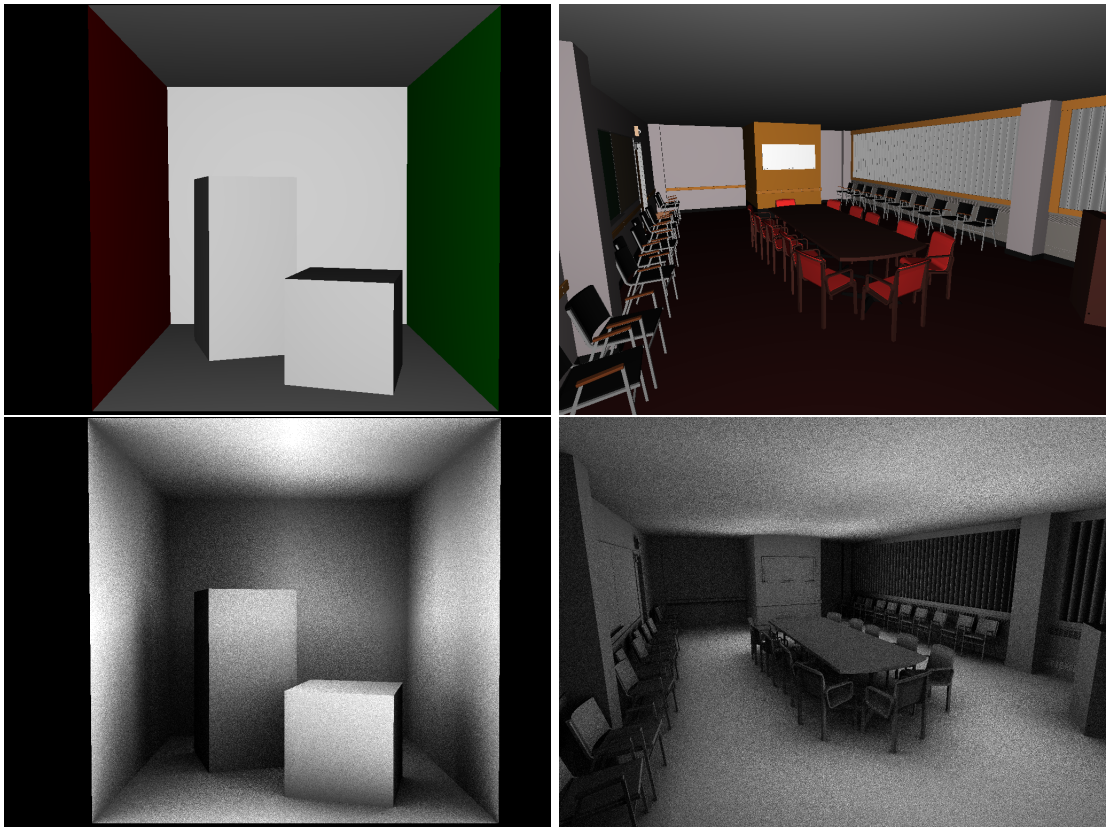
T-111.5310 Vuorovaikutteisen tietokonegrafiikan jatkokurssi

Assignment 1: Accelerated Ray Tracing

(Updated 4.2.2013)

Due February 27th at 23:59.

In this assignment, you are provided with a basic implementation of a naive ray tracer. It is capable of rendering images, but in practice it is prohibitively slow because it does not use an acceleration structure. Your task is to implement the missing bounding volume hierarchy builder routine, and to modify the ray intersection function to take advantage of the BVH. In addition, you will use the ray tracer to implement a simple but surprisingly effective shading trick called ambient occlusion (AO).



The remainder of this document is organized as follows.

1. Getting Started
2. Summary of Requirements
3. BVH Construction and Traversal
4. Ambient Occlusion
5. Extra Credit
6. Submission Instructions

1 Getting Started

1.1 Sample solution

We provide a sample solution executable (`code/sample_solution.exe`) that contains a full implementation of the required features. Familiarize yourself with the sample solution to see what you'll be implementing. The viewer comes preloaded with three sample scenes, which can be loaded by pressing 1, 2 and 3 on the keyboard. You can fly around in the model using the mouse and/or the arrow and W, A, S and D keys. Use the mouse wheel to adjust the flight speed. You can save the current camera position and other settings to disk by pressing Alt and a number key; the saved state can later be loaded by pressing the corresponding number key.

The three scenes included are the Cornell box, which is a very simple and low-polygon model consisting of various boxes. The second scene is the same model, except with a finer tesselation; this may be useful when debugging your implementation. The third scene is a much more complex model of a conference room.

The real-time viewer uses OpenGL to render the scene, and it is provided to you in the base solution.

The features you will implement are the following. When you click the "Trace Rays" button (or press Enter), the program renders the model from the current viewpoint using the ray tracer. The scene will then freeze, as the ray tracer is not real time. You can toggle between the ray traced image and the real time viewer using the "Show Ray Tracer result" button (or Space). The ray tracer uses a very simple headlight shading that does not produce particularly pretty pictures. To switch to the more advanced ambient occlusion shading, click the "Ambient Occlusion shading" button (F2) and re-render. The headlight is again activated by "Headlight shading" button (F1). