```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;

    vec4 norm;
    //float test2;

//32
    int material_index;
} U_ALIGN(16) plane;
float does_collide_plane(plane, ray);

ray generate_ray(int x, int y, int width, int height, float fov);
float* matvec_mul(mat4 m, vec4 v);
/*****************************************/
/* NOTE: Irradiance Caching is Incomplete */
/*****************************************/

#pragma once
#include <stdint.h>
#include <alignment_util.h>
```

```

#define NUM_MIPMAPS 4 //NOTE: 1080/(2^4) != integer

typedef struct _rt_ctx raytracer_context;

//    0 = 000: x-, y-, z-
//    1 = 001: x-, y-, z+
//    2 = 010: x-, y+, z-
//    3 = 011: x-, y+, z+
//    4 = 100: x+, y-, z-
//    5 = 101: x+, y-, z+
//    6 = 110: x+, y+, z-
//    7 = 111: x+, y+, z+

typedef struct
{
    vec3 point;
    vec3 normal;

    float rad;

    vec3 col;

    vec3 gpos;
    vec3 gdir;
} ic_ir_value;

typedef struct _ic_octree_node ic_octree_node;

struct _ic_octree_node
{
    bool leaf;
    bool active;

    union
    {
        struct
        {
            unsigned int buffer_offset;
            unsigned int num_elems;
        } leaf;
        struct
        {

            ic_octree_node* children[8];
        } branch;
    } data;
    vec3 min;
    vec3 max;
};

typedef struct
{
    ic_octree_node* root;
    int node_count;
    unsigned int width;
    unsigned int max_depth;
} ic_octree;

typedef struct
{
    //vec4* texture;
    cl_mem cl_image_ref;
    unsigned int width, height;
}  ic_mipmap_gb;

typedef struct
{
    //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 1.0f //Cost for traversal
#define KDTREE_KI 1.5f //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



    unsigned int num_nodes_total;
    unsigned int num_tris_padded;
    unsigned int num_traversal_nodes;
    unsigned int num_leaves;
    unsigned int num_indices_total;

    unsigned int max_recurse;
    unsigned int tri_for_leaf_threshold;

    scene* s;
    AABB bounds;

    //Serialized form.
    char* buffer;
    unsigned int buffer_size;
    cl_mem cl_kd_tree_buffer;

    //AABB V; //Total bounding box

} kd_tree;


kd_tree*      kd_tree_init();
kd_tree_node* kd_tree_node_init();

bool kd_tree_node_is_leaf(kd_tree_node*);
void kd_tree_construct(kd_tree* tree); //O(n log^2 n) implementation
void kd_tree_generate_serialized(kd_tree* tree);
#pragma once
#include <scene.h>
#include <alignment_util.h>

scene* load_scene_json(char* data);
scene* load_scene_json_url(char* url);
#pragma once

typedef struct
{
    void (*start_func)();
    void (*loop_start_func)();
    void (*update_func)();
    void (*sleep_func)(int);
    void (*draw_weird)();
    void* (*get_bitmap_memory_func)();
    int  (*get_time_mili_func)();
    int  (*get_width_func)();
    int  (*get_height_func)();
    void (*start_thread_func)(void (*func)(void*), void* data);
} os_abs;

void os_start(os_abs);
```

```
void os_loop_start(os_abs);
void os_update(os_abs);
void os_sleep(os_abs, int);
void os_draw_weird(os_abs abs);
void* os_get_bitmap_memory(os_abs);
int os_get_time_mili(os_abs);
int os_get_width(os_abs);
int os_get_height(os_abs);
void os_start_thread(os_abs, void (*func)(void*), void* data);
#pragma once
#include <time.h>
#include <os_abs.h>

os_abs init_osx_abs();

void osx_start();
void osx_loop_start();
void osx_enqueue_update();
void osx_sleep(int miliseconds);
void* osx_get_bitmap_memory();
int osx_get_time_mili();
int osx_get_width();
int osx_get_height();
void osx_start_thread(void (*func)(void*), void* data);
#pragma once
#include <alignment_util.h>

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

#define MACRO_GEN(n, t, v,  i)                  \
    char n[64];                                 \
    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

struct _rt_ctx;


typedef struct ui_ctx
{
    struct _rt_ctx* rctx; //General Raytracer Context

} ui_ctx;

void web_server_start(void*);
#pragma once
#include <windows.h>
#include <stdbool.h>
#include <os_abs.h>

typedef struct
{
    HINSTANCE instance;
    int       nCmdShow;
    WNDCLASSEX wc;
    HWND     win;

    int width, height;

    BITMAPINFO bitmap_info;
    void*      bitmap_memory;

    // HDC        render_device_context;

    bool       shouldRun;
    //Bitbuffer
} win32_context;


os_abs init_win32_abs();

void win32_start_thread(void (*func)(void*), void* data);

```

```
//void create_win32_window();
void win32_start();
void win32_loop();

void win32_update();

void win32_sleep(int);

void* win32_get_bitmap_memory();

int win32_get_time_mili();

int win32_get_width();
int win32_get_height();
#define CL_TARGET_OPENCL_VERSION 120

#include <math.h>
#include <stdlib.h>

#define MMX_IMPLEMENTATION
#include <vec.h>
#undef  MMX_IMPLEMENTATION
#define TINYOBJ_LOADER_C_IMPLEMENTATION
#include <tinyobj_loader_c.h>
#undef TINYOBJ_LOADER_C_IMPLEMENTATION


#include <mongoose.c>
#include <parson.c>

#ifdef _WIN32
#define WIN32 // I don't want to fix all of my accidents right now.
#endif



//REMOVE FOR PRESENTATION
#define DEV_MODE



#ifdef WIN32
#include <win32.c>
#else
#include <osx.m>
#endif

//#define _MEM_DEBUG //Enable verbose memory allocation, movement and freeing

#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;
        result->max[k] = -1000000;
    }

    for(int i = 0; i < 3; i++)
    {
        float* vertex = vertices[indices[i][0]];

        for(int k = 0; k < 3; k++)
        {
            if(vertex[k] < result->min[k])
                result->min[k] = vertex[k];

            if(vertex[k] > result->max[k])
                result->max[k] = vertex[k];
        }
    }
}

inline void AABB_construct_from_vertices(AABB* result, vec3* vertices,
                                         unsigned int num_vertices)
{
    for(int k = 0; k < 3; k++)
    {
        result->min[k] =  1000000;
        result->max[k] = -1000000;
    }
    for(int i = 0; i < num_vertices; i++)
    {
        for(int k = 0; k < 3; k++)
        {
            if(vertices[i][k] < result->min[k])
                result->min[k] = vertices[i][k];

            if(vertices[i][k] > result->max[k])
                result->max[k] = vertices[i][k];
        }
    }
}

inline bool AABB_is_planar(AABB* source, uint8_t k)
{
    if(source->max[k]-source->min[k] == 0.0f) //TODO: use epsilon instead of 0
        return true;
    return false;
}

inline float AABB_ilerp(AABB source, uint8_t k, float world_b)
{
    return (world_b - source.min[k]) / (source.max[k] - source.min[k]);
}

inline float does_collide_sphere(sphere s, ray r)
{
    float t0, t1; // solutions for t if the ray intersects


    vec3 L;
    xv_sub(L, r.orig, s.pos, 3);


    float a = 1.0f; //NOTE: we always normalize the direction vector.
    float b = xv3_dot(r.dir, L) * 2.0f;
    float c = xv3_dot(L, L) - (s.radius*s.radius); //NOTE: square can be optimized out.
    if (!solve_quadratic(&a, &b, &c, &t0, &t1)) return -1.0f;

    if (t0 > t1) swap_float(&t0, &t1);

    if (t0 < 0) {
        t0 = t1; // if t0 is negative, use t1 instead
        if (t0 < 0) return -1.0f; // both t0 and t1 are negative
    }

    return t0;
}

inline float does_collide_plane(plane p, ray r)
{
    float denom = xv3_dot(r.dir, p.norm);
    if (denom > 1e-6)
    {
        vec3 l;
        xv_sub(l, p.pos, r.orig, 3);
        float t = xv3_dot(l, p.norm) / denom;
        if (t >= 0)
            return -1.0;
        return t;
    }
    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);
}

void ic_octree_make_branch(ic_octree* tree, ic_octree_node* node)
{

    node->leaf = false;
    for(int i = 0; i < 8; i++)
    {
        node->data.branch.children[i] = malloc(sizeof(ic_octree_node));
        ic_octree_init_leaf(node->data.branch.children[i], node, i);
        tree->node_count++;
    }
}

//TODO: test if points are the same
void _ic_octree_rec_resolve(ic_context* ictx, ic_octree_node* leaf, unsigned int node1, unsigned int node2,
                            unsigned int depth)
{
    if(depth > ictx->octree.max_depth)
    {
        //TODO: just group buffers together
        printf("ERROR: octree reached max depth when trying to resolve collision. (INCOMPLETE)\n");
        exit(1);
    }
    vec3 mid_point;
```

```
    xv_sub(mid_point, leaf->max, leaf->min, 3);
    xv_divieq(mid_point, 2, 3);
    unsigned int i1 =
        ((mid_point[0]<ictx->ir_buf[node1].point[0])<<2) |
        ((mid_point[1]<ictx->ir_buf[node1].point[1])<<1) |
        ((mid_point[2]<ictx->ir_buf[node1].point[2]));
    unsigned int i2 =
        ((mid_point[0]<ictx->ir_buf[node2].point[0])<<2) |
        ((mid_point[1]<ictx->ir_buf[node2].point[1])<<1) |
        ((mid_point[2]<ictx->ir_buf[node2].point[2]));
    ic_octree_make_branch(&ictx->octree, leaf);
    if(i1==i2)
        _ic_octree_rec_resolve(ictx, leaf->data.branch.children[i1], node1, node2, depth+1);
    else
    { //happiness
        leaf->data.branch.children[i1]->data.leaf.buffer_offset = node1;
        leaf->data.branch.children[i1]->data.leaf.num_elems = 1;
        leaf->data.branch.children[i2]->data.leaf.buffer_offset = node2;
        leaf->data.branch.children[i2]->data.leaf.num_elems = 1;
    }
}

void _ic_octree_rec_insert(ic_context* ictx, ic_octree_node* node, unsigned int v_ptr, unsigned int depth)
{
    if(node->leaf && !node->active)
    {
        node->active = true;
        node->data.leaf.buffer_offset = v_ptr;
        node->data.leaf.num_elems     = 1; //TODO: add suport for more than 1.
        return;
    }
    else if(node->leaf)
    {
        //resolve
        _ic_octree_rec_resolve(ictx, node, v_ptr, node->data.leaf.buffer_offset, depth+1);
    }
    else
    {
        ic_octree_node* new_node = node->data.branch.children[
            ((ictx->ir_buf[node->data.leaf.buffer_offset].point[0]<ictx->ir_buf[v_ptr].point[0])<<2) |
                ((ictx->ir_buf[node->data.leaf.buffer_offset].point[1]<ictx->ir_buf[v_ptr].point[1])<<1) |
                ((ictx->ir_buf[node->data.leaf.buffer_offset].point[2]<ictx->ir_buf[v_ptr].point[2]))];
        _ic_octree_rec_insert(ictx, new_node, v_ptr, depth+1);
    }
}

void ic_octree_insert(ic_context* ictx, vec3 point, vec3 normal)
{
    if(ictx->ir_buf_current_offset==ictx->ir_buf_size) //TODO: dynamically resize or do something else
    {
        printf("ERROR: irradiance buffer is full!\n");
        exit(1);
    }
    ic_ir_value irradiance_value; //TODO: EVALUATE THIS
    irradiance_value.rad = 0.f; //Gets rid of error, this doesn't work anyways so its good enough.
    ictx->ir_buf[ictx->ir_buf_current_offset++] = irradiance_value;
    _ic_octree_rec_insert(ictx, ictx->octree.root, ictx->ir_buf_current_offset, 0);
}

//NOTE: outBuffer is only bools but using char for safety accross compilers.
//      Also assuming that buf is grayscale
void dither(float* buf, const int width, const int height)
{
    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);
```

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

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

```
                        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);

    printf("Retrieved Discontinuity Data\nDithering Discontinuity Map\n");
    //NOTE: read buffer is blocking so we don't need clFinish
    dither(c_fin_disc_map, rctx->width, rctx->height);
    err = clEnqueueWriteBuffer(rctx->rcl->commands, fin_disc_map,
                               CL_TRUE, 0,
                               rctx->width*rctx->height*sizeof(float),
                               c_fin_disc_map, 0, 0, NULL);
    ASRT_CL("Failed to write dithered discontinuity map");


    //INSERT
    cl_kernel kernel = rctx->program->raw_kernels[BLIT_FLOAT_OUTPUT_INDX];

    clSetKernelArg(kernel, 0, sizeof(cl_mem), &rctx->cl_output_buffer);
    clSetKernelArg(kernel, 1, sizeof(cl_mem), &fin_disc_map);
    clSetKernelArg(kernel, 2, sizeof(int),    &rctx->width);
    clSetKernelArg(kernel, 3, sizeof(int),    &rctx->height);

    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 BLIT_FLOAT_OUTPUT_INDX");

    clFinish(rctx->rcl->commands);

    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 Buffer");
    printf("test!!\n");


}
#include <kdtree.h>
#include <scene.h>

#define KDTREE_EPSILON 0.001f

#define KDTREE_BOTH  0
#define KDTREE_LEFT  1
#define KDTREE_RIGHT 2

#define KDTREE_END   0
#define KDTREE_PLANAR 1
#define KDTREE_START  2

//Literally an index buffer to the index buffer
typedef struct kd_tree_event
{
    unsigned int tri_index_offset;
    float   b;
    uint8_t k;
    uint8_t type;
} kd_tree_event;

typedef struct kd_tree_sah_results
{
    float cost;
    uint8_t side; //1 left, 2 right
} kd_tree_sah_results;

inline kd_tree_sah_results kd_tree_sah_results_c(float cost, uint8_t side)
{
    kd_tree_sah_results r;
    r.cost = cost;
    r.side = side;
    return r;
}

typedef struct kd_tree_find_plane_results
{
    kd_tree_event p;
    unsigned int NL;
    unsigned int NR;
    unsigned int NP;
    uint8_t side;
    float cost;

} kd_tree_find_plane_results;


inline 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;

    return event_out;
}

kd_tree_event_buffer kd_tree_mergesort_event_buffer(kd_tree_event_buffer buf)
{

    if(buf.num_events == 1)
        return buf;


    int firstHalf = (int)ceil( (float)buf.num_events / 2.f);


    kd_tree_event_buffer buf1 = {buf.events, firstHalf, };
    kd_tree_event_buffer buf2 = {buf.events+firstHalf, buf.num_events-firstHalf};


    buf1 = kd_tree_mergesort_event_buffer(buf1);
    buf2 = kd_tree_mergesort_event_buffer(buf2);


    return kd_tree_merge_event_buffers(buf1, buf2);
}


