CS350 Project 4

Synopsis

Build a Kd-tree from the list of triangles, using the algorithm $O(N \log^2 N)$ algorithm from the paper: https://faculty.digipen.edu/~gherron/content/kdtree-2.pdf , and demonstrate its use with the ray-tracing portion of the framework.

Instructions:

Kd-tree tree construction:

From the list of triangle, create a Kd-tree using the top-down Surface-Area-Heuristic Sweep $O(N \log^2 N)$ algorithm. Ignore the $O(N^2)$ algorithm as too slow, and the $O(N \log N)$ as it offers too little gain for the effort.

The details of the SAH-Sweep algorithm can be found in a separate document in the online lecture notes. In brief, to split a node V with triangle list T, a large batch of potential split points p_k are produced on each axis. For each p_k, a cost for that split $C_V(p_k)$, and an overall cost for not splitting $C_V(\emptyset)$ are all calculated. The minimum of all costs is used to provided both the **termination criteria** and the **split plane decision**.

Kd-tree tree traversal:

The code will be run for grading twice, once with **MAIN** in **student_code.h** defined with **#define MAIN SCAN_LINE** and again with **#define MAIN RAY_TRACE.**

Here are three specific documentation requirements:

- For **SCAN_LINE** mode: Implement both **Collision Detection** and **Triangle Selection** operations as in project 3. Document the tree as you did in project 2, with both:
  - a report containing
    number of triangles
    number of leaf-nodes
    min and max depths.
  - a way to display all the bounding boxes in the tree. **(See the note below.)**

- For **RAY_TRACE** mode**:** This provides the "stress test" for this project. Implement a very basic ray casting operation. (Ray casting is tracing **one ray per pixel**, and coloring that pixel with a simple lighting calculation.) For this mode, include in the report both
  - the frame rate and number of pixels (100x100 is the built-in default) when running with no Kd-tree (that is, just looping through the linear list of triangles)
  - the frame rate and number of pixels when running with the Kd-tree.
  - **Expect** a huge speedup between the with- and without- Kd-tree runs, but
  - **Do not expect** normal real-time refresh rates.
  - **See a section below with more ray-casting details.**

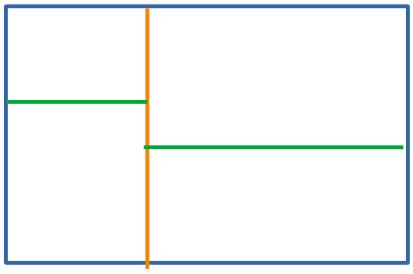
- Continued on next page …

- For full credit, you **must implement the incremental-sweep algorithm** from the fast-KD-tree paper. As such, your report must contain pointers to the code where you've implemented each of the steps of the sweep algorithm. Use this notation in your report:

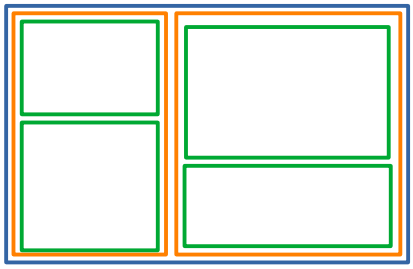
Gather:	**line-number-begin – line-number-end**
Sort:	**line-number-begin – line-number-end**
Group:	**line-number-begin – line-number-end**
Sweep:	**line-number-begin – line-number-end**
terminate decision:	**line-number-begin – line-number-end**
split-plane decision:	**line-number-begin – line-number-end**

Special note about displaying bounding boxes:

Displaying bounding boxes of Kd-trees presents a problem that did not come up in the display of BVH boxes in the previous project. In Kd-trees, child nodes share all but one boundary with their parent nodes, so displaying the two children of a node will completely and exactly overwrite any drawing of the parent node (or vise-versa depending on the draw order). To avoid this, **draw each bounding box very slightly smaller than its parents**, by say 1% per level. That is drawing with $h_{draw} = pow(0.99, depth) * h$ will poduce results like this:



Not this: Drawing boxes as C,h bottom-up. Top-down is worse.



