Introduction to Computer Graphics
Assignment 3 – Ray Tracing Meshes

Handout date: 13.03.2018

Submission deadline: 22.03.2018, 13:00 h

Late submissions are not accepted

Figure 1: Expected result for ray tracing toon face triangle meshes.

In this assignment, you will implement ray tracing for triangle meshes. The framework code for this assignment extends the one from last week; if you download a fresh copy from Moodle you will need to copy your solutions from the Lighting assignment into the file `Scene.cpp` and also move over your ray-cylinder and ray-plane intersections from Assignment 1. Furthermore,"`todo`" comments have been inserted in `Mesh.cpp` to indicate where you need to add your implementations and a new `debug_aabb.cpp` file has been included. If you already set up a GitHub repository to collaborate with your fellow group members, you can just copy the TODO comments from `Mesh.cpp` over to your repository (or just note where your implementation needs to go and get started). You also need to add the `debug_aabb.cpp` file in the `src/` folder and copy the new versions of `src/CMakeLists.txt` and `src/Scene.h` files.

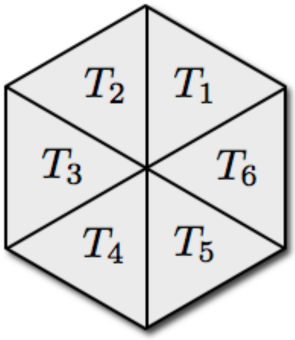
In the `expected_results` directory, we provide the images you should expect your finished code to produce for a subset of the provided scenes. One such result is shown in Figure 1.

Vertex Normals

The starting point of this assignment is the `compute_normals()` function in the file `Mesh.cpp`. Here, you will have to compute the vertex normals weighted by the opening angles. Before you start implementing this function, thoroughly study the file `Mesh.h`, making note of the member variables and methods and understanding the purpose of each one. Also check how functions `Mesh::read()`, `angleWeights()`, and function `Mesh::intersect()` are implemented in `Mesh.cpp`.

Compute each vertex normal $\mathbf{n}(V_{i_k})$ as an average of incident triangles' normals $\mathbf{n}(T_i)$, weighted by the opening angle $w_k(T_i)$.

$$\mathbf{n}(V_{i_k}) = \frac{\sum_{T_i \ni V_{i_k}} w_k(T_i)\,\mathbf{n}(T_i)}{\left\| \sum_{T_i \ni V_{i_k}} w_k(T_i)\,\mathbf{n}(T_i) \right\|}$$



To make your implementation more efficient, instead of traversing through all the neighboring triangles of each vertex, you should loop through all the triangles of the mesh and visit each triangle T_i only once. For each vertex V_{i_k} of the triangle T_i, add the contribution $w_k(T_i)\mathbf{n}(T_i)$ to the normal of the vertex V_{i_k} (see the exercise slides). Use the function `angleWeights()` to get the weights w_k for each triangle. After all triangles are visited, you should normalize all computed vertex normals.

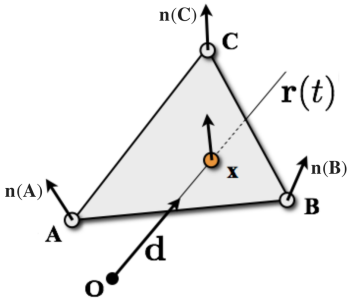
Ray-triangle Intersection

The second part of this assignment is to implement the ray-triangle intersection presented in the second lecture using the explicit barycentric coordinates representation (see also the exercise slides and formula below).

$$\mathbf{o} + t\,\mathbf{d} = \alpha\mathbf{A} + \beta\mathbf{B} + \gamma\mathbf{C}$$

where $\alpha + \beta + \gamma = 1$

and $\alpha, \beta, \gamma \geq 0$



Use Cramer's rule to solve the above system and follow the comments in `Mesh.cpp` to add the missing code in `intersect_triangle()`.

In certain scenes (e.g the office scene) some objects should be *flat shaded* (e.g. the desk) while other objects should be *Phong shaded* (e.g. chairs) to appear more realistic. This

difference lies in how the mesh's surface normal vector $\mathbf{n}(\mathbf{x})$ is computed for lighting the point $\mathbf{x}$. A member variable `draw_mode_` read from a scene file determines whether an mesh object should be *flat shaded* or *Phong shaded*. For *flat shaded* objects, the intersection normal is simply the normal of the intersected triangle. For *Phong shaded* objects you should use the normal interpolation formula below to compute the intersection normal $\mathbf{n}(\mathbf{x})$ at point $\mathbf{x}$ (interpolating the intersected triangle's three vertex normals).

$$\mathbf{n}(\mathbf{x}) = \alpha\mathbf{n}(\mathbf{A}) + \beta\mathbf{n}(\mathbf{B}) + \gamma\mathbf{n}(\mathbf{C})$$

Here α, β and γ are the barycentric coordinates of $\mathbf{x}$ computed with Cramer's rule, and $\mathbf{n}(\mathbf{A})$, $\mathbf{n}(\mathbf{B})$ and $\mathbf{n}(\mathbf{C})$ are vertex normals of vertices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ respectively. Be sure to normalize your normal vectors after interpolating them.

