

CS535: Final Assignment

Out: October 20, 2005

Back/Due: December 6, 2005

Objective

Now that you have an understanding of basic interactive computer graphics, the focus of this final assignment is to give you the freedom to extend your knowledge in one particular area of interest to you. Below are four suggested individual projects in the areas of simplification, ray-tracing, occlusion culling, and collision detection.

Alternatively, you may also suggest your own project (which if you are doing, you should have communicated to be by now). All projects will be demonstrated to a public audience on the due date. The demo includes a live presentation of your project/program and a short PowerPoint presentation explaining how it works. The time slot is to be determined but roughly 10 to 20 minutes per person.

Please choose one of the projects below and send me (and the TA) an email before next class time (October 25, 4:30pm) indicating which project you will be doing.

The below are guidelines for the projects – I encourage all to go above and beyond the descriptions given below. Grading will be based on how well you implement the project and demonstrate it (live demo and PowerPoint presentation), your understanding of the general concepts, and the features and functionality of your program.

During the week of November 28th, I will hold open office hours where you can come and show me your program so far – more detailed schedule information later. I **strongly** encourage you to come and verify you have the correct project concepts. I will provide preliminary feedback to you. You may of course setup an appointment with me or the TA at other times if you have questions.

This final assignment will take effort and I vehemently encourage you to start working on the assignment well before the due date – i.e. today! As with the previous assignments, I prefer a nicely working, well-implemented and demonstrated project as opposed to a half-working buggy project with lots of half-baked complex features.

1. Simplification

The goal of simplification is to accelerate rendering by simplifying the objects and thus reducing the rendering cost. Simplification is usually done either by specifying a maximum allowable error from the original model or by indicating a desired performance and simplifying the model as necessary to achieve the goal. One way to accomplish this is to use a hierarchical spatial subdivision data structure as a simplification data structure. The simplification can be focused on single “objects” or can work for any collection of polygons and objects (e.g., an entire scene).

For example, using an octtree, you can implement a vertex clustering tree. The leaves of the tree contain one vertex per octtree node. Higher-level nodes can be used to represent a cluster of points with a single point (e.g., the midpoint of the octtree node). The trick here is to maintain reasonable triangle connectivity.

Another option, is to replace each octtree node with a colored box that approximates the geometry inside the node. Thus a very simplified object will have a “boxy” look to it. An alternative is to use a hierarchy of spheres instead of boxes (if you believe spheres are better approximations to a group of geometry than boxes).

Your imagination might find other approximation schemes.

In all cases, you must be able to “extract” a version of the scene/object at some desired level of detail. The user should be able to interactively specify (1) an error tolerance (e.g., via a GLUI slider) and this should result in a particular simplification; or (2) the user should be able to specify a desired rendering performance (i.e., a desired number of polygons to render) and the system should extract a model of (at most) that number of polygons.

Bells and whistles include allowing the simplified object to move (e.g., bounce) in a world box so that the performance acceleration can be viewed, automatically choosing a level of detail based on distance to the viewer, nicely handling shading, lighting, and texturing effects, and visualization tools to help the observer understand the simplification process.

The models provided for Assignment #3 maybe used as input. If you require additional models, please contact me and depending on your needs I will see what we can find.

2. Ray-Tracer

A ray-tracing program provides a totally different way for you to generate rendered images. Ray-tracing uses a very simple method and easily supports very apparently-complicated effects. Unfortunately, the computational cost of ray tracing is usually very high. It may take from minutes to hours to render a single image.

As explained in class, a ray-tracer shoots imaginary rays from the viewer through each pixel of the image plane and bounces the ray off opaque objects in the scene. The color of the ray is the accumulation of the colors seen (optionally attenuated by distance) of the first N-reflections. A ray-tracer also supports a myriad of other effects such as transparency, inter-object reflections, refraction, and shadows.

This assignment consists in creating a ray-tracer for at least several simple object types (e.g., sphere, box, etc). The scene should be described via some file format that allows easy editing of the scene. The ray-tracer should at least support first-order reflections

(this means if you have two shiny spheres next to each other, they will be able to see a reflection of each other on the surface of each other). A typical ray-traced image might be 256x256 or 512x512 pixels in size and each ray might bounce 1 or 2 times. This means 256x256x3 or 512x512x3 rays might be traced in the scene.

To accelerate rendering, a hierarchical spatial subdivision should be used to accelerate the ray-object intersection tests. For example, when using an octtree, rays are recursively checked for intersection against octtree boxes until a leaf node is reached. Then, testing is done against the object (or polygons) stored in the leaf node.

Additional features include supporting more object types (e.g., cone, cylinder, triangles, polygonal objects, etc), transparency, refraction, shadows, etc.

3. Occlusion Culling

The goal of occlusion culling is to determine, before transforming all the geometry, what subsets will not be visible in the current frame. This accelerates interactive rendering performance. Your program should exhibit interactive performance.

View-independent and view-dependent occlusion-culling algorithms are possible. View-independent occlusion-culling algorithms determine what subsets of the model are not visible from an area of the model regardless of exactly where the viewpoint is within that area (if the “area” is the entire model, it determines subsets of the model never visible). View-dependent occlusion-culling algorithms determine for the current viewpoint what subsets are not visible.

An approximate algorithm can be implemented by using a hierarchical spatial subdivision and assigning to each node of the tree an “opaqueness” value. Then, given a viewpoint, the opaqueness of a sequence of visible octtree nodes is accumulated and once it passes a threshold value, all subsequent nodes behind the last node are considered not visible. A conservative algorithm would tend to give low opaqueness to each node and cull fewer objects but tend to produce visually correct algorithms. An aggressive algorithm might cull a significant amount of the model in each frame but might also cull visible geometry.

You can compute approximate view-dependent opaqueness values for a node by rendering the node from different viewpoints and storing an estimated opaqueness. This value could be an estimate of how much of the background was covered by the geometry in the node when it was rendered from the current viewpoint.

Your imagination might find other approximation schemes.

Additional features include being able to control the aggressiveness/conservativeness of the occlusion culling algorithm, displaying rendering performance improvements, and allowing the object or scene to bounce within a world box.

4. Collision Detection

Collision detection is the art of detecting if two objects collide (or have already penetrated each other) and then performing some sort of action (e.g., a bounce). Exact collision detection for general objects is hard and time consuming.

The goal of this final assignment is to implement an approximate collision detection scheme using hierarchical spatial subdivisions. The assignment should have a large static scene (e.g., one of the models provided for assignment #3) and then allow one or more objects to bounce around the scene. The allowed complexity of the moving object and physical correctness of the bounce operation depends on the sophistication of your project. Your program should exhibit interactive performance.

A simple scenario is to create an octree data structure for the large static scene and to create a moving octree data structure for a moving object with an approximately convex shape. The collision algorithm is thus to intersect two octree data structures. If there is an intersection, you perform a bounce operation. The approximate nature of the algorithm comes from only intersecting octree nodes. You may also have more than one object bouncing within the scene.

Additional features include supporting angular velocity, more complex moving objects, and having a virtual “cannon” out of which the moving objects are shot out from – this allows for a basic game setup.

Demo Day

Final Assignment demo day is December 6th. Closer to that date we will setup a schedule via a democratic algorithm. Essentially, we will put up department-wide announcements, provide food and snacks, and have a demo-fest. Each student will present and demonstrate her/his assignment. You must also provide on that same day a CD containing your source code, binaries, project and presentation – program must be executable from the CD. No extensions, no late penalties, no late passes, and no exceptions to this due date will be given.

If you have more questions, please see myself or the TA.

Good luck!