*This!!: Drawing boxes as C, pow(0.99,depth)*h*

Special note on ray-casting and lighting:

Some of this code is already in the framework, but commented out. This section clarifies that code and contains far more details about computing the lighting at a pixel.

```
void StudentCode::GenerateRayTracedImage(int width, int height, float* image,
                                                    vec3& lightPos)
{
    int N = 100;  // Number of pixels to ray-trace
    vec3 eye = Hdiv(ViewInverse*vec4(0,0,0,1));

    for (int i=width/2-N;  i<=width/2+N; i++) {
        for (int j=height/2-N;  j<=height/2+N; j++) {
            // Pixel center converted from (I,j) index to (X,Y,Z) in NDC.
            float X = 2.0*(i+0.5)/width  - 1;
            float Y = 2.0*(j+0.5)/height – 1;
            float Z = -1;  // Any value would do here;
            vec3 T = Hdiv(ViewInverse*ProjInverse*vec4(X, Y, -1, 1));

            Ray ray(eye, T-eye);

            // Traverse Kd-tree finding the ray's front most intersection, returning:
            //        frontTriangle: pointer to the front most Triangle or NULL
            //        tMin: ray parameter value of rays intersection

            if (frontTriangle) {
                // The color of a triangle:
                //   Add an object pointer to each triangle.  The call to
                //   ObjectTriangles contains the object; you probably ignored it.
                vec3 Kd = frontTriangle->obj->diffuseColor;

                // The normal to a triangle: (not interpolated across triangle)
                vec3 P0 = frontTriangle->points[0];
                vec3 P1 = frontTriangle->points[1];
                vec3 P2 = frontTriangle->points[2];
                vec3 N = normalize(cross(P2-P0, P1-P0));

                // The direction from the intersection point toward the light:
                vec3 intersectionPoint = ray.eval(tMin); // implement this if necessary
                vec3 L = normalize(lightPos - intersectionPoint);

                // Implementing simple lighting calculation:
                //    ambient:  (0.5, 0.5, 0.5) * Kd
                //    diffuse:  (1, 1, 1) * dot(N,L) * Kd
                //    specular: none so no need to calculate direction toward eye
                // Use abs to light both sides of all polygons.
                float NL = std::fabs(dot(N,L));
                color = vec3(0.5+NL)* Kd; }
            else
                color = vec3(0.25);  // Dark gray background

            float* pixel = image + 3*(j*width + i);
            *pixel++ = color.x;
            *pixel++ = color.y;
            *pixel++ = color.z; } }
}
```

What to submit

Submit to Moodle a single ZIP file containing SOURCE CODE ONLY: (*.cpp and *.h) and the **report.pdf** if you decided to report your results that way. Include **ALL** source code. (Including only source files you have changed is sufficient, but it's easy to forget headers files and project 1's geomlib-advanced.cpp, so submitting **all** source files is less error prone.)

Do not include any of the following:

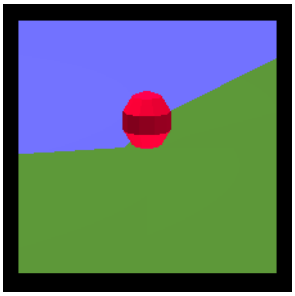
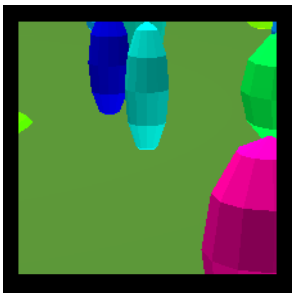
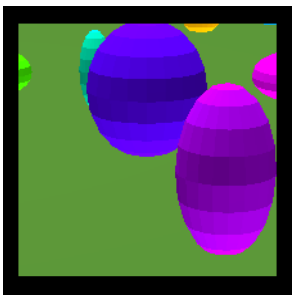
- Build artifacts (*.o, *.pdb, *.sdf, etc...) ,
- Executable files (*.exe),
- Debug, Release, or .vs folders.

