i < num_kernels; i++)
3006     {
3007         // Create the compute kernel in the program we wish to run
3008         //
3009
3010         program->raw_kernels[i] = clCreateKernel(program->program, kernels[i], &err);
3011         if (!program->raw_kernels[i] || err != CL_SUCCESS)
3012         {
3013             printf("Error: Failed to create compute kernel! %s\n", kernels[i]);
3014             exit(1);
3015         }
3016
3017     }
3018
3019     program->raw_data = fin_data;
3020
3021 }
3022
3023 void load_program_url(rcl_ctx* ctx, char* url,
3024                         char** kernels, unsigned int num_kernels,
3025                         rcl_program* program, char** macros, unsigned int num_macros)
3026 {
3027     char * buffer = 0;
3028     long length;
3029     FILE * f = fopen (url, "rb");
3030
3031     if (f)
3032     {
3033         fseek (f, 0, SEEK_END);
3034         length = ftell (f);
3035         fseek (f, 0, SEEK_SET);
3036         buffer = malloc (length+2);
3037         if (buffer)
3038         {
3039             fread (buffer, 1, length, f);
3040         }
3041         fclose (f);
3042     }
3043     if (buffer)

```

```

3044     {
3045         buffer[length] = '\0';
3046
3047         load_program_raw(ctx, buffer, kernels, num_kernels, program,
3048                         macros, num_macros);
3049     }
3050
3051 }
3052
3053 //NOTE: old
3054 void test_sphere_raytracer(rcl_ctx* ctx, rcl_program* program,
3055     sphere* spheres, int num_spheres,
3056     uint32_t* bitmap, int width, int height)
3057 {
3058     int err;
3059
3060     static cl_mem tex;
3061     static cl_mem s_buf;
3062     static bool init = false; //temporary
3063
3064     if(!init)
3065     {
3066         //New Texture
3067         tex = clCreateBuffer(ctx->context, CL_MEM_WRITE_ONLY,
3068                             width*height*4, NULL, &err);
3069
3070         //Spheres
3071         s_buf = clCreateBuffer(ctx->context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
3072                               sizeof(float)*4*num_spheres, spheres, &err);
3073         if (err != CL_SUCCESS)
3074         {
3075             printf("Error: Failed to create Sphere Buffer! %d\n", err);
3076             return;
3077         }
3078         init = true;
3079     }
3080     else
3081     {
3082         clEnqueueWriteBuffer (    ctx->commands,
3083                             s_buf,
3084                             CL_TRUE,
3085                             0,
3086                             sizeof(float)*4*num_spheres,
3087                             spheres,
3088                             0,
3089                             NULL,
3090                             NULL);
3091     }
3092
3093
3094
3095     cl_kernel kernel = program->raw_kernels[0]; //just use the first one
3096
3097     clSetKernelArg(kernel, 0, sizeof(cl_mem), &tex);
3098     clSetKernelArg(kernel, 1, sizeof(cl_mem), &s_buf);
3099     clSetKernelArg(kernel, 2, sizeof(unsigned int), &width);
3100     clSetKernelArg(kernel, 3, sizeof(unsigned int), &height);
3101
3102
3103     size_t global;
3104     size_t local = 0;
3105
3106     err = clGetKernelWorkGroupInfo(kernel, ctx->device_id, CL_KERNEL_WORK_GROUP_SIZE,
3107                                   sizeof(local), &local, NULL);
3108     if (err != CL_SUCCESS)
3109     {
3110         printf("Error: Failed to retrieve kernel work group info! %d\n", err);
3111         return;
3112     }
3113
3114     // Execute the kernel over the entire range of our 1d input data set
3115     // using the maximum number of work group items for this device
3116     //
3117     //printf("STARTING\n");
3118     global = width*height;
3119     err = clEnqueueNDRangeKernel(ctx->commands, kernel, 1, NULL, &global, NULL, 0, NULL, NULL);
3120     if (err)
3121     {
3122         printf("Error: Failed to execute kernel! %i\n",err);
3123         return;
3124     }
3125
3126
3127     clFinish(ctx->commands);
3128     //printf("STOPPING\n");
3129
3130     err = clEnqueueReadBuffer(ctx->commands, tex, CL_TRUE, 0, width*height*4, bitmap, 0, NULL, NULL );
3131     if (err != CL_SUCCESS)
3132     {
3133         printf("Error: Failed to read output array! %d\n", err);
3134         exit(1);
3135     }
3136 }
3137 #include <path_raytracer.h>
3138
3139 path_raytracer_context* init_path_raytracer_context(struct _rt_ctx* rctx)
3140 {
3141     path_raytracer_context* prctx = (path_raytracer_context*) malloc(sizeof(path_raytracer_context));
3142     prctx->rctx = rctx;
3143     prctx->up_to_date = false;
3144     prctx->num_samples = 128;//arbitrary default
3145     int err;
3146     printf("Generating Pathtracer Buffers...\n");
3147     prctx->cl_path_fresh_frame_buffer = clCreateBuffer(rctx->rcl->context, CL_MEM_READ_WRITE,
3148                                                       rctx->width*rctx->height*sizeof(vec4),

```

```

3149             NULL, &err);
3150     ASRT_CL("Error Creating OpenCL Fresh Frame Buffer.");
3151     prctx->cl_path_output_buffer = clCreateBuffer(rctx->rcl->context,
3152                                                 CL_MEM_READ_WRITE,
3153                                                 rctx->width*rctx->height*sizeof(vec4),
3154                                                 NULL, &err);
3155     ASRT_CL("Error Creating OpenCL Path Tracer Output Buffer.");
3156
3157     printf("Generated Pathtracer Buffers...\n");
3158     return prctx;
3159 }
3160
3161 //NOTE: the more divisions the slower.
3162 #define WATCHDOG_DIVISIONS_X 2 //TODO: REMOVE THE WATCHDOG DIVISION SYSTEM
3163 #define WATCHDOG_DIVISIONS_Y 2
3164 void path_raytracer_path_trace(path_raytracer_context* prctx)
3165 {
3166     int err;
3167
3168     const unsigned x_div = prctx->rctx->width/WATCHDOG_DIVISIONS_X;
3169     const unsigned y_div = prctx->rctx->height/WATCHDOG_DIVISIONS_Y;
3170
3171     //scene_resource_push(rctx); //Update Scene buffers if necessary.
3172
3173     cl_kernel kernel = prctx->rctx->program->raw_kernels[PATH_TRACE_KRNL_INDX]; //just use the first one
3174
3175     float zeroed[] = {0., 0., 0., 1.};
3176     float* result = matvec_mul(prctx->rctx->stat_scene->camera_world_matrix, zeroed);
3177
3178     clSetKernelArg(kernel, 0, sizeof(cl_mem), &prctx->cl_path_fresh_frame_buffer);
3179     clSetKernelArg(kernel, 1, sizeof(cl_mem), &prctx->rctx->cl_ray_buffer);
3180     clSetKernelArg(kernel, 2, sizeof(cl_mem), &prctx->rctx->stat_scene->cl_material_buffer);
3181     clSetKernelArg(kernel, 3, sizeof(cl_mem), &prctx->rctx->stat_scene->cl_sphere_buffer);
3182     clSetKernelArg(kernel, 4, sizeof(cl_mem), &prctx->rctx->stat_scene->cl_plane_buffer);
3183     clSetKernelArg(kernel, 5, sizeof(cl_mem), &prctx->rctx->stat_scene->cl_mesh_buffer);
3184     clSetKernelArg(kernel, 6, sizeof(cl_mem), &prctx->rctx->stat_scene->cl_mesh_index_buffer.image);
3185     clSetKernelArg(kernel, 7, sizeof(cl_mem), &prctx->rctx->stat_scene->cl_mesh_vert_buffer.image);
3186     clSetKernelArg(kernel, 8, sizeof(cl_mem), &prctx->rctx->stat_scene->cl_mesh_norml_buffer.image);
3187
3188     clSetKernelArg(kernel, 9, sizeof(int), &prctx->rctx->width);
3189     clSetKernelArg(kernel, 10, sizeof(vec4), result);
3190     clSetKernelArg(kernel, 11, sizeof(int), &prctx->current_sample); //NOTE: I don't think this is used
3191
3192     size_t global[2] = {x_div, y_div};
3193
3194     //NOTE: tripping watchdog timer
3195     if(global[0]*WATCHDOG_DIVISIONS_X*global[1]*WATCHDOG_DIVISIONS_Y!=
3196        prctx->rctx->width*prctx->rctx->height)
3197     {
3198         printf("Watchdog divisions are incorrect!\n");
3199         exit(1);
3200     }
3201
3202     size_t offset[2];
3203
3204     for(int x = 0; x < WATCHDOG_DIVISIONS_X; x++)
3205     {
3206         for(int y = 0; y < WATCHDOG_DIVISIONS_Y; y++)
3207         {
3208             offset[0] = x_div*x;
3209             offset[1] = y_div*y;
3210             err = clEnqueueNDRangeKernel(prctx->rctx->rcl->commands, kernel, 2,
3211                                         offset, global, NULL, 0, NULL, NULL);
3212             ASRT_CL("Failed to execute path trace kernel");
3213         }
3214     }
3215
3216     err = clFinish(prctx->rctx->rcl->commands);
3217     ASRT_CL("Something happened while executing path trace kernel");
3218 }
3219
3220
3221 void path_raytracer_average_buffers(path_raytracer_context* prctx)
3222 {
3223     int err;
3224
3225     cl_kernel kernel = prctx->rctx->program->raw_kernels[F_BUFFER_AVG_KRNL_INDX];
3226     clSetKernelArg(kernel, 0, sizeof(cl_mem), &prctx->cl_path_output_buffer);
3227     clSetKernelArg(kernel, 1, sizeof(cl_mem), &prctx->cl_path_fresh_frame_buffer);
3228     clSetKernelArg(kernel, 2, sizeof(unsigned int), &prctx->rctx->width);
3229     clSetKernelArg(kernel, 3, sizeof(unsigned int), &prctx->rctx->height);
3230     clSetKernelArg(kernel, 4, sizeof(unsigned int), &prctx->num_samples);
3231     clSetKernelArg(kernel, 5, sizeof(unsigned int), &prctx->current_sample);
3232
3233     size_t global;
3234     size_t local = get_workgroup_size(prctx->rctx, kernel);
3235
3236     // Execute the kernel over the entire range of our 1d input data set
3237     // using the maximum number of work group items for this device
3238     //
3239     global = prctx->rctx->width*prctx->rctx->height;
3240     err = clEnqueueNDRangeKernel(prctx->rctx->rcl->commands, kernel, 1, NULL,
3241                                 &global, NULL, 0, NULL, NULL);
3242     ASRT_CL("Failed to execute kernel");
3243     err = clFinish(prctx->rctx->rcl->commands);
3244     ASRT_CL("Something happened while waiting for kernel to finish");
3245 }
3246
3247 void path_raytracer_push_path(path_raytracer_context* prctx)
3248 {
3249     int err;
3250
3251     cl_kernel kernel = prctx->rctx->program->raw_kernels[F_BUF_TO_BYTE_BUF_KRNL_INDX];
3252     clSetKernelArg(kernel, 0, sizeof(cl_mem), &prctx->rctx->cl_output_buffer);
3253     clSetKernelArg(kernel, 1, sizeof(cl_mem), &prctx->cl_path_output_buffer);

```

```

3254 clSetKernelArg(kernel, 2, sizeof(unsigned int), &prctx->rctx->width);
3255 clSetKernelArg(kernel, 3, sizeof(unsigned int), &prctx->rctx->height);
3256
3257
3258 size_t global;
3259 size_t local = get_workgroup_size(prctx->rctx, kernel);
3260
3261 // Execute the kernel over the entire range of our 1d input data set
3262 // using the maximum number of work group items for this device
3263 //
3264 global = prctx->rctx->width*prctx->rctx->height;
3265 err = clEnqueueNDRangeKernel(prctx->rctx->rcl->commands, kernel, 1,
3266                             NULL, &global, NULL, 0, NULL, NULL);
3267 ASRT_CL("Failed to execute kernel");
3268
3269 err = clFinish(prctx->rctx->rcl->commands);
3270 ASRT_CL("Something happened while waiting for kernel to finish");
3271
3272
3273 err = clEnqueueReadBuffer(prctx->rctx->rcl->commands, prctx->rctx->cl_output_buffer, CL_TRUE, 0,
3274                         prctx->rctx->width*prctx->rctx->height*sizeof(int),
3275                         prctx->rctx->output_buffer,
3276                         0, NULL, NULL );
3277 ASRT_CL("Failed to read output array");
3278 //printf("RENDER\n");
3279
3280 }
3281
3282
3283
3284 void path_raytracer_render(path_raytracer_context* prctx)
3285 {
3286     int local_start_time = os_get_time_mili(abst);
3287     prctx->current_sample++;
3288     if(prctx->current_sample>prctx->num_samples)
3289     {
3290         prctx->render_complete = true;
3291         printf("Render took %d ms\n", os_get_time_mili(abst)-prctx->start_time);
3292         return;
3293     }
3294     _raytracer_gen_ray_buffer(prctx->rctx);
3295
3296     path_raytracer_path_trace(prctx);
3297
3298     if(prctx->current_sample == 1) //needs to be here
3299     {
3300         int err;
3301         err = clEnqueueCopyBuffer ( prctx->rctx->rcl->commands,
3302                                     prctx->cl_path_fresh_frame_buffer,
3303                                     prctx->cl_path_output_buffer,
3304                                     0,
3305                                     prctx->rctx->width*prctx->rctx->height*sizeof(vec4),
3306                                     0,
3307                                     0,
3308                                     0,
3309                                     NULL);
3310         ASRT_CL("Error copying OpenCL Output Buffer");
3311
3312         err = clFinish(prctx->rctx->rcl->commands);
3313         ASRT_CL("Something happened while waiting for copy to finish");
3314     }
3315     path_raytracer_average_buffers(prctx);
3316     path_raytracer_push_path(prctx);
3317     printf("Total time for sample group: %d\n", os_get_time_mili(abst)-local_start_time);
3318 }
3319
3320 void path_raytracer_prepass(path_raytracer_context* prctx)
3321 {
3322     raytracer_prepass(prctx->rctx); //Nothing Special
3323     prctx->current_sample = 0;
3324     prctx->start_time = os_get_time_mili(abst);
3325 }
3326 #include <raytracer.h>
3327 #include <parallel.h>
3328 //binary resources
3329 #include <test.cl.h> //test kernel
3330
3331
3332
3333 //NOTE: we are assuming the output buffer will be the right size
3334 raytracer_context* raytracer_init(unsigned int width, unsigned int height,
3335                                     uint32_t* output_buffer, rcl_ctx* rcl)
3336 {
3337     raytracer_context* rctx = (raytracer_context*) malloc(sizeof(raytracer_context));
3338     rctx->width = width;
3339     rctx->height = height;
3340     rctx->ray_buffer = (float*) malloc(width * height * sizeof(float));
3341     rctx->output_buffer = output_buffer;
3342     //rctx->fresh_buffer = (uint32_t*) malloc(width * height * sizeof(uint32_t));
3343     rctx->rcl = rcl;
3344     rctx->program = (rcl_program*) malloc(sizeof(rcl_program));
3345     rctx->ic_ctx = (ic_context*) malloc(sizeof(ic_context));
3346     //ic_init(rctx);
3347     rctx->render_complete = false;
3348     rctx->num_samples = 64; //NOTE: arbitrary default
3349     rctx->current_sample = 0;
3350     rctx->event_position = 0;
3351     rctx->block_size_y = 0;
3352     rctx->block_size_x = 0;
3353     return rctx;
3354 }
3355
3356 void raytracer_build_kernels(raytracer_context* rctx)
3357 {
3358     printf("Building Kernels...\n");

```

```

3359 char* kernels[] = KERNELS;
3360 printf("Generating Kernel Macros...\n");
3361 //Macros
3362 unsigned int num_macros = 0;
3363 #ifdef _WIN32
3364 char os_macro[] = "#define _WIN32 1";
3365 #else
3366 char os_macro[] = "#define _OSX 1";
3367 #endif
3368 num_macros++;
3369
3370 MACRO_GEN(sphere_macro, SCENE_NUM_SPHERES %i, rctx->stat_scene->num_spheres, num_macros);
3371 MACRO_GEN(plane_macro, SCENE_NUM_PLANES %i, rctx->stat_scene->num_planes, num_macros);
3372 MACRO_GEN(index_macro, SCENE_NUM_INDICES %i, rctx->stat_scene->num_mesh_indices, num_macros);
3373 MACRO_GEN(mesh_macro, SCENE_NUM_MESHES %i, rctx->stat_scene->num_meshes, num_macros);
3374 MACRO_GEN(material_macro, SCENE_NUM_MATERIALS %i, rctx->stat_scene->num_materials, num_macros);
3375 MACRO_GEN(blockx_macro, BLOCKSIZE_X %i, rctx->rcl->sint_size, num_macros);
3376 MACRO_GEN(blocky_macro, BLOCKSIZE_Y %i, rctx->rcl->num_sint_per_multiprocessor, num_macros);
3377
3378 char min_macro[64];
3379 sprintf(min_macro, "#define SCENE_MIN (%f, %f, %f)",
3380     rctx->stat_scene->kdt->bounds.min[0],
3381     rctx->stat_scene->kdt->bounds.min[1],
3382     rctx->stat_scene->kdt->bounds.min[2]);
3383 num_macros++;
3384 char max_macro[64];
3385 sprintf(max_macro, "#define SCENE_MAX (%f, %f, %f)",
3386     rctx->stat_scene->kdt->bounds.max[0],
3387     rctx->stat_scene->kdt->bounds.max[1],
3388     rctx->stat_scene->kdt->bounds.max[2]);
3389 num_macros++;
3390
3391 //TODO: do something better than this
3392 char* macros[] = {sphere_macro, plane_macro, mesh_macro, index_macro,
3393     material_macro, os_macro, blockx_macro, blocky_macro,
3394     min_macro, max_macro};
3395 printf("Macros Generated.\n");
3396
3397 load_program_raw(rctx->rcl,
3398     all_kernels_cl, //NOTE: Binary resource
3399     kernels, NUM_KERNELS, rctx->program,
3400     macros, num_macros);
3401 printf("Kernels built.\n");
3402
3403 }
3404 }
3405
3406 void raytracer_build(raytracer_context* rctx)
3407 {
3408     //CL init
3409     printf("Building Scene...\n");
3410
3411     int err = CL_SUCCESS;
3412
3413     printf("Initializing Scene Resources On GPU.\n");
3414     scene_init_resources(rctx);
3415     rctx->stat_scene->kdt->s = rctx->stat_scene;
3416     printf("Initialized Scene Resources On GPU.\n");
3417
3418
3419     printf("Building/Rebuilding k-d tree.\n");
3420     kd_tree_construct(rctx->stat_scene->kdt);
3421     printf("Done Building/Rebuilding k-d tree.\n");
3422
3423
3424
3425     //Kernels
3426     raytracer_build_kernels(rctx);
3427
3428     //Buffers
3429     printf("Generating Buffers...\n");
3430     rctx->cl_ray_buffer = clCreateBuffer(rctx->rcl->context,
3431         CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
3432         rctx->width*rctx->height*sizeof(ray),
3433         rctx->ray_buffer, &err);
3434     ASRT_CL("Error Creating OpenCL Ray Buffer.");
3435     rctx->cl_path_output_buffer = clCreateBuffer(rctx->rcl->context,
3436         CL_MEM_READ_WRITE,
3437         rctx->width*rctx->height*sizeof(vec4),
3438         NULL, &err);
3439     ASRT_CL("Error Creating OpenCL Path Tracer Output Buffer.");
3440
3441     rctx->cl_output_buffer = clCreateBuffer(rctx->rcl->context, CL_MEM_READ_WRITE,
3442         rctx->width*rctx->height*4, NULL, &err);
3443     ASRT_CL("Error Creating OpenCL Output Buffer.");
3444
3445     //TODO: all output buffers and frame buffers should be images.
3446     rctx->cl_path_fresh_frame_buffer = clCreateBuffer(rctx->rcl->context, CL_MEM_READ_WRITE,
3447         rctx->width*rctx->height*sizeof(vec4), NULL, &err);
3448     ASRT_CL("Error Creating OpenCL Fresh Frame Buffer.");
3449
3450     printf("Generated Buffers...\n");
3451 }
3452
3453 void raytracer_prepass(raytracer_context* rctx)
3454 {
3455     printf("Starting Raytracer Prepass.\n");
3456
3457     scene_resource_push(rctx);
3458
3459     printf("Finished Raytracer Prepass.\n");
3460 }
3461
3462 void raytracer_render(raytracer_context* rctx)
3463 {

```

```

3464     _raytracer_gen_ray_buffer(rctx);
3465
3466     _raytracer_cast_rays(rctx);
3467 }
3468
3469 //#define JANK_SAMPLES 32
3470 void raytracer_refined_render(raytracer_context* rctx)
3471 {
3472     rctx->current_sample++;
3473     if(rctx->current_sample>rctx->num_samples)
3474     {
3475         rctx->render_complete = true;
3476         return;
3477     }
3478     _raytracer_gen_ray_buffer(rctx);
3479
3480     _raytracer_path_trace(rctx, rctx->current_sample);
3481
3482     if(rctx->current_sample==1) //really terrible place for path tracer initialization...
3483     {
3484         int err;
3485         char pattern = 0;
3486         err = clEnqueueCopyBuffer (    rctx->rcl->commands,
3487                                     rctx->cl_path_fresh_frame_buffer,
3488                                     rctx->cl_path_output_buffer,
3489                                     0,
3490                                     0,
3491                                     rctx->width*rctx->height*sizeof(vec4),
3492                                     0,
3493                                     0,
3494                                     NULL);
3495         ASRT_CL("Error copying OpenCL Output Buffer");
3496
3497         err = clFinish(rctx->rcl->commands);
3498         ASRT_CL("Something happened while waiting for copy to finish");
3499     }
3500
3501 //Nothings wrong I just am currently refactoring this
3502 //_raytracer_average_buffers(rctx, rctx->current_sample);
3503 _raytracer_push_path(rctx);
3504
3505 }
3506
3507 void _raytracer_gen_ray_buffer(raytracer_context* rctx)
3508 {
3509     int err;
3510
3511     cl_kernel kernel = rctx->program->raw_kernels[RAY_BUFFER_KRNL_IDX];
3512     clSetKernelArg(kernel, 0, sizeof(cl_mem), &rctx->cl_ray_buffer);
3513     clSetKernelArg(kernel, 1, sizeof(unsigned int), &rctx->width);
3514     clSetKernelArg(kernel, 2, sizeof(unsigned int), &rctx->height);
3515     clSetKernelArg(kernel, 3, sizeof(mat4), rctx->stat_scene->camera_world_matrix);
3516
3517
3518     size_t global;
3519
3520
3521     global = rctx->width*rctx->height;
3522     err = clEnqueueNDRangeKernel(rctx->rcl->commands, kernel, 1, NULL, &global, 0, NULL, NULL);
3523     ASRT_CL("Failed to execute kernel");
3524
3525
3526 //Wait for completion
3527     err = clFinish(rctx->rcl->commands);
3528     ASRT_CL("Something happened while waiting for kernel raybuf to finish");
3529
3530
3531 }
3532
3533
3534 void _raytracer_push_path(raytracer_context* rctx)
3535 {
3536     int err;
3537
3538     cl_kernel kernel = rctx->program->raw_kernels[F_BUF_TO_BYTE_BUF_KRNL_IDX];
3539     clSetKernelArg(kernel, 0, sizeof(cl_mem), &rctx->cl_output_buffer);
3540     clSetKernelArg(kernel, 1, sizeof(cl_mem), &rctx->cl_path_output_buffer);
3541     clSetKernelArg(kernel, 2, sizeof(unsigned int), &rctx->width);
3542     clSetKernelArg(kernel, 3, sizeof(unsigned int), &rctx->height);
3543
3544
3545     size_t global;
3546     size_t local = get_workgroup_size(rctx, kernel);
3547
3548 // Execute the kernel over the entire range of our 1d input data set
3549 // using the maximum number of work group items for this device
3550 //
3551     global = rctx->width*rctx->height;
3552     err = clEnqueueNDRangeKernel(rctx->rcl->commands, kernel, 1, NULL, &global, 0, NULL, NULL);
3553     ASRT_CL("Failed to execute kernel");
3554
3555
3556     err = clFinish(rctx->rcl->commands);
3557     ASRT_CL("Something happened while waiting for kernel to finish");
3558
3559
3560     err = clEnqueueReadBuffer(rctx->rcl->commands, rctx->cl_output_buffer, CL_TRUE, 0,
3561                             rctx->width*rctx->height*sizeof(int), rctx->output_buffer,
3562                             0, NULL, NULL );
3563     ASRT_CL("Failed to read output array");
3564
3565 }
3566
3567 //NOTE: the more divisions the slower.
3568 #define WATCHDOG_DIVISIONS_X 2

```