```

```
kd_tree* kd_tree_init()
{
    kd_tree* tree = malloc(sizeof(kd_tree));
    tree->root = NULL;
    //Defaults
    tree->k    = 3;
    tree->max_recurse = 50;
    tree->tri_for_leaf_threshold = 2;
    tree->num_nodes_total     = 0;
    tree->num_tris_padded     = 0;
    tree->num_traversal_nodes = 0;
    tree->num_leaves          = 0;
    tree->num_indices_total   = 0;
    tree->buffer_size         = 0;
    tree->buffer              = NULL;
    tree->cl_kd_tree_buffer   = NULL;
    xv3_zero(tree->bounds.min);
    xv3_zero(tree->bounds.max);
    return tree;
}

kd_tree_node* kd_tree_node_init()
{
    kd_tree_node* node = malloc(sizeof(kd_tree_node));
    node->k = 0;
    node->b = 0.5f; //generic default, shouldn't matter with SAH anyways

    node->left  = NULL;
    node->right = NULL;

    return node;
}

bool kd_tree_node_is_leaf(kd_tree_node* node)
{
    if(node->left == NULL || node->right == NULL)
    {
        assert(node->left == NULL && node->right == NULL);
        return true;
    }

    return false;
}



kd_tree_find_plane_results kd_tree_find_plane(kd_tree* tree, AABB V,
                                              kd_tree_triangle_buffer tri_buf)
{
    float     best_cost = INFINITY;
     kd_tree_find_plane_results result;


    for(int k = 0; k < tree->k; k++)
    {
        kd_tree_event_buffer event_buf = {NULL, 0}; //gets rid of an initialization warning I guess?
        {// Generate events
            //Divide by three because we only want tris
            event_buf.num_events = tri_buf.num_triangles*2;

            event_buf.events = malloc(sizeof(kd_tree_event)*event_buf.num_events);
            unsigned int j = 0;
            for (int i = 0; i < tri_buf.num_triangles; i++)
            {
                AABB tv, B;
                AABB_construct_from_triangle(&tv, tree->s->mesh_indices+tri_buf.triangle_buffer[i],
                                             tree->s->mesh_verts);
                AABB_clip(&B, &tv, &V);
                if(AABB_is_planar(&B, k))
                {
                    event_buf.events[j++] = (kd_tree_event) {i*3, B.min[k], k, KDTREE_PLANAR};
                }
                else
                {
                    event_buf.events[j++] = (kd_tree_event) {i*3, B.min[k], k, KDTREE_START};
                    event_buf.events[j++] = (kd_tree_event) {i*3, B.max[k], k, KDTREE_END};
                }
            }
            event_buf.num_events = j;

            int last_num_events = event_buf.num_events;
            event_buf = kd_tree_mergesort_event_buffer(event_buf);
            assert(event_buf.num_events == last_num_events);
        }

        int NL, NP, NR;
        NL = NP = 0;
        NR = tri_buf.num_triangles;
        for (int i = 0; i < event_buf.num_events;)
        {
            kd_tree_event p = event_buf.events[i];
            int Ps, Pe, Pp;
            Ps = Pe = Pp = 0;
            while(i < event_buf.num_events && event_buf.events[i].b == p.b && event_buf.events[i].type == KDTREE_END)
            {
                Pe += 1;
                i++;
            }
            while(i < event_buf.num_events && event_buf.events[i].b == p.b && event_buf.events[i].type == KDTREE_PLANAR)
            {
                Pp += 1;
                i++;
            }
            while(i < event_buf.num_events && event_buf.events[i].b == p.b && event_buf.events[i].type == KDTREE_START)
            {
```

```
                Ps += 1;
                i++;
            }

            NP =  Pp;
            NR -= Pp;
            NR -= Pe;

            kd_tree_sah_results results = kd_tree_SAH(k, AABB_ilerp(V, k, p.b), V, NL, NR, NP);

            if (results.cost < best_cost)
            {
                best_cost = results.cost;
                result.p = p;
                result.side = results.side;

                result.NL = NL;
                result.NR = NR;
                result.NP = NP;
                result.cost = results.cost; //just the min cost, really confusing syntax
            }

            NL += Ps;
            NL += NP;
            NP =  0;

        }
        free(event_buf.events);
    }

    return result;
}

void kd_tree_classify(kd_tree* tree, kd_tree_triangle_buffer tri_buf,
                      kd_tree_find_plane_results results,
                      kd_tree_triangle_buffer* TL_out, kd_tree_triangle_buffer* TR_out)
{
    kd_tree_triangle_buffer TL;
    kd_tree_triangle_buffer TR;
    TL.num_triangles   = results.NL + (results.side == KDTREE_LEFT ? results.NP : 0);
    TL.triangle_buffer = (unsigned int*) malloc(sizeof(unsigned int)*TL.num_triangles); //NOTE: memory leak, never freed.
    TR.num_triangles   = results.NR + (results.side == KDTREE_RIGHT ? results.NP : 0);
    TR.triangle_buffer = (unsigned int*) malloc(sizeof(unsigned int)*TR.num_triangles);
    unsigned int TLI, TRI;
    TLI = TRI = 0;
    for(int i = 0; i < tri_buf.num_triangles; i++)
    {
        bool isLeft = false;
        bool isRight = false;
        for(int j = 0; j < 3; j++)
        {

            float p = tree->s->mesh_verts
                  [ tree->s->mesh_indices
                  [ tri_buf.triangle_buffer[i]+j ][0] ][results.p.k];
            if(p < results.p.b)
                isLeft = true;
            if(p > results.p.b)
                isRight = true;
        }

        //Favour the right rn
        if(isLeft && isRight) //should be splitting.
        {
            TR.triangle_buffer[TRI++] = tri_buf.triangle_buffer[i];
            TL.triangle_buffer[TLI++] = tri_buf.triangle_buffer[i];
        }
        else if(!isLeft && !isRight)
        {
            if(results.side == KDTREE_LEFT)
                TL.triangle_buffer[TLI++] = tri_buf.triangle_buffer[i];
            else if(results.side == KDTREE_RIGHT)
                TR.triangle_buffer[TRI++] = tri_buf.triangle_buffer[i];
            else
            {//implement this
                printf("really bad\n");
                assert(1!=1);
            }
        }
        else if(isLeft)
            TL.triangle_buffer[TLI++] = tri_buf.triangle_buffer[i];
        else if(isRight)
            TR.triangle_buffer[TRI++] = tri_buf.triangle_buffer[i];
    }
    *TL_out = TL;
    *TR_out = TR;

}

bool kd_tree_should_terminate(kd_tree* tree, unsigned int num_tris, AABB V, unsigned int depth)
{
    for(int k = 0; k < tree->k; k++)
        if(AABB_is_planar(&V, k))
            return true;
    if(depth == tree->max_recurse)
        return true;
    if(num_tris <= tree->tri_for_leaf_threshold)
        return true;

    return false;
}

kd_tree_node* kd_tree_construct_rec(kd_tree* tree, AABB V, kd_tree_triangle_buffer tri_buf,
                                    unsigned int depth)
{
```

```
    kd_tree_node* node = kd_tree_node_init();

    tree->num_nodes_total++;
    if(kd_tree_should_terminate(tree, tri_buf.num_triangles, V, depth))
    {
        node->triangles = tri_buf;
        tree->num_leaves++;
        tree->num_indices_total += tri_buf.num_triangles;
        tree->num_tris_padded   += tri_buf.num_triangles % 8;
        return node;
    }

    kd_tree_find_plane_results res = kd_tree_find_plane(tree, V, tri_buf);

    if(res.cost > KDTREE_KI*(float)tri_buf.num_triangles)
    {
        node->triangles = tri_buf;
        tree->num_leaves++;
        tree->num_indices_total += tri_buf.num_triangles;
        tree->num_tris_padded   += tri_buf.num_triangles % 8;

        return node;
    }


    tree->num_traversal_nodes++;


    uint8_t     k = res.p.k;
    float world_b = res.p.b;

    node->k = k;
    node->b = world_b; //Local b is honestly useless

    assert(node->b != V.min[k]);
    assert(node->b != V.max[k]);

    AABB VL;
    AABB VR;
    AABB_divide_world(V, k, world_b, &VL, &VR);

    kd_tree_triangle_buffer TL, TR;
    kd_tree_classify(tree, tri_buf, res, &TL, &TR);

    node->left  = kd_tree_construct_rec(tree, VL, TL, depth+1);
    node->right = kd_tree_construct_rec(tree, VR, TR, depth+1);

    return node;
}

kd_tree_triangle_buffer kd_tree_gen_initial_tri_buf(kd_tree* tree)
{
    assert(tree->s->num_mesh_indices % 3 == 0);
    kd_tree_triangle_buffer buf;
    buf.num_triangles   = tree->s->num_mesh_indices/3;
    buf.triangle_buffer = (unsigned int*) malloc(sizeof(unsigned int) * buf.num_triangles);

    for (int i = 0; i < buf.num_triangles; i++)
        buf.triangle_buffer[i] = i * 3;

    return buf;
}

void kd_tree_construct(kd_tree* tree) //O(n log^2 n) implementation
{
    assert(tree->s != NULL);

    if(tree->s->num_mesh_indices == 0)
    {
        printf("WARNING: Skipping k-d tree Construction, num_mesh_indices is 0.\n");
        return;
    }

    AABB V;
    AABB_construct_from_vertices(&V, tree->s->mesh_verts, tree->s->num_mesh_verts); //works
    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]);

    tree->bounds = V;

    tree->root = kd_tree_construct_rec(tree, V, kd_tree_gen_initial_tri_buf(tree), 0);
}

inline unsigned int _kd_tree_write_buf(char* buffer, unsigned int offset,
                                                    void* data, size_t size)
{
    memcpy(buffer+offset, data, size);
    return offset + size;
}

//returns finishing offset
unsigned int kd_tree_generate_serialized_buf_rec(kd_tree* tree, kd_tree_node* node, unsigned int offset)
{
    //NOTE: this could really just be two functions
    if(kd_tree_node_is_leaf(node)) // leaf
    {

        { //Leaf body
            _skd_tree_leaf_node l;
            l.type = KDTREE_LEAF;
            l.num_triangles = node->triangles.num_triangles;
            //printf("TEST %u \n", l.num_triangles);
            //assert(l.num_triangles != 0);
            offset = _kd_tree_write_buf(tree->buffer, offset, &l, sizeof(_skd_tree_leaf_node));
        }

```

```
        for(int i = 0; i < node->triangles.num_triangles; i++) //triangle indices
        {
            offset = _kd_tree_write_buf(tree->buffer, offset,
                                        node->triangles.triangle_buffer+i, sizeof(unsigned int));
        }
        if(node->triangles.num_triangles % 2)
            offset += 4;//if it isn't alligned with a long add 4 bytes (8 byte allignment)

        return offset;
    }
    else // traversal node
    {
        _skd_tree_traversal_node n;
        n.type = KDTREE_NODE;
        n.k = node->k;
        n.b = node->b;
        unsigned int struct_start_offset = offset;
        offset += sizeof(_skd_tree_traversal_node);

        unsigned int left_offset  = kd_tree_generate_serialized_buf_rec(tree, node->left, offset);
        //this goes after the left node
        unsigned int right_offset = kd_tree_generate_serialized_buf_rec(tree, node->right, left_offset);

        n.left_ind  = offset;
        n.right_ind = left_offset;

        memcpy(tree->buffer+struct_start_offset, &n, sizeof(_skd_tree_traversal_node));

        return right_offset;
    }
}

void kd_tree_generate_serialized(kd_tree* tree)
{
    if(tree->s->num_mesh_indices == 0)
    {
        printf("WARNING: Skipping k-d tree Serialization, num_mesh_indices is 0.\n");
        tree->buffer_size = 0;
        tree->buffer = malloc(1);
        return;
    }

    unsigned int mem_needed = 0;

    mem_needed += tree->num_traversal_nodes * sizeof(_skd_tree_traversal_node); //traversal nodes
    mem_needed += tree->num_leaves * sizeof(_skd_tree_leaf_node); //leaf nodes
    mem_needed += (tree->num_indices_total+tree->num_tris_padded) * sizeof(unsigned int); //triangle indices

    tree->buffer_size = mem_needed;
    printf("k-d tree is %d bytes long...", mem_needed);

    tree->buffer = malloc(mem_needed);
    kd_tree_generate_serialized_buf_rec(tree, tree->root, 0);

}
#include <loader.h>
#include <parson.h>
#include <vec.h>
#include <float.h>
#include <tinyobj_loader_c.h>
#include <assert.h>



#ifndef WIN32
#include <libproc.h>
#include <unistd.h>

#define FILE_SEP '/'

char* _get_os_pid_bin_path()
{
    static bool initialised = false;
    static char path[PROC_PIDPATHINFO_MAXSIZE];
    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);