Efficient Ray-Mesh Intersections

To accelerate rendering, you will need to implement the *bounding box test* for triangle meshes. Fill in the missing code in `Mesh::intersect_bounding_box()`. This function should check whether a ray intersects with the current mesh's *bounding box*. Each mesh's *axis-aligned bounding box* has already been computed by `Mesh::compute_bounding_box()` (you do not need to edit this) and corner points stored in class members `Mesh::bb_min_` and `Mesh::bb_max_`. You will use these two bounding box corners in your intersection implementation.

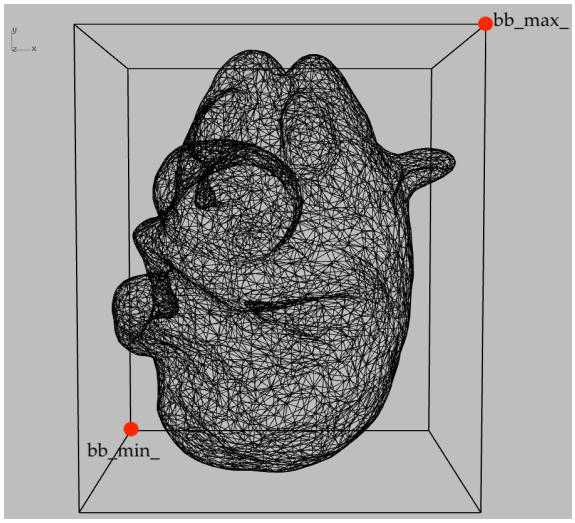


Figure 2: Axis-aligned bounding box example.

To check if your ray-box intersection implementation is correct, we provide another executable, `debug_aabb`, which visualizes the bounding box intersections for all meshes in a scene. Ray-box intersection points are visualized in red, and the brightness indicates how many bounding boxes intersect with the ray. Run `./debug_aabb 0` and compare your output images with the ones provided in `expected_results`. To use this executable in your existing codebase, you'll need to copy the new versions of `src/Scene.h`, `src/CMakeLists.txt`, and `src/debug_abb.cpp` from this assignment release.

Parallelization

(NOTE: this section is optional and it is NOT graded.) To make the computation even faster, you can use multiple cores of your machine to parallelize the rendering. You can do this with either OpenMP or TBB. On Linux/GCC, OpenMP should just work without any additional libraries. On macOS, only the latest versions of clang support OpenMP, so you may be better off installing TBB. TBB can be installed from you package manager (e.g., `apt` on Ubuntu, MacPorts or Homebrew on macOS), or directly from the official release page (https://github.com/01org/tbb/releases).

Whether you opt for OpenMP or TBB, you will need to use the new version of the `Scene::render()` method from `Scene.cpp` provided with this assignment. This version adds `#pragma omp parallel for` and `tbb::parallel_for(...)` to parallelize the ray tracing, depending on which library is available. Also note that you need to include TBB in the `Scene.cpp` (copy lines 24-27 from provided `Scene.cpp`).

Grading

Each part of this assignment is weighted as follows:

- Vertex normals: 20%
- Ray-triangle intersection: 50%
- Bounding box intersection: 30%

What to hand in

A .zip compressed file renamed to `Exercise`*n*`-Group`*i*`.zip` where *n* is the number of the current exercise sheet and *i* is the number of your group. It should contain:

- Hand in **only** the files you changed (in this case, `Mesh.cpp`) and the requested program output (in this case, `cube.png`, `rings.png`, `mask.png`, `office.png`, and `toon_faces.png`). It is up to you to make sure that all files that you have changed are in the zip.
- A `readme.txt` file containing a description on how you solved each exercise (use the same numbers and titles) and the encountered problems. Indicate what fraction of the total workload each project member contributed.
- Other files that are required by your `readme.txt` file. For example, if you mention some screenshot images in `readme.txt`, these images need to be submitted too.

Submit solutions to Moodle before the deadline. Late submissions receive 0 points!