```c
#pragma once

//TODO: @REFACTOR file to just be memory_util


#ifdef _WIN32

#define W_ALIGN(x) __declspec( align (x) )
#define U_ALIGN(x) /*nothing*/
//This isn't specifically alignment.

#define alloca _alloca

#else

#define W_ALIGN(x) /*nothing*/
#define U_ALIGN(x) __attribute__(( aligned (x) ))

#endif
#pragma once
#include <alignment_util.h>
#include <stdbool.h>
#include <stdint.h>

typedef int   ivec3[4]; //1 int padding
typedef float vec2[2];
typedef float vec3[4];  //1 float padding
typedef float vec4[4];
typedef float mat4[16];

/*******/
/* Ray */
/*******/
typedef struct ray
{
    vec3 orig;
    vec3 dir;
    //float t_min, t_max;
} ray; //already aligned


/**************/
/* Voxel/AABB */
/**************/

typedef struct AABB
{
    vec3 max;
    vec3 min;
} AABB;

void  AABB_divide(AABB, uint8_t, float, AABB*, AABB*);
void  AABB_divide_world(AABB, uint8_t,  float, AABB*, AABB*);
float AABB_surface_area(AABB);
void  AABB_clip(AABB*, AABB*, AABB*);
float AABB_ilerp(AABB, uint8_t, float);
bool  AABB_is_planar(AABB*, uint8_t);

void  AABB_construct_from_vertices(AABB*, vec3*,  unsigned int);
void  AABB_construct_from_triangle(AABB*, ivec3*, vec3*);
/**********/
/* Sphere */
/**********/

//NOTE:  less memory efficient but aligns with opencl
typedef W_ALIGN(16) struct //sphere
{
    vec4 pos; //GPU stores all vec3s as vec4s in memory so we need the padding.

    float radius;
    int material_index;

}  U_ALIGN(16) sphere;


float does_collide_sphere(sphere, ray);


/*********/
/* Plane */
/*********/

typedef W_ALIGN(16) struct plane // bytes
{
    vec4 pos; //12
    //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
149             ic_octree_node* children[8];
150         } branch;
151     } data;
152     vec3 min;
153     vec3 max;
154 };
155
156 typedef struct
157 {
158     ic_octree_node* root;
159     int node_count;
160     unsigned int width;
161     unsigned int max_depth;
162 } ic_octree;
163
164 typedef struct
165 {
166     //vec4* texture;
167     cl_mem cl_image_ref;
168     unsigned int width, height;
169 }  ic_mipmap_gb;
170
171 typedef struct
172 {
173     //float* texture;
```

```
    cl_mem cl_image_ref;
    unsigned int width, height;
}  ic_mipmap_f;

typedef struct
{

    cl_image_format cl_standard_format;
    cl_image_desc   cl_standard_descriptor;
    ic_octree       octree;
    ic_ir_value*    ir_buf;
    unsigned int ir_buf_size;
    unsigned int ir_buf_current_offset;
}  ic_context;

void ic_init(raytracer_context*);
void ic_screenspace(raytracer_context*);
void ic_octree_init_branch(ic_octree_node*);
void ic_octree_insert(ic_context*, vec3 point, vec3 normal);
#pragma once
#include <stdint.h>
#include <stdbool.h>

struct scene;
//struct AABB;
//TODO: make these variable from the ui, eventually
#define KDTREE_KT 2.0f //Cost for traversal
#define KDTREE_KI 1.0f //Cost for intersection

#define KDTREE_LEAF 1
#define KDTREE_NODE 2


//serializable kd traversal node
typedef struct W_ALIGN(16) _skd_tree_traversal_node
{
    uint8_t type;
    uint8_t k;
    float   b;

    size_t left_ind;   //NOTE: always going to be aligned by at least 8 (could multiply by 8 on gpu)
    size_t right_ind;
} U_ALIGN(16) _skd_tree_traversal_node;


//serializable kd leaf node
typedef struct  W_ALIGN(16) _skd_tree_leaf_node
{
    uint8_t type;
    unsigned int num_triangles;
    //uint tri 1
    //uint tri 2
    //uint etc...
} U_ALIGN(16) _skd_tree_leaf_node;

typedef struct kd_tree_triangle_buffer
{
    unsigned int* triangle_buffer;
    unsigned int  num_triangles;
} kd_tree_triangle_buffer;

//NOTE: not using a vec3 for the floats because it would be a waste of space.
typedef struct kd_tree_collision_result
{
    unsigned int triangle_index;
    float t;
    float u;
    float v;
} kd_tree_collision_result;

//NOTE: should the depth be stored in here?
typedef struct kd_tree_node
{
    uint8_t k; //Splittign Axis
    float   b; //World Split plane

    struct kd_tree_node* left;
    struct kd_tree_node* right;

    kd_tree_triangle_buffer triangles;

} kd_tree_node;

typedef struct kd_tree
{
    kd_tree_node* root;
    unsigned int k; //Num dimensions, should always be three in this case
```

```
261
262
263
264     unsigned int num_nodes_total;
265     unsigned int num_tris_padded;
266     unsigned int num_traversal_nodes;
267     unsigned int num_leaves;
268     unsigned int num_indices_total;
269
270     unsigned int max_recurse;
271     unsigned int tri_for_leaf_threshold;
272
273     scene* s;
274     AABB bounds;
275
276     //Serialized form.
277     char* buffer;
278     unsigned int buffer_size;
279     cl_mem cl_kd_tree_buffer;
280
281     //AABB V; //Total bounding box
282
283 } kd_tree;
284
285
286 kd_tree*      kd_tree_init();
287 kd_tree_node* kd_tree_node_init();
288
289 bool kd_tree_node_is_leaf(kd_tree_node*);
290 void kd_tree_construct(kd_tree* tree); //O(n log^2 n) implementation
291 void kd_tree_generate_serialized(kd_tree* tree);
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 miliseconds);
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);              \
```

```
    i++;                                    \


typedef struct _rt_ctx raytracer_context;

typedef struct
{
    cl_platform_id platform_id;
    cl_device_id device_id;             // compute device id
    cl_context context;                // compute context
    cl_command_queue commands;         // compute command queue

    unsigned int simt_size;
    unsigned int num_simt_per_multiprocessor;
    unsigned int num_multiprocessors;
    unsigned int num_cores;

} rcl_ctx;

typedef struct
{
    cl_program program;
    cl_kernel* raw_kernels; //NOTE: not a good solution
    char*      raw_data;

} rcl_program;

typedef struct rcl_img_buf
{
    cl_mem buffer;
    cl_mem image;
    size_t size;
} rcl_img_buf;

void cl_info();
void create_context(rcl_ctx* context);
void load_program_raw(rcl_ctx* ctx, char* data, char** kernels, unsigned int num_kernels,
                      rcl_program* program, char** macros, unsigned int num_macros);
void load_program_url(rcl_ctx* ctx, char* url,  char** kernels, unsigned int num_kernels,
                      rcl_program* program, char** macros, unsigned int num_macros);
void test_sphere_raytracer(rcl_ctx* ctx, rcl_program* program,
                           sphere* spheres, int num_spheres,
                           uint32_t* bitmap, int width, int height);
cl_mem gen_rgb_image(raytracer_context* rctx,
                     const unsigned int width,
                     const unsigned int height);
cl_mem gen_grayscale_buffer(raytracer_context* rctx,
                            const unsigned int width,
                            const unsigned int height);
cl_mem gen_1d_image(raytracer_context* rctx, size_t t, void* ptr);
rcl_img_buf gen_1d_image_buffer(raytracer_context* rctx, size_t t, void* ptr);
void retrieve_buf(raytracer_context* rctx, cl_mem g_buf, void* c_buf, size_t);

void zero_buffer_img(raytracer_context* rctx, cl_mem buf, size_t element,
                 const unsigned int width,
                 const unsigned int height);
void zero_buffer(raytracer_context* rctx, cl_mem buf, size_t size);
size_t get_workgroup_size(raytracer_context* rctx, cl_kernel kernel);
#pragma once

struct _rt_ctx;

typedef struct path_raytracer_context
{
    struct _rt_ctx* rctx; //General Raytracer Context
    bool up_to_date;

    unsigned int num_samples;
    unsigned int current_sample;
    bool render_complete;
    int start_time;

    cl_mem cl_path_output_buffer;
    cl_mem cl_path_fresh_frame_buffer; //Only exists on GPU TODO: put in path tracer file.


} path_raytracer_context;

path_raytracer_context* init_path_raytracer_context(struct _rt_ctx*);

void path_raytracer_render(path_raytracer_context*);
void path_raytracer_prepass(path_raytracer_context*);
#pragma once
#include <alignment_util.h>

#include <stdint.h>
#include <parallel.h>
```

```
#include <CL/opencl.h>
#include <scene.h>
#include <irradiance_cache.h>

#define SS_RAYTRACER 0
#define PATH_RAYTRACER 1
#define SPLIT_PATH_RAYTRACER 2

//Cheap, quick, and dirty way of managing kernels.
#define KERNELS {"cast_ray_test", "generate_rays", "path_trace",        \
                 "buffer_average", "f_buffer_average",                  \
                 "f_buffer_to_byte_buffer",                             \
                 "ic_screen_textures", "generate_discontinuity",        \
                 "float_average", "mip_single_upsample", "mip_upsample",\
                 "mip_upsample_scaled", "mip_single_upsample_scaled",   \
                 "mip_reduce", "blit_float_to_output",                  \
                 "blit_float3_to_output", "kdtree_intersection",        \
                 "kdtree_test_draw", "segmented_path_trace",            \
                 "f_buffer_to_byte_buffer_avg", "segmented_path_trace_init", \
                 "kdtree_ray_draw", "xorshift_batch"}
#define NUM_KERNELS 23
#define RAY_CAST_KRNL_INDX 0
#define RAY_BUFFER_KRNL_INDX 1
#define PATH_TRACE_KRNL_INDX 2
#define BUFFER_AVG_KRNL_INDX 3
#define F_BUFFER_AVG_KRNL_INDX 4
#define F_BUF_TO_BYTE_BUF_KRNL_INDX 5
#define IC_SCREEN_TEX_KRNL_INDX 6
#define IC_GEN_DISC_KRNL_INDX 7
#define IC_FLOAT_AVG_KRNL_INDX 8
#define IC_MIP_S_UPSAMPLE_KRNL_INDX 9
#define IC_MIP_UPSAMPLE_KRNL_INDX 10
#define IC_MIP_UPSAMPLE_SCALED_KRNL_INDX 11
#define IC_MIP_S_UPSAMPLE_SCALED_KRNL_INDX 12
#define IC_MIP_REDUCE_KRNL_INDX 13
#define BLIT_FLOAT_OUTPUT_INDX 14
#define BLIT_FLOAT3_OUTPUT_INDX 15
#define KDTREE_INTERSECTION_INDX 16
#define KDTREE_TEST_DRAW_INDX 17
#define SEGMENTED_PATH_TRACE_INDX 18
#define F_BUF_TO_BYTE_BUF_AVG_KRNL_INDX 19
#define SEGMENTED_PATH_TRACE_INIT_INDX 20
#define KDTREE_RAY_DRAW_INDX 21
#define XORSHIFT_BATCH_INDX 22

typedef struct _rt_ctx raytracer_context;

typedef struct rt_vtable //NOTE: @REFACTOR not used anymore should delete
{
    bool up_to_date;
    void (*build)(void*);
    void (*pre_pass)(void*);
    void (*render_frame)(void*);
} rt_vtable;


struct _rt_ctx
{
    unsigned int width, height;

    float* ray_buffer;
    vec4*  path_output_buffer; //TODO: put in path tracer output
    uint32_t* output_buffer;
    //uint32_t* fresh_frame_buffer;

    scene* stat_scene;
    ic_context* ic_ctx;

    unsigned int block_size_y;
    unsigned int block_size_x;

    unsigned int event_stack[32];
    unsigned int event_position;

    //TODO: seperate into contexts for each integrator.
    //Path tracing only

    unsigned int num_samples;    //TODO: put in path tracer file.
    unsigned int current_sample; //TODO: put in path tracer file.
    bool render_complete;

    //CL
    rcl_ctx* rcl;
    rcl_program* program;

    cl_mem cl_ray_buffer;
    cl_mem cl_output_buffer;
```

```
    cl_mem cl_path_output_buffer; //TODO: put in path tracer file
    cl_mem cl_path_fresh_frame_buffer; //Only exists on GPU TODO: put in path tracer file.

};

raytracer_context* raytracer_init(unsigned int width, unsigned int height,
                                  uint32_t* output_buffer, rcl_ctx* ctx);

void raytracer_build(raytracer_context*);
void raytracer_prepass(raytracer_context*); //NOTE: I would't call it a prepass, its more like a build
void raytracer_render(raytracer_context*);
void raytracer_refined_render(raytracer_context*);
void _raytracer_gen_ray_buffer(raytracer_context*);
void _raytracer_path_trace(raytracer_context*, unsigned int);
void _raytracer_average_buffers(raytracer_context*, unsigned int); //NOTE: DEPRECATED
void _raytracer_push_path(raytracer_context*);
void _raytracer_cast_rays(raytracer_context*); //NOTE: DEPRECATED
#pragma once
#include <alignment_util.h>
#include <vec.h>
//typedef struct{} sphere;
//struct sphere;
//struct plane;
//struct kd_tree;

typedef struct _rt_ctx raytracer_context;

typedef W_ALIGN(16) struct
{
    vec4  colour;

    float reflectivity;

    //TODO: add more.
} U_ALIGN(16) material;



typedef W_ALIGN(32) struct
{
    mat4 model;

    vec4 max;
    vec4 min;

    int index_offset;
    int num_indices;

    int material_index;
} U_ALIGN(32) mesh;

typedef struct
{

    mat4 camera_world_matrix;

    //Materials
    material* materials;
    cl_mem cl_material_buffer;
    unsigned int num_materials;
    bool materials_changed;
    //Primatives

    //Spheres
    sphere* spheres;
    cl_mem cl_sphere_buffer;
    unsigned int num_spheres; //NOTE: must be constant.
    bool spheres_changed;
    //Planes
    plane* planes;
    cl_mem cl_plane_buffer;
    unsigned int num_planes; //NOTE: must be constant.
    bool planes_changed;

    //Meshes
    mesh* meshes; //All vertex data is stored contiguously
    cl_mem cl_mesh_buffer;
    unsigned int num_meshes;
    bool meshes_changed;

    //Trying to remember how I got all of the other structs to use typedefs...
    //kd_tree
    struct kd_tree* kdt;


    //NOTE: we could store vertices, normals, and texcoords contiguously as 1 buffer.
    vec3* mesh_verts;
```

```
    rcl_img_buf cl_mesh_vert_buffer;
    unsigned int num_mesh_verts; //NOTE: must be constant.

    vec3* mesh_nrmls;
    rcl_img_buf cl_mesh_nrml_buffer;
    unsigned int num_mesh_nrmls; //NOTE: must be constant.

    vec2* mesh_texcoords;
    rcl_img_buf cl_mesh_texcoord_buffer;
    unsigned int num_mesh_texcoords; //NOTE: must be constant.

    ivec3* mesh_indices;
    rcl_img_buf cl_mesh_index_buffer;
    unsigned int num_mesh_indices; //NOTE: must be constant.

} scene;


void scene_resource_push(raytracer_context*);
void scene_init_resources(raytracer_context*);
void scene_generate_resources(raytracer_context*); //k-d tree generation
#pragma once

struct _rt_ctx;


typedef struct spath_raytracer_context
{
    struct _rt_ctx* rctx; //General Raytracer Context
    bool up_to_date;

    unsigned int num_iterations;
    unsigned int current_iteration;
    bool render_complete;

    //unsigned int segment_width;
    //unsigned int segment_offset;

    unsigned int start_time;

    unsigned int* random_buffer;

    cl_mem cl_path_output_buffer;
    cl_mem cl_path_ray_origin_buffer; //Only exists on GPU
    cl_mem cl_path_collision_result_buffer; //Only exists on GPU
    cl_mem cl_spath_progress_buffer; //Only exists on GPU
    cl_mem cl_path_origin_collision_result_buffer; //Only exists on GPU

    cl_mem cl_random_buffer; //Only exists on GPU


    cl_mem cl_bad_api_design_buffer;


} spath_raytracer_context;

spath_raytracer_context* init_spath_raytracer_context(struct _rt_ctx*);

void spath_raytracer_render(spath_raytracer_context*);
//void ss_raytracer_build(ss_raytracer_context*);
void spath_raytracer_prepass(spath_raytracer_context*);
#pragma once

struct _rt_ctx;

typedef struct ss_raytracer_context
{
    struct _rt_ctx* rctx; //General Raytracer Context
    bool up_to_date;
} ss_raytracer_context;


//TODO: create function table;

rt_vtable get_ss_raytracer_vtable();

ss_raytracer_context* init_ss_raytracer_context(struct _rt_ctx*);

void ss_raytracer_render(ss_raytracer_context*);
//void ss_raytracer_build(ss_raytracer_context*);
void ss_raytracer_prepass(ss_raytracer_context*);
#pragma once

int startup();
void loop_exit();
void loop_pause();
#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();
737 void win32_loop();
738
739 void win32_update();
740
741 void win32_sleep(int);
742
743 void* win32_get_bitmap_memory();
744
745 int win32_get_time_mili();
746
747 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 #endif
779 //NOTE: osx.m is compiled seperatly and then linked at the end.
780
781 //#define _MEM_DEBUG //Enable verbose memory allocation, movement and freeing
782
```

```
#include <CL/opencl.h>

#include <debug.c>

#include <os_abs.c>
#include <startup.c>
#include <scene.c>
#include <geom.c>
#include <loader.c>
#include <parallel.c>
#include <ui.c>
#include <irradiance_cache.c>
#include <raytracer.c>
#include <ss_raytracer.c>
#include <path_raytracer.c>
#include <spath_raytracer.c>
#include <kdtree.c>
#ifdef _MEM_DEBUG
void* _debug_memcpy(void* dest, void* from, size_t size, int line, const char *func)
{
    printf("\n-");
    memcpy(dest, from, size);
    printf("- memcpy at %i, %s, %p[%li]\n\n", line, func, dest, size);
    fflush(stdout);
    return dest;
}
void* _debug_malloc(size_t size, int line, const char *func)
{
    printf("\n-");
    void *p = malloc(size);
    printf("- Allocation at %i, %s, %p[%li]\n\n", line, func, p, size);
    fflush(stdout);
    return p;
}

void _debug_free(void* ptr, int line, const char *func)
{
    printf("\n-");
    free(ptr);
    printf("- Free at %i, %s, %p\n\n", line, func, ptr);
    fflush(stdout);
}


#define malloc(X) _debug_malloc( X, __LINE__, __FUNCTION__)
#define free(X) _debug_free( X, __LINE__, __FUNCTION__)
#define memcpy(X, Y, Z) _debug_memcpy( X, Y, Z, __LINE__, __FUNCTION__)

#endif

#ifdef WIN32
#define DEBUG_BREAK __debugbreak
#define _FILE_SEP '\\'
#else
#define DEBUG_BREAK
#define _FILE_SEP '/'
#endif

#define __FILENAME__ (strrchr(__FILE__, _FILE_SEP) ? strrchr(__FILE__, _FILE_SEP) + 1 : __FILE__)


//TODO: replace all errors with this.
#define ASRT_CL(m)                                                                      \
    if(err!=CL_SUCCESS)                                                                 \
    {                                                                                   \
        fprintf(stderr, "ERROR: %s. (code: %i, line: %i, file:%s)\nPRESS ENTER TO EXIT\n", \
            m, err, __LINE__, __FILENAME__);                                            \
        fflush(stderr);                                                                 \
        while(1){char c; scanf("%c",&c); exit(1);}                                      \
    }
//DEBUG_BREAK();                                                    \
#include <geom.h>
#define DEBUG_PRINT_VEC3(n, v) printf(n ": (%f, %f, %f)\n", v[0], v[1], v[2])


bool solve_quadratic(float *a, float *b, float *c, float *x0, float *x1)
{
    float discr = (*b) * (*b) - 4 * (*a) * (*c);

    if (discr < 0) return false;
    else if (discr == 0) {
        (*x0) = (*x1) = - 0.5 * (*b) / (*a);
    }
    else {
        float q = (*b > 0) ?
            -0.5 * (*b + sqrt(discr)) :
            -0.5 * (*b - sqrt(discr));
```

```
        *x0 = q / *a;
        *x1 = *c / q;
    }

    return true;
}

float* matvec_mul(mat4 m, vec4 v)
{
    float* out_float = (float*)malloc(sizeof(vec4));

    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];
    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];
    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];
    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];

    return out_float;
}

void swap_float(float *f1, float *f2)
{
    float temp = *f2;
    *f2 = *f1;
    *f1 = temp;
}


inline void AABB_divide(AABB source, uint8_t k, float b, AABB* left, AABB* right)
{
    vec3 new_min, new_max;
    memcpy(new_min, source.min, sizeof(vec3));
    memcpy(new_max, source.max, sizeof(vec3));

    float wrld_split = source.min[k] + (source.max[k] - source.min[k]) * b;
    new_min[k] = new_max[k] = wrld_split;

    memcpy(left->min,  source.min, sizeof(vec3));
    memcpy(left->max,  new_max,     sizeof(vec3));
    memcpy(right->min, new_min,     sizeof(vec3));
    memcpy(right->max, source.max, sizeof(vec3));
}


inline void AABB_divide_world(AABB source, uint8_t k, float world_b, AABB* left, AABB* right)
{
    vec3 new_min, new_max;
    memcpy(new_min, source.min, sizeof(vec3));
    memcpy(new_max, source.max, sizeof(vec3));

    new_min[k] = new_max[k] = world_b;

    memcpy(left->min,  source.min, sizeof(vec3));
    memcpy(left->max,  new_max,    sizeof(vec3));
    memcpy(right->min, new_min,    sizeof(vec3));
    memcpy(right->max, source.max, sizeof(vec3));
}


inline float AABB_surface_area(AABB source)
{
    vec3 diff;

    xv_sub(diff, source.max, source.min, 3);

    return (diff[0]*diff[1]*2 +
            diff[1]*diff[2]*2 +
            diff[0]*diff[2]*2);
}

inline void AABB_clip(AABB* result, AABB* target, AABB* container)
{
    memcpy(result,  target, sizeof(AABB));

    for (int i = 0; i < 3; i++)
    {
        if(result->min[i] < container->min[i])
            result->min[i] = container->min[i];
        if(result->max[i] > container->max[i])
            result->max[i] = container->max[i];
    }
}

inline void AABB_construct_from_triangle(AABB* result, ivec3* indices, vec3* vertices)
{
    for(int k = 0; k < 3; k++)
    {
        result->min[k] =  1000000;
```

```
957         result->max[k] = -1000000;
958     }
959 
960     for(int i = 0; i < 3; i++)
961     {
962         float* vertex = vertices[indices[i][0]];
963 
964         for(int k = 0; k < 3; k++)
965         {
966             if(vertex[k] < result->min[k])
967                 result->min[k] = vertex[k];
968 
969             if(vertex[k] > result->max[k])
970                 result->max[k] = vertex[k];
971         }
972     }
973 }
974 
975 inline void AABB_construct_from_vertices(AABB* result, vec3* vertices,
976                                          unsigned int num_vertices)
977 {
978     for(int k = 0; k < 3; k++)
979     {
980         result->min[k] =  1000000;
981         result->max[k] = -1000000;
982     }
983     for(int i = 0; i < num_vertices; i++)
984     {
985         for(int k = 0; k < 3; k++)
986         {
987             if(vertices[i][k] < result->min[k])
988                 result->min[k] = vertices[i][k];
989 
990             if(vertices[i][k] > result->max[k])
991                 result->max[k] = vertices[i][k];
992         }
993     }
994 }
995 
996 inline bool AABB_is_planar(AABB* source, uint8_t k)
997 {
998     if(source->max[k]-source->min[k] == 0.0f) //TODO: use epsilon instead of 0
999         return true;
1000     return false;
1001 }
1002 
1003 inline float AABB_ilerp(AABB source, uint8_t k, float world_b)
1004 {
1005     return (world_b - source.min[k]) / (source.max[k] - source.min[k]);
1006 }
1007 
1008 inline float does_collide_sphere(sphere s, ray r)
1009 {
1010     float t0, t1; // solutions for t if the ray intersects
1011 
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 inline float does_collide_plane(plane p, ray r)
1033 {
1034     float denom = xv3_dot(r.dir, p.norm);
1035     if (denom > 1e-6)
1036     {
1037         vec3 l;
1038         xv_sub(l, p.pos, r.orig, 3);
1039         float t = xv3_dot(l, p.norm) / denom;
1040         if (t >= 0)
1041             return -1.0;
1042         return t;
1043     }
```

```
    return -1.0;
}

ray generate_ray(int x, int y, int width, int height, float fov)
{
    ray r;

    //Simplified
    /* float ndc_x =((float)x+0.5)/width; */
    /* float ndc_y =((float)x+0.5)/height; */
    /* float screen_x = 2 * ndc_x - 1; */
    /* float screen_y = 1 - 2 * ndc_y; */
    /* float aspect_ratio = width/height; */
    /* float cam_x =(2*screen_x-1) * tan(fov / 2 * M_PI / 180) * aspect_ratio; */
    /* float cam_y = (1-2*screen_y) * tan(fov / 2 * M_PI / 180); */

    float aspect_ratio = width / (float)height; // assuming width > height
    float cam_x = (2 * (((float)x + 0.5) / width) - 1) * tan(fov / 2 * M_PI / 180) * aspect_ratio;
    float cam_y = (1 - 2 * (((float)y + 0.5) / height)) * tan(fov / 2 * M_PI / 180);


    xv3_zero(r.orig);
    vec3 v1 = {cam_x, cam_y, -1};
    xv_sub(r.dir, v1, r.orig, 3);
    xv_normeq(r.dir, 3);

    return r;
}
/****************************************/
/* NOTE: Irradiance Caching is Incomplete */
/****************************************/

#include <irradiance_cache.h>
#include <raytracer.h>
#include <parallel.h>


void ic_init(raytracer_context* rctx)
{
    rctx->ic_ctx->cl_standard_format.image_channel_order     = CL_RGBA;
    rctx->ic_ctx->cl_standard_format.image_channel_data_type = CL_FLOAT;

    rctx->ic_ctx->cl_standard_descriptor.image_type = CL_MEM_OBJECT_IMAGE2D;
    rctx->ic_ctx->cl_standard_descriptor.image_width = rctx->width;
    rctx->ic_ctx->cl_standard_descriptor.image_height = rctx->height;
    rctx->ic_ctx->cl_standard_descriptor.image_depth  = 0;
    rctx->ic_ctx->cl_standard_descriptor.image_array_size  = 0;
    rctx->ic_ctx->cl_standard_descriptor.image_row_pitch  = 0;
    rctx->ic_ctx->cl_standard_descriptor.num_mip_levels = 0;
    rctx->ic_ctx->cl_standard_descriptor.num_samples = 0;
    rctx->ic_ctx->cl_standard_descriptor.buffer = NULL;

    rctx->ic_ctx->octree.node_count = 1; //root
    //TODO: add as parameter
    rctx->ic_ctx->octree.max_depth = 8;  //arbitrary
    rctx->ic_ctx->octree.width     = 15; //arbitrary

    rctx->ic_ctx->octree.root = (ic_octree_node*) malloc(sizeof(ic_octree_node));
    rctx->ic_ctx->octree.root->min[0] = (float)-rctx->ic_ctx->octree.width;
    rctx->ic_ctx->octree.root->min[1] = (float)-rctx->ic_ctx->octree.width;
    rctx->ic_ctx->octree.root->min[2] = (float)-rctx->ic_ctx->octree.width;
    rctx->ic_ctx->octree.root->max[0] = (float) rctx->ic_ctx->octree.width;
    rctx->ic_ctx->octree.root->max[1] = (float) rctx->ic_ctx->octree.width;
    rctx->ic_ctx->octree.root->max[2] = (float) rctx->ic_ctx->octree.width;
    rctx->ic_ctx->octree.root->leaf = false;
    rctx->ic_ctx->octree.root->active = false;
}

void ic_octree_init_leaf(ic_octree_node* node, ic_octree_node* parent, unsigned int i)
{
    float xhalf = (parent->max[0]-parent->min[0])/2;
    float yhalf = (parent->max[1]-parent->min[1])/2;
    float zhalf = (parent->max[2]-parent->min[2])/2;
    node->active = false;

    node->leaf = true;
    for(int i = 0; i < 8; i++)
        node->data.branch.children[i] = NULL;
    node->min[0] = parent->min[0] + ( (i&4) ? xhalf : 0);
    node->min[1] = parent->min[1] + ( (i&2) ? yhalf : 0);
    node->min[2] = parent->min[2] + ( (i&1) ? zhalf : 0);
    node->max[0] = parent->max[0] - (!(i&4) ? xhalf : 0);
    node->max[1] = parent->max[1] - (!(i&2) ? yhalf : 0);
    node->max[2] = parent->max[2] - (!(i&1) ? zhalf : 0);
}

```

```
1130 void ic_octree_make_branch(ic_octree* tree, ic_octree_node* node)
1131 {
1132
1133     node->leaf = false;
1134     for(int i = 0; i < 8; i++)
1135     {
1136         node->data.branch.children[i] = malloc(sizeof(ic_octree_node));
1137         ic_octree_init_leaf(node->data.branch.children[i], node, i);
1138         tree->node_count++;
1139     }
1140 }
1141
1142 //TODO: test if points are the same
1143 void _ic_octree_rec_resolve(ic_context* ictx, ic_octree_node* leaf, unsigned int node1, unsigned int node2,
1144                             unsigned int depth)
1145 {
1146     if(depth > ictx->octree.max_depth)
1147     {
1148         //TODO: just group buffers together
1149         printf("ERROR: octree reached max depth when trying to resolve collision. (INCOMPLETE)\n");
1150         exit(1);
1151     }
1152     vec3 mid_point;
1153     xv_sub(mid_point, leaf->max, leaf->min, 3);
1154     xv_divieq(mid_point, 2, 3);
1155     unsigned int i1 =
1156         ((mid_point[0]<ictx->ir_buf[node1].point[0])<<2) |
1157         ((mid_point[1]<ictx->ir_buf[node1].point[1])<<1) |
1158         ((mid_point[2]<ictx->ir_buf[node1].point[2]));
1159     unsigned int i2 =
1160         ((mid_point[0]<ictx->ir_buf[node2].point[0])<<2) |
1161         ((mid_point[1]<ictx->ir_buf[node2].point[1])<<1) |
1162         ((mid_point[2]<ictx->ir_buf[node2].point[2]));
1163     ic_octree_make_branch(&ictx->octree, leaf);
1164     if(i1==i2)
1165         _ic_octree_rec_resolve(ictx, leaf->data.branch.children[i1], node1, node2, depth+1);
1166     else
1167     { //happiness
1168         leaf->data.branch.children[i1]->data.leaf.buffer_offset = node1;
1169         leaf->data.branch.children[i1]->data.leaf.num_elems = 1;
1170         leaf->data.branch.children[i2]->data.leaf.buffer_offset = node2;
1171         leaf->data.branch.children[i2]->data.leaf.num_elems = 1;
1172     }
1173 }
1174
1175 void _ic_octree_rec_insert(ic_context* ictx, ic_octree_node* node, unsigned int v_ptr, unsigned int depth)
1176 {
1177     if(node->leaf && !node->active)
1178     {
1179         node->active = true;
1180         node->data.leaf.buffer_offset = v_ptr;
1181         node->data.leaf.num_elems     = 1; //TODO: add suport for more than 1.
1182         return;
1183     }
1184     else if(node->leaf)
1185     {
1186         //resolve
1187         _ic_octree_rec_resolve(ictx, node, v_ptr, node->data.leaf.buffer_offset, depth+1);
1188     }
1189     else
1190     {
1191         ic_octree_node* new_node = node->data.branch.children[
1192             ((ictx->ir_buf[node->data.leaf.buffer_offset].point[0]<ictx->ir_buf[v_ptr].point[0])<<2) |
1193                 ((ictx->ir_buf[node->data.leaf.buffer_offset].point[1]<ictx->ir_buf[v_ptr].point[1])<<1) |
1194                 ((ictx->ir_buf[node->data.leaf.buffer_offset].point[2]<ictx->ir_buf[v_ptr].point[2]))];
1195         _ic_octree_rec_insert(ictx, new_node, v_ptr, depth+1);
1196     }
1197 }
1198
1199 void ic_octree_insert(ic_context* ictx, vec3 point, vec3 normal)
1200 {
1201     if(ictx->ir_buf_current_offset==ictx->ir_buf_size) //TODO: dynamically resize or do something else
1202     {
1203         printf("ERROR: irradiance buffer is full!\n");
1204         exit(1);
1205     }
1206     ic_ir_value irradiance_value; //TODO: EVALUATE THIS
1207     irradiance_value.rad = 0.f; //Gets rid of error, this doesn't work anyways so its good enough.
1208     ictx->ir_buf[ictx->ir_buf_current_offset++] = irradiance_value;
1209     _ic_octree_rec_insert(ictx, ictx->octree.root, ictx->ir_buf_current_offset, 0);
1210 }
1211
1212 //NOTE: outBuffer is only bools but using char for safety accross compilers.
