# Worksheet 4

One of the key strengths of computer graphics techniques is that they work in general. Ray tracing is, for example, well known in numerous fields of research. The distinguishing feature of ray tracers in computer graphics is their ability to efficiently handle nearly arbitrary scenes. This set of exercises helps you load and efficiently render large triangle meshes.

## Learning Objectives

- Accelerate rendering techniques using spatial data structures.
- Implement ray tracing of a triangle mesh (indexed face set).
- Use a BSP tree for efficient space subdivision.
- Interpolate normals and texture coordinates across a triangle.

## Triangle Meshes

One way to handle arbitrary surfaces is using triangles. In particular, we often work with an indexed face set. This is a collection of vertices, normals, and texture coordinates, each with an index, and a list of index triples each of which defines a triangle and its vertex normals and texture coordinates (if present). One way to store an indexed face set on disk is using a Wavefront OBJ file.[1] This is the file format used by the framework. Most 3D modelling software packages can export to this format, and a couple of the classic computer graphics scenes have been included with the framework (the Stanford bunny, the Utah teapot, and the Cornell box, see the `models` subfolder).

- Load the Cornell box with blocks (`CornellBox.obj` and `CornellBlocks.obj`) and display it in preview. This is a simple triangle mesh often used for testing rendering algorithms. The framework loads one or more OBJ files if you provide them as command line arguments.[2] Take a screenshot of the (base colour) preview.
- Render the triangle mesh in base colours. Do this by implementing the function `intersect` in the file `TriMesh.cpp`. This function should perform ray-triangle intersection, where the triangle is a face in an indexed face set. A face consists of three indices that point out three vertices in an array which you can access using the function `geometry.vertex(i)`, where `i` is an index (`face.x`, for example). You can do the ray-triangle intersection using the same function (`intersect_triangle`) as what you used for rendering the triangle in the default scene. Compare the result to the preview.
- Render the triangle mesh using face normals (press '1' on the keyboard before rendering). This is referred to as flat shading. If your scene contains triangles with a material that has ambient colour, these triangles are extracted by the framework and stored as an area light. If not, a directional light is used as a default source. The Cornell box has an area light. To enable this source, implement the `sample` function in `AreaLight.cpp`. Approximate the area light with a point light by finding a representative position and intensity for it. Use the center of the axis aligned bounding box of the area light mesh as the position. Loop over the faces in the mesh to compute the intensity using face emission and face areas. The normalized sum of the vertex normals can be used as the face normal.
- Load the Utah teapot (`teapot.obj`), which is a larger triangle mesh with a smooth surface represented by vertex normals. The teapot will be lit by the default directional light. Enable this source by implementing the `sample` function in `Directional.cpp`. Render the teapot with the Lambertian

[1] http://paulbourke.net/dataformats/obj/

[2] In Visual Studio, command line arguments are provided in the project properties under Debugging.

shader (press '1'). The rendering time will be longer as the mesh is larger, and we have no space subdivision to accelerate the intersection tests. The next section is about space subdivision. For now, observe that the rendered result is not smooth.

- Render the teapot using interpolation of vertex normals across triangles (Phong shading). To do this, return to the `intersect` function in `TriMesh.cpp`. Instead of setting the shading normal equal to the geometric normal, use the barycentric coordinates to interpolate vertex normals across triangles. The rendered result should now be smooth.

## Space Subdivision

As observed in the section about triangle meshes, rendering is slow if we have no spatial data structure for acceleration of the intersection tests. The following exercises are about using an axis-aligned BSP tree for space subdivision.

- Use an axis-aligned BSP tree for finding the closest intersection of a ray with a triangle mesh. Investigate the implementation of this spatial data structure in `BspTree.cpp`. Explain how it works and modify the functions `closest_hit` and `any_hit` so that they use the BSP tree instead of the simple looping from Worksheet 1.
- Render the Utah teapot using the BSP tree. Check that the rendered result is the same as before using the BSP tree and note down the rendering times and the number of triangles in the mesh. Do this for the Cornell box (`CornellBox.obj` and `CornellBlocks.obj`) and Stanford bunny (`bunny.obj`) as well. Find the speed-up factors and describe the relation between number of triangles and speed-up factors for looping versus axis-aligned BSP tree.

## Worksheet 4 Deliverables

Renderings of the Cornell box, the Utah teapot, and the Stanford bunny. Also provide the explanations, comparisons, and performance measurements mentioned above. Include relevant code snippets. Please insert all this into your lab journal.

## Reading Material

The curriculum for Worksheet 4 is (43 pages)

**B** Chapter 12. *Data Structures for Graphics*.

**B** Section 10.2.2. *Surface Normal Vector Interpolation*.

Supplementary reading material:

- Kammaje, R. P., and Mora, B. A study of restricted BSP trees for ray tracing. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing (RT '07)*, pp. 55–62, October 2007.
- Sung, K., and Shirley, P. Ray tracing with the BSP tree. In D. Kirk, editor, *Graphics Gems III*, Chapter VI.1, pp. 271–274, Academic Press, 1995.