Expected results

You may compare your results to the following images. There are several considerations:

- After starting the application, hit the TAB key to zoom the eye back away from the origin.
- These are screen captures of the center 200x200 pixels, ignoring the large black portions of non-ray-traced pixels.
- These screen captures are created with none of the extra-credit options implemented.
- Your frame-rates may be slower than my relatively high-end but 2 year old laptop. However, the difference between Kd-tree traversal and NO-Kd-tree traversal should be abundantly clear as you increase the triangle count.
- The frame rate, computed by GLFW and reported in the upper left corner or the window, is inaccurate when the actual frame rate is low, say below about 10 FPS. The slowest of the frame rates mentioned below are judged by rotating the image by mouse and counting approximately the frames-per-second (or seconds-per frame in the slowest case).
- Continued on next page ...

•

Image (After typing **TAB** key to zoom the eye further out from the origin.)	GOAL setting Triangle count Tree build time **The increase in build time is consistent with the paper's claim of** $O(N \log^2 N)$	Frame rate with NO Kd-tree traversal. Note the HUGE decrease in frame rate with increasing triangle count. This demonstrates $O(N)$ growth. **This demonstrates why we need a spatial data structure.**	Frame rate with Kd-tree traversal. Note the SLIGHT decrease in frame rate with increasing triangle count. This demonstrates $O(\log N)$ growth. **This is proof that the KD-tree achieves its goal.**
	GOAL=100 134 triangles Builds in <1 second	~4 FPS	7 FPS
	GOAL=1000 1244 triangles Builds in <1 second	~1/2 FPS Approximately 2 seconds for each frame	5 FPS
	GOAL=10*thousand 10930 triangles Builds <1 second	~1/15 FPS Approximately 15 seconds for each frame	3.5 FPS
	GOAL=100*thousand 104114 triangles Builds in ~3 seconds	Do not try this! Extrapolating from the runs above, this is expected to take **minutes** per frame. Don't do that to your CPU/GPU/OS.	3 FPS
	GOAL = 1*million Builds in ~40 seconds		

Extra Credit

Synopsis

In an attempt to put some actual computer graphics into this computer graphics course, here are three extra-credit add-ons for the ray tracing portion of project 4. Each one, if implemented correctly, will earn 5% extra credit. Each can be implemented in about 6 to 12 lines of code.

- Simple shadows: Calculate shadows cast by the singe point light source. (A full path tracer can handle any number of lights of any shape/size.)

- BRDF lighting: Replace the suggested overly-simple lighting calculation with a more accurate BRDF lighting calculation. (All modern uses of graphics implement BRDF lighting, rather than the beginner's level Phong lighting.)

- Normal interpolation: Smoothly interpolate vertex normals across a triangle. (This makes triangulated objects look smooth instead of faceted.)

Details for shadow calculation

The main algorithm consists of two operations: **ray-casting** and **lighting**. The **shadow calculation** is a new step inserted between those two operations.

- The **ray-casting step** determines what point is visible at a pixel by casting a view ray through that pixel and tracking the front most intersection of that ray with triangles in the scene. This step uses the Kd-tree traversal.

- The (new) **shadow calculation step** fires a ray from that intersection point toward the light, and notes if the ray hits anything **before** reaching the light. A Boolean **inShadow** is set accordingly. This step also uses the Kd-tree traversal. (Hint: Put the traversal in a separate procedure to be called in each of these steps.)

  Pseudo-code:

  P_I is the intersection point

  P_L is the light position

  Create ray with

  direction: $L = normalize(P_L - P_I)$, and

  origin: $O = P_I + \epsilon L$ (Use $\epsilon = 10^{-3}$ to avoid round-off at P_I.)

  t = intersection parameter for ray's intersection with front most triangle.

  If t is between 0 and $|D|$, then set Boolean **inShadow** TRUE, else FALSE.

- The **lighting step** now must choose between two lighting scenarios. (Note: The lighting in the main project uses **ambient** and **diffuse** but no **specular**.)

```
if (inShadow)
    color = <ambient only>  // No direct light for shadowed points
else
    color = <ambient + diffuse + specular> // Full lighting
```

Details for normal interpolation

The goal is to produce a normal at an intersection point which is a smooth blend of the normals at the three vertices. We use "barycentric" coordinates – three coordinates that describe a point's position in a triangle. These coordinates can be calculated as the ratio of areas, and areas can be calculated with a 3x3 determinant.

Notation and calculation:

- Let the triangle vertices be P_1, P_2, and P_3, and the intersection point P.
- Let the corresponding normals be N_1, N_2, and N_3. *(Note: The Triangle class (and its constructor) in the geometry library will have to be enhanced with three normals in addition to the three vertices.)*
- Then the barycentric coordinates are these ratios:

$$d = det3(P_1, P_2, P_3)$$
$$b_1 = det3(P, P_2, P_3)/d$$
$$b_2 = det3(P_1, P, P_3)/d$$
$$b_3 = det3(P_1, P_2, P)/d$$

where the 3x3 determinants can be calculated with

```
float det3(vec3 a, vec3 b, vec3 c) {
    return   a.x * (b.y*c.z-b.z*c.y)
           + a.y * (b.z*c.x-b.x*c.z)
           + a.z * (b.x*c.y-b.y*c.x); }
```

- The intersection point's normal, is interpolated from the three vertex normals:

$$N = b_1 N_1 + b_2 N_2 + b_3 N_3$$

Details for the BRDF lighting calculation

(Many more details can be found in a separate document.)

The list of input values for a lighting calculation:

(See next page for how to access/compute these values.)

Surface parameters:

- vec3 K_d : diffuse color
- vec3 K_s : specular color
- float α : rough/shiny parameter (1-5: rough, 10-20 mid, 100-200 smooth/shiny)

Geometry parameters:

- vec3 N : unit length surface normal
- vec3 L : unit vector toward light
- vec3 V : unit vector toward eye

Light parameters:

- vec3 I_i : brightness of light; Try: approximately vec3(3,3,3).
- vec3 I_a : brightness of ambient light; Try approximately vec3(0.5, 0.5, 0.5).

The general BRDF lighting calculation (including ambient term) is

$$I = I_a\, K_d \;+\; I_i\, (N \cdot L)_+\, BRDF$$

where the BRDF portion is:

$$BRDF = \frac{K_d}{\pi} \;+\; \frac{D(H)\;\; F(L,\, H)\;\; G(L,\, V,\, H)}{4\,(L \cdot N)_+\,(V \cdot N)_+} \quad \text{(but see below)}$$

and where

- $H \;=\; normalize(L+V)$; the half-vector, halfway between L and V
- $D(H) \;=\; \frac{\alpha+2}{2\pi}(H \cdot N)_+^{\alpha}$; The **corrected** Phong specular term
- $F(L,H) = K_s + (1-K_s)(1-(L \cdot H)_+)^5$; Schlick's approximation of the Fresnel term
- $G(L,\, V,\, H) \;=\; \frac{(L \cdot N)_+\,(V \cdot N)_+}{(L \cdot H)_+^2}$; An approximation to the shadow/occlusion term

The + subscript notation on dot products means clamp to non-negative values.
Programmatically this is

$(A \cdot B)_+ \;=\; max(A \cdot B, 0)$.

Use this for ALL dot products: $(H \cdot N)_+$, $(L \cdot N)_+$, $(V \cdot N)_+$, and $(L \cdot H)_+$

The (often used) approximation to the $G(...)$ term results in some cancellation, so the BRDF term becomes

$$BRDF = \frac{K_d}{\pi} \;+\; \frac{D(H)\;\; F(L,\, H)\;\; G(L,\, V,\, H)}{4\,(L \cdot H)_+^2}$$