1213 //      Also assuming that buf is grayscale
1214 void dither(float* buf, const int width, const int height)
1215 {
1216     for(int y = 0; y < height; y++ )
```

```
    {
        for(int x = 0; x < width; x++ )
        {
            float oldpixel  = buf[x+y*width];
            float newpixel  = oldpixel>0.5f ? 1 : 0;
            buf[x+y*width]  = newpixel;
            float err = oldpixel - newpixel;

            if( (x != (width-1)) && (x != 0) && (y != (height-1)) )
            {
                buf[(x+1)+(y  )*width] = buf[(x+1)+(y  )*width] + err  * (7.f / 16.f);
                buf[(x-1)+(y+1)*width] = buf[(x-1)+(y+1)*width] + err  * (3.f / 16.f);
                buf[(x  )+(y+1)*width] = buf[(x  )+(y+1)*width] + err  * (5.f / 16.f);
                buf[(x+1)+(y+1)*width] = buf[(x+1)+(y+1)*width] + err  * (1.f / 16.f);
            }
        }
    }
}


void get_geom_maps(raytracer_context* rctx, cl_mem positions, cl_mem normals)
{
    int err;

    cl_kernel kernel = rctx->program->raw_kernels[IC_SCREEN_TEX_KRNL_INDX];

    float zeroed[] = {0., 0., 0., 1.};
    float* result = matvec_mul(rctx->stat_scene->camera_world_matrix, zeroed);

    //SO MANY ARGUEMENTS
    clSetKernelArg(kernel, 0, sizeof(cl_mem),  &positions);
    clSetKernelArg(kernel, 1, sizeof(cl_mem),  &normals);
    clSetKernelArg(kernel, 2, sizeof(int),     &rctx->width);
    clSetKernelArg(kernel, 3, sizeof(int),     &rctx->height);
    clSetKernelArg(kernel, 4, sizeof(cl_mem),  &rctx->cl_ray_buffer);
    clSetKernelArg(kernel, 5, sizeof(vec4),    result);
    clSetKernelArg(kernel, 6, sizeof(cl_mem),  &rctx->stat_scene->cl_material_buffer);
    clSetKernelArg(kernel, 7, sizeof(cl_mem),  &rctx->stat_scene->cl_sphere_buffer);
    clSetKernelArg(kernel, 8, sizeof(cl_mem),  &rctx->stat_scene->cl_plane_buffer);
    clSetKernelArg(kernel, 9, sizeof(cl_mem),  &rctx->stat_scene->cl_mesh_buffer);
    clSetKernelArg(kernel, 10, sizeof(cl_mem), &rctx->stat_scene->cl_mesh_index_buffer);
    clSetKernelArg(kernel, 11, sizeof(cl_mem), &rctx->stat_scene->cl_mesh_vert_buffer);
    clSetKernelArg(kernel, 12, sizeof(cl_mem), &rctx->stat_scene->cl_mesh_nrml_buffer);

    size_t global = rctx->width*rctx->height;
    size_t local = 0;
    err = clGetKernelWorkGroupInfo(kernel, rctx->rcl->device_id, CL_KERNEL_WORK_GROUP_SIZE,
                                   sizeof(local), &local, NULL);
    ASRT_CL("Failed to Retrieve Kernel Work Group Info");

    err = clEnqueueNDRangeKernel(rctx->rcl->commands, kernel, 1, NULL, &global,
                                 NULL, 0, NULL, NULL);
    ASRT_CL("Failed to Enqueue kernel IC_SCREEN_TEX");

    //Wait for completion
    err = clFinish(rctx->rcl->commands);
    ASRT_CL("Something happened while waiting for kernel to finish");
}

void gen_mipmap_chain_gb(raytracer_context* rctx, cl_mem texture,
                         ic_mipmap_gb* mipmaps, int num_mipmaps)
{
    int err;
    unsigned int width  = rctx->width;
    unsigned int height = rctx->height;
    cl_kernel kernel = rctx->program->raw_kernels[IC_MIP_REDUCE_KRNL_INDX];
    for(int i = 0; i < num_mipmaps; i++)
    {
        mipmaps[i].width  = width;
        mipmaps[i].height = height;

        if(i==0)
        {
            mipmaps[0].cl_image_ref = texture;

            height /= 2;
            width /= 2;
            continue;
        }

        clSetKernelArg(kernel, 0, sizeof(cl_mem), &mipmaps[i-1].cl_image_ref);
        clSetKernelArg(kernel, 1, sizeof(cl_mem), &mipmaps[i].cl_image_ref);
        clSetKernelArg(kernel, 2, sizeof(int),    &width);
        clSetKernelArg(kernel, 3, sizeof(int),    &height);

        size_t global = width*height;
        size_t local = get_workgroup_size(rctx, kernel);
```

```
1304
1305        err = clEnqueueNDRangeKernel(rctx->rcl->commands, kernel, 1,
1306                                     NULL, &global, NULL, 0, NULL, NULL);
1307        ASRT_CL("Failed to Enqueue kernel IC_MIP_REDUCE");
1308
1309        height /= 2;
1310        width /= 2;
1311        //Wait for completion before doing next mip
1312        err = clFinish(rctx->rcl->commands);
1313        ASRT_CL("Something happened while waiting for kernel to finish");
1314    }
1315 }
1316
1317 void upsample_mipmaps_f(raytracer_context* rctx, cl_mem texture,
1318                         ic_mipmap_f* mipmaps, int num_mipmaps)
1319 {
1320     int err;
1321
1322     cl_mem* full_maps = (cl_mem*) alloca(sizeof(cl_mem)*num_mipmaps);
1323     for(int i = 1; i < num_mipmaps; i++)
1324     {
1325         full_maps[i] = gen_grayscale_buffer(rctx, 0, 0);
1326     }
1327     full_maps[0] = texture;
1328     { //Upsample
1329         for(int i = 0; i < num_mipmaps; i++) //First one is already at proper resolution
1330         {
1331             cl_kernel kernel = rctx->program->raw_kernels[IC_MIP_S_UPSAMPLE_SCALED_KRNL_INDX];
1332
1333             clSetKernelArg(kernel, 0, sizeof(cl_mem), &mipmaps[i].cl_image_ref);
1334             clSetKernelArg(kernel, 1, sizeof(cl_mem), &full_maps[i]); //NOTE: need to generate this for the function
1335             clSetKernelArg(kernel, 2, sizeof(int),    &i);
1336             clSetKernelArg(kernel, 3, sizeof(int),    &rctx->width);
1337             clSetKernelArg(kernel, 4, sizeof(int),    &rctx->height);
1338
1339             size_t global = rctx->width*rctx->height;
1340             size_t local = get_workgroup_size(rctx, kernel);
1341
1342             err = clEnqueueNDRangeKernel(rctx->rcl->commands, kernel, 1,
1343                                          NULL, &global, NULL, 0, NULL, NULL);
1344             ASRT_CL("Failed to Enqueue kernel IC_MIP_S_UPSAMPLE_SCALED");
1345
1346         }
1347         err = clFinish(rctx->rcl->commands);
1348         ASRT_CL("Something happened while waiting for kernel to finish");
1349     }
1350     printf("Upsampled Discontinuity Mipmaps\nAveraging Upsampled Discontinuity Mipmaps\n");
1351
1352     { //Average
1353         int total = num_mipmaps;
1354         for(int i = 0; i < num_mipmaps; i++) //First one is already at proper resolution
1355         {
1356             cl_kernel kernel = rctx->program->raw_kernels[IC_FLOAT_AVG_KRNL_INDX];
1357
1358             clSetKernelArg(kernel, 0, sizeof(cl_mem), &full_maps[i]);
1359             clSetKernelArg(kernel, 1, sizeof(cl_mem), &texture);
1360             clSetKernelArg(kernel, 2, sizeof(int),    &rctx->width);
1361             clSetKernelArg(kernel, 3, sizeof(int),    &rctx->height);
1362             clSetKernelArg(kernel, 4, sizeof(int),    &total);
1363
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 void gen_discontinuity_maps(raytracer_context* rctx, ic_mipmap_gb* pos_mipmaps,
1386                             ic_mipmap_gb* nrm_mipmaps, ic_mipmap_f* disc_mipmaps,
1387                             int num_mipmaps)
1388 {
1389     int err;
1390     //TODO: tune k and intensity
```

```
    const float k = 1.6f;
    const float intensity = 0.02f;
    for(int i = 0; i < num_mipmaps; i++)
    {
        cl_kernel kernel = rctx->program->raw_kernels[IC_GEN_DISC_KRNL_INDX];
        disc_mipmaps[i].width  = pos_mipmaps[i].width;
        disc_mipmaps[i].height = pos_mipmaps[i].height;

        clSetKernelArg(kernel, 0, sizeof(cl_mem), &pos_mipmaps[i].cl_image_ref);


        clSetKernelArg(kernel, 1, sizeof(cl_mem), &nrm_mipmaps[i].cl_image_ref);
        clSetKernelArg(kernel, 2, sizeof(cl_mem), &disc_mipmaps[i].cl_image_ref);
        clSetKernelArg(kernel, 3, sizeof(float),  &k);
        clSetKernelArg(kernel, 4, sizeof(float),  &intensity);
        clSetKernelArg(kernel, 5, sizeof(int),    &pos_mipmaps[i].width);
        clSetKernelArg(kernel, 6, sizeof(int),    &pos_mipmaps[i].height);

        size_t global = pos_mipmaps[i].width*pos_mipmaps[i].height;
        size_t local = get_workgroup_size(rctx, kernel);

        err = clEnqueueNDRangeKernel(rctx->rcl->commands, kernel, 1,
                                     NULL, &global, NULL, 0, NULL, NULL);
        ASRT_CL("Failed to Enqueue kernel IC_GEN_DISC");

    }
    err = clFinish(rctx->rcl->commands);
    ASRT_CL("Something happened while waiting for kernel to finish");
}

void ic_screenspace(raytracer_context* rctx)
{
    int err;


    vec4*   pos_tex = (vec4*) malloc(rctx->width*rctx->height*sizeof(vec4));
    vec4*   nrm_tex = (vec4*) malloc(rctx->width*rctx->height*sizeof(vec4));
    float*  c_fin_disc_map = (float*) malloc(rctx->width*rctx->height*sizeof(float));

    ic_mipmap_gb pos_mipmaps [NUM_MIPMAPS]; //A lot of buffers
    ic_mipmap_gb nrm_mipmaps [NUM_MIPMAPS];
    ic_mipmap_f  disc_mipmaps[NUM_MIPMAPS];
    cl_mem       fin_disc_map;
    //OpenCL
    cl_mem cl_pos_tex;
    cl_mem cl_nrm_tex;
    cl_image_desc cl_mipmap_descriptor = rctx->ic_ctx->cl_standard_descriptor;

    { //OpenCL Init
        cl_pos_tex = gen_rgb_image(rctx, 0,0);
        cl_nrm_tex = gen_rgb_image(rctx, 0,0);

        fin_disc_map = gen_grayscale_buffer(rctx, 0,0);
        zero_buffer_img(rctx, fin_disc_map, sizeof(float), 0, 0);


        unsigned int width  = rctx->width,
        height = rctx->height;
        for(int i = 0; i < NUM_MIPMAPS; i++)
        {
            if(i!=0)
            {
                pos_mipmaps[i].cl_image_ref = gen_rgb_image(rctx, width, height);
                nrm_mipmaps[i].cl_image_ref = gen_rgb_image(rctx, width, height);
            }
            disc_mipmaps[i].cl_image_ref = gen_grayscale_buffer(rctx, width, height);

            width /= 2;
            height /= 2;
        }
    }
    printf("Initialised Irradiance Cache Screenspace Buffers\nGetting Screenspace Geometry Data\n");
    get_geom_maps(rctx, cl_pos_tex, cl_nrm_tex);
    printf("Got Screenspace Geometry Data\nGenerating MipMaps\n");
    gen_mipmap_chain_gb(rctx, cl_pos_tex,
                        pos_mipmaps, NUM_MIPMAPS);
    gen_mipmap_chain_gb(rctx, cl_nrm_tex,
                        nrm_mipmaps, NUM_MIPMAPS);
    printf("Generated MipMaps\nGenerating Discontinuity Map for each Mip\n");
    gen_discontinuity_maps(rctx, pos_mipmaps, nrm_mipmaps, disc_mipmaps, NUM_MIPMAPS);
    printf("Generated Discontinuity Map for each Mip\nUpsampling Discontinuity Mipmaps\n");
    upsample_mipmaps_f(rctx, fin_disc_map, disc_mipmaps, NUM_MIPMAPS);
    printf("Averaged Upsampled Discontinuity Mipmaps\nRetrieving Discontinuity Data\n");
    retrieve_buf(rctx, fin_disc_map, c_fin_disc_map,
                 rctx->width*rctx->height*sizeof(float));
    retrieve_image(rctx, cl_pos_tex, pos_tex, 0, 0);
    retrieve_image(rctx, cl_pos_tex, pos_tex, 0, 0);
```

```
1478
1479     printf("Retrieved Discontinuity Data\nDithering Discontinuity Map\n");
1480     //NOTE: read buffer is blocking so we don't need clFinish
1481     dither(c_fin_disc_map, rctx->width, rctx->height);
1482     err = clEnqueueWriteBuffer(rctx->rcl->commands, fin_disc_map,
1483                                CL_TRUE, 0,
1484                                rctx->width*rctx->height*sizeof(float),
1485                                c_fin_disc_map, 0, 0, NULL);
1486     ASRT_CL("Failed to write dithered discontinuity map");
1487
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 #include <kdtree.h>
1517 #include <scene.h>
1518
1519 #define KDTREE_EPSILON 0.001f
1520
1521 #define KDTREE_BOTH  0
1522 #define KDTREE_LEFT  1
1523 #define KDTREE_RIGHT 2
1524
1525 #define KDTREE_END   0
1526 #define KDTREE_PLANAR 1
1527 #define KDTREE_START  2
1528
1529 //Literally an index buffer to the index buffer
1530 typedef struct kd_tree_event
1531 {
1532     unsigned int tri_index_offset;
1533     float   b;
1534     uint8_t k;
1535     uint8_t type;
1536 } kd_tree_event;
1537
1538 typedef struct kd_tree_sah_results
1539 {
1540     float cost;
1541     uint8_t side; //1 left, 2 right
1542 } kd_tree_sah_results;
1543
1544 kd_tree_sah_results kd_tree_sah_results_c(float cost, uint8_t side)
1545 {
1546     kd_tree_sah_results r;
1547     r.cost = cost;
1548     r.side = side;
1549     return r;
1550 }
1551
1552 typedef struct kd_tree_find_plane_results
1553 {
1554     kd_tree_event p;
1555     unsigned int NL;
1556     unsigned int NR;
1557     unsigned int NP;
1558     uint8_t side;
1559     float cost;
1560
1561 } kd_tree_find_plane_results;
1562
1563
1564 bool kd_tree_event_lt(kd_tree_event* left, kd_tree_event* right)
```

```
{
    return
        (left->b <  right->b)                           ||
        (left->b == right->b && left->type < right->type) ||
        (left->k >  right->k);
}

typedef struct kd_tree_event_buffer
{
    kd_tree_event* events;
    unsigned int  num_events;

} kd_tree_event_buffer;



//Optional Lambda
float kd_tree_lambda(int NL, int NR, float PL, float PR)
{
    if( (NL == 0 || NR == 0) && !(PL == 1.0f || PR == 1.0f) ) //TODO: be less exact for pl pr check, add epsilon
        return 0.8f;
    return 1.0f;
}

//Cost function
float kd_tree_C(float PL, float PR, uint32_t NL, uint32_t NR)
{
    return kd_tree_lambda(NL, NR, PL, PR) *(KDTREE_KT + KDTREE_KI*(PL*NL + PR*NR));
}

kd_tree_sah_results kd_tree_SAH(uint8_t k, float b, AABB V, int NL, int NR, int NP)
{
    AABB VL;
    AABB VR;
    AABB_divide(V, k, b, &VL, &VR);
    float PL = AABB_surface_area(VL) / AABB_surface_area(V);
    float PR = AABB_surface_area(VR) / AABB_surface_area(V);

    if (PL >= 1-KDTREE_EPSILON || PR >= 1-KDTREE_EPSILON) //NOTE: doesn't look like it but potential source of issues
        return kd_tree_sah_results_c(1000000000.0f, 0);

    float CPL = kd_tree_C(PL, PR, NL+NP, NR);
    float CPR = kd_tree_C(PL, PR, NL, NR+NP);


    if(CPL < CPR)
        return kd_tree_sah_results_c(CPL, KDTREE_LEFT);
    else
        return kd_tree_sah_results_c(CPR, KDTREE_RIGHT);
}


kd_tree_event_buffer kd_tree_merge_event_buffers(kd_tree_event_buffer buf1, kd_tree_event_buffer buf2)
{
    //buffer 1 is guarenteed to be to the direct left of buffer 2
    kd_tree_event_buffer event_out;
    event_out.num_events = buf1.num_events + buf2.num_events;

    event_out.events = (kd_tree_event*) malloc(sizeof(kd_tree_event) * event_out.num_events);


    uint32_t buf1_i, buf2_i, eo_i;
    buf1_i = buf2_i = eo_i = 0;

    while(buf1_i != buf1.num_events || buf2_i != buf2.num_events)
    {
        if(buf1_i == buf1.num_events)
        {
            event_out.events[eo_i++] = buf2.events[buf2_i++];
            continue;
        }

        if(buf2_i == buf2.num_events)
        {
            event_out.events[eo_i++] = buf1.events[buf1_i++];
            continue;
        }

        if( kd_tree_event_lt(buf1.events+buf1_i, buf2.events+buf2_i) )
            event_out.events[eo_i++] = buf1.events[buf1_i++];
        else
            event_out.events[eo_i++] = buf2.events[buf2_i++];
    }
    assert(eo_i == event_out.num_events);
    memcpy(buf1.events, event_out.events, sizeof(kd_tree_event) * event_out.num_events);
    free(event_out.events);
    event_out.events = buf1.events;
```

```
1652
1653     return event_out;
1654 }
1655
1656 kd_tree_event_buffer kd_tree_mergesort_event_buffer(kd_tree_event_buffer buf)
1657 {
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 kd_tree* kd_tree_init()
1679 {
1680     kd_tree* tree = malloc(sizeof(kd_tree));
1681     tree->root = NULL;
1682     //Defaults
1683     tree->k    = 3;
1684     tree->max_recurse = 50;
1685     tree->tri_for_leaf_threshold = 2;
1686     tree->num_nodes_total     = 0;
1687     tree->num_tris_padded     = 0;
1688     tree->num_traversal_nodes = 0;
1689     tree->num_leaves          = 0;
1690     tree->num_indices_total   = 0;
1691     tree->buffer_size         = 0;
1692     tree->buffer              = NULL;
1693     tree->cl_kd_tree_buffer   = NULL;
1694     xv3_zero(tree->bounds.min);
1695     xv3_zero(tree->bounds.max);
1696     return tree;
1697 }
1698
1699 kd_tree_node* kd_tree_node_init()
1700 {
1701     kd_tree_node* node = malloc(sizeof(kd_tree_node));
1702     node->k = 0;
1703     node->b = 0.5f; //generic default, shouldn't matter with SAH anyways
1704
1705     node->left  = NULL;
1706     node->right = NULL;
1707
1708     return node;
1709 }
1710
1711 bool kd_tree_node_is_leaf(kd_tree_node* node)
1712 {
1713     if(node->left == NULL || node->right == NULL)
1714     {
1715         assert(node->left == NULL && node->right == NULL);
1716         return true;
1717     }
1718
1719     return false;
1720 }
1721
1722
1723
1724 kd_tree_find_plane_results kd_tree_find_plane(kd_tree* tree, AABB V,
1725                                               kd_tree_triangle_buffer tri_buf)
1726 {
1727     float     best_cost = INFINITY;
1728      kd_tree_find_plane_results result;
1729
1730
1731     for(int k = 0; k < tree->k; k++)
1732     {
1733         kd_tree_event_buffer event_buf = {NULL, 0}; //gets rid of an initialization warning I guess?
1734         {// Generate events
1735             //Divide by three because we only want tris
1736             event_buf.num_events = tri_buf.num_triangles*2;
1737
1738             event_buf.events = malloc(sizeof(kd_tree_event)*event_buf.num_events);
```

```
1739            unsigned int j = 0;
1740            for (int i = 0; i < tri_buf.num_triangles; i++)
1741            {
1742                AABB tv, B;
1743                AABB_construct_from_triangle(&tv, tree->s->mesh_indices+tri_buf.triangle_buffer[i],
1744                                             tree->s->mesh_verts);
1745                AABB_clip(&B, &tv, &V);
1746                if(AABB_is_planar(&B, k))
1747                {
1748                    event_buf.events[j++] = (kd_tree_event) {i*3, B.min[k], k, KDTREE_PLANAR};
1749                }
1750                else
1751                {
1752                    event_buf.events[j++] = (kd_tree_event) {i*3, B.min[k], k, KDTREE_START};
1753                    event_buf.events[j++] = (kd_tree_event) {i*3, B.max[k], k, KDTREE_END};
1754                }
1755            }
1756            event_buf.num_events = j;
1757
1758            int last_num_events = event_buf.num_events;
1759            event_buf = kd_tree_mergesort_event_buffer(event_buf);
1760            assert(event_buf.num_events == last_num_events);
1761        }
1762
1763        int NL, NP, NR;
1764        NL = NP = 0;
1765        NR = tri_buf.num_triangles;
1766        for (int i = 0; i < event_buf.num_events;)
1767        {
1768            kd_tree_event p = event_buf.events[i];
1769            int Ps, Pe, Pp;
1770            Ps = Pe = Pp = 0;
1771            while(i < event_buf.num_events && event_buf.events[i].b == p.b && event_buf.events[i].type == KDTREE_END)
1772            {
1773                Pe += 1;
1774                i++;
1775            }
1776            while(i < event_buf.num_events && event_buf.events[i].b == p.b && event_buf.events[i].type == KDTREE_PLANAR)
1777            {
1778                Pp += 1;
1779                i++;
1780            }
1781            while(i < event_buf.num_events && event_buf.events[i].b == p.b && event_buf.events[i].type == KDTREE_START)
1782            {
1783                Ps += 1;
1784                i++;
1785            }
1786
1787            NP =  Pp;
1788            NR -= Pp;
1789            NR -= Pe;
1790
1791            kd_tree_sah_results results = kd_tree_SAH(k, AABB_ilerp(V, k, p.b), V, NL, NR, NP);
1792
1793            if (results.cost < best_cost)
1794            {
1795                best_cost = results.cost;
1796                result.p = p;
1797                result.side = results.side;
1798
1799                result.NL = NL;
1800                result.NR = NR;
1801                result.NP = NP;
1802                result.cost = results.cost; //just the min cost, really confusing syntax
1803            }
1804
1805            NL += Ps;
1806            NL += NP;
1807            NP =  0;
1808
1809        }
1810        free(event_buf.events);
1811    }
1812
1813    return result;
1814 }
1815
1816 void kd_tree_classify(kd_tree* tree, kd_tree_triangle_buffer tri_buf,
1817                       kd_tree_find_plane_results results,
1818                       kd_tree_triangle_buffer* TL_out, kd_tree_triangle_buffer* TR_out)
1819 {
1820     kd_tree_triangle_buffer TL;
1821     kd_tree_triangle_buffer TR;
1822     TL.num_triangles   = results.NL + (results.side == KDTREE_LEFT ? results.NP : 0);
1823     TL.triangle_buffer = (unsigned int*) malloc(sizeof(unsigned int)*TL.num_triangles); //NOTE: memory leak, never freed.
1824     TR.num_triangles   = results.NR + (results.side == KDTREE_RIGHT ? results.NP : 0);
1825     TR.triangle_buffer = (unsigned int*) malloc(sizeof(unsigned int)*TR.num_triangles);
```

```
1826     unsigned int TLI, TRI;
1827     TLI = TRI = 0;
1828     for(int i = 0; i < tri_buf.num_triangles; i++)
1829     {
1830         bool isLeft = false;
1831         bool isRight = false;
1832         for(int j = 0; j < 3; j++)
1833         {
1834 
1835             float p = tree->s->mesh_verts
1836                   [ tree->s->mesh_indices
1837                   [ tri_buf.triangle_buffer[i]+j ][0] ][results.p.k];
1838             if(p < results.p.b)
1839                 isLeft = true;
1840             if(p > results.p.b)
1841                 isRight = true;
1842         }
1843 
1844         //Favour the right rn
1845         if(isLeft && isRight) //should be splitting.
1846         {
1847             TR.triangle_buffer[TRI++] = tri_buf.triangle_buffer[i];
1848             TL.triangle_buffer[TLI++] = tri_buf.triangle_buffer[i];
1849         }
1850         else if(!isLeft && !isRight)
1851         {
1852             if(results.side == KDTREE_LEFT)
1853                 TL.triangle_buffer[TLI++] = tri_buf.triangle_buffer[i];
1854             else if(results.side == KDTREE_RIGHT)
1855                 TR.triangle_buffer[TRI++] = tri_buf.triangle_buffer[i];
1856             else
1857             {//implement this
1858                 printf("really bad\n");
1859                 assert(1!=1);
1860             }
1861         }
1862         else if(isLeft)
1863             TL.triangle_buffer[TLI++] = tri_buf.triangle_buffer[i];
1864         else if(isRight)
1865             TR.triangle_buffer[TRI++] = tri_buf.triangle_buffer[i];
1866     }
1867     *TL_out = TL;
1868     *TR_out = TR;
1869 
1870 }
1871 
1872 bool kd_tree_should_terminate(kd_tree* tree, unsigned int num_tris, AABB V, unsigned int depth)
1873 {
1874     for(int k = 0; k < tree->k; k++)
1875         if(AABB_is_planar(&V, k))
1876             return true;
1877     if(depth == tree->max_recurse)
1878         return true;
1879     if(num_tris <= tree->tri_for_leaf_threshold)
1880         return true;
1881 
1882     return false;
1883 }
1884 
1885 kd_tree_node* kd_tree_construct_rec(kd_tree* tree, AABB V, kd_tree_triangle_buffer tri_buf,
1886                                     unsigned int depth)
1887 {
1888     kd_tree_node* node = kd_tree_node_init();
1889 
1890     tree->num_nodes_total++;
1891     if(kd_tree_should_terminate(tree, tri_buf.num_triangles, V, depth))
1892     {
1893         node->triangles = tri_buf;
1894         tree->num_leaves++;
1895         tree->num_indices_total += tri_buf.num_triangles;
1896         tree->num_tris_padded   += tri_buf.num_triangles % 8;
1897         return node;
1898     }
1899 
1900     kd_tree_find_plane_results res = kd_tree_find_plane(tree, V, tri_buf);
1901 
1902     if(res.cost > KDTREE_KI*(float)tri_buf.num_triangles)
1903     {
1904         node->triangles = tri_buf;
1905         tree->num_leaves++;
1906         tree->num_indices_total += tri_buf.num_triangles;
1907         tree->num_tris_padded   += tri_buf.num_triangles % 8;
1908 
1909         return node;
1910     }
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     return node;
1936 }
1937
1938 kd_tree_triangle_buffer kd_tree_gen_initial_tri_buf(kd_tree* tree)
1939 {
1940     assert(tree->s->num_mesh_indices % 3 == 0);
1941     kd_tree_triangle_buffer buf;
1942     buf.num_triangles   = tree->s->num_mesh_indices/3;
1943     buf.triangle_buffer = (unsigned int*) malloc(sizeof(unsigned int) * buf.num_triangles);
1944
1945     for (int i = 0; i < buf.num_triangles; i++)
1946         buf.triangle_buffer[i] = i * 3;
1947
1948     return buf;
1949 }
1950
1951 void kd_tree_construct(kd_tree* tree) //O(n log^2 n) implementation
1952 {
1953     assert(tree->s != NULL);
1954
1955     if(tree->s->num_mesh_indices == 0)
1956     {
1957         printf("WARNING: Skipping k-d tree Construction, num_mesh_indices is 0.\n");
1958         return;
1959     }
1960
1961     AABB V;
1962     AABB_construct_from_vertices(&V, tree->s->mesh_verts, tree->s->num_mesh_verts); //works
1963     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]);
1964
1965     tree->bounds = V;
1966
1967     tree->root = kd_tree_construct_rec(tree, V, kd_tree_gen_initial_tri_buf(tree), 0);
1968 }
1969
1970 unsigned int _kd_tree_write_buf(char* buffer, unsigned int offset,
1971                                                 void* data, size_t size)
1972 {
1973     memcpy(buffer+offset, data, size);
1974     return offset + size;
1975 }
1976
1977 //returns finishing offset
1978 unsigned int kd_tree_generate_serialized_buf_rec(kd_tree* tree, kd_tree_node* node, unsigned int offset)
1979 {
1980     //NOTE: this could really just be two functions
1981     if(kd_tree_node_is_leaf(node)) // leaf
1982     {
1983
1984         { //leaf body
1985             _skd_tree_leaf_node l;
1986             l.type = KDTREE_LEAF;
1987             l.num_triangles = node->triangles.num_triangles;
1988             //printf("TEST %u \n", l.num_triangles);
1989             //assert(l.num_triangles != 0);
1990             offset = _kd_tree_write_buf(tree->buffer, offset, &l, sizeof(_skd_tree_leaf_node));
1991         }
1992
1993         for(int i = 0; i < node->triangles.num_triangles; i++) //triangle indices
1994         {
1995             offset = _kd_tree_write_buf(tree->buffer, offset,
1996                                         node->triangles.triangle_buffer+i, sizeof(unsigned int));
1997         }
1998         if(node->triangles.num_triangles % 2)
1999             offset += 4;//if it isn't alligned with a long add 4 bytes (8 byte allignment)
```

```
2000
2001        return offset;
2002    }
2003    else // traversal node
2004    {
2005        _skd_tree_traversal_node n;
2006        n.type = KDTREE_NODE;
2007        n.k = node->k;
2008        n.b = node->b;
2009        unsigned int struct_start_offset = offset;
2010        offset += sizeof(_skd_tree_traversal_node);
2011
2012        unsigned int left_offset  = kd_tree_generate_serialized_buf_rec(tree, node->left, offset);
2013        //this goes after the left node
2014        unsigned int right_offset = kd_tree_generate_serialized_buf_rec(tree, node->right, left_offset);
2015
2016        n.left_ind  = offset/8;
2017        n.right_ind = left_offset/8;
2018
2019        memcpy(tree->buffer+struct_start_offset, &n, sizeof(_skd_tree_traversal_node));
2020
2021        return right_offset;
2022    }
2023 }
2024
2025 void kd_tree_generate_serialized(kd_tree* tree)
2026 {
2027    if(tree->s->num_mesh_indices == 0)
2028    {
2029        printf("WARNING: Skipping k-d tree Serialization, num_mesh_indices is 0.\n");
2030        tree->buffer_size = 0;
2031        tree->buffer = malloc(1);
2032        return;
2033    }
2034
2035    unsigned int mem_needed = 0;
2036
2037    mem_needed += tree->num_traversal_nodes * sizeof(_skd_tree_traversal_node); //traversal nodes
2038    mem_needed += tree->num_leaves * sizeof(_skd_tree_leaf_node); //leaf nodes
2039    mem_needed += (tree->num_indices_total+tree->num_tris_padded) * sizeof(unsigned int); //triangle indices
2040
2041    //char* name = malloc(256);
2042    //sprintf(name, "%d.bkdt", mem_needed);
2043
2044    tree->buffer_size = mem_needed;
2045    printf("k-d tree is %d bytes long...", mem_needed);
2046
2047    tree->buffer = malloc(mem_needed);
2048
2049
2050    /*FILE* f = fopen(name, "r");
2051    if(f!=NULL)
2052    {
2053        printf("Using cached kd tree.\n");
2054        fread(tree->buffer, 1, mem_needed, f);
2055        fclose(f);
2056    }
2057    else*/
2058    kd_tree_generate_serialized_buf_rec(tree, tree->root, 0);
2059
2060        /*{
2061        f = fopen(name, "w");
2062        fwrite(tree->buffer, 1, mem_needed, f);
2063        fclose(f);
2064    }
2065    free(name);*/
2066 }
2067 #include <loader.h>
2068 #include <parson.h>
2069 #include <vec.h>
2070 #include <float.h>
2071 #include <tinyobj_loader_c.h>
2072 #include <assert.h>
2073
2074
2075
2076 #ifndef WIN32
2077 #include <libproc.h>
2078 #include <unistd.h>
2079
2080 #define FILE_SEP '/'
2081
2082 char* _get_os_pid_bin_path()
2083 {
2084    static bool initialised = false;
2085    static char path[PROC_PIDPATHINFO_MAXSIZE];
2086    if(!initialised)
```

```
    {
        int ret;
        pid_t pid;
        //char path[PROC_PIDPATHINFO_MAXSIZE];

        pid = getpid();
        ret = proc_pidpath(pid, path, sizeof(path));

        if(ret <= 0)
        {
            printf("Error: couldn't get bin path.\n");
            exit(1);
        }
        *strrchr(path, FILE_SEP) = '\0';
    }
    printf("TEST: %s !\n", path);
    return path;
}
#else
#include <windows.h>
#define FILE_SEP '\\'

char* _get_os_pid_bin_path()
{
    static bool initialised = false;
    static char path[260];
    if(!initialised)
    {
        HMODULE hModule = GetModuleHandleW(NULL);

        WCHAR tpath[260];
        GetModuleFileNameW(hModule, tpath, 260);

        char DefChar = ' ';
        WideCharToMultiByte(CP_ACP, 0, tpath, -1, path, 260, &DefChar, NULL);

        *(strrchr(path, FILE_SEP)) = '\0'; //get last occurence;

    }
    return path;
}
#endif

char* load_file(const char* url, long *ret_length)
{
    char real_url[260];
    sprintf(real_url, "%s%cres%c%s", _get_os_pid_bin_path(), FILE_SEP, FILE_SEP, url);

    char * buffer = 0;
    long length;
    FILE * f = fopen (real_url, "rb");

    if (f)
    {
        fseek (f, 0, SEEK_END);
        length = ftell (f)+1;
        fseek (f, 0, SEEK_SET);
        buffer = malloc (length);
        if (buffer)
        {
            fread (buffer, 1, length, f);
        }
        fclose (f);
    }
    if (buffer)
    {
        buffer[length-1] = '\0';

        *ret_length = length;
        return buffer;
    }
    else
    {
        printf("Error: Couldn't load file '%s'.\n", real_url);
        exit(1);
    }
}


//Linked List for Mesh loading
struct obj_list_elem
{
    struct obj_list_elem* next;
    tinyobj_attrib_t attrib;
    tinyobj_shape_t* shapes;
    size_t num_shapes;
    int mat_index;
```

```
2174     mat4 model_mat;
2175 };
2176
2177 void obj_pre_load(char* data, long data_len, struct obj_list_elem* elem,
2178                   int* num_meshes, unsigned int* num_indices, unsigned int* num_vertices,
2179                   unsigned int* num_normals, unsigned int* num_texcoords)
2180 {
2181
2182     tinyobj_material_t* materials = NULL; //NOTE: UNUSED
2183     size_t num_materials;                 //NOTE: UNUSED
2184
2185
2186     {
2187         unsigned int flags = TINYOBJ_FLAG_TRIANGULATE;
2188         int ret = tinyobj_parse_obj(&elem->attrib, &elem->shapes, &elem->num_shapes, &materials,
2189                                     &num_materials, data, data_len, flags);
2190         if (ret != TINYOBJ_SUCCESS) {
2191             printf("Error: Couldn't parse mesh.\n");
2192             exit(1);
2193         }
2194     }
2195
2196     *num_vertices  += elem->attrib.num_vertices;
2197     *num_normals   += elem->attrib.num_normals;
2198     *num_texcoords += elem->attrib.num_texcoords;
2199     *num_meshes    += elem->num_shapes;
2200     //tinyobjloader has dumb variable names: attrib.num_faces =  num_vertices+num_faces
2201     *num_indices   += elem->attrib.num_faces;
2202 }
2203
2204
2205
2206 void load_obj(struct obj_list_elem elem, int* mesh_offset, int* vert_offset, int* nrml_offset,
2207                       int* texcoord_offset, int* index_offset, scene* out_scene)
2208 {
2209     for(int i = 0; i < elem.num_shapes; i++)
2210     {
2211         tinyobj_shape_t shape = elem.shapes[i];
2212
2213         //Get mesh and increment offset.