```

3569 #define WATCHDOG_DIVISIONS_Y 2
3570 void _raytracer_path_trace(raytracer_context* rctx, unsigned int sample_num)
3571 {
3572     int err;
3573
3574     const unsigned x_div = rctx->width/WATCHDOG_DIVISIONS_X;
3575     const unsigned y_div = rctx->height/WATCHDOG_DIVISIONS_Y;
3576
3577     //scene_resource_push(rctx); //Update Scene buffers if necessary.
3578
3579     cl_kernel kernel = rctx->program->raw_kernels[PATH_TRACE_KRNL_INDX]; //just use the first one
3580
3581     float zeroed[] = {0., 0., 0., 1.};
3582     float* result = matvec_mul(rctx->stat_scene->camera_world_matrix, zeroed);
3583
3584     clSetKernelArg(kernel, 0, sizeof(cl_mem), &rctx->cl_path_fresh_frame_buffer);
3585     clSetKernelArg(kernel, 1, sizeof(cl_mem), &rctx->cl_ray_buffer);
3586     clSetKernelArg(kernel, 2, sizeof(cl_mem), &rctx->stat_scene->cl_material_buffer);
3587     clSetKernelArg(kernel, 3, sizeof(cl_mem), &rctx->stat_scene->cl_sphere_buffer);
3588     clSetKernelArg(kernel, 4, sizeof(cl_mem), &rctx->stat_scene->cl_plane_buffer);
3589     clSetKernelArg(kernel, 5, sizeof(cl_mem), &rctx->stat_scene->cl_mesh_buffer);
3590     clSetKernelArg(kernel, 6, sizeof(cl_mem), &rctx->stat_scene->cl_mesh_index_buffer.image);
3591     clSetKernelArg(kernel, 7, sizeof(cl_mem), &rctx->stat_scene->cl_mesh_vert_buffer.image);
3592     clSetKernelArg(kernel, 8, sizeof(cl_mem), &rctx->stat_scene->cl_mesh_nrml_buffer.image);
3593
3594     clSetKernelArg(kernel, 9, sizeof(int), &rctx->width);
3595     clSetKernelArg(kernel, 10, sizeof(vec4), result);
3596     clSetKernelArg(kernel, 11, sizeof(int), &sample_num); //NOTE: I don't think this is used
3597
3598     size_t global[2] = {x_div, y_div};
3599
3600     //NOTE: tripping watchdog timer
3601     if(global[0]*WATCHDOG_DIVISIONS_X*global[1]*WATCHDOG_DIVISIONS_Y!=rctx->width*rctx->height)
3602     {
3603         printf("Watchdog divisions are incorrect!\n");
3604         exit(1);
3605     }
3606
3607     size_t offset[2];
3608
3609     for(int x = 0; x < WATCHDOG_DIVISIONS_X; x++)
3610     {
3611         for(int y = 0; y < WATCHDOG_DIVISIONS_Y; y++)
3612         {
3613             offset[0] = x_div*x;
3614             offset[1] = y_div*y;
3615             err = clEnqueueNDRangeKernel(rctx->rcl->commands, kernel, 2,
3616                                         offset, global, NULL, 0, NULL, NULL);
3617             ASRT_CL("Failed to execute path trace kernel");
3618         }
3619     }
3620
3621     err = clFinish(rctx->rcl->commands);
3622     ASRT_CL("Something happened while executing path trace kernel");
3623 }
3624
3625
3626 void _raytracer_cast_rays(raytracer_context* rctx) //TODO: do more path tracing stuff here
3627 {
3628     int err;
3629
3630     scene_resource_push(rctx); //Update Scene buffers if necessary.
3631
3632
3633     cl_kernel kernel = rctx->program->raw_kernels[RAY_CAST_KRNL_INDX]; //just use the first one
3634
3635     float zeroed[] = {0., 0., 0., 1.};
3636     float* result = matvec_mul(rctx->stat_scene->camera_world_matrix, zeroed);
3637     clSetKernelArg(kernel, 0, sizeof(cl_mem), &rctx->cl_output_buffer);
3638     clSetKernelArg(kernel, 1, sizeof(cl_mem), &rctx->cl_ray_buffer);
3639     clSetKernelArg(kernel, 2, sizeof(cl_mem), &rctx->stat_scene->cl_material_buffer);
3640     clSetKernelArg(kernel, 3, sizeof(cl_mem), &rctx->stat_scene->cl_sphere_buffer);
3641     clSetKernelArg(kernel, 4, sizeof(cl_mem), &rctx->stat_scene->cl_plane_buffer);
3642     clSetKernelArg(kernel, 5, sizeof(cl_mem), &rctx->stat_scene->cl_mesh_buffer);
3643     clSetKernelArg(kernel, 6, sizeof(cl_mem), &rctx->stat_scene->cl_mesh_index_buffer.image);
3644     clSetKernelArg(kernel, 7, sizeof(cl_mem), &rctx->stat_scene->cl_mesh_vert_buffer.image);
3645     clSetKernelArg(kernel, 8, sizeof(cl_mem), &rctx->stat_scene->cl_mesh_nrml_buffer.image);
3646
3647     clSetKernelArg(kernel, 9, sizeof(unsigned int), &rctx->width);
3648     clSetKernelArg(kernel, 10, sizeof(unsigned int), &rctx->height);
3649     clSetKernelArg(kernel, 11, sizeof(float)*4, result); //we only need 3
3650     //free(result);
3651
3652     size_t global;
3653
3654     global = rctx->width*rctx->height;
3655     err = clEnqueueNDRangeKernel(rctx->rcl->commands, kernel, 1, NULL, &global, 0, NULL, NULL);
3656     ASRT_CL("Failed to Execute Kernel");
3657
3658     err = clFinish(rctx->rcl->commands);
3659     ASRT_CL("Something happened during kernel execution");
3660
3661     err = clEnqueueReadBuffer(rctx->rcl->commands, rctx->cl_output_buffer, CL_TRUE, 0,
3662                               rctx->width*rctx->height*sizeof(int), rctx->output_buffer, 0, NULL, NULL );
3663     ASRT_CL("Failed to read output array");
3664
3665 }
3666 #include <scene.h>
3667 #include <raytracer.h>
3668 #include <kdtree.h>
3669 #include <geom.h>
3670 #include <CL/cl.h>
3671
3672 void scene_init_resources(raytracer_context* rctx)
3673 {

```

```

3674     int err;
3675
3676     //initialise kd tree
3677     rctx->stat_scene->kdt = kd_tree_init();
3678
3679
3680     //Scene Buffers
3681     rctx->stat_scene->cl_sphere_buffer = clCreateBuffer(rctx->rcl->context,
3682                                         CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
3683                                         sizeof(sphere)*rctx->stat_scene->num_spheres,
3684                                         rctx->stat_scene->spheres, &err);
3685     ASRT_CL("Error Creating OpenCL Scene Sphere Buffer.");
3686
3687     rctx->stat_scene->cl_plane_buffer = clCreateBuffer(rctx->rcl->context,
3688                                         CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
3689                                         sizeof(plane)*rctx->stat_scene->num_planes,
3690                                         rctx->stat_scene->planes, &err);
3691     ASRT_CL("Error Creating OpenCL Scene Plane Buffer.");
3692
3693
3694     rctx->stat_scene->cl_material_buffer = clCreateBuffer(rctx->rcl->context,
3695                                         CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
3696                                         sizeof(material)*
3697                                         rctx->stat_scene->num_materials,
3698                                         rctx->stat_scene->materials, &err);
3699     ASRT_CL("Error Creating OpenCL Scene Plane Buffer.");
3700
3701
3702     //Mesh
3703     rctx->stat_scene->cl_mesh_buffer = clCreateBuffer(rctx->rcl->context,
3704                                         CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
3705                                         rctx->stat_scene->num_meshes==0 ? 1 :
3706                                         sizeof(mesh)*rctx->stat_scene->num_meshes,
3707                                         rctx->stat_scene->meshes, &err);
3708     ASRT_CL("Error Creating OpenCL Scene Mesh Buffer.");
3709
3710     //mesh data is stored as images for faster access
3711     rctx->stat_scene->cl_mesh_vert_buffer =
3712         gen_id_image_buffer(rctx, rctx->stat_scene->num_mesh_verts==0 ? 1 :
3713             sizeof(vec3)*rctx->stat_scene->num_mesh_verts,
3714             rctx->stat_scene->mesh_verts);
3715
3716     rctx->stat_scene->cl_mesh_nrm1_buffer =
3717         gen_id_image_buffer(rctx, rctx->stat_scene->num_mesh_nrm1s==0 ? 1 :
3718             sizeof(vec3)*rctx->stat_scene->num_mesh_nrm1s,
3719             rctx->stat_scene->mesh_nrm1s);
3720
3721     rctx->stat_scene->cl_mesh_index_buffer =
3722         gen_id_image_buffer(rctx, rctx->stat_scene->num_mesh_indices==0 ? 1 :
3723             sizeof(ivec3)*
3724             rctx->stat_scene->num_mesh_indices,//maybe
3725             rctx->stat_scene->mesh_indices);
3726
3727
3728
3729
3730 }
3731
3732
3733 void scene_resource_push(raytracer_context* rctx)
3734 {
3735     int err;
3736
3737     //if(rctx->stat_scene->kdt->cl_kd_tree_buffer != NULL)
3738     //    exit(1);
3739     printf("Pushing Scene Resources...");
3740
3741     printf("Serializing k-d tree...");
3742     kd_tree_generate_serialized(rctx->stat_scene->kdt);
3743
3744     //NOTE: SUPER SCUFFED
3745     if(rctx->stat_scene->kdt->cl_kd_tree_buffer == NULL)
3746     {
3747         rctx->stat_scene->kdt->cl_kd_tree_buffer =
3748             clCreateBuffer(rctx->rcl->context,
3749                         CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
3750                         rctx->stat_scene->kdt->buffer_size,
3751                         rctx->stat_scene->kdt->buffer, &err);
3752         ASRT_CL("Couldn't create kd tree buffer.");
3753     }
3754     printf("Pushing Buffers...");
3755     if(rctx->stat_scene->meshes_changed)
3756     {
3757         clEnqueueWriteBuffer (   rctx->rcl->commands,
3758                                 rctx->stat_scene->cl_mesh_buffer,
3759                                 CL_TRUE,
3760                                 0,
3761                                 sizeof(mesh)*rctx->stat_scene->num_meshes,
3762                                 rctx->stat_scene->meshes,
3763                                 0,
3764                                 NULL,
3765                                 NULL);
3766     }
3767
3768     if(rctx->stat_scene->spheres_changed)
3769     {
3770         clEnqueueWriteBuffer (   rctx->rcl->commands,
3771                                 rctx->stat_scene->cl_sphere_buffer,
3772                                 CL_TRUE,
3773                                 0,
3774                                 sizeof(sphere)*rctx->stat_scene->num_spheres,
3775                                 rctx->stat_scene->spheres,
3776                                 0,
3777                                 NULL,
3778                                 NULL);
3779

```

```

3779 }
3780
3781 if(rctx->stat_scene->planes_changed)
3782 {
3783     clEnqueueWriteBuffer (    rctx->rcl->commands,
3784                             rctx->stat_scene->cl_plane_buffer,
3785                             CL_TRUE,
3786                             0,
3787                             sizeof(plane)*rctx->stat_scene->num_planes,
3788                             rctx->stat_scene->planes,
3789                             0,
3790                             NULL,
3791                             NULL);
3792 }
3793
3794 if(rctx->stat_scene->materials_changed)
3795 {
3796     clEnqueueWriteBuffer (    rctx->rcl->commands,
3797                             rctx->stat_scene->cl_material_buffer,
3798                             CL_TRUE,
3799                             0,
3800                             sizeof(material)*rctx->stat_scene->num_materials,
3801                             rctx->stat_scene->materials,
3802                             0,
3803                             NULL,
3804                             NULL);
3805 }
3806 }
3807
3808 printf("Done.\n");
3809 }
3810 #include <spath_raytracer.h>
3811 #include <kdtree.h>
3812 #include <raytracer.h>
3813 #include <stdlib.h>
3814 //##include <windows.h>
3815 typedef struct W_ALIGN(16) spath_progress
3816 {
3817     unsigned int sample_num;
3818     unsigned int bounce_num;
3819     vec3 mask;
3820     vec3 accum_color;
3821 } U_ALIGN(16) spath_progress; //NOTE: space for two more 32 bit dudes
3822
3823
3824 void bad_buf_update(spath_raytracer_context* sprctx)
3825 {
3826     int err;
3827
3828     unsigned int bad_buf[4*4+1];
3829     bad_buf[4*4] = 0;
3830     {
3831         //good thing this is the same transposed. Also this is stupid, but endorsed by AMD
3832         unsigned int mat[4*4] = {0xffffffff, 0, 0, 0,
3833                               0, 0xffffffff, 0, 0,
3834                               0, 0, 0xffffffff, 0,
3835                               0, 0, 0, 0xffffffff};
3836         memcpy(bad_buf, mat, 4*4*sizeof(unsigned int));
3837     }
3838
3839     err = clEnqueueWriteBuffer(sprctx->rctx->rcl->commands, sprctx->cl_bad_api_design_buffer, CL_TRUE,
3840                               0, (4*4+1)*sizeof(float),bad_buf,
3841                               0, NULL, NULL);
3842     ASRT_CL("Error Creating OpenCL BAD API DESIGN! Buffer.");
3843
3844     err = clFinish(sprctx->rctx->rcl->commands);
3845     ASRT_CL("Something happened while waiting for copy to finish");
3846 }
3847
3848 spath_raytracer_context* init_spath_raytracer_context(struct _rt_ctx* rctx)
3849 {
3850     spath_raytracer_context* sprctx = (spath_raytracer_context*) malloc(sizeof(spath_raytracer_context));
3851     sprctx->rctx = rctx;
3852     sprctx->up_to_date = false;
3853
3854     int err;
3855     printf("Generating Split Pathtracer Buffers...\n");
3856
3857
3858     sprctx->cl_path_output_buffer = clCreateBuffer(rctx->rcl->context,
3859                                                 CL_MEM_READ_WRITE,
3860                                                 rctx->width*rctx->height*sizeof(vec4),
3861                                                 NULL, &err);
3862     ASRT_CL("Error Creating OpenCL Split Path Tracer Output Buffer.");
3863
3864     sprctx->cl_path_ray_origin_buffer = clCreateBuffer(rctx->rcl->context,
3865                                                 CL_MEM_READ_WRITE,
3866                                                 rctx->width*rctx->height*
3867                                                 sizeof(ray),
3868                                                 NULL, &err);
3869     ASRT_CL("Error Creating OpenCL Split Path Tracer Collision Result Buffer.");
3870
3871     sprctx->cl_path_collision_result_buffer = clCreateBuffer(rctx->rcl->context,
3872                                                 CL_MEM_READ_WRITE,
3873                                                 rctx->width*rctx->height*
3874                                                 sizeof(kd_tree_collision_result),
3875                                                 NULL, &err);
3876     ASRT_CL("Error Creating OpenCL Split Path Tracer Collision Result Buffer.");
3877
3878     sprctx->cl_path_origin_collision_result_buffer = clCreateBuffer(rctx->rcl->context,
3879                                                 CL_MEM_READ_WRITE,
3880                                                 rctx->width*rctx->height*
3881                                                 sizeof(kd_tree_collision_result),
3882                                                 NULL, &err);
3883     ASRT_CL("Error Creating OpenCL Split Path Tracer ORIGIN Collision Result Buffer.");

```

```

3884 sprctx->cl_random_buffer = clCreateBuffer(rctx->rcl->context,
3885                                     CL_MEM_READ_WRITE,
3886                                     rctx->width * rctx->height * sizeof(unsigned int),
3887                                     NULL, &err);
3888 ASRT_CL("Error Creating OpenCL Random Buffer.");
3889
3890 sprctx->random_buffer = (unsigned int*) malloc(rctx->width * rctx->height * sizeof(unsigned int));
3891
3892 sprctx->cl_spath_progress_buffer = clCreateBuffer(rctx->rcl->context,
3893                                     CL_MEM_READ_WRITE,
3894                                     rctx->width * rctx->height *
3895                                     sizeof(spath_progress),
3896                                     NULL, &err);
3897 zero_buffer(rctx, sprctx->cl_spath_progress_buffer, rctx->width * rctx->height * sizeof(spath_progress));
3898
3899 ASRT_CL("Error Creating OpenCL Split Path Tracer Collision Result Buffer.");
3900 {
3901     unsigned int bad_buf[4*4+1];
3902     bad_buf[4*4] = 0;
3903     {
3904         //good thing this is the same transposed. Also this is stupid, but endorsed by AMD
3905         unsigned int mat[4*4] = {0xffffffff, 0, 0, 0,
3906                                 0, 0xffffffff, 0, 0,
3907                                 0, 0, 0xffffffff, 0,
3908                                 0, 0, 0, 0xffffffff};
3909         memcpy(bad_buf, mat, 4*4*sizeof(unsigned int));
3910     }
3911
3912     sprctx->cl_bad_api_design_buffer = clCreateBuffer(rctx->rcl->context,
3913                                     CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
3914                                     (4*4+1)*sizeof(float),
3915                                     bad_buf, &err);
3916     ASRT_CL("Error Creating OpenCL BAD API DESIGN! Buffer.");
3917
3918     err = clFinish(rctx->rcl->commands);
3919     ASRT_CL("Something happened while waiting for copy to finish");
3920 }
3921 printf("Generated Split Pathtracer Buffers.\n");
3922 return sprctx;
3923
3924 }
3925
3926 void spath_raytracer_update_random(spath_raytracer_context* sprctx)
3927 {
3928     for(int i = 0; i < sprctx->rctx->width*sprctx->rctx->height; i++)
3929         sprctx->random_buffer[i] = rand();
3930
3931     int err;
3932
3933     err = clEnqueueWriteBuffer (sprctx->rctx->rcl->commands,
3934                             sprctx->cl_random_buffer,
3935                             CL_TRUE, 0,
3936                             sprctx->rctx->width * sprctx->rctx->height * sizeof(unsigned int),
3937                             sprctx->random_buffer,
3938                             0, NULL, NULL);
3939     ASRT_CL("Couldn't Push Random Buffer to GPU.");
3940 }
3941
3942
3943
3944 //NOTE: might need to do watchdog division for this, hopefully not though.
3945 void spath_raytracer_kd_collision(spath_raytracer_context* sprctx)
3946 {
3947     int err;
3948
3949     cl_kernel kernel = sprctx->rctx->program->raw_kernels[KDTREE_INTERSECTION_INDX];
3950
3951
3952     clSetKernelArg(kernel, 0, sizeof(cl_mem), &sprctx->cl_path_collision_result_buffer);
3953     clSetKernelArg(kernel, 1, sizeof(cl_mem), &sprctx->rctx->cl_ray_buffer);
3954
3955     clSetKernelArg(kernel, 2, sizeof(cl_mem), &sprctx->cl_bad_api_design_buffer); //BAD
3956
3957     clSetKernelArg(kernel, 3, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_buffer);
3958     clSetKernelArg(kernel, 4, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_index_buffer.image);
3959     clSetKernelArg(kernel, 5, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_vert_buffer.image);
3960
3961     clSetKernelArg(kernel, 6, sizeof(cl_mem), &sprctx->rctx->stat_scene->kdt->cl_kd_tree_buffer);
3962 //NOTE: WILL NOT WORK WITH ALL SITUATIONS:
3963     unsigned int num_rays = sprctx->rctx->width*sprctx->rctx->height;
3964     clSetKernelArg(kernel, 7, sizeof(unsigned int), &num_rays);
3965
3966
3967
3968     size_t global[1] = {sprctx->rctx->rcl->num_cores*16}; //ok I give up with the persistent threading.
3969     size_t local[1] = {sprctx->rctx->rcl->sint_size * sprctx->rctx->rcl->num_sint_per_multiprocessor}; //sprctx->rctx->rcl->sint_size; sprctx->rctx->rcl->num_sint_per_
3970
3971     err = clEnqueueNDRangeKernel(sprctx->rctx->rcl->commands, kernel, 1,
3972                               NULL, global, local, 0, NULL, NULL);
3973     ASRT_CL("Failed to execute kd tree traversal kernel");
3974
3975     err = clFinish(sprctx->rctx->rcl->commands);
3976     ASRT_CL("Something happened while executing kd tree traversal kernel");
3977
3978 }
3979
3980 void spath_raytracer_ray_test(spath_raytracer_context* sprctx)
3981 {
3982     int err;
3983
3984     cl_kernel kernel = sprctx->rctx->program->raw_kernels[KDTREE_RAY_DRAW_INDX];
3985
3986     clSetKernelArg(kernel, 0, sizeof(cl_mem), &sprctx->rctx->cl_output_buffer);
3987     clSetKernelArg(kernel, 1, sizeof(cl_mem), &sprctx->rctx->cl_ray_buffer);
3988

```

```

3989 clSetKernelArg(kernel, 2, sizeof(unsigned int), &sprctx->rctx->width);
3990 //NOTE: WILL NOT WORK WITH ALL SITUATIONS:
3991 unsigned int num_rays = sprctx->rctx->width*sprctx->rctx->height;
3992
3993 size_t global[1] = {num_rays};
3994
3995 err = clEnqueueNDRangeKernel(sprctx->rctx->rcl->commands, kernel, 1,
3996                             NULL, global, NULL, 0, NULL, NULL);
3997 ASRT_CL("Failed to execute kd tree traversal kernel");
3998
4000 err = clFinish(sprctx->rctx->rcl->commands);
4001 ASRT_CL("Something happened while executing kd tree traversal kernel");
4002
4003 err = clEnqueueReadBuffer(sprctx->rctx->rcl->commands, sprctx->rctx->cl_output_buffer, CL_TRUE, 0,
4004                             sprctx->rctx->width*sprctx->rctx->height*sizeof(int),
4005                             sprctx->rctx->output_buffer, 0, NULL, NULL );
4006 ASRT_CL("Failed to read output array");
4007
4008 }
4009
4010 void spath_raytracer_kd_test(spath_raytracer_context* sprctx)
4011 {
4012     int err;
4013
4014     cl_kernel kernel = sprctx->rctx->program->raw_kernels[KDTREE_TEST_DRAW_INDX];
4015
4016     clSetKernelArg(kernel, 0, sizeof(cl_mem), &sprctx->rctx->cl_output_buffer);
4017     clSetKernelArg(kernel, 1, sizeof(cl_mem), &sprctx->cl_path_collision_result_buffer);
4018
4019     clSetKernelArg(kernel, 2, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_material_buffer);
4020     clSetKernelArg(kernel, 3, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_buffer);
4021     clSetKernelArg(kernel, 4, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_index_buffer.image);
4022     clSetKernelArg(kernel, 5, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_vert_buffer.image);
4023     clSetKernelArg(kernel, 6, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_nrml_buffer.image);
4024
4025     clSetKernelArg(kernel, 7, sizeof(unsigned int), &sprctx->rctx->width);
4026 //NOTE: WILL NOT WORK WITH ALL SITUATIONS:
4027     unsigned int num_rays = sprctx->rctx->width*sprctx->rctx->height;
4028
4029     size_t global[1] = {num_rays};
4030
4031     err = clEnqueueNDRangeKernel(sprctx->rctx->rcl->commands, kernel, 1,
4032                             NULL, global, NULL, 0, NULL, NULL);
4033     ASRT_CL("Failed to execute kd tree traversal kernel");
4034
4035     err = clFinish(sprctx->rctx->rcl->commands);
4036     ASRT_CL("Something happened while executing kd tree test kernel");
4037
4038     err = clEnqueueReadBuffer(sprctx->rctx->rcl->commands, sprctx->rctx->cl_output_buffer, CL_TRUE, 0,
4039                             sprctx->rctx->width*sprctx->rctx->height*sizeof(int),
4040                             sprctx->rctx->output_buffer, 0, NULL, NULL );
4041
4042     ASRT_CL("Failed to read output array");
4043 }
4044
4045 void spath_raytracer_xor_rng(spath_raytracer_context* sprctx)
4046 {
4047     int err;
4048     cl_kernel kernel = sprctx->rctx->program->raw_kernels[XORSHIFT_BATCH_INDX];
4049     clSetKernelArg(kernel, 0, sizeof(cl_mem), &sprctx->cl_random_buffer);
4050
4051     size_t global = sprctx->rctx->width*sprctx->rctx->height;
4052
4053     err = clEnqueueNDRangeKernel(sprctx->rctx->rcl->commands, kernel, 1, NULL,
4054                             &global, NULL, 0, NULL, NULL);
4055     ASRT_CL("Failed to execute kernel");
4056     err = clFinish(sprctx->rctx->rcl->commands);
4057     ASRT_CL("Something happened while waiting for kernel to finish");
4058 }
4059
4060 void spath_raytracer_avg_to_out(spath_raytracer_context* sprctx)
4061 {
4062     int err;
4063     int useless = 0;
4064     cl_kernel kernel = sprctx->rctx->program->raw_kernels[F_BUF_TO_BYTE_BUF_AVG_KRNL_INDX];
4065     clSetKernelArg(kernel, 0, sizeof(cl_mem), &sprctx->rctx->cl_output_buffer);
4066     clSetKernelArg(kernel, 1, sizeof(cl_mem), &sprctx->cl_path_output_buffer);
4067     clSetKernelArg(kernel, 2, sizeof(cl_mem), &sprctx->cl_spath_progress_buffer);
4068     clSetKernelArg(kernel, 3, sizeof(unsigned int), &sprctx->rctx->width);
4069
4070     clSetKernelArg(kernel, 4, sizeof(unsigned int), &useless);
4071
4072     size_t global = sprctx->rctx->width*sprctx->rctx->height;
4073
4074     err = clEnqueueNDRangeKernel(sprctx->rctx->rcl->commands, kernel, 1, NULL,
4075                             &global, NULL, 0, NULL, NULL);
4076     ASRT_CL("Failed to execute kernel");
4077     err = clFinish(sprctx->rctx->rcl->commands);
4078     ASRT_CL("Something happened while waiting for kernel to finish");
4079
4080     err = clEnqueueReadBuffer(sprctx->rctx->rcl->commands, sprctx->rctx->cl_output_buffer, CL_TRUE, 0,
4081                             sprctx->rctx->width*sprctx->rctx->height*sizeof(int),
4082                             sprctx->rctx->output_buffer, 0, NULL, NULL );
4083
4084     ASRT_CL("Failed to read output array");
4085 }
4086
4087
4088 void spath_raytracer_trace_init(spath_raytracer_context* sprctx)
4089 {
4090     int err;
4091     unsigned int random_value_WACKO = rand();
4092     cl_kernel kernel = sprctx->rctx->program->raw_kernels[SEGMENTED_PATH_TRACE_INIT_INDX];
4093

```

```

4094 clSetKernelArg(kernel, 0, sizeof(cl_mem), &sprctx->cl_path_output_buffer);
4095 clSetKernelArg(kernel, 1, sizeof(cl_mem), &sprctx->rctx->cl_ray_buffer);
4096 clSetKernelArg(kernel, 2, sizeof(cl_mem), &sprctx->cl_path_ray_origin_buffer);
4097
4098 clSetKernelArg(kernel, 3, sizeof(cl_mem), &sprctx->cl_path_collision_result_buffer);
4099 clSetKernelArg(kernel, 4, sizeof(cl_mem), &sprctx->cl_path_origin_collision_result_buffer);
4100
4101 //SPATH DATA
4102 clSetKernelArg(kernel, 5, sizeof(cl_mem), &sprctx->cl_spath_progress_buffer);
4103
4104
4105 clSetKernelArg(kernel, 6, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_material_buffer);
4106 clSetKernelArg(kernel, 7, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_buffer);
4107 clSetKernelArg(kernel, 8, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_index_buffer.image);
4108 clSetKernelArg(kernel, 9, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_vert_buffer.image);
4109 clSetKernelArg(kernel, 10, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_nrml_buffer.image);
4110
4111 clSetKernelArg(kernel, 11, sizeof(unsigned int), &sprctx->rctx->width);
4112 clSetKernelArg(kernel, 12, sizeof(unsigned int), &random_value_WACKO);
4113
4114
4115 size_t global[1] = {sprctx->rctx->width*sprctx->rctx->height};
4116
4117 err = clEnqueueNDRangeKernel(sprctx->rctx->rcl->commands, kernel, 1,
4118 NULL, global, NULL, 0, NULL, NULL);
4119 ASRT_CL("Failed to execute kd tree traversal kernel");
4120
4121 err = clFinish(sprctx->rctx->rcl->commands);
4122 ASRT_CL("Something happened while executing kd init kernel");
4123
4124 }
4125
4126 void spath_raytracer_trace(spath_raytracer_context* sprctx)
4127 {
4128     int err;
4129     unsigned int random_value_WACKO = rand(); // sprctx->current_iteration; //TODO: make an actual random number
4130     cl_kernel kernel = sprctx->rctx->program->raw_kernels[SEGMENTED_PATH_TRACE_INDX];
4131
4132     unsigned int karg = 0;
4133     clSetKernelArg(kernel, karg++, sizeof(cl_mem), &sprctx->cl_path_output_buffer);
4134     clSetKernelArg(kernel, karg++, sizeof(cl_mem), &sprctx->rctx->cl_ray_buffer);
4135     clSetKernelArg(kernel, karg++, sizeof(cl_mem), &sprctx->cl_path_ray_origin_buffer);
4136
4137     clSetKernelArg(kernel, karg++, sizeof(cl_mem), &sprctx->cl_path_collision_result_buffer);
4138     clSetKernelArg(kernel, karg++, sizeof(cl_mem), &sprctx->cl_path_origin_collision_result_buffer);
4139     clSetKernelArg(kernel, karg++, sizeof(cl_mem), &sprctx->cl_spath_progress_buffer);
4140
4141     clSetKernelArg(kernel, karg++, sizeof(cl_mem), &sprctx->cl_random_buffer);
4142
4143     clSetKernelArg(kernel, karg++, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_material_buffer);
4144     clSetKernelArg(kernel, karg++, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_buffer);
4145     clSetKernelArg(kernel, karg++, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_index_buffer.image);
4146     clSetKernelArg(kernel, karg++, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_vert_buffer.image);
4147     clSetKernelArg(kernel, karg++, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_nrml_buffer.image);
4148
4149     clSetKernelArg(kernel, karg++, sizeof(unsigned int), &sprctx->rctx->width);
4150     clSetKernelArg(kernel, karg++, sizeof(unsigned int), &random_value_WACKO);
4151
4152     size_t global[1] = {sprctx->rctx->width*sprctx->rctx->height};
4153
4154     err = clEnqueueNDRangeKernel(sprctx->rctx->rcl->commands, kernel, 1,
4155         NULL, global, NULL, 0, NULL, NULL);
4156     ASRT_CL("Failed to execute kd tree traversal kernel");
4157
4158     err = clFinish(sprctx->rctx->rcl->commands);
4159     ASRT_CL("Something happened while executing kd tree traversal kernel");
4160
4161 }
4162
4163 void spath_raytracer_render(spath_raytracer_context* sprctx)
4164 {
4165     static int tbottle = 0;
4166     //Sleep(5000);
4167     int t1 = os_get_time_mili(abst);
4168
4169     //spath_raytracer_update_random(sprctx);
4170     spath_raytracer_xor_rng(sprctx);
4171     sprctx->current_iteration++;
4172     if(sprctx->current_iteration>sprctx->num_iterations)
4173     {
4174         if(!sprctx->render_complete)
4175             printf("Render took: %d ms", (unsigned int)os_get_time_mili(abst)-sprctx->start_time);
4176         sprctx->render_complete = true;
4177
4178         return;
4179     }
4180
4181     //spath_raytracer_ray_test(sprctx);
4182
4183
4184     bad_buf_update(sprctx);
4185     int t2 = os_get_time_mili(abst);
4186     spath_raytracer_kd_collision(sprctx);
4187     int t3 = os_get_time_mili(abst);
4188     spath_raytracer_trace(sprctx);
4189     int t4 = os_get_time_mili(abst);
4190     spath_raytracer_avg_to_out(sprctx);
4191     int t5 = os_get_time_mili(abst);
4192
4193     printf("num_gen: %d, collision: %d, trace: %d, draw: %d, time_since: %d, total: %d\n",
4194     t2-t1, t3-t2, t4-t3, t5-t4, t1-tbottle, t5-tbottle);
4195     //spath_raytracer_kd_test(sprctx);
4196     tbottle = os_get_time_mili(abst);
4197 }
4198

```

```

4199 void spath_raytracer_prepass(spath_raytracer_context* sprctx)
4200 {
4201     printf("Starting Split Path Raytracer Prepass. \n");
4202     sprctx->render_complete = false;
4203     sprctx->num_iterations = 256*4;//arbitrary default
4204     srand((unsigned int)os_get_time_mili(abst));
4205     sprctx->start_time = (unsigned int) os_get_time_mili(abst);
4206     bad_buf_update(sprctx);
4207
4208     zero_buffer(sprctx->rctx, sprctx->cl_path_output_buffer,
4209                 sprctx->rctx->width*sprctx->rctx->height*sizeof(vec4));
4210
4211     raytracer_prepass(sprctx->rctx);
4212
4213     sprctx->current_iteration = 0;
4214     zero_buffer(sprctx->rctx, sprctx->cl_spath_progress_buffer,
4215                 sprctx->rctx->width*sprctx->rctx->height*sizeof(spath_progress));
4216     _raytracer_gen_ray_buffer(sprctx->rctx);
4217
4218
4219
4220     spath_raytracer_kd_collision(sprctx);
4221
4222     spath_raytracer_trace_init(sprctx);
4223
4224     spath_raytracer_update_random(sprctx);
4225
4226     zero_buffer(sprctx->rctx, sprctx->rctx->cl_ray_buffer,
4227                 sprctx->rctx->width*sprctx->rctx->height*sizeof(ray));
4228
4229     printf("Finished Split Path Raytracer Prepass. \n");
4230 }
4231
4232 #include <ss_raytracer.h>
4233 #include <scene.h>
4234 #include <kdtree.h>
4235 #include <raytracer.h>
4236
4237 //Single sweep, as close to real time as this thing can support.
4238 void ss_raytracer_render(ss_raytracer_context* srctx)
4239 {
4240     int err;
4241     int start_time = os_get_time_mili(abst);
4242
4243     //TODO: @REFACTOR and remove prefix underscore and move to prepass
4244     _raytracer_gen_ray_buffer(srctx->rctx);
4245
4246
4247     cl_kernel kernel = srctx->rctx->program->raw_kernels[RAY_CAST_KRNL_INDX]; //just use the first one
4248
4249     float zeroed[] = {0., 0., 0., 1.};
4250     float* result = matvec_mul(srctx->rctx->stat_scene->camera_world_matrix, zeroed);
4251     clSetKernelArg(kernel, 0, sizeof(cl_mem), &srctx->rctx->cl_output_buffer);
4252     clSetKernelArg(kernel, 1, sizeof(cl_mem), &srctx->rctx->cl_ray_buffer);
4253     clSetKernelArg(kernel, 2, sizeof(cl_mem), &srctx->rctx->stat_scene->cl_material_buffer);
4254     clSetKernelArg(kernel, 3, sizeof(cl_mem), &srctx->rctx->stat_scene->cl_sphere_buffer);
4255     clSetKernelArg(kernel, 4, sizeof(cl_mem), &srctx->rctx->stat_scene->cl_plane_buffer);
4256     clSetKernelArg(kernel, 5, sizeof(cl_mem), &srctx->rctx->stat_scene->cl_mesh_buffer);
4257     clSetKernelArg(kernel, 6, sizeof(cl_mem), &srctx->rctx->stat_scene->cl_mesh_index_buffer.image);
4258     clSetKernelArg(kernel, 7, sizeof(cl_mem), &srctx->rctx->stat_scene->cl_mesh_vert_buffer.image);
4259     clSetKernelArg(kernel, 8, sizeof(cl_mem), &srctx->rctx->stat_scene->cl_mesh_nrml_buffer.image);
4260     clSetKernelArg(kernel, 9, sizeof(unsigned int), &srctx->rctx->width);
4261     clSetKernelArg(kernel, 10, sizeof(unsigned int), &srctx->rctx->height);
4262     clSetKernelArg(kernel, 11, sizeof(float)*4, result); //we only need 3
4263     //free(result);
4264
4265     size_t global;
4266
4267     global = srctx->rctx->width*srctx->rctx->height;
4268     err = clEnqueueNDRangeKernel(srctx->rctx->rcl->commands, kernel, 1, NULL, &global,
4269                                 NULL, 0, NULL, NULL);
4270     ASRT_CL("Failed to Execute Kernel");
4271
4272     err = clFinish(srctx->rctx->rcl->commands);
4273     ASRT_CL("Something happened during kernel execution");
4274
4275     err = clEnqueueReadBuffer(srctx->rctx->rcl->commands, srctx->rctx->cl_output_buffer, CL_TRUE, 0,
4276                               srctx->rctx->width*srctx->rctx->height*sizeof(int),
4277                               srctx->rctx->output_buffer, 0, NULL, NULL );
4278     ASRT_CL("Failed to read output array");
4279
4280     printf("SS Render Took %d ms.\n", os_get_time_mili(abst)-start_time);
4281 }
4282
4283 ss_raytracer_context* init_ss_raytracer_context(struct _rt_ctx* rctx)
4284 {
4285     ss_raytracer_context* ssctx = malloc(sizeof(ss_raytracer_context));
4286
4287     ssctx->rctx = rctx;
4288     ssctx->up_to_date = false;
4289     return ssctx;
4290 }
4291
4292
4293 //NOTE: @REFACTOR not used anymore should delete
4294 rt_vtable get_ss_raytracer_vtable()//TODO: don't use tbh.
4295 {
4296     rt_vtable v;
4297     v.up_to_date = false;
4298     //v.build      = &ss_raytracer_build;
4299     v.pre_pass    = &ss_raytracer_prepass;
4300     v.render_frame = &ss_raytracer_render;
4301     return v;
4302 }
4303

```



```

4409 //scene* rscene = (scene*) malloc(sizeof(scene));
4410
4411 os_start_thread(abst, web_server_start, rctx);
4412
4413 #ifdef DEV_MODE
4414     rctx->event_stack[rctx->event_position++] = SPLIT_PATH_RAYTRACER;
4415 #endif
4416
4417
4418
4419 scene* rscene = load_scene_json_url("scenes/path_obj3.rsc"); //TODO: support changing this during runtime
4420
4421 rctx->stat_scene = rscene;
4422 rctx->num_samples = 128; //NOTE: never actually used
4423
4424 ss_raytracer_context* ssrctx = NULL;
4425 path_raytracer_context* prctx = NULL;
4426 spath_raytracer_context* sprctx = NULL;
4427 int current_renderer = -1;
4428 bool global_up_to_date = false;
4429 while(should_run)
4430 {
4431     if(rctx->event_position)
4432     {
4433         if(!global_up_to_date)
4434         {
4435             raytracer_build(rctx);
4436             xm4_identity(rctx->stat_scene->camera_world_matrix);
4437             global_up_to_date = true;
4438         }
4439         switch(rctx->event_stack[--rctx->event_position])
4440         {
4441             case(SS_RAYTRACER):
4442             {
4443                 printf("Switching To SS Raytracer\n");
4444
4445                 if(current_renderer==SS_RAYTRACER)
4446                     break;
4447                 current_renderer = SS_RAYTRACER;
4448
4449                 os_draw_weird(abst);
4450                 os_update(abst);
4451
4452                 if(ssrctx==NULL)
4453                     ssrctx = init_ss_raytracer_context(rctx);
4454
4455                 ss_raytracer_prepass(ssrctx);
4456
4457                 break;
4458             }
4459             case(PATH_RAYTRACER):
4460             {
4461                 printf("Switching To Path Tracer\n");
4462                 if(current_renderer==PATH_RAYTRACER)
4463                     break;
4464                 current_renderer = PATH_RAYTRACER;
4465
4466                 os_draw_weird(abst);
4467                 os_update(abst);
4468
4469                 if(prctx==NULL)
4470                     prctx = init_path_raytracer_context(rctx);
4471
4472                 path_raytracer_prepass(prctx);
4473
4474                 break;
4475             }
4476             case(SPLIT_PATH_RAYTRACER):
4477             {
4478                 printf("Switching To Split Path Tracer\n");
4479                 if(current_renderer==SPLIT_PATH_RAYTRACER)
4480                     break;
4481                 current_renderer = SPLIT_PATH_RAYTRACER;
4482
4483                 os_draw_weird(abst);
4484                 os_update(abst);
4485
4486                 if(sprctx==NULL)
4487                     sprctx = init_spath_raytracer_context(rctx);
4488
4489                 spath_raytracer_prepass(sprctx);
4490
4491                 break;
4492             }
4493         }
4494     }
4495 }
4496
4497 switch(current_renderer)
4498 {
4499     case(SS_RAYTRACER):
4500     {
4501         ss_raytracer_render(ssrctx);
4502         break;
4503     }
4504     case(PATH_RAYTRACER):
4505     {
4506         path_raytracer_render(prctx);
4507         break;
4508     }
4509     case(SPLIT_PATH_RAYTRACER):
4510     {
4511         spath_raytracer_render(sprctx);
4512         break;
4513     }
}

```

```

4514     }
4515     os_update(abst);
4516 }
4517 //all of below shouldn't be a thing.
4518
4519 raytracer_build(rctx);
4520 raytracer_prepass(rctx);
4522
4523 xm4_identity(rctx->stat_scene->camera_world_matrix);
4524
4525 float dist = 0.f;
4526
4527
4528 int _timer_store = 0;
4529 int _timer_counter = 0;
4530 float _timer_average = 0.0f;
4531 printf("Rendering:\n\n");
4532
4533 /* static float t = 0.0f; */
4534 /* t += 0.0005f; */
4535 /* dist = sin(t)+1; */
4536 /* //mat4 temp; */
4537 /* xm4_translate(rctx->stat_scene->camera_world_matrix, 0, dist, 0); */
4538 int real_start = os_get_time_mili(abst);
4539 while(should_run)
4540 {
4541
4542     if(should_pause)
4543         continue;
4544     int last_time = os_get_time_mili(abst);
4545
4546     if(kbhit())
4547     {
4548         switch (getc(stdin))
4549         {
4550             case 'c':
4551                 exit(1);
4552                 break;
4553             case 27: //ESCAPE
4554                 exit(1);
4555                 break;
4556             default:
4557                 break;
4558         }
4559     }
4560
4561 //raytracer_refined_render(rctx);
4562 raytracer_render(rctx);
4563 if(rctx->render_complete)
4564 {
4565     printf("\n\nRender took: %02i ms (%d samples)\n\n",
4566           os_get_time_mili(abst)-real_start, rctx->num_samples);
4567     break;
4568 }
4569
4570
4571 int mili = os_get_time_mili(abst)-last_time;
4572 _timer_store += mili;
4573 _timer_counter++;
4574 printf("\nFrame took: %02i ms, average per 20 frames: %0.2f, avg fps: %03.2f (%d/%d)    ",
4575       mili, _timer_average, 1000.0f/_timer_average,
4576       rctx->current_sample, rctx->num_samples);
4577 fflush(stdout);
4578 if(_timer_counter>20)
4579 {
4580     _timer_counter = 0;
4581     _timer_average = (float)(_timer_store)/20.f;
4582     _timer_store = 0;
4583 }
4584 os_update(abst);
4585 }
4586
4587
4588 }
4589
4590 int startup() //main function called from win32 abstraction
4591 {
4592 #ifdef WIN32
4593     abst = init_win32_abs();
4594 #else
4595     abst = init_osx_abs();
4596 #endif
4597     os_start(abst);
4598     os_start_thread(abst, run, NULL);
4599 //win32_start_thread(run, NULL);
4600
4601     os_loop_start(abst);
4602     return 0;
4603     /*
4604     printf("Hello World\n");
4605     testWin32();
4606     return 0; */
4607 }
4608 #include <ui.h>
4609 #include <ui_web.h> //TODO: rename to ui_data or something
4610 #include <mongoose.h>
4611 #include <parson.h>
4612 #include <raytracer.h>
4613
4614 static ui_ctx uctx;
4615
4616 //Mostly based off of the example code for the library.
4617
4618

```

```

4619 static const char *s_http_port = "8000";
4620 static struct mg_serve_http_opts s_http_server_opts;
4621
4622
4623 void handle_ws_request(struct mg_connection *c, char* data)
4624 {
4625
4626     JSON_Value *root_value;
4627     JSON_Object *root_object;
4628     root_value = json_parse_string(data);
4629     root_object = json_value_get_object(root_value);
4630
4631     switch((unsigned int)json_object_dotget_number(root_object, "type"))
4632     {
4633         case 0: //init
4634         {
4635             char buf[] = "{ \"type\":0, \"message\":\"Nothing Right Now.\"}";
4636             mg_send_websocket_frame(c, WEBSOCKET_OP_TEXT, buf, strlen(buf));
4637
4638             return;
4639         }
4640     }
4641     case 1: //action
4642     {
4643         switch((unsigned int)json_object_dotget_number(root_object, "action.type"))
4644         {
4645             case SS_RAYTRACER:
4646             {
4647                 if(uctx.rctx->event_position==32)
4648                     return;
4649                 printf("UI Event Queued: Switch To Single Bounce\n");
4650                 uctx.rctx->event_stack[uctx.rctx->event_position++] = SS_RAYTRACER;
4651                 return;
4652             }
4653             case PATH_RAYTRACER: //prepass
4654             {
4655                 if(uctx.rctx->event_position==32)
4656                     return;
4657                 printf("UI Event Queued: Switch To Path Raytracer\n");
4658                 uctx.rctx->event_stack[uctx.rctx->event_position++] = PATH_RAYTRACER;
4659                 return;
4660             }
4661             case SPLIT_PATH_RAYTRACER: //start render
4662             {
4663                 if(uctx.rctx->event_position==32)
4664                     return;
4665                 printf("UI Event Queued: Switch To Split Path Raytracer\n");
4666                 uctx.rctx->event_stack[uctx.rctx->event_position++] = SPLIT_PATH_RAYTRACER;
4667                 return;
4668             }
4669             case 3: //start render
4670             {
4671                 if(uctx.rctx->event_position==32)
4672                     return;
4673                 printf("Change Scene %s\n", json_object_dotget_string(root_object, "action.scene"));
4674                 uctx.rctx->event_stack[uctx.rctx->event_position++] = 3;
4675                 printf("Not supported\n");
4676                 return;
4677             }
4678         }
4679         break;
4680     }
4681 }
4682 case 2: //send kd tree to GE2
4683 {
4684
4685     printf("GE2 requested k-d tree.\n");
4686     //char buf[] = "{ \"type\":0, \"message\":\"Nothing Right Now.\"}";
4687     if(uctx.rctx->stat_scene->kdt->buffer!=NULL)
4688     {
4689
4690         mg_send_websocket_frame(c, WEBSOCKET_OP_TEXT, //TODO: put something for this (IT'S NOT TEXT)
4691                             uctx.rctx->stat_scene->kdt->buffer,
4692                             uctx.rctx->stat_scene->kdt->buffer_size);
4693     }
4694     else
4695         printf("ERROR: no k-d tree.\n");
4696
4697     break;
4698 }
4699 }
4700
4701 }
4702
4703 static void ev_handler(struct mg_connection *c, int ev, void *p) {
4704     if (ev == MG_EV_HTTP_REQUEST) {
4705         struct http_message *hm = (struct http_message *) p;
4706
4707         // We have received an HTTP request. Parsed request is contained in `hm`.
4708         // Send HTTP reply to the client which shows full original request.
4709         mg_send_head(c, 200, __src_ui_index_html_len, "Content-Type: text/html");
4710         mg_printf(c, "%.*s", (int) __src_ui_index_html_len, __src_ui_index_html);
4711     }
4712 }
4713
4714
4715 static void handle_ws(struct mg_connection *c, int ev, void* ev_data) {
4716     switch (ev)
4717     { //ignore confusing indentation
4718         case MG_EV_HTTP_REQUEST:
4719         {
4720             struct http_message *hm = (struct http_message *) ev_data;
4721             //TODO: do something here
4722             mg_send_head(c, 200, __src_ui_index_html_len, "Content-Type: text/html");
4723             mg_printf(c, "%.*s", (int) __src_ui_index_html_len, __src_ui_index_html);

```

```

4724     break;
4725 }
4726 case MG_EV_WEBSOCKET_HANDSHAKE_DONE:
4727 {
4728     printf("Websocket Handshake\n");
4729     break;
4730 }
4731 case MG_EV_WEBSOCKET_FRAME:
4732 {
4733     struct websocket_message *wm = (struct websocket_message *) ev_data;
4734     /* New websocket message. Tell everybody. */
4735     //struct mg_str d = {(char *) wm->data, wm->size};
4736     //printf("WOW K: %s\n", wm->data);
4737     handle_ws_request(c, wm->data);
4738     break;
4739 }
4740 }
4741
4742 //printf("TEST 3\n");
4743 //c->flags |= MG_F_SEND_AND_CLOSE;
4744 }
4745
4746 static void handle_ocp_li(struct mg_connection *c, int ev, void* ev_data) {
4747 if (ev == MG_EV_HTTP_REQUEST) {
4748     struct http_message *hm = (struct http_message *) ev_data;
4749
4750     // We have received an HTTP request. Parsed request is contained in `hm`.
4751     // Send HTTP reply to the client which shows full original request.
4752     mg_send_head(c, 200, __src_ui_ocp_li_woff_len, "Content-Type: application/font-woff");
4753     //c->send_mbuf = __src_ui_ocp_li_woff;
4754     //c->content_len = __src_ui_ocp_li_woff_len;
4755
4756     mg_send(c, __src_ui_ocp_li_woff, __src_ui_ocp_li_woff_len);
4757     //mg_printf(c, "%.*s", (int) __src_ui_ocp_li_woff_len, __src_ui_ocp_li_woff);
4758 }
4759 //printf("TEST 2\n");
4760 c->flags |= MG_F_SEND_AND_CLOSE;
4761 }
4762
4763
4764 static void handle_style(struct mg_connection* c, int ev, void* ev_data) {
4765 if (ev == MG_EV_HTTP_REQUEST) {
4766     struct http_message *hm = (struct http_message *) ev_data;
4767
4768     // We have received an HTTP request. Parsed request is contained in `hm`.
4769     // Send HTTP reply to the client which shows full original request.
4770     mg_send_head(c, 200, __src_ui_style_css_len, "Content-Type: text/css");
4771     mg_printf(c, "%.*s", (int) __src_ui_style_css_len, __src_ui_style_css);
4772 }
4773 //printf("TEST\n");
4774 c->flags |= MG_F_SEND_AND_CLOSE;
4775 }
4776
4777 void web_server_start(void* rctx)
4778 {
4779     uctx.rctx = rctx;
4780     struct mg_mgr mgr;
4781     struct mg_connection *c;
4782
4783     mg_mgr_init(&mgr, NULL);
4784     c = mg_bind(&mgr, s_http_port, ev_handler);
4785     mg_set_protocol_http_websocket(c);
4786     mg_register_http_endpoint(c, "/ocp_li.woff", handle_ocp_li);
4787     mg_register_http_endpoint(c, "/style.css", handle_style);
4788     mg_register_http_endpoint(c, "/ws", handle_ws);
4789
4790     printf("Web UI Hosted On Port %s\n", s_http_port);
4791
4792     for (;;) {
4793         mg_mgr_poll(&mgr, 1000);
4794     }
4795     mg_mgr_free(&mgr);
4796
4797     exit(1);
4798
4799 }
4800 #include <win32.h>
4801 #include <startup.h>
4802 #include <windows.h>
4803 #include <math.h>
4804 #include <stdio.h>
4805 #include <stdint.h>
4806 #include <assert.h>
4807 #include <stdio.h>
4808 #include <iо.h>
4809 #include <fcntl.h>
4810 const char CLASS_NAME[] = "Raytracer";
4811
4812
4813 static win32_context* ctx;
4814
4815 void win32_draw_meme(); //vague predef
4816
4817 os_abs init_win32_abs()
4818 {
4819     os_abs abstraction;
4820     abstraction.start_func = &win32_start;
4821     abstraction.loop_start_func = &win32_loop;
4822     abstraction.update_func = &win32_update;
4823     abstraction.sleep_func = &win32_sleep;
4824     abstraction.get_bitmap_memory_func = &win32_get_bitmap_memory;
4825     abstraction.get_time_mili_func = &win32_get_time_mili;
4826     abstraction.get_width_func = &win32_get_width;
4827     abstraction.get_height_func = &win32_get_height;
4828     abstraction.start_thread_func = &win32_start_thread;

```

```

4829     abstraction.draw_weird = &win32_draw_meme;
4830     return abstraction;
4831 }
4832
4833 void* get_bitmap_memory()
4834 {
4835     return ctx->bitmap_memory;
4836 }
4837
4838 void win32_draw_meme()
4839 {
4840     int width = ctx->width;
4841     int height = ctx->height;
4842
4843     int pitch = width*4;
4844     uint8_t* row = (uint8_t*)ctx->bitmap_memory;
4845
4846     for(int y = 0; y < height; y++)
4847     {
4848         uint8_t* pixel = (uint8_t*)row;
4849         for(int x = 0; x < width; x++)
4850         {
4851             *pixel = sin((float)x)/150)*255;
4852             ++pixel;
4853
4854             *pixel = cos((float)x)/10)*100;
4855             ++pixel;
4856
4857             *pixel = cos((float)y)/50)*255;
4858             ++pixel;
4859
4860             *pixel = 0;
4861             ++pixel;
4862             /* ((char*)ctx->bitmap_memory)[(x+y*width)*4] = (y%2) ? 0xff : 0x00; */
4863             /* ((char*)ctx->bitmap_memory)[(x+4*y*width)+1] = 0x00; */
4864             /* ((char*)ctx->bitmap_memory)[(x+4*y*width)+2] = (y%2) ? 0xff : 0x00; */
4865             /* ((char*)ctx->bitmap_memory)[(x+4*y*width)+3] = 0x00; */
4866         }
4867         row += pitch;
4868     }
4869 }
4870
4871 void win32_sleep(int mili)
4872 {
4873     Sleep(mili);
4874 }
4875
4876 void win32_resize_dib_section(int width, int height)
4877 {
4878     if(ctx->bitmap_memory)
4879         VirtualFree(ctx->bitmap_memory, 0, MEM_RELEASE);
4880
4881     ctx->width = width;
4882     ctx->height = height;
4883
4884     ctx->bitmap_info.bmiHeader.biSize          = sizeof(ctx->bitmap_info.bmiHeader);
4885     ctx->bitmap_info.bmiHeader.biWidth         = width;
4886     ctx->bitmap_info.bmiHeader.biHeight        = -height;
4887     ctx->bitmap_info.bmiHeader.biPlanes        = 1;
4888     ctx->bitmap_info.bmiHeader.biBitCount      = 32; //8 bits of paddingll
4889     ctx->bitmap_info.bmiHeader.biCompression    = BI_RGB;
4890     ctx->bitmap_info.bmiHeader.biSizeImage     = 0;
4891     ctx->bitmap_info.bmiHeader.biPelsPerMeter   = 0;
4892     ctx->bitmap_info.bmiHeader.biYpelsPerMeter  = 0;
4893     ctx->bitmap_info.bmiHeader.biClrUsed       = 0;
4894     ctx->bitmap_info.bmiHeader.biClrImportant   = 0;
4895
4896     //I could use BitBlit if it would increase spread.
4897     int bytes_per_pixel = 4;
4898     int bitmap_memory_size = (width*height)*bytes_per_pixel;
4899     ctx->bitmap_memory = VirtualAlloc(0, bitmap_memory_size, MEM_COMMIT, PAGE_READWRITE);
4900
4901 }
4902
4903 void win32_update_window(HDC device_context, HWND win, int width, int height)
4904 {
4905
4906     int window_height = height;//window_rect.bottom - window_rect.top;
4907     int window_width  = width;//window_rect.right - window_rect.left;
4908
4909
4910     //TODO: Replace with BitBlt this is way too slow... (we don't even need the scaling);
4911     StretchDIBits(device_context,
4912                     /* x, y, width, height, */
4913                     /* x, y, width, height, */
4914                     0, 0, ctx->width, ctx->height,
4915                     0, 0, window_width, window_height,
4916
4917                     ctx->bitmap_memory,
4918                     &ctx->bitmap_info,
4919                     DIB_RGB_COLORS, SRCCOPY);
4920 }
4921
4922
4923 LRESULT CALLBACK WndProc(HWND win, UINT msg, WPARAM wParam, LPARAM lParam)
4924 {
4925     switch(msg)
4926     {
4927         case WM_KEYDOWN:
4928             switch (wParam)
4929             {
4930                 case VK_ESCAPE:
4931                     loop_exit();
4932                     ctx->shouldRun = false;
4933                     break;

```

```

4934
4935     case VK_SPACE:
4936         loop_pause();
4937         break;
4938     default:
4939         break;
4940     }
4941     break;
4942 case WM_SIZE:
4943 {
4944     RECT drawable_rect;
4945     GetClientRect(win, &drawable_rect);
4946
4947     int height = drawable_rect.bottom - drawable_rect.top;
4948     int width = drawable_rect.right - drawable_rect.left;
4949     win32_resize_dib_section(width, height);
4950
4951     win32_draw_meme();
4952 } break;
4953 case WM_CLOSE:
4954     ctx->shouldRun = false;
4955     break;
4956 case WM_DESTROY:
4957     ctx->shouldRun = false;
4958     break;
4959 case WM_ACTIVATEAPP:
4960     OutputDebugStringA("WM_ACTIVATEAPP\n");
4961     break;
4962 case WM_PAINT:
4963 {
4964     PAINTSTRUCT paint;
4965     HDC device_context = BeginPaint(win, &paint);
4966     EndPaint(win, &paint);
4967
4968     /*int x = paint.rcPaint.Left;
4969     int y = paint.rcPaint.Top;
4970     int height = paint.rcPaint.Bottom - paint.rcPaint.Top;
4971     int width = paint.rcPaint.Right - paint.rcPaint.Left; */
4972     //PatBlt(device_context, x, y, width, height, BLACKNESS);
4973
4974     RECT drawable_rect;
4975     GetClientRect(win, &drawable_rect);
4976
4977     int height = drawable_rect.bottom - drawable_rect.top;
4978     int width = drawable_rect.right - drawable_rect.left;
4979
4980     GetClientRect(win, &drawable_rect);
4981     win32_update_window(device_context,
4982                          win, width, height);
4983
4984 } break;
4985 default:
4986     return DefWindowProc(win, msg, wParam, lParam);
4987 }
4988
4989 return 0;
4990
4991
4992
4993 int _WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
4994                 LPSTR lpCmdLine, int nCmdShow)
4995 {
4996
4997     ctx = (win32_context*) malloc(sizeof(win32_context));
4998
4999     ctx->instance = hInstance;
5000     ctx->nCmdShow = nCmdShow;
5001     ctx->wc.cbSize = sizeof(WNDCLASSEX);
5002     ctx->wc.style = CS_OWNDC|CS_HREDRAW|CS_VREDRAW;
5003     ctx->wc.lpfnWndProc = WndProc;
5004     ctx->wc.cbClsExtra = 0;
5005     ctx->wc.cbWndExtra = 0;
5006     ctx->wc.hInstance = hInstance;
5007     ctx->wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
5008     ctx->wc.hCursor = LoadCursor(NULL, IDC_ARROW);
5009     ctx->wc.hbrBackground = 0;//(HBRUSH)(COLOR_WINDOW+1);
5010     ctx->wc.lpszMenuName = NULL;
5011     ctx->wc.lpszClassName = CLASS_NAME;
5012     ctx->wc.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
5013
5014     if(!SetPriorityClass(
5015         GetCurrentProcess(),
5016         HIGH_PRIORITY_CLASS
5017     ))
5018     {
5019         printf("FUCKKKK!!!\n");
5020     }
5021
5022
5023
5024     startup();
5025
5026     return 0;
5027 }
5028
5029 int main()
5030 {
5031     //printf("JANKY WINMAIN OVERRIDE\n");
5032     return _WinMain(GetModuleHandle(NULL), NULL, GetCommandLineA(), SW_SHOWNORMAL);
5033 }
5034
5035 //Should Block the Win32 Update Loop.
5036 #define WIN32_SHOULD_BLOCK_LOOP
5037
5038 void win32_loop()

```

```

5039 {
5040     printf("Starting WIN32 Window Loop\n");
5041     MSG msg;
5042     ctx->shouldRun = true;
5043     while(ctx->shouldRun)
5044     {
5045 #ifdef WIN32_SHOULD_BLOCK_LOOP
5046
5047         if(GetMessage(&msg, 0, 0, 0) > 0)
5048         {
5049             TranslateMessage(&msg);
5050             DispatchMessage(&msg);
5051         }
5052     }
5053
5054 #else
5055     while(PeekMessage(&msg, 0, 0, 0, PM_REMOVE))
5056     {
5057         if(msg.message == WM_QUIT)
5058         {
5059             ctx->shouldRun = false;
5060         }
5061         TranslateMessage(&msg);
5062         DispatchMessage(&msg);
5063     }
5064 #endif
5065     //win32_draw_meme();
5066     //win32_update_window();
5067 }
5068 }
5069
5070
5071 void create_win32_window()
5072 {
5073     printf("Creating WIN32 Window\n");
5074
5075     ctx->win = CreateWindowEx(
5076         0,
5077         CLASS_NAME,
5078         CLASS_NAME,
5079         /* WS_OVERLAPPEDWINDOW, */
5080         (WS_POPUP| WS_SYSMENU | WS_MAXIMIZEBOX | WS_MINIMIZEBOX),
5081         CW_USEDEFAULT, CW_USEDEFAULT, 1920, 1080,
5082         NULL, NULL, ctx->instance, NULL);
5083
5084     if(ctx->win == NULL)
5085     {
5086         MessageBox(NULL, "Window Creation Failed!", "Error!",
5087                 MB_ICONEXCLAMATION | MB_OK);
5088         return;
5089     }
5090
5091     ShowWindow(ctx->win, ctx->nCmdShow);
5092     UpdateWindow(ctx->win);
5093 }
5094
5095
5096
5097 //NOTE: Should the start func start the Loop
5098 //#define WIN32_SHOULD_START_LOOP_ON_START
5099 void win32_start()
5100 {
5101     if(RegisterClassEx(&ctx->wc))
5102     {
5103         MessageBox(NULL, "Window Registration Failed!", "Error!",
5104                 MB_ICONEXCLAMATION | MB_OK);
5105         return;
5106     }
5107     create_win32_window();
5108 #ifdef WIN32_SHOULD_START_LOOP_ON_START
5109     win32_loop();
5110 #endif
5111 }
5112
5113
5114 int win32_get_time_mili()
5115 {
5116     SYSTEMTIME st;
5117     GetSystemTime(&st);
5118     return (int) st.wMilliseconds+(st.wSecond*1000)+(st.wMinute*1000*60);
5119 }
5120
5121 void win32_update()
5122 {
5123     //RECT win_rect;
5124     //GetClientRect(ctx->win, &win_rect);
5125     HDC dc = GetDC(ctx->win);
5126     win32_update_window(dc, ctx->win, ctx->width, ctx->height);
5127     ReleaseDC(ctx->win, dc);
5128 }
5129
5130
5131
5132 int win32_get_width()
5133 {
5134     return ctx->width;
5135 }
5136
5137 int win32_get_height()
5138 {
5139     return ctx->height;
5140 }
5141
5142 void* win32_get_bitmap_memory()
5143 {

```

```

5144     return ctx->bitmap_memory;
5145 }
5146
5147
5148 typedef struct
5149 {
5150     void* data;
5151     void (*func)(void*);
5152 } thread_func_meta;
5153
5154 DWORD WINAPI thread_func(void* data)
5155 {
5156     if(!SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_HIGHEST))
5157     {
5158         DWORD dwError;
5159         dwError = GetLastError();
5160         printf(TEXT("Failed to change thread priority (%d)\n"), dwError);
5161     }
5162
5163     thread_func_meta* meta = (thread_func_meta*) data;
5164     (meta->func)(meta->data); //confusing syntax: call the passed function with the passed data
5165     free(meta);
5166     return 0;
5167 }
5168
5169 void win32_start_thread(void (*func)(void*), void* data)
5170 {
5171     thread_func_meta* meta = (thread_func_meta*) malloc(sizeof(thread_func_meta));
5172     meta->data = data;
5173     meta->func = func;
5174     HANDLE t = CreateThread(NULL, 0, thread_func, meta, 0, NULL);
5175     //if(SetThreadPriority(t, THREAD_PRIORITY_HIGHEST)==0)
5176     //    assert(false);
5177 }
5178 *****
5179 /* Types */
5180 *****
5181 *****
5182
5183 #define MESH_SCENE_DATA_PARAM imageid_buffer_t indices, imageid_buffer_t vertices, imageid_buffer_t normals
5184 #define MESH_SCENE_DATA      indices, vertices, normals
5185
5186 typedef struct //16 bytes
5187 {
5188     vec3 colour;
5189
5190     float reflectivity;
5191 } _attribute_((aligned(16))) material;
5192
5193 typedef struct
5194 {
5195     vec3 orig;
5196     vec3 dir;
5197 } ray;
5198
5199 typedef struct
5200 {
5201     bool did_hit;
5202     vec3 normal;
5203     vec3 point;
5204     float dist;
5205     material mat;
5206 } collision_result;
5207
5208 typedef struct //32 bytes (one word)
5209 {
5210     vec3 pos;
5211     //4 bytes padding
5212     float radius;
5213     int material_index;
5214     //8 bytes padding
5215 } _attribute_((aligned(16))) sphere;
5216
5217 typedef struct plane
5218 {
5219     vec3 pos;
5220     vec3 normal;
5221
5222     int material_index;
5223 } _attribute_((aligned(16))) plane;
5224
5225 typedef struct
5226 {
5227
5228     mat4 model;
5229
5230     vec3 max;
5231     vec3 min;
5232
5233     int index_offset;
5234     int num_indices;
5235
5236
5237     int material_index;
5238 } _attribute_((aligned(32))) mesh; //TODO: align with cpu NOTE: I don't think we need 32
5239
5240 typedef struct
5241 {
5242     const __global material* material_buffer;
5243     const __global sphere* spheres;
5244     const __global plane* planes;
5245     //Mesh
5246     const __global mesh* meshes;
5247 } scene;
5248

```

```

5249 bool getTBoundingBox(vec3 vmin, vec3 vmax,
5250     ray r, float* tmin, float* tmax) //NOTE: could be wrong
5251 {
5252
5253     vec3 invD = 1/r.dir;///vec3(1/dir.x, 1/dir.y, 1/dir.z);
5254     vec3 t0s = (vmin - r.orig) * invD;
5255     vec3 tis = (vmax - r.orig) * invD;
5256
5257     vec3 tsmaller = min(t0s, tis);
5258     vec3 tbigger = max(t0s, tis);
5259
5260     *tmin = max(*tmin, max(tsmaller.x, max(tsmaller.y, tsmaller.z)));
5261     *tmax = min(*tmax, min(tbigger.x, min(tbigger.y, tbigger.z)));
5262
5263     return (*tmin < *tmax);
5264
5265     /* vec3 tmin = (vmin - r.orig) / r.dir; */
5266     /* vec3 tmax = (vmax - r.orig) / r.dir; */
5267
5268     /* vec3 real_min = min(tmin, tmax); */
5269     /* vec3 real_max = max(tmin, tmax); */
5270
5271     /* vec3 minmax = min(min(real_max.x, real_max.y), real_max.z); */
5272     /* vec3 maxmin = max(max(real_min.x, real_min.y), real_min.z); */
5273
5274     /* if (dot(minmax,minmax) >= dot(maxmin, maxmin)) */
5275     /* { */
5276     /*     *t_min = sqrt(dot(maxmin,maxmin)); */
5277     /*     *t_max = sqrt(dot(minmax,minmax)); */
5278     /*     return (dot(maxmin, maxmin) > 0.001f ? true : false); */
5279     /* } */
5280
5281     /* else return false; */
5282
5283
5284 bool hitBoundingBox(vec3 vmin, vec3 vmax,
5285     ray r)
5286 {
5287     vec3 tmin = (vmin - r.orig) / r.dir;
5288     vec3 tmax = (vmax - r.orig) / r.dir;
5289
5290     vec3 real_min = min(tmin, tmax);
5291     vec3 real_max = max(tmin, tmax);
5292
5293     vec3 minmax = min(min(real_max.x, real_max.y), real_max.z);
5294     vec3 maxmin = max(max(real_min.x, real_min.y), real_min.z);
5295
5296     if (dot(minmax,minmax) >= dot(maxmin, maxmin))
5297     { return (dot(maxmin, maxmin) > 0.001f ? true : false); }
5298     else return false;
5299 }
5300
5301
5302
5303 ****
5304 /*
5305  *      Primitives
5306  */
5307 ****
5308
5309 ****
5310 /* Triangle */
5311 ****
5312
5313 //Moller-Trumbore
5314 //t u v = x y z
5315 bool does_collide_triangle(vec3 tri[4], vec3* hit_coords, ray r) //tri has extra for padding
5316 {
5317
5318     vec3 ab = tri[1] - tri[0];
5319     vec3 ac = tri[2] - tri[0];
5320
5321     vec3 pvec = cross(r.dir, ac); //Triple product
5322     float det = dot(ab, pvec);
5323
5324     if (det < EPSILON) // Behind or close to parallel.
5325         return false;
5326
5327     float invDet = 1.f / det;
5328     vec3 tvec = r.orig - tri[0];
5329
5330     //u
5331     hit_coords->y = dot(tvec, pvec) * invDet;
5332     if(hit_coords->y < 0 || hit_coords->y > 1)
5333         return false;
5334
5335     //v
5336     vec3 qvec = cross(tvec, ab);
5337     hit_coords->z = dot(r.dir, qvec) * invDet;
5338     if (hit_coords->z < 0 || hit_coords->y + hit_coords->z > 1)
5339         return false;
5340
5341     //t
5342     hit_coords->x = dot(ac, qvec) * invDet;
5343
5344
5345     return true; //goose
5346 }
5347
5348
5349 ****
5350 /* Sphere */
5351 ****
5352
5353 bool does_collide_sphere(sphere s, ray r, float *dist)

```

```

5354 {
5355     float t0, t1; // solutions for t if the ray intersects
5356
5357     // analytic solution
5358     vec3 L = s.pos - r.orig;
5359     float b = dot(r.dir, L); /* 2.0f;
5360     float c = dot(L, L) - (s.radius*s.radius); //NOTE: you can optimize out the square.
5361
5362     float disc = b * b - c/*a*/; /* discriminant of quadratic formula */
5363
5364     /* solve for t (distance to hitpoint along ray) */
5365     float t = false;
5366
5367     if (disc < 0.0f) return false;
5368     else t = b - sqrt(disc);
5369
5370     if (t < 0.0f)
5371     {
5372         t = b + sqrt(disc);
5373         if (t < 0.0f) return false;
5374     }
5375     *dist = t;
5376     return true;
5377 }
5378
5379
5380
5381 *****/
5382 /* Plane */
5383 *****/
5384
5385 bool does_collide_plane(plane p, ray r, float *dist)
5386 {
5387     float denom = dot(r.dir, p.normal);
5388     if (denom < EPSILON) //Counter intuitive.
5389     {
5390         vec3 l = p.pos - r.orig;
5391         float t = dot(l, p.normal) / denom;
5392         if (t >= 0)
5393         {
5394             *dist = t;
5395             return true;
5396         }
5397     }
5398     return false;
5399 }
5400 }
5401
5402
5403 *****/
5404 /* */
5405 /* Meshes */
5406 /* */
5407 *****/
5408
5409
5410 bool does_collide_with_mesh(mesh collider, ray r, vec3* normal, float* dist, scene s,
5411                             MESH_SCENE_DATA_PARAM)
5412 {
5413     //TODO: k-d trees
5414     *dist = FAR_PLANE;
5415     float min_t = FAR_PLANE;
5416     vec3 hit_coord; //NOTE: currently unused
5417     ray r2 = r;
5418     if(!hitBoundingBox(collider.min, collider.max, r))
5419     {
5420         return false;
5421     }
5422
5423     for(int i = 0; i < collider.num_indices/3; i++) // each ivec3
5424     {
5425         vec3 tri[4];
5426
5427         //get vertex (first element of each index)
5428
5429         int4 idx_0 = read_imagei(indices, i*3+collider.index_offset+0);
5430         int4 idx_1 = read_imagei(indices, i*3+collider.index_offset+1);
5431         int4 idx_2 = read_imagei(indices, i*3+collider.index_offset+2);
5432
5433         tri[0] = read_imaged(vertices, idx_0.x).xyz;
5434         tri[1] = read_imaged(vertices, idx_1.x).xyz;
5435         tri[2] = read_imaged(vertices, idx_2.x).xyz;
5436
5437         //printf("%d/%d: (%f, %f, %f)\n", idx_0.x, collider.num_indices/3, tri[0].x, tri[0].y, tri[0].z);
5438         //printf("%d/%d: (%f, %f, %f)\n", idx_1.x, collider.num_indices/3, tri[1].x, tri[1].y, tri[1].z);
5439
5440         vec3 bc_hit_coords = (vec3)(0.f); //t u v = x y z
5441         if(does_collide_triangle(tri, &bc_hit_coords, r) &&
5442             bc_hit_coords.x<min_t && bc_hit_coords.x>0)
5443         {
5444             min_t = bc_hit_coords.x; //t (distance along direction)
5445             *normal =
5446                 read_imaged(normals, idx_0.y).xyz*(1-bc_hit_coords.y-bc_hit_coords.z) +
5447                 read_imaged(normals, idx_1.y).xyz*bc_hit_coords.y +
5448                 read_imaged(normals, idx_2.y).xyz*bc_hit_coords.z;
5449             //break; //convex optimization
5450         }
5451     }
5452
5453
5454
5455     *dist = min_t;
5456     return min_t != FAR_PLANE;
5457 }
5458 }
```

```

5459
5460 bool does_collide_with_mesh_nieve(mesh collider, ray r, vec3* normal, float* dist, scene s,
5461                                     imageId_buffer_t tree, MESH_SCENE_DATA_PARAM)
5462 {
5463     //TODO: k-d trees
5464     *dist = FAR_PLANE;
5465     float min_t = FAR_PLANE;
5466     vec3 hit_coord; //NOTE: currently unused
5467     ray r2 = r;
5468     if(!hitBoundingBox(collider.min, collider.max, r))
5469     {
5470         return false;
5471     }
5472
5473     for(int i = 0; i < collider.num_indices/3; i++) // each ivec3
5474     {
5475         vec3 tri[4];
5476
5477         //get vertex (first element of each index)
5478
5479         int4 idx_0 = read_imagei(indices, i*3+collider.index_offset+0);
5480         int4 idx_1 = read_imagei(indices, i*3+collider.index_offset+1);
5481         int4 idx_2 = read_imagei(indices, i*3+collider.index_offset+2);
5482
5483         tri[0] = read_imaged(vertices, idx_0.x).xyz;
5484         tri[1] = read_imaged(vertices, idx_1.x).xyz;
5485         tri[2] = read_imaged(vertices, idx_2.x).xyz;
5486
5487         //printf("%d/%d: (%f, %f, %f)\n", idx_0.x, collider.num_indices/3, tri[0].x, tri[0].y, tri[0].z);
5488         //printf("%d/%d: (%f, %f, %f)\n", idx_1.x, collider.num_indices/3, tri[1].x, tri[1].y, tri[1].z);
5489
5490         vec3 bc_hit_coords = (vec3)(0.f); //t u v = x y z
5491         if(does_collide_triangle(tri, &bc_hit_coords, r) &&
5492             bc_hit_coords.x<min_t && bc_hit_coords.x>0)
5493         {
5494             min_t = bc_hit_coords.x; //t (distance along direction)
5495             *normal =
5496                 read_imaged(normals, idx_0.y).xyz*(1-bc_hit_coords.y-bc_hit_coords.z) +
5497                 read_imaged(normals, idx_1.y).xyz*bc_hit_coords.y +
5498                 read_imaged(normals, idx_2.y).xyz*bc_hit_coords.z;
5499             //break; //convex optimization
5500         }
5501     }
5502 }
5503
5504
5505     *dist = min_t;
5506     return min_t != FAR_PLANE;
5507 }
5508
5509 bool does_collide_with_mesh_alt(mesh collider, ray r, vec3* normal, float* dist, scene s,
5510                                   MESH_SCENE_DATA_PARAM)
5511 {
5512     *dist = FAR_PLANE;
5513     float min_t = FAR_PLANE;
5514     vec3 hit_coord; //NOTE: currently unused
5515     ray r2 = r;
5516
5517     for(int i = 0; i < SCENE_NUM_INDICES/3; i++)
5518     {
5519         vec3 tri[4];
5520
5521         //get vertex (first element of each index)
5522
5523         int4 idx_0 = read_imagei(indices, i*3+collider.index_offset+0);
5524         int4 idx_1 = read_imagei(indices, i*3+collider.index_offset+1);
5525         int4 idx_2 = read_imagei(indices, i*3+collider.index_offset+2);
5526
5527         tri[0] = read_imaged(vertices, idx_0.x).xyz;
5528         tri[1] = read_imaged(vertices, idx_1.x).xyz;
5529         tri[2] = read_imaged(vertices, idx_2.x).xyz;
5530
5531
5532         vec3 bc_hit_coords = (vec3)(0.f); //t u v = x y z
5533         if(does_collide_triangle(tri, &bc_hit_coords, r) &&
5534             bc_hit_coords.x<min_t && bc_hit_coords.x>0)
5535         {
5536             min_t = bc_hit_coords.x; //t (distance along direction)
5537             *normal =
5538                 read_imaged(normals, idx_0.y).xyz*(1-bc_hit_coords.y-bc_hit_coords.z) +
5539                 read_imaged(normals, idx_1.y).xyz*bc_hit_coords.y +
5540                 read_imaged(normals, idx_2.y).xyz*bc_hit_coords.z;
5541         }
5542     }
5543 }
5544
5545
5546     *dist = min_t;
5547     return min_t != FAR_PLANE;
5548 }
5549 }
5550
5551
5552 ****
5553 /* High Level Collision */
5554 ****
5555 ****
5556
5557
5558 bool collide_meshes(ray r, collision_result* result, scene s, MESH_SCENE_DATA_PARAM)
5559 {
5560
5561     float dist = FAR_PLANE;
5562     result->did_hit = false;
5563     result->dist = FAR_PLANE;

```

```

5564
5565     for(int i = 0; i < SCENE_NUM_MESHES; i++)
5566     {
5567         mesh current_mesh = s.meshes[i];
5568         float local_dist = FAR_PLANE;
5569         vec3 normal;
5570         if(does_collide_with_mesh(current_mesh, r, &normal, &local_dist, s, MESH_SCENE_DATA))
5571         {
5572             if(local_dist<dist)
5573             {
5574                 dist = local_dist;
5575                 result->dist = dist;
5576                 result->normal = normal;
5577                 result->point = (r.dir*dist)+r.orig;
5578                 result->mat = s.material_buffer[current_mesh.material_index];
5579                 result->did_hit = true;
5580             }
5581         }
5582     }
5583     return result->did_hit;
5584 }
5585
5586
5587 bool collide_primitives(ray r, collision_result* result, scene s)
5588 {
5589
5590     float dist = FAR_PLANE;
5591     result->did_hit = false;
5592     result->dist = FAR_PLANE;
5593     for(int i = 0; i < SCENE_NUM_SPHERES; i++)
5594     {
5595         sphere current_sphere = s.spheres[i];//get_sphere(spheres, i);
5596         float local_dist = FAR_PLANE;
5597         if(does_collide_sphere(current_sphere, r, &local_dist))
5598         {
5599             if(local_dist<dist)
5600             {
5601                 dist = local_dist;
5602                 result->did_hit = true;
5603                 result->dist = dist;
5604                 result->point = r.dir*dist+r.orig;
5605                 result->normal = normalize(result->point - current_sphere.pos);
5606                 result->mat = s.material_buffer[current_sphere.material_index];
5607             }
5608         }
5609     }
5610
5611     for(int i = 0; i < SCENE_NUM_PLANES; i++)
5612     {
5613         plane current_plane = s.planes[i];//get_plane(planes, i);
5614         float local_dist = FAR_PLANE;
5615         if(does_collide_plane(current_plane, r, &local_dist))
5616         {
5617             if(local_dist<dist)
5618             {
5619                 dist = local_dist;
5620                 result->did_hit = true;
5621                 result->dist = dist;
5622                 result->point = r.dir*dist+r.orig;
5623                 result->normal = current_plane.normal;
5624                 result->mat = s.material_buffer[current_plane.material_index];
5625             }
5626         }
5627     }
5628
5629     return dist != FAR_PLANE;
5630 }
5631
5632 bool collide_all(ray r, collision_result* result, scene s, MESH_SCENE_DATA_PARAM)
5633 {
5634     float dist = FAR_PLANE;
5635     if(collide_primitives(r, result, s))
5636         dist = result->dist;
5637
5638     collision_result m_result;
5639     if(collide_meshes(r, &m_result, s, MESH_SCENE_DATA))
5640         if(m_result.dist < dist)
5641             *result = m_result;
5642
5643     return result->did_hit;
5644 }
5645 *****
5646 /* NOTE: Irradiance Caching is Incomplete */
5647 *****
5648
5649 *****
5650 /* Irradiance Caching */
5651 *****
5652
5653 __kernel void ic_hemisphere_sample(
5654     )
5655 {
5656
5657
5658
5659
5660 }
5661
5662 __kernel void ic_screen_textures(
5663     __write_only image2d_t pos_tex,
5664     __write_only image2d_t nrm_tex,
5665     const unsigned int width,
5666     const unsigned int height,
5667     const global float* ray_buffer,
5668     const vec4 pos,

```

```

5669 const __global material* material_buffer,
5670 const __global sphere* spheres,
5671 const __global plane* planes,
5672 const __global mesh* meshes,
5673 imageid_buffer_t indices,
5674 imageid_buffer_t vertices,
5675 imageid_buffer_t normals)
5676 {
5677     scene s;
5678     s.material_buffer = material_buffer;
5679     s.spheres = spheres;
5680     s.planes = planes;
5681     s.meshes = meshes;
5682
5683
5684     int id = get_global_id(0);
5685     int x = id%width;
5686     int y = id/width;
5687     int offset = x+y*width;
5688     int ray_offset = offset*3;
5689
5690     ray r;
5691     r.orig = pos.xyz; //NOTE: slow unaligned memory access.
5692     r.dir.x = ray_buffer[ray_offset];
5693     r.dir.y = ray_buffer[ray_offset+1];
5694     r.dir.z = ray_buffer[ray_offset+2];
5695
5696     collision_result result;
5697     if(!collide_all(r, &result, s, MESH_SCENE_DATA))
5698     {
5699         write_imagef(pos_tex, (int2)(x,y), (vec4)(0));
5700         write_imagef(nrm_tex, (int2)(x,y), (vec4)(0));
5701         return;
5702     }
5703
5704     write_imagef(pos_tex, (int2)(x,y), (vec4)(result.point,0)); //Maybe ???
5705     write_imagef(nrm_tex, (int2)(x,y), (vec4)(result.normal,0));
5706
5707     /* pos_tex[offset] = (vec4)(result.point,0); */
5708     /* nrm_tex[offset] = (vec4)(result.normal,0); */
5709 }
5710
5711
5712
5713 __kernel void generate_discontinuity(
5714     image2d_t pos_tex,
5715     image2d_t nrm_tex,
5716     __global float* out_tex,
5717     const float k,
5718     const float intensity,
5719     const unsigned int width,
5720     const unsigned int height)
5721 {
5722     int id = get_global_id(0);
5723     int x = id%width;
5724     int y = id/width;
5725     int offset = x+y*width;
5726
5727     //NOTE: this is fine for edges because the sampler is clamped
5728
5729     //Positions
5730     vec4 pm = read_imagef(pos_tex, sampler, (int2)(x,y));
5731     vec4 pu = read_imagef(pos_tex, sampler, (int2)(x,y+1));
5732     vec4 pd = read_imagef(pos_tex, sampler, (int2)(x,y-1));
5733     vec4 pr = read_imagef(pos_tex, sampler, (int2)(x+1,y));
5734     vec4 pl = read_imagef(pos_tex, sampler, (int2)(x-1,y));
5735
5736     //NOTE: slow doing this many distance calculations
5737     float posDiff = max(distance(pu,pm),
5738                         max(distance(pd,pm),
5739                             max(distance(pr,pm),
5740                                 distance(pl,pm))));;
5741     posDiff = clamp(posDiff, 0.f, 1.f);
5742     posDiff *= intensity;
5743
5744     //Normals
5745     vec4 nm = read_imagef(nrm_tex, sampler, (int2)(x,y));
5746
5747     vec4 nu = read_imagef(nrm_tex, sampler, (int2)(x,y+1));
5748     vec4 nd = read_imagef(nrm_tex, sampler, (int2)(x,y-1));
5749     vec4 nr = read_imagef(nrm_tex, sampler, (int2)(x+1,y));
5750     vec4 nl = read_imagef(nrm_tex, sampler, (int2)(x-1,y));
5751     //NOTE: slow doing this many distance calculations
5752     float nrmDiff = max(distance(nu,nm),
5753                         max(distance(nd,nm),
5754                             max(distance(nr,nm),
5755                                 distance(nl,nm))));;
5756     nrmDiff = clamp(nrmDiff, 0.f, 1.f);
5757     nrmDiff *= intensity;
5758
5759     out_tex[offset] = k*nrmDiff+posDiff;
5760 }
5761
5762 __kernel void float_average(
5763     __global float* in_tex,
5764     __global float* out_tex,
5765     const unsigned int width,
5766     const unsigned int height,
5767     const int total)
5768 {
5769     int id = get_global_id(0);
5770     int x = id%width;
5771     int y = id/width;
5772     int offset = x+y*width;
5773

```

```

5774     out_tex[offset] += in_tex[offset]/(float)total;
5775 }
5776 }
5777 }
5778
5779 __kernel void mip_single_upsample( //nearest neighbour upsample.
5780     __global float* in_tex,
5781     __global float* out_tex,
5782     const unsigned int width, //Of upsampled
5783     const unsigned int height)//Of upsampled
5784 {
5785     int id = get_global_id(0);
5786     int x = id%width;
5787     int y = id/width;
5788     int offset = x+y*width;
5789
5790     out_tex[offset] = in_tex[(x+y*width)/2]; //truncated
5791 }
5792
5793 __kernel void mip_upsample( //nearest neighbour upsample.
5794     image2d_t in_tex,
5795     __write_only image2d_t out_tex, //NOTE: not having __write_only caused it to crash without err
5796     const unsigned int width, //Of upsampled
5797     const unsigned int height)//Of upsampled
5798 {
5799     int id = get_global_id(0);
5800     int x = id%width;
5801     int y = id/width;
5802
5803     write_imagef(out_tex, (int2)(x,y),
5804         read_imagef(in_tex, sampler, (float2)((float)x/.f, (float)y/.f)));
5805 }
5806
5807 __kernel void mip_upsample_scaled( //nearest neighbour upsample.
5808     image2d_t in_tex,
5809     __write_only image2d_t out_tex,
5810     const int s,
5811     const unsigned int width, //Of upsampled
5812     const unsigned int height)//Of upsampled
5813 {
5814     int id = get_global_id(0);
5815     int x = id%width;
5816     int y = id/width;
5817     float factor = pow(2.f, (float)s);
5818     write_imagef(out_tex, (int2)(x,y),
5819         read_imagef(in_tex, sampler, (float2)((float)x/factor, (float)y/factor)));
5820 }
5821 __kernel void mip_single_upsample_scaled( //nearest neighbour upsample.
5822     __global float* in_tex,
5823     __global float* out_tex,
5824     const unsigned int s,
5825     const unsigned int width, //Of upsampled
5826     const unsigned int height)//Of upsampled
5827 {
5828     int id = get_global_id(0);
5829     int x = id%width;
5830     int y = id/width;
5831     int factor = (int) pow(2.f, (float)s);
5832     int offset = x+y*width;
5833     int fwidth = width/factor;
5834     int fheight = height/factor;
5835
5836     out_tex[offset] = in_tex[(x/factor)+(y/factor)*(width/factor)]; //truncated
5837 }
5838
5839 //NOTE: not used
5840 __kernel void mip_reduce( //not the best
5841     image2d_t in_tex,
5842     __write_only image2d_t out_tex,
5843     const unsigned int width, //Of reduced
5844     const unsigned int height)//Of reduced
5845 {
5846     int id = get_global_id(0);
5847     int x = id%width;
5848     int y = id/width;
5849
5850
5851
5852     vec4 p00 = read_imagef(in_tex, sampler, (int2)(x*2, y*2));
5853
5854     vec4 p01 = read_imagef(in_tex, sampler, (int2)(x*2+1, y*2));
5855
5856     vec4 p10 = read_imagef(in_tex, sampler, (int2)(x*2, y*2+1));
5857
5858     vec4 p11 = read_imagef(in_tex, sampler, (int2)(x*2+1, y*2+1));
5859
5860     write_imagef(out_tex, (int2)(x,y), p00+p01+p10+p11/4.f);
5861 }
5862 #define KDTREE_LEAF 1
5863 #define KDTREE_NODE 2
5864
5865 //TODO: put in util
5866 #define DEBUG
5867 #ifdef DEBUG
5868 //NOTE: this will be slow.
5869 #define assert(x) \
5870     if (! (x)) \
5871     { \
5872         int i = 0;while(i++ < 100)printf("Assert(%s) failed in %s:%d\n", #x, __FUNCTION__, __LINE__); \
5873         return; \
5874     }
5875 #else
5876 #define assert(x) //NOTHING
5877 #endif
5878 typedef struct

```

```

5879 {
5880     uchar type;
5881
5882     uint num_triangles;
5883 } __attribute__((aligned(16))) kd_tree_leaf_template;
5884
5885 typedef struct
5886 {
5887     uchar type;
5888
5889     uint num_triangles;
5890     ulong triangle_start;
5891 } kd_tree_leaf;
5892
5893 typedef struct
5894 {
5895     uchar type;
5896     uchar k;
5897     float b;
5898
5899     ulong left_index;
5900     ulong right_index;
5901 } __attribute__((aligned(16))) kd_tree_node;
5902
5903
5904 typedef union a_vec3
5905 {
5906     vec3 v;
5907     float a[4];
5908 } a_vec3;
5909
5910 typedef struct kd_stack_elem
5911 {
5912     ulong node;
5913     float min;
5914     float max;
5915 } kd_stack_elem;
5916
5917 typedef __global uint4* kd_44_matrix;
5918
5919
5920 void kd_update_state(__global char* kd_tree, ulong indx, uchar* type,
5921                         kd_tree_node* node, kd_tree_leaf* leaf)
5922 {
5923
5924     *type = *((__global uchar*)(kd_tree+indx));
5925
5926     if(*type == KDTREE_LEAF)
5927     {
5928         kd_tree_leaf_template template = *((__global kd_tree_leaf_template*) (kd_tree + indx));
5929         leaf->type = template.type;
5930         leaf->num_triangles = template.num_triangles;
5931
5932         leaf->triangle_start = indx + sizeof(kd_tree_leaf_template);
5933     }
5934     else
5935         *node = *((__global kd_tree_node*) (kd_tree + indx));
5936
5937
5938 }
5939
5940 void dbg_print_node(kd_tree_node n)
5941 {
5942     printf("\nNODE: type: %u, k: %u, b: %f, l: %llu, r: %llu \n",
5943           (unsigned int) n.type, (unsigned int) n.k, n.b,
5944           n.left_index, n.right_index);
5945 }
5946
5947 void dbg_print_matrix(kd_44_matrix m)
5948 {
5949     printf("[%2u %2u %2u %2u]\n" \
5950           "[%2u %2u %2u %2u]\n" \
5951           "[%2u %2u %2u %2u]\n" \
5952           "[%2u %2u %2u %2u]\n\n",
5953           m[0].x, m[0].y, m[0].z, m[0].w,
5954           m[1].x, m[1].y, m[1].z, m[1].w,
5955           m[2].x, m[2].y, m[2].z, m[2].w,
5956           m[3].x, m[3].y, m[3].z, m[3].w);
5957 }
5958
5959 inline float get_elem(vec3 v, uchar k, kd_44_matrix mask)
5960 {
5961     k = min(k,(uchar)2);
5962     vec3 nv = select((vec3)(0), v, mask[k].xyz); //NOTE: it has to be MSB on the mask
5963
5964     return nv.x + nv.y + nv.z;
5965 }
5966
5967 //#define B 3*32 //batch size
5968 #define STACK_SIZE 16 //tune later
5969 #define LOAD_BALANCER_BATCH_SIZE 32
5970
5971
5972 _kernel void kdtree_intersection(
5973     __global kd_tree_collision_result* out_buf,
5974     __global ray* ray_buffer, //TODO: make vec4
5975
5976     __global uint* dumb_data, //NOTE: REALLY DUMB, you can't JUST have a global variable in ocl.
5977
5978 //Mesh
5979     __global mesh* meshes,
5980     imageid_buffer_t indices,
5981     imageid_buffer_t vertices,
5982     __global char* kd_tree, //TODO: use a higher alignment type
5983

```

```

5984     unsigned int num_rays)
5985 {
5986     const uint blocksize_x = BLOCKSIZE_X; //should be 32 //NOTE: REMOVED A THING
5987     const uint blocksize_y = BLOCKSIZE_Y;
5988
5989     uint x = get_local_id(0) % BLOCKSIZE_X; //id within the warp
5990     uint y = get_local_id(0) / BLOCKSIZE_X; //id of the warp in the SM
5991
5992     __local volatile int next_ray_array[BLOCKSIZE_Y];
5993     __local volatile int ray_count_array[BLOCKSIZE_Y];
5994     next_ray_array[y] = 0;
5995     ray_count_array[y] = 0;
5996
5997     kd_stack_elem stack[STACK_SIZE];
5998     uint stack_length = 0;
5999
6000 //NOTE: IT WAS CRASHING WHEN THE VECTORS WERENT ALLIGNED!!!!
6001     kd_44_matrix elem_mask = (kd_44_matrix)(dumb_data);
6002     __global uint* warp_counter = dumb_data+16;
6003
6004
6005 //NOTE: this block of variables is probably pretty bad for the cache
6006     ray r;
6007     float t_hit = INFINITY;
6008     vec2 hit_info = (vec2)(0,0);
6009     unsigned int tri_idx;
6010     float t_min, t_max;
6011     float scene_t_min = 0, scene_t_max = INFINITY;
6012     kd_tree_node node;
6013     kd_tree_leaf leaf;
6014     uchar current_type = KDTREE_NODE;
6015     bool pushdown = false;
6016     kd_tree_node root;
6017     uint ray_idx;
6018     uint ray_idx;
6019
6020     while(true)
6021     {
6022         uint tidx = x; // SINGLE WARPS WORTH OF WORK 0-32
6023         uint widx = y; // WARPS PER SM 0-4 (for example)
6024         __local volatile int* local_pool_ray_count = ray_count_array+widx;
6025         __local volatile int* local_pool_next_ray = next_ray_array+widx;
6026
6027         //Grab new rays
6028         if(tidx == 0 && *local_pool_ray_count <= 0) //only the first work item gets memory
6029         {
6030             *local_pool_next_ray = atomic_add(warp_counter, LOAD_BALANCER_BATCH_SIZE); //batch complete
6031
6032             *local_pool_ray_count = LOAD_BALANCER_BATCH_SIZE;
6033         }
6034     }
6035 //lol help there are no barriers
6036
6037     {
6038         ray_idx = *local_pool_next_ray + tidx;
6039
6040         if(ray_idx >= num_rays) //ray index is past num rays, work is done
6041             break;
6042
6043         if(tidx == 0)
6044         {
6045             *local_pool_next_ray += 32;
6046             *local_pool_ray_count -= 32;
6047         }
6048
6049         r = ray_buffer[ray_idx];
6050
6051         t_hit = INFINITY; //infinity
6052
6053         if(!getTBoundingBox((vec3) SCENE_MIN, (vec3) SCENE_MAX, r, &scene_t_min, &scene_t_max)) //SCENE_MIN is a macro
6054         {
6055             scene_t_max = -INFINITY;
6056         }
6057
6058         t_max = t_min = scene_t_min;
6059
6060         stack_length = 0;
6061         root = *((__global kd_tree_node*) kd_tree);
6062
6063     }
6064     stack_length = 0;
6065
6066     while(t_max < scene_t_max)
6067     {
6068         if(stack_length == (uint) 0)
6069         {
6070             node = root; //root
6071             current_type = KDTREE_NODE;
6072             t_min = t_max;
6073             t_max = scene_t_max;
6074             pushdown = true;
6075         }
6076         else
6077         {
6078             t_min = stack[stack_length-1].min;
6079             t_max = stack[stack_length-1].max;
6080             kd_update_state(kd_tree, stack[stack_length-1].node, &current_type, &node, &leaf);
6081
6082             stack_length--;
6083             pushdown = false;
6084         }
6085
6086         stack_length--;
6087         pushdown = false;
6088     }

```

```

6089
6090
6091     while(current_type != KDTREE_LEAF)
6092     {
6093         unsigned char k = node.k;
6094
6095         float t_split = (node.b - get_elem(r.orig, k, elem_mask)) /
6096                         get_elem(r.dir, k, elem_mask);
6097
6098         bool left_close =
6099             (get_elem(r.orig, k, elem_mask) < node.b) ||
6100             (get_elem(r.orig, k, elem_mask) == node.b && get_elem(r.dir, k, elem_mask) <= 0);
6101         ulong thing = left_close ? 0xffffffffffff : 0;
6102         ulong first = select(node.right_index, node.left_index,
6103                               thing);
6104         ulong second = select(node.left_index, node.right_index,
6105                               thing);
6106
6107         if( t_split > t_max || t_split <= 0) //NOTE: branching necessary
6108         {
6109             kd_update_state(kd_tree, first, &current_type, &node, &leaf);
6110         }
6111         else if(t_split < t_min)
6112         {
6113             kd_update_state(kd_tree, second, &current_type, &node, &leaf);
6114         }
6115         else
6116         {
6117             //assert(stack_length!=(ulong)STACK_SIZE-1);
6118
6119             stack[stack_length++] = (kd_stack_elem) {second, t_split, t_max}; //push
6120             kd_update_state(kd_tree, first, &current_type, &node, &leaf);
6121
6122             t_max = t_split;
6123             pushdown = false;
6124         }
6125
6126         if(pushdown)
6127         {
6128             root = node;//UPDATE
6129         }
6130     }
6131
6132     //Found Leaf
6133     for(ulong t = 0; t < leaf.num_triangles; t++)
6134     {
6135         //assert(leaf.triangle_start-t == 0);
6136         vec3 tri[4];
6137         unsigned int index_offset =
6138             *(__global uint*)(kd_tree+leaf.triangle_start+(t*sizeof(unsigned int)));
6139         //get vertex (first element of each index)
6140         int4 idx_0 = read_imagei(indices, index_offset+0);
6141         int4 idx_1 = read_imagei(indices, index_offset+1);
6142         int4 idx_2 = read_imagei(indices, index_offset+2);
6143
6144         tri[0] = read_imagef(vertices, idx_0.x).xyz;
6145         tri[1] = read_imagef(vertices, idx_1.x).xyz;
6146         tri[2] = read_imagef(vertices, idx_2.x).xyz;
6147         /*printf("%f %f %f : %f %f %f %llu\n",
6148            tri[0].x, tri[0].y, tri[0].z,
6149            tri[1].x, tri[1].y, tri[1].z,
6150            tri[2].x, tri[2].y, tri[2].z,
6151            t);*/
6152
6153
6154         vec3 hit_coords; // t u v
6155         if(does_collide_triangle(tri, &hit_coords, r)) //TODO: optimize
6156         {
6157             //printf("COLLISION\n");
6158             if(hit_coords.x<0)
6159                 continue;
6160             if(hit_coords.x < t_hit)
6161             {
6162                 t_hit = hit_coords.x; //t
6163                 hit_info = hit_coords.yz; //u v
6164                 tri_indx = index_offset;
6165             }
6166
6167             if(t_hit < t_min) // goes by closest to furthest, so if it hits it will be the closest
6168             {//early exit
6169                 //remove that
6170
6171                 scene_t_min = -INFINITY;
6172                 //break; //TODO: do something NOTE: COULD BE EVEN FASTER WHEN I ACTUALLY FIX THIS
6173             }
6174         }
6175     }
6176 }
6177
6178 }
6179 //By this point a triangle will have been found.
6180 kd_tree_collision_result result = {0};
6181
6182 if(!isinf(t_hit)//if t_hit != INFINITY
6183 {
6184     result.triangle_index = tri_indx;
6185     result.t = t_hit;
6186     result.u = hit_info.x;
6187     result.v = hit_info.y;
6188 }
6189
6190 out_buf[ray_indx] = result;
6191 }
6192
6193

```

```

6194 }
6195
6196 __kernel void kdtree_ray_draw(
6197     __global unsigned int* out_tex,
6198     __global ray* rays,
6199
6200     const unsigned int width)
6201 {
6202     const vec4 sky = (vec4) (0.84, 0.87, 0.93, 0);
6203     //return;
6204     int id = get_global_id(0);
6205     int x = id%width;
6206     int y = id/width;
6207     int offset = x+y*width;
6208
6209     ray r = rays[offset];
6210
6211     r.orig = (r.orig+1) / 2;
6212
6213     out_tex[offset] = get_colour( (vec4) (r.orig,1) );
6214 }
6215
6216
6217 __kernel void kdtree_test_draw(
6218     __global unsigned int* out_tex,
6219     __global kd_tree_collision_result* kd_results,
6220
6221     const __global material* material_buffer,
6222     //meshes
6223     __global mesh* meshes,
6224
6225     imageid_buffer_t indices,
6226     imageid_buffer_t vertices,
6227     imageid_buffer_t normals,
6228     const unsigned int width)
6229 {
6230     const vec4 sky = (vec4) (0.84, 0.87, 0.93, 0);
6231     //return;
6232     int id = get_global_id(0);
6233     int x = id%width;
6234     int y = id/width;
6235     int offset = x+y*width;
6236
6237     kd_tree_collision_result res = kd_results[offset];
6238     if(res.t==0)
6239     {
6240         out_tex[offset] = get_colour( (vec4) (0) );
6241         return;
6242     }
6243     int4 i1 = read_imagei(indices, res.triangle_index);
6244     int4 i2 = read_imagei(indices, res.triangle_index+1);
6245     int4 i3 = read_imagei(indices, res.triangle_index+2);
6246     mesh m = meshes[i1.w];
6247     material mat = material_buffer[m.material_index];
6248
6249     vec3 normal =
6250         read_imagef(normals, i1.y).xyz*(1-res.u-res.v) +
6251         read_imagef(normals, i2.y).xyz*res.u +
6252         read_imagef(normals, i3.y).xyz*res.v;
6253
6254     normal = (normal+1) / 2;
6255
6256     out_tex[offset] = get_colour( (vec4) (normal,1) );
6257 }
6258
6259 //TODO: ADD A THING FOR THIS
6260 //#pragma OPENCL EXTENSION cl_nv_pragma_unroll : enable
6261
6262 vec3 uniformSampleHemisphere(const float r1, const float r2)
6263 {
6264     float sinTheta = sqrt(1 - r1 * r1);
6265     float phi = 2 * M_PI_F * r2;
6266     float x = sinTheta * cos(phi);
6267     float z = sinTheta * sin(phi);
6268     return (vec3)(x, r1, z);
6269 }
6270 vec3 cosineSampleHemisphere(float u1, float u2, vec3 normal)
6271 {
6272     const float r = sqrt(u1);
6273     const float theta = 2 * M_PI_F * u2;
6274
6275     vec3 w = normal;
6276     vec3 axis = fabs(w.x) > 0.1f ? (vec3)(0.0f, 1.0f, 0.0f) : (vec3)(1.0f, 0.0f, 0.0f);
6277     vec3 u = normalize(cross(axis, w));
6278     vec3 v = cross(w, u);
6279
6280     /* use the coordinate frame and random numbers to compute the next ray direction */
6281     return normalize(u * cos(theta)*r + v*sin(theta)*r + w*sqrt(1.0f - u1));
6282 }
6283
6284 #define NUM_BOUNCES 4
6285 #define NUM_SAMPLES 4
6286
6287 typedef struct spath_progress
6288 {
6289     unsigned int sample_num;
6290     unsigned int bounce_num;
6291     vec3 mask;
6292     vec3 accum_color;
6293 } __attribute__((aligned (16))) spath_progress; //NOTE: space for two more 32 bit dudes
6294
6295 __kernel void segmented_path_trace_init(
6296     __global vec4* out_tex,
6297     __global ray* ray_buffer,
6298     __global ray* ray_origin_buffer,

```

```

6299     __global kd_tree_collision_result* kd_results,
6300     __global kd_tree_collision_result* kd_source_results,
6301     __global spath_progress* spath_data,
6302
6303     const __global material* material_buffer,
6304
6305 //Mesh
6306     const __global mesh* meshes,
6307     imageid_buffer_t indices,
6308     imageid_buffer_t vertices,
6309     imageid_buffer_t normals,
6310     /* const __global vec2* texcoords, */
6311     const unsigned int width,
6312     const unsigned int random_value)
6313 {
6314     const vec4 sky = (vec4) (0.16, 0.2, 0.2, 0)*2;
6315     int x = get_global_id(0)%width;
6316     int y = get_global_id(0)/width;
6317     int offset = (x+y*width);
6318
6319     kd_tree_collision_result res = kd_results[offset];
6320     ray r = ray_buffer[offset];
6321     ray_origin_buffer[offset] = r;
6322     kd_source_results[offset] = res;
6323
6324     spath_progress spd;
6325     spd.mask = (vec3)(1.0f, 1.0f, 1.0f);
6326     spd.accum_color = (vec3) (0, 0, 0);
6327
6328     if(res.t==0)
6329     {
6330         out_tex[offset] += sky;
6331         //return;
6332     }
6333
6334     unsigned int seed1 = random_value * x;
6335     unsigned int seed2 = random_value * y;
6336
6337     //if(spd.bounce_num == 0)
6338     //    spd.mask *= mat.colour;
6339
6340 #pragma unroll //NOTE: NVIDIA plugin
6341     for(int i = 0; i < 7; i++)
6342         get_random(&seed1, &seed2);
6343
6344
6345
6346 //MESSY CODE!
6347     float rand1 = get_random(&seed1, &seed2);
6348     float rand2 = get_random(&seed1, &seed2);
6349
6350
6351     int4 i1 = read_imagei(indices, res.triangle_index);
6352     int4 i2 = read_imagei(indices, res.triangle_index+1);
6353     int4 i3 = read_imagei(indices, res.triangle_index+2);
6354     mesh m = meshes[i1.w];
6355     material mat = material_buffer[m.material_index];
6356     vec3 pos = r.orig + r.dir*res.t;
6357
6358     vec3 normal =
6359         read_imagef(normals, i1.y).xyz*(1-res.u-res.v) +
6360         read_imagef(normals, i2.y).xyz*res.u +
6361         read_imagef(normals, i3.y).xyz*res.v;
6362
6363     spd.mask *= mat.colour;
6364
6365     ray sr;
6366     vec3 sample_dir = cosineSampleHemisphere(rand1, rand2, normal);
6367     sr.orig = pos + normal * 0.0001f; //sweet spot for epsilon
6368     sr.dir = sample_dir;
6369
6370     ray_buffer[offset] = sr;
6371     spath_data[offset] = spd;
6372 }
6373
6374 __kernel void segmented_path_trace(
6375     __global vec4* out_tex,
6376     __global ray* ray_buffer,
6377     __global ray* ray_origin_buffer,
6378     __global kd_tree_collision_result* kd_results,
6379     __global kd_tree_collision_result* kd_source_results,
6380     __global spath_progress* spath_data,
6381
6382     const __global unsigned int* random_buffer,
6383
6384     const __global material* material_buffer,
6385
6386 //Mesh
6387     const __global mesh* meshes,
6388     imageid_buffer_t indices,
6389     imageid_buffer_t vertices,
6390     imageid_buffer_t normals,
6391     /* const __global vec2* texcoords, */
6392     const unsigned int width,
6393     //const unsigned int rwidth,
6394     //const unsigned int soffset,
6395     const unsigned int random_value)
6396 {
6397     const vec4 sky = (vec4) (0.16, 0.2, 0.2, 0);
6398     // int x = (soffset*width)%get_global_id(0)%width;
6399     int x = get_global_id(0)%width;
6400     int y = get_global_id(0)/width;
6401     int offset = (x+y*width);
6402
6403     spath_progress spd = spath_data[offset];

```

```

6404
6405     if(spd.sample_num==256) //get this from the cpu
6406     {
6407         ray nr;
6408         nr.orig = (vec3)(0);
6409         nr.dir = (vec3)(0);
6410         ray_buffer[offset] = nr;
6411         return;
6412     }
6413     kd_tree_collision_result res;
6414     ray r;
6415
6416     if(spd.bounce_num > NUM_BOUNCES)
6417         printf("SHIT\n");
6418
6419
6420     res = kd_results[offset];
6421     r = ray_buffer[offset];
6422     //out_tex[offset] = (vec4) (1,0,1,1);
6423     //return;
6424
6425
6426     //RETRIEVE DATA
6427     int4 i1 = read_imagei(indices, res.triangle_index);
6428     int4 i2 = read_imagei(indices, res.triangle_index+1);
6429     int4 i3 = read_imagei(indices, res.triangle_index+2);
6430     mesh m = meshes[i1.w];
6431     material mat = material_buffer[m.material_index];
6432     vec3 pos = r.orig + r.dir*res.t;
6433     //pos = (vec3) (0, 0, -2);
6434
6435     vec3 normal =
6436         read_imagef(normals, i1.y).xyz*(1-res.u-res.v) +
6437         read_imagef(normals, i2.y).xyz*res.u +
6438         read_imagef(normals, i3.y).xyz*res.v;
6439
6440     //TODO: BETTER RANDOM PLEASE
6441
6442     //unsigned int seed1 = x*(1920-x)*((x*x*y*y*random_value)%get_global_id(0));
6443     //unsigned int seed2 = y*(1080-y)*((x*x*y*y*random_value)%get_global_id(0)); //random_value+(unsigned int)(sin((float)get_global_id(0))*get_global_id(0));
6444
6445     /* union { */
6446     /*     float f; */
6447     /*     unsigned int ui; */
6448     /* } res2; */
6449
6450     /* res2.f = (float)random_buffer[offset]*M_PI_F+x; //fill up the mantissa. */
6451     /* unsigned int seed1 = res2.ui + (int)(sin((float)x)*7.1f); */
6452
6453     /* res2.f = (float)random_buffer[offset]*M_PI_F+y; */
6454     /* unsigned int seed2 = y + (int)(sin((float)res2.ui)*7*3.f); */
6455
6456     unsigned int seed1 = random_buffer[offset]*random_value;
6457     unsigned int seed2 = random_buffer[offset];
6458
6459     //printf("%u\n", random_value);
6460
6461     //if(spd.bounce_num == 0)
6462     //    spd.mask *= mat.colour;
6463
6464     //#pragma unroll //NOTE: NVIDIA plugin
6465     for(int i = 0; i < 7; i++)
6466         get_random(&seed1, &seed2);
6467
6468     barrier();
6469
6470     //MESSY CODE!
6471     float rand1 = get_random(&seed1, &seed2);
6472     float rand2 = get_random(&seed1, &seed2);
6473
6474     //out_tex[offset] += (vec4)((vec3)(rand2*2) ,1);
6475     //return;
6476
6477     ray sr;
6478
6479     vec3 sample_dir = cosineSampleHemisphere(rand1, rand2, normal);
6480     sr.orig = pos + normal * 0.0001f; //sweet spot for epsilon
6481     sr.dir = sample_dir;
6482
6483
6484     //printf("%f help\n", res.t);
6485     //THE NEXT PART
6486     if(res.t==0)
6487     {
6488         //if(get_global_id(0)==500)
6489         //printf("SHIT PANT\n");
6490         spd.bounce_num = NUM_BOUNCES; //TODO: uncomment
6491         spd.accum_color += spd.mask * sky.xyz;
6492     }
6493     else
6494     {
6495         //NOTE: janky emission, if reflectivity is 1 emission is 2 (only for tests)
6496         spd.accum_color += spd.mask * (float)(mat.reflectivity==1.)*2; //NOTE: JUST ADD EMISSION
6497
6498         spd.mask *= mat.colour;
6499
6500         spd.mask *= dot(sr.dir, normal);
6501     }
6502
6503     spd.bounce_num++;
6504
6505     if(spd.bounce_num >= NUM_BOUNCES)
6506     {
6507         //if(get_global_id(0)==0)
6508         //printf("PUSH\n");

```

```

6509     spd.bounce_num = 0;
6510     spd.sample_num++;
6511     out_tex[offset] += (vec4) (clamp(spd.accum_color, 0.f, 1.f),1);
6512
6513 //START OF NEW
6514
6515
6516     res = kd_source_results[offset];
6517     r = ray_origin_buffer[offset];
6518     spd.mask = (vec3)(1.0f, 1.0f, 1.0f);
6519     spd.accum_color = (vec3) (0, 0, 0);
6520
6521
6522     if(res.t==0)
6523     {
6524         out_tex[offset] += sky;
6525         //printf("SHI\n");
6526         //return;
6527     }
6528
6529     i1 = read_imagei(indices, res.triangle_index);
6530     i2 = read_imagei(indices, res.triangle_index+1);
6531     i3 = read_imagei(indices, res.triangle_index+2);
6532     m = meshes[i1.w];
6533     mat = material_buffer[m.material_index];
6534     pos = r.orig + r.dir*res.t;
6535     //pos = (vec3) (0, 0, -2);
6536
6537     normal =
6538         read_imagef(normals, i1.y).xyz*(1-res.u-res.v) +
6539         read_imagef(normals, i2.y).xyz*res.u +
6540         read_imagef(normals, i3.y).xyz*res.v;
6541
6542     spd.mask *= mat.colour;
6543     if( (float)(mat.reflectivity==1.)) //TODO: just add an emmision value in material
6544     {
6545         spd.accum_color += spd.mask*2;
6546     }
6547
6548     sample_dir = cosineSampleHemisphere(rand1, rand2, normal);
6549     sr.orig = pos + normal * 0.0001f; //sweet spot for epsilon
6550     sr.dir = sample_dir;
6551     //printf("GOOD %f %f %f\n",spd.accum_color.x, spd.accum_color.y, spd.accum_color.z);
6552 }
6553
6554 ray_buffer[offset] = sr;
6555
6556 spath_data[offset] = spd;
6557
6558 }
6559
6560 __kernel void path_trace(
6561     __global vec4* out_tex,
6562     const __global ray* ray_buffer,
6563     const __global material* material_buffer,
6564     const __global sphere* spheres,
6565     const __global plane* planes,
6566     //Mesh
6567     const __global mesh* meshes,
6568     imageid_buffer_t indices,
6569     imageid_buffer_t vertices,
6570     imageid_buffer_t normals,
6571     /* const __global vec2* texcoords, */
6572     const unsigned int width,
6573     const vec4 pos,
6574     unsigned int magic)
6575 {
6576     scene s;
6577     s.material_buffer = material_buffer;
6578     s.spheres = spheres;
6579     s.planes = planes;
6580     s.meshes = meshes;
6581
6582
6583     const vec4 sky = (vec4) (0.16, 0.2, 0.2, 0);
6584     //return;
6585     int x = get_global_id(0);
6586     int y = get_global_id(1);
6587     //int x = id%width+ get_global_offset(0)%total_width;
6588     //int y = id/width/* + get_global_offset(0)/total_width*/;
6589     int offset = (x+y*width);
6590     //int ray_offset = offset; //NOTE: unnecessary w/ new rays
6591
6592     ray r;
6593     r = ray_buffer[offset];
6594     r.orig = pos.xyz;
6595     union {
6596         float f;
6597         unsigned int ui;
6598     } res;
6599
6600     res.f = (float)magic*M_PI_F+x; //fill up the mantissa.
6601     unsigned int seed1 = res.ui + (int)(sin((float)x)*7.1f);
6602
6603     res.f = (float)magic*M_PI_F+y;
6604     unsigned int seed2 = y + (int)(sin((float)res.ui)*7*3.f);
6605
6606     collision_result initial_result;
6607     if(!collide_all(r, &initial_result, s, MESH_SCENE_DATA))
6608     {
6609         out_tex[x+y*width] = sky;
6610         return;
6611     }
6612     barrier(0); //good ?
6613

```

```

6614 vec3 fin_colour = (vec3)(0.0f, 0.0f, 0.0f);
6615 for(int i = 0; i < NUM_SAMPLES; i++)
6616 {
6617
6618     vec3 accum_color = (vec3)(0.0f, 0.0f, 0.0f);
6619     vec3 mask = (vec3)(1.0f, 1.0f, 1.0f);
6620     ray sr;
6621     float rand1 = get_random(&seed1, &seed2);
6622     float rand2 = get_random(&seed1, &seed2);
6623
6624
6625     vec3 sample_dir = cosineSampleHemisphere(rand1, rand2, initial_result.normal);
6626     sr.orig = initial_result.point + initial_result.normal * 0.0001f; //sweet spot for epsilon
6627     sr.dir = sample_dir;
6628     mask *= initial_result.mat.colour;
6629     for(int bounces = 0; bounces < NUM_BOUNCES; bounces++)
6630     {
6631         collision_result result;
6632         if(!collide_all(sr, &result, s, MESH_SCENE_DATA))
6633         {
6634             accum_color += mask * sky.xyz;
6635             break;
6636         }
6637
6638
6639         rand1 = get_random(&seed1, &seed2);
6640         rand2 = get_random(&seed1, &seed2);
6641
6642         sample_dir = cosineSampleHemisphere(rand1, rand2, result.normal);
6643
6644         sr.orig = result.point + result.normal * 0.0001f; //sweet spot for epsilon
6645         sr.dir = sample_dir;
6646
6647         //NOTE: janky emission, if reflectivity is 1 emission is 2 (only for tests)
6648         accum_color += mask * (float)(result.mat.reflectivity==1.)*2; //NOTE: EMISSION
6649
6650
6651         mask *= result.mat.colour;
6652
6653         mask *= dot(sample_dir, result.normal);
6654     }
6655
6656     //barrier(0); //good?
6657
6658     accum_color = clamp(accum_color, 0.f, 1.f);
6659
6660     fin_colour += accum_color * (1.f/NUM_SAMPLES);
6661 }
6662 #ifdef _WIN32
6663 out_tx[offset] = (vec4)(fin_colour, 1);
6664 #else
6665 out_tx[offset] = (vec4)(fin_colour.zyx, 1);
6666 #endif
6667 }
6668
6669
6670 __kernel void buffer_average(
6671     __global uchar4* out_tx,
6672     __global uchar4* fresh_frame_tx,
6673     const unsigned int width,
6674     const unsigned int height,
6675     const unsigned int sample
6676     /*const unsigned int num_samples*/
6677 {
6678     int id = get_global_id(0);
6679     int x = id%width;
6680     int y = id/width;
6681     int offset = (x + y * width);
6682     // (n - 1) m[n-1] + a[n]
6683     // m[n] = -----
6684     // n
6685
6686     float x2 = ((float)sample-1.f)*( (float)out_tx[offset].x + (float)fresh_frame_tx[sample].x) /
6687     (float)sample;
6688
6689 //wo
6690     /*float4 temp = mix((float4)
6691     // (float)fresh_frame_tx[offset].x,
6692     // (float)fresh_frame_tx[offset].y,
6693     // (float)fresh_frame_tx[offset].z,
6694     // (float)fresh_frame_tx[offset].w),
6695     (float4)(
6696     // (float)out_tx[offset].x,
6697     // (float)out_tx[offset].y,
6698     // (float)out_tx[offset].z,
6699     // (float)out_tx[offset].w), 0.5f+((float)sample/2048.f/2.f));// );*/
6700
6701     /*vec4 temp = (float)
6702     // (float)fresh_frame_tx[offset].x,
6703     // (float)fresh_frame_tx[offset].y,
6704     // (float)fresh_frame_tx[offset].z,
6705     // (float)fresh_frame_tx[offset].w)/12.f;*/
6706     out_tx[offset] = (uchar4) ((unsigned char)x2,
6707                             (unsigned char)0,
6708                             (unsigned char)0,
6709                             (unsigned char)1.f);
6710
6711     /*fresh_frame_tx[offset]/(unsigned char)(1.f/(1-(float)sample/255))
6712     + out_tx[offset]/(unsigned char)(1.f/((float)sample/255));*/
6713
6714 __kernel void f_buffer_average(
6715     __global vec4* out_tx,
6716     __global vec4* fresh_frame_tx,
6717     const unsigned int width,
6718     const unsigned int height,

```

```

6719 const unsigned int num_samples,
6720 const unsigned int sample)
6721 {
6722     int id = get_global_id(0);
6723     int x = id%width;
6724     int y = id/width;
6725     int offset = (x + y * width);
6726
6727     //      (n - 1) m[n-1] + a[n]
6728     // m[n] = -----
6729     //           n
6730
6731     out_tex[offset] = ((sample-1) * out_tex[offset] + fresh_frame_tex[offset]) / (float) sample;
6732
6733
6734     //out_tex[offset] = mix(fresh_frame_tex[offset], out_tex[offset],
6735     //((float)sample)/(float)num_samples);
6736 }
6737
6738 __kernel void xorshift_batch(__global unsigned int* data)
6739 { //get_global_id is just a register, not a function
6740     uint d = data[get_global_id(0)];
6741     data[get_global_id(0)] = ((d << 1) | (d >> (sizeof(int)*8 - 1)))+1;//circular shift +1
6742 }
6743
6744 __kernel void f_buffer_to_byte_buffer_avg(
6745     __global unsigned int* out_tex,
6746     __global vec4* fresh_frame_tex,
6747     __global spath_progress* spath_data,
6748     const unsigned int width,
6749     const unsigned int sample_num)
6750 {
6751     int id = get_global_id(0);
6752     int x = id%width;
6753     int y = id/width;
6754     int offset = (x + y * width);
6755     //int roffset = (x + y * real);
6756
6757     vec4 data = fresh_frame_tex[offset];
6758     vec4 colour = data.w==0 ? (vec4)(0,0,0,0) : data.xyzw/data.w;
6759
6760     /* if(get_global_id(0)%(width*100) == 0) */
6761     /*     printf("%f %f %f %f \n", */
6762     /*             fresh_frame_tex[offset].x, */
6763     /*             fresh_frame_tex[offset].y, */
6764     /*             fresh_frame_tex[offset].z, */
6765     /*             fresh_frame_tex[offset].w, */
6766     /*             colour.w); */
6767     out_tex[offset] = get_colour(colour);///sample_num;
6768 }
6769
6770
6771 __kernel void f_buffer_to_byte_buffer(
6772     __global unsigned int* out_tex,
6773     __global vec4* fresh_frame_tex,
6774     const unsigned int width,
6775     const unsigned int height)
6776 {
6777     int id = get_global_id(0);
6778     int x = id%width;
6779     int y = id/width;
6780     int offset = (x + y * width);
6781     out_tex[offset] = get_colour(fresh_frame_tex[offset]);
6782 }
6783
6784 vec4 shade(collision_result result, scene s, MESH_SCENE_DATA_PARAM)
6785 {
6786     const vec3 light_pos = (vec3)(1,2, 0);
6787     vec3 nspace_light_dir = normalize(light_pos-result.point);
6788     vec4 test_lighting = (vec4) (clamp((float)dot(result.normal, nspace_light_dir), 0.0f, 1.0f));
6789     ray r;
6790     r.dir = nspace_light_dir;
6791     r.orig = result.point + nspace_light_dir*0.00001f;
6792     collision_result _cr;
6793     bool visible = !collide_all(r, &_cr, s, MESH_SCENE_DATA);
6794     test_lighting *= (vec4)(result.mat.colour, 1.0f);
6795     return visible*test_lighting/2;
6796 }
6797
6798
6799 __kernel void cast_ray_test(
6800     __global unsigned int* out_tex,
6801     const __global ray* ray_buffer,
6802     const __global material* material_buffer,
6803     const __global sphere* spheres,
6804     const __global plane* planes,
6805     //Mesh
6806     const __global mesh* meshes,
6807     imageid_buffer_t indices,
6808     imageid_buffer_t vertices,
6809     imageid_buffer_t normals,
6810     /* const __global vec2* texcoords, */
6811     /* , */
6812
6813
6814     const unsigned int width,
6815     const unsigned int height,
6816     const vec4 pos)
6817 {
6818     scene s;
6819     s.material_buffer = material_buffer;
6820     s.spheres = spheres;
6821     s.planes = planes;
6822     s.meshes = meshes;
6823

```

```

6824 const vec4 sky = (vec4) (0.84, 0.87, 0.93, 0);
6825 //return;
6826 int id = get_global_id(0);
6827 int x = id%width;
6828 int y = id/width;
6829 int offset = x+y*width;
6830 int ray_offset = offset;
6831
6832 ray r;
6833 r = ray_buffer[ray_offset];
6834 r.orig = pos.xyz; //NOTE: unnecesary rn, in progress of updating kernels w/ the new ray buffers.
6835
6836 //r.dir = (vec3)(0,0,-1);
6837
6838 //out_tex[x+y*width] = get_colour_signed((vec4)(r.dir,0));
6839 //out_tex[x+y*width] = get_colour_signed((vec4)(1,1,0,0));
6840 collision_result result;
6841 if(!collide_all(r, &result, s, MESH_SCENE_DATA))
6842 {
6843     out_tex[x+y*width] = get_colour( sky );
6844     return;
6845 }
6846
6847 vec4 colour = shade(result, s, MESH_SCENE_DATA);
6848
6849
6850 #define NUM_REFLECTIONS 2
6851 ray rays[NUM_REFLECTIONS];
6852 collision_result results[NUM_REFLECTIONS];
6853 vec4 colours[NUM_REFLECTIONS];
6854 int early_exit_num = NUM_REFLECTIONS;
6855 for(int i = 0; i < NUM_REFLECTIONS; i++)
6856 {
6857     if(i==0)
6858     {
6859         rays[i].orig = result.point + result.normal * 0.0001f; //NOTE: BIAS
6860         rays[i].dir = reflect(r.dir, result.normal);
6861     }
6862     else
6863     {
6864         rays[i].orig = results[i-1].point + results[i-1].normal * 0.0001f; //NOTE: BIAS
6865         rays[i].dir = reflect(rays[i-1].dir, results[i-1].normal);
6866     }
6867     if(collide_all(rays[i], results+i, s, MESH_SCENE_DATA))
6868     {
6869         colours[i] = shade(results[i], s, MESH_SCENE_DATA);
6870     }
6871     else
6872     {
6873         colours[i] = sky;
6874         early_exit_num = i;
6875         break;
6876     }
6877 }
6878 for(int i = early_exit_num-1; i > -1; i--)
6879 {
6880     if(i==NUM_REFLECTIONS-1)
6881         colours[i] = mix(colours[i], sky, results[i].mat.reflectivity);
6882     else
6883         colours[i] = mix(colours[i], colours[i+1], results[i].mat.reflectivity);
6884 }
6885
6886 colour = mix(colour, colours[0], result.mat.reflectivity);
6887
6888 out_tex[offset] = get_colour( colour );
6889
6890
6891 }
6892
6893
6894 //NOTE: it might be faster to make the ray buffer a multiple of 4 just to align with words...
6895 __kernel void generate_rays(
6896     __global ray* out_tx,
6897     const unsigned int width,
6898     const unsigned int height,
6899     const t_mat4 wcm)
6900 {
6901     int id = get_global_id(0);
6902     int x = id%width;
6903     int y = id/width;
6904     int offset = (x + y * width);
6905
6906     ray r;
6907
6908     float aspect_ratio = width / (float)height; // assuming width > height
6909     float cam_x = (2 * ((float)x + 0.5) / width) - 1) * tan(FOV / 2 * M_PI_F / 180) * aspect_ratio;
6910     float cam_y = (1 - 2 * (((float)y + 0.5) / height)) * tan(FOV / 2 * M_PI_F / 180);
6911
6912     //r.orig = matvec((float*)&wcm, (vec4)(0.0, 0.0, 0.0, 1.0)).xyz;
6913     //r.dir = matvec((float*)&wcm, (vec4)(cam_x, cam_y, -1.0f, 1)).xyz - r.orig;
6914
6915     r.orig = (vec3)(0, 0, 0);
6916     r.dir = (vec3)(cam_x, cam_y, -1.0f) - r.orig;
6917
6918     r.dir = normalize(r.dir);
6919
6920     out_tx[offset] = r;
6921 }
6922 #define FOV 80.0f
6923
6924 #define vec2 float2
6925 #define vec3 float3
6926 #define vec4 float4
6927
6928 #define EPSILON 0.0000001f

```

```

6929 #define FAR_PLANE 100000000
6930
6931 typedef float mat4[16];
6932
6933
6934
6935 *****
6936 /* Util */
6937 *****
6938
6939
6940 _constant sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE |
6941     CLK_ADDRESS_CLAMP_TO_EDGE |
6942     CLK_FILTER_NEAREST;
6943
6944 typedef struct
6945 {
6946     vec4 x;
6947     vec4 y;
6948     vec4 z;
6949     vec4 w;
6950 } _attribute_((aligned(16))) t_mat4;
6951
6952 typedef struct kd_tree_collision_result
6953 {
6954     unsigned int triangle_index;
6955     float t;
6956     float u;
6957     float v;
6958 } kd_tree_collision_result;
6959
6960 void swap_float(float *f1, float *f2)
6961 {
6962     float temp = *f2;
6963     *f2 = *f1;
6964     *f1 = temp;
6965 }
6966
6967 vec4 matvec(float* m, vec4 v)
6968 {
6969     return (vec4) (
6970         m[0+0*4]*v.x + m[1+0*4]*v.y + m[2+0*4]*v.z + m[3+0*4]*v.w,
6971         m[0+1*4]*v.x + m[1+1*4]*v.y + m[2+1*4]*v.z + m[3+1*4]*v.w,
6972         m[0+2*4]*v.x + m[1+2*4]*v.y + m[2+2*4]*v.z + m[3+2*4]*v.w,
6973         m[0+3*4]*v.x + m[1+3*4]*v.y + m[2+3*4]*v.z + m[3+3*4]*v.w );
6974 }
6975
6976 unsigned int get_colour(vec4 col)
6977 {
6978     unsigned int outCol = 0;
6979
6980     col = clamp(col, 0.0f, 1.0f);
6981
6982     outCol |= 0xff000000 & (unsigned int)(col.w*255)<<24;
6983     outCol |= 0x00ff0000 & (unsigned int)(col.x*255)<<16;
6984     outCol |= 0x0000ff00 & (unsigned int)(col.y*255)<<8;
6985     //outCol |= 0x000000ff & (unsigned int)(col.z*255);
6986     outCol |= 0x000000ff & (unsigned int)(col.z*255);
6987
6988     /* outCol |= 0xff000000 & min((unsigned int)(col.w*255), (unsigned int)255)<<24; */
6989     /* outCol |= 0x00ff0000 & min((unsigned int)(col.x*255), (unsigned int)255)<<16; */
6990     /* outCol |= 0x0000ff00 & min((unsigned int)(col.y*255), (unsigned int)255)<<8; */
6991     /* outCol |= 0x000000ff & min((unsigned int)(col.z*255), (unsigned int)255); */
6992     return outCol;
6993 }
6994
6995 static float get_random(unsigned int *seed0, unsigned int *seed1)
6996 {
6997     /* hash the seeds using bitwise AND operations and bitshifts */
6998     *seed0 = 36969 * ((*seed0) & 65535) + ((*seed0) >> 16);
6999     *seed1 = 18000 * ((*seed1) & 65535) + ((*seed1) >> 16);
7000     unsigned int ires = ((*seed0) << 16) + (*seed1);
7001     /* use union struct to convert int to float */
7002     union {
7003         float f;
7004         unsigned int ui;
7005     } res;
7006     //Maybe good, maybe not
7007
7008     res.ui = (ires & 0x007fffff) | 0x40000000; /* bitwise AND, bitwise OR */
7009     return (res.f - 2.0f) / 2.0f;
7010 }
7011
7012 uint MWC64X(uint2 *state) //http://cas.ee.ic.ac.uk/people/dt10/research/rngs-gpu-mwc64x.html
7013 {
7014     enum { A=429488335U };
7015     uint x=(*state).x, c=(*state).y; // Unpack the state
7016     uint res=x*c; // Calculate the result
7017     uint hi=mul_hi(x,A); // Step the RNG
7018     x=x*A+c;
7019     c=hi+(x<c);
7020     *state=(uint2)(x,c); // Pack the state back up
7021     return res; // Return the next result
7022 }
7023
7024 vec3 reflect(vec3 incidentVec, vec3 normal)
7025 {
7026     return incidentVec - 2.0f * dot(incidentVec, normal) * normal;
7027 }
7028
7029 _kernel void blit_float_to_output(
7030     _global unsigned int* out_tex,
7031     _global float* in�ts,
7032     const unsigned int width,
7033     const unsigned int height)

```

```

7034 {
7035     int id = get_global_id();
7036     int x = id%width;
7037     int y = id/width;
7038     int offset = x+y*width;
7039     out_tex[offset] = get_colour((vec4)(in_flts[offset]));
7040 }
7041
7042 __kernel void blit_float3_to_output(
7043     __global unsigned int* out_tex,
7044     image2d_t in_flts,
7045     const unsigned int width,
7046     const unsigned int height)
7047 {
7048     int id = get_global_id();
7049     int x = id%width;
7050     int y = id/width;
7051     int offset = x+y*width;
7052     out_tex[offset] = get_colour(read_imagef(in_flts, sampler, (float2)(x, y)));
7053 }
7054 <!DOCTYPE html>
7055 <html>
7056     <head>
7057         <link rel="stylesheet" href="./style.css">
7058         <title>Path Tracer UI</title>
7059     </head>
7060     <body>
7061         <div class="titleBody">
7062             <h1>Path Tracer UI</h1>
7063             <hr class = "titleBar">
7064         </div>
7065         <div style="text-align: right;">
7066             <p id="status"></p>
7067         </div>
7068         <div>
7069             <h2>Info:</h2>
7070             <p id="info_para"></p>
7071         </div>
7072
7073     <button onclick="send_sb_cmd()">Simple Raytracer</button>
7074     <button onclick="send_ss_cmd()">Path Raytracer</button>
7075     <button onclick="send_path_cmd()">Split Path Tracer</button>
7076
7077     <div>
7078         <input id="scene" type="text" value="scenes/path_obj_test.rsc">
7079         <button onclick="send_scene_change_cmd()">Change Scene</button>
7080     </div>
7081
7082
7083     <script language="javascript" type="text/javascript">
7084         var ws;
7085         function connect()
7086     {
7087         ws = new WebSocket('ws://' + Location.host + '/ws');
7088         if (!window.console) { window.console = { log: function() {} } };
7089         ws.onopen = function(ev)
7090         {
7091             console.log(ev);
7092             document.getElementById("status").innerHTML = "Connected."
7093             document.getElementById("status").style.color = "green";
7094             ws.send("{\"type\":0}"); //get init info.
7095         };
7096         ws.onerror = function(ev) { console.log(ev); };
7097         ws.onclose = function(ev) {
7098             console.log(ev);
7099             document.getElementById("status").innerHTML = "Disconnected."
7100             document.getElementById("status").style.color = "red";
7101             setTimeout(function() { connect(); }, 1000);
7102             ws = null;
7103         };
7104         ws.onmessage = function(ev) {
7105             console.log(ev);
7106             console.log(ev.data);
7107             parse_ws(JSON.parse(ev.data));
7108         };
7109     }
7110     connect();
7111
7112
7113
7114     function send_sb_cmd()
7115     {
7116         data = {
7117             type:1,
7118             action:{
7119                 type:0
7120             }
7121         }
7122         ws.send(JSON.stringify(data));
7123     }
7124     function send_ss_cmd()
7125     {
7126         data = {
7127             type:1,
7128             action:{
7129                 type:1
7130             }
7131         }
7132         ws.send(JSON.stringify(data));
7133     }
7134     function send_path_cmd()
7135     {
7136         data = {
7137             type:1,
7138             action:{
```