```

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

```
        m->num_indices  =  shape.length*3;
        m->material_index    =  elem.mat_index;

        for(int f = 0; f < shape.length; f++)
        {
            //TODO: don't do this error check for each iteration
            if(elem.attrib.face_num_verts[f+shape.face_offset]!=3)
            {
                //This should never get called because the mesh gets triangulated when loaded.
                printf("Error: the obj loader only supports triangulated meshes!\n");
                exit(1);
            }
            for(int j = 0; j < 3; j++)
            {
                tinyobj_vertex_index_t face_index = elem.attrib.faces[(f+shape.face_offset)*3+j];

                vec3 vertex;
                vertex[0] = elem.attrib.vertices[3*face_index.v_idx+0];
                vertex[1] = elem.attrib.vertices[3*face_index.v_idx+1];
                vertex[2] = elem.attrib.vertices[3*face_index.v_idx+2];

                m->min[0] = vertex[0] < m->min[0] ? vertex[0] : m->min[0]; //X min
                m->min[1] = vertex[1] < m->min[1] ? vertex[1] : m->min[1]; //Y min
                m->min[2] = vertex[2] < m->min[2] ? vertex[2] : m->min[2]; //Z min

                m->max[0] = vertex[0] > m->max[0] ? vertex[0] : m->max[0]; //X max
                m->max[1] = vertex[1] > m->max[1] ? vertex[1] : m->max[1]; //Y max
                m->max[2] = vertex[2] > m->max[2] ? vertex[2] : m->max[2]; //Z max

                ivec3 index;
                index[0] = (*vert_offset)+face_index.v_idx;
                index[1] = (*nrml_offset)+face_index.vn_idx;
                index[2] = (*texcoord_offset)+face_index.vt_idx;
                out_scene->mesh_indices[(*index_offset)][0] = index[0];
                out_scene->mesh_indices[(*index_offset)][1] = index[1];
                out_scene->mesh_indices[(*index_offset)][2] = index[2];
                //Sorry to anyone reading this line...
                *((int*)out_scene->mesh_indices[(*index_offset)]+3) = (*mesh_offset)-1; //current mesh

                //xv3_cpy(out_scene->mesh_indices + (*index_offset), index);
                (*index_offset)++;
            }
        }
    }

    //__debugbreak();


    //GPU MEMORY ALIGNMENT FUN
    //NOTE: this is done because the gpu stores all vec3s 4 floats for memory alignment
    //      and it is actually faster if they are aligned like this even
    //      though it wastes more memory.
    for(int i = 0; i < elem.attrib.num_vertices; i++)
    {

        memcpy(out_scene->mesh_verts + (*vert_offset),
               elem.attrib.vertices+3*i,
               sizeof(float)*3); //evem though our buffer is alligned theres is
        (*vert_offset) += 1;
    }
    for(int i = 0; i < elem.attrib.num_normals; i++)
    {
        memcpy(out_scene->mesh_nrmls + (*nrml_offset),
               elem.attrib.normals+3*i,
               sizeof(float)*3);
        (*nrml_offset) += 1;
    }
    //NOTE: the texcoords are already aligned because they only have 2 elements.
    memcpy(out_scene->mesh_texcoords + (*texcoord_offset), elem.attrib.texcoords,
           elem.attrib.num_texcoords*sizeof(vec2));
    (*texcoord_offset) += elem.attrib.num_texcoords;
}

scene* load_scene_json(char* json)
{
    printf("Beginning scene loading...\n");
    scene* out_scene = (scene*) malloc(sizeof(scene));
    JSON_Value *root_value;
    JSON_Object *root_object;
    root_value = json_parse_string(json);
    root_object = json_value_get_object(root_value);


    //Name
    {
        const char* name = json_object_get_string(root_object, "name");
        printf("Scene name: %s\n", name);
    }

    //Version
    {//TODO: do something with this.
        int major  = (int)json_object_dotget_number(root_object, "version.major");
        int minor  = (int)json_object_dotget_number(root_object, "version.major");
        const char* type =      json_object_dotget_string(root_object, "version.type");
    }

    //Materials
    {
        JSON_Array* material_array = json_object_get_array(root_object, "materials");
        out_scene->num_materials   = json_array_get_count(material_array);
        out_scene->materials = (material*) malloc(out_scene->num_materials*sizeof(material));
        assert(out_scene->num_materials>0);
        for(int i = 0; i < out_scene->num_materials; i++)
        {
            JSON_Object* mat = json_array_get_object(material_array, i);
```

```
            xv_x(out_scene->materials[i].colour) = json_object_get_number(mat, "r");
            xv_y(out_scene->materials[i].colour) = json_object_get_number(mat, "g");
            xv_z(out_scene->materials[i].colour) = json_object_get_number(mat, "b");
            out_scene->materials[i].reflectivity = json_object_get_number(mat, "reflectivity");
        }
        printf("Materials: %d\n", out_scene->num_materials);
    }

    //Primitives
    {

        JSON_Object* primitive_object = json_object_get_object(root_object, "primitives");

        //Spheres
        {
            JSON_Array* sphere_array = json_object_get_array(primitive_object, "spheres");
            int num_spheres = json_array_get_count(sphere_array);

            out_scene->spheres = malloc(sizeof(sphere)*num_spheres);
            out_scene->num_spheres = num_spheres;

            for(int i = 0; i < num_spheres; i++)
            {
                JSON_Object* sphere = json_array_get_object(sphere_array, i);
                out_scene->spheres[i].pos[0] = json_object_get_number(sphere, "x");
                out_scene->spheres[i].pos[1] = json_object_get_number(sphere, "y");
                out_scene->spheres[i].pos[2] = json_object_get_number(sphere, "z");
                out_scene->spheres[i].radius = json_object_get_number(sphere, "radius");
                out_scene->spheres[i].material_index = json_object_get_number(sphere, "mat_index");
            }
            printf("Spheres: %d\n", out_scene->num_spheres);
        }

        //Planes
        {
            JSON_Array* plane_array = json_object_get_array(primitive_object, "planes");
            int num_planes = json_array_get_count(plane_array);

            out_scene->planes = malloc(sizeof(plane)*num_planes);
            out_scene->num_planes = num_planes;

            for(int i = 0; i < num_planes; i++)
            {
                JSON_Object* plane = json_array_get_object(plane_array, i);
                out_scene->planes[i].pos[0] = json_object_get_number(plane, "x");
                out_scene->planes[i].pos[1] = json_object_get_number(plane, "y");
                out_scene->planes[i].pos[2] = json_object_get_number(plane, "z");
                out_scene->planes[i].norm[0] = json_object_get_number(plane, "nx");
                out_scene->planes[i].norm[1] = json_object_get_number(plane, "ny");
                out_scene->planes[i].norm[2] = json_object_get_number(plane, "nz");

                out_scene->planes[i].material_index = json_object_get_number(plane, "mat_index");
            }
            printf("Planes: %d\n", out_scene->num_planes);
        }

    }


    //Meshes
    {
        JSON_Array* mesh_array = json_object_get_array(root_object, "meshes");

        int num_meshes = json_array_get_count(mesh_array);

        out_scene->num_meshes = 0;
        out_scene->num_mesh_verts = 0;
        out_scene->num_mesh_nrmls = 0;
        out_scene->num_mesh_texcoords = 0;
        out_scene->num_mesh_indices = 0;


        struct obj_list_elem* first = (struct obj_list_elem*) malloc(sizeof(struct obj_list_elem));
        struct obj_list_elem* current = first;

        //Pre evaluation
        for(int i = 0; i < num_meshes; i++)
        {
            JSON_Object* mesh = json_array_get_object(mesh_array, i);
            const char* url = json_object_get_string(mesh, "url");
            long length;
            char* data = load_file(url, &length);
            obj_pre_load(data, length, current, &out_scene->num_meshes, &out_scene->num_mesh_indices,
                         &out_scene->num_mesh_verts, &out_scene->num_mesh_nrmls,
                         &out_scene->num_mesh_texcoords);
            current->mat_index = (int) json_object_get_number(mesh, "mat_index");
            //mat4 model_mat;
            {
                //xm4_identity(model_mat);
                mat4 translation_mat;
                xm4_translatev(translation_mat,
                               json_object_get_number(mesh, "px"),
                               json_object_get_number(mesh, "py"),
                               json_object_get_number(mesh, "pz"));
                mat4 scale_mat;
                xm4_scalev(scale_mat,
                           json_object_get_number(mesh, "sx"),
                           json_object_get_number(mesh, "sy"),
                           json_object_get_number(mesh, "sz"));
                //TODO: add rotation.
                xm4_mul(current->model_mat, translation_mat, scale_mat);
            }
            free(data);

            if(i!=num_meshes-1) //messy but it works
```

```
            {
                current->next = (struct obj_list_elem*) malloc(sizeof(struct obj_list_elem));
                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);
}

void* os_get_bitmap_memory(os_abs abs)
{
    return (*abs.get_bitmap_memory_func)();
}

void os_draw_weird(os_abs abs)
{
    (*abs.draw_weird)();
}

int os_get_time_mili(os_abs abs)
{
    return (*abs.get_time_mili_func)();
}
```

```

int os_get_width(os_abs abs)
{
    return (*abs.get_width_func)();
}

int os_get_height(os_abs abs)
{
    return (*abs.get_height_func)();
}

void os_start_thread(os_abs abs, void (*func)(void*), void* data)
{
    (*abs.start_thread_func)(func, data);
}
#include <CL/opencl.h>
#include <raytracer.h>
//Parallel util.

void cl_info()
{

    int i, j;
    char* value;
    size_t valueSize;
    cl_uint platformCount;
    cl_platform_id* platforms;
    cl_uint deviceCount;
    cl_device_id* devices;
    cl_uint maxComputeUnits;
    cl_uint recommendedWorkgroupSize = 0;

    // get all platforms
    clGetPlatformIDs(0, NULL, &platformCount);
    platforms = (cl_platform_id*) malloc(sizeof(cl_platform_id) * platformCount);
    clGetPlatformIDs(platformCount, platforms, NULL);

    for (i = 0; i < platformCount; i++) {

        // get all devices
        clGetDeviceIDs(platforms[i], CL_DEVICE_TYPE_ALL, 0, NULL, &deviceCount);
        devices = (cl_device_id*) malloc(sizeof(cl_device_id) * deviceCount);
        clGetDeviceIDs(platforms[i], CL_DEVICE_TYPE_ALL, deviceCount, devices, NULL);

        // for each device print critical attributes
        for (j = 0; j < deviceCount; j++) {

            // print device name
            clGetDeviceInfo(devices[j], CL_DEVICE_NAME, 0, NULL, &valueSize);
            value = (char*) malloc(valueSize);
            clGetDeviceInfo(devices[j], CL_DEVICE_NAME, valueSize, value, NULL);
            printf("%i.%d. Device: %s\n", i, j+1, value);
            free(value);

            // print hardware device version
            clGetDeviceInfo(devices[j], CL_DEVICE_VERSION, 0, NULL, &valueSize);
            value = (char*) malloc(valueSize);
            clGetDeviceInfo(devices[j], CL_DEVICE_VERSION, valueSize, value, NULL);
            printf(" %i.%d.%d Hardware version: %s\n", i, j+1, 1, value);
            free(value);

            // print software driver version
            clGetDeviceInfo(devices[j], CL_DRIVER_VERSION, 0, NULL, &valueSize);
            value = (char*) malloc(valueSize);
            clGetDeviceInfo(devices[j], CL_DRIVER_VERSION, valueSize, value, NULL);
            printf(" %i.%d.%d Software version: %s\n", i, j+1, 2, value);
            free(value);

            // print c version supported by compiler for device
            clGetDeviceInfo(devices[j], CL_DEVICE_OPENCL_C_VERSION, 0, NULL, &valueSize);
            value = (char*) malloc(valueSize);
            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");
        exit(1);
    }


    // Create a compute context
    //
    ctx->context = clCreateContext(0, 1, &ctx->device_id, &pfn_notify, NULL, &err);
    if (!ctx->context)
    {
        printf("Error: Failed to create a compute context!\n");
        exit(1);
    }

    // Create a command commands
    //
    ctx->commands = clCreateCommandQueue(ctx->context, ctx->device_id, 0, &err);
    if (!ctx->commands)
    {
        printf("Error: Failed to create a command commands!\n");
        return;
    }
    ASRT_CL("Failed to Initialise OpenCL");

    { // num compute cores
        unsigned int id;
        clGetDeviceInfo(ctx->device_id, CL_DEVICE_VENDOR_ID, sizeof(unsigned int), &id, NULL);
        switch(id)
        {
        case(0x10DE): //NVIDIA
        {
            unsigned int warp_size;
            unsigned int compute_capability;
            unsigned int num_sm;
            unsigned int warps_per_sm;
            clGetDeviceInfo(ctx->device_id, CL_DEVICE_WARP_SIZE_NV, //warp size
                            sizeof(unsigned int), &warp_size, NULL);
            clGetDeviceInfo(ctx->device_id, CL_DEVICE_COMPUTE_CAPABILITY_MAJOR_NV, //compute capability
                            sizeof(unsigned int), &compute_capability, NULL);
            clGetDeviceInfo(ctx->device_id, CL_DEVICE_MAX_COMPUTE_UNITS, //number of stream multiprocessors
                            sizeof(unsigned int), &num_sm, NULL);

            switch(compute_capability)
            { //nvidia skipped 4 btw
            case 2: warps_per_sm = 1; break; //FERMI  (GK104/GK110)
            case 3: warps_per_sm = 6; break; //KEPLER (GK104/GK110) NOTE: ONLY 4 WARP SCHEDULERS THOUGH!
            case 5: warps_per_sm = 4; break; //Maxwell
            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
            case 7: warps_per_sm = 2; break; //Volta/Turing Might not be correct(NOTE: 16 FP32 PER CORE? what about warps?)
            }

            printf("NVIDIA INFO: SM: %d,  WARP SIZE: %d, COMPUTE CAPABILITY: %d, WARPS PER SM: %d, TOTAL STREAM PROCESSORS: %d\n\n",
```

```
                    num_sm, warp_size, compute_capability, warps_per_sm, warps_per_sm*warp_size*num_sm);
            ctx->simt_size = warp_size;
            ctx->num_simt_per_multiprocessor = warps_per_sm;
            ctx->num_multiprocessors = num_sm;
            ctx->num_cores = warps_per_sm*warp_size*num_sm;
            break;
        }
        case(0x1002): //AMD
        {
            printf("AMD GPU INFO NOT SUPPORTED YET!\n");
            break;
        }
        case(0x8086): //INTEL
        {
            printf("INTEL INFO NOT SUPPORTED YET!\n");
            break;
        }
        }

    }

}

cl_mem gen_rgb_image(raytracer_context* rctx,
                     const unsigned int width,
                     const unsigned int height)
{
    cl_image_desc cl_standard_descriptor;
    cl_image_format     cl_standard_format;
    cl_standard_format.image_channel_order     = CL_RGBA;
    cl_standard_format.image_channel_data_type = CL_FLOAT;

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

    int err;

    cl_mem img = clCreateImage(rctx->rcl->context,
                               CL_MEM_READ_WRITE,
                               &cl_standard_format,
                              &cl_standard_descriptor,
                               NULL,
                               &err);
    ASRT_CL("Couldn't Create OpenCL Texture");
    return img;
}

rcl_img_buf gen_1d_image_buffer(raytracer_context* rctx, size_t t, void* ptr)
{
    int err = CL_SUCCESS;


    rcl_img_buf ib;
    ib.size = t;

    ib.buffer = clCreateBuffer(rctx->rcl->context,
                               CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                               t,
                               ptr,
                               &err);
    ASRT_CL("Error Creating OpenCL ImageBuffer Buffer");


    cl_image_desc cl_standard_descriptor;
    cl_image_format     cl_standard_format;
    cl_standard_format.image_channel_order     = CL_RGBA;
    cl_standard_format.image_channel_data_type = CL_FLOAT; //prob should be float

    cl_standard_descriptor.image_type = CL_MEM_OBJECT_IMAGE1D_BUFFER;
    cl_standard_descriptor.image_width = t/4 == 0 ? 1 : t/sizeof(float)/4;
    cl_standard_descriptor.image_height = 0;
    cl_standard_descriptor.image_depth  = 0;
    cl_standard_descriptor.image_array_size  = 0;
    cl_standard_descriptor.image_row_pitch  = 0;
    cl_standard_descriptor.image_slice_pitch = 0;
    cl_standard_descriptor.num_mip_levels = 0;
    cl_standard_descriptor.num_samples = 0;
    cl_standard_descriptor.buffer = ib.buffer;


    ib.image = clCreateImage(rctx->rcl->context,
                             CL_MEM_READ_WRITE,
                             &cl_standard_format,
                             &cl_standard_descriptor,
                             NULL,//ptr,
                             &err);
    ASRT_CL("Error Creating OpenCL ImageBuffer Image");

    return ib;
}
cl_mem gen_1d_image(raytracer_context* rctx, size_t t, void* ptr)
{

    cl_image_desc cl_standard_descriptor;
    cl_image_format     cl_standard_format;
    cl_standard_format.image_channel_order     = CL_RGBA;
    cl_standard_format.image_channel_data_type = CL_FLOAT; //prob should be float

```

```
    cl_standard_descriptor.image_type = CL_MEM_OBJECT_IMAGE1D;
    cl_standard_descriptor.image_width = t/4 == 0 ? 1 : t/sizeof(float)/4;// t / 4 == 0 ? 1 : t / 4; //what?
    cl_standard_descriptor.image_height = 0;
    cl_standard_descriptor.image_depth  = 0;
    cl_standard_descriptor.image_array_size  = 0;
    cl_standard_descriptor.image_row_pitch  = 0;
    cl_standard_descriptor.image_slice_pitch = 0;
    cl_standard_descriptor.num_mip_levels = 0;
    cl_standard_descriptor.num_samples = 0;
    cl_standard_descriptor.buffer = NULL;


    int err = CL_SUCCESS;


    cl_mem img = clCreateImage(rctx->rcl->context,
                               CL_MEM_READ_WRITE | (/*ptr == NULL ? 0 :*/ CL_MEM_COPY_HOST_PTR),
                               &cl_standard_format,
                               &cl_standard_descriptor,
                               ptr,
                               &err);
    ASRT_CL("Couldn't Create OpenCL Texture");
    return img;
}
cl_mem gen_grayscale_buffer(raytracer_context* rctx,
                            const unsigned int width,
                            const unsigned int height)
{
    int err;

    cl_mem buf = clCreateBuffer(rctx->rcl->context, CL_MEM_READ_WRITE,
                                (width==0  ? rctx->width  : width)*
                                (height==0 ? rctx->height : height)*
                                sizeof(float),
                                NULL, &err);
    ASRT_CL("Couldn't Create OpenCL Float Buffer Image");
    return buf;
}

void retrieve_image(raytracer_context* rctx, cl_mem g_buf, void* c_buf,
                    const unsigned int width,
                    const unsigned int height)
{
    int err;
    size_t origin[3] = {0,0,0};
    size_t region[3] = {(width==0 ? rctx->width : width),
                        (height==0 ? rctx->height : height),
                        1};
    err = clEnqueueReadImage (rctx->rcl->commands,
                              g_buf,
                              CL_TRUE,
                              origin,
                              region,
                              0,
                              0,
                              c_buf,
                              0,
                              0,
                              NULL);
    ASRT_CL("Failed to retrieve Opencl Image");
}

void retrieve_buf(raytracer_context* rctx, cl_mem g_buf, void* c_buf, size_t size)
{
    int err;
    err = clEnqueueReadBuffer(rctx->rcl->commands, g_buf, CL_TRUE, 0,
                              size, c_buf,
                              0, NULL, NULL );
    ASRT_CL("Failed to retrieve Opencl Buffer");
}

void zero_buffer(raytracer_context* rctx, cl_mem buf, size_t size)
{
    int err;
    char pattern = 0;
    err =  clEnqueueFillBuffer (rctx->rcl->commands,
                                buf,
                                &pattern, 1 ,0,
                                size,
                                0, NULL, NULL);
    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);
```

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

```
    {
        buffer[length] = '\0';

        load_program_raw(ctx, buffer, kernels, num_kernels, program,
                         macros, num_macros);
    }

}

//NOTE: old
void test_sphere_raytracer(rcl_ctx* ctx, rcl_program* program,
        sphere* spheres, int num_spheres,
        uint32_t* bitmap, int width, int height)
{
    int err;

    static cl_mem tex;
    static cl_mem s_buf;
    static bool init = false; //temporary

    if(!init)
    {
        //New Texture
        tex = clCreateBuffer(ctx->context, CL_MEM_WRITE_ONLY,
                                    width*height*4, NULL, &err);

        //Spheres
        s_buf = clCreateBuffer(ctx->context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                               sizeof(float)*4*num_spheres, spheres, &err);
        if (err != CL_SUCCESS)
        {
            printf("Error: Failed to create Sphere Buffer! %d\n", err);
            return;
        }
        init = true;
    }
    else
    {
        clEnqueueWriteBuffer (    ctx->commands,
                                  s_buf,
                                  CL_TRUE,
                                  0,
                                  sizeof(float)*4*num_spheres,
                                  spheres,
                                  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.};
    float* result = matvec_mul(prctx->rctx->stat_scene->camera_world_matrix, zeroed);

    clSetKernelArg(kernel, 0, sizeof(cl_mem), &prctx->cl_path_fresh_frame_buffer);
    clSetKernelArg(kernel, 1, sizeof(cl_mem), &prctx->rctx->cl_ray_buffer);
    clSetKernelArg(kernel, 2, sizeof(cl_mem), &prctx->rctx->stat_scene->cl_material_buffer);
    clSetKernelArg(kernel, 3, sizeof(cl_mem), &prctx->rctx->stat_scene->cl_sphere_buffer);
    clSetKernelArg(kernel, 4, sizeof(cl_mem), &prctx->rctx->stat_scene->cl_plane_buffer);
    clSetKernelArg(kernel, 5, sizeof(cl_mem), &prctx->rctx->stat_scene->cl_mesh_buffer);
    clSetKernelArg(kernel, 6, sizeof(cl_mem), &prctx->rctx->stat_scene->cl_mesh_index_buffer.image);
    clSetKernelArg(kernel, 7, sizeof(cl_mem), &prctx->rctx->stat_scene->cl_mesh_vert_buffer.image);
    clSetKernelArg(kernel, 8, sizeof(cl_mem), &prctx->rctx->stat_scene->cl_mesh_nrml_buffer.image);

    clSetKernelArg(kernel, 9,  sizeof(int),     &prctx->rctx->width);
    clSetKernelArg(kernel, 10, sizeof(vec4),    result);
    clSetKernelArg(kernel, 11, sizeof(int),     &prctx->current_sample); //NOTE: I don't think this is used

    size_t global[2] = {x_div, y_div};

    //NOTE: tripping watchdog timer
    if(global[0]*WATCHDOG_DIVISIONS_X*global[1]*WATCHDOG_DIVISIONS_Y!=
       prctx->rctx->width*prctx->rctx->height)
    {
        printf("Watchdog divisions are incorrect!\n");
        exit(1);
    }

    size_t offset[2];

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

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


void path_raytracer_average_buffers(path_raytracer_context* prctx)
{
    int err;

    cl_kernel kernel = prctx->rctx->program->raw_kernels[F_BUFFER_AVG_KRNL_INDX];
    clSetKernelArg(kernel, 0, sizeof(cl_mem), &prctx->cl_path_output_buffer);
    clSetKernelArg(kernel, 1, sizeof(cl_mem), &prctx->cl_path_fresh_frame_buffer);
    clSetKernelArg(kernel, 2, sizeof(unsigned int), &prctx->rctx->width);
    clSetKernelArg(kernel, 3, sizeof(unsigned int), &prctx->rctx->height);
    clSetKernelArg(kernel, 4, sizeof(unsigned int), &prctx->num_samples);
    clSetKernelArg(kernel, 5, sizeof(unsigned int), &prctx->current_sample);

    size_t global;
    size_t local = get_workgroup_size(prctx->rctx, kernel);

    // Execute the kernel over the entire range of our 1d input data set
    // using the maximum number of work group items for this device
    //
    global =  prctx->rctx->width*prctx->rctx->height;
    err = clEnqueueNDRangeKernel(prctx->rctx->rcl->commands, kernel, 1, NULL,
                                 &global, NULL, 0, NULL, NULL);
    ASRT_CL("Failed to execute kernel");
    err = clFinish(prctx->rctx->rcl->commands);
    ASRT_CL("Something happened while waiting for kernel to finish");
}

void path_raytracer_push_path(path_raytracer_context* prctx)
{
    int err;

    cl_kernel kernel = prctx->rctx->program->raw_kernels[F_BUF_TO_BYTE_BUF_KRNL_INDX];
    clSetKernelArg(kernel, 0, sizeof(cl_mem), &prctx->rctx->cl_output_buffer);
    clSetKernelArg(kernel, 1, sizeof(cl_mem), &prctx->cl_path_output_buffer);
```

```
    clSetKernelArg(kernel, 2, sizeof(unsigned int), &prctx->rctx->width);
    clSetKernelArg(kernel, 3, sizeof(unsigned int), &prctx->rctx->height);



    size_t global;
    size_t local = get_workgroup_size(prctx->rctx, kernel);

    // Execute the kernel over the entire range of our 1d input data set
    // using the maximum number of work group items for this device
    //
    global =  prctx->rctx->width*prctx->rctx->height;
    err = clEnqueueNDRangeKernel(prctx->rctx->rcl->commands, kernel, 1,
                                 NULL, &global, NULL, 0, NULL, NULL);
    ASRT_CL("Failed to execute kernel");

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


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

}


void path_raytracer_render(path_raytracer_context* prctx)
{
    int local_start_time = os_get_time_mili(abst);
    prctx->current_sample++;
    if(prctx->current_sample>prctx->num_samples)
    {
        prctx->render_complete = true;
        printf("Render took %d ms\n", os_get_time_mili(abst)-prctx->start_time);
        return;
    }
    _raytracer_gen_ray_buffer(prctx->rctx);

    path_raytracer_path_trace(prctx);

    if(prctx->current_sample == 1) //needs to be here
    {
        int err;
        err = clEnqueueCopyBuffer (    prctx->rctx->rcl->commands,
                                   prctx->cl_path_fresh_frame_buffer,
                                   prctx->cl_path_output_buffer,
                                   0,
                                   0,
                                   prctx->rctx->width*prctx->rctx->height*sizeof(vec4),
                                   0,
                                   0,
                                   NULL);
        ASRT_CL("Error copying OpenCL Output Buffer");

        err = clFinish(prctx->rctx->rcl->commands);
        ASRT_CL("Something happened while waiting for copy to finish");
    }
    path_raytracer_average_buffers(prctx);
    path_raytracer_push_path(prctx);
    printf("Total time for sample group: %d\n", os_get_time_mili(abst)-local_start_time);
}

void path_raytracer_prepass(path_raytracer_context* prctx)
{
    raytracer_prepass(prctx->rctx); //Nothing Special
    prctx->current_sample = 0;
    prctx->start_time = os_get_time_mili(abst);
}
#include <raytracer.h>
#include <parallel.h>
//binary resources
#include <test.cl.h> //test kernel



//NOTE: we are assuming the output buffer will be the right size
raytracer_context* raytracer_init(unsigned int width, unsigned int height,
                                  uint32_t* output_buffer, rcl_ctx* rcl)
{
    raytracer_context* rctx = (raytracer_context*) malloc(sizeof(raytracer_context));
    rctx->width  = width;
    rctx->height = height;
    rctx->ray_buffer = (float*) malloc(width * height * sizeof(ray));
    rctx->output_buffer = output_buffer;
    //rctx->fresh_buffer = (uint32_t*) malloc(width * height * sizeof(uint32_t));
    rctx->rcl = rcl;
    rctx->program = (rcl_program*) malloc(sizeof(rcl_program));
    rctx->ic_ctx = (ic_context*) malloc(sizeof(ic_context));
    //ic_init(rctx);
    rctx->render_complete = false;
    rctx->num_samples     = 64; //NOTE: arbitrary default
    rctx->current_sample  = 0;
    rctx->event_position = 0;
    rctx->block_size_y = 0;
    rctx->block_size_x = 0;
    return rctx;
}

void raytracer_build_kernels(raytracer_context* rctx)
{
    printf("Building Kernels...\n");
```

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

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

```
#define WATCHDOG_DIVISIONS_Y 2
void _raytracer_path_trace(raytracer_context* rctx, unsigned int sample_num)
{
    int err;

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

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

    cl_kernel kernel = rctx->program->raw_kernels[PATH_TRACE_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_path_fresh_frame_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(int),     &rctx->width);
    clSetKernelArg(kernel, 10, sizeof(vec4),    result);
    clSetKernelArg(kernel, 11, sizeof(int),     &sample_num); //NOTE: I don't think this is used

    size_t global[2] = {x_div, y_div};

    //NOTE: tripping watchdog timer
    if(global[0]*WATCHDOG_DIVISIONS_X*global[1]*WATCHDOG_DIVISIONS_Y!=rctx->width*rctx->height)
    {
        printf("Watchdog divisions are incorrect!\n");
        exit(1);
    }

    size_t offset[2];

    for(int x = 0; x < WATCHDOG_DIVISIONS_X; x++)
    {
        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)
{
```

```
3674     int err;
3675 
3676     //initialise kd tree
3677     rctx->stat_scene->kdt = kd_tree_init();
3678 
3679 
3680     //Scene Buffers
3681     rctx->stat_scene->cl_sphere_buffer = clCreateBuffer(rctx->rcl->context,
3682                                                         CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
3683                                                         sizeof(sphere)*rctx->stat_scene->num_spheres,
3684                                                         rctx->stat_scene->spheres, &err);
3685     ASRT_CL("Error Creating OpenCL Scene Sphere Buffer.");
3686 
3687     rctx->stat_scene->cl_plane_buffer = clCreateBuffer(rctx->rcl->context,
3688                                                        CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
3689                                                        sizeof(plane)*rctx->stat_scene->num_planes,
3690                                                        rctx->stat_scene->planes, &err);
3691     ASRT_CL("Error Creating OpenCL Scene Plane Buffer.");
3692 
3693 
3694     rctx->stat_scene->cl_material_buffer = clCreateBuffer(rctx->rcl->context,
3695                                                           CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
3696                                                           sizeof(material)*
3697                                                           rctx->stat_scene->num_materials,
3698                                                           rctx->stat_scene->materials, &err);
3699     ASRT_CL("Error Creating OpenCL Scene Plane Buffer.");
3700 
3701 
3702     //Mesh
3703     rctx->stat_scene->cl_mesh_buffer = clCreateBuffer(rctx->rcl->context,
3704                                                       CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
3705                                                       rctx->stat_scene->num_meshes==0 ? 1 :
3706                                                       sizeof(mesh)*rctx->stat_scene->num_meshes,
3707                                                       rctx->stat_scene->meshes, &err);
3708     ASRT_CL("Error Creating OpenCL Scene Mesh Buffer.");
3709 
3710     //mesh data is stored as images for faster access
3711     rctx->stat_scene->cl_mesh_vert_buffer =
3712         gen_1d_image_buffer(rctx, rctx->stat_scene->num_mesh_verts==0 ? 1 :
3713                             sizeof(vec3)*rctx->stat_scene->num_mesh_verts,
3714                             rctx->stat_scene->mesh_verts);
3715 
3716     rctx->stat_scene->cl_mesh_nrml_buffer =
3717         gen_1d_image_buffer(rctx, rctx->stat_scene->num_mesh_nrmls==0 ? 1 :
3718                             sizeof(vec3)*rctx->stat_scene->num_mesh_nrmls,
3719                             rctx->stat_scene->mesh_nrmls);
3720 
3721     rctx->stat_scene->cl_mesh_index_buffer =
3722         gen_1d_image_buffer(rctx, rctx->stat_scene->num_mesh_indices==0 ? 1 :
3723                             sizeof(ivec3)*
3724                             rctx->stat_scene->num_mesh_indices,//maybe
3725                             rctx->stat_scene->mesh_indices);
3726 
3727 
3728 
3729 
3730 }
3731 
3732 
3733 void scene_resource_push(raytracer_context* rctx)
3734 {
3735     int err;
3736 
3737     //if(rctx->stat_scene->kdt->cl_kd_tree_buffer != NULL)
3738         //    exit(1);
3739     printf("Pushing Scene Resources...");
3740 
3741     printf("Serializing k-d tree...");
3742     kd_tree_generate_serialized(rctx->stat_scene->kdt);
3743 
3744 //NOTE: SUPER SCUFFED
3745     if(rctx->stat_scene->kdt->cl_kd_tree_buffer == NULL)
3746     {
3747         rctx->stat_scene->kdt->cl_kd_tree_buffer =
3748             clCreateBuffer(rctx->rcl->context,
3749                            CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
3750                            rctx->stat_scene->kdt->buffer_size,
3751                            rctx->stat_scene->kdt->buffer, &err);
3752         ASRT_CL("Couldn't create kd tree buffer.");
3753     }
3754     printf("Pushing Buffers...");
3755     if(rctx->stat_scene->meshes_changed)
3756     {
3757         clEnqueueWriteBuffer (    rctx->rcl->commands,
3758                                   rctx->stat_scene->cl_mesh_buffer,
3759                                   CL_TRUE,
3760                                   0,
3761                                   sizeof(mesh)*rctx->stat_scene->num_meshes,
3762                                   rctx->stat_scene->meshes,
3763                                   0,
3764                                   NULL,
3765                                   NULL);
3766     }
3767 
3768     if(rctx->stat_scene->spheres_changed)
3769     {
3770         clEnqueueWriteBuffer (    rctx->rcl->commands,
3771                                   rctx->stat_scene->cl_sphere_buffer,
3772                                   CL_TRUE,
3773                                   0,
3774                                   sizeof(sphere)*rctx->stat_scene->num_spheres,
3775                                   rctx->stat_scene->spheres,
3776                                   0,
3777                                   NULL,
3778                                   NULL);
```

```
    }

    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.");
```

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

```
3989     clSetKernelArg(kernel, 2, sizeof(unsigned int), &sprctx->rctx->width);
3990     //NOTE: WILL NOT WORK WITH ALL SITUATIONS:
3991     unsigned int num_rays = sprctx->rctx->width*sprctx->rctx->height;
3992 
3993 
3994     size_t global[1] = {num_rays};
3995 
3996     err = clEnqueueNDRangeKernel(sprctx->rctx->rcl->commands, kernel, 1,
3997                                  NULL, global, NULL, 0, NULL, NULL);
3998     ASRT_CL("Failed to execute kd tree traversal kernel");
3999 
4000     err = clFinish(sprctx->rctx->rcl->commands);
4001     ASRT_CL("Something happened while executing kd tree traversal kernel");
4002 
4003     err = clEnqueueReadBuffer(sprctx->rctx->rcl->commands, sprctx->rctx->cl_output_buffer, CL_TRUE, 0,
4004                               sprctx->rctx->width*sprctx->rctx->height*sizeof(int),
4005                               sprctx->rctx->output_buffer, 0, NULL, NULL );
4006     ASRT_CL("Failed to read output array");
4007 
4008 }
4009 
4010 void spath_raytracer_kd_test(spath_raytracer_context* sprctx)
4011 {
4012     int err;
4013 
4014     cl_kernel kernel = sprctx->rctx->program->raw_kernels[KDTREE_TEST_DRAW_INDX];
4015 
4016     clSetKernelArg(kernel, 0, sizeof(cl_mem), &sprctx->rctx->cl_output_buffer);
4017     clSetKernelArg(kernel, 1, sizeof(cl_mem), &sprctx->cl_path_collision_result_buffer);
4018 
4019     clSetKernelArg(kernel, 2, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_material_buffer);
4020     clSetKernelArg(kernel, 3, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_buffer);
4021     clSetKernelArg(kernel, 4, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_index_buffer.image);
4022     clSetKernelArg(kernel, 5, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_vert_buffer.image);
4023     clSetKernelArg(kernel, 6, sizeof(cl_mem), &sprctx->rctx->stat_scene->cl_mesh_nrml_buffer.image);
4024 
4025     clSetKernelArg(kernel, 7, sizeof(unsigned int), &sprctx->rctx->width);
4026     //NOTE: WILL NOT WORK WITH ALL SITUATIONS:
4027     unsigned int num_rays = sprctx->rctx->width*sprctx->rctx->height;
4028 
4029 
4030     size_t global[1] = {num_rays};
4031 
4032     err = clEnqueueNDRangeKernel(sprctx->rctx->rcl->commands, kernel, 1,
4033                                  NULL, global, NULL, 0, NULL, NULL);
4034     ASRT_CL("Failed to execute kd tree traversal kernel");
4035 
4036     err = clFinish(sprctx->rctx->rcl->commands);
4037     ASRT_CL("Something happened while executing kd tree test kernel");
4038 
4039     err = clEnqueueReadBuffer(sprctx->rctx->rcl->commands, sprctx->rctx->cl_output_buffer, CL_TRUE, 0,
4040                               sprctx->rctx->width*sprctx->rctx->height*sizeof(int),
4041                               sprctx->rctx->output_buffer, 0, NULL, NULL );
4042     ASRT_CL("Failed to read output array");
4043 }
4044 
4045 void spath_raytracer_xor_rng(spath_raytracer_context* sprctx)
4046 {
4047     int err;
4048     cl_kernel kernel = sprctx->rctx->program->raw_kernels[XORSHIFT_BATCH_INDX];
4049     clSetKernelArg(kernel, 0, sizeof(cl_mem), &sprctx->cl_random_buffer);
4050 
4051     size_t global = sprctx->rctx->width*sprctx->rctx->height;
4052 
4053     err = clEnqueueNDRangeKernel(sprctx->rctx->rcl->commands, kernel, 1, NULL,
4054                                  &global, NULL, 0, NULL, NULL);
4055     ASRT_CL("Failed to execute kernel");
4056     err = clFinish(sprctx->rctx->rcl->commands);
4057     ASRT_CL("Something happened while waiting for kernel to finish");
4058 }
4059 
4060 void spath_raytracer_avg_to_out(spath_raytracer_context* sprctx)
4061 {
4062     int err;
4063     int useless = 0;
4064     cl_kernel kernel = sprctx->rctx->program->raw_kernels[F_BUF_TO_BYTE_BUF_AVG_KRNL_INDX];
4065     clSetKernelArg(kernel, 0, sizeof(cl_mem), &sprctx->rctx->cl_output_buffer);
4066     clSetKernelArg(kernel, 1, sizeof(cl_mem), &sprctx->cl_path_output_buffer);
4067     clSetKernelArg(kernel, 2, sizeof(cl_mem), &sprctx->cl_spath_progress_buffer);
4068     clSetKernelArg(kernel, 3, sizeof(unsigned int), &sprctx->rctx->width);
4069 
4070     clSetKernelArg(kernel, 4, sizeof(unsigned int), &useless);
4071 
4072 
4073     size_t global = sprctx->rctx->width*sprctx->rctx->height;
4074 
4075     err = clEnqueueNDRangeKernel(sprctx->rctx->rcl->commands, kernel, 1, NULL,
4076                                  &global, NULL, 0, NULL, NULL);
4077     ASRT_CL("Failed to execute kernel");
4078     err = clFinish(sprctx->rctx->rcl->commands);
4079     ASRT_CL("Something happened while waiting for kernel to finish");
4080 
4081     err = clEnqueueReadBuffer(sprctx->rctx->rcl->commands, sprctx->rctx->cl_output_buffer, CL_TRUE, 0,
4082                               sprctx->rctx->width*sprctx->rctx->height*sizeof(int),
4083                               sprctx->rctx->output_buffer, 0, NULL, NULL );
4084     ASRT_CL("Failed to read output array");
4085 }
4086 
4087 
4088 void spath_raytracer_trace_init(spath_raytracer_context* sprctx)
4089 {
4090     int err;
4091     unsigned int random_value_WACKO = rand();
4092     cl_kernel kernel = sprctx->rctx->program->raw_kernels[SEGMENTED_PATH_TRACE_INIT_INDX];
4093 
```

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

```
void spath_raytracer_prepass(spath_raytracer_context* sprctx)
{
    printf("Starting Split Path Raytracer Prepass. \n");
    sprctx->render_complete = false;
    sprctx->num_iterations = 256*4;//arbitrary default
    srand((unsigned int)os_get_time_mili(abst));
    sprctx->start_time = (unsigned int) os_get_time_mili(abst);
    bad_buf_update(sprctx);


    zero_buffer(sprctx->rctx, sprctx->cl_path_output_buffer,
                sprctx->rctx->width*sprctx->rctx->height*sizeof(vec4));

    raytracer_prepass(sprctx->rctx);

    sprctx->current_iteration = 0;
    zero_buffer(sprctx->rctx, sprctx->cl_spath_progress_buffer,
                sprctx->rctx->width*sprctx->rctx->height*sizeof(spath_progress));
    _raytracer_gen_ray_buffer(sprctx->rctx);



    spath_raytracer_kd_collision(sprctx);

    spath_raytracer_trace_init(sprctx);

    spath_raytracer_update_random(sprctx);

    zero_buffer(sprctx->rctx, sprctx->rctx->cl_ray_buffer,
                sprctx->rctx->width*sprctx->rctx->height*sizeof(ray));

    printf("Finished Split Path Raytracer Prepass. \n");
}
#include <ss_raytracer.h>
#include <scene.h>
#include <kdtree.h>
#include <raytracer.h>

//Single sweep, as close to real time as this thing can support.
void ss_raytracer_render(ss_raytracer_context* srctx)
{
    int err;
    int start_time = os_get_time_mili(abst);

    //TODO: @REFACTOR and remove prefix underscore and move to prepass
    _raytracer_gen_ray_buffer(srctx->rctx);


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

    float zeroed[] = {0., 0., 0., 1.};
    float* result = matvec_mul(srctx->rctx->stat_scene->camera_world_matrix, zeroed);
    clSetKernelArg(kernel, 0, sizeof(cl_mem), &srctx->rctx->cl_output_buffer);
    clSetKernelArg(kernel, 1, sizeof(cl_mem), &srctx->rctx->cl_ray_buffer);
    clSetKernelArg(kernel, 2, sizeof(cl_mem), &srctx->rctx->stat_scene->cl_material_buffer);
    clSetKernelArg(kernel, 3, sizeof(cl_mem), &srctx->rctx->stat_scene->cl_sphere_buffer);
    clSetKernelArg(kernel, 4, sizeof(cl_mem), &srctx->rctx->stat_scene->cl_plane_buffer);
    clSetKernelArg(kernel, 5, sizeof(cl_mem), &srctx->rctx->stat_scene->cl_mesh_buffer);
    clSetKernelArg(kernel, 6, sizeof(cl_mem), &srctx->rctx->stat_scene->cl_mesh_index_buffer.image);
    clSetKernelArg(kernel, 7, sizeof(cl_mem), &srctx->rctx->stat_scene->cl_mesh_vert_buffer.image);
    clSetKernelArg(kernel, 8, sizeof(cl_mem), &srctx->rctx->stat_scene->cl_mesh_nrml_buffer.image);
    clSetKernelArg(kernel, 9, sizeof(unsigned int), &srctx->rctx->width);
    clSetKernelArg(kernel, 10, sizeof(unsigned int), &srctx->rctx->height);
    clSetKernelArg(kernel, 11, sizeof(float)*4, result); //we only need 3
    //free(result);

    size_t global;

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

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

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

    printf("SS Render Took %d ms.\n", os_get_time_mili(abst)-start_time);
}

ss_raytracer_context* init_ss_raytracer_context(struct _rt_ctx* rctx)
{
    ss_raytracer_context* ssctx = malloc(sizeof(ss_raytracer_context));

    ssctx->rctx = rctx;
    ssctx->up_to_date = false;
    return ssctx;
}


//NOTE: @REFACTOR not used anymore should delete
rt_vtable get_ss_raytracer_vtable()//TODO: don't use tbh.
{
    rt_vtable v;
    v.up_to_date = false;
    //v.build      = &ss_raytracer_build;
    v.pre_pass     =&ss_raytracer_prepass;
    v.render_frame = &ss_raytracer_render;
    return v;
}

```

```
void ss_raytracer_build(ss_raytracer_context* srctx)
{
    raytracer_build(srctx->rctx); //nothing special
}

void ss_raytracer_prepass(ss_raytracer_context* srctx)
{
    raytracer_prepass(srctx->rctx); //Nothing Special
}
#include <os_abs.h>
#include <stdint.h>
#include <startup.h>
#include <stdio.h>
#include <raytracer.h>
#include <mongoose.h>

#include <ui.h>
#include <ss_raytracer.h>
#include <path_raytracer.h>
#include <spath_raytracer.h>

#ifdef WIN32
#include <win32.h>
#else
#include <osx.h>
#include <stdio.h>
#include <termios.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/time.h>
#endif

//#include <time.h>
#define _USE_MATH_DEFINES
#include <math.h>
#include <geom.h>
#include <parallel.h>
#include <loader.h>
#define NUM_SPHERES 5
#define NUM_PLANES  1

#define STRFY(x) #x
#define DBL_STRFY(x) STRFY(x)

os_abs abst;

#ifndef _WIN32
char kbhit()
{
    static char initialised = false;
    //NOTE: we are never going to need to actually echo the characters
    if(!initialised)
    {
        initialised = true;
        struct termios term, old;
        tcgetattr(STDIN_FILENO, &old);
        term = old;
        term.c_lflag &= -(ICANON | ECHO);
        tcsetattr(STDIN_FILENO, TCSANOW, &term);
    }
    struct timeval tv;
    fd_set rdfs;

    tv.tv_sec = 0;
    tv.tv_usec = 0;

    FD_ZERO(&rdfs);
    FD_SET(STDIN_FILENO, &rdfs);

    select(STDIN_FILENO+1, &rdfs, NULL, NULL, &tv);
    return FD_ISSET(STDIN_FILENO, &rdfs);
}
#endif


bool should_run = true;
bool should_pause = false;
void loop_exit()
{
    should_run = false;
}

void loop_pause()
{
    should_pause = !should_pause;
}


void run(void* unnused_rn)
{


    const int width = os_get_width(abst);
    const int height = os_get_height(abst);

    const int pitch = width *4;
    uint32_t* row = (uint32_t*)os_get_bitmap_memory(abst);

    cl_info();

    rcl_ctx* rcl = (rcl_ctx*) malloc(sizeof(rcl_ctx));
    create_context(rcl);

    raytracer_context* rctx = raytracer_init((unsigned int)width, (unsigned int)height,
                                             row, rcl);
```

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

```
4514         }
4515         os_update(abst);
4516     }
4517 
4518     //all of below shouldn't be a thing.
4519 
4520     raytracer_build(rctx);
4521     raytracer_prepass(rctx);
4522 
4523     xm4_identity(rctx->stat_scene->camera_world_matrix);
4524 
4525     float dist = 0.f;
4526 
4527 
4528     int _timer_store = 0;
4529     int _timer_counter = 0;
4530     float _timer_average = 0.0f;
4531     printf("Rendering:\n\n");
4532 
4533     /* static float t = 0.0f; */
4534     /* t += 0.0005f; */
4535     /* dist = sin(t)+1; */
4536     /* //mat4 temp; */
4537     /* xm4_translatev(rctx->stat_scene->camera_world_matrix, 0, dist, 0); */
4538     int real_start = os_get_time_mili(abst);
4539     while(should_run)
4540     {
4541 
4542         if(should_pause)
4543             continue;
4544         int last_time = os_get_time_mili(abst);
4545 
4546         if(kbhit())
4547         {
4548             switch (getc(stdin))
4549             {
4550             case 'c':
4551                 exit(1);
4552                 break;
4553             case 27: //ESCAPE
4554                 exit(1);
4555                 break;
4556             default:
4557                 break;
4558             }
4559         }
4560 
4561         //raytracer_refined_render(rctx);
4562         raytracer_render(rctx);
4563         if(rctx->render_complete)
4564         {
4565             printf("\n\nRender took: %02i ms (%d samples)\n\n",
4566                    os_get_time_mili(abst)-real_start, rctx->num_samples);
4567             break;
4568         }
4569 
4570 
4571         int mili = os_get_time_mili(abst)-last_time;
4572         _timer_store += mili;
4573         _timer_counter++;
4574         printf("\rFrame took: %02i ms, average per 20 frames: %0.2f, avg fps: %03.2f (%d/%d)    ",
4575                mili, _timer_average, 1000.0f/_timer_average,
4576                rctx->current_sample, rctx->num_samples);
4577         fflush(stdout);
4578         if(_timer_counter>20)
4579         {
4580             _timer_counter = 0;
4581             _timer_average = (float)(_timer_store)/20.f;
4582             _timer_store = 0;
4583         }
4584         os_update(abst);
4585     }
4586 
4587 
4588 }
4589 
4590 int startup() //main function called from win32 abstraction
4591 {
4592 #ifdef WIN32
4593     abst = init_win32_abs();
4594 #else
4595     abst = init_osx_abs();
4596 #endif
4597     os_start(abst);
4598     os_start_thread(abst, run, NULL);
4599     //win32_start_thread(run, NULL);
4600 
4601     os_loop_start(abst);
4602     return 0;
4603     /*
4604     printf("Hello World\n");
4605     testWin32();
4606     return 0;*/
4607 }
4608 #include <ui.h>
4609 #include <ui_web.h> //TODO: rename to ui_data or something
4610 #include <mongoose.h>
4611 #include <parson.h>
4612 #include <raytracer.h>
4613 
4614 static ui_ctx uctx;
4615 
4616 //Mostly based off of the exampel code for the library.
4617 
4618 
```

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

```
        break;
    }
    case MG_EV_WEBSOCKET_HANDSHAKE_DONE:
    {
        printf("Webscoket Handshake\n");
        break;
    }
    case MG_EV_WEBSOCKET_FRAME:
    {
        struct websocket_message *wm = (struct websocket_message *) ev_data;
        /* New websocket message. Tell everybody. */
        //struct mg_str d = {(char *) wm->data, wm->size};
        //printf("WOW K: %s\n", wm->data);
        handle_ws_request(c, wm->data);
        break;
    }
    }

    //printf("TEST 3\n");
    //c->flags |= MG_F_SEND_AND_CLOSE;
}

static void handle_ocp_li(struct mg_connection *c, int ev, void* ev_data) {
    if (ev == MG_EV_HTTP_REQUEST) {
        struct http_message *hm = (struct http_message *) ev_data;

        // We have received an HTTP request. Parsed request is contained in `hm`.
        // Send HTTP reply to the client which shows full original request.
        mg_send_head(c, 200, ___src_ui_ocp_li_woff_len, "Content-Type: application/font-woff");
        //c->send_mbuf = ___src_ui_ocp_li_woff;
        //c->content_len = ___src_ui_ocp_li_woff_len;

        mg_send(c, ___src_ui_ocp_li_woff, ___src_ui_ocp_li_woff_len);
        //mg_printf(c, "%.*s", (int)___src_ui_ocp_li_woff_len, ___src_ui_ocp_li_woff);
    }
    //printf("TEST 2\n");
    c->flags |= MG_F_SEND_AND_CLOSE;
}


static void handle_style(struct mg_connection* c, int ev, void* ev_data) {
    if (ev == MG_EV_HTTP_REQUEST) {
        struct http_message *hm = (struct http_message *) ev_data;

        // We have received an HTTP request. Parsed request is contained in `hm`.
        // Send HTTP reply to the client which shows full original request.
        mg_send_head(c, 200, ___src_ui_style_css_len, "Content-Type: text/css");
        mg_printf(c, "%.*s", (int)___src_ui_style_css_len, ___src_ui_style_css);
    }
    //printf("TEST\n");
    c->flags |= MG_F_SEND_AND_CLOSE;
}

void web_server_start(void* rctx)
{
    uctx.rctx = rctx;
    struct mg_mgr mgr;
    struct mg_connection *c;

    mg_mgr_init(&mgr, NULL);
    c = mg_bind(&mgr, s_http_port, ev_handler);
    mg_set_protocol_http_websocket(c);
    mg_register_http_endpoint(c, "/ocp_li.woff", handle_ocp_li);
    mg_register_http_endpoint(c, "/style.css", handle_style);
    mg_register_http_endpoint(c, "/ws", handle_ws);

    printf("Web UI Hosted On Port %s\n", s_http_port);

    for (;;) {
        mg_mgr_poll(&mgr, 1000);
    }
    mg_mgr_free(&mgr);

    exit(1);

}
#include <win32.h>
#include <startup.h>
#include <windows.h>
#include <math.h>
#include <stdio.h>
#include <stdint.h>
#include <assert.h>
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
const char CLASS_NAME[] = "Raytracer";


static win32_context* ctx;

void win32_draw_meme(); //vague predef

os_abs init_win32_abs()
{
    os_abs abstraction;
    abstraction.start_func = &win32_start;
    abstraction.loop_start_func = &win32_loop;
    abstraction.update_func = &win32_update;
    abstraction.sleep_func = &win32_sleep;
    abstraction.get_bitmap_memory_func = &win32_get_bitmap_memory;
    abstraction.get_time_mili_func = &win32_get_time_mili;
    abstraction.get_width_func = &win32_get_width;
    abstraction.get_height_func = &win32_get_height;
    abstraction.start_thread_func = &win32_start_thread;
```

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

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

```
{
    printf("Starting WIN32 Window Loop\n");
    MSG msg;
    ctx->shouldRun = true;
    while(ctx->shouldRun)
    {
#ifdef WIN32_SHOULD_BLOCK_LOOP


        if(GetMessage(&msg, 0, 0, 0) > 0)
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }

#else
        while(PeekMessage(&msg, 0, 0, 0, PM_REMOVE))
        {
            if(msg.message == WM_QUIT)
            {
                ctx->shouldRun = false;
            }
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
#endif
        //win32_draw_meme();
        //win32_update_window();
    }
}


void create_win32_window()
{
    printf("Creating WIN32 Window\n");

    ctx->win = CreateWindowEx(
        0,
        CLASS_NAME,
        CLASS_NAME,
        /* WS_OVERLAPPEDWINDOW, */
        (WS_POPUP| WS_SYSMENU | WS_MAXIMIZEBOX | WS_MINIMIZEBOX),
        CW_USEDEFAULT, CW_USEDEFAULT, 1920, 1080,
        NULL, NULL, ctx->instance, NULL);

    if(ctx->win == NULL)
    {
        MessageBox(NULL, "Window Creation Failed!", "Error!",
                   MB_ICONEXCLAMATION | MB_OK);
        return;
    }

    ShowWindow(ctx->win, ctx->nCmdShow);
    UpdateWindow(ctx->win);

}


//NOTE: Should the start func start the loop
//#define WIN32_SHOULD_START_LOOP_ON_START
void win32_start()
{
    if(!RegisterClassEx(&ctx->wc))
    {
        MessageBox(NULL, "Window Registration Failed!", "Error!",
                   MB_ICONEXCLAMATION | MB_OK);
        return;
    }
    create_win32_window();
#ifdef WIN32_SHOULD_START_LOOP_ON_START
    win32_loop();
#endif

}

int win32_get_time_mili()
{
    SYSTEMTIME st;
    GetSystemTime(&st);
    return (int) st.wMilliseconds+(st.wSecond*1000)+(st.wMinute*1000*60);
}

void win32_update()
{
    //RECT win_rect;
    //GetClientRect(ctx->win, &win_rect);
    HDC dc = GetDC(ctx->win);
    win32_update_window(dc, ctx->win, ctx->width, ctx->height);
    ReleaseDC(ctx->win, dc);

}


int win32_get_width()
{
    return ctx->width;
}

int win32_get_height()
{
    return ctx->height;
}

void* win32_get_bitmap_memory()
{
```

```
    return ctx->bitmap_memory;
}


typedef struct
{
    void* data;
    void (*func)(void*);
} thread_func_meta;

DWORD WINAPI thread_func(void* data)
{
    if(!SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_HIGHEST))
    {
        DWORD dwError;
        dwError = GetLastError();
        printf(TEXT("Failed to change thread priority (%d)\n"), dwError);
    }

    thread_func_meta* meta = (thread_func_meta*) data;
    (meta->func)(meta->data); //confusing syntax: call the passed function with the passed data
    free(meta);
    return 0;
}

void win32_start_thread(void (*func)(void*), void* data)
{
    thread_func_meta* meta = (thread_func_meta*) malloc(sizeof(thread_func_meta));
    meta->data = data;
    meta->func = func;
    HANDLE t = CreateThread(NULL, 0, thread_func, meta, 0, NULL);
    //if(SetThreadPriority(t, THREAD_PRIORITY_HIGHEST)==0)
    //    assert(false);

}
/*********/
/* Types */
/*********/

#define MESH_SCENE_DATA_PARAM image1d_buffer_t indices, image1d_buffer_t vertices, image1d_buffer_t normals
#define MESH_SCENE_DATA       indices, vertices, normals

typedef struct //16 bytes
{
    vec3 colour;

    float reflectivity;
} __attribute__ ((aligned (16))) material;

typedef struct
{
    vec3 orig;
    vec3 dir;
} ray;

typedef struct
{
    bool did_hit;
    vec3 normal;
    vec3 point;
    float dist;
    material mat;
} collision_result;

typedef struct //32 bytes (one word)
{
    vec3 pos;
    //4 bytes padding
    float radius;
    int material_index;
    //8 bytes padding
} __attribute__ ((aligned (16))) sphere;

typedef struct plane
{
    vec3 pos;
    vec3 normal;

    int material_index;
} __attribute__ ((aligned (16))) plane;

typedef struct
{

    mat4 model;

    vec3 max;
    vec3 min;

    int index_offset;
    int num_indices;


    int material_index;
} __attribute__((aligned (32))) mesh; //TODO: align with cpu NOTE: I don't think we need 32

typedef struct
{
    const __global material* material_buffer;
    const __global sphere* spheres;
    const __global plane* planes;
    //Mesh
    const __global mesh* meshes;
} scene;

```

```
bool getTBoundingBox(vec3 vmin, vec3 vmax,
                     ray r, float* tmin, float* tmax) //NOTE: could be wrong
{

    vec3 invD = 1/r.dir;///vec3(1/dir.x, 1/dir.y, 1/dir.z);
    vec3 t0s = (vmin - r.orig) * invD;
      vec3 t1s = (vmax - r.orig) * invD;

      vec3 tsmaller = min(t0s, t1s);
    vec3 tbigger  = max(t0s, t1s);

    *tmin = max(*tmin, max(tsmaller.x, max(tsmaller.y, tsmaller.z)));
    *tmax = min(*tmax, min(tbigger.x,  min(tbigger.y, tbigger.z)));

    return (*tmin < *tmax);

    /* vec3 tmin = (vmin - r.orig) / r.dir; */
    /* vec3 tmax = (vmax - r.orig) / r.dir; */

    /* vec3 real_min = min(tmin, tmax); */
    /* vec3 real_max = max(tmin, tmax); */

    /* vec3 minmax = min(min(real_max.x, real_max.y), real_max.z); */
    /* vec3 maxmin = max(max(real_min.x, real_min.y), real_min.z); */

    /* if (dot(minmax,minmax) >= dot(maxmin, maxmin)) */
    /* { */
    /*     *t_min = sqrt(dot(maxmin,maxmin)); */
    /*     *t_max = sqrt(dot(minmax,minmax)); */
    /*     return (dot(maxmin, maxmin) > 0.001f ? true : false); */
    /* } */
    /* else return false; */
}


bool hitBoundingBox(vec3 vmin, vec3 vmax,
                    ray r)
{
    vec3 tmin = (vmin - r.orig) / r.dir;
    vec3 tmax = (vmax - r.orig) / r.dir;

    vec3 real_min = min(tmin, tmax);
    vec3 real_max = max(tmin, tmax);

    vec3 minmax = min(min(real_max.x, real_max.y), real_max.z);
    vec3 maxmin = max(max(real_min.x, real_min.y), real_min.z);

    if (dot(minmax,minmax) >= dot(maxmin, maxmin))
    { return (dot(maxmin, maxmin) > 0.001f ? true : false); }
    else return false;
}



/*********************/
/*                   */
/*     Primitives    */
/*                   */
/*********************/

/************/
/* Triangle */
/************/

//Moller-Trumbore
//t u v = x y z
bool does_collide_triangle(vec3 tri[4], vec3* hit_coords, ray r) //tri has extra for padding
{

    vec3 ab = tri[1] - tri[0];
    vec3 ac = tri[2] - tri[0];

    vec3 pvec = cross(r.dir, ac); //Triple product
    float det = dot(ab, pvec);

    if (det < EPSILON) // Behind or close to parallel.
        return false;

    float invDet = 1.f / det;
    vec3 tvec = r.orig - tri[0];

    //u
    hit_coords->y = dot(tvec, pvec) * invDet;
    if(hit_coords->y < 0 || hit_coords->y > 1)
        return false;

    //v
    vec3 qvec = cross(tvec, ab);
    hit_coords->z = dot(r.dir, qvec) * invDet;
    if (hit_coords->z < 0 || hit_coords->y + hit_coords->z > 1)
        return false;

    //t
    hit_coords->x = dot(ac, qvec) * invDet;


    return true; //goose
}


/**********/
/* Sphere */
/**********/

bool does_collide_sphere(sphere s, ray r, float *dist)
```

```
{
    float t0, t1; // solutions for t if the ray intersects

    // analytic solution
    vec3 L = s.pos- r.orig;
    float b = dot(r.dir, L) ;//* 2.0f;
    float c = dot(L, L) - (s.radius*s.radius); //NOTE: you can optimize out the square.

    float disc = b * b - c/**a*/; /* discriminant of quadratic formula */

    /* solve for t (distance to hitpoint along ray) */
    float t = false;

    if (disc < 0.0f) return false;
    else t = b - sqrt(disc);

    if (t < 0.0f)
    {
        t = b + sqrt(disc);
        if (t < 0.0f) return false;
    }
    *dist = t;
    return true;
}



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

bool does_collide_plane(plane p, ray r, float *dist)
{
    float denom = dot(r.dir, p.normal);
    if (denom < EPSILON) //Counter intuitive.
    {
        vec3 l = p.pos - r.orig;
        float t = dot(l, p.normal) / denom;
        if (t >= 0)
        {
            *dist = t;
            return true;
        }

    }
    return false;
}


/********************/
/*                  */
/*      Meshes      */
/*                  */
/********************/


bool does_collide_with_mesh(mesh collider, ray r, vec3* normal, float* dist, scene s,
                            MESH_SCENE_DATA_PARAM)
{
    //TODO: k-d trees
    *dist = FAR_PLANE;
    float min_t = FAR_PLANE;
    vec3 hit_coord; //NOTE: currently unused
    ray r2 = r;
    if(!hitBoundingBox(collider.min, collider.max, r))
    {
        return false;
    }

    for(int i = 0; i < collider.num_indices/3; i++) // each ivec3
    {
        vec3 tri[4];

        //get vertex (first element of each index)

        int4 idx_0 = read_imagei(indices, i*3+collider.index_offset+0);
        int4 idx_1 = read_imagei(indices, i*3+collider.index_offset+1);
        int4 idx_2 = read_imagei(indices, i*3+collider.index_offset+2);

        tri[0] = read_imagef(vertices, idx_0.x).xyz;
        tri[1] = read_imagef(vertices, idx_1.x).xyz;
        tri[2] = read_imagef(vertices, idx_2.x).xyz;

        //printf("%d/%d: (%f, %f, %f)\n", idx_0.x, collider.num_indices/3, tri[0].x, tri[0].y, tri[0].z);
        //printf("%d/%d: (%f, %f, %f)\n", idx_1.x, collider.num_indices/3, tri[1].x, tri[1].y, tri[1].z);

        vec3 bc_hit_coords = (vec3)(0.f); //t u v = x y z
        if(does_collide_triangle(tri, &bc_hit_coords, r) &&
           bc_hit_coords.x<min_t && bc_hit_coords.x>0)
        {
            min_t = bc_hit_coords.x; //t (distance along direction)
            *normal =
                read_imagef(normals, idx_0.y).xyz*(1-bc_hit_coords.y-bc_hit_coords.z)+
                read_imagef(normals, idx_1.y).xyz*bc_hit_coords.y+
                read_imagef(normals, idx_2.y).xyz*bc_hit_coords.z;
                //break; //convex optimization
        }

    }


    *dist = min_t;
    return min_t != FAR_PLANE;

}
```

```
5459
5460 bool does_collide_with_mesh_nieve(mesh collider, ray r, vec3* normal, float* dist, scene s,
5461                                   image1d_buffer_t tree, MESH_SCENE_DATA_PARAM)
5462 {
5463         //TODO: k-d trees
5464     *dist = FAR_PLANE;
5465     float min_t = FAR_PLANE;
5466     vec3 hit_coord; //NOTE: currently unused
5467     ray r2 = r;
5468     if(!hitBoundingBox(collider.min, collider.max, r))
5469     {
5470         return false;
5471     }
5472
5473     for(int i = 0; i < collider.num_indices/3; i++) // each ivec3
5474     {
5475         vec3 tri[4];
5476
5477         //get vertex (first element of each index)
5478
5479         int4 idx_0 = read_imagei(indices, i*3+collider.index_offset+0);
5480         int4 idx_1 = read_imagei(indices, i*3+collider.index_offset+1);
5481         int4 idx_2 = read_imagei(indices, i*3+collider.index_offset+2);
5482
5483         tri[0] = read_imagef(vertices, idx_0.x).xyz;
5484         tri[1] = read_imagef(vertices, idx_1.x).xyz;
5485         tri[2] = read_imagef(vertices, idx_2.x).xyz;
5486
5487         //printf("%d/%d: (%f, %f, %f)\n", idx_0.x, collider.num_indices/3, tri[0].x, tri[0].y, tri[0].z);
5488         //printf("%d/%d: (%f, %f, %f)\n", idx_1.x, collider.num_indices/3, tri[1].x, tri[1].y, tri[1].z);
5489
5490         vec3 bc_hit_coords = (vec3)(0.f); //t u v = x y z
5491         if(does_collide_triangle(tri, &bc_hit_coords, r) &&
5492            bc_hit_coords.x<min_t && bc_hit_coords.x>0)
5493         {
5494             min_t = bc_hit_coords.x; //t (distance along direction)
5495             *normal =
5496                 read_imagef(normals, idx_0.y).xyz*(1-bc_hit_coords.y-bc_hit_coords.z)+
5497                 read_imagef(normals, idx_1.y).xyz*bc_hit_coords.y+
5498                 read_imagef(normals, idx_2.y).xyz*bc_hit_coords.z;
5499                 //break; //convex optimization
5500         }
5501
5502     }
5503
5504
5505     *dist = min_t;
5506     return min_t != FAR_PLANE;
5507 }
5508
5509 bool does_collide_with_mesh_alt(mesh collider, ray r, vec3* normal, float* dist, scene s,
5510                                 MESH_SCENE_DATA_PARAM)
5511 {
5512     *dist = FAR_PLANE;
5513     float min_t = FAR_PLANE;
5514     vec3 hit_coord; //NOTE: currently unused
5515     ray r2 = r;
5516
5517     for(int i = 0; i < SCENE_NUM_INDICES/3; i++)
5518     {
5519         vec3 tri[4];
5520
5521         //get vertex (first element of each index)
5522
5523         int4 idx_0 = read_imagei(indices, i*3+collider.index_offset+0);
5524         int4 idx_1 = read_imagei(indices, i*3+collider.index_offset+1);
5525         int4 idx_2 = read_imagei(indices, i*3+collider.index_offset+2);
5526
5527         tri[0] = read_imagef(vertices, idx_0.x).xyz;
5528         tri[1] = read_imagef(vertices, idx_1.x).xyz;
5529         tri[2] = read_imagef(vertices, idx_2.x).xyz;
5530
5531
5532         vec3 bc_hit_coords = (vec3)(0.f); //t u v = x y z
5533         if(does_collide_triangle(tri, &bc_hit_coords, r) &&
5534            bc_hit_coords.x<min_t && bc_hit_coords.x>0)
5535         {
5536                 min_t = bc_hit_coords.x; //t (distance along direction)
5537                 *normal =
5538                     read_imagef(normals, idx_0.y).xyz*(1-bc_hit_coords.y-bc_hit_coords.z)+
5539                     read_imagef(normals, idx_1.y).xyz*bc_hit_coords.y+
5540                     read_imagef(normals, idx_2.y).xyz*bc_hit_coords.z;
5541         }
5542
5543     }
5544
5545
5546     *dist = min_t;
5547     return min_t != FAR_PLANE;
5548
5549 }
5550
5551
5552
5553 /************************/
5554 /* High Level Collision */
5555 /************************/
5556
5557
5558 bool collide_meshes(ray r, collision_result* result, scene s, MESH_SCENE_DATA_PARAM)
5559 {
5560
5561     float dist = FAR_PLANE;
5562     result->did_hit = false;
5563     result->dist = FAR_PLANE;
```

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

```
    const __global material* material_buffer,
    const __global sphere* spheres,
    const __global plane* planes,
    const __global mesh* meshes,
    image1d_buffer_t indices,
    image1d_buffer_t vertices,
    image1d_buffer_t normals)
{
    scene s;
    s.material_buffer = material_buffer;
    s.spheres         = spheres;
    s.planes          = planes;
    s.meshes          = meshes;


    int id = get_global_id(0);
    int x  = id%width;
    int y  = id/width;
    int offset = x+y*width;
    int ray_offset = offset*3;

    ray r;
    r.orig = pos.xyz; //NOTE: slow unaligned memory access.
    r.dir.x = ray_buffer[ray_offset];
    r.dir.y = ray_buffer[ray_offset+1];
    r.dir.z = ray_buffer[ray_offset+2];

    collision_result result;
    if(!collide_all(r, &result, s, MESH_SCENE_DATA))
    {
        write_imagef(pos_tex, (int2)(x,y), (vec4)(0));
        write_imagef(nrm_tex, (int2)(x,y), (vec4)(0));
        return;
    }

    write_imagef(pos_tex, (int2)(x,y), (vec4)(result.point,0)); //Maybe ???
    write_imagef(nrm_tex, (int2)(x,y), (vec4)(result.normal,0));

    /* pos_tex[offset] = (vec4)(result.point,0); */
    /* nrm_tex[offset] = (vec4)(result.normal,0); */
}



__kernel void generate_discontinuity(
    image2d_t pos_tex,
    image2d_t nrm_tex,
    __global float* out_tex,
    const float k,
    const float intensity,
    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;

    //NOTE: this is fine for edges because the sampler is clamped

    //Positions
    vec4 pm = read_imagef(pos_tex, sampler, (int2)(x,y));
    vec4 pu = read_imagef(pos_tex, sampler, (int2)(x,y+1));
    vec4 pd = read_imagef(pos_tex, sampler, (int2)(x,y-1));
    vec4 pr = read_imagef(pos_tex, sampler, (int2)(x+1,y));
    vec4 pl = read_imagef(pos_tex, sampler, (int2)(x-1,y));

    //NOTE: slow doing this many distance calculations
    float posDiff = max(distance(pu,pm),
                        max(distance(pd,pm),
                            max(distance(pr,pm),
                                distance(pl,pm))));
    posDiff = clamp(posDiff, 0.f, 1.f);
    posDiff *= intensity;

    //Normals
    vec4 nm = read_imagef(nrm_tex, sampler, (int2)(x,y));

    vec4 nu = read_imagef(nrm_tex, sampler, (int2)(x,y+1));
    vec4 nd = read_imagef(nrm_tex, sampler, (int2)(x,y-1));
    vec4 nr = read_imagef(nrm_tex, sampler, (int2)(x+1,y));
    vec4 nl = read_imagef(nrm_tex, sampler, (int2)(x-1,y));
    //NOTE: slow doing this many distance calculations
    float nrmDiff = max(distance(nu,nm),
                        max(distance(nd,nm),
                            max(distance(nr,nm),
                                distance(nl,nm))));
    nrmDiff = clamp(nrmDiff, 0.f, 1.f);
    nrmDiff *= intensity;

    out_tex[offset] = k*nrmDiff+posDiff;
}

__kernel void float_average(
    __global float* in_tex,
    __global float* out_tex,
    const unsigned int width,
    const unsigned int height,
    const int total)
{
    int id = get_global_id(0);
    int x  = id%width;
    int y  = id/width;
    int offset = x+y*width;

```

```
    out_tex[offset] += in_tex[offset]/(float)total;

}


__kernel void mip_single_upsample( //nearest neighbour upsample.
    __global float* in_tex,
    __global float* out_tex,
    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 offset = x+y*width;

    out_tex[offset] = in_tex[(x+y*width)/2]; //truncated
}

__kernel void mip_upsample( //nearest neighbour upsample.
    image2d_t in_tex,
    __write_only image2d_t out_tex, //NOTE: not having __write_only caused it to crash without err
    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;

typedef struct
{
    uchar type;

    uint num_triangles;
    ulong triangle_start;
} kd_tree_leaf;

typedef struct
{
    uchar type;
    uchar k;
    float b;

    ulong left_index;
    ulong right_index;
} __attribute__ ((aligned (16))) kd_tree_node;


typedef union a_vec3
{
    vec3 v;
    float a[4];
} a_vec3;

typedef struct kd_stack_elem
{
    ulong node;
    float min;
    float max;
} kd_stack_elem;

typedef __global uint4* kd_44_matrix;


void kd_update_state(__global char* kd_tree, ulong indx, uchar* type,
                     kd_tree_node* node, kd_tree_leaf* leaf)
{


    *type = *((__global uchar*)(kd_tree+indx));

    if(*type == KDTREE_LEAF)
    {
        kd_tree_leaf_template template = *((__global kd_tree_leaf_template*) (kd_tree + indx));
        leaf->type = template.type;
        leaf->num_triangles = template.num_triangles;

        leaf->triangle_start = indx + sizeof(kd_tree_leaf_template);
    }
    else
        *node = *((__global kd_tree_node*) (kd_tree + indx));

}

void dbg_print_node(kd_tree_node n)
{
    printf("\nNODE: type: %u, k: %u, b: %f, l: %llu, r: %llu \n",
           (unsigned int) n.type, (unsigned int) n.k, n.b,
           n.left_index, n.right_index);
}

void dbg_print_matrix(kd_44_matrix m)
{
    printf("[%2u %2u %2u %2u]\n" \
           "[%2u %2u %2u %2u]\n" \
           "[%2u %2u %2u %2u]\n" \
           "[%2u %2u %2u %2u]\n\n",
           m[0].x, m[0].y, m[0].z, m[0].w,
           m[1].x, m[1].y, m[1].z, m[1].w,
           m[2].x, m[2].y, m[2].z, m[2].w,
           m[3].x, m[3].y, m[3].z, m[3].w);
}

inline float get_elem(vec3 v, uchar k, kd_44_matrix mask)
{
    k = min(k,(uchar)2);
    vec3 nv = select((vec3)(0), v, mask[k].xyz);//NOTE: it has to be MSB on the mask

    return nv.x + nv.y + nv.z;
}

//#define B 3*32 //batch size
#define STACK_SIZE 16 //tune later
#define LOAD_BALANCER_BATCH_SIZE        32


__kernel void kdtree_intersection(
    __global kd_tree_collision_result* out_buf,
    __global ray* ray_buffer, //TODO: make vec4

    __global uint* dumb_data, //NOTE: REALLY DUMB, you can't JUST have a global variable in ocl.

//Mesh
    __global mesh* meshes,
    image1d_buffer_t     indices,
    image1d_buffer_t     vertices,
    __global char* kd_tree,   //TODO: use a higher allignment type

```

```
    unsigned int num_rays)
{

    const uint blocksize_x = BLOCKSIZE_X; //should be 32 //NOTE: REMOVED A THING
    const uint blocksize_y = BLOCKSIZE_Y;

    uint x = get_local_id(0) % BLOCKSIZE_X; //id within the warp
    uint y = get_local_id(0) / BLOCKSIZE_X; //id of the warp in the SM

    __local volatile int next_ray_array[BLOCKSIZE_Y];
    __local volatile int ray_count_array[BLOCKSIZE_Y];
    next_ray_array[y]  = 0;
    ray_count_array[y] = 0;

    kd_stack_elem stack[STACK_SIZE];
    uint stack_length = 0;

    //NOTE: IT WAS CRASHING WHEN THE VECTORS WERENT ALLIGNED!!!!
    kd_44_matrix elem_mask = (kd_44_matrix)(dumb_data);
    __global uint* warp_counter = dumb_data+16;


    //NOTE: this block of variables is probably pretty bad for the cache
    ray           r;
    float         t_hit = INFINITY;
    vec2          hit_info = (vec2)(0,0);
    unsigned int  tri_indx;
    float         t_min, t_max;
    float         scene_t_min = 0, scene_t_max = INFINITY;
    kd_tree_node  node;
    kd_tree_leaf  leaf;
    uchar         current_type = KDTREE_NODE;
    bool          pushdown = false;
    kd_tree_node  root;
    uint          ray_indx;

    while(true)
    {
        uint tidx = x; // SINGLE WARPS WORTH OF WORK 0-32
        uint widx = y; // WARPS PER SM 0-4 (for example)
        __local volatile int* local_pool_ray_count = ray_count_array+widx;
        __local volatile int* local_pool_next_ray  = next_ray_array+widx;

        //Grab new rays
        if(tidx == 0 && *local_pool_ray_count <= 0) //only the first work item gets memory
        {
            *local_pool_next_ray = atomic_add(warp_counter, LOAD_BALANCER_BATCH_SIZE); //batch complete


            *local_pool_ray_count = LOAD_BALANCER_BATCH_SIZE;
        }
//lol help there are no barriers

        {

            ray_indx = *local_pool_next_ray + tidx;

            if(ray_indx >= num_rays) //ray index is past num rays, work is done
                break;

            if(tidx == 0)
            {
                *local_pool_next_ray += 32;
                *local_pool_ray_count -= 32;
            }

            r = ray_buffer[ray_indx];

            t_hit = INFINITY; //infinity

            if(!getTBoundingBox((vec3) SCENE_MIN, (vec3) SCENE_MAX, r, &scene_t_min, &scene_t_max)) //SCENE_MIN is a macro
            {
                scene_t_max = -INFINITY;

            }


            t_max = t_min = scene_t_min;

            stack_length = 0;
            root = *((__global kd_tree_node*) kd_tree);
        }
        stack_length = 0;

        while(t_max < scene_t_max)
        {

            if(stack_length == (uint) 0)
            {
                node  = root; //root
                current_type = KDTREE_NODE;
                t_min = t_max;
                t_max = scene_t_max;
                pushdown = true;
            }
            else
            { //pop

                t_min = stack[stack_length-1].min;
                t_max = stack[stack_length-1].max;
                kd_update_state(kd_tree, stack[stack_length-1].node, &current_type, &node, &leaf);

                stack_length--;
                pushdown = false;
            }
```

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

```
}

__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) );
}


__kernel void kdtree_test_draw(
    __global unsigned int* out_tex,
    __global kd_tree_collision_result* kd_results,

    const __global material* material_buffer,
    //meshes
    __global mesh* meshes,

    image1d_buffer_t     indices,
    image1d_buffer_t     vertices,
    image1d_buffer_t     normals,
    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;

    kd_tree_collision_result res = kd_results[offset];
    if(res.t==0)
    {
        out_tex[offset] = get_colour( (vec4) (0) );
        return;
    }
    int4 i1 = read_imagei(indices, res.triangle_index);
    int4 i2 = read_imagei(indices, res.triangle_index+1);
    int4 i3 = read_imagei(indices, res.triangle_index+2);
    mesh m = meshes[i1.w];
    material mat = material_buffer[m.material_index];

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

    normal = (normal+1) / 2;

    out_tex[offset] = get_colour( (vec4) (normal,1) );
}

//TODO: ADD A THING FOR THIS
//#pragma OPENCL EXTENSION cl_nv_pragma_unroll : enable

vec3 uniformSampleHemisphere(const float r1, const float r2)
{
    float sinTheta = sqrt(1 - r1 * r1);
    float phi = 2 * M_PI_F * r2;
    float x = sinTheta * cos(phi);
    float z = sinTheta * sin(phi);
    return (vec3)(x, r1, z);
}
vec3 cosineSampleHemisphere(float u1, float u2, vec3 normal)
{
    const float r = sqrt(u1);
    const float theta = 2 * M_PI_F * u2;

    vec3 w = normal;
    vec3 axis = fabs(w.x) > 0.1f ? (vec3)(0.0f, 1.0f, 0.0f) : (vec3)(1.0f, 0.0f, 0.0f);
    vec3 u = normalize(cross(axis, w));
    vec3 v = cross(w, u);

    /* use the coordinte frame and random numbers to compute the next ray direction */
    return normalize(u * cos(theta)*r + v*sin(theta)*r + w*sqrt(1.0f - u1));
}

#define NUM_BOUNCES 4
#define NUM_SAMPLES 4

typedef struct spath_progress
{
    unsigned int sample_num;
    unsigned int bounce_num;
    vec3 mask;
    vec3 accum_color;
} __attribute__((aligned (16))) spath_progress; //NOTE: space for two more 32 bit dudes

__kernel void segmented_path_trace_init(
    __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 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, res.triangle_index);
    int4 i2 = read_imagei(indices, res.triangle_index+1);
    int4 i3 = read_imagei(indices, 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, i1.y).xyz*(1-res.u-res.v)+
        read_imagef(normals, i2.y).xyz*res.u+
        read_imagef(normals, 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,
    image1d_buffer_t normals,
    /* const __global vec2* texcoords, */
    const unsigned int width,
    //const unsigned int rwidth,
    //const unsigned int soffset,
    const unsigned int random_value)
{
    const vec4 sky = (vec4) (0.16, 0.2, 0.2, 0);
    // int x = (soffset*width)+get_global_id(0)%width;
    int x = get_global_id(0)%width;
    int y = get_global_id(0)/width;
    int offset = (x+y*width);

    spath_progress spd = spath_data[offset];
```

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

```
6509         spd.bounce_num = 0;
6510         spd.sample_num++;
6511         out_tex[offset] += (vec4) (clamp(spd.accum_color, 0.f, 1.f),1);
6512 
6513         //START OF NEW
6514 
6515 
6516         res = kd_source_results[offset];
6517         r = ray_origin_buffer[offset];
6518         spd.mask = (vec3)(1.0f, 1.0f, 1.0f);
6519         spd.accum_color = (vec3) (0, 0, 0);
6520 
6521 
6522         if(res.t==0)
6523         {
6524             out_tex[offset] += sky;
6525             //printf("SHI\n");
6526             //return;
6527         }
6528 
6529         i1 = read_imagei(indices, res.triangle_index);
6530         i2 = read_imagei(indices, res.triangle_index+1);
6531         i3 = read_imagei(indices, res.triangle_index+2);
6532         m = meshes[i1.w];
6533         mat = material_buffer[m.material_index];
6534         pos = r.orig + r.dir*res.t;
6535         //pos = (vec3) (0, 0, -2);
6536 
6537         normal =
6538             read_imagef(normals, i1.y).xyz*(1-res.u-res.v)+
6539             read_imagef(normals, i2.y).xyz*res.u+
6540             read_imagef(normals, i3.y).xyz*res.v;
6541 
6542         spd.mask *= mat.colour;
6543         if( (float)(mat.reflectivity==1.)) //TODO: just add an emmision value in material
6544         {
6545             spd.accum_color += spd.mask*2;
6546         }
6547 
6548         sample_dir = cosineSampleHemisphere(rand1, rand2, normal);
6549         sr.orig = pos + normal * 0.0001f; //sweet spot for epsilon
6550         sr.dir = sample_dir;
6551         //printf("GOOD %f %f %f\n",spd.accum_color.x, spd.accum_color.y, spd.accum_color.z);
6552     }
6553 
6554     ray_buffer[offset] = sr;
6555 
6556     spath_data[offset] = spd;
6557 
6558 }
6559 
6560 __kernel void path_trace(
6561     __global vec4* out_tex,
6562     const __global ray* ray_buffer,
6563     const __global material* material_buffer,
6564     const __global sphere* spheres,
6565     const __global plane* planes,
6566 //Mesh
6567     const __global mesh* meshes,
6568     image1d_buffer_t indices,
6569     image1d_buffer_t vertices,
6570     image1d_buffer_t normals,
6571     /* const __global vec2* texcoords, */
6572     const unsigned int width,
6573     const vec4 pos,
6574     unsigned int magic)
6575 {
6576     scene s;
6577     s.material_buffer = material_buffer;
6578     s.spheres         = spheres;
6579     s.planes          = planes;
6580     s.meshes          = meshes;
6581 
6582 
6583     const vec4 sky = (vec4) (0.16, 0.2, 0.2, 0);
6584     //return;
6585     int x = get_global_id(0);
6586     int y = get_global_id(1);
6587     //int x  = id%width+ get_global_offset(0)%total_width;
6588     //int y  = id/width/* + get_global_offset(0)/total_width*/;
6589     int offset = (x+y*width);
6590     //int ray_offset = offset; //NOTE: unnecessary w/ new rays
6591 
6592     ray r;
6593     r = ray_buffer[offset];
6594     r.orig = pos.xyz;
6595     union {
6596         float f;
6597         unsigned int ui;
6598     } res;
6599 
6600     res.f = (float)magic*M_PI_F+x;//fill up the mantissa.
6601     unsigned int seed1 = res.ui + (int)(sin((float)x)*7.1f);
6602 
6603     res.f = (float)magic*M_PI_F+y;
6604     unsigned int seed2 = y + (int)(sin((float)res.ui)*7*3.f);
6605 
6606     collision_result initial_result;
6607     if(!collide_all(r, &initial_result, s, MESH_SCENE_DATA))
6608     {
6609         out_tex[x+y*width] = sky;
6610         return;
6611     }
6612     barrier(0); //good ?
6613 
```

```
    vec3 fin_colour = (vec3)(0.0f, 0.0f, 0.0f);
    for(int i = 0; i < NUM_SAMPLES; i++)
    {

        vec3 accum_color = (vec3)(0.0f, 0.0f, 0.0f);
        vec3 mask        = (vec3)(1.0f, 1.0f, 1.0f);
        ray sr;
        float rand1 = get_random(&seed1, &seed2);
        float rand2 = get_random(&seed1, &seed2);


        vec3 sample_dir =  cosineSampleHemisphere(rand1, rand2, initial_result.normal);
        sr.orig = initial_result.point + initial_result.normal * 0.0001f; //sweet spot for epsilon
        sr.dir = sample_dir;
        mask *= initial_result.mat.colour;
        for(int bounces = 0; bounces < NUM_BOUNCES; bounces++)
        {
            collision_result result;
            if(!collide_all(sr, &result, s, MESH_SCENE_DATA))
            {
                accum_color += mask * sky.xyz;
                break;
            }


            rand1 = get_random(&seed1, &seed2);
            rand2 = get_random(&seed1, &seed2);

            sample_dir =  cosineSampleHemisphere(rand1, rand2, result.normal);

            sr.orig = result.point + result.normal * 0.0001f; //sweet spot for epsilon
            sr.dir = sample_dir;

            //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],
    ///((float)sample)/(float)num_samples);
}

__kernel void xorshift_batch(__global unsigned int* data)
{ //get_global_id is just a register, not a function
    uint d = data[get_global_id(0)];
    data[get_global_id(0)] = ((d << 1) | (d >> (sizeof(int)*8 - 1)))+1;//circular shift +1
}

__kernel void f_buffer_to_byte_buffer_avg(
    __global unsigned int* out_tex,
    __global vec4* fresh_frame_tex,
    __global spath_progress* spath_data,
    const unsigned int width,
    const unsigned int sample_num)
{
    int id = get_global_id(0);
    int x  = id%width;
    int y  = id/width;
    int offset = (x + y * width);
    //int roffset = (x + y * real);

    vec4 data   = fresh_frame_tex[offset];
    vec4 colour = data.w==0 ? (vec4)(0,0,0,0) : data.xyzw/data.w;

    /* if(get_global_id(0)%(width*100) == 0) */
    /*     printf("%f %f %f %f %f \n", */
    /*            fresh_frame_tex[offset].x, */
    /*            fresh_frame_tex[offset].y, */
    /*            fresh_frame_tex[offset].z, */
    /*            fresh_frame_tex[offset].w, */
    /*            colour.w); */
    out_tex[offset] = get_colour(colour);///sample_num);
}


__kernel void f_buffer_to_byte_buffer(
    __global unsigned int* out_tex,
    __global vec4* fresh_frame_tex,
    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(fresh_frame_tex[offset]);
}

vec4 shade(collision_result result, scene s, MESH_SCENE_DATA_PARAM)
{
    const vec3 light_pos = (vec3)(1,2, 0);
    vec3 nspace_light_dir = normalize(light_pos-result.point);
    vec4 test_lighting = (vec4) (clamp((float)dot(result.normal, nspace_light_dir), 0.0f, 1.0f));
    ray r;
    r.dir  = nspace_light_dir;
    r.orig = result.point + nspace_light_dir*0.00001f;
    collision_result _cr;
    bool visible = !collide_all(r, &_cr, s, MESH_SCENE_DATA);
    test_lighting *= (vec4)(result.mat.colour, 1.0f);
    return visible*test_lighting/2;
}


__kernel void cast_ray_test(
    __global unsigned int* out_tex,
    const __global ray* ray_buffer,
    const __global material* material_buffer,
    const __global sphere* spheres,
    const __global plane* planes,
//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 height,
    const vec4 pos)
{
    scene s;
    s.material_buffer = material_buffer;
    s.spheres         = spheres;
    s.planes          = planes;
    s.meshes          = meshes;

```

```
    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;
    int ray_offset = offset;


    ray r;
    r = ray_buffer[ray_offset];
    r.orig = pos.xyz; //NOTE: unnecesesary rn, in progress of updating kernels w/ the new ray buffers.

    //r.dir  = (vec3)(0,0,-1);

    //out_tex[x+y*width] = get_colour_signed((vec4)(r.dir,0));
    //out_tex[x+y*width] = get_colour_signed((vec4)(1,1,0,0));
    collision_result result;
    if(!collide_all(r, &result, s, MESH_SCENE_DATA))
    {
        out_tex[x+y*width] = get_colour( sky );
        return;
    }
    vec4 colour = shade(result, s, MESH_SCENE_DATA);


    #define NUM_REFLECTIONS 2
    ray rays[NUM_REFLECTIONS];
    collision_result results[NUM_REFLECTIONS];
    vec4 colours[NUM_REFLECTIONS];
    int early_exit_num = NUM_REFLECTIONS;
    for(int i = 0; i < NUM_REFLECTIONS; i++)
    {
        if(i==0)
        {
            rays[i].orig = result.point + result.normal * 0.0001f; //NOTE: BIAS
            rays[i].dir  = reflect(r.dir, result.normal);
        }
        else
        {
            rays[i].orig = results[i-1].point + results[i-1].normal * 0.0001f; //NOTE: BIAS
            rays[i].dir  = reflect(rays[i-1].dir, results[i-1].normal);
        }
        if(collide_all(rays[i], results+i, s, MESH_SCENE_DATA))
        {
            colours[i] = shade(results[i], s, MESH_SCENE_DATA);
        }
        else
        {
            colours[i] = sky;
            early_exit_num = i;
            break;
        }
    }
    for(int i = early_exit_num-1; i > -1; i--)
    {
        if(i==NUM_REFLECTIONS-1)
            colours[i] = mix(colours[i], sky, results[i].mat.reflectivity);

        else
            colours[i] = mix(colours[i], colours[i+1], results[i].mat.reflectivity);

    }

    colour = mix(colour, colours[0],  result.mat.reflectivity);

    out_tex[offset] = get_colour( colour );
}


//NOTE: it might be faster to make the ray buffer a multiple of 4 just to align with words...
__kernel void generate_rays(
    __global ray* out_tex,
    const unsigned int width,
    const unsigned int height,
    const t_mat4 wcm)
{
    int id = get_global_id(0);
    int x  = id%width;
    int y  = id/width;
    int offset = (x + y * width);

    ray r;

    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)));
}
<!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>
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);
}#include <KdTreeComponent.h>
#include <ge/entity/component/batches/PipelineComponentBatch.h>
#include <ge/entity/component/ComponentManager.h>
#include <ge/graphics/GraphicsCore.h>
#include <ge/console/Log.h>
```

```
#include <ge/entity/Entity.h>
#include <ge/entity/EntityManager.h>
#include <ge/entity/component/components/TransformComponent.h>
#include <ge/entity/component/components/HLMeshComponent.h>
#include <ge/engine/scene/Scene.h>
#include <ge/debug/DebugBox.h>
#include <ge/input/KeyboardHandler.h>
#include <web_stuff.h>
#include <GL/glew.h>

ge::Component* _constructor_KdTreeComponent(ge::Entity* ent)
{
    return new KdTreeComponent(ent);
}

ge::ComponentConstructorRegistry::StartupHook KdTreeComponent::_hook("KdTreeComponent", _constructor_KdTreeComponent);


KdTreeComponent::KdTreeComponent(ge::Entity* e) : ge::Component(e)
{
    addPublicVar("Maximum Depth",  {ge::DataType::INT, &kd_max_depth});
    addPublicVar("Maximum Traversal Depth",  {ge::DataType::INT, &kd_traversal_max_depth});
    addPublicVar("Traverse",  {ge::DataType::BOOL, &kd_traverse});

    kd_max_depth = 5;
    kd_traversal_max_depth = 1;
    kd_traverse = false;
}

void KdTreeComponent::defaultInit()
{
    //TODO: probably should remove this.
}

void KdTreeComponent::insertToDefaultBatch() //TODO: add ingroup priorities
{

    if(!ge::ComponentManager::containsComponentBatch("PipelineComponentBatch", getTypeName()))
    {
        ge::PipelineComponentBatch* cmp = new ge::PipelineComponentBatch();
        cmp->setComponentType(getTypeName());

        ge::ComponentManager::registerComponentBatch(cmp);
    }

    ge::ComponentManager::getComponentBatch("PipelineComponentBatch", getTypeName())->softInsert(this);
}

void GenerateBox(glm::vec3 min, glm::vec3 max) //NOTE: not optomized
{
    ge::Entity* ent = new ge::Entity();
    ent->name = "_tree_node";
    ge::EntityManager::registerEntity(ent);

    ge::TransformComponent* transform = new ge::TransformComponent(ent);
    transform->insertToDefaultBatch();
    transform->dynamic = true; //whatever tbh
    transform->setPosition((min + max)/2);
    transform->setScale(max-min);
    ent->insertComponent(transform);

    ge::HLMeshComponent* renderer = new ge::HLMeshComponent(ent);
    renderer->insertToDefaultBatch();
    renderer->setMeshData("demo/meshes/cube.obj");
    renderer->setMaterial("CubeWire");
    renderer->mesh->getMesh()->cullBackface = false;
    ent->insertComponent(renderer);
}

void GenerateBoxMAIN(glm::vec3 min, glm::vec3 max) //NOTE: not optomized
{
    ge::Entity* ent = new ge::Entity();
    ent->name = "_tree_node";
    ge::EntityManager::registerEntity(ent);

    ge::TransformComponent* transform = new ge::TransformComponent(ent);
    transform->insertToDefaultBatch();
    transform->dynamic = true; //whatever tbh
    transform->setPosition((min + max)/2);
    transform->setScale(max-min);
    ent->insertComponent(transform);

    ge::HLMeshComponent* renderer = new ge::HLMeshComponent(ent);
    renderer->insertToDefaultBatch();
    renderer->setMeshData("demo/meshes/cube.obj");
    renderer->setMaterial("CubeWireRed");
    renderer->mesh->getMesh()->cullBackface = false;
    ent->insertComponent(renderer);
}

void GenerateBoxTRAV(glm::vec3 min, glm::vec3 max) //NOTE: not optomized
{
    ge::Entity* ent = new ge::Entity();
    ent->name = "_tree_node";
    ge::EntityManager::registerEntity(ent);

    ge::TransformComponent* transform = new ge::TransformComponent(ent);
    transform->insertToDefaultBatch();
    transform->dynamic = true; //whatever tbh
    transform->setPosition((min + max)/2);
    transform->setScale(max-min);
    ent->insertComponent(transform);

    ge::HLMeshComponent* renderer = new ge::HLMeshComponent(ent);
    renderer->insertToDefaultBatch();
```

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

```
void kd_update_state(size_t indx, unsigned char* type,
                     skd_tree_traversal_node* node, skd_tree_leaf_node* leaf)
{


    *type = kd_tree[indx];

    if(*type == KDTREE_LEAF)
    {
        *leaf = *((skd_tree_leaf_node*) (kd_tree + indx));

    }
    else
        *node = *((skd_tree_traversal_node*) (kd_tree + indx));

}

typedef struct kd_stack_elem
{
    size_t node;
    glm::vec3 Vmin;
    glm::vec3 Vmax;
    float min;
    float max;
} kd_stack_elem;

void KdTreeComponent::traverse(glm::vec3 origin, glm::vec3 direction, glm::vec3 min, glm::vec3 max)
{

    kd_stack_elem stack[STACK_SIZE];
    unsigned int stack_length = 0;


    float         t_hit = INFINITY;
    glm::vec2     hit_info = glm::vec2(0,0);
    unsigned int  tri_indx;
    float         t_min, t_max;
    float         scene_t_min = -10000, scene_t_max = 10000;
    skd_tree_traversal_node  node;
    skd_tree_leaf_node  leaf;
    unsigned char     current_type = KDTREE_NODE;
    bool          pushdown = false;
    skd_tree_traversal_node  root;
    unsigned int          ray_indx;

    glm::vec3 current_min = min;
    glm::vec3 current_max = max;

    root = *(skd_tree_traversal_node*)kd_tree;
    if(!getTBoundingBox(origin, direction, min, max,
                        &scene_t_min, &scene_t_max)) //SCENE_MIN is a macro
    {
        scene_t_max = -INFINITY;//t_max = INFINITY;
        printf("Shit\n");
    }
    t_max = t_min = scene_t_min;
    printf("scene smin:%f smax:%f\n", scene_t_min, scene_t_max);

    while(t_max < scene_t_max)
    {
        //printf("HIT BOUNDING BOX");
        if(stack_length == 0)
        {
            node  = root; //root
            current_type = KDTREE_NODE;
            t_min = t_max;
            t_max = scene_t_max;
            pushdown = true;
        }
        else
        { //pop

            //TODO: update this
            //WRONG: node  = stack[stack_length-1].node;
            //assert(stack_length == 1);
            //printf("K it went around once!\n");
            t_min = stack[stack_length-1].min;
            t_max = stack[stack_length-1].max;
            current_min = stack[stack_length-1].Vmin;
            current_max = stack[stack_length-1].Vmax;
            current_type = *(kd_tree+stack[stack_length-1].node);
            //printf("K it retrieved some data from stack!\n");
            stack_length--;
            pushdown = false;
            printf("RETRIEVED FROM STACK\n");
        }


        while(current_type != KDTREE_LEAF)
        {
            GenerateBoxMAIN(current_min*0.99f, current_max*0.99f);
            //printf(":) ok test meme\n");

            //NOTE: none of this branches
            unsigned char k = node.k;
            //assert(k<=2);
            //assert(node.type<=2);
            float t_split = (node.b - origin[k]) / direction[k];
            //if(get_local_id(0)==0)
            printf("\n%f\n", t_split);
            //NOTE: CRASH HEPPENSE SOMEWHERE HERE
            //NOTE: something weird happens to the indexs of the nodes
            size_t first  = origin[k] < node.b ? node.left_ind : node.right_ind; //wrong
            size_t second = origin[k] < node.b ? node.right_ind : node.left_ind; //wrong
            //printf(":) ok test meme %d %llu %llu %llu %llu\n", (int) node.type, first, second, node.left_index, node.right_index);
```

```
            //dbg_print_node(node);
            //assert(first<150000);
            //assert(second<150000);
            printf("Thing %d\n", (unsigned char)current_type);

            if( t_split >= t_max || t_split < 0)  //NOTE: branching necessary
            {
                if(first == node.left_ind)
                    current_max[k] = node.b;
                else
                    current_min[k] = node.b;

                kd_update_state(first, &current_type, &node, &leaf);
            }
            else if(t_split <= t_min)
            {
                if(second == node.left_ind)
                    current_max[k] = node.b;
                else
                    current_min[k] = node.b;
                kd_update_state(second, &current_type, &node, &leaf);
            }
            else
            {
                //assert(stack_length==(ulong)STACK_SIZE);
                //update
                glm::vec3 new_min = min, new_max = max;

                if(second == node.left_ind)
                    new_min[k] = node.b;
                else
                    new_max[k] = node.b;


                stack[stack_length++] = {second, new_min, new_max, t_split, t_max}; //push
                kd_update_state(first, &current_type, &node, &leaf);

                if(first == node.left_ind)
                    current_max[k] = node.b;
                else
                    current_min[k] = node.b;

                //printf("test meme 1 complete\n");
                printf("test meme 1 %llu\n", stack_length);

                t_max = t_split;
                pushdown = false;
            }
            printf("Thing2 %d\n", (unsigned char)current_type);

            if(pushdown)
            {
                root = node;//UPDATE
            }

        }
        //printf("HEY found something %d\n", leaf.num_triangles);
        //Found leaf
        if(leaf.num_triangles>0)
        {
            printf("ENTERED LEAF MAIN INTERSECTION test meme\n");

            return;
            /*vec3 tri[4];
            unsigned int index_offset = *(__global uint*)(kd_tree+t);
            //get vertex (first element of each index)
            int4 idx_0 = read_imagei(indices, index_offset+0);
            int4 idx_1 = read_imagei(indices, index_offset+1);
            int4 idx_2 = read_imagei(indices, index_offset+2);

            tri[0] = read_imagef(vertices, idx_0.x).xyz;
            tri[1] = read_imagef(vertices, idx_1.x).xyz;
            tri[2] = read_imagef(vertices, idx_2.x).xyz;

            vec3 hit_coords; // t u v
            if(does_collide_triangle(tri, &hit_coords, r))
            {
                if(hit_coords.x < t_hit)
                {
                    t_hit = hit_coords.x;     //t
                    hit_info = hit_coords.yz; //u v
                    tri_indx = index_offset;
                }
                printf("COLLISION\n");
                //t_hit = min(t_hit, hit_coords.x); //t  NOTE: this is faster but it saves computation is we store u and v aswell
                if(t_hit < t_min) // goes by closest to furthest, so if it hits it will be the closest
                {//early exit
                    break; //TODO: do something
                }
                }*/
        }
    }
    printf("THIS THING SUCKS.");
}

void KdTreeComponent::cycle()
{
    static bool last_pressed  = false;
    static bool last_pressedT = false;

    if(ge::GraphicsCore::ctx->currentPipeline->getState()!=ge::PipelineState::Render)
        return;
    if(ge::GraphicsCore::ctx->currentPipeline->getCurrentStage()->type!=ge::PipelineDrawType::Default)
        return;

```

```
    if(transformComponent == nullptr)
    {

        if(ent->components.count("TransformComponent"))
        {
            transformComponent = (ge::TransformComponent*) ent->components.at("TransformComponent");


        }
        else
        {
            ge::Log::wrn("KdTreeComponent", "A KdTreeComponent requires a TransformComponent.");
            return;
        }
    }
    if(ge::KeyboardHandler::keyDownSticky(ge::KeyboardKey::K) && !last_pressed)
    {
        retrieve_data = true;
        last_pressed = true;
    }
    else
        if(!ge::KeyboardHandler::keyDownSticky(ge::KeyboardKey::K))
            last_pressed = false;

    if(ge::KeyboardHandler::keyDownSticky(ge::KeyboardKey::T) && !last_pressedT)
    {
        kd_traverse = true;
        last_pressedT = true;
    }
    else
        if(!ge::KeyboardHandler::keyDownSticky(ge::KeyboardKey::T))
            last_pressedT = false;
    const glm::vec3 min(-30.976175, -13.904981, -31.015106);
    const glm::vec3 max(30.976175, 17.071194, 25.015106);

    if(new_data_retrieved)
    {
        new_data_retrieved = false;
        lkd_tree = kd_tree;
        lkd_tree_size = kd_tree_size;
        ge::Log::dbg("KdTreeComponent", "New Data has been retrieved.");

        //glm::vec3 min(-1.328727, -1.108050, -10.190243);
        //glm::vec3 max(1.281850, 1.446441, -6.997774);
        //glm::vec3 min(-2.230403, -1.011691, -3.411920);
        //glm::vec3 max(1.244233, 0.678684, -2.960731);

        traversekd_and_gen((unsigned char*)lkd_tree, (unsigned char*)lkd_tree, 0,
                           min, max);
        GenerateBoxMAIN(min,max);
        //THREE OF THEME ARE CORRECT.
    }

    if(kd_traverse)
    {
        kd_traverse = false;
        traverse({0,0,0}, {0,-0.25,-1}, min, max);
    }
    //const glm::vec3 min(-1, -1, -1);
    //const glm::vec3 max(-1, -1, -1);

    //ge::Debug::DebugBox::draw(glm::vec3(1,0.8,0.8), min+max/2, glm::vec3(0.2f));


}




void KdTreeComponent::destroy()
{

}

std::string KdTreeComponent::getTypeName()
{
    return "KdTreeComponent";
}
```