2214         mesh* m = (out_scene->meshes) + (*mesh_offset)++;
2215
2216         m->min[0] = m->min[1] = m->min[2] =  FLT_MAX;
2217         m->max[0] = m->max[1] = m->max[2] = -FLT_MAX;
2218
2219         memcpy(m->model, elem.model_mat, 4*4*sizeof(float));
2220
2221         m->index_offset = *index_offset;
2222         m->num_indices  =  shape.length*3;
2223         m->material_index    =  elem.mat_index;
2224
2225         for(int f = 0; f < shape.length; f++)
2226         {
2227             //TODO: don't do this error check for each iteration
2228             if(elem.attrib.face_num_verts[f+shape.face_offset]!=3)
2229             {
2230                 //This should never get called because the mesh gets triangulated when loaded.
2231                 printf("Error: the obj loader only supports triangulated meshes!\n");
2232                 exit(1);
2233             }
2234             for(int j = 0; j < 3; j++)
2235             {
2236                 tinyobj_vertex_index_t face_index = elem.attrib.faces[(f+shape.face_offset)*3+j];
2237
2238                 vec3 vertex;
2239                 vertex[0] = elem.attrib.vertices[3*face_index.v_idx+0];
2240                 vertex[1] = elem.attrib.vertices[3*face_index.v_idx+1];
2241                 vertex[2] = elem.attrib.vertices[3*face_index.v_idx+2];
2242
2243                 m->min[0] = vertex[0] < m->min[0] ? vertex[0] : m->min[0]; //X min
2244                 m->min[1] = vertex[1] < m->min[1] ? vertex[1] : m->min[1]; //Y min
2245                 m->min[2] = vertex[2] < m->min[2] ? vertex[2] : m->min[2]; //Z min
2246
2247                 m->max[0] = vertex[0] > m->max[0] ? vertex[0] : m->max[0]; //X max
2248                 m->max[1] = vertex[1] > m->max[1] ? vertex[1] : m->max[1]; //Y max
2249                 m->max[2] = vertex[2] > m->max[2] ? vertex[2] : m->max[2]; //Z max
2250
2251                 ivec3 index;
2252                 index[0] = (*vert_offset)+face_index.v_idx;
2253                 index[1] = (*nrml_offset)+face_index.vn_idx;
2254                 index[2] = (*texcoord_offset)+face_index.vt_idx;
2255                 out_scene->mesh_indices[(*index_offset)][0] = index[0];
2256                 out_scene->mesh_indices[(*index_offset)][1] = index[1];
2257                 out_scene->mesh_indices[(*index_offset)][2] = index[2];
2258                 //Sorry to anyone reading this line...
2259                 *((int*)out_scene->mesh_indices[(*index_offset)]+3) = (*mesh_offset)-1; //current mesh
2260
```

```
2261                //xv3_cpy(out_scene->mesh_indices + (*index_offset), index);
2262                (*index_offset)++;
2263            }
2264        }
2265    }
2266
2267    //__debugbreak();
2268
2269
2270    //GPU MEMORY ALIGNMENT FUN
2271    //NOTE: this is done because the gpu stores all vec3s 4 floats for memory alignment
2272    //      and it is actually faster if they are aligned like this even
2273    //      though it wastes more memory.
2274    for(int i = 0; i < elem.attrib.num_vertices; i++)
2275    {
2276
2277        memcpy(out_scene->mesh_verts + (*vert_offset),
2278               elem.attrib.vertices+3*i,
2279               sizeof(float)*3); //evem though our buffer is alligned theres is
2280        (*vert_offset) += 1;
2281    }
2282    for(int i = 0; i < elem.attrib.num_normals; i++)
2283    {
2284        memcpy(out_scene->mesh_nrmls + (*nrml_offset),
2285               elem.attrib.normals+3*i,
2286               sizeof(float)*3);
2287        (*nrml_offset) += 1;
2288    }
2289    //NOTE: the texcoords are already aligned because they only have 2 elements.
2290    memcpy(out_scene->mesh_texcoords + (*texcoord_offset), elem.attrib.texcoords,
2291           elem.attrib.num_texcoords*sizeof(vec2));
2292    (*texcoord_offset) += elem.attrib.num_texcoords;
2293 }
2294
2295 scene* load_scene_json(char* json)
2296 {
2297    printf("Beginning scene loading...\n");
2298    scene* out_scene = (scene*) malloc(sizeof(scene));
2299    JSON_Value *root_value;
2300    JSON_Object *root_object;
2301    root_value = json_parse_string(json);
2302    root_object = json_value_get_object(root_value);
2303
2304
2305    //Name
2306    {
2307        const char* name = json_object_get_string(root_object, "name");
2308        printf("Scene name: %s\n", name);
2309    }
2310
2311    //Version
2312    {//TODO: do something with this.
2313        int major  = (int)json_object_dotget_number(root_object, "version.major");
2314        int minor  = (int)json_object_dotget_number(root_object, "version.major");
2315        const char* type =      json_object_dotget_string(root_object, "version.type");
2316    }
2317
2318    //Materials
2319    {
2320        JSON_Array* material_array = json_object_get_array(root_object, "materials");
2321        out_scene->num_materials   = json_array_get_count(material_array);
2322        out_scene->materials = (material*) malloc(out_scene->num_materials*sizeof(material));
2323        assert(out_scene->num_materials>0);
2324        for(int i = 0; i < out_scene->num_materials; i++)
2325        {
2326            JSON_Object* mat = json_array_get_object(material_array, i);
2327            xv_x(out_scene->materials[i].colour) = json_object_get_number(mat, "r");
2328            xv_y(out_scene->materials[i].colour) = json_object_get_number(mat, "g");
2329            xv_z(out_scene->materials[i].colour) = json_object_get_number(mat, "b");
2330            out_scene->materials[i].reflectivity = json_object_get_number(mat, "reflectivity");
2331        }
2332        printf("Materials: %d\n", out_scene->num_materials);
2333    }
2334
2335    //Primitives
2336    {
2337
2338        JSON_Object* primitive_object = json_object_get_object(root_object, "primitives");
2339
2340        //Spheres
2341        {
2342            JSON_Array* sphere_array = json_object_get_array(primitive_object, "spheres");
2343            int num_spheres = json_array_get_count(sphere_array);
2344
2345            out_scene->spheres = malloc(sizeof(sphere)*num_spheres);
2346            out_scene->num_spheres = num_spheres;
2347
```

```
2348            for(int i = 0; i < num_spheres; i++)
2349            {
2350                JSON_Object* sphere = json_array_get_object(sphere_array, i);
2351                out_scene->spheres[i].pos[0] = json_object_get_number(sphere, "x");
2352                out_scene->spheres[i].pos[1] = json_object_get_number(sphere, "y");
2353                out_scene->spheres[i].pos[2] = json_object_get_number(sphere, "z");
2354                out_scene->spheres[i].radius = json_object_get_number(sphere, "radius");
2355                out_scene->spheres[i].material_index = json_object_get_number(sphere, "mat_index");
2356            }
2357            printf("Spheres: %d\n", out_scene->num_spheres);
2358        }
2359
2360        //Planes
2361        {
2362            JSON_Array* plane_array = json_object_get_array(primitive_object, "planes");
2363            int num_planes = json_array_get_count(plane_array);
2364
2365            out_scene->planes = malloc(sizeof(plane)*num_planes);
2366            out_scene->num_planes = num_planes;
2367
2368            for(int i = 0; i < num_planes; i++)
2369            {
2370                JSON_Object* plane = json_array_get_object(plane_array, i);
2371                out_scene->planes[i].pos[0] = json_object_get_number(plane, "x");
2372                out_scene->planes[i].pos[1] = json_object_get_number(plane, "y");
2373                out_scene->planes[i].pos[2] = json_object_get_number(plane, "z");
2374                out_scene->planes[i].norm[0] = json_object_get_number(plane, "nx");
2375                out_scene->planes[i].norm[1] = json_object_get_number(plane, "ny");
2376                out_scene->planes[i].norm[2] = json_object_get_number(plane, "nz");
2377
2378                out_scene->planes[i].material_index = json_object_get_number(plane, "mat_index");
2379            }
2380            printf("Planes: %d\n", out_scene->num_planes);
2381        }
2382
2383    }
2384
2385
2386    //Meshes
2387    {
2388        JSON_Array* mesh_array = json_object_get_array(root_object, "meshes");
2389
2390        int num_meshes = json_array_get_count(mesh_array);
2391
2392        out_scene->num_meshes = 0;
2393        out_scene->num_mesh_verts = 0;
2394        out_scene->num_mesh_nrmls = 0;
2395        out_scene->num_mesh_texcoords = 0;
2396        out_scene->num_mesh_indices = 0;
2397
2398
2399        struct obj_list_elem* first = (struct obj_list_elem*) malloc(sizeof(struct obj_list_elem));
2400        struct obj_list_elem* current = first;
2401
2402        //Pre evaluation
2403        for(int i = 0; i < num_meshes; i++)
2404        {
2405            JSON_Object* mesh = json_array_get_object(mesh_array, i);
2406            const char* url = json_object_get_string(mesh, "url");
2407            long length;
2408            char* data = load_file(url, &length);
2409            obj_pre_load(data, length, current, &out_scene->num_meshes, &out_scene->num_mesh_indices,
2410                         &out_scene->num_mesh_verts, &out_scene->num_mesh_nrmls,
2411                         &out_scene->num_mesh_texcoords);
2412            current->mat_index = (int) json_object_get_number(mesh, "mat_index");
2413            //mat4 model_mat;
2414            {
2415                //xm4_identity(model_mat);
2416                mat4 translation_mat;
2417                xm4_translatev(translation_mat,
2418                               json_object_get_number(mesh, "px"),
2419                               json_object_get_number(mesh, "py"),
2420                               json_object_get_number(mesh, "pz"));
2421                mat4 scale_mat;
2422                xm4_scalev(scale_mat,
2423                           json_object_get_number(mesh, "sx"),
2424                           json_object_get_number(mesh, "sy"),
2425                           json_object_get_number(mesh, "sz"));
2426                //TODO: add rotation.
2427                xm4_mul(current->model_mat, translation_mat, scale_mat);
2428            }
2429            free(data);
2430
2431            if(i!=num_meshes-1) //messy but it works
2432            {
2433                current->next = (struct obj_list_elem*) malloc(sizeof(struct obj_list_elem));
2434                current = current->next;
```

```
            }
            current->next = NULL;
        }

        //Allocation
        out_scene->meshes          = (mesh*) malloc(sizeof(mesh)*out_scene->num_meshes);
        out_scene->mesh_verts      = (vec3*) malloc(sizeof(vec3)*out_scene->num_mesh_verts);
        out_scene->mesh_nrmls      = (vec3*) malloc(sizeof(vec3)*out_scene->num_mesh_nrmls);
        out_scene->mesh_texcoords  = (vec2*) malloc(sizeof(vec2)*out_scene->num_mesh_texcoords);
        out_scene->mesh_indices    = (ivec3*) malloc(sizeof(ivec3)*out_scene->num_mesh_indices);

        assert(out_scene->meshes!=NULL);
        assert(out_scene->mesh_verts!=NULL);
        assert(out_scene->mesh_nrmls!=NULL);
        assert(out_scene->mesh_texcoords!=NULL);
        assert(out_scene->mesh_indices!=NULL);

        //Parsing and Assignment
        int mesh_offset = 0;
        int vert_offset = 0;
        int nrml_offset = 0;
        int texcoord_offset = 0;
        int index_offset = 0;


        current = first;
        while(current != NULL && num_meshes)
        {

            load_obj(*current, &mesh_offset, &vert_offset, &nrml_offset, &texcoord_offset,
                     &index_offset, out_scene);

            current = current->next;
        }
        printf("%i and %i\n", vert_offset, out_scene->num_mesh_verts);
        assert(mesh_offset==out_scene->num_meshes);
        assert(vert_offset==out_scene->num_mesh_verts);
        assert(nrml_offset==out_scene->num_mesh_nrmls);
        assert(texcoord_offset==out_scene->num_mesh_texcoords);

        assert(index_offset==out_scene->num_mesh_indices);

        printf("Meshes: %d\nVertices: %d\nIndices: %d\n",
               out_scene->num_meshes, out_scene->num_mesh_verts, out_scene->num_mesh_indices);

    }

    out_scene->materials_changed = true;
    out_scene->spheres_changed = true;
    out_scene->planes_changed = true;
    out_scene->meshes_changed = true;


    printf("Finshed scene loading.\n\n");

    json_value_free(root_value);
    return out_scene;
}


scene* load_scene_json_url(char* url)
{
    long variable_doesnt_matter;

    return load_scene_json( load_file(url, &variable_doesnt_matter) ); //TODO: put data
}
#include <os_abs.h>

void os_start(os_abs abs)
{
    (*abs.start_func)();
}

void os_loop_start(os_abs abs)
{
    (*abs.loop_start_func)();
}

void os_update(os_abs abs)
{
    (*abs.update_func)();
}

void os_sleep(os_abs abs, int num)
{
    (*abs.sleep_func)(num);
}
```

```
2522
2523 void* os_get_bitmap_memory(os_abs abs)
2524 {
2525     return (*abs.get_bitmap_memory_func)();
2526 }
2527
2528 void os_draw_weird(os_abs abs)
2529 {
2530     (*abs.draw_weird)();
2531 }
2532
2533 int os_get_time_mili(os_abs abs)
2534 {
2535     return (*abs.get_time_mili_func)();
2536 }
2537
2538 int os_get_width(os_abs abs)
2539 {
2540     return (*abs.get_width_func)();
2541 }
2542
2543 int os_get_height(os_abs abs)
2544 {
2545     return (*abs.get_height_func)();
2546 }
2547
2548 void os_start_thread(os_abs abs, void (*func)(void*), void* data)
2549 {
2550     (*abs.start_thread_func)(func, data);
2551 }
2552 #include <CL/opencl.h>
2553 #include <raytracer.h>
2554 //Parallel util.
2555
2556 void cl_info()
2557 {
2558
2559     int i, j;
2560     char* value;
2561     size_t valueSize;
2562     cl_uint platformCount;
2563     cl_platform_id* platforms;
2564     cl_uint deviceCount;
2565     cl_device_id* devices;
2566     cl_uint maxComputeUnits;
2567     cl_uint recommendedWorkgroupSize = 0;
2568
2569     // get all platforms
2570     clGetPlatformIDs(0, NULL, &platformCount);
2571     platforms = (cl_platform_id*) malloc(sizeof(cl_platform_id) * platformCount);
2572     clGetPlatformIDs(platformCount, platforms, NULL);
2573
2574     for (i = 0; i < platformCount; i++) {
2575
2576         // get all devices
2577         clGetDeviceIDs(platforms[i], CL_DEVICE_TYPE_ALL, 0, NULL, &deviceCount);
2578         devices = (cl_device_id*) malloc(sizeof(cl_device_id) * deviceCount);
2579         clGetDeviceIDs(platforms[i], CL_DEVICE_TYPE_ALL, deviceCount, devices, NULL);
2580
2581         // for each device print critical attributes
2582         for (j = 0; j < deviceCount; j++) {
2583
2584             // print device name
2585             clGetDeviceInfo(devices[j], CL_DEVICE_NAME, 0, NULL, &valueSize);
2586             value = (char*) malloc(valueSize);
2587             clGetDeviceInfo(devices[j], CL_DEVICE_NAME, valueSize, value, NULL);
2588             printf("%i.%d. Device: %s\n", i, j+1, value);
2589             free(value);
2590
2591             // print hardware device version
2592             clGetDeviceInfo(devices[j], CL_DEVICE_VERSION, 0, NULL, &valueSize);
2593             value = (char*) malloc(valueSize);
2594             clGetDeviceInfo(devices[j], CL_DEVICE_VERSION, valueSize, value, NULL);
2595             printf(" %i.%d.%d Hardware version: %s\n", i, j+1, 1, value);
2596             free(value);
2597
2598             // print software driver version
2599             clGetDeviceInfo(devices[j], CL_DRIVER_VERSION, 0, NULL, &valueSize);
2600             value = (char*) malloc(valueSize);
2601             clGetDeviceInfo(devices[j], CL_DRIVER_VERSION, valueSize, value, NULL);
2602             printf(" %i.%d.%d Software version: %s\n", i, j+1, 2, value);
2603             free(value);
2604
2605             // print c version supported by compiler for device
2606             clGetDeviceInfo(devices[j], CL_DEVICE_OPENCL_C_VERSION, 0, NULL, &valueSize);
2607             value = (char*) malloc(valueSize);
2608             clGetDeviceInfo(devices[j], CL_DEVICE_OPENCL_C_VERSION, valueSize, value, NULL);
```

```
            printf(" %i.%d.%d OpenCL C version: %s\n", i, j+1, 3, value);
            free(value);
            // print parallel compute units
            clGetDeviceInfo(devices[j], CL_DEVICE_MAX_COMPUTE_UNITS,
                    sizeof(maxComputeUnits), &maxComputeUnits, NULL);
            printf(" %i.%d.%d Parallel compute units: %d\n", i,  j+1, 4, maxComputeUnits);

            size_t max_work_group_size;
            clGetDeviceInfo(devices[j], CL_DEVICE_MAX_WORK_GROUP_SIZE,
                            sizeof(max_work_group_size), &max_work_group_size, NULL); //NOTE: just reuse var
            printf(" %i.%d.%d Max work group size: %zu\n", i,  j+1, 4, max_work_group_size);

            //clGetDeviceInfo(devices[j], CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE,
            //sizeof(recommendedWorkgroupSize), &recommendedWorkgroupSize, NULL);
            //printf(" %i.%d.%d Recommended work group size: %d\n", i,  j+1, 4, recommendedWorkgroupSize);

        }

        free(devices);

    }
    printf("\n");
    free(platforms);
    return;
}
void pfn_notify (
    const char *errinfo,
    const void *private_info,
    size_t cb,
    void *user_data)
{
    fprintf(stderr, "\n--\nOpenCL ERROR: %s\n--\n", errinfo);
    fflush(stderr);
}
void create_context(rcl_ctx* ctx)
{
    int err = CL_SUCCESS;


    unsigned int num_of_platforms;

    if (clGetPlatformIDs(0, NULL, &num_of_platforms) != CL_SUCCESS)
    {
        printf("Error: Unable to get platform_id\n");
        exit(1);
    }
    cl_platform_id *platform_ids = malloc(num_of_platforms*sizeof(cl_platform_id));
    if (clGetPlatformIDs(num_of_platforms, platform_ids, NULL) != CL_SUCCESS)
    {
        printf("Error: Unable to get platform_id\n");
        exit(1);
    }
    bool found = false;
    for(int i=0; i<num_of_platforms; i++)
    {
        cl_device_id device_ids[8];
        unsigned int num_devices = 0;

        //arbitrarily choosing 8 as the max gpus on a platform. TODO: ADD ERROR IF NUM DEVICES EXCEEDS 8
        if(clGetDeviceIDs(platform_ids[i], CL_DEVICE_TYPE_GPU, 8, device_ids, &num_devices) == CL_SUCCESS)
        {
            for(int j = 0; j < num_devices; j++)
            {
                char* value;
                size_t valueSize;
                clGetDeviceInfo(device_ids[j], CL_DEVICE_NAME, 0, NULL, &valueSize);
                value = (char*) malloc(valueSize);
                clGetDeviceInfo(device_ids[j], CL_DEVICE_NAME, valueSize, value, NULL);
                if(value[0]=='H'&&value[1]=='D') //janky but whatever
                {
                    printf("WARNING: Skipping over '%s' during device selection\n", value);
                    free(value);
                    continue;
                }
                free(value);

                found = true;
                ctx->platform_id = platform_ids[i];
                ctx->device_id = device_ids[j];
                break;
            }
        }
        if(found)
            break;
    }
    if(!found){
        printf("Error: Unable to get a GPU device_id\n");
```

```
2696        exit(1);
2697    }
2698
2699
2700    // Create a compute context
2701    //
2702    ctx->context = clCreateContext(0, 1, &ctx->device_id, &pfn_notify, NULL, &err);
2703    if (!ctx->context)
2704    {
2705        printf("Error: Failed to create a compute context!\n");
2706        exit(1);
2707    }
2708
2709    // Create a command commands
2710    //
2711    ctx->commands = clCreateCommandQueue(ctx->context, ctx->device_id, 0, &err);
2712    if (!ctx->commands)
2713    {
2714        printf("Error: Failed to create a command commands!\n");
2715        return;
2716    }
2717    ASRT_CL("Failed to Initialise OpenCL");
2718
2719    { // num compute cores
2720        unsigned int id;
2721        clGetDeviceInfo(ctx->device_id, CL_DEVICE_VENDOR_ID, sizeof(unsigned int), &id, NULL);
2722        switch(id)
2723        {
2724        case(0x10DE): //NVIDIA
2725        {
2726            unsigned int warp_size;
2727            unsigned int compute_capability;
2728            unsigned int num_sm;
2729            unsigned int warps_per_sm;
2730            clGetDeviceInfo(ctx->device_id, CL_DEVICE_WARP_SIZE_NV, //warp size
2731                            sizeof(unsigned int), &warp_size, NULL);
2732            clGetDeviceInfo(ctx->device_id, CL_DEVICE_COMPUTE_CAPABILITY_MAJOR_NV, //compute capability
2733                            sizeof(unsigned int), &compute_capability, NULL);
2734            clGetDeviceInfo(ctx->device_id, CL_DEVICE_MAX_COMPUTE_UNITS, //number of stream multiprocessors
2735                            sizeof(unsigned int), &num_sm, NULL);
2736
2737            switch(compute_capability)
2738            { //nvidia skipped 4 btw
2739            case 2: warps_per_sm = 1; break; //FERMI  (GK104/GK110)
2740            case 3: warps_per_sm = 6; break; //KEPLER (GK104/GK110) NOTE: ONLY 4 WARP SCHEDULERS THOUGH!
2741            case 5: warps_per_sm = 4; break; //Maxwell
2742            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
2743            case 7: warps_per_sm = 2; break; //Volta/Turing Might not be correct(NOTE: 16 FP32 PER CORE? what about warps?)
2744            }
2745
2746            printf("NVIDIA INFO: SM: %d,  WARP SIZE: %d, COMPUTE CAPABILITY: %d, WARPS PER SM: %d, TOTAL STREAM PROCESSORS: %d\n\n",
2747                   num_sm, warp_size, compute_capability, warps_per_sm, warps_per_sm*warp_size*num_sm);
2748            ctx->simt_size = warp_size;
2749            ctx->num_simt_per_multiprocessor = warps_per_sm;
2750            ctx->num_multiprocessors = num_sm;
2751            ctx->num_cores = warps_per_sm*warp_size*num_sm;
2752            break;
2753        }
2754        case(0x1002): //AMD
2755        {
2756            printf("AMD GPU INFO NOT SUPPORTED YET!\n");
2757            break;
2758        }
2759        case(0x8086): //INTEL
2760        {
2761            printf("INTEL INFO NOT SUPPORTED YET!\n");
2762            break;
2763        }
2764        default: //APPLE is really bad and doesn't return the correct vendor id.
2765        {        //Just going to use manually enter in data.
2766                printf("WARNING: Unknown Device Manufacturer %u (%04X)\n", id, id);
2767                unsigned int warp_size;
2768                unsigned int compute_capability;
2769                unsigned int num_sm;
2770                unsigned int warps_per_sm = 6; //my laptop uses kepler
2771                clGetDeviceInfo(ctx->device_id, CL_DEVICE_WARP_SIZE_NV, //warp size NOT WORKING ON OSX
2772                                sizeof(unsigned int), &warp_size, NULL);
2773                warp_size = 32;
2774                clGetDeviceInfo(ctx->device_id, CL_DEVICE_COMPUTE_CAPABILITY_MAJOR_NV, //compute capability
2775                                sizeof(unsigned int), &compute_capability, NULL);
2776                clGetDeviceInfo(ctx->device_id, CL_DEVICE_MAX_COMPUTE_UNITS, //number of stream multiprocessors
2777                                sizeof(unsigned int), &num_sm, NULL);
2778
2779                printf("ASSUMING NVIDIA.\nNVIDIA INFO: SM: %d,  WARP SIZE: %d, COMPUTE CAPABILITY: %d, WARPS PER SM: %d, TOTAL STREAM PROCES
2780                       num_sm, warp_size, compute_capability, warps_per_sm, warps_per_sm*warp_size*num_sm);
2781                ctx->simt_size = warp_size;
2782                ctx->num_simt_per_multiprocessor = warps_per_sm;
```

```
2783                ctx->num_multiprocessors = num_sm;
2784                ctx->num_cores = warps_per_sm*warp_size*num_sm;
2785
2786                break;
2787            }
2788        }
2789
2790    }
2791
2792 }
2793
2794 cl_mem gen_rgb_image(raytracer_context* rctx,
2795                      const unsigned int width,
2796                      const unsigned int height)
2797 {
2798     cl_image_desc cl_standard_descriptor;
2799     cl_image_format     cl_standard_format;
2800     cl_standard_format.image_channel_order     = CL_RGBA;
2801     cl_standard_format.image_channel_data_type = CL_FLOAT;
2802
2803     cl_standard_descriptor.image_type = CL_MEM_OBJECT_IMAGE2D;
2804     cl_standard_descriptor.image_width = width==0  ? rctx->width  : width;
2805     cl_standard_descriptor.image_height = height==0 ? rctx->height : height;
2806     cl_standard_descriptor.image_depth  = 0;
2807     cl_standard_descriptor.image_array_size  = 0;
2808     cl_standard_descriptor.image_row_pitch  = 0;
2809     cl_standard_descriptor.num_mip_levels = 0;
2810     cl_standard_descriptor.num_samples = 0;
2811     cl_standard_descriptor.buffer = NULL;
2812
2813     int err;
2814
2815     cl_mem img = clCreateImage(rctx->rcl->context,
2816                                CL_MEM_READ_WRITE,
2817                                &cl_standard_format,
2818                               &cl_standard_descriptor,
2819                                NULL,
2820                                &err);
2821     ASRT_CL("Couldn't Create OpenCL Texture");
2822     return img;
2823 }
2824
2825 rcl_img_buf gen_1d_image_buffer(raytracer_context* rctx, size_t t, void* ptr)
2826 {
2827     int err = CL_SUCCESS;
2828
2829
2830     rcl_img_buf ib;
2831     ib.size = t;
2832
2833     ib.buffer = clCreateBuffer(rctx->rcl->context,
2834                                CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
2835                                t,
2836                                ptr,
2837                                &err);
2838     ASRT_CL("Error Creating OpenCL ImageBuffer Buffer");
2839
2840
2841     cl_image_desc cl_standard_descriptor;
2842     cl_image_format     cl_standard_format;
2843     cl_standard_format.image_channel_order     = CL_RGBA;
2844     cl_standard_format.image_channel_data_type = CL_FLOAT; //prob should be float
2845
2846     cl_standard_descriptor.image_type = CL_MEM_OBJECT_IMAGE1D_BUFFER;
2847     cl_standard_descriptor.image_width = t/4 == 0 ? 1 : t/sizeof(float)/4;
2848     cl_standard_descriptor.image_height = 0;
2849     cl_standard_descriptor.image_depth  = 0;
2850     cl_standard_descriptor.image_array_size  = 0;
2851     cl_standard_descriptor.image_row_pitch  = 0;
2852     cl_standard_descriptor.image_slice_pitch = 0;
2853     cl_standard_descriptor.num_mip_levels = 0;
2854     cl_standard_descriptor.num_samples = 0;
2855     cl_standard_descriptor.buffer = ib.buffer;
2856
2857
2858     ib.image = clCreateImage(rctx->rcl->context,
2859                              0,
2860                              &cl_standard_format,
2861                              &cl_standard_descriptor,
2862                              NULL,//ptr,
2863                              &err);
2864     ASRT_CL("Error Creating OpenCL ImageBuffer Image");
2865
2866     return ib;
2867 }
2868 cl_mem gen_1d_image(raytracer_context* rctx, size_t t, void* ptr)
2869 {
```

```
2870
2871     cl_image_desc cl_standard_descriptor;
2872     cl_image_format     cl_standard_format;
2873     cl_standard_format.image_channel_order     = CL_RGBA;
2874     cl_standard_format.image_channel_data_type = CL_FLOAT; //prob should be float
2875
2876     cl_standard_descriptor.image_type = CL_MEM_OBJECT_IMAGE1D;
2877     cl_standard_descriptor.image_width = t/4 == 0 ? 1 : t/sizeof(float)/4;// t / 4 == 0 ? 1 : t / 4; //what?
2878     cl_standard_descriptor.image_height = 0;
2879     cl_standard_descriptor.image_depth  = 0;
2880     cl_standard_descriptor.image_array_size  = 0;
2881     cl_standard_descriptor.image_row_pitch  = 0;
2882     cl_standard_descriptor.image_slice_pitch = 0;
2883     cl_standard_descriptor.num_mip_levels = 0;
2884     cl_standard_descriptor.num_samples = 0;
2885     cl_standard_descriptor.buffer = NULL;
2886
2887
2888     int err = CL_SUCCESS;
2889
2890
2891     cl_mem img = clCreateImage(rctx->rcl->context,
2892                                CL_MEM_READ_WRITE | (/*ptr == NULL ? 0 :*/ CL_MEM_COPY_HOST_PTR),
2893                                &cl_standard_format,
2894                                &cl_standard_descriptor,
2895                                ptr,
2896                                &err);
2897     ASRT_CL("Couldn't Create OpenCL Texture");
2898     return img;
2899 }
2900 cl_mem gen_grayscale_buffer(raytracer_context* rctx,
2901                             const unsigned int width,
2902                             const unsigned int height)
2903 {
2904     int err;
2905
2906     cl_mem buf = clCreateBuffer(rctx->rcl->context, CL_MEM_READ_WRITE,
2907                                 (width==0  ? rctx->width  : width)*
2908                                 (height==0 ? rctx->height : height)*
2909                                 sizeof(float),
2910                                 NULL, &err);
2911     ASRT_CL("Couldn't Create OpenCL Float Buffer Image");
2912     return buf;
2913 }
2914
2915 void retrieve_image(raytracer_context* rctx, cl_mem g_buf, void* c_buf,
2916                     const unsigned int width,
2917                     const unsigned int height)
2918 {
2919     int err;
2920     size_t origin[3] = {0,0,0};
2921     size_t region[3] = {(width==0 ? rctx->width : width),
2922                         (height==0 ? rctx->height : height),
2923                         1};
2924     err = clEnqueueReadImage (rctx->rcl->commands,
2925                               g_buf,
2926                               CL_TRUE,
2927                               origin,
2928                               region,
2929                               0,
2930                               0,
2931                               c_buf,
2932                               0,
2933                               0,
2934                               NULL);
2935     ASRT_CL("Failed to retrieve Opencl Image");
2936 }
2937
2938 void retrieve_buf(raytracer_context* rctx, cl_mem g_buf, void* c_buf, size_t size)
2939 {
2940     int err;
2941     err = clEnqueueReadBuffer(rctx->rcl->commands, g_buf, CL_TRUE, 0,
2942                               size, c_buf,
2943                               0, NULL, NULL );
2944     ASRT_CL("Failed to retrieve Opencl Buffer");
2945 }
2946
2947 void zero_buffer(raytracer_context* rctx, cl_mem buf, size_t size)
2948 {
2949     int err;
2950     char pattern = 0;
2951     err =  clEnqueueFillBuffer (rctx->rcl->commands,
2952                                 buf,
2953                                 &pattern, 1 ,0,
2954                                 size,
2955                                 0, NULL, NULL);
2956     ASRT_CL("Couldn't Zero OpenCL Buffer");
```

```
}
void zero_buffer_img(raytracer_context* rctx, cl_mem buf, size_t element,
                const unsigned int width,
                const unsigned int height)
{
    int err;

    char pattern = 0;
    err =  clEnqueueFillBuffer (rctx->rcl->commands,
                                buf,
                                &pattern, 1 ,0,
                                (width==0  ? rctx->width  : width)*
                                (height==0 ? rctx->height : height)*
                                element,
                                0, NULL, NULL);
    ASRT_CL("Couldn't Zero OpenCL Buffer");
}
size_t get_workgroup_size(raytracer_context* rctx, cl_kernel kernel)
{
    int err;
    size_t local = 0;
    err = clGetKernelWorkGroupInfo(kernel, rctx->rcl->device_id,
                                   CL_KERNEL_WORK_GROUP_SIZE,
                                   sizeof(local), &local, NULL);
    ASRT_CL("Failed to Retrieve Kernel Work Group Info");
    return local;
}


void load_program_raw(rcl_ctx* ctx, char* data,
                      char** kernels, unsigned int num_kernels,
                       rcl_program* program, char** macros, unsigned int num_macros)
{
    int err;

    char* fin_data = (char*) malloc(strlen(data)+1);
    strcpy(fin_data, data);

    for(int i = 0; i < num_macros; i++) //TODO: make more efficient, don't copy all kernel code
    {
        int length = strlen(macros[i]);
        char* buf  = (char*) malloc(length+strlen(fin_data)+3);
        sprintf(buf, "%s\n%s", macros[i], fin_data);
        free(fin_data);
        fin_data = buf;
    }

    program->program = clCreateProgramWithSource(ctx->context, 1, (const char **) &fin_data, NULL, &err);
    if (!program->program)
    {
        printf("Error: Failed to create compute program!\n");
        exit(1);
    }

    // Build the program executable
    //
    err = clBuildProgram(program->program, 0, NULL, NULL, NULL, NULL);
    if (err != CL_SUCCESS)
    {
        size_t len;
        char buffer[2048*25];
        buffer[0] = '!';
        buffer[1] = '\0';


        printf("Error: Failed to build program executable!\n");
        printf("KERNEL:\n %s\nprogram done\n", fin_data);
        int n_err = clGetProgramBuildInfo(program->program, ctx->device_id, CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &len);
        if(n_err != CL_SUCCESS)
        {
            printf("The error had an error, I hate this. err:%i\n",n_err);
        }
        printf("err code:%i\n %s\n", err, buffer);
        exit(1);
    }
    else
    {
        size_t len;
        char buffer[2048 * 25];
        buffer[0] = '!';
        buffer[1] = '\0';
        int n_err = clGetProgramBuildInfo(program->program, ctx->device_id, CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &len);
        if (n_err != CL_SUCCESS)
        {
            printf("The error had an error, I hate this. err:%i\n", n_err);
        }
        printf("Build info: %s\n", buffer);
```

```
3044     }
3045
3046     program->raw_kernels = malloc(sizeof(cl_kernel)*num_kernels);
3047     for(int i = 0; i < num_kernels; i++)
3048     {
3049         // Create the compute kernel in the program we wish to run
3050         //
3051
3052         program->raw_kernels[i] = clCreateKernel(program->program, kernels[i], &err);
3053         if (!program->raw_kernels[i] || err != CL_SUCCESS)
3054         {
3055             printf("Error: Failed to create compute kernel! %s\n", kernels[i]);
3056             exit(1);
3057         }
3058
3059     }
3060
3061     program->raw_data = fin_data;
3062
3063 }
3064
3065 void load_program_url(rcl_ctx* ctx, char* url,
3066                       char** kernels, unsigned int num_kernels,
3067                       rcl_program* program, char** macros, unsigned int num_macros)
3068 {
3069     char * buffer = 0;
3070     long length;
3071     FILE * f = fopen (url, "rb");
3072
3073     if (f)
3074     {
3075         fseek (f, 0, SEEK_END);
3076         length = ftell (f);
3077         fseek (f, 0, SEEK_SET);
3078         buffer = malloc (length+2);
3079         if (buffer)
3080         {
3081             fread (buffer, 1, length, f);
3082         }
3083         fclose (f);
3084     }
3085     if (buffer)
3086     {
3087         buffer[length] = '\0';
3088
3089         load_program_raw(ctx, buffer, kernels, num_kernels, program,
3090                          macros, num_macros);
3091     }
3092
3093 }
3094
3095 //NOTE: old
3096 void test_sphere_raytracer(rcl_ctx* ctx, rcl_program* program,
3097         sphere* spheres, int num_spheres,
3098         uint32_t* bitmap, int width, int height)
3099 {
3100     int err;
3101
3102     static cl_mem tex;
3103     static cl_mem s_buf;
3104     static bool init = false; //temporary
3105
3106     if(!init)
3107     {
3108         //New Texture
3109         tex = clCreateBuffer(ctx->context, CL_MEM_WRITE_ONLY,
3110                                    width*height*4, NULL, &err);
3111
3112         //Spheres
3113         s_buf = clCreateBuffer(ctx->context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
3114                                sizeof(float)*4*num_spheres, spheres, &err);
3115         if (err != CL_SUCCESS)
3116         {
3117             printf("Error: Failed to create Sphere Buffer! %d\n", err);
3118             return;
3119         }
3120         init = true;
3121     }
3122     else
3123     {
3124         clEnqueueWriteBuffer (    ctx->commands,
3125                                   s_buf,
3126                                   CL_TRUE,
3127                                   0,
3128                                   sizeof(float)*4*num_spheres,
3129                                   spheres,
3130                                   0,
```

```
                                NULL,
                                NULL);
    }



    cl_kernel kernel = program->raw_kernels[0]; //just use the first one

    clSetKernelArg(kernel, 0, sizeof(cl_mem), &tex);
    clSetKernelArg(kernel, 1, sizeof(cl_mem), &s_buf);
    clSetKernelArg(kernel, 2, sizeof(unsigned int), &width);
    clSetKernelArg(kernel, 3, sizeof(unsigned int), &height);


    size_t global;
    size_t local = 0;

    err = clGetKernelWorkGroupInfo(kernel, ctx->device_id, CL_KERNEL_WORK_GROUP_SIZE,
        sizeof(local), &local, NULL);
    if (err != CL_SUCCESS)
    {
        printf("Error: Failed to retrieve kernel work group info! %d\n", err);
        return;
    }

    // Execute the kernel over the entire range of our 1d input data set
    // using the maximum number of work group items for this device
    //
    //printf("STARTING\n");
    global =  width*height;
    err = clEnqueueNDRangeKernel(ctx->commands, kernel, 1, NULL, &global, NULL, 0, NULL, NULL);
    if (err)
    {
        printf("Error: Failed to execute kernel! %i\n",err);
        return;
    }


    clFinish(ctx->commands);
    //printf("STOPPING\n");

    err = clEnqueueReadBuffer(ctx->commands, tex, CL_TRUE, 0, width*height*4, bitmap, 0, NULL, NULL );
    if (err != CL_SUCCESS)
    {
        printf("Error: Failed to read output array! %d\n", err);
        exit(1);
    }
}
#include <path_raytracer.h>

path_raytracer_context* init_path_raytracer_context(struct _rt_ctx* rctx)
{
    path_raytracer_context* prctx = (path_raytracer_context*) malloc(sizeof(path_raytracer_context));
    prctx->rctx = rctx;
    prctx->up_to_date = false;
    prctx->num_samples = 128;//arbitrary default
    int err;
    printf("Generating Pathtracer Buffers...\n");
    prctx->cl_path_fresh_frame_buffer = clCreateBuffer(rctx->rcl->context, CL_MEM_READ_WRITE,
                                                       rctx->width*rctx->height*sizeof(vec4),
                                                       NULL, &err);
    ASRT_CL("Error Creating OpenCL Fresh Frame Buffer.");
    prctx->cl_path_output_buffer = clCreateBuffer(rctx->rcl->context,
                                                  CL_MEM_READ_WRITE,
                                                  rctx->width*rctx->height*sizeof(vec4),
                                                  NULL, &err);
    ASRT_CL("Error Creating OpenCL Path Tracer Output Buffer.");

    printf("Generated Pathtracer Buffers...\n");
    return prctx;
}

//NOTE: the more divisions the slower.