```

7139             type:2
7140         }
7141     }
7142     ws.send(JSON.stringify(data));
7143 }
7144
7145 function send_scene_change_cmd()
7146 {
7147     data = {
7148         type:1,
7149         action:{
7150             type : 3,
7151             scene : document.getElementById("scene").value
7152         }
7153     }
7154     ws.send(JSON.stringify(data));
7155 }
7156
7157 function parse_ws(data)
7158 {
7159     switch(data.type)
7160     {
7161         case 0:
7162         {
7163             document.getElementById('info_para').innerHTML = data.message;
7164             break;
7165         }
7166     }
7167 }
7168 /*window.onload = function() {
7169     document.getElementById('send_button').onClick = function(ev) {
7170         var msg = document.getElementById('send_input').value;
7171         document.getElementById('send_input').value = '';
7172         ws.send(msg);
7173     };
7174     document.getElementById('send_input').onkeypress = function(ev) {
7175         if (ev.keyCode == 13 || ev.which == 13) {
7176             document.getElementById('send_button').click();
7177         }
7178     };
7179 };
7180 */;
7181 </script>
7182
7183 </body>
7184 </html>
7185 h {
7186     color: black;
7187     font-family: office_code_pro_li;
7188     font-size: 72pt;
7189     /*text-align: center;*/
7190 }
7191 .titleBody {
7192     text-align: center;
7193 }
7194 h2{
7195     color: black;
7196     font-family: office_code_pro_li;
7197     font-size: 30pt;
7198 }
7199
7200 input[type=text] {
7201     background-color: #fff;
7202     border: 2px solid #000;
7203     color: black;
7204     font-family: office_code_pro_li;
7205     font-size: 10pt;
7206     margin: 4px 2px;
7207     padding: 12px 20px;
7208
7209     cursor: pointer;
7210     width: 40%;
7211 }
7212
7213 p{
7214     color: black;
7215     font-family: office_code_pro_li;
7216 }
7217
7218 button {
7219     background-color: #fff; /* Green */
7220     border: 2px solid #000;
7221     color: black;
7222     padding: 15px 32px;
7223     font-family: office_code_pro_li;
7224     text-align: center;
7225     text-decoration: none;
7226     display: inline-block;
7227     font-size: 16px;
7228     margin: 4px 2px;
7229     cursor: pointer;
7230 }
7231
7232 hr.titleBar {
7233     margin-block-start: 0;
7234 }
7235
7236 @font-face {
7237     font-family: office_code_pro_li;
7238     src: url(.ocp_li.woff);
7239 }#include <KdTreeComponent.h>
7240 #include <ge/entity/component/batches/PipelineComponentBatch.h>
7241 #include <ge/entity/component/ComponentManager.h>
7242 #include <ge/graphics/GraphicsCore.h>
7243 #include <ge/console/Log.h>

```

```

7244 #include <ge/entity/Entity.h>
7245 #include <ge/entity/EntityManager.h>
7246 #include <ge/entity/component/components/TransformComponent.h>
7247 #include <ge/entity/component/components/HLMeshComponent.h>
7248 #include <ge/engine/scene/Scene.h>
7249 #include <ge/debug/DebugBox.h>
7250 #include <ge/input/KeyboardHandler.h>
7251 #include <web_stuff.h>
7252 #include <GL/glew.h>
7253
7254 ge::Component* _constructor_KdTreeComponent(ge::Entity* ent)
7255 {
7256     return new KdTreeComponent(ent);
7257 }
7258
7259 ge::ComponentConstructorRegistry::StartupHook KdTreeComponent::_hook("KdTreeComponent", _constructor_KdTreeComponent);
7260
7261
7262 KdTreeComponent::KdTreeComponent(ge::Entity* e) : ge::Component(e)
7263 {
7264     addPublicVar("Maximum Depth", {ge::DataType::INT, &kd_max_depth});
7265     addPublicVar("Maximum Traversal Depth", {ge::DataType::INT, &kd_traversal_max_depth});
7266     addPublicVar("Traverse", {ge::DataType::BOOL, &kd_traverse});
7267
7268     kd_max_depth = 5;
7269     kd_traversal_max_depth = 1;
7270     kd_traverse = false;
7271 }
7272
7273 void KdTreeComponent::defaultInit()
7274 {
7275     //TODO: probably should remove this.
7276 }
7277
7278 void KdTreeComponent::insertToDefaultBatch() //TODO: add ingroup priorities
7279 {
7280     if(!ge::ComponentManager::containsComponentBatch("PipelineComponentBatch", getTypeName()))
7281     {
7282         ge::PipelineComponentBatch* cmp = new ge::PipelineComponentBatch();
7283         cmp->setComponentType(getTypeName());
7284
7285         ge::ComponentManager::registerComponentBatch(cmp);
7286     }
7287
7288     ge::ComponentManager::getComponentBatch("PipelineComponentBatch", getTypeName())->softInsert(this);
7289 }
7290
7291
7292 void GenerateBox(glm::vec3 min, glm::vec3 max) //NOTE: not optomized
7293 {
7294     ge::Entity* ent = new ge::Entity();
7295     ent->name = "_tree_node";
7296     ge::EntityManager::registerEntity(ent);
7297
7298     ge::TransformComponent* transform = new ge::TransformComponent(ent);
7299     transform->insertToDefaultBatch();
7300     transform->dynamic = true; //whatever tbh
7301     transform->setPosition((min + max)/2);
7302     transform->setScale(max-min);
7303     ent->insertComponent(transform);
7304
7305     ge::HLMeshComponent* renderer = new ge::HLMeshComponent(ent);
7306     renderer->insertToDefaultBatch();
7307     renderer->setMeshData("demo/meshes/cube.obj");
7308     renderer->setMaterial("CubeWire");
7309     renderer->mesh->getMesh()->cullBackface = false;
7310     ent->insertComponent(renderer);
7311 }
7312
7313 void GenerateBoxMAIN(glm::vec3 min, glm::vec3 max) //NOTE: not optomized
7314 {
7315     ge::Entity* ent = new ge::Entity();
7316     ent->name = "_tree_node";
7317     ge::EntityManager::registerEntity(ent);
7318
7319     ge::TransformComponent* transform = new ge::TransformComponent(ent);
7320     transform->insertToDefaultBatch();
7321     transform->dynamic = true; //whatever tbh
7322     transform->setPosition((min + max)/2);
7323     transform->setScale(max-min);
7324     ent->insertComponent(transform);
7325
7326     ge::HLMeshComponent* renderer = new ge::HLMeshComponent(ent);
7327     renderer->insertToDefaultBatch();
7328     renderer->setMeshData("demo/meshes/cube.obj");
7329     renderer->setMaterial("CubeWireRed");
7330     renderer->mesh->getMesh()->cullBackface = false;
7331     ent->insertComponent(renderer);
7332 }
7333
7334 void GenerateBoxTRAV(glm::vec3 min, glm::vec3 max) //NOTE: not optomized
7335 {
7336     ge::Entity* ent = new ge::Entity();
7337     ent->name = "_tree_node";
7338     ge::EntityManager::registerEntity(ent);
7339
7340     ge::TransformComponent* transform = new ge::TransformComponent(ent);
7341     transform->insertToDefaultBatch();
7342     transform->dynamic = true; //whatever tbh
7343     transform->setPosition((min + max)/2);
7344     transform->setScale(max-min);
7345     ent->insertComponent(transform);
7346
7347     ge::HLMeshComponent* renderer = new ge::HLMeshComponent(ent);
7348     renderer->insertToDefaultBatch();

```

```

7349 renderer->setMeshData("demo/meshes/cube.obj");
7350 renderer->setMaterial("OnyxTile");
7351 renderer->mesh->getMesh()->cullBackface = true;
7352 ent->insertComponent(renderer);
7353 }
7354
7355 typedef struct skd_tree_traversal_node
7356 {
7357     uint8_t type;
7358     uint8_t k;
7359     float b;
7360
7361     size_t left_ind; //NOTE: always going to be aligned by at least 4 (could multiply by four on gpu)
7362     size_t right_ind; //NOTE: I GIVE UP WITH LONGS JUST USE SIZE_T!
7363 } skd_tree_traversal_node;
7364
7365
7366 //serializable kd Leaf node
7367 typedef struct skd_tree_leaf_node
7368 {
7369     uint8_t type;
7370     unsigned int num_triangles;
7371     //uint tri 1
7372     //uint tri 2
7373     //uint etc...
7374 } skd_tree_leaf_node;
7375
7376 void KdTreeComponent::traversekd_and_gen(unsigned char* kd_start, unsigned char* kd, unsigned int depth,
7377                                         glm::vec3 min, glm::vec3 max)
7378 {
7379     if(depth>kd_max_depth)
7380         return;
7381     if(kd[0] == 2) //node
7382     {
7383         skd_tree_traversal_node n = *((skd_tree_traversal_node*)kd);
7384
7385         glm::vec3 l_max, r_min;
7386
7387         r_min = min;
7388         l_max = max;
7389
7390         r_min[n.k] = n.b;
7391         l_max[n.k] = n.b;
7392
7393         printf("test: %d %f\n", (int)n.k, n.b);
7394
7395         if(n.b > max[n.k])
7396             printf("BAD THING OVER MAX: %f\n", n.b);
7397         if(n.b < min[n.k])
7398             printf("BAD THING UNDER MIN: %f\n", n.b);
7399         GenerateBox(min, l_max);
7400         GenerateBox(r_min, max);
7401
7402         traversekd_and_gen(kd_start, kd_start+n.left_ind, depth+1, min, l_max);
7403         traversekd_and_gen(kd_start, kd_start+n.right_ind, depth+1, r_min, max);
7404     }
7405     else //Leaf
7406     {
7407         return;
7408     }
7409 }
7410 }
7411
7412 #define INFINITY 100000000
7413 #define KDTREE_LEAF 1
7414 #define STACK_SIZE 32
7415 #define KDTREE_NODE 2
7416
7417 bool getTBoundingBox(glm::vec3 origin, glm::vec3 dir,
7418                       glm::vec3 min, glm::vec3 max,
7419                       float* tmin, float* tmax) {
7420     glm::vec3 invD = glm::vec3(1/dir.x, 1/dir.y, 1/dir.z);
7421     glm::vec3 t0s = (min - origin) * invD;
7422     glm::vec3 t1s = (max - origin) * invD;
7423
7424     glm::vec3 tsmaller = glm::min(t0s, t1s);
7425     glm::vec3 tbigger = glm::max(t0s, t1s);
7426
7427     *tmin = glm::max(*tmin, glm::max(tsmaller[0], glm::max(tsmaller[1], tsmaller[2])));
7428     *tmax = glm::min(*tmax, glm::min(tbigger[0], glm::min(tbigger[1], tbigger[2])));
7429
7430     return (*tmin < *tmax);
7431 }
7432
7433 bool getTBoundingBox_old(glm::vec3 vmin, glm::vec3 vmax,
7434                           glm::vec3 rorig, glm::vec3 rdir, float* t_min, float* t_max) //NOTE: could be wrong
7435 {
7436     glm::vec3 tmin = (vmin - rorig) / rdir;
7437     glm::vec3 tmax = (vmax - rorig) / rdir;
7438
7439     glm::vec3 real_min = glm::min(tmin, tmax);
7440     glm::vec3 real_max = glm::max(tmin, tmax);
7441
7442     glm::vec3 minmax = glm::vec3(glm::min(glm::min(real_max.x, real_max.y), real_max.z)); //NOTE: wrong
7443     glm::vec3 maxmin = glm::vec3(glm::max(glm::max(real_min.x, real_min.y), real_min.z)); //NOTE: wrong
7444
7445     if (glm::dot(minmax, minmax) >= glm::dot(maxmin, maxmin))
7446     {
7447         *t_min = glm::dot(maxmin, maxmin);
7448         *t_max = glm::dot(minmax, minmax);
7449         return (glm::dot(maxmin, maxmin) > 0.001f ? true : false);
7450     }
7451     else return false;
7452 }
7453

```

```

7454 void kd_update_state(size_t idx, unsigned char* type,
7455             skd_tree_traversal_node* node, skd_tree_leaf_node* leaf)
7456 {
7457
7458     *type = kd_tree[idx];
7459
7460     if(*type == KDTREE_LEAF)
7461     {
7462         *leaf = *((skd_tree_leaf_node*) (kd_tree + idx));
7463     }
7464     else
7465     {
7466         *node = *((skd_tree_traversal_node*) (kd_tree + idx));
7467     }
7468 }
7469
7470
7471 typedef struct kd_stack_elem
7472 {
7473     size_t node;
7474     glm::vec3 Vmin;
7475     glm::vec3 Vmax;
7476     float min;
7477     float max;
7478 } kd_stack_elem;
7479
7480 void KdTreeComponent::traverse(glm::vec3 origin, glm::vec3 direction, glm::vec3 min, glm::vec3 max)
7481 {
7482
7483     kd_stack_elem stack[STACK_SIZE];
7484     unsigned int stack_length = 0;
7485
7486
7487     float t_hit = INFINITY;
7488     glm::vec2 hit_info = glm::vec2(0,0);
7489     unsigned int tri_idx;
7490     float t_min, t_max;
7491     float scene_t_min = -10000, scene_t_max = 10000;
7492     skd_tree_traversal_node node;
7493     skd_tree_leaf_node leaf;
7494     unsigned char current_type = KDTREE_NODE;
7495     bool pushdown = false;
7496     skd_tree_traversal_node root;
7497     unsigned int ray_idx;
7498
7499     glm::vec3 current_min = min;
7500     glm::vec3 current_max = max;
7501
7502     root = *(skd_tree_traversal_node*)kd_tree;
7503     if(!getTBoundingBox(origin, direction, min, max,
7504                         &scene_t_min, &scene_t_max)) //SCENE_MIN is a macro
7505     {
7506         scene_t_max = -INFINITY;//t_max = INFINITY;
7507         printf("Shit\n");
7508     }
7509     t_max = t_min = scene_t_min;
7510     printf("scene smin:%f smax:%f\n", scene_t_min, scene_t_max);
7511
7512     while(t_max < scene_t_max)
7513     {
7514         //printf("HIT BOUNDING BOX");
7515         if(stack_length == 0)
7516         {
7517             node = root; //root
7518             current_type = KDTREE_NODE;
7519             t_min = t_max;
7520             t_max = scene_t_max;
7521             pushdown = true;
7522         }
7523         else
7524         { //pop
7525
7526             //TODO: update this
7527             //WRONG: node = stack[stack_length-1].node;
7528             //assert(stack_length == 1);
7529             //printf("K it went around once!\n");
7530             t_min = stack[stack_length-1].min;
7531             t_max = stack[stack_length-1].max;
7532             current_min = stack[stack_length-1].Vmin;
7533             current_max = stack[stack_length-1].Vmax;
7534             current_type = *(kd_tree+stack[stack_length-1].node);
7535             //printf("K it retrieved some data from stack!\n");
7536             stack_length--;
7537             pushdown = false;
7538             printf("RETRIEVED FROM STACK\n");
7539         }
7540
7541
7542         while(current_type != KDTREE_LEAF)
7543         {
7544             GenerateBoxMAIN(current_min*0.99f, current_max*0.99f);
7545             //printf(":) ok test meme\n");
7546
7547             //NOTE: none of this branches
7548             unsigned char k = node.k;
7549             //assert(k<=2);
7550             //assert(node.type<=2);
7551             float t_split = (node.b - origin[k]) / direction[k];
7552             //if(get_local_id(&)==0)
7553             printf("\n%f\n", t_split);
7554             //NOTE: CRASH HEPENSE SOMEWHERE HERE
7555             //NOTE: something weird happens to the indexs of the nodes
7556             size_t first = origin[k] < node.b ? node.left_idx : node.right_idx; //wrong
7557             size_t second = origin[k] < node.b ? node.right_idx : node.left_idx; //wrong
7558             //printf(":) ok test meme %d %llu %llu %llu %llu\n", (int) node.type, first, second, node.left_index, node.right_index);

```

```

7559     //dbg_print_node(node);
7560     //assert(first<150000);
7561     //assert(second<150000);
7562     printf("Thing %d\n", (unsigned char)current_type);
7563
7564     if( t_split >= t_max || t_split < 0) //NOTE: branching necessary
7565     {
7566         if(first == node.left_ind)
7567             current_max[k] = node.b;
7568         else
7569             current_min[k] = node.b;
7570
7571         kd_update_state(first, &current_type, &node, &leaf);
7572     }
7573     else if(t_split <= t_min)
7574     {
7575         if(second == node.left_ind)
7576             current_max[k] = node.b;
7577         else
7578             current_min[k] = node.b;
7579         kd_update_state(second, &current_type, &node, &leaf);
7580     }
7581     else
7582     {
7583         //assert(stack_length==(ulong)STACK_SIZE);
7584         //update
7585         glm::vec3 new_min = min, new_max = max;
7586
7587         if(second == node.left_ind)
7588             new_min[k] = node.b;
7589         else
7590             new_max[k] = node.b;
7591
7592         stack[stack_length++] = {second, new_min, new_max, t_split, t_max}; //push
7593         kd_update_state(first, &current_type, &node, &leaf);
7594
7595         if(first == node.left_ind)
7596             current_max[k] = node.b;
7597         else
7598             current_min[k] = node.b;
7599
7600         //printf("test meme 1 complete\n");
7601         printf("test meme 1 %llu\n", stack_length);
7602
7603         t_max = t_split;
7604         pushdown = false;
7605     }
7606     printf("Thing2 %d\n", (unsigned char)current_type);
7607
7608     if(pushdown)
7609     {
7610         root = node;//UPDATE
7611     }
7612
7613 }
7614 //printf("HEY found something %d\n", Leaf.num_triangles);
7615 //Found Leaf
7616 if(leaf.num_triangles>0)
7617 {
7618     printf("ENTERED LEAF MAIN INTERSECTION test meme\n");
7619
7620     return;
7621     /*vec3 tri[4];
7622     unsigned int index_offset = *(_global uint*)(kd_tree+t);
7623     //get vertex (first element of each index)
7624     int4 idx_0 = read_imagei(indices, index_offset+0);
7625     int4 idx_1 = read_imagei(indices, index_offset+1);
7626     int4 idx_2 = read_imagei(indices, index_offset+2);
7627
7628     tri[0] = read_imagef(vertices, idx_0.x).xyz;
7629     tri[1] = read_imagef(vertices, idx_1.x).xyz;
7630     tri[2] = read_imagef(vertices, idx_2.x).xyz;
7631
7632     vec3 hit_coords; // t u v
7633     if(does_collide_triangle(tri, &hit_coords, r))
7634     {
7635         if(hit_coords.x < t_hit)
7636         {
7637             t_hit = hit_coords.x; //t
7638             hit_info = hit_coords.yz; //u v
7639             tri_idx = index_offset;
7640         }
7641         printf("COLLISION\n");
7642         //t_hit = min(t_hit, hit_coords.x); //t NOTE: this is faster but it saves computation is we store u and v aswell
7643         if(t_hit < t_min) // goes by closest to furthest, so if it hits it will be the closest
7644             //early exit
7645             break; //TODO: do something
7646         }
7647     }*/
7648 }
7649 }
7650 }
7651 printf("THIS THING SUCKS.");
7652 }
7653
7654 void KdTreeComponent::cycle()
7655 {
7656     static bool last_pressed = false;
7657     static bool last_pressedT = false;
7658
7659     if(ge::GraphicsCore::ctx->currentPipeline->getState()!=ge::PipelineState::Render)
7660         return;
7661     if(ge::GraphicsCore::ctx->currentPipeline->getCurrentStage()->type!=ge::PipelineDrawType::Default)
7662         return;
7663 }
```

```

7664 if(transformComponent == nullptr)
7665 {
7666     if(ent->components.count("TransformComponent"))
7667     {
7668         transformComponent = (ge::TransformComponent*) ent->components.at("TransformComponent");
7669     }
7670     else
7671     {
7672         ge::Log::wrn("KdTreeComponent", "A KdTreeComponent requires a TransformComponent.");
7673         return;
7674     }
7675 }
7676 if(ge::KeyboardHandler::keyDownSticky(ge::KeyboardKey::K) && !last_pressed)
7677 {
7678     retrieve_data = true;
7679     last_pressed = true;
7680 }
7681 else
7682     if(!ge::KeyboardHandler::keyDownSticky(ge::KeyboardKey::K))
7683         last_pressed = false;
7684
7685 if(ge::KeyboardHandler::keyDownSticky(ge::KeyboardKey::T) && !last_pressedT)
7686 {
7687     kd_traverse = true;
7688     last_pressedT = true;
7689 }
7690 else
7691     if(!ge::KeyboardHandler::keyDownSticky(ge::KeyboardKey::T))
7692         last_pressedT = false;
7693 const glm::vec3 min(-30.976175, -13.04981, -31.015106);
7694 const glm::vec3 max(30.976175, 17.071194, 25.015106);
7695
7696 if(new_data_retrieved)
7697 {
7698     new_data_retrieved = false;
7699     lkd_tree = kd_tree;
7700     lkd_tree_size = kd_tree_size;
7701     ge::Log::dbg("KdTreeComponent", "New Data has been retrieved.");
7702
7703     //glm::vec3 min(-1.328727, -1.108050, -10.190243);
7704     //glm::vec3 max(1.281850, 1.446441, -6.997774);
7705     //glm::vec3 min(-2.230403, -1.011691, -3.411920);
7706     //glm::vec3 max(1.244233, 0.678684, -2.960731);
7707
7708     traversekd_and_gen((unsigned char*)lkd_tree, (unsigned char*)lkd_tree, 0,
7709                         min, max);
7710     GenerateBoxMAIN(min,max);
7711     //THREE OF THEM ARE CORRECT.
7712 }
7713
7714 if(kd_traverse)
7715 {
7716     kd_traverse = false;
7717     traverse({0,0,0}, {0,-0.25,-1}, min, max);
7718 }
7719 //const glm::vec3 min(-1, -1, -1);
7720 //const glm::vec3 max(-1, -1, -1);
7721
7722 //ge::Debug::DebugBox::draw(glm::vec3(1,0.8,0.8), min+max/2, glm::vec3(0.2f));
7723
7724
7725
7726
7727
7728 }
7729
7730
7731
7732
7733 void KdTreeComponent::destroy()
7734 {
7735
7736 }
7737
7738 std::string KdTreeComponent::getTypeName()
7739 {
7740     return "KdTreeComponent";
7741 }

```