#define WATCHDOG_DIVISIONS_X 2 //TODO: REMOVE THE WATCHDOG DIVISION SYSTEM
#define WATCHDOG_DIVISIONS_Y 2
void path_raytracer_path_trace(path_raytracer_context* prctx)
{
    int err;

    const unsigned x_div = prctx->rctx->width/WATCHDOG_DIVISIONS_X;
    const unsigned y_div = prctx->rctx->height/WATCHDOG_DIVISIONS_Y;

    //scene_resource_push(rctx); //Update Scene buffers if necessary.

    cl_kernel kernel = prctx->rctx->program->raw_kernels[PATH_TRACE_KRNL_INDX]; //just use the first one

    float zeroed[] = {0., 0., 0., 1.};
```

```
3218     float* result = matvec_mul(prctx->rctx->stat_scene->camera_world_matrix, zeroed);
3219 
3220     clSetKernelArg(kernel, 0, sizeof(cl_mem), &prctx->cl_path_fresh_frame_buffer);
3221     clSetKernelArg(kernel, 1, sizeof(cl_mem), &prctx->rctx->cl_ray_buffer);
3222     clSetKernelArg(kernel, 2, sizeof(cl_mem), &prctx->rctx->stat_scene->cl_material_buffer);
3223     clSetKernelArg(kernel, 3, sizeof(cl_mem), &prctx->rctx->stat_scene->cl_sphere_buffer);
3224     clSetKernelArg(kernel, 4, sizeof(cl_mem), &prctx->rctx->stat_scene->cl_plane_buffer);
3225     clSetKernelArg(kernel, 5, sizeof(cl_mem), &prctx->rctx->stat_scene->cl_mesh_buffer);
3226     clSetKernelArg(kernel, 6, sizeof(cl_mem), &prctx->rctx->stat_scene->cl_mesh_index_buffer.image);
3227     clSetKernelArg(kernel, 7, sizeof(cl_mem), &prctx->rctx->stat_scene->cl_mesh_vert_buffer.image);
3228     clSetKernelArg(kernel, 8, sizeof(cl_mem), &prctx->rctx->stat_scene->cl_mesh_nrml_buffer.image);
3229 
3230     clSetKernelArg(kernel, 9,  sizeof(int),     &prctx->rctx->width);
3231     clSetKernelArg(kernel, 10, sizeof(vec4),    result);
3232     clSetKernelArg(kernel, 11, sizeof(int),     &prctx->current_sample); //NOTE: I don't think this is used
3233 
3234     size_t global[2] = {x_div, y_div};
3235 
3236     //NOTE: tripping watchdog timer
3237     if(global[0]*WATCHDOG_DIVISIONS_X*global[1]*WATCHDOG_DIVISIONS_Y!=
3238        prctx->rctx->width*prctx->rctx->height)
3239     {
3240         printf("Watchdog divisions are incorrect!\n");
3241         exit(1);
3242     }
3243 
3244     size_t offset[2];
3245 
3246     for(int x = 0; x < WATCHDOG_DIVISIONS_X; x++)
3247     {
3248         for(int y = 0; y < WATCHDOG_DIVISIONS_Y; y++)
3249         {
3250             offset[0] = x_div*x;
3251             offset[1] = y_div*y;
3252             err = clEnqueueNDRangeKernel(prctx->rctx->rcl->commands, kernel, 2,
3253                                          offset, global, NULL, 0, NULL, NULL);
3254             ASRT_CL("Failed to execute path trace kernel");
3255         }
3256     }
3257 
3258     err = clFinish(prctx->rctx->rcl->commands);
3259     ASRT_CL("Something happened while executing path trace kernel");
3260 }
3261 
3262 
3263 void path_raytracer_average_buffers(path_raytracer_context* prctx)
3264 {
3265     int err;
3266 
3267     cl_kernel kernel = prctx->rctx->program->raw_kernels[F_BUFFER_AVG_KRNL_INDX];
3268     clSetKernelArg(kernel, 0, sizeof(cl_mem), &prctx->cl_path_output_buffer);
3269     clSetKernelArg(kernel, 1, sizeof(cl_mem), &prctx->cl_path_fresh_frame_buffer);
3270     clSetKernelArg(kernel, 2, sizeof(unsigned int), &prctx->rctx->width);
3271     clSetKernelArg(kernel, 3, sizeof(unsigned int), &prctx->rctx->height);
3272     clSetKernelArg(kernel, 4, sizeof(unsigned int), &prctx->num_samples);
3273     clSetKernelArg(kernel, 5, sizeof(unsigned int), &prctx->current_sample);
3274 
3275     size_t global;
3276     size_t local = get_workgroup_size(prctx->rctx, kernel);
3277 
3278     // Execute the kernel over the entire range of our 1d input data set
3279     // using the maximum number of work group items for this device
3280     //
3281     global =  prctx->rctx->width*prctx->rctx->height;
3282     err = clEnqueueNDRangeKernel(prctx->rctx->rcl->commands, kernel, 1, NULL,
3283                                  &global, NULL, 0, NULL, NULL);
3284     ASRT_CL("Failed to execute kernel");
3285     err = clFinish(prctx->rctx->rcl->commands);
3286     ASRT_CL("Something happened while waiting for kernel to finish");
3287 }
3288 
3289 void path_raytracer_push_path(path_raytracer_context* prctx)
3290 {
3291     int err;
3292 
3293     cl_kernel kernel = prctx->rctx->program->raw_kernels[F_BUF_TO_BYTE_BUF_KRNL_INDX];
3294     clSetKernelArg(kernel, 0, sizeof(cl_mem), &prctx->rctx->cl_output_buffer);
3295     clSetKernelArg(kernel, 1, sizeof(cl_mem), &prctx->cl_path_output_buffer);
3296     clSetKernelArg(kernel, 2, sizeof(unsigned int), &prctx->rctx->width);
3297     clSetKernelArg(kernel, 3, sizeof(unsigned int), &prctx->rctx->height);
3298 
3299 
3300 
3301     size_t global;
3302     size_t local = get_workgroup_size(prctx->rctx, kernel);
3303 
3304     // Execute the kernel over the entire range of our 1d input data set
```

```
3305     // using the maximum number of work group items for this device
3306     //
3307     global =  prctx->rctx->width*prctx->rctx->height;
3308     err = clEnqueueNDRangeKernel(prctx->rctx->rcl->commands, kernel, 1,
3309                                  NULL, &global, NULL, 0, NULL, NULL);
3310     ASRT_CL("Failed to execute kernel");
3311 
3312     err = clFinish(prctx->rctx->rcl->commands);
3313     ASRT_CL("Something happened while waiting for kernel to finish");
3314 
3315 
3316     err = clEnqueueReadBuffer(prctx->rctx->rcl->commands, prctx->rctx->cl_output_buffer, CL_TRUE, 0,
3317                               prctx->rctx->width*prctx->rctx->height*sizeof(int),
3318                               prctx->rctx->output_buffer,
3319                               0, NULL, NULL );
3320     ASRT_CL("Failed to read output array");
3321     //printf("RENDER\n");
3322 
3323 }
3324 
3325 
3326 void path_raytracer_render(path_raytracer_context* prctx)
3327 {
3328     int local_start_time = os_get_time_mili(abst);
3329     prctx->current_sample++;
3330     if(prctx->current_sample>prctx->num_samples)
3331     {
3332         prctx->render_complete = true;
3333         printf("Render took %d ms\n", os_get_time_mili(abst)-prctx->start_time);
3334         return;
3335     }
3336     _raytracer_gen_ray_buffer(prctx->rctx);
3337 
3338     path_raytracer_path_trace(prctx);
3339 
3340     if(prctx->current_sample == 1) //needs to be here
3341     {
3342         int err;
3343         err = clEnqueueCopyBuffer (    prctx->rctx->rcl->commands,
3344                                    prctx->cl_path_fresh_frame_buffer,
3345                                    prctx->cl_path_output_buffer,
3346                                    0,
3347                                    0,
3348                                    prctx->rctx->width*prctx->rctx->height*sizeof(vec4),
3349                                    0,
3350                                    0,
3351                                    NULL);
3352         ASRT_CL("Error copying OpenCL Output Buffer");
3353 
3354         err = clFinish(prctx->rctx->rcl->commands);
3355         ASRT_CL("Something happened while waiting for copy to finish");
3356     }
3357     path_raytracer_average_buffers(prctx);
3358     path_raytracer_push_path(prctx);
3359     printf("Total time for sample group: %d\n", os_get_time_mili(abst)-local_start_time);
3360 }
3361 
3362 void path_raytracer_prepass(path_raytracer_context* prctx)
3363 {
3364     raytracer_prepass(prctx->rctx); //Nothing Special
3365     prctx->current_sample = 0;
3366     prctx->start_time = os_get_time_mili(abst);
3367 }
3368 #include <raytracer.h>
3369 #include <parallel.h>
3370 //binary resources
3371 #include <test.cl.h> //test kernel
3372 
3373 
3374 
3375 //NOTE: we are assuming the output buffer will be the right size
3376 raytracer_context* raytracer_init(unsigned int width, unsigned int height,
3377                                   uint32_t* output_buffer, rcl_ctx* rcl)
3378 {
3379     raytracer_context* rctx = (raytracer_context*) malloc(sizeof(raytracer_context));
3380     rctx->width  = width;
3381     rctx->height = height;
3382     rctx->ray_buffer = (float*) malloc(width * height * sizeof(ray));
3383     rctx->output_buffer = output_buffer;
3384     //rctx->fresh_buffer = (uint32_t*) malloc(width * height * sizeof(uint32_t));
3385     rctx->rcl = rcl;
3386     rctx->program = (rcl_program*) malloc(sizeof(rcl_program));
3387     rctx->ic_ctx = (ic_context*) malloc(sizeof(ic_context));
3388     //ic_init(rctx);
3389     rctx->render_complete = false;
3390     rctx->num_samples     = 64; //NOTE: arbitrary default
3391     rctx->current_sample  = 0;
```

```
3392     rctx->event_position = 0;
3393     rctx->block_size_y = 0;
3394     rctx->block_size_x = 0;
3395     return rctx;
3396 }
3397
3398 void raytracer_build_kernels(raytracer_context* rctx)
3399 {
3400     printf("Building Kernels...\n");
3401     char* kernels[] = KERNELS;
3402     printf("Generating Kernel Macros...\n");
3403     //Macros
3404     unsigned int num_macros = 0;
3405     #ifdef _WIN32
3406     char os_macro[] = "#define _WIN32 1";
3407     #else
3408     char os_macro[] = "#define _OSX 1";
3409     #endif
3410     num_macros++;
3411
3412     MACRO_GEN(sphere_macro,   SCENE_NUM_SPHERES %i, rctx->stat_scene->num_spheres, num_macros);
3413     MACRO_GEN(plane_macro,    SCENE_NUM_PLANES  %i, rctx->stat_scene->num_planes,  num_macros);
3414     MACRO_GEN(index_macro,    SCENE_NUM_INDICES %i, rctx->stat_scene->num_mesh_indices, num_macros);
3415     MACRO_GEN(mesh_macro,     SCENE_NUM_MESHES  %i, rctx->stat_scene->num_meshes, num_macros);
3416     MACRO_GEN(material_macro, SCENE_NUM_MATERIALS  %i, rctx->stat_scene->num_materials, num_macros);
3417     MACRO_GEN(blockx_macro,   BLOCKSIZE_X  %i, rctx->rcl->simt_size, num_macros);
3418     MACRO_GEN(blocky_macro,   BLOCKSIZE_Y  %i, rctx->rcl->num_simt_per_multiprocessor, num_macros);
3419
3420     char min_macro[64];
3421     sprintf(min_macro, "#define SCENE_MIN (%f, %f, %f)",
3422             rctx->stat_scene->kdt->bounds.min[0],
3423             rctx->stat_scene->kdt->bounds.min[1],
3424             rctx->stat_scene->kdt->bounds.min[2]);
3425     num_macros++;
3426     char max_macro[64];
3427     sprintf(max_macro, "#define SCENE_MAX (%f, %f, %f)",
3428             rctx->stat_scene->kdt->bounds.max[0],
3429             rctx->stat_scene->kdt->bounds.max[1],
3430             rctx->stat_scene->kdt->bounds.max[2]);
3431     num_macros++;
3432
3433
3434     //TODO: do something better than this
3435     char* macros[]  = {sphere_macro, plane_macro, mesh_macro, index_macro,
3436                        material_macro, os_macro, blockx_macro, blocky_macro,
3437                        min_macro, max_macro};
3438     printf("Macros Generated.\n");
3439
3440     load_program_raw(rctx->rcl,
3441                      all_kernels_cl, //NOTE: Binary resource
3442                      kernels, NUM_KERNELS, rctx->program,
3443                      macros, num_macros);
3444     printf("Kernels built.\n");
3445
3446 }
3447
3448 void raytracer_build(raytracer_context* rctx)
3449 {
3450     //CL init
3451     printf("Building Scene...\n");
3452
3453     int err = CL_SUCCESS;
3454
3455     printf("Initializing Scene Resources On GPU.\n");
3456     scene_init_resources(rctx);
3457     rctx->stat_scene->kdt->s = rctx->stat_scene;
3458     printf("Initialized Scene Resources On GPU.\n");
3459
3460
3461     printf("Building/Rebuilding k-d tree.\n");
3462     kd_tree_construct(rctx->stat_scene->kdt);
3463     printf("Done Building/Rebuilding k-d tree.\n");
3464
3465
3466
3467     //Kernels
3468     raytracer_build_kernels(rctx);
3469
3470     //Buffers
3471     printf("Generating Buffers...\n");
3472     rctx->cl_ray_buffer = clCreateBuffer(rctx->rcl->context,
3473                                          CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
3474                                          rctx->width*rctx->height*sizeof(ray),
3475                                          rctx->ray_buffer, &err);
3476     ASRT_CL("Error Creating OpenCL Ray Buffer.");
3477     rctx->cl_path_output_buffer = clCreateBuffer(rctx->rcl->context,
3478                                          CL_MEM_READ_WRITE,
```

```
3479                                        rctx->width*rctx->height*sizeof(vec4),
3480                                        NULL, &err);
3481     ASRT_CL("Error Creating OpenCL Path Tracer Output Buffer.");
3482 
3483     rctx->cl_output_buffer = clCreateBuffer(rctx->rcl->context, CL_MEM_READ_WRITE,
3484                                             rctx->width*rctx->height*4, NULL, &err);
3485     ASRT_CL("Error Creating OpenCL Output Buffer.");
3486 
3487     //TODO: all output buffers and frame buffers should be images.
3488     rctx->cl_path_fresh_frame_buffer = clCreateBuffer(rctx->rcl->context, CL_MEM_READ_WRITE,
3489                                                       rctx->width*rctx->height*sizeof(vec4), NULL, &err);
3490     ASRT_CL("Error Creating OpenCL Fresh Frame Buffer.");
3491 
3492     printf("Generated Buffers...\n");
3493 }
3494 
3495 void raytracer_prepass(raytracer_context* rctx)
3496 {
3497     printf("Starting Raytracer Prepass.\n");
3498 
3499     scene_resource_push(rctx);
3500 
3501     printf("Finished Raytracer Prepass.\n");
3502 }
3503 
3504 void raytracer_render(raytracer_context* rctx)
3505 {
3506     _raytracer_gen_ray_buffer(rctx);
3507 
3508     _raytracer_cast_rays(rctx);
3509 }
3510 
3511 //#define JANK_SAMPLES 32
3512 void raytracer_refined_render(raytracer_context* rctx)
3513 {
3514     rctx->current_sample++;
3515     if(rctx->current_sample>rctx->num_samples)
3516     {
3517         rctx->render_complete = true;
3518         return;
3519     }
3520     _raytracer_gen_ray_buffer(rctx);
3521 
3522     _raytracer_path_trace(rctx, rctx->current_sample);
3523 
3524     if(rctx->current_sample==1) //really terrible place for path tracer initialization...
3525     {
3526         int err;
3527         char pattern = 0;
3528         err = clEnqueueCopyBuffer (    rctx->rcl->commands,
3529                                    rctx->cl_path_fresh_frame_buffer,
3530                                    rctx->cl_path_output_buffer,
3531                                    0,
3532                                    0,
3533                                    rctx->width*rctx->height*sizeof(vec4),
3534                                    0,
3535                                    0,
3536                                    NULL);
3537         ASRT_CL("Error copying OpenCL Output Buffer");
3538 
3539         err = clFinish(rctx->rcl->commands);
3540         ASRT_CL("Something happened while waiting for copy to finish");
3541     }
3542 
3543     //Nothings wrong I just am currently refactoring this
3544     //_raytracer_average_buffers(rctx, rctx->current_sample);
3545     _raytracer_push_path(rctx);
3546 
3547 }
3548 
3549 void _raytracer_gen_ray_buffer(raytracer_context* rctx)
3550 {
3551     int err;
3552 
3553     cl_kernel kernel = rctx->program->raw_kernels[RAY_BUFFER_KRNL_INDX];
3554     clSetKernelArg(kernel, 0, sizeof(cl_mem), &rctx->cl_ray_buffer);
3555     clSetKernelArg(kernel, 1, sizeof(unsigned int), &rctx->width);
3556     clSetKernelArg(kernel, 2, sizeof(unsigned int), &rctx->height);
3557     clSetKernelArg(kernel, 3, sizeof(mat4), rctx->stat_scene->camera_world_matrix);
3558 
3559 
3560     size_t global;
3561 
3562 
3563     global =  rctx->width*rctx->height;
3564     err = clEnqueueNDRangeKernel(rctx->rcl->commands, kernel, 1, NULL, &global, NULL, 0, NULL, NULL);
3565     ASRT_CL("Failed to execute kernel");
```

```
3566
3567
3568     //Wait for completion
3569     err = clFinish(rctx->rcl->commands);
3570     ASRT_CL("Something happened while waiting for kernel raybuf to finish");
3571
3572
3573 }
3574
3575
3576 void _raytracer_push_path(raytracer_context* rctx)
3577 {
3578     int err;
3579
3580     cl_kernel kernel = rctx->program->raw_kernels[F_BUF_TO_BYTE_BUF_KRNL_INDX];
3581     clSetKernelArg(kernel, 0, sizeof(cl_mem), &rctx->cl_output_buffer);
3582     clSetKernelArg(kernel, 1, sizeof(cl_mem), &rctx->cl_path_output_buffer);
3583     clSetKernelArg(kernel, 2, sizeof(unsigned int), &rctx->width);
3584     clSetKernelArg(kernel, 3, sizeof(unsigned int), &rctx->height);
3585
3586
3587
3588     size_t global;
3589     size_t local = get_workgroup_size(rctx, kernel);
3590
3591     // Execute the kernel over the entire range of our 1d input data set
3592     // using the maximum number of work group items for this device
3593     //
3594     global =  rctx->width*rctx->height;
3595     err = clEnqueueNDRangeKernel(rctx->rcl->commands, kernel, 1, NULL, &global, NULL, 0, NULL, NULL);
3596     ASRT_CL("Failed to execute kernel");
3597
3598
3599     err = clFinish(rctx->rcl->commands);
3600     ASRT_CL("Something happened while waiting for kernel to finish");
3601
3602     err = clEnqueueReadBuffer(rctx->rcl->commands, rctx->cl_output_buffer, CL_TRUE, 0,
3603                               rctx->width*rctx->height*sizeof(int), rctx->output_buffer,
3604                               0, NULL, NULL );
3605     ASRT_CL("Failed to read output array");
3606
3607 }
3608
3609 //NOTE: the more divisions the slower.
3610 #define WATCHDOG_DIVISIONS_X 2
3611 #define WATCHDOG_DIVISIONS_Y 2
3612 void _raytracer_path_trace(raytracer_context* rctx, unsigned int sample_num)
3613 {
3614     int err;
3615
3616     const unsigned x_div = rctx->width/WATCHDOG_DIVISIONS_X;
3617     const unsigned y_div = rctx->height/WATCHDOG_DIVISIONS_Y;
3618
3619     //scene_resource_push(rctx); //Update Scene buffers if necessary.
3620
3621     cl_kernel kernel = rctx->program->raw_kernels[PATH_TRACE_KRNL_INDX]; //just use the first one
3622
3623     float zeroed[] = {0., 0., 0., 1.};
3624     float* result = matvec_mul(rctx->stat_scene->camera_world_matrix, zeroed);
3625
3626     clSetKernelArg(kernel, 0, sizeof(cl_mem), &rctx->cl_path_fresh_frame_buffer);
3627     clSetKernelArg(kernel, 1, sizeof(cl_mem), &rctx->cl_ray_buffer);
3628     clSetKernelArg(kernel, 2, sizeof(cl_mem), &rctx->stat_scene->cl_material_buffer);
3629     clSetKernelArg(kernel, 3, sizeof(cl_mem), &rctx->stat_scene->cl_sphere_buffer);
3630     clSetKernelArg(kernel, 4, sizeof(cl_mem), &rctx->stat_scene->cl_plane_buffer);
3631     clSetKernelArg(kernel, 5, sizeof(cl_mem), &rctx->stat_scene->cl_mesh_buffer);
3632     clSetKernelArg(kernel, 6, sizeof(cl_mem), &rctx->stat_scene->cl_mesh_index_buffer.image);
3633     clSetKernelArg(kernel, 7, sizeof(cl_mem), &rctx->stat_scene->cl_mesh_vert_buffer.image);
3634     clSetKernelArg(kernel, 8, sizeof(cl_mem), &rctx->stat_scene->cl_mesh_nrml_buffer.image);
3635
3636     clSetKernelArg(kernel, 9,  sizeof(int),     &rctx->width);
3637     clSetKernelArg(kernel, 10, sizeof(vec4),    result);
3638     clSetKernelArg(kernel, 11, sizeof(int),     &sample_num); //NOTE: I don't think this is used
3639
3640     size_t global[2] = {x_div, y_div};
3641
3642     //NOTE: tripping watchdog timer
3643     if(global[0]*WATCHDOG_DIVISIONS_X*global[1]*WATCHDOG_DIVISIONS_Y!=rctx->width*rctx->height)
3644     {
3645         printf("Watchdog divisions are incorrect!\n");
3646         exit(1);
3647     }
3648
3649     size_t offset[2];
3650
3651     for(int x = 0; x < WATCHDOG_DIVISIONS_X; x++)
3652     {
```

```
        for(int y = 0; y < WATCHDOG_DIVISIONS_Y; y++)
        {
            offset[0] = x_div*x;
            offset[1] = y_div*y;
            err = clEnqueueNDRangeKernel(rctx->rcl->commands, kernel, 2,
                                         offset, global, NULL, 0, NULL, NULL);
            ASRT_CL("Failed to execute path trace kernel");
        }
    }

    err = clFinish(rctx->rcl->commands);
    ASRT_CL("Something happened while executing path trace kernel");
}


void _raytracer_cast_rays(raytracer_context* rctx) //TODO: do more path tracing stuff here
{
    int err;

    scene_resource_push(rctx); //Update Scene buffers if necessary.


    cl_kernel kernel = rctx->program->raw_kernels[RAY_CAST_KRNL_INDX]; //just use the first one

    float zeroed[] = {0., 0., 0., 1.};
    float* result = matvec_mul(rctx->stat_scene->camera_world_matrix, zeroed);
    clSetKernelArg(kernel, 0, sizeof(cl_mem), &rctx->cl_output_buffer);
    clSetKernelArg(kernel, 1, sizeof(cl_mem), &rctx->cl_ray_buffer);
    clSetKernelArg(kernel, 2, sizeof(cl_mem), &rctx->stat_scene->cl_material_buffer);
    clSetKernelArg(kernel, 3, sizeof(cl_mem), &rctx->stat_scene->cl_sphere_buffer);
    clSetKernelArg(kernel, 4, sizeof(cl_mem), &rctx->stat_scene->cl_plane_buffer);
    clSetKernelArg(kernel, 5, sizeof(cl_mem), &rctx->stat_scene->cl_mesh_buffer);
    clSetKernelArg(kernel, 6, sizeof(cl_mem), &rctx->stat_scene->cl_mesh_index_buffer.image);
    clSetKernelArg(kernel, 7, sizeof(cl_mem), &rctx->stat_scene->cl_mesh_vert_buffer.image);
    clSetKernelArg(kernel, 8, sizeof(cl_mem), &rctx->stat_scene->cl_mesh_nrml_buffer.image);

    clSetKernelArg(kernel, 9, sizeof(unsigned int), &rctx->width);
    clSetKernelArg(kernel, 10, sizeof(unsigned int), &rctx->height);
    clSetKernelArg(kernel, 11, sizeof(float)*4, result); //we only need 3
    //free(result);

    size_t global;

    global =  rctx->width*rctx->height;
    err = clEnqueueNDRangeKernel(rctx->rcl->commands, kernel, 1, NULL, &global, NULL, 0, NULL, NULL);
    ASRT_CL("Failed to Execute Kernel");

    err = clFinish(rctx->rcl->commands);
    ASRT_CL("Something happened during kernel execution");

    err = clEnqueueReadBuffer(rctx->rcl->commands, rctx->cl_output_buffer, CL_TRUE, 0,
                              rctx->width*rctx->height*sizeof(int), rctx->output_buffer, 0, NULL, NULL );
    ASRT_CL("Failed to read output array");

}
#include <scene.h>
#include <raytracer.h>
#include <kdtree.h>
#include <geom.h>
#include <CL/cl.h>

void scene_init_resources(raytracer_context* rctx)
{
    int err;

    //initialise kd tree
    rctx->stat_scene->kdt = kd_tree_init();


    //Scene Buffers
    rctx->stat_scene->cl_sphere_buffer = clCreateBuffer(rctx->rcl->context,
                                                        CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                                                        sizeof(sphere)*rctx->stat_scene->num_spheres,
                                                        rctx->stat_scene->spheres, &err);
    ASRT_CL("Error Creating OpenCL Scene Sphere Buffer.");

    rctx->stat_scene->cl_plane_buffer = clCreateBuffer(rctx->rcl->context,
                                                       CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                                                       sizeof(plane)*rctx->stat_scene->num_planes,
                                                       rctx->stat_scene->planes, &err);
    ASRT_CL("Error Creating OpenCL Scene Plane Buffer.");


    rctx->stat_scene->cl_material_buffer = clCreateBuffer(rctx->rcl->context,
                                                          CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                                                          sizeof(material)*
                                                          rctx->stat_scene->num_materials,
```

```
                                                    rctx->stat_scene->materials, &err);
    ASRT_CL("Error Creating OpenCL Scene Plane Buffer.");


    //Mesh
    rctx->stat_scene->cl_mesh_buffer = clCreateBuffer(rctx->rcl->context,
                                                      CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                                                      rctx->stat_scene->num_meshes==0 ? 1 :
                                                      sizeof(mesh)*rctx->stat_scene->num_meshes,
                                                      rctx->stat_scene->meshes, &err);
    ASRT_CL("Error Creating OpenCL Scene Mesh Buffer.");

    //mesh data is stored as images for faster access
    rctx->stat_scene->cl_mesh_vert_buffer =
        gen_1d_image_buffer(rctx, rctx->stat_scene->num_mesh_verts==0 ? 1 :
                            sizeof(vec3)*rctx->stat_scene->num_mesh_verts,
                            rctx->stat_scene->mesh_verts);

    rctx->stat_scene->cl_mesh_nrml_buffer =
        gen_1d_image_buffer(rctx, rctx->stat_scene->num_mesh_nrmls==0 ? 1 :
                            sizeof(vec3)*rctx->stat_scene->num_mesh_nrmls,
                            rctx->stat_scene->mesh_nrmls);

    rctx->stat_scene->cl_mesh_index_buffer =
        gen_1d_image_buffer(rctx, rctx->stat_scene->num_mesh_indices==0 ? 1 :
                            sizeof(ivec3)*
                            rctx->stat_scene->num_mesh_indices,//maybe
                            rctx->stat_scene->mesh_indices);




}


void scene_resource_push(raytracer_context* rctx)
{
    int err;

    //if(rctx->stat_scene->kdt->cl_kd_tree_buffer != NULL)
        //    exit(1);
    printf("Pushing Scene Resources...");

    printf("Serializing k-d tree...");
    kd_tree_generate_serialized(rctx->stat_scene->kdt);

//NOTE: SUPER SCUFFED
    if(rctx->stat_scene->kdt->cl_kd_tree_buffer == NULL)
    {
        rctx->stat_scene->kdt->cl_kd_tree_buffer =
            clCreateBuffer(rctx->rcl->context,
                           CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                           rctx->stat_scene->kdt->buffer_size,
                           rctx->stat_scene->kdt->buffer, &err);
        ASRT_CL("Couldn't create kd tree buffer.");
    }
    printf("Pushing Buffers...");
    if(rctx->stat_scene->meshes_changed)
    {
        clEnqueueWriteBuffer (    rctx->rcl->commands,
                                  rctx->stat_scene->cl_mesh_buffer,
                                  CL_TRUE,
                                  0,
                                  sizeof(mesh)*rctx->stat_scene->num_meshes,
                                  rctx->stat_scene->meshes,
                                  0,
                                  NULL,
                                  NULL);
    }

    if(rctx->stat_scene->spheres_changed)
    {
        clEnqueueWriteBuffer (    rctx->rcl->commands,
                                  rctx->stat_scene->cl_sphere_buffer,
                                  CL_TRUE,
                                  0,
                                  sizeof(sphere)*rctx->stat_scene->num_spheres,
                                  rctx->stat_scene->spheres,
                                  0,
                                  NULL,
                                  NULL);
    }

    if(rctx->stat_scene->planes_changed)
    {
        clEnqueueWriteBuffer (    rctx->rcl->commands,
                                  rctx->stat_scene->cl_plane_buffer,
```

```
                                CL_TRUE,
                                0,
                                sizeof(plane)*rctx->stat_scene->num_planes,
                                rctx->stat_scene->planes,
                                0,
                                NULL,
                                NULL);
    }


    if(rctx->stat_scene->materials_changed)
    {
        clEnqueueWriteBuffer (    rctx->rcl->commands,
                                rctx->stat_scene->cl_material_buffer,
                                CL_TRUE,
                                0,
                                sizeof(material)*rctx->stat_scene->num_materials,
                                rctx->stat_scene->materials,
                                0,
                                NULL,
                                NULL);
    }

    printf("Done.\n");
}
#include <spath_raytracer.h>
#include <kdtree.h>
#include <raytracer.h>
#include <stdlib.h>
//#include <windows.h>
typedef struct W_ALIGN(16) spath_progress
{
    unsigned int sample_num;
    unsigned int bounce_num;
    vec3 mask;
    vec3 accum_color;
} U_ALIGN(16) spath_progress; //NOTE: space for two more 32 bit dudes


void bad_buf_update(spath_raytracer_context* sprctx)
{
    int err;

    unsigned int bad_buf[4*4+1];
    bad_buf[4*4] = 0;
    {
        //good thing this is the same transposed. Also this is stupid, but endorced by AMD
        unsigned int mat[4*4] = {0xffffffff, 0,          0,          0,
                                 0,          0xffffffff, 0,          0,
                                 0,          0,          0xffffffff, 0,
                                 0,          0,          0,          0xffffffff};
        memcpy(bad_buf, mat, 4*4*sizeof(unsigned int));
    }

    err = clEnqueueWriteBuffer(sprctx->rctx->rcl->commands, sprctx->cl_bad_api_design_buffer, CL_TRUE,
                               0, (4*4+1)*sizeof(float),bad_buf,
                               0, NULL, NULL);
    ASRT_CL("Error Creating OpenCL BAD API DESIGN! Buffer.");

    err = clFinish(sprctx->rctx->rcl->commands);
    ASRT_CL("Something happened while waiting for copy to finish");
}

spath_raytracer_context* init_spath_raytracer_context(struct _rt_ctx* rctx)
{
    spath_raytracer_context* sprctx = (spath_raytracer_context*) malloc(sizeof(spath_raytracer_context));
    sprctx->rctx = rctx;
    sprctx->up_to_date = false;

    int err;
    printf("Generating Split Pathtracer Buffers...\n");


    sprctx->cl_path_output_buffer = clCreateBuffer(rctx->rcl->context,
                                                   CL_MEM_READ_WRITE,
                                                   rctx->width*rctx->height*sizeof(vec4),
                                                   NULL, &err);
    ASRT_CL("Error Creating OpenCL Split Path Tracer Output Buffer.");

    sprctx->cl_path_ray_origin_buffer = clCreateBuffer(rctx->rcl->context,
                                                       CL_MEM_READ_WRITE,
                                                       rctx->width*rctx->height*
                                                       sizeof(ray),
                                                       NULL, &err);
    ASRT_CL("Error Creating OpenCL Split Path Tracer Collision Result Buffer.");

    sprctx->cl_path_collision_result_buffer = clCreateBuffer(rctx->rcl->context,
```

```
                                                        CL_MEM_READ_WRITE,
                                                        rctx->width*rctx->height*
                                                        sizeof(kd_tree_collision_result),
                                                        NULL, &err);
    ASRT_CL("Error Creating OpenCL Split Path Tracer Collision Result Buffer.");

    sprctx->cl_path_origin_collision_result_buffer = clCreateBuffer(rctx->rcl->context,
                                                                    CL_MEM_READ_WRITE,
                                                                    rctx->width*rctx->height*
                                                                    sizeof(kd_tree_collision_result),
                                                                    NULL, &err);
    ASRT_CL("Error Creating OpenCL Split Path Tracer ORIGIN Collision Result Buffer.");

    sprctx->cl_random_buffer = clCreateBuffer(rctx->rcl->context,
                                              CL_MEM_READ_WRITE,
                                              rctx->width * rctx->height * sizeof(unsigned int),
                                              NULL, &err);
    ASRT_CL("Error Creating OpenCL Random Buffer.");

    sprctx->random_buffer = (unsigned int*) malloc(rctx->width * rctx->height * sizeof(unsigned int));


    sprctx->cl_spath_progress_buffer = clCreateBuffer(rctx->rcl->context,
                                                      CL_MEM_READ_WRITE,
                                                      rctx->width*rctx->height*
                                                      sizeof(spath_progress),
                                                      NULL, &err);
    zero_buffer(rctx, sprctx->cl_spath_progress_buffer, rctx->width*rctx->height*sizeof(spath_progress));

    ASRT_CL("Error Creating OpenCL Split Path Tracer Collision Result Buffer.");
    {
        unsigned int bad_buf[4*4+1];
        bad_buf[4*4] = 0;
        {
            //good thing this is the same transposed. Also this is stupid, but endorced by AMD
            unsigned int mat[4*4] = {0xffffffff, 0,          0,          0,
                                     0,          0xffffffff, 0,          0,
                                     0,          0,          0xffffffff, 0,
                                     0,          0,          0,          0xffffffff};
            memcpy(bad_buf, mat, 4*4*sizeof(unsigned int));
        }

        sprctx->cl_bad_api_design_buffer = clCreateBuffer(rctx->rcl->context,
                                                          CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
                                                          (4*4+1)*sizeof(float),
                                                          bad_buf, &err);
        ASRT_CL("Error Creating OpenCL BAD API DESIGN! Buffer.");

        err = clFinish(rctx->rcl->commands);
        ASRT_CL("Something happened while waiting for copy to finish");
    }
    printf("Generated Split Pathtracer Buffers.\n");
    return sprctx;
}

void spath_raytracer_update_random(spath_raytracer_context* sprctx)
{
    for(int i= 0; i < sprctx->rctx->width*sprctx->rctx->height; i++)
        sprctx->random_buffer[i] = rand();

    int err;

    err =  clEnqueueWriteBuffer (    sprctx->rctx->rcl->commands,
                                     sprctx->cl_random_buffer,
                                     CL_TRUE, 0,
                                     sprctx->rctx->width * sprctx->rctx->height * sizeof(unsigned int),
                                     sprctx->random_buffer,
                                     0, NULL, NULL);
    ASRT_CL("Couldn't Push Random Buffer to GPU.");
}


//NOTE: might need to do watchdog division for this, hopefully not though.
void spath_raytracer_kd_collision(spath_raytracer_context* sprctx)
{
    int err;

    cl_kernel kernel = sprctx->rctx->program->raw_kernels[KDTREE_INTERSECTION_INDX];


    clSetKernelArg(kernel, 0, sizeof(cl_mem), &sprctx->cl_path_collision_result_buffer);
    clSetKernelArg(kernel, 1, sizeof(cl_mem), &sprctx->rctx->cl_ray_buffer);

    clSetKernelArg(kernel, 2, sizeof(cl_mem), &sprctx->cl_bad_api_design_buffer); //BAD

    clSetKernelArg(kernel, 3, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_buffer);
    clSetKernelArg(kernel, 4, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_index_buffer.image);
```

```
4001     clSetKernelArg(kernel, 5, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_vert_buffer.image);
4002
4003     clSetKernelArg(kernel, 6, sizeof(cl_mem), &sprctx->rctx->stat_scene->kdt->cl_kd_tree_buffer);
4004     //NOTE: WILL NOT WORK WITH ALL SITUATIONS:
4005     unsigned int num_rays = sprctx->rctx->width*sprctx->rctx->height;
4006     clSetKernelArg(kernel, 7, sizeof(unsigned int), &num_rays);
4007
4008
4009
4010
4011     size_t global[1] = {sprctx->rctx->rcl->num_cores * 16};//sprctx->rctx->rcl->simt_size; sprctx->rctx->rcl->num_simt_per_multiprocessor};/
4012     size_t local[1]  = {sprctx->rctx->rcl->simt_size};//sprctx->rctx->rcl->simt_size; sprctx->rctx->rcl->num_simt_per_multiprocessor};// * s
4013     err = clEnqueueNDRangeKernel(sprctx->rctx->rcl->commands, kernel, 1,
4014                                  NULL, global, local, 0, NULL, NULL);
4015     ASRT_CL("Failed to execute kd tree traversal kernel");
4016
4017     err = clFinish(sprctx->rctx->rcl->commands);
4018     ASRT_CL("Something happened while executing kd tree traversal kernel");
4019
4020 }
4021
4022 void spath_raytracer_ray_test(spath_raytracer_context* sprctx)
4023 {
4024     int err;
4025
4026     cl_kernel kernel = sprctx->rctx->program->raw_kernels[KDTREE_RAY_DRAW_INDX];
4027
4028     clSetKernelArg(kernel, 0, sizeof(cl_mem), &sprctx->rctx->cl_output_buffer);
4029     clSetKernelArg(kernel, 1, sizeof(cl_mem), &sprctx->rctx->cl_ray_buffer);
4030     clSetKernelArg(kernel, 2, sizeof(unsigned int), &sprctx->rctx->width);
4031     //NOTE: WILL NOT WORK WITH ALL SITUATIONS:
4032     unsigned int num_rays = sprctx->rctx->width*sprctx->rctx->height;
4033
4034
4035     size_t global[1] = {num_rays};
4036
4037     err = clEnqueueNDRangeKernel(sprctx->rctx->rcl->commands, kernel, 1,
4038                                  NULL, global, NULL, 0, NULL, NULL);
4039     ASRT_CL("Failed to execute kd tree traversal kernel");
4040
4041     err = clFinish(sprctx->rctx->rcl->commands);
4042     ASRT_CL("Something happened while executing kd tree traversal kernel");
4043
4044     err = clEnqueueReadBuffer(sprctx->rctx->rcl->commands, sprctx->rctx->cl_output_buffer, CL_TRUE, 0,
4045                               sprctx->rctx->width*sprctx->rctx->height*sizeof(int),
4046                               sprctx->rctx->output_buffer, 0, NULL, NULL );
4047     ASRT_CL("Failed to read output array");
4048
4049 }
4050
4051 void spath_raytracer_kd_test(spath_raytracer_context* sprctx)
4052 {
4053     int err;
4054
4055     cl_kernel kernel = sprctx->rctx->program->raw_kernels[KDTREE_TEST_DRAW_INDX];
4056
4057     clSetKernelArg(kernel, 0, sizeof(cl_mem), &sprctx->rctx->cl_output_buffer);
4058     clSetKernelArg(kernel, 1, sizeof(cl_mem), &sprctx->cl_path_collision_result_buffer);
4059
4060     clSetKernelArg(kernel, 2, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_material_buffer);
4061     clSetKernelArg(kernel, 3, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_buffer);
4062     clSetKernelArg(kernel, 4, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_index_buffer.image);
4063     clSetKernelArg(kernel, 5, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_vert_buffer.image);
4064     clSetKernelArg(kernel, 6, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_nrml_buffer.image);
4065
4066     clSetKernelArg(kernel, 7, sizeof(unsigned int), &sprctx->rctx->width);
4067     //NOTE: WILL NOT WORK WITH ALL SITUATIONS:
4068     unsigned int num_rays = sprctx->rctx->width*sprctx->rctx->height;
4069
4070
4071     size_t global[1] = {num_rays};
4072
4073     err = clEnqueueNDRangeKernel(sprctx->rctx->rcl->commands, kernel, 1,
4074                                  NULL, global, NULL, 0, NULL, NULL);
4075     ASRT_CL("Failed to execute kd tree traversal kernel");
4076
4077     err = clFinish(sprctx->rctx->rcl->commands);
4078     ASRT_CL("Something happened while executing kd tree test kernel");
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     ASRT_CL("Failed to read output array");
4084 }
4085
4086 void spath_raytracer_xor_rng(spath_raytracer_context* sprctx)
4087 {
```

```
4088     int err;
4089     cl_kernel kernel = sprctx->rctx->program->raw_kernels[XORSHIFT_BATCH_INDX];
4090     clSetKernelArg(kernel, 0, sizeof(cl_mem), &sprctx->cl_random_buffer);
4091 
4092     size_t global = sprctx->rctx->width*sprctx->rctx->height;
4093 
4094     err = clEnqueueNDRangeKernel(sprctx->rctx->rcl->commands, kernel, 1, NULL,
4095                                  &global, NULL, 0, NULL, NULL);
4096     ASRT_CL("Failed to execute kernel");
4097     err = clFinish(sprctx->rctx->rcl->commands);
4098     ASRT_CL("Something happened while waiting for kernel to finish");
4099 }
4100 
4101 void spath_raytracer_avg_to_out(spath_raytracer_context* sprctx)
4102 {
4103     int err;
4104     int useless = 0;
4105     cl_kernel kernel = sprctx->rctx->program->raw_kernels[F_BUF_TO_BYTE_BUF_AVG_KRNL_INDX];
4106     clSetKernelArg(kernel, 0, sizeof(cl_mem), &sprctx->rctx->cl_output_buffer);
4107     clSetKernelArg(kernel, 1, sizeof(cl_mem), &sprctx->cl_path_output_buffer);
4108     clSetKernelArg(kernel, 2, sizeof(cl_mem), &sprctx->cl_spath_progress_buffer);
4109     clSetKernelArg(kernel, 3, sizeof(unsigned int), &sprctx->rctx->width);
4110 
4111     clSetKernelArg(kernel, 4, sizeof(unsigned int), &useless);
4112 
4113 
4114     size_t global = sprctx->rctx->width*sprctx->rctx->height;
4115 
4116     err = clEnqueueNDRangeKernel(sprctx->rctx->rcl->commands, kernel, 1, NULL,
4117                                  &global, NULL, 0, NULL, NULL);
4118     ASRT_CL("Failed to execute kernel");
4119     err = clFinish(sprctx->rctx->rcl->commands);
4120     ASRT_CL("Something happened while waiting for kernel to finish");
4121 
4122     err = clEnqueueReadBuffer(sprctx->rctx->rcl->commands, sprctx->rctx->cl_output_buffer, CL_TRUE, 0,
4123                               sprctx->rctx->width*sprctx->rctx->height*sizeof(int),
4124                               sprctx->rctx->output_buffer, 0, NULL, NULL );
4125     ASRT_CL("Failed to read output array");
4126 }
4127 
4128 
4129 void spath_raytracer_trace_init(spath_raytracer_context* sprctx)
4130 {
4131     int err;
4132     unsigned int random_value_WACKO = rand();
4133     cl_kernel kernel = sprctx->rctx->program->raw_kernels[SEGMENTED_PATH_TRACE_INIT_INDX];
4134 
4135     clSetKernelArg(kernel, 0, sizeof(cl_mem), &sprctx->cl_path_output_buffer);
4136     clSetKernelArg(kernel, 1, sizeof(cl_mem), &sprctx->rctx->cl_ray_buffer);
4137     clSetKernelArg(kernel, 2, sizeof(cl_mem), &sprctx->cl_path_ray_origin_buffer);
4138 
4139     clSetKernelArg(kernel, 3, sizeof(cl_mem),  &sprctx->cl_path_collision_result_buffer);
4140     clSetKernelArg(kernel, 4,  sizeof(cl_mem), &sprctx->cl_path_origin_collision_result_buffer);
4141 
4142 //SPATH DATA
4143     clSetKernelArg(kernel, 5, sizeof(cl_mem), &sprctx->cl_spath_progress_buffer);
4144 
4145 
4146     clSetKernelArg(kernel, 6, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_material_buffer);
4147     clSetKernelArg(kernel, 7, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_buffer);
4148     clSetKernelArg(kernel, 8, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_index_buffer.image);
4149     clSetKernelArg(kernel, 9, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_vert_buffer.image);
4150     clSetKernelArg(kernel, 10, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_nrml_buffer.image);
4151 
4152     clSetKernelArg(kernel, 11, sizeof(unsigned int), &sprctx->rctx->width);
4153     clSetKernelArg(kernel, 12, sizeof(unsigned int), &random_value_WACKO);
4154 
4155 
4156     size_t global[1] = {sprctx->rctx->width*sprctx->rctx->height};
4157 
4158     err = clEnqueueNDRangeKernel(sprctx->rctx->rcl->commands, kernel, 1,
4159                                  NULL, global, NULL, 0, NULL, NULL);
4160     ASRT_CL("Failed to execute kd tree traversal kernel");
4161 
4162     err = clFinish(sprctx->rctx->rcl->commands);
4163     ASRT_CL("Something happened while executing kd init kernel");
4164 
4165 }
4166 
4167 void spath_raytracer_trace(spath_raytracer_context* sprctx)
4168 {
4169     int err;
4170     unsigned int random_value_WACKO = rand();// sprctx->current_iteration; //TODO: make an actual random number
4171     cl_kernel kernel = sprctx->rctx->program->raw_kernels[SEGMENTED_PATH_TRACE_INDX];
4172 
4173     unsigned int karg = 0;
4174     clSetKernelArg(kernel, karg++, sizeof(cl_mem), &sprctx->cl_path_output_buffer);
```

```
4175     clSetKernelArg(kernel, karg++, sizeof(cl_mem), &sprctx->rctx->cl_ray_buffer);
4176     clSetKernelArg(kernel, karg++, sizeof(cl_mem), &sprctx->cl_path_ray_origin_buffer);
4177 
4178     clSetKernelArg(kernel, karg++, sizeof(cl_mem),  &sprctx->cl_path_collision_result_buffer);
4179     clSetKernelArg(kernel, karg++,  sizeof(cl_mem), &sprctx->cl_path_origin_collision_result_buffer);
4180     clSetKernelArg(kernel, karg++, sizeof(cl_mem), &sprctx->cl_spath_progress_buffer);
4181 
4182     clSetKernelArg(kernel, karg++, sizeof(cl_mem), &sprctx->cl_random_buffer);
4183 
4184     clSetKernelArg(kernel, karg++, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_material_buffer);
4185     clSetKernelArg(kernel, karg++, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_buffer);
4186     clSetKernelArg(kernel, karg++, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_index_buffer.image);
4187     clSetKernelArg(kernel, karg++, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_vert_buffer.image);
4188     clSetKernelArg(kernel, karg++, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_nrml_buffer.image);
4189 
4190     clSetKernelArg(kernel, karg++, sizeof(unsigned int), &sprctx->rctx->width);
4191     clSetKernelArg(kernel, karg++, sizeof(unsigned int), &random_value_WACKO);
4192 
4193     size_t global[1] = {sprctx->rctx->width*sprctx->rctx->height};
4194 
4195     err = clEnqueueNDRangeKernel(sprctx->rctx->rcl->commands, kernel, 1,
4196                                  NULL, global, NULL, 0, NULL, NULL);
4197     ASRT_CL("Failed to execute kd tree traversal kernel");
4198 
4199     err = clFinish(sprctx->rctx->rcl->commands);
4200     ASRT_CL("Something happened while executing kd tree traversal kernel");
4201 
4202 }
4203 
4204 void spath_raytracer_render(spath_raytracer_context* sprctx)
4205 {
4206     static int tbottle = 0;
4207     int t1, t2, t3, t4, t5;
4208 
4209     //Sleep(5000);
4210     if((sprctx->current_iteration+1)%50 == 0)
4211         t1 = os_get_time_mili(abst);
4212 
4213     //spath_raytracer_update_random(sprctx);
4214     spath_raytracer_xor_rng(sprctx);
4215     sprctx->current_iteration++;
4216     if(sprctx->current_iteration>sprctx->num_iterations)
4217     {
4218         if(!sprctx->render_complete)
4219             printf("Render took: %d ms", (unsigned int) os_get_time_mili(abst)-sprctx->start_time);
4220         sprctx->render_complete = true;
4221 
4222         return;
4223     }
4224 
4225     //spath_raytracer_ray_test(sprctx);
4226 
4227 
4228     bad_buf_update(sprctx);
4229 
4230     if(sprctx->current_iteration%50 == 0)
4231         t2 = os_get_time_mili(abst);
4232 
4233     spath_raytracer_kd_collision(sprctx);
4234     if(sprctx->current_iteration%50 == 0)
4235         t3 = os_get_time_mili(abst);
4236 
4237     spath_raytracer_trace(sprctx);
4238     if(sprctx->current_iteration%50 == 0)
4239         t4 = os_get_time_mili(abst);
4240 
4241     if(sprctx->current_iteration%50 == 0)
4242         spath_raytracer_avg_to_out(sprctx);
4243 
4244     if(sprctx->current_iteration%50 == 0)
4245         t5 = os_get_time_mili(abst);
4246 
4247     if(sprctx->current_iteration%50 == 0)
4248         printf("num_gen: %d, collision: %d, trace: %d, draw: %d, time_since: %d, total: %d    %d.%d/%d    %d:%d:%d\n",
4249                t2-t1, t3-t2, t4-t3, t5-t4, t1-tbottle, t5-tbottle,
4250                sprctx->current_iteration/4, sprctx->current_iteration%4, sprctx->num_iterations/4,
4251                ((t5-sprctx->start_time)/1000)/60, ((t5-sprctx->start_time)/1000)%60, (t5-sprctx->start_time)%1000);
4252     //spath_raytracer_kd_test(sprctx);
4253     tbottle = os_get_time_mili(abst);
4254 }
4255 
4256 void spath_raytracer_prepass(spath_raytracer_context* sprctx)
4257 {
4258     printf("Starting Split Path Raytracer Prepass. \n");
4259     sprctx->render_complete = false;
4260     sprctx->num_iterations = 2048*4;//arbitrary default
4261     srand((unsigned int)os_get_time_mili(abst));
```

```
4262     sprctx->start_time = (unsigned int) os_get_time_mili(abst);
4263     bad_buf_update(sprctx);
4264 
4265 
4266     zero_buffer(sprctx->rctx, sprctx->cl_path_output_buffer,
4267                 sprctx->rctx->width*sprctx->rctx->height*sizeof(vec4));
4268 
4269     raytracer_prepass(sprctx->rctx);
4270 
4271     sprctx->current_iteration = 0;
4272     zero_buffer(sprctx->rctx, sprctx->cl_spath_progress_buffer,
4273                 sprctx->rctx->width*sprctx->rctx->height*sizeof(spath_progress));
4274     _raytracer_gen_ray_buffer(sprctx->rctx);
4275 
4276 
4277 
4278     spath_raytracer_kd_collision(sprctx);
4279 
4280     spath_raytracer_trace_init(sprctx);
4281 
4282     spath_raytracer_update_random(sprctx);
4283 
4284     zero_buffer(sprctx->rctx, sprctx->rctx->cl_ray_buffer,
4285                 sprctx->rctx->width*sprctx->rctx->height*sizeof(ray));
4286 
4287     printf("Finished Split Path Raytracer Prepass. \n");
4288 }
4289 #include <ss_raytracer.h>
4290 #include <scene.h>
4291 #include <kdtree.h>
4292 #include <raytracer.h>
4293 
4294 //Single sweep, as close to real time as this thing can support.
4295 void ss_raytracer_render(ss_raytracer_context* srctx)
4296 {
4297     int err;
4298     int start_time = os_get_time_mili(abst);
4299 
4300     //TODO: @REFACTOR and remove prefix underscore and move to prepass
4301     _raytracer_gen_ray_buffer(srctx->rctx);
4302 
4303 
4304     cl_kernel kernel = srctx->rctx->program->raw_kernels[RAY_CAST_KRNL_INDX]; //just use the first one
4305 
4306     float zeroed[] = {0., 0., 0., 1.};
4307     float* result = matvec_mul(srctx->rctx->stat_scene->camera_world_matrix, zeroed);
4308     clSetKernelArg(kernel, 0, sizeof(cl_mem), &srctx->rctx->cl_output_buffer);
4309     clSetKernelArg(kernel, 1, sizeof(cl_mem), &srctx->rctx->cl_ray_buffer);
4310     clSetKernelArg(kernel, 2, sizeof(cl_mem), &srctx->rctx->stat_scene->cl_material_buffer);
4311     clSetKernelArg(kernel, 3, sizeof(cl_mem), &srctx->rctx->stat_scene->cl_sphere_buffer);
4312     clSetKernelArg(kernel, 4, sizeof(cl_mem), &srctx->rctx->stat_scene->cl_plane_buffer);
4313     clSetKernelArg(kernel, 5, sizeof(cl_mem), &srctx->rctx->stat_scene->cl_mesh_buffer);
4314     clSetKernelArg(kernel, 6, sizeof(cl_mem), &srctx->rctx->stat_scene->cl_mesh_index_buffer.image);
4315     clSetKernelArg(kernel, 7, sizeof(cl_mem), &srctx->rctx->stat_scene->cl_mesh_vert_buffer.image);
4316     clSetKernelArg(kernel, 8, sizeof(cl_mem), &srctx->rctx->stat_scene->cl_mesh_nrml_buffer.image);
4317     clSetKernelArg(kernel, 9, sizeof(unsigned int), &srctx->rctx->width);
4318     clSetKernelArg(kernel, 10, sizeof(unsigned int), &srctx->rctx->height);
4319     clSetKernelArg(kernel, 11, sizeof(float)*4, result); //we only need 3
4320     //free(result);
4321 
4322     size_t global;
4323 
4324     global =  srctx->rctx->width*srctx->rctx->height;
4325     err = clEnqueueNDRangeKernel(srctx->rctx->rcl->commands, kernel, 1, NULL, &global,
4326                                  NULL, 0, NULL, NULL);
4327     ASRT_CL("Failed to Execute Kernel");
4328 
4329     err = clFinish(srctx->rctx->rcl->commands);
4330     ASRT_CL("Something happened during kernel execution");
4331 
4332     err = clEnqueueReadBuffer(srctx->rctx->rcl->commands, srctx->rctx->cl_output_buffer, CL_TRUE, 0,
4333                               srctx->rctx->width*srctx->rctx->height*sizeof(int),
4334                               srctx->rctx->output_buffer, 0, NULL, NULL );
4335     ASRT_CL("Failed to read output array");
4336 
4337     printf("SS Render Took %d ms.\n", os_get_time_mili(abst)-start_time);
4338 }
4339 
4340 ss_raytracer_context* init_ss_raytracer_context(struct _rt_ctx* rctx)
4341 {
4342     ss_raytracer_context* ssctx = malloc(sizeof(ss_raytracer_context));
4343 
4344     ssctx->rctx = rctx;
4345     ssctx->up_to_date = false;
4346     return ssctx;
4347 }
4348 
```

```
4349
4350 //NOTE: @REFACTOR not used anymore should delete
4351 rt_vtable get_ss_raytracer_vtable()//TODO: don't use tbh.
4352 {
4353     rt_vtable v;
4354     v.up_to_date = false;
4355     //v.build      = &ss_raytracer_build;
4356     v.pre_pass     =&ss_raytracer_prepass;
4357     v.render_frame = &ss_raytracer_render;
4358     return v;
4359 }
4360
4361 void ss_raytracer_build(ss_raytracer_context* srctx)
4362 {
4363     raytracer_build(srctx->rctx); //nothing special
4364 }
4365
4366 void ss_raytracer_prepass(ss_raytracer_context* srctx)
4367 {
4368     raytracer_prepass(srctx->rctx); //Nothing Special
4369 }
4370 #include <os_abs.h>
4371 #include <stdint.h>
4372 #include <startup.h>
4373 #include <stdio.h>
4374 #include <raytracer.h>
4375 #include <mongoose.h>
4376
4377 #include <ui.h>
4378 #include <ss_raytracer.h>
4379 #include <path_raytracer.h>
4380 #include <spath_raytracer.h>
4381
4382 #ifdef WIN32
4383 #include <win32.h>
4384 #else
4385 #include <osx.h>
4386 #include <stdio.h>
4387 #include <termios.h>
4388 #include <unistd.h>
4389 #include <sys/types.h>
4390 #include <sys/time.h>
4391 #endif
4392
4393 //#include <time.h>
4394 #define _USE_MATH_DEFINES
4395 #include <math.h>
4396 #include <geom.h>
4397 #include <parallel.h>
4398 #include <loader.h>
4399 #define NUM_SPHERES 5
4400 #define NUM_PLANES  1
4401
4402 #define STRFY(x) #x
4403 #define DBL_STRFY(x) STRFY(x)
4404
4405 os_abs abst;
4406
4407 #ifndef _WIN32
4408 char kbhit()
4409 {
4410     static char initialised = false;
4411     //NOTE: we are never going to need to actually echo the characters
4412     if(!initialised)
4413     {
4414         initialised = true;
4415         struct termios term, old;
4416         tcgetattr(STDIN_FILENO, &old);
4417         term = old;
4418         term.c_lflag &= ~(ICANON | ECHO);
4419         tcsetattr(STDIN_FILENO, TCSANOW, &term);
4420     }
4421     struct timeval tv;
4422     fd_set rdfs;
4423
4424     tv.tv_sec = 0;
4425     tv.tv_usec = 0;
4426
4427     FD_ZERO(&rdfs);
4428     FD_SET(STDIN_FILENO, &rdfs);
4429
4430     select(STDIN_FILENO+1, &rdfs, NULL, NULL, &tv);
4431     return FD_ISSET(STDIN_FILENO, &rdfs);
4432 }
4433 #endif
4434
4435
```

```
4436 bool should_run = true;
4437 bool should_pause = false;
4438 void loop_exit()
4439 {
4440     should_run = false;
4441 }
4442
4443 void loop_pause()
4444 {
4445     should_pause = !should_pause;
4446 }
4447
4448
4449 void run(void* unnused_rn)
4450 {
4451
4452
4453     const int width = os_get_width(abst);
4454     const int height = os_get_height(abst);
4455
4456     const int pitch = width *4;
4457     uint32_t* row = (uint32_t*)os_get_bitmap_memory(abst);
4458
4459     cl_info();
4460
4461     rcl_ctx* rcl = (rcl_ctx*) malloc(sizeof(rcl_ctx));
4462     create_context(rcl);
4463
4464     raytracer_context* rctx = raytracer_init((unsigned int)width, (unsigned int)height,
4465                                              row, rcl);
4466     //scene* rscene = (scene*) malloc(sizeof(scene));
4467
4468
4469     os_start_thread(abst, web_server_start, rctx);
4470
4471 #ifdef DEV_MODE
4472     rctx->event_stack[rctx->event_position++] = SPLIT_PATH_RAYTRACER;
4473 #endif
4474
4475
4476
4477     scene* rscene = load_scene_json_url("scenes/path_obj2.rsc"); //TODO: support changing this during runtime
4478
4479     rctx->stat_scene = rscene;
4480     rctx->num_samples = 512; //NOTE: never actually used
4481
4482     ss_raytracer_context* ssrctx = NULL;
4483     path_raytracer_context* prctx = NULL;
4484     spath_raytracer_context* sprctx = NULL;
4485     int current_renderer = -1;
4486     bool global_up_to_date = false;
4487     while(should_run)
4488     {
4489         if(rctx->event_position)
4490         {
4491             if(!global_up_to_date)
4492             {
4493                 raytracer_build(rctx);
4494                 xm4_identity(rctx->stat_scene->camera_world_matrix);
4495                 global_up_to_date = true;
4496             }
4497             switch(rctx->event_stack[--rctx->event_position])
4498             {
4499             case(SS_RAYTRACER):
4500             {
4501                 printf("Switching To SS Raytracer\n");
4502
4503                 if(current_renderer==SS_RAYTRACER)
4504                     break;
4505                 current_renderer = SS_RAYTRACER;
4506
4507                 os_draw_weird(abst);
4508                 os_update(abst);
4509
4510                 if(ssrctx==NULL)
4511                     ssrctx = init_ss_raytracer_context(rctx);
4512
4513                 ss_raytracer_prepass(ssrctx);
4514
4515                 break;
4516             }
4517             case(PATH_RAYTRACER):
4518             {
4519                 printf("Switching To Path Tracer\n");
4520                 if(current_renderer==PATH_RAYTRACER)
4521                     break;
4522                 current_renderer = PATH_RAYTRACER;
```

```
4523
4524                os_draw_weird(abst);
4525                os_update(abst);
4526
4527                if(prctx==NULL)
4528                    prctx = init_path_raytracer_context(rctx);
4529
4530                path_raytracer_prepass(prctx);
4531
4532                break;
4533            }
4534            case(SPLIT_PATH_RAYTRACER):
4535            {
4536                printf("Switching To Split Path Tracer\n");
4537                if(current_renderer==SPLIT_PATH_RAYTRACER)
4538                    break;
4539                current_renderer = SPLIT_PATH_RAYTRACER;
4540
4541                os_draw_weird(abst);
4542                os_update(abst);
4543
4544                if(sprctx==NULL)
4545                    sprctx = init_spath_raytracer_context(rctx);
4546
4547                spath_raytracer_prepass(sprctx);
4548
4549                break;
4550            }
4551            }
4552        }
4553
4554        switch(current_renderer)
4555        {
4556        case(SS_RAYTRACER):
4557        {
4558            ss_raytracer_render(ssrctx);
4559            break;
4560        }
4561        case(PATH_RAYTRACER):
4562        {
4563            path_raytracer_render(prctx);
4564            break;
4565        }
4566        case(SPLIT_PATH_RAYTRACER):
4567        {
4568            spath_raytracer_render(sprctx);
4569            break;
4570        }
4571        }
4572        os_update(abst);
4573    }
4574
4575    //all of below shouldn't be a thing.
4576
4577    raytracer_build(rctx);
4578    raytracer_prepass(rctx);
4579
4580    xm4_identity(rctx->stat_scene->camera_world_matrix);
4581
4582    float dist = 0.f;
4583
4584
4585    int _timer_store = 0;
4586    int _timer_counter = 0;
4587    float _timer_average = 0.0f;
4588    printf("Rendering:\n\n");
4589
4590    /* static float t = 0.0f; */
4591    /* t += 0.0005f; */
4592    /* dist = sin(t)+1; */
4593    /* //mat4 temp; */
4594    /* xm4_translatev(rctx->stat_scene->camera_world_matrix, 0, dist, 0); */
4595    int real_start = os_get_time_mili(abst);
4596    while(should_run)
4597    {
4598
4599        if(should_pause)
4600            continue;
4601        int last_time = os_get_time_mili(abst);
4602
4603        if(kbhit())
4604        {
4605            switch (getc(stdin))
4606            {
4607            case 'c':
4608                exit(1);
4609                break;
```

```
4610            case 27: //ESCAPE
4611                exit(1);
4612                break;
4613            default:
4614                break;
4615            }
4616        }
4617
4618        //raytracer_refined_render(rctx);
4619        raytracer_render(rctx);
4620        if(rctx->render_complete)
4621        {
4622            printf("\n\nRender took: %02i ms (%d samples)\n\n",
4623                   os_get_time_mili(abst)-real_start, rctx->num_samples);
4624            break;
4625        }
4626
4627
4628        int mili = os_get_time_mili(abst)-last_time;
4629        _timer_store += mili;
4630        _timer_counter++;
4631        printf("\rFrame took: %02i ms, average per 20 frames: %0.2f, avg fps: %03.2f (%d/%d)    ",
4632               mili, _timer_average, 1000.0f/_timer_average,
4633               rctx->current_sample, rctx->num_samples);
4634        fflush(stdout);
4635        if(_timer_counter>20)
4636        {
4637            _timer_counter = 0;
4638            _timer_average = (float)(_timer_store)/20.f;
4639            _timer_store = 0;
4640        }
4641        os_update(abst);
4642    }
4643
4644
4645 }
4646
4647 int startup() //main function called from win32 abstraction
4648 {
4649 #ifdef WIN32
4650     abst = init_win32_abs();
4651 #else
4652     abst = init_osx_abs();
4653 #endif
4654     os_start(abst);
4655     os_start_thread(abst, run, NULL);
4656     //win32_start_thread(run, NULL);
4657
4658     os_loop_start(abst);
4659     return 0;
4660     /*
4661     printf("Hello World\n");
4662     testWin32();
4663     return 0;*/
4664 }
4665 #include <ui.h>
4666 #include <ui_web.h> //TODO: rename to ui_data or something
4667 #include <mongoose.h>
4668 #include <parson.h>
4669 #include <raytracer.h>
4670
4671 static ui_ctx uctx;
4672
4673 //Mostly based off of the exampel code for the library.
4674
4675
4676 static const char *s_http_port = "8000";
4677 static struct mg_serve_http_opts s_http_server_opts;
4678
4679
4680 void handle_ws_request(struct mg_connection *c, char* data)
4681 {
4682
4683
4684     JSON_Value *root_value;
4685     JSON_Object *root_object;
4686     root_value = json_parse_string(data);
4687     root_object = json_value_get_object(root_value);
4688
4689     switch((unsigned int)json_object_dotget_number(root_object, "type"))
4690     {
4691     case 0: //init
4692     {
4693         char buf[] = "{ \"type\":0, \"message\":\"Nothing Right Now.\"}";
4694         mg_send_websocket_frame(c, WEBSOCKET_OP_TEXT, buf, strlen(buf));
4695
4696         return;
```

```
4697     }
4698     case 1: //action
4699     {
4700         switch((unsigned int)json_object_dotget_number(root_object, "action.type"))
4701         {
4702         case SS_RAYTRACER:
4703         {
4704             if(uctx.rctx->event_position==32)
4705                 return;
4706             printf("UI Event Queued: Switch To Single Bounce\n");
4707             uctx.rctx->event_stack[uctx.rctx->event_position++] = SS_RAYTRACER;
4708             return;
4709         }
4710         case PATH_RAYTRACER: //prepass
4711         {
4712             if(uctx.rctx->event_position==32)
4713                 return;
4714             printf("UI Event Queued: Switch To Path Raytracer\n");
4715             uctx.rctx->event_stack[uctx.rctx->event_position++] = PATH_RAYTRACER;
4716             return;
4717         }
4718         case SPLIT_PATH_RAYTRACER: //start render
4719         {
4720             if(uctx.rctx->event_position==32)
4721                 return;
4722             printf("UI Event Queued: Switch To Split Path Raytracer\n");
4723             uctx.rctx->event_stack[uctx.rctx->event_position++] = SPLIT_PATH_RAYTRACER;
4724             return;
4725         }
4726         case 3: //start render
4727         {
4728             if(uctx.rctx->event_position==32)
4729                 return;
4730             printf("Change Scene %s\n", json_object_dotget_string(root_object, "action.scene"));
4731             uctx.rctx->event_stack[uctx.rctx->event_position++] = 3;
4732             printf("Not supported\n");
4733             return;
4734         }
4735         }
4736         break;
4737 
4738     }
4739     case 2: //send kd tree to GE2
4740     {
4741 
4742         printf("GE2 requested k-d tree.\n");
4743         //char buf[] = "{ \"type\":0, \"message\":\"Nothing Right Now.\"}";
4744         if(uctx.rctx->stat_scene->kdt->buffer!=NULL)
4745         {
4746 
4747             mg_send_websocket_frame(c, WEBSOCKET_OP_TEXT, //TODO: put something for this (IT'S NOT TEXT)
4748                                     uctx.rctx->stat_scene->kdt->buffer,
4749                                     uctx.rctx->stat_scene->kdt->buffer_size);
4750         }
4751         else
4752             printf("ERROR: no k-d tree.\n");
4753 
4754         break;
4755     }
4756     }
4757 
4758 }
4759 
4760 static void ev_handler(struct mg_connection *c, int ev, void *p) {
4761     if (ev == MG_EV_HTTP_REQUEST) {
4762         struct http_message *hm = (struct http_message *) p;
4763 
4764         // We have received an HTTP request. Parsed request is contained in `hm`.
4765         // Send HTTP reply to the client which shows full original request.
4766         mg_send_head(c, 200, ___src_ui_index_html_len, "Content-Type: text/html");
4767         mg_printf(c, "%.*s", (int)___src_ui_index_html_len, ___src_ui_index_html);
4768     }
4769 }
4770 
4771 
4772 static void handle_ws(struct mg_connection *c, int ev, void* ev_data) {
4773     switch (ev)
4774     { //ignore confusing indentation
4775     case MG_EV_HTTP_REQUEST:
4776     {
4777         struct http_message *hm = (struct http_message *) ev_data;
4778         //TODO: do something here
4779         mg_send_head(c, 200, ___src_ui_index_html_len, "Content-Type: text/html");
4780         mg_printf(c, "%.*s", (int)___src_ui_index_html_len, ___src_ui_index_html);
4781         break;
4782     }
4783     case MG_EV_WEBSOCKET_HANDSHAKE_DONE:
```

```
4784     {
4785         printf("Webscoket Handshake\n");
4786         break;
4787     }
4788     case MG_EV_WEBSOCKET_FRAME:
4789     {
4790         struct websocket_message *wm = (struct websocket_message *) ev_data;
4791         /* New websocket message. Tell everybody. */
4792         //struct mg_str d = {(char *) wm->data, wm->size};
4793         //printf("WOW K: %s\n", wm->data);
4794         handle_ws_request(c, wm->data);
4795         break;
4796     }
4797     }
4798 
4799     //printf("TEST 3\n");
4800     //c->flags |= MG_F_SEND_AND_CLOSE;
4801 }
4802 
4803 static void handle_ocp_li(struct mg_connection *c, int ev, void* ev_data) {
4804     if (ev == MG_EV_HTTP_REQUEST) {
4805         struct http_message *hm = (struct http_message *) ev_data;
4806 
4807         // We have received an HTTP request. Parsed request is contained in `hm`.
4808         // Send HTTP reply to the client which shows full original request.
4809         mg_send_head(c, 200, ___src_ui_ocp_li_woff_len, "Content-Type: application/font-woff");
4810         //c->send_mbuf = ___src_ui_ocp_li_woff;
4811         //c->content_len = ___src_ui_ocp_li_woff_len;
4812 
4813         mg_send(c, ___src_ui_ocp_li_woff, ___src_ui_ocp_li_woff_len);
4814         //mg_printf(c, "%.*s", (int)___src_ui_ocp_li_woff_len, ___src_ui_ocp_li_woff);
4815     }
4816     //printf("TEST 2\n");
4817     c->flags |= MG_F_SEND_AND_CLOSE;
4818 }
4819 
4820 
4821 static void handle_style(struct mg_connection* c, int ev, void* ev_data) {
4822     if (ev == MG_EV_HTTP_REQUEST) {
4823         struct http_message *hm = (struct http_message *) ev_data;
4824 
4825         // We have received an HTTP request. Parsed request is contained in `hm`.
4826         // Send HTTP reply to the client which shows full original request.
4827         mg_send_head(c, 200, ___src_ui_style_css_len, "Content-Type: text/css");
4828         mg_printf(c, "%.*s", (int)___src_ui_style_css_len, ___src_ui_style_css);
4829     }
4830     //printf("TEST\n");
4831     c->flags |= MG_F_SEND_AND_CLOSE;
4832 }
4833 
4834 void web_server_start(void* rctx)
4835 {
4836     uctx.rctx = rctx;
4837     struct mg_mgr mgr;
4838     struct mg_connection *c;
4839 
4840     mg_mgr_init(&mgr, NULL);
4841     c = mg_bind(&mgr, s_http_port, ev_handler);
4842     mg_set_protocol_http_websocket(c);
4843     mg_register_http_endpoint(c, "/ocp_li.woff", handle_ocp_li);
4844     mg_register_http_endpoint(c, "/style.css", handle_style);
4845     mg_register_http_endpoint(c, "/ws", handle_ws);
4846 
4847     printf("Web UI Hosted On Port %s\n", s_http_port);
4848 
4849     for (;;) {
4850         mg_mgr_poll(&mgr, 1000);
4851     }
4852     mg_mgr_free(&mgr);
4853 
4854     exit(1);
4855 
4856 }
4857 #include <win32.h>
4858 #include <startup.h>
4859 #include <windows.h>
4860 #include <math.h>
4861 #include <stdio.h>
4862 #include <stdint.h>
4863 #include <assert.h>
4864 #include <stdio.h>
4865 #include <io.h>
4866 #include <fcntl.h>
4867 const char CLASS_NAME[] = "Raytracer";
4868 
4869 
4870 static win32_context* ctx;
```

```
4871
4872 void win32_draw_meme(); //vague predef
4873
4874 os_abs init_win32_abs()
4875 {
4876     os_abs abstraction;
4877     abstraction.start_func = &win32_start;
4878     abstraction.loop_start_func = &win32_loop;
4879     abstraction.update_func = &win32_update;
4880     abstraction.sleep_func = &win32_sleep;
4881     abstraction.get_bitmap_memory_func = &win32_get_bitmap_memory;
4882     abstraction.get_time_mili_func = &win32_get_time_mili;
4883     abstraction.get_width_func = &win32_get_width;
4884     abstraction.get_height_func = &win32_get_height;
4885     abstraction.start_thread_func = &win32_start_thread;
4886     abstraction.draw_weird = &win32_draw_meme;
4887     return abstraction;
4888 }
4889
4890 void* get_bitmap_memory()
4891 {
4892     return ctx->bitmap_memory;
4893 }
4894
4895 void win32_draw_meme()
4896 {
4897     int width  = ctx->width;
4898     int height = ctx->height;
4899
4900     int pitch = width*4;
4901     uint8_t* row = (uint8_t*)ctx->bitmap_memory;
4902
4903     for(int y = 0; y < height; y++)
4904     {
4905         uint8_t* pixel = (uint8_t*)row;
4906         for(int x = 0; x < width; x++)
4907         {
4908             *pixel = sin(((float)x)/150)*255;
4909             ++pixel;
4910
4911             *pixel = cos(((float)x)/10)*100;
4912             ++pixel;
4913
4914             *pixel = cos(((float)y)/50)*255;
4915             ++pixel;
4916
4917             *pixel = 0;
4918             ++pixel;
4919             /* ((char*)ctx->bitmap_memory)[(x+y*width)*4]   =  (y%2) ? 0xff : 0x00; */
4920             /* ((char*)ctx->bitmap_memory)[(x*4+y*width)+1] =  0x00; */
4921             /* ((char*)ctx->bitmap_memory)[(x*4+y*width)+2] =  (y%2) ? 0xff : 0x00; */
4922             /* ((char*)ctx->bitmap_memory)[(x*4+y*width)+3] =  0x00; */
4923         }
4924         row += pitch;
4925     }
4926 }
4927
4928 void win32_sleep(int mili)
4929 {
4930     Sleep(mili);
4931 }
4932
4933 void win32_resize_dib_section(int width, int height)
4934 {
4935     if(ctx->bitmap_memory)
4936         VirtualFree(ctx->bitmap_memory, 0, MEM_RELEASE);
4937
4938     ctx->width = width;
4939     ctx->height = height;
4940
4941     ctx->bitmap_info.bmiHeader.biSize          = sizeof(ctx->bitmap_info.bmiHeader);
4942     ctx->bitmap_info.bmiHeader.biWidth         = width;
4943     ctx->bitmap_info.bmiHeader.biHeight        = -height;
4944     ctx->bitmap_info.bmiHeader.biPlanes        = 1;
4945     ctx->bitmap_info.bmiHeader.biBitCount      = 32; //8 bits of paddingll
4946     ctx->bitmap_info.bmiHeader.biCompression   = BI_RGB;
4947     ctx->bitmap_info.bmiHeader.biSizeImage     = 0;
4948     ctx->bitmap_info.bmiHeader.biXPelsPerMeter = 0;
4949     ctx->bitmap_info.bmiHeader.biYPelsPerMeter = 0;
4950     ctx->bitmap_info.bmiHeader.biClrUsed       = 0;
4951     ctx->bitmap_info.bmiHeader.biClrImportant  = 0;
4952
4953     //I could use BitBlit if it would increase spead.
4954     int bytes_per_pixel = 4;
4955     int bitmap_memory_size = (width*height)*bytes_per_pixel;
4956     ctx->bitmap_memory = VirtualAlloc(0, bitmap_memory_size, MEM_COMMIT, PAGE_READWRITE);
4957
```

```
4958 }
4959
4960 void win32_update_window(HDC device_context, HWND win, int width, int height)
4961 {
4962
4963     int window_height = height;//window_rect.bottom - window_rect.top;
4964     int window_width  = width;//window_rect.right - window_rect.left;
4965
4966
4967     //TODO: Replace with BitBlt this is way too slow... (we don't even need the scaling);
4968     StretchDIBits(device_context,
4969                   /* x, y, width, height, */
4970                   /* x, y, width, height, */
4971                   0, 0, ctx->width, ctx->height,
4972                   0, 0, window_width, window_height,
4973
4974                   ctx->bitmap_memory,
4975                   &ctx->bitmap_info,
4976                   DIB_RGB_COLORS, SRCCOPY);
4977 }
4978
4979
4980 LRESULT CALLBACK WndProc(HWND win, UINT msg, WPARAM wParam, LPARAM lParam)
4981 {
4982     switch(msg)
4983     {
4984     case WM_KEYDOWN:
4985         switch (wParam)
4986         {
4987         case VK_ESCAPE:
4988             loop_exit();
4989             ctx->shouldRun = false;
4990             break;
4991
4992         case VK_SPACE:
4993             loop_pause();
4994             break;
4995         default:
4996             break;
4997         }
4998         break;
4999     case WM_SIZE:
5000     {
5001         RECT drawable_rect;
5002         GetClientRect(win, &drawable_rect);
5003
5004         int height = drawable_rect.bottom - drawable_rect.top;
5005         int width  = drawable_rect.right - drawable_rect.left;
5006         win32_resize_dib_section(width, height);
5007
5008         win32_draw_meme();
5009     } break;
5010     case WM_CLOSE:
5011         ctx->shouldRun = false;
5012         break;
5013     case WM_DESTROY:
5014         ctx->shouldRun = false;
5015         break;
5016     case WM_ACTIVATEAPP:
5017         OutputDebugStringA("WM_ACTIVATEAPP\n");
5018         break;
5019     case WM_PAINT:
5020     {
5021         PAINTSTRUCT paint;
5022         HDC device_context = BeginPaint(win, &paint);
5023         EndPaint(win, &paint);
5024
5025         /*int x = paint.rcPaint.left;
5026         int y = paint.rcPaint.top;
5027         int height = paint.rcPaint.bottom - paint.rcPaint.top;
5028         int width  = paint.rcPaint.right - paint.rcPaint.left;*/
5029         //PatBlt(device_context, x, y, width, height, BLACKNESS);
5030
5031         RECT drawable_rect;
5032         GetClientRect(win, &drawable_rect);
5033
5034         int height = drawable_rect.bottom - drawable_rect.top;
5035         int width  = drawable_rect.right - drawable_rect.left;
5036
5037         GetClientRect(win, &drawable_rect);
5038         win32_update_window(device_context,
5039                             win, width, height);
5040
5041     } break;
5042     default:
5043         return DefWindowProc(win, msg, wParam, lParam);
5044     }
```

```
5045     return 0;
5046 }
5047
5048
5049
5050 int _WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
5051                   LPSTR lpCmdLine, int nCmdShow)
5052 {
5053
5054     ctx = (win32_context*) malloc(sizeof(win32_context));
5055
5056     ctx->instance = hInstance;
5057     ctx->nCmdShow = nCmdShow;
5058     ctx->wc.cbSize        = sizeof(WNDCLASSEX);
5059     ctx->wc.style         = CS_OWNDC|CS_HREDRAW|CS_VREDRAW;
5060     ctx->wc.lpfnWndProc   = WndProc;
5061     ctx->wc.cbClsExtra    = 0;
5062     ctx->wc.cbWndExtra    = 0;
5063     ctx->wc.hInstance     = hInstance;
5064     ctx->wc.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
5065     ctx->wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
5066     ctx->wc.hbrBackground = 0;//(HBRUSH)(COLOR_WINDOW+1);
5067     ctx->wc.lpszMenuName  = NULL;
5068     ctx->wc.lpszClassName = CLASS_NAME;
5069     ctx->wc.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);
5070
5071     if(!SetPriorityClass(
5072            GetCurrentProcess(),
5073            HIGH_PRIORITY_CLASS
5074            ))
5075     {
5076         printf("FUCKKKK!!!\n");
5077     }
5078
5079
5080
5081     startup();
5082
5083     return 0;
5084 }
5085
5086 int main()
5087 {
5088     //printf("JANKY WINMAIN OVERRIDE\n");
5089     return _WinMain(GetModuleHandle(NULL), NULL, GetCommandLineA(), SW_SHOWNORMAL);
5090 }
5091
5092 //Should Block the Win32 Update Loop.
5093 #define WIN32_SHOULD_BLOCK_LOOP
5094
5095 void win32_loop()
5096 {
5097     printf("Starting WIN32 Window Loop\n");
5098     MSG msg;
5099     ctx->shouldRun = true;
5100     while(ctx->shouldRun)
5101     {
5102 #ifdef WIN32_SHOULD_BLOCK_LOOP
5103
5104
5105         if(GetMessage(&msg, 0, 0, 0) > 0)
5106         {
5107             TranslateMessage(&msg);
5108             DispatchMessage(&msg);
5109         }
5110
5111 #else
5112         while(PeekMessage(&msg, 0, 0, 0, PM_REMOVE))
5113         {
5114             if(msg.message == WM_QUIT)
5115             {
5116                 ctx->shouldRun = false;
5117             }
5118             TranslateMessage(&msg);
5119             DispatchMessage(&msg);
5120         }
5121 #endif
5122         //win32_draw_meme();
5123         //win32_update_window();
5124     }
5125 }
5126
5127
5128 void create_win32_window()
5129 {
5130     printf("Creating WIN32 Window\n");
5131
```

```
5132     ctx->win = CreateWindowEx(
5133         0,
5134         CLASS_NAME,
5135         CLASS_NAME,
5136         /* WS_OVERLAPPEDWINDOW, */
5137         (WS_POPUP| WS_SYSMENU | WS_MAXIMIZEBOX | WS_MINIMIZEBOX),
5138         CW_USEDEFAULT, CW_USEDEFAULT, 1920, 1080,
5139         NULL, NULL, ctx->instance, NULL);
5140 
5141     if(ctx->win == NULL)
5142     {
5143         MessageBox(NULL, "Window Creation Failed!", "Error!",
5144                    MB_ICONEXCLAMATION | MB_OK);
5145         return;
5146     }
5147 
5148     ShowWindow(ctx->win, ctx->nCmdShow);
5149     UpdateWindow(ctx->win);
5150 
5151 }
5152 
5153 
5154 //NOTE: Should the start func start the loop
5155 //#define WIN32_SHOULD_START_LOOP_ON_START
5156 void win32_start()
5157 {
5158     if(!RegisterClassEx(&ctx->wc))
5159     {
5160         MessageBox(NULL, "Window Registration Failed!", "Error!",
5161                    MB_ICONEXCLAMATION | MB_OK);
5162         return;
5163     }
5164     create_win32_window();
5165 #ifdef WIN32_SHOULD_START_LOOP_ON_START
5166     win32_loop();
5167 #endif
5168 
5169 }
5170 
5171 int win32_get_time_mili()
5172 {
5173     SYSTEMTIME st;
5174     GetSystemTime(&st);
5175     return (int) st.wMilliseconds+(st.wSecond*1000)+(st.wMinute*1000*60);
5176 }
5177 
5178 void win32_update()
5179 {
5180     //RECT win_rect;
5181     //GetClientRect(ctx->win, &win_rect);
5182     HDC dc = GetDC(ctx->win);
5183     win32_update_window(dc, ctx->win, ctx->width, ctx->height);
5184     ReleaseDC(ctx->win, dc);
5185 
5186 }
5187 
5188 
5189 int win32_get_width()
5190 {
5191     return ctx->width;
5192 }
5193 
5194 int win32_get_height()
5195 {
5196     return ctx->height;
5197 }
5198 
5199 void* win32_get_bitmap_memory()
5200 {
5201     return ctx->bitmap_memory;
5202 }
5203 
5204 
5205 typedef struct
5206 {
5207     void* data;
5208     void (*func)(void*);
5209 } thread_func_meta;
5210 
5211 DWORD WINAPI thread_func(void* data)
5212 {
5213     if(!SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_HIGHEST))
5214     {
5215         DWORD dwError;
5216         dwError = GetLastError();
5217         printf(TEXT("Failed to change thread priority (%d)\n"), dwError);
5218     }
```

```
5219
5220     thread_func_meta* meta = (thread_func_meta*) data;
5221     (meta->func)(meta->data); //confusing syntax: call the passed function with the passed data
5222     free(meta);
5223     return 0;
5224 }
5225
5226 void win32_start_thread(void (*func)(void*), void* data)
5227 {
5228     thread_func_meta* meta = (thread_func_meta*) malloc(sizeof(thread_func_meta));
5229     meta->data = data;
5230     meta->func = func;
5231     HANDLE t = CreateThread(NULL, 0, thread_func, meta, 0, NULL);
5232     //if(SetThreadPriority(t, THREAD_PRIORITY_HIGHEST)==0)
5233     //    assert(false);
5234
5235 }
5236 #import <Cocoa/Cocoa.h>
5237 #include <osx.h>
5238 #include <startup.h>
5239 #include <sys/types.h>
5240
5241 #include <os_abs.h>
5242 #include <stdio.h>
5243 #include <time.h>
5244 #include <stdlib.h>
5245 #include <pthread.h>
5246
5247 #if 1
5248 int main()
5249 {
5250     startup();
5251 }
5252 #endif
5253
5254 typedef struct
5255 {
5256     unsigned char* bitmap_memory;
5257
5258     unsigned int width;
5259     unsigned int height;
5260
5261     dispatch_queue_t main_queue;
5262
5263     NSBitmapImageRep* bitmap;
5264 } osx_ctx;
5265 //NOTE: probably not good
5266 static osx_ctx* ctx;
5267
5268
5269 void osx_sleep(int miliseconds)
5270 {
5271     struct timespec ts;
5272     ts.tv_sec = miliseconds/1000;
5273     ts.tv_nsec = (miliseconds%1000)*1000000;
5274     nanosleep(&ts, NULL);
5275 }
5276
5277 void* osx_get_bitmap_memory()
5278 {
5279     return ctx->bitmap_memory;
5280 }
5281
5282 int osx_get_time_mili()
5283 {
5284     int err = 0;
5285     struct timespec ts;
5286     if((err = clock_gettime(CLOCK_REALTIME, &ts)))
5287     {
5288         printf("ERROR: failed to retrieve time. (osx abstraction) %i", err);
5289         exit(1);
5290     }
5291     return (ts.tv_sec*1000)+(ts.tv_nsec/1000000);
5292 }
5293
5294 int osx_get_width()
5295 {
5296     return ctx->width;
5297 }
5298 int osx_get_height()
5299 {
5300     return ctx->height;
5301 }
5302
5303 void initBitmapData(unsigned char* bmap, float offset, unsigned int width, unsigned int height)
5304 {
5305     int pitch = width*4;
```

```
5306     uint8_t* row = bmap;
5307 
5308     for(int y = 0; y < height; y++)
5309     {
5310         uint8_t* pixel = (uint8_t*)row;
5311         for(int x = 0; x < width; x++)
5312         {
5313             *pixel = sin(((float)x+offset)/150)*255;
5314             ++pixel;
5315 
5316             *pixel = cos(((float)x-offset)/10)*100;
5317             ++pixel;
5318 
5319             *pixel = cos(((float)y*(offset+1))/50)*255;
5320             ++pixel;
5321 
5322             *pixel = 255;
5323             ++pixel;
5324         }
5325         row += pitch;
5326     }
5327 }
5328 
5329 void doesnt_work_on_osx()
5330 {
5331     //printf("I hope this works.\n");
5332     initBitmapData(ctx->bitmap_memory, 0, ctx->width, ctx->height);
5333 }
5334 
5335 
5336 //Create OS Virtual Function Struct
5337 os_abs init_osx_abs()
5338 {
5339     os_abs abstraction;
5340     abstraction.start_func = &osx_start;
5341     abstraction.loop_start_func = &osx_loop_start;
5342     abstraction.update_func = &osx_enqueue_update;
5343     abstraction.sleep_func = &osx_sleep;
5344     abstraction.get_bitmap_memory_func = &osx_get_bitmap_memory;
5345     abstraction.get_time_mili_func = &osx_get_time_mili;
5346     abstraction.get_width_func = &osx_get_width;
5347     abstraction.get_height_func = &osx_get_height;
5348     abstraction.start_thread_func = &osx_start_thread;
5349     abstraction.draw_weird = &doesnt_work_on_osx;
5350     return abstraction;
5351 }
5352 
5353 
5354 @interface CustomView : NSView
5355 @end
5356 @implementation CustomView
5357 - (void)drawRect:(NSRect)dirtyRect {
5358     CGContextRef gctx = [[NSGraphicsContext currentContext] CGContext];
5359     CGRect myBoundingBox;
5360     myBoundingBox = CGRectMake(0,0, ctx->width, ctx->height);
5361     CGColorSpaceRef colorSpace = CGColorSpaceCreateWithName(kCGColorSpaceGenericRGB);
5362     int bitmapBytesPerRow = ctx->width*4;
5363     static float thingy = 0;
5364     //NOTE: not sure if _backBuffer should be stored?? probably not right.
5365     CGContextRef _backBuffer = CGBitmapContextCreate(ctx->bitmap_memory, ctx->width, ctx->height, 8,
5366                                                      bitmapBytesPerRow, colorSpace, kCGImageAlphaPremultipliedLast); //NOTE: nonpremultiplie
5367 
5368     //CGContextSetRGBFillColor(_backBuffer, 0.5, 0.5, 1, 0.1f);
5369     //CGContextFillRect(_backBuffer, CGRectMake(0,40, 800,780));
5370 
5371     CGImageRef backImage = CGBitmapContextCreateImage(_backBuffer);
5372 
5373     //double _color[] = {1.0f,0.0f,1.0f,1.0f};
5374     //CGColorRef color = CGColorCreate(colorSpace, _color);
5375     CGColorSpaceRelease(colorSpace);
5376 
5377     //CGContextSetFillColorWithColor(gctx, color);
5378     //CGContextSetRGBFillColor(gctx, 1, 0.5, 1, 1);
5379     //CGContextFillRect(gctx, CGRectMake(340,40, 480,480));
5380     CGContextDrawImage(gctx, myBoundingBox, backImage);
5381 
5382 
5383     CGContextRelease(_backBuffer);
5384     CGImageRelease(backImage);
5385 }
5386 @end
5387 
5388 @interface AppDelegate : NSObject <NSApplicationDelegate>
5389 @end
5390 @implementation AppDelegate
5391 
5392 - (NSApplicationTerminateReply)applicationShouldTerminate:(NSApplication *)sender
```

```
{
    //exit(0);
    //printf("NUT\n");
    return NSTerminateNow;

}

- (void)applicationDidFinishLaunching:(NSNotification *)notification
{
    //[NSApp stop:nil];
    //printf("NUT Butter\n");
    id menubar = [[NSMenu new] autorelease];
    id appMenuItem = [[NSMenuItem new] autorelease];
    [menubar addItem:appMenuItem];
    [NSApp setMainMenu:menubar];
    id appMenu = [[NSMenu new] autorelease];
    id appName = [[NSProcessInfo processInfo] processName];
    id quitTitle = [@"Quit " stringByAppendingString:appName];
    id quitMenuItem = [[[NSMenuItem alloc] initWithTitle:quitTitle
                                                  action:@selector(terminate:) keyEquivalent:@"q"] autorelease];
    [appMenu addItem:quitMenuItem];
    [appMenuItem setSubmenu:appMenu];
    NSRect frame = NSMakeRect(0, 0, ctx->width, ctx->height);
    NSUInteger windowStyle = NSWindowStyleMaskTitled;//NSWindowStyleMaskBorderless;
    NSWindow* window  = [[[NSWindow alloc]
                          initWithContentRect:frame
                                    styleMask:windowStyle
                                      backing:NSBackingStoreBuffered
                                        defer:NO] autorelease];

    [window setBackgroundColor:[NSColor grayColor]];
    [window makeKeyAndOrderFront:nil];
    [window cascadeTopLeftFromPoint:NSMakePoint(20,20)];

    //NSSize size = NSMakeSize(ctx->width, ctx->height);

    //NSImageView* imageView = [[NSImageView alloc] initWithFrame:frame];
    /*NSBitmapImageRep* bitmap = [[NSBitmapImageRep alloc] initWithBitmapDataPlanes:NULL
                                                                        pixelsWide:800
                                                                        pixelsHigh:800
                                                                     bitsPerSample:8
                                                                   samplesPerPixel:4
                                                                          hasAlpha:YES
                                                                          isPlanar:NO
                                                                    colorSpaceName:NSDeviceRGBColorSpace
                                                                      bitmapFormat:NSBitmapFormatAlphaNonpremultiplied
                                                                       bytesPerRow:0
                                                                       bitsPerPixel:0];*/



    //ctx->bitmap_memory = [bitmap bitmapData];
    //ctx->bitmap = bitmap;
    //NSImage *myImage = [[NSImage alloc] initWithSize:size];
    //[myImage addRepresentation:bitmap];
    //myImage.cacheMode = NSImageCacheNever;
    CustomView* cv = [[CustomView alloc] initWithFrame:frame];
    // [imageView setImage:myImage];

    //NSTextView * textView = [[NSTextView alloc] initWithFrame:frame];
    [window setContentView:cv];

    initBitmapData(ctx->bitmap_memory, 0, ctx->width, ctx->height);
    //[cv drawRect:NSMakeRect(0,0,800,800)];
    //imageView.editable = NO;


}
@end

void osx_start()
{
    printf("Initialising OSX context.\n");
    ctx = (osx_ctx*) malloc(sizeof(osx_ctx));

    ctx->width  = 800;
    ctx->height = 800;
    ctx->main_queue = dispatch_get_main_queue();
    ctx->bitmap_memory = malloc(ctx->width*ctx->height*sizeof(int));

    [NSApplication sharedApplication];
    [NSApp setActivationPolicy:NSApplicationActivationPolicyRegular];
    NSApp.delegate = [AppDelegate alloc];
}

void osx_loop_start()
{
```

```
5480     printf("Starting OSX Run loop.\n");
5481
5482     //printf("starting\n");
5483     [NSApp activateIgnoringOtherApps:YES];
5484     //[NSApp.delegate start];
5485     [NSApp run];
5486 }
5487
5488 void osx_start_thread(void (*func)(void*), void* data)
5489 {
5490     pthread_t thread;
5491     pthread_create(&thread, NULL, (void *(*)(void*))func, data);
5492 }
5493 float offset;
5494 void osx_enqueue_update() //TODO: implement, re-blit the bitmap
5495 {
5496     //return;
5497     dispatch_async(ctx->main_queue,
5498                    ^{
5499                        NSApp.windows[0].title =
5500                            [NSString stringWithFormat:@"Pathtracer %f", offset];
5501                        CustomView* view = (CustomView*) NSApp.windows[0].contentView;
5502                        //NSImageView* test_img_view = (NSImageView*) test_view;
5503
5504                        //[test_img_view.image recache];
5505
5506                        // BULLSHIT START
5507                        //[test_img_view.image lockFocus];
5508                        //[test_img_view.image unlockFocus];
5509                        // BULLSHIT END
5510                        //[view lockFocus];
5511                        //[view drawRect:NSMakeRect(0,0,800,800)];
5512                        //[view unlockFocus];
5513                        [view setNeedsDisplay:YES];
5514
5515                        [NSApp.windows[0] display]; //This should also call display on view
5516                    });
5517 }
5518
5519 void _test_thing(void* data)
5520 {
5521     //osx_sleep(500);
5522     offset = 40.0f;
5523     printf("test start\n");
5524     while(true)
5525     {
5526         osx_sleep(1);
5527         initBitmapData(ctx->bitmap_memory, offset, ctx->width, ctx->height);
5528         osx_enqueue_update();
5529         offset += 10.0f;
5530         if(offset>300)
5531             offset = 0;
5532         printf("test loop\n");
5533     }
5534 }
5535
5536 #if 0
5537 int main ()
5538 {
5539     osx_start();
5540
5541     //[NSApplication sharedApplication];
5542     //[NSApp setActivationPolicy:NSApplicationActivationPolicyRegular];
5543     //NSApp.delegate = [AppDelegate alloc];
5544
5545
5546     //NSWindowController * windowController = [[NSWindowController alloc] initWithWindow:window];
5547     //[windowController autorelease];
5548     //osx_start_thread(_test_thing, NULL);
5549     osx_loop_start();
5550
5551     //[NSApp activateIgnoringOtherApps:YES];
5552     //[NSApp run];
5553
5554
5555     return 0;
5556 }
5557 #endif
5558 /*********/
5559 /* Types */
5560 /*********/
5561
5562 #define MESH_SCENE_DATA_PARAM image1d_buffer_t indices, image1d_buffer_t vertices, image1d_buffer_t normals
5563 #define MESH_SCENE_DATA       indices, vertices, normals
5564
5565 typedef struct //16 bytes
5566 {
```

```
5567     vec3 colour;
5568 
5569     float reflectivity;
5570 } __attribute__ ((aligned (16))) material;
5571 
5572 typedef struct
5573 {
5574     vec3 orig;
5575     vec3 dir;
5576 } ray;
5577 
5578 typedef struct
5579 {
5580     bool did_hit;
5581     vec3 normal;
5582     vec3 point;
5583     float dist;
5584     material mat;
5585 } collision_result;
5586 
5587 typedef struct //32 bytes (one word)
5588 {
5589     vec3 pos;
5590     //4 bytes padding
5591     float radius;
5592     int material_index;
5593     //8 bytes padding
5594 } __attribute__ ((aligned (16))) sphere;
5595 
5596 typedef struct plane
5597 {
5598     vec3 pos;
5599     vec3 normal;
5600 
5601     int material_index;
5602 } __attribute__ ((aligned (16))) plane;
5603 
5604 typedef struct
5605 {
5606 
5607     mat4 model;
5608 
5609     vec3 max;
5610     vec3 min;
5611 
5612     int index_offset;
5613     int num_indices;
5614 
5615 
5616     int material_index;
5617 } __attribute__((aligned (32))) mesh; //TODO: align with cpu NOTE: I don't think we need 32
5618 
5619 typedef struct
5620 {
5621     const __global material* material_buffer;
5622     const __global sphere* spheres;
5623     const __global plane* planes;
5624     //Mesh
5625     const __global mesh* meshes;
5626 } scene;
5627 
5628 bool getTBoundingBox(vec3 vmin, vec3 vmax,
5629                      ray r, float* tmin, float* tmax) //NOTE: could be wrong
5630 {
5631 
5632     vec3 invD = 1/r.dir;///vec3(1/dir.x, 1/dir.y, 1/dir.z);
5633     vec3 t0s = (vmin - r.orig) * invD;
5634       vec3 t1s = (vmax - r.orig) * invD;
5635 
5636       vec3 tsmaller = min(t0s, t1s);
5637     vec3 tbigger  = max(t0s, t1s);
5638 
5639     *tmin = max(*tmin, max(tsmaller.x, max(tsmaller.y, tsmaller.z)));
5640     *tmax = min(*tmax, min(tbigger.x,  min(tbigger.y, tbigger.z)));
5641 
5642     return (*tmin < *tmax);
5643 
5644     /* vec3 tmin = (vmin - r.orig) / r.dir; */
5645     /* vec3 tmax = (vmax - r.orig) / r.dir; */
5646 
5647     /* vec3 real_min = min(tmin, tmax); */
5648     /* vec3 real_max = max(tmin, tmax); */
5649 
5650     /* vec3 minmax = min(min(real_max.x, real_max.y), real_max.z); */
5651     /* vec3 maxmin = max(max(real_min.x, real_min.y), real_min.z); */
5652 
5653     /* if (dot(minmax,minmax) >= dot(maxmin, maxmin)) */
```

```
5654     /* { */
5655     /*     *t_min = sqrt(dot(maxmin,maxmin)); */
5656     /*     *t_max = sqrt(dot(minmax,minmax)); */
5657     /*     return (dot(maxmin, maxmin) > 0.001f ? true : false); */
5658     /* } */
5659     /* else return false; */
5660 }
5661
5662
5663 bool hitBoundingBox(vec3 vmin, vec3 vmax,
5664                     ray r)
5665 {
5666     vec3 tmin = (vmin - r.orig) / r.dir;
5667     vec3 tmax = (vmax - r.orig) / r.dir;
5668
5669     vec3 real_min = min(tmin, tmax);
5670     vec3 real_max = max(tmin, tmax);
5671
5672     vec3 minmax = min(min(real_max.x, real_max.y), real_max.z);
5673     vec3 maxmin = max(max(real_min.x, real_min.y), real_min.z);
5674
5675     if (dot(minmax,minmax) >= dot(maxmin, maxmin))
5676     { return (dot(maxmin, maxmin) > 0.001f ? true : false); }
5677     else return false;
5678 }
5679
5680
5681
5682 /*********************/
5683 /*                   */
5684 /*     Primitives    */
5685 /*                   */
5686 /*********************/
5687
5688 /************/
5689 /* Triangle */
5690 /************/
5691
5692 //Moller-Trumbore
5693 //t u v = x y z
5694 bool does_collide_triangle(vec3 tri[4], vec3* hit_coords, ray r) //tri has extra for padding
5695 {
5696
5697     vec3 ab = tri[1] - tri[0];
5698     vec3 ac = tri[2] - tri[0];
5699
5700     vec3 pvec = cross(r.dir, ac); //Triple product
5701     float det = dot(ab, pvec);
5702
5703     if (det < EPSILON) // Behind or close to parallel.
5704         return false;
5705
5706     float invDet = 1.f / det;
5707     vec3 tvec = r.orig - tri[0];
5708
5709     //u
5710     hit_coords->y = dot(tvec, pvec) * invDet;
5711     if(hit_coords->y < 0 || hit_coords->y > 1)
5712         return false;
5713
5714     //v
5715     vec3 qvec = cross(tvec, ab);
5716     hit_coords->z = dot(r.dir, qvec) * invDet;
5717     if (hit_coords->z < 0 || hit_coords->y + hit_coords->z > 1)
5718         return false;
5719
5720     //t
5721     hit_coords->x = dot(ac, qvec) * invDet;
5722
5723
5724     return true; //goose
5725 }
5726
5727
5728 /**********/
5729 /* Sphere */
5730 /**********/
5731
5732 bool does_collide_sphere(sphere s, ray r, float *dist)
5733 {
5734     float t0, t1; // solutions for t if the ray intersects
5735
5736     // analytic solution
5737     vec3 L = s.pos- r.orig;
5738     float b = dot(r.dir, L) ;//* 2.0f;
5739     float c = dot(L, L) - (s.radius*s.radius); //NOTE: you can optimize out the square.
5740
```

```
5741     float disc = b * b - c/**a*/; /* discriminant of quadratic formula */
5742
5743     /* solve for t (distance to hitpoint along ray) */
5744     float t = false;
5745
5746     if (disc < 0.0f) return false;
5747     else t = b - sqrt(disc);
5748
5749     if (t < 0.0f)
5750     {
5751         t = b + sqrt(disc);
5752         if (t < 0.0f) return false;
5753     }
5754     *dist = t;
5755     return true;
5756 }
5757
5758
5759
5760 /*********/
5761 /* Plane */
5762 /*********/
5763
5764 bool does_collide_plane(plane p, ray r, float *dist)
5765 {
5766     float denom = dot(r.dir, p.normal);
5767     if (denom < EPSILON) //Counter intuitive.
5768     {
5769         vec3 l = p.pos - r.orig;
5770         float t = dot(l, p.normal) / denom;
5771         if (t >= 0)
5772         {
5773             *dist = t;
5774             return true;
5775         }
5776
5777     }
5778     return false;
5779 }
5780
5781
5782 /********************/
5783 /*                  */
5784 /*      Meshes      */
5785 /*                  */
5786 /********************/
5787
5788
5789 bool does_collide_with_mesh(mesh collider, ray r, vec3* normal, float* dist, scene s,
5790                             MESH_SCENE_DATA_PARAM)
5791 {
5792     //TODO: k-d trees
5793     *dist = FAR_PLANE;
5794     float min_t = FAR_PLANE;
5795     vec3 hit_coord; //NOTE: currently unused
5796     ray r2 = r;
5797     if(!hitBoundingBox(collider.min, collider.max, r))
5798     {
5799         return false;
5800     }
5801
5802     for(int i = 0; i < collider.num_indices/3; i++) // each ivec3
5803     {
5804         vec3 tri[4];
5805
5806         //get vertex (first element of each index)
5807
5808         int4 idx_0 = read_imagei(indices, i*3+collider.index_offset+0);
5809         int4 idx_1 = read_imagei(indices, i*3+collider.index_offset+1);
5810         int4 idx_2 = read_imagei(indices, i*3+collider.index_offset+2);
5811
5812         tri[0] = read_imagef(vertices, idx_0.x).xyz;
5813         tri[1] = read_imagef(vertices, idx_1.x).xyz;
5814         tri[2] = read_imagef(vertices, idx_2.x).xyz;
5815
5816         //printf("%d/%d: (%f, %f, %f)\n", idx_0.x, collider.num_indices/3, tri[0].x, tri[0].y, tri[0].z);
5817         //printf("%d/%d: (%f, %f, %f)\n", idx_1.x, collider.num_indices/3, tri[1].x, tri[1].y, tri[1].z);
5818
5819         vec3 bc_hit_coords = (vec3)(0.f); //t u v = x y z
5820         if(does_collide_triangle(tri, &bc_hit_coords, r) &&
5821            bc_hit_coords.x<min_t && bc_hit_coords.x>0)
5822         {
5823             min_t = bc_hit_coords.x; //t (distance along direction)
5824             *normal =
5825                 read_imagef(normals, idx_0.y).xyz*(1-bc_hit_coords.y-bc_hit_coords.z)+
5826                 read_imagef(normals, idx_1.y).xyz*bc_hit_coords.y+
5827                 read_imagef(normals, idx_2.y).xyz*bc_hit_coords.z;
```

```
5828                 //break; //convex optimization
5829         }
5830
5831     }
5832
5833
5834     *dist = min_t;
5835     return min_t != FAR_PLANE;
5836
5837 }
5838
5839 bool does_collide_with_mesh_nieve(mesh collider, ray r, vec3* normal, float* dist, scene s,
5840                                   image1d_buffer_t tree, MESH_SCENE_DATA_PARAM)
5841 {
5842         //TODO: k-d trees
5843     *dist = FAR_PLANE;
5844     float min_t = FAR_PLANE;
5845     vec3 hit_coord; //NOTE: currently unused
5846     ray r2 = r;
5847     if(!hitBoundingBox(collider.min, collider.max, r))
5848     {
5849         return false;
5850     }
5851
5852     for(int i = 0; i < collider.num_indices/3; i++) // each ivec3
5853     {
5854         vec3 tri[4];
5855
5856         //get vertex (first element of each index)
5857
5858         int4 idx_0 = read_imagei(indices, i*3+collider.index_offset+0);
5859         int4 idx_1 = read_imagei(indices, i*3+collider.index_offset+1);
5860         int4 idx_2 = read_imagei(indices, i*3+collider.index_offset+2);
5861
5862         tri[0] = read_imagef(vertices, idx_0.x).xyz;
5863         tri[1] = read_imagef(vertices, idx_1.x).xyz;
5864         tri[2] = read_imagef(vertices, idx_2.x).xyz;
5865
5866         //printf("%d/%d: (%f, %f, %f)\n", idx_0.x, collider.num_indices/3, tri[0].x, tri[0].y, tri[0].z);
5867         //printf("%d/%d: (%f, %f, %f)\n", idx_1.x, collider.num_indices/3, tri[1].x, tri[1].y, tri[1].z);
5868
5869         vec3 bc_hit_coords = (vec3)(0.f); //t u v = x y z
5870         if(does_collide_triangle(tri, &bc_hit_coords, r) &&
5871            bc_hit_coords.x<min_t && bc_hit_coords.x>0)
5872         {
5873             min_t = bc_hit_coords.x; //t (distance along direction)
5874             *normal =
5875                 read_imagef(normals, idx_0.y).xyz*(1-bc_hit_coords.y-bc_hit_coords.z)+
5876                 read_imagef(normals, idx_1.y).xyz*bc_hit_coords.y+
5877                 read_imagef(normals, idx_2.y).xyz*bc_hit_coords.z;
5878                 //break; //convex optimization
5879         }
5880
5881     }
5882
5883
5884     *dist = min_t;
5885     return min_t != FAR_PLANE;
5886 }
5887
5888 bool does_collide_with_mesh_alt(mesh collider, ray r, vec3* normal, float* dist, scene s,
5889                                 MESH_SCENE_DATA_PARAM)
5890 {
5891     *dist = FAR_PLANE;
5892     float min_t = FAR_PLANE;
5893     vec3 hit_coord; //NOTE: currently unused
5894     ray r2 = r;
5895
5896     for(int i = 0; i < SCENE_NUM_INDICES/3; i++)
5897     {
5898         vec3 tri[4];
5899
5900         //get vertex (first element of each index)
5901
5902         int4 idx_0 = read_imagei(indices, i*3+collider.index_offset+0);
5903         int4 idx_1 = read_imagei(indices, i*3+collider.index_offset+1);
5904         int4 idx_2 = read_imagei(indices, i*3+collider.index_offset+2);
5905
5906         tri[0] = read_imagef(vertices, idx_0.x).xyz;
5907         tri[1] = read_imagef(vertices, idx_1.x).xyz;
5908         tri[2] = read_imagef(vertices, idx_2.x).xyz;
5909
5910
5911         vec3 bc_hit_coords = (vec3)(0.f); //t u v = x y z
5912         if(does_collide_triangle(tri, &bc_hit_coords, r) &&
5913            bc_hit_coords.x<min_t && bc_hit_coords.x>0)
5914         {
```

```
5915                 min_t = bc_hit_coords.x; //t (distance along direction)
5916                 *normal =
5917                     read_imagef(normals, idx_0.y).xyz*(1-bc_hit_coords.y-bc_hit_coords.z)+
5918                     read_imagef(normals, idx_1.y).xyz*bc_hit_coords.y+
5919                     read_imagef(normals, idx_2.y).xyz*bc_hit_coords.z;
5920         }
5921 
5922     }
5923 
5924 
5925     *dist = min_t;
5926     return min_t != FAR_PLANE;
5927 
5928 }
5929 
5930 
5931 
5932 /************************/
5933 /* High Level Collision */
5934 /************************/
5935 
5936 
5937 bool collide_meshes(ray r, collision_result* result, scene s, MESH_SCENE_DATA_PARAM)
5938 {
5939 
5940     float dist = FAR_PLANE;
5941     result->did_hit = false;
5942     result->dist = FAR_PLANE;
5943 
5944     for(int i = 0; i < SCENE_NUM_MESHES; i++)
5945     {
5946         mesh current_mesh = s.meshes[i];
5947         float local_dist = FAR_PLANE;
5948         vec3 normal;
5949         if(does_collide_with_mesh(current_mesh, r, &normal,  &local_dist, s, MESH_SCENE_DATA))
5950         {
5951 
5952             if(local_dist<dist)
5953             {
5954                 dist = local_dist;
5955                 result->dist = dist;
5956                 result->normal = normal;
5957                 result->point = (r.dir*dist)+r.orig;
5958                 result->mat = s.material_buffer[current_mesh.material_index];
5959                 result->did_hit = true;
5960             }
5961         }
5962     }
5963     return result->did_hit;
5964 }
5965 
5966 bool collide_primitives(ray r, collision_result* result, scene s)
5967 {
5968 
5969     float dist = FAR_PLANE;
5970     result->did_hit = false;
5971     result->dist = FAR_PLANE;
5972     for(int i = 0; i < SCENE_NUM_SPHERES; i++)
5973     {
5974         sphere current_sphere = s.spheres[i];//get_sphere(spheres, i);
5975         float local_dist = FAR_PLANE;
5976         if(does_collide_sphere(current_sphere, r, &local_dist))
5977         {
5978             if(local_dist<dist)
5979             {
5980                 dist = local_dist;
5981                 result->did_hit = true;
5982                 result->dist    = dist;
5983                 result->point   = r.dir*dist+r.orig;
5984                 result->normal  = normalize(result->point - current_sphere.pos);
5985                 result->mat     = s.material_buffer[current_sphere.material_index];
5986             }
5987         }
5988     }
5989 
5990     for(int i = 0; i < SCENE_NUM_PLANES; i++)
5991     {
5992         plane current_plane = s.planes[i];//get_plane(planes, i);
5993         float local_dist =  FAR_PLANE;
5994         if(does_collide_plane(current_plane, r, &local_dist))
5995         {
5996             if(local_dist<dist)
5997             {
5998                 dist = local_dist;
5999                 result->did_hit = true;
6000                 result->dist    = dist;
6001                 result->point   = r.dir*dist+r.orig;
```

```
6002                 result->normal  = current_plane.normal;
6003                 result->mat     = s.material_buffer[current_plane.material_index];
6004             }
6005         }
6006     }
6007
6008     return dist != FAR_PLANE;
6009 }
6010
6011 bool collide_all(ray r, collision_result* result, scene s, MESH_SCENE_DATA_PARAM)
6012 {
6013     float dist = FAR_PLANE;
6014     if(collide_primitives(r, result, s))
6015         dist = result->dist;
6016
6017     collision_result m_result;
6018     if(collide_meshes(r, &m_result, s, MESH_SCENE_DATA))
6019         if(m_result.dist < dist)
6020             *result = m_result;
6021
6022     return result->did_hit;
6023 }
6024 /****************************************/
6025 /* NOTE: Irradiance Caching is Incomplete */
6026 /****************************************/
6027
6028 /**********************/
6029 /* Irradiance Caching */
6030 /**********************/
6031
6032 __kernel void ic_hemisphere_sample(
6033
6034     )
6035 {
6036
6037
6038
6039 }
6040
6041 __kernel void ic_screen_textures(
6042     __write_only image2d_t pos_tex,
6043     __write_only image2d_t nrm_tex,
6044     const unsigned int width,
6045     const unsigned int height,
6046     const __global float* ray_buffer,
6047     const vec4 pos,
6048     const __global material* material_buffer,
6049     const __global sphere* spheres,
6050     const __global plane* planes,
6051     const __global mesh* meshes,
6052     image1d_buffer_t indices,
6053     image1d_buffer_t vertices,
6054     image1d_buffer_t normals)
6055 {
6056     scene s;
6057     s.material_buffer = material_buffer;
6058     s.spheres         = spheres;
6059     s.planes          = planes;
6060     s.meshes          = meshes;
6061
6062
6063     int id = get_global_id(0);
6064     int x  = id%width;
6065     int y  = id/width;
6066     int offset = x+y*width;
6067     int ray_offset = offset*3;
6068
6069     ray r;
6070     r.orig = pos.xyz; //NOTE: slow unaligned memory access.
6071     r.dir.x = ray_buffer[ray_offset];
6072     r.dir.y = ray_buffer[ray_offset+1];
6073     r.dir.z = ray_buffer[ray_offset+2];
6074
6075     collision_result result;
6076     if(!collide_all(r, &result, s, MESH_SCENE_DATA))
6077     {
6078         write_imagef(pos_tex, (int2)(x,y), (vec4)(0));
6079         write_imagef(nrm_tex, (int2)(x,y), (vec4)(0));
6080         return;
6081     }
6082
6083     write_imagef(pos_tex, (int2)(x,y), (vec4)(result.point,0)); //Maybe ???
6084     write_imagef(nrm_tex, (int2)(x,y), (vec4)(result.normal,0));
6085
6086     /* pos_tex[offset] = (vec4)(result.point,0); */
6087     /* nrm_tex[offset] = (vec4)(result.normal,0); */
6088 }
```

```
6089
6090
6091
6092 __kernel void generate_discontinuity(
6093     image2d_t pos_tex,
6094     image2d_t nrm_tex,
6095     __global float* out_tex,
6096     const float k,
6097     const float intensity,
6098     const unsigned int width,
6099     const unsigned int height)
6100 {
6101     int id = get_global_id(0);
6102     int x  = id%width;
6103     int y  = id/width;
6104     int offset = x+y*width;
6105
6106     //NOTE: this is fine for edges because the sampler is clamped
6107
6108     //Positions
6109     vec4 pm = read_imagef(pos_tex, sampler, (int2)(x,y));
6110     vec4 pu = read_imagef(pos_tex, sampler, (int2)(x,y+1));
6111     vec4 pd = read_imagef(pos_tex, sampler, (int2)(x,y-1));
6112     vec4 pr = read_imagef(pos_tex, sampler, (int2)(x+1,y));
6113     vec4 pl = read_imagef(pos_tex, sampler, (int2)(x-1,y));
6114
6115     //NOTE: slow doing this many distance calculations
6116     float posDiff = max(distance(pu,pm),
6117                         max(distance(pd,pm),
6118                             max(distance(pr,pm),
6119                                 distance(pl,pm))));
6120     posDiff = clamp(posDiff, 0.f, 1.f);
6121     posDiff *= intensity;
6122
6123     //Normals
6124     vec4 nm = read_imagef(nrm_tex, sampler, (int2)(x,y));
6125
6126     vec4 nu = read_imagef(nrm_tex, sampler, (int2)(x,y+1));
6127     vec4 nd = read_imagef(nrm_tex, sampler, (int2)(x,y-1));
6128     vec4 nr = read_imagef(nrm_tex, sampler, (int2)(x+1,y));
6129     vec4 nl = read_imagef(nrm_tex, sampler, (int2)(x-1,y));
6130     //NOTE: slow doing this many distance calculations
6131     float nrmDiff = max(distance(nu,nm),
6132                         max(distance(nd,nm),
6133                             max(distance(nr,nm),
6134                                 distance(nl,nm))));
6135     nrmDiff = clamp(nrmDiff, 0.f, 1.f);
6136     nrmDiff *= intensity;
6137
6138     out_tex[offset] = k*nrmDiff+posDiff;
6139 }
6140
6141 __kernel void float_average(
6142     __global float* in_tex,
6143     __global float* out_tex,
6144     const unsigned int width,
6145     const unsigned int height,
6146     const int total)
6147 {
6148     int id = get_global_id(0);
6149     int x  = id%width;
6150     int y  = id/width;
6151     int offset = x+y*width;
6152
6153     out_tex[offset] += in_tex[offset]/(float)total;
6154
6155 }
6156
6157
6158 __kernel void mip_single_upsample( //nearest neighbour upsample.
6159     __global float* in_tex,
6160     __global float* out_tex,
6161     const unsigned int width, //Of upsampled
6162     const unsigned int height)//Of upsampled
6163 {
6164     int id = get_global_id(0);
6165     int x  = id%width;
6166     int y  = id/width;
6167     int offset = x+y*width;
6168
6169     out_tex[offset] = in_tex[(x+y*width)/2]; //truncated
6170 }
6171
6172 __kernel void mip_upsample( //nearest neighbour upsample.
6173     image2d_t in_tex,
6174     __write_only image2d_t out_tex, //NOTE: not having __write_only caused it to crash without err
6175     const unsigned int width, //Of upsampled
```

```
    const unsigned int height)//Of upsampled
{
    int id = get_global_id(0);
    int x  = id%width;
    int y  = id/width;

    write_imagef(out_tex, (int2)(x,y),
                 read_imagef(in_tex, sampler, (float2)((float)x/2.f, (float)y/2.f)));
}

__kernel void mip_upsample_scaled( //nearest neighbour upsample.
    image2d_t in_tex,
    __write_only image2d_t out_tex,
    const int s,
    const unsigned int width, //Of upsampled
    const unsigned int height)//Of upsampled
{
    int id = get_global_id(0);
    int x  = id%width;
    int y  = id/width;
    float factor = pow(2.f, (float)s);
    write_imagef(out_tex, (int2)(x,y),
                 read_imagef(in_tex, sampler, (float2)((float)x/factor, (float)y/factor)));
}
__kernel void mip_single_upsample_scaled( //nearest neighbour upsample.
    __global float* in_tex,
    __global float* out_tex,
    const unsigned int s,
    const unsigned int width, //Of upsampled
    const unsigned int height)//Of upsampled
{
    int id = get_global_id(0);
    int x  = id%width;
    int y  = id/width;
    int factor = (int) pow(2.f, (float)s);
    int offset = x+y*width;
    int fwidth = width/factor;
    int fheight = height/factor;

    out_tex[offset] = in_tex[(x/factor)+(y/factor)*(width/factor)]; //truncated
}

//NOTE: not used
__kernel void mip_reduce( //not the best
    image2d_t in_tex,
    __write_only image2d_t out_tex,
    const unsigned int width, //Of reduced
    const unsigned int height)//Of reduced
{
    int id = get_global_id(0);
    int x  = id%width;
    int y  = id/width;



    vec4 p00 = read_imagef(in_tex, sampler, (int2)(x*2,   y*2  ));

    vec4 p01 = read_imagef(in_tex, sampler, (int2)(x*2+1, y*2  ));

    vec4 p10 = read_imagef(in_tex, sampler, (int2)(x*2,   y*2+1));

    vec4 p11 = read_imagef(in_tex, sampler, (int2)(x*2+1, y*2+1));

    write_imagef(out_tex, (int2)(x,y), p00+p01+p10+p11/4.f);
}
#define KDTREE_LEAF 1
#define KDTREE_NODE 2

//TODO: put in util
#define DEBUG
#ifdef DEBUG
//NOTE: this will be slow.
#define assert(x)                                                       \
    if (! (x))                                                          \
    {                                                                   \
        int i = 0;while(i++ < 100)printf("Assert(%s) failed in %s:%d\n", #x, __FUNCTION__, __LINE__); \
        return;                                                         \
    }
#else
#define assert(x)  //NOTHING
#endif
typedef struct
{
    uchar type;

    uint num_triangles;
} __attribute__ ((aligned (16))) kd_tree_leaf_template;
```

```
6263
6264 typedef struct
6265 {
6266     uchar type;
6267
6268     uint num_triangles;
6269     ulong triangle_start;
6270 } kd_tree_leaf;
6271
6272 typedef struct
6273 {
6274     uchar type;
6275     uchar k;
6276     float b;
6277
6278     ulong left_index;
6279     ulong right_index;
6280 } __attribute__ ((aligned (16))) kd_tree_node;
6281
6282
6283 typedef union a_vec3
6284 {
6285     vec3 v;
6286     float a[4];
6287 } a_vec3;
6288
6289 typedef struct kd_stack_elem
6290 {
6291     ulong node;
6292     float min;
6293     float max;
6294 } kd_stack_elem;
6295
6296 typedef __global uint4* kd_44_matrix;
6297
6298
6299 void kd_update_state(__global long* kd_tree, ulong indx, uchar* type,
6300                      kd_tree_node* node, kd_tree_leaf* leaf)
6301 {
6302
6303
6304     *type = *((__global uchar*)(kd_tree+indx));
6305
6306     if(*type == KDTREE_LEAF)
6307     {
6308         kd_tree_leaf_template template = *((__global kd_tree_leaf_template*) (kd_tree + indx));
6309         leaf->type = template.type;
6310         leaf->num_triangles = template.num_triangles;
6311
6312         leaf->triangle_start = indx + sizeof(kd_tree_leaf_template)/8;
6313     }
6314     else
6315         *node = *((__global kd_tree_node*) (kd_tree + indx));
6316
6317 }
6318
6319 void dbg_print_node(kd_tree_node n)
6320 {
6321     printf("\nNODE: type: %u, k: %u, b: %f, l: %llu, r: %llu \n",
6322            (unsigned int) n.type, (unsigned int) n.k, n.b,
6323            n.left_index, n.right_index);
6324 }
6325
6326 void dbg_print_matrix(kd_44_matrix m)
6327 {
6328     printf("[%2u %2u %2u %2u]\n" \
6329            "[%2u %2u %2u %2u]\n" \
6330            "[%2u %2u %2u %2u]\n" \
6331            "[%2u %2u %2u %2u]\n\n",
6332            m[0].x, m[0].y, m[0].z, m[0].w,
6333            m[1].x, m[1].y, m[1].z, m[1].w,
6334            m[2].x, m[2].y, m[2].z, m[2].w,
6335            m[3].x, m[3].y, m[3].z, m[3].w);
6336 }
6337
6338 inline float get_elem(vec3 v, uchar k, kd_44_matrix mask)
6339 {
6340     k = min(k,(uchar)2);
6341     vec3 nv = select((vec3)(0), v, mask[k].xyz);//NOTE: it has to be MSB on the mask
6342
6343     return nv.x + nv.y + nv.z;
6344 }
6345
6346 #define BLOCKSIZE_Y 1
6347 #define STACK_SIZE 16 //tune later
6348 #define LOAD_BALANCER_BATCH_SIZE        32*3
6349
```

```
6350 //#define BLOCKSIZE_Y 1 //NOTE: TEST
6351 __kernel void kdtree_intersection(
6352     __global kd_tree_collision_result* out_buf,
6353     __global ray* ray_buffer, //TODO: make vec4
6354 
6355     __global uint* dumb_data, //NOTE: REALLY DUMB, you can't JUST have a global variable in ocl.
6356 
6357 //Mesh
6358     __global mesh* meshes,
6359     image1d_buffer_t     indices,
6360     image1d_buffer_t     vertices,
6361     __global long* kd_tree,   //TODO: use a higher allignment type
6362 
6363     unsigned int num_rays)
6364 {
6365 
6366     const uint blocksize_x = BLOCKSIZE_X; //should be 32 //NOTE: REMOVED A THING
6367     const uint blocksize_y = BLOCKSIZE_Y;
6368 
6369     //NOTE: not technically correct, but kinda is
6370     uint x = get_local_id(0) % BLOCKSIZE_X; //id within the warp
6371     uint y = get_local_id(0) / BLOCKSIZE_X; //id of the warp in the SM
6372 
6373     __local volatile int next_ray_array[BLOCKSIZE_Y];
6374     __local volatile int ray_count_array[BLOCKSIZE_Y];
6375     next_ray_array[y]  = 0;
6376     ray_count_array[y] = 0;
6377     //printf("%llu", get_global_id(0));
6378     //printf("%llu %llu %llu    ", get_local_size(0), get_num_groups(0), get_global_size(0));
6379     kd_stack_elem stack[STACK_SIZE];
6380     uint stack_length = 0;
6381 
6382     //NOTE: IT WAS CRASHING WHEN THE VECTORS WERENT ALLIGNED!!!!
6383     kd_44_matrix elem_mask = (kd_44_matrix)(dumb_data);
6384     __global uint* warp_counter = dumb_data+16;
6385 
6386 
6387     //NOTE: this block of variables is probably pretty bad for the cache
6388     ray           r;
6389     float         t_hit = INFINITY;
6390     vec2          hit_info = (vec2)(0,0);
6391     unsigned int  tri_indx;
6392     float         t_min, t_max;
6393     float         scene_t_min = 0, scene_t_max = INFINITY;
6394     kd_tree_node  node;
6395     kd_tree_leaf  leaf;
6396     uchar         current_type = KDTREE_NODE;
6397     bool          pushdown = false;
6398     kd_tree_node  root;
6399     uint          ray_indx;
6400 
6401     while(true)
6402     {
6403         uint tidx = x; // SINGLE WARPS WORTH OF WORK 0-32
6404         uint widx = y; // WARPS PER SM 0-4 (for example)
6405         __local volatile int* local_pool_ray_count = ray_count_array+widx; //get warp ids pool
6406         __local volatile int* local_pool_next_ray  = next_ray_array+widx;
6407 
6408         //Grab new rays
6409         if(tidx == 0 && *local_pool_ray_count <= 0) //only the first work (of the pool) item gets memory
6410         {
6411             *local_pool_next_ray = atomic_add(warp_counter, LOAD_BALANCER_BATCH_SIZE); //batch complete
6412 
6413 
6414             *local_pool_ray_count = LOAD_BALANCER_BATCH_SIZE;
6415         }
6416         barrier(CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE);
6417 
6418 //lol help there are no barriers
6419         {
6420 
6421             ray_indx = *local_pool_next_ray + tidx;
6422             barrier(CLK_LOCAL_MEM_FENCE);
6423 
6424             if(ray_indx >= num_rays) //ray index is past num rays, work is done
6425                 break;
6426 
6427             if(tidx == 0) //NOTE: this doesn't guarentee
6428             {
6429                 *local_pool_next_ray  += BLOCKSIZE_X;
6430                 *local_pool_ray_count -= BLOCKSIZE_X;
6431             }
6432             barrier(CLK_LOCAL_MEM_FENCE);
6433 
6434             r = ray_buffer[ray_indx];
6435 
6436             t_hit = INFINITY; //infinity
```

```
6437
6438            if(!getTBoundingBox((vec3) SCENE_MIN, (vec3) SCENE_MAX, r, &scene_t_min, &scene_t_max)) //SCENE_MIN is a macro
6439            {
6440                scene_t_max = -INFINITY;
6441            }
6442
6443
6444            t_max = t_min = scene_t_min;
6445
6446            stack_length = 0;
6447            root = *((__global kd_tree_node*) kd_tree);
6448        }
6449        stack_length = 0;
6450        //barrier(CLK_LOCAL_MEM_FENCE);
6451        while(t_max < scene_t_max)
6452        {
6453
6454            if(stack_length == (uint) 0)
6455            {
6456                node  = root; //root
6457                current_type = KDTREE_NODE;
6458                t_min = t_max;
6459                t_max = scene_t_max;
6460                pushdown = true;
6461            }
6462            else
6463            { //pop
6464
6465                t_min = stack[stack_length-1].min;
6466                t_max = stack[stack_length-1].max;
6467                kd_update_state(kd_tree, stack[stack_length-1].node, &current_type, &node, &leaf);
6468
6469                stack_length--;
6470                pushdown = false;
6471            }
6472
6473
6474            while(current_type != KDTREE_LEAF)
6475            {
6476                unsigned char k = node.k;
6477
6478                float t_split = (node.b - get_elem(r.orig, k, elem_mask)) /
6479                                 get_elem(r.dir, k, elem_mask);
6480
6481                bool left_close =
6482                    (get_elem(r.orig, k, elem_mask) < node.b) ||
6483                    (get_elem(r.orig, k, elem_mask) == node.b && get_elem(r.dir, k, elem_mask) <= 0);
6484                ulong thing = left_close ? 0xffffffffffffffff : 0;
6485                ulong first  = select(node.right_index, node.left_index,
6486                                      thing);
6487                ulong second = select(node.left_index, node.right_index,
6488                                      thing);
6489
6490
6491                kd_update_state(kd_tree,
6492                                ( t_split > t_max || t_split <= 0)|||(t_split < t_min) ? first : second,
6493                                &current_type, &node, &leaf);
6494
6495                if( !(t_split > t_max || t_split <= 0) && !(t_split < t_min))
6496                {
6497                    stack[stack_length++] = (kd_stack_elem) {second, t_split, t_max}; //push
6498                    t_max = t_split;
6499                    pushdown = false;
6500                }
6501
6502                /*
6503                if( t_split > t_max || t_split <= 0)  //NOTE: branching necessary
6504                {
6505                    kd_update_state(kd_tree, first, &current_type, &node, &leaf);
6506                }
6507                else if(t_split < t_min)
6508                {
6509                    kd_update_state(kd_tree, second, &current_type, &node, &leaf);
6510                }
6511                else
6512                {
6513                    //assert(stack_length!=(ulong)STACK_SIZE-1);
6514
6515                    stack[stack_length++] = (kd_stack_elem) {second, t_split, t_max}; //push
6516                    kd_update_state(kd_tree, first, &current_type, &node, &leaf);
6517
6518                    t_max = t_split;
6519                    pushdown = false;
6520                    }*/
6521
6522                root = pushdown ? node : root;
6523
```

```
            }
            //barrier(0);
            //Found leaf
            for(ulong t = 0; t <leaf.num_triangles; t++)
            {
                //assert(leaf.triangle_start-t == 0);
                vec3 tri[4];
                unsigned int index_offset =
                    *((__global uint*)(kd_tree+leaf.triangle_start)+t);
                //get vertex (first element of each index)
                const int4 idx_0 = read_imagei(indices, (int)index_offset+0);
                const int4 idx_1 = read_imagei(indices, (int)index_offset+1);
                const int4 idx_2 = read_imagei(indices, (int)index_offset+2);

                tri[0] = read_imagef(vertices, (int)idx_0.x).xyz;
                tri[1] = read_imagef(vertices, (int)idx_1.x).xyz;
                tri[2] = read_imagef(vertices, (int)idx_2.x).xyz;
                /*printf("%f %f %f : %f %f %f : %f %f %f %llu\n",
                       tri[0].x, tri[0].y, tri[0].z,
                       tri[1].x, tri[1].y, tri[1].z,
                       tri[2].x, tri[2].y, tri[2].z,
                       t);*/

                vec3 hit_coords; // t u v
                if(does_collide_triangle(tri, &hit_coords, r)) //TODO: optimize
                {
                    //printf("COLLISION\n");
                    if(hit_coords.x<=0)
                        continue;
                    if(hit_coords.x < t_hit)
                    {
                        t_hit = hit_coords.x;     //t
                        hit_info = hit_coords.yz; //u v
                        tri_indx = index_offset;

                        if(t_hit < t_min) // goes by closest to furthest, so if it hits it will be the closest
                        {//early exit
                            //remove that

                            //scene_t_min = -INFINITY;
                            //scene_t_max = -INFINITY;
                            //break;
                        }

                    }

                }

            }

        }
        //By this point a triangle will have been found.
        kd_tree_collision_result result = {0};

        if(!isinf(t_hit))//if t_hit != INFINITY
        {
            result.triangle_index = tri_indx;
            result.t = t_hit;
            result.u = hit_info.x;
            result.v = hit_info.y;
        }

        out_buf[ray_indx] = result;
    }

}

__kernel void kdtree_ray_draw(
    __global unsigned int* out_tex,
    __global ray* rays,

    const unsigned int width)
{
    const vec4 sky = (vec4) (0.84, 0.87, 0.93, 0);
    //return;
    int id = get_global_id(0);
    int x  = id%width;
    int y  = id/width;
    int offset = x+y*width;

    ray r = rays[offset];

    r.orig = (r.orig+1) / 2;

    out_tex[offset] = get_colour( (vec4) (r.orig,1) );
}

```

```
6611
6612 __kernel void kdtree_test_draw(
6613     __global unsigned int* out_tex,
6614     __global kd_tree_collision_result* kd_results,
6615
6616     const __global material* material_buffer,
6617     //meshes
6618     __global mesh* meshes,
6619
6620     image1d_buffer_t     indices,
6621     image1d_buffer_t     vertices,
6622     image1d_buffer_t     normals,
6623     const unsigned int width)
6624 {
6625     const vec4 sky = (vec4) (0.84, 0.87, 0.93, 0);
6626     //return;
6627     int id = get_global_id(0);
6628     int x  = id%width;
6629     int y  = id/width;
6630     int offset = x+y*width;
6631
6632     kd_tree_collision_result res = kd_results[offset];
6633     if(res.t==0)
6634     {
6635         out_tex[offset] = get_colour( (vec4) (0) );
6636         return;
6637     }
6638     int4 i1 = read_imagei(indices, (int)res.triangle_index);
6639     int4 i2 = read_imagei(indices, (int)res.triangle_index+1);
6640     int4 i3 = read_imagei(indices, (int)res.triangle_index+2);
6641     mesh m = meshes[i1.w];
6642     material mat = material_buffer[m.material_index];
6643
6644     vec3 normal =
6645         read_imagef(normals, (int)i1.y).xyz*(1-res.u-res.v)+
6646         read_imagef(normals, (int)i2.y).xyz*res.u+
6647         read_imagef(normals, (int)i3.y).xyz*res.v;
6648
6649     normal = (normal+1) / 2;
6650
6651     out_tex[offset] = get_colour( (vec4) (normal,1) );
6652 }
6653
6654 //TODO: ADD A THING FOR THIS
6655 //#pragma OPENCL EXTENSION cl_nv_pragma_unroll : enable
6656
6657 vec3 uniformSampleHemisphere(const float r1, const float r2)
6658 {
6659     float sinTheta = sqrt(1 - r1 * r1);
6660     float phi = 2 * M_PI_F * r2;
6661     float x = sinTheta * cos(phi);
6662     float z = sinTheta * sin(phi);
6663     return (vec3)(x, r1, z);
6664 }
6665 vec3 cosineSampleHemisphere(float u1, float u2, vec3 normal)
6666 {
6667     const float r = sqrt(u1);
6668     const float theta = 2.f * M_PI_F * u2;
6669
6670     vec3 w = normal;
6671     vec3 axis = fabs(w.x) > 0.1f ? (vec3)(0.0f, 1.0f, 0.0f) : (vec3)(1.0f, 0.0f, 0.0f);
6672     vec3 u = normalize(cross(axis, w));
6673     vec3 v = cross(w, u);
6674
6675     /* use the coordinte frame and random numbers to compute the next ray direction */
6676     return normalize(u * cos(theta)*r + v*sin(theta)*r + w*sqrt(1.0f - u1));
6677 }
6678
6679 #define NUM_BOUNCES 4
6680 #define NUM_SAMPLES 4
6681
6682 typedef struct spath_progress
6683 {
6684     unsigned int sample_num;
6685     unsigned int bounce_num;
6686     vec3 mask;
6687     vec3 accum_color;
6688 } __attribute__((aligned (16))) spath_progress; //NOTE: space for two more 32 bit dudes
6689
6690 __kernel void segmented_path_trace_init(
6691     __global vec4* out_tex,
6692     __global ray* ray_buffer,
6693     __global ray* ray_origin_buffer,
6694     __global kd_tree_collision_result* kd_results,
6695     __global kd_tree_collision_result* kd_source_results,
6696     __global spath_progress* spath_data,
6697
```

```
    const __global material* material_buffer,

//Mesh
    const __global mesh* meshes,
    image1d_buffer_t indices,
    image1d_buffer_t vertices,
    image1d_buffer_t normals,
    /* const __global vec2* texcoords, */
    const unsigned int width,
    const unsigned int random_value)
{
    const vec4 sky = (vec4) (0.16, 0.2, 0.2, 0)*2;
    int x = get_global_id(0)%width;
    int y = get_global_id(0)/width;
    int offset = (x+y*width);

    kd_tree_collision_result res = kd_results[offset];
    ray r = ray_buffer[offset];
    ray_origin_buffer[offset] = r;
    kd_source_results[offset] = res;

    spath_progress spd;
    spd.mask = (vec3)(1.0f, 1.0f, 1.0f);
    spd.accum_color = (vec3) (0, 0, 0);


    if(res.t==0)
    {
        out_tex[offset] += sky;
        //return;
    }

    unsigned int seed1 = random_value * x;
    unsigned int seed2 = random_value * y;

    //if(spd.bounce_num == 0)
    //    spd.mask *= mat.colour;

#pragma unroll   //NOTE: NVIDIA plugin
    for(int i = 0; i < 7; i++)
        get_random(&seed1, &seed2);


    //MESSY CODE!
    float rand1 = get_random(&seed1, &seed2);
    float rand2 = get_random(&seed1, &seed2);


    int4 i1 = read_imagei(indices, (int)res.triangle_index);
    int4 i2 = read_imagei(indices, (int)res.triangle_index+1);
    int4 i3 = read_imagei(indices, (int)res.triangle_index+2);
    mesh m = meshes[i1.w];
    material mat = material_buffer[m.material_index];
    vec3 pos = r.orig + r.dir*res.t;

    vec3 normal =
        read_imagef(normals, (int)i1.y).xyz*(1-res.u-res.v)+
        read_imagef(normals, (int)i2.y).xyz*res.u+
        read_imagef(normals, (int)i3.y).xyz*res.v;

    spd.mask *= mat.colour;

    ray sr;
    vec3 sample_dir = cosineSampleHemisphere(rand1, rand2, normal);
    sr.orig = pos + normal * 0.0001f; //sweet spot for epsilon
    sr.dir = sample_dir;

    ray_buffer[offset] = sr;
    spath_data[offset] = spd;
}

__kernel void segmented_path_trace(
    __global vec4* out_tex,
    __global ray* ray_buffer,
    __global ray* ray_origin_buffer,
    __global kd_tree_collision_result* kd_results,
    __global kd_tree_collision_result* kd_source_results,
    __global spath_progress* spath_data,

    const __global unsigned int* random_buffer,

    const __global material* material_buffer,

//Mesh
    const __global mesh* meshes,
    image1d_buffer_t indices,
    image1d_buffer_t vertices,
```

```
6785     image1d_buffer_t normals,
6786     /* const __global vec2* texcoords, */
6787     const unsigned int width,
6788     //const unsigned int rwidth,
6789     //const unsigned int soffset,
6790     const unsigned int random_value)
6791 {
6792     const vec4 sky = (vec4) (0.16, 0.2, 0.2, 0);
6793     // int x = (soffset*width)+get_global_id(0)%width;
6794     int x = get_global_id(0)%width;
6795     int y = get_global_id(0)/width;
6796     int offset = (x+y*width);
6797 
6798     spath_progress spd = spath_data[offset];
6799 
6800     if(spd.sample_num==2048) //get this from the cpu
6801     {
6802         ray nr;
6803         nr.orig = (vec3)(0);
6804         nr.dir  = (vec3)(0);
6805         ray_buffer[offset] = nr;
6806         return;
6807     }
6808     kd_tree_collision_result res;
6809     ray r;
6810 
6811     if(spd.bounce_num > NUM_BOUNCES)
6812         printf("SHIT\n");
6813 
6814 
6815     res = kd_results[offset];
6816     r = ray_buffer[offset];
6817     //out_tex[offset] = (vec4) (1,0,1,1);
6818     //return;
6819 
6820 
6821     //RETRIEVE DATA
6822     int4 i1 = read_imagei(indices, (int)res.triangle_index);
6823     int4 i2 = read_imagei(indices, (int)res.triangle_index+1);
6824     int4 i3 = read_imagei(indices, (int)res.triangle_index+2);
6825     mesh m = meshes[i1.w];
6826     material mat = material_buffer[m.material_index];
6827     vec3 pos = r.orig + r.dir*res.t;
6828     //pos = (vec3) (0, 0, -2);
6829 
6830     vec3 normal =
6831         read_imagef(normals, (int)i1.y).xyz*(1-res.u-res.v)+
6832         read_imagef(normals, (int)i2.y).xyz*res.u+
6833         read_imagef(normals, (int)i3.y).xyz*res.v;
6834 
6835     //TODO: BETTER RANDOM PLEASE
6836 
6837     //unsigned int seed1 = x*(1920-x)*((x*x*y*y*random_value)%get_global_id(0));
6838     //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))*
6839 
6840     /* union { */
6841     /*     float f; */
6842     /*     unsigned int ui; */
6843     /* } res2; */
6844 
6845     /* res2.f = (float)random_buffer[offset]*M_PI_F+x;//fill up the mantissa. */
6846     /* unsigned int seed1 = res2.ui + (int)(sin((float)x)*7.1f); */
6847 
6848     /* res2.f = (float)random_buffer[offset]*M_PI_F+y; */
6849     /* unsigned int seed2 = y + (int)(sin((float)res2.ui)*7*3.f); */
6850 
6851     unsigned int seed1 = random_buffer[offset]*random_value;
6852     unsigned int seed2 = random_buffer[offset];
6853 
6854 //printf("%u\n",random_value);
6855 
6856     //if(spd.bounce_num == 0)
6857     //    spd.mask *= mat.colour;
6858 
6859 //#pragma unroll   //NOTE: NVIDIA plugin
6860     for(int i = 0; i < 7; i++)
6861         get_random(&seed1, &seed2);
6862 
6863      //MESSY CODE!
6864     float rand1 = get_random(&seed1, &seed2);
6865     float rand2 = get_random(&seed2, &seed1);
6866 
6867     //out_tex[offset] += (vec4)((vec3)(clamp((rand2*8)-2.f, 0.f, 1.f)), 1.f);
6868     //return;
6869 
6870     ray sr;
6871 
```

```
    vec3 sample_dir = cosineSampleHemisphere(rand1, rand2, normal);
    sr.orig = pos + normal * 0.0001f; //sweet spot for epsilon
    sr.dir = sample_dir;


    //printf("%f help\n", res.t);
    //THE NEXT PART
    if(res.t==0)
    {
        //if(get_global_id(0)==500)
        //printf("SHIT PANT\n");
        spd.bounce_num = NUM_BOUNCES; //TODO: uncomment
        spd.accum_color += spd.mask * sky.xyz;
        //sr.orig = (vec3)(0);
        //sr.dir = (vec3)(0);
    }
    else
    {
        //NOTE: janky emission, if reflectivity is 1 emission is 2 (only for tests)
        spd.accum_color += spd.mask * (float)(mat.reflectivity==1.f)*2.f; //NOTE: JUST ADD EMMISION

        spd.mask *= mat.colour;

        spd.mask *= dot(sr.dir, normal);
    }

    spd.bounce_num++;

    if(spd.bounce_num >= NUM_BOUNCES)
    {
        //if(get_global_id(0)==0)
        //printf("PUSH\n");
        spd.bounce_num = 0;
        spd.sample_num++;
#ifdef _WIN32
        out_tex[offset] += (vec4)(spd.accum_color, 1);
#else
        out_tex[offset] += (vec4)(spd.accum_color.zyx, 1);
#endif
        //START OF NEW


        res = kd_source_results[offset];
        r = ray_origin_buffer[offset];
        spd.mask = (vec3)(1.0f, 1.0f, 1.0f);
        spd.accum_color = (vec3) (0, 0, 0);


        if(res.t==0)
        {
            out_tex[offset] += sky;
            //printf("SHI\n");
            //return;
        }

        i1 = read_imagei(indices, (int)res.triangle_index);
        i2 = read_imagei(indices, (int)res.triangle_index+1);
        i3 = read_imagei(indices, (int)res.triangle_index+2);
        m = meshes[i1.w];
        mat = material_buffer[m.material_index];
        pos = r.orig + r.dir*res.t;
        //pos = (vec3) (0, 0, -2);

        normal =
            read_imagef(normals, (int)i1.y).xyz*(1-res.u-res.v)+
            read_imagef(normals, (int)i2.y).xyz*res.u+
            read_imagef(normals, (int)i3.y).xyz*res.v;

        spd.mask *= mat.colour;
        if( (float)(mat.reflectivity==1.)) //TODO: just add an emmision value in material
        {
            spd.accum_color += spd.mask*2;
        }

        sample_dir = cosineSampleHemisphere(rand1, rand2, normal);
        sr.orig = pos + normal * 0.0001f; //sweet spot for epsilon
        sr.dir = sample_dir;
        //printf("GOOD %f %f %f\n",spd.accum_color.x, spd.accum_color.y, spd.accum_color.z);
    }

    ray_buffer[offset] = sr;

    spath_data[offset] = spd;

}

__kernel void path_trace(
```

```
6959     __global vec4* out_tex,
6960     const __global ray* ray_buffer,
6961     const __global material* material_buffer,
6962     const __global sphere* spheres,
6963     const __global plane* planes,
6964 //Mesh
6965     const __global mesh* meshes,
6966     image1d_buffer_t indices,
6967     image1d_buffer_t vertices,
6968     image1d_buffer_t normals,
6969     /* const __global vec2* texcoords, */
6970     const unsigned int width,
6971     const vec4 pos,
6972     unsigned int magic)
6973 {
6974     scene s;
6975     s.material_buffer = material_buffer;
6976     s.spheres         = spheres;
6977     s.planes          = planes;
6978     s.meshes          = meshes;
6979 
6980 
6981     const vec4 sky = (vec4) (0.16, 0.2, 0.2, 0);
6982     //return;
6983     int x = get_global_id(0);
6984     int y = get_global_id(1);
6985     //int x  = id%width+ get_global_offset(0)%total_width;
6986     //int y  = id/width/* + get_global_offset(0)/total_width*/;
6987     int offset = (x+y*width);
6988     //int ray_offset = offset; //NOTE: unnecessary w/ new rays
6989 
6990     ray r;
6991     r = ray_buffer[offset];
6992     r.orig = pos.xyz;
6993     union {
6994         float f;
6995         unsigned int ui;
6996     } res;
6997 
6998     res.f = (float)magic*M_PI_F+x;//fill up the mantissa.
6999     unsigned int seed1 = res.ui + (int)(sin((float)x)*7.1f);
7000 
7001     res.f = (float)magic*M_PI_F+y;
7002     unsigned int seed2 = y + (int)(sin((float)res.ui)*7*3.f);
7003 
7004     collision_result initial_result;
7005     if(!collide_all(r, &initial_result, s, MESH_SCENE_DATA))
7006     {
7007         out_tex[x+y*width] = sky;
7008         return;
7009     }
7010     barrier(0); //good ?
7011 
7012     vec3 fin_colour = (vec3)(0.0f, 0.0f, 0.0f);
7013     for(int i = 0; i < NUM_SAMPLES; i++)
7014     {
7015 
7016         vec3 accum_color = (vec3)(0.0f, 0.0f, 0.0f);
7017         vec3 mask        = (vec3)(1.0f, 1.0f, 1.0f);
7018         ray sr;
7019         float rand1 = get_random(&seed1, &seed2);
7020         float rand2 = get_random(&seed1, &seed2);
7021 
7022 
7023         vec3 sample_dir =  cosineSampleHemisphere(rand1, rand2, initial_result.normal);
7024         sr.orig = initial_result.point + initial_result.normal * 0.0001f; //sweet spot for epsilon
7025         sr.dir = sample_dir;
7026         mask *= initial_result.mat.colour;
7027         for(int bounces = 0; bounces < NUM_BOUNCES; bounces++)
7028         {
7029             collision_result result;
7030             if(!collide_all(sr, &result, s, MESH_SCENE_DATA))
7031             {
7032                 accum_color += mask * sky.xyz;
7033                 break;
7034             }
7035 
7036 
7037             rand1 = get_random(&seed1, &seed2);
7038             rand2 = get_random(&seed1, &seed2);
7039 
7040             sample_dir =  cosineSampleHemisphere(rand1, rand2, result.normal);
7041 
7042             sr.orig = result.point + result.normal * 0.0001f; //sweet spot for epsilon
7043             sr.dir = sample_dir;
7044 
7045             //NOTE: janky emission, if reflectivity is 1 emission is 2 (only for tests)
```

```
            accum_color += mask * (float)(result.mat.reflectivity==1.)*2; //NOTE: EMMISION


            mask *= result.mat.colour;

            mask *= dot(sample_dir, result.normal);
        }

        //barrier(0); //good?

        accum_color = clamp(accum_color, 0.f, 1.f);

        fin_colour += accum_color * (1.f/NUM_SAMPLES);
    }
    #ifdef _WIN32
    out_tex[offset] = (vec4)(fin_colour, 1);
    #else
    out_tex[offset] = (vec4)(fin_colour.zyx, 1);
    #endif
}


__kernel void buffer_average(
    __global uchar4* out_tex,
    __global uchar4* fresh_frame_tex,
    const unsigned int width,
    const unsigned int height,
    const unsigned int sample
    /*const unsigned int num_samples*/)
{
    int id = get_global_id(0);
    int x  = id%width;
    int y  = id/width;
    int offset = (x + y * width);
    //        (n - 1) m[n-1] + a[n]
    // m[n] = ---------------------
    //                  n

    float x2 = ((float)sample-1.f)*( (float)out_tex[offset].x + (float)fresh_frame_tex[sample].x)  /
               (float)sample;

//wo
    /*float4 temp = mix((float4)(
                            (float)fresh_frame_tex[offset].x,
                           (float)fresh_frame_tex[offset].y,
                           (float)fresh_frame_tex[offset].z,
                           (float)fresh_frame_tex[offset].w),
                       (float4)(
                           (float)out_tex[offset].x,
                           (float)out_tex[offset].y,
                           (float)out_tex[offset].z,
                           (float)out_tex[offset].w), 0.5f+((float)sample/2048.f/2.f));// );*/
    /*vec4 temp =  (float)(
        (float)fresh_frame_tex[offset].x,
        (float)fresh_frame_tex[offset].y,
        (float)fresh_frame_tex[offset].z,
        (float)fresh_frame_tex[offset].w)/12.f;*/
    out_tex[offset] = (uchar4) ((unsigned char)x2,
                                (unsigned char)0,
                                (unsigned char)0,
                                (unsigned char)1.f);
/*
        fresh_frame_tex[offset]/(unsigned char)(1.f/(1-(float)sample/255))
        + out_tex[offset]/(unsigned char)(1.f/((float)sample/255));*/
}

__kernel void f_buffer_average(
    __global vec4* out_tex,
    __global vec4* fresh_frame_tex,
    const unsigned int width,
    const unsigned int height,
    const unsigned int num_samples,
    const unsigned int sample)
{
    int id = get_global_id(0);
    int x  = id%width;
    int y  = id/width;
    int offset = (x + y * width);

    //        (n - 1) m[n-1] + a[n]
    // m[n] = ---------------------
    //                  n

    out_tex[offset] = ((sample-1) * out_tex[offset] + fresh_frame_tex[offset]) / (float) sample;


    //out_tex[offset] = mix(fresh_frame_tex[offset], out_tex[offset],
```

```
7133     ///((float)sample)/(float)num_samples);
7134 }
7135
7136 __kernel void xorshift_batch(__global unsigned int* data)
7137 { //get_global_id is just a register, not a function
7138     uint d = data[get_global_id(0)];
7139     data[get_global_id(0)] = ((d << 1) | (d >> (sizeof(int)*8 - 1)))+1;//circular shift +1
7140 }
7141
7142 __kernel void f_buffer_to_byte_buffer_avg(
7143     __global unsigned int* out_tex,
7144     __global vec4* fresh_frame_tex,
7145     __global spath_progress* spath_data,
7146     const unsigned int width,
7147     const unsigned int sample_num)
7148 {
7149     int id = get_global_id(0);
7150     int x  = id%width;
7151     int y  = id/width;
7152     int offset = (x + y * width);
7153     //int roffset = (x + y * real);
7154
7155     vec4 data   = fresh_frame_tex[offset];
7156     vec4 colour = data.w==0 ? (vec4)(0,0,0,0) : data.xyzw/data.w;
7157
7158     /* if(get_global_id(0)%(width*100) == 0) */
7159     /*     printf("%f %f %f %f %f \n", */
7160     /*            fresh_frame_tex[offset].x, */
7161     /*            fresh_frame_tex[offset].y, */
7162     /*            fresh_frame_tex[offset].z, */
7163     /*            fresh_frame_tex[offset].w, */
7164     /*            colour.w); */
7165     out_tex[offset] = get_colour(colour);///sample_num);
7166 }
7167
7168
7169 __kernel void f_buffer_to_byte_buffer(
7170     __global unsigned int* out_tex,
7171     __global vec4* fresh_frame_tex,
7172     const unsigned int width,
7173     const unsigned int height)
7174 {
7175     int id = get_global_id(0);
7176     int x  = id%width;
7177     int y  = id/width;
7178     int offset = (x + y * width);
7179     out_tex[offset] = get_colour(fresh_frame_tex[offset]);
7180 }
7181
7182 vec4 shade(collision_result result, scene s, MESH_SCENE_DATA_PARAM)
7183 {
7184     const vec3 light_pos = (vec3)(1,2, 0);
7185     vec3 nspace_light_dir = normalize(light_pos-result.point);
7186     vec4 test_lighting = (vec4) (clamp((float)dot(result.normal, nspace_light_dir), 0.0f, 1.0f));
7187     ray r;
7188     r.dir  = nspace_light_dir;
7189     r.orig = result.point + nspace_light_dir*0.00001f;
7190     collision_result _cr;
7191     bool visible = !collide_all(r, &_cr, s, MESH_SCENE_DATA);
7192     test_lighting *= (vec4)(result.mat.colour, 1.0f);
7193     return visible*test_lighting/2;
7194 }
7195
7196
7197 __kernel void cast_ray_test(
7198     __global unsigned int* out_tex,
7199     const __global ray* ray_buffer,
7200     const __global material* material_buffer,
7201     const __global sphere* spheres,
7202     const __global plane* planes,
7203 //Mesh
7204     const __global mesh* meshes,
7205     image1d_buffer_t indices,
7206     image1d_buffer_t vertices,
7207     image1d_buffer_t normals,
7208     /* const __global vec2* texcoords, */
7209     /* , */
7210
7211
7212     const unsigned int width,
7213     const unsigned int height,
7214     const vec4 pos)
7215 {
7216     scene s;
7217     s.material_buffer = material_buffer;
7218     s.spheres         = spheres;
7219     s.planes          = planes;
```

```
7220     s.meshes          = meshes;
7221
7222     const vec4 sky = (vec4) (0.84, 0.87, 0.93, 0);
7223     //return;
7224     int id = get_global_id(0);
7225     int x  = id%width;
7226     int y  = id/width;
7227     int offset = x+y*width;
7228     int ray_offset = offset;
7229
7230
7231     ray r;
7232     r = ray_buffer[ray_offset];
7233     r.orig = pos.xyz; //NOTE: unnecesesary rn, in progress of updating kernels w/ the new ray buffers.
7234
7235     //r.dir  = (vec3)(0,0,-1);
7236
7237     //out_tex[x+y*width] = get_colour_signed((vec4)(r.dir,0));
7238     //out_tex[x+y*width] = get_colour_signed((vec4)(1,1,0,0));
7239     collision_result result;
7240     if(!collide_all(r, &result, s, MESH_SCENE_DATA))
7241     {
7242         out_tex[x+y*width] = get_colour( sky );
7243         return;
7244     }
7245     vec4 colour = shade(result, s, MESH_SCENE_DATA);
7246
7247
7248     #define NUM_REFLECTIONS 2
7249     ray rays[NUM_REFLECTIONS];
7250     collision_result results[NUM_REFLECTIONS];
7251     vec4 colours[NUM_REFLECTIONS];
7252     int early_exit_num = NUM_REFLECTIONS;
7253     for(int i = 0; i < NUM_REFLECTIONS; i++)
7254     {
7255         if(i==0)
7256         {
7257             rays[i].orig = result.point + result.normal * 0.0001f; //NOTE: BIAS
7258             rays[i].dir  = reflect(r.dir, result.normal);
7259         }
7260         else
7261         {
7262             rays[i].orig = results[i-1].point + results[i-1].normal * 0.0001f; //NOTE: BIAS
7263             rays[i].dir  = reflect(rays[i-1].dir, results[i-1].normal);
7264         }
7265         if(collide_all(rays[i], results+i, s, MESH_SCENE_DATA))
7266         {
7267             colours[i] = shade(results[i], s, MESH_SCENE_DATA);
7268         }
7269         else
7270         {
7271             colours[i] = sky;
7272             early_exit_num = i;
7273             break;
7274         }
7275     }
7276     for(int i = early_exit_num-1; i > -1; i--)
7277     {
7278         if(i==NUM_REFLECTIONS-1)
7279             colours[i] = mix(colours[i], sky, results[i].mat.reflectivity);
7280
7281         else
7282             colours[i] = mix(colours[i], colours[i+1], results[i].mat.reflectivity);
7283
7284     }
7285
7286     colour = mix(colour, colours[0],  result.mat.reflectivity);
7287
7288     out_tex[offset] = get_colour( colour );
7289 }
7290
7291
7292 //NOTE: it might be faster to make the ray buffer a multiple of 4 just to align with words...
7293 __kernel void generate_rays(
7294     __global ray* out_tex,
7295     const unsigned int width,
7296     const unsigned int height,
7297     const t_mat4 wcm)
7298 {
7299     int id = get_global_id(0);
7300     int x  = id%width;
7301     int y  = id/width;
7302     int offset = (x + y * width);
7303
7304     ray r;
7305
7306     float aspect_ratio = width / (float)height; // assuming width > height
```

```
    float cam_x = (2 * (((float)x + 0.5) / width) - 1) * tan(FOV / 2 * M_PI_F / 180) * aspect_ratio;
    float cam_y = (1 - 2 * (((float)y + 0.5) / height)) * tan(FOV / 2 * M_PI_F / 180);

    //r.orig = matvec((float*)&wcm, (vec4)(0.0, 0.0, 0.0, 1.0)).xyz;
    //r.dir  = matvec((float*)&wcm, (vec4)(cam_x, cam_y, -1.0f, 1)).xyz - r.orig;

    r.orig = (vec3)(0, 0, 0);
    r.dir  = (vec3)(cam_x, cam_y, -1.0f) - r.orig;

    r.dir = normalize(r.dir);

    out_tex[offset]   = r;
}
#define FOV 80.0f

#define vec2 float2
#define vec3 float3
#define vec4 float4

#define EPSILON 0.0000001f
#define FAR_PLANE 100000000

typedef float mat4[16];



/********/
/* Util */
/********/


__constant sampler_t sampler = CLK_NORMALIZED_COORDS_FALSE |
    CLK_ADDRESS_CLAMP_TO_EDGE   |
    CLK_FILTER_NEAREST;

typedef struct
{
    vec4 x;
    vec4 y;
    vec4 z;
    vec4 w;
} __attribute__((aligned (16))) t_mat4;

typedef struct kd_tree_collision_result
{
    unsigned int triangle_index;
    float t;
    float u;
    float v;
} kd_tree_collision_result;

void swap_float(float *f1, float *f2)
{
    float temp = *f2;
    *f2 = *f1;
    *f1 = temp;
}

vec4 matvec(float* m, vec4 v)
{
    return (vec4) (
        m[0+0*4]*v.x + m[1+0*4]*v.y + m[2+0*4]*v.z + m[3+0*4]*v.w,
        m[0+1*4]*v.x + m[1+1*4]*v.y + m[2+1*4]*v.z + m[3+1*4]*v.w,
        m[0+2*4]*v.x + m[1+2*4]*v.y + m[2+2*4]*v.z + m[3+2*4]*v.w,
        m[0+3*4]*v.x + m[1+3*4]*v.y + m[2+3*4]*v.z + m[3+3*4]*v.w );
}

unsigned int get_colour(vec4 col)
{
    unsigned int outCol = 0;

    col = clamp(col, 0.0f, 1.0f);

    outCol |= 0xff000000 & (unsigned int)(col.w*255)<<24;
    outCol |= 0x00ff0000 & (unsigned int)(col.x*255)<<16;
    outCol |= 0x0000ff00 & (unsigned int)(col.y*255)<<8;
    //outCol |= 0x000000ff & (unsigned int)(col.z*255);
    outCol |= 0x000000ff & (unsigned int)(col.z*255);

    /* outCol |= 0xff000000 & min((unsigned int)(col.w*255), (unsigned int)255)<<24; */
    /* outCol |= 0x00ff0000 & min((unsigned int)(col.x*255), (unsigned int)255)<<16; */
    /* outCol |= 0x0000ff00 & min((unsigned int)(col.y*255), (unsigned int)255)<<8; */
    /* outCol |= 0x000000ff & min((unsigned int)(col.z*255), (unsigned int)255); */
    return outCol;
}

static float get_random(unsigned int *seed0, unsigned int *seed1)
```

```
{
    /* hash the seeds using bitwise AND operations and bitshifts */
    *seed0 = 36969 * ((*seed0) & 65535) + ((*seed0) >> 16);
    *seed1 = 18000 * ((*seed1) & 65535) + ((*seed1) >> 16);
    unsigned int ires = ((*seed0) << 16) + (*seed1);
    /* use union struct to convert int to float */
    union {
        float f;
        unsigned int ui;
    } res;
    //Maybe good, maybe not

    res.ui = (ires & 0x007fffff) | 0x40000000;  /* bitwise AND, bitwise OR */
    return (res.f - 2.0f) / 2.0f;
}

uint MWC64X(uint2 *state) //http://cas.ee.ic.ac.uk/people/dt10/research/rngs-gpu-mwc64x.html
{
    enum { A=4294883355U};
    uint x=(*state).x, c=(*state).y;  // Unpack the state
    uint res=x^c;                     // Calculate the result
    uint hi=mul_hi(x,A);              // Step the RNG
    x=x*A+c;
    c=hi+(x<c);
    *state=(uint2)(x,c);               // Pack the state back up
    return res;                       // Return the next result
}

vec3 reflect(vec3 incidentVec, vec3 normal)
{
    return incidentVec - 2.f * dot(incidentVec, normal) * normal;
}

__kernel void blit_float_to_output(
    __global unsigned int* out_tex,
    __global float* in_flts,
    const unsigned int width,
    const unsigned int height)
{
    int id = get_global_id(0);
    int x  = id%width;
    int y  = id/width;
    int offset = x+y*width;
    out_tex[offset] = get_colour((vec4)(in_flts[offset]));
}

__kernel void blit_float3_to_output(
    __global unsigned int* out_tex,
    image2d_t in_flts,
    const unsigned int width,
    const unsigned int height)
{
    int id = get_global_id(0);
    int x  = id%width;
    int y  = id/width;
    int offset = x+y*width;
    out_tex[offset] = get_colour(read_imagef(in_flts, sampler, (float2)(x, y)));
}
h {
      color: black;
      font-family: office_code_pro_li;
      font-size: 72pt;
      /*text-align: center;*/
}
.titleBody {
    text-align: center;
}
h2{
    color: black;
      font-family: office_code_pro_li;
      font-size: 30pt;
}

input[type=text] {
      background-color: #fff;
      border: 2px solid #000;
      color: black;
      font-family: office_code_pro_li;
      font-size: 10pt;
      margin: 4px 2px;
      padding: 12px 20px;

      cursor: pointer;
      width: 40%;
}

p{
```

```
    color: black;
      font-family: office_code_pro_li;
}

button {
  background-color: #fff; /* Green */
  border: 2px solid #000;
  color: black;
  padding: 15px 32px;
  font-family: office_code_pro_li;
  text-align: center;
  text-decoration: none;
  display: inline-block;
  font-size: 16px;
  margin: 4px 2px;
  cursor: pointer;
}

hr.titleBar {
    margin-block-start: 0;
}

@font-face {
  font-family: office_code_pro_li;
  src: url(./ocp_li.woff);
}<!DOCTYPE html>
<html>
  <head>
      <link rel="stylesheet" href="./style.css">
    <title>Path Tracer UI</title>
  </head>
  <body>
      <div class="titleBody">
          <h>Path Tracer UI</h>
          <hr class = "titleBar">
      </div>
      <div style="text-align: right;">
          <p id="status"></p>
      </div>
      <div>
          <h2>Info:</h2>
          <p id="info_para"></p>
      </div>

      <button onclick="send_sb_cmd()">Simple Raytracer</button>
      <button onclick="send_ss_cmd()">Path Raytracer</button>
      <button onclick="send_path_cmd()">Split Path Tracer</button>

    <div>
          <input id="scene" type="text" value="scenes/path_obj_test.rsc">
          <button onclick="send_scene_change_cmd()">Change Scene</button>
      </div>


      <script language="javascript" type="text/javascript">
          var ws;
          function connect()
          {
              ws = new WebSocket('ws://' + location.host + '/ws');
              if (!window.console) { window.console = { log: function() {} } };
              ws.onopen = function(ev)
              {
                  console.log(ev);
                  document.getElementById("status").innerHTML = "Connected."
                  document.getElementById("status").style.color = "green";
               ws.send("{\"type\":0}"); //get init info.
               };
               ws.onerror = function(ev) { console.log(ev); };
               ws.onclose = function(ev) {
                   console.log(ev);
                   document.getElementById("status").innerHTML = "Disconnected."
                   document.getElementById("status").style.color = "red";
                   setTimeout(function() { connect(); }, 1000);
                 ws = null;
               };
               ws.onmessage = function(ev) {
                   console.log(ev);
                   console.log(ev.data);
                   parse_ws(JSON.parse(ev.data));
               };
           }
           connect();



        function send_sb_cmd()
        {
```

```
            data = {
                type:1,
                action:{
                    type:0
                }
            }
            ws.send(JSON.stringify(data));
        }
        function send_ss_cmd()
        {
            data = {
                type:1,
                action:{
                    type:1
                }
            }
            ws.send(JSON.stringify(data));
           }
           function send_path_cmd()
           {
               data = {
                   type:1,
                   action:{
                       type:2
                   }
               }
               ws.send(JSON.stringify(data));
           }

           function send_scene_change_cmd()
           {
               data = {
                   type:1,
                   action:{
                       type : 3,
                       scene : document.getElementById("scene").value
                   }
               }
               ws.send(JSON.stringify(data));
           }

           function parse_ws(data)
           {
               switch(data.type)
               {
                   case 0:
                   {
                       document.getElementById('info_para').innerHTML = data.message;
                       break;
                   }
               }

           }
          /*window.onload = function() {
              document.getElementById('send_button').onclick = function(ev) {
                  var msg = document.getElementById('send_input').value;
                  document.getElementById('send_input').value = '';
                  ws.send(msg);
              };
              document.getElementById('send_input').onkeypress = function(ev) {
                  if (ev.keyCode == 13 || ev.which == 13) {
                      document.getElementById('send_button').click();
                  }
              };
          };*/
</script>

  </body>
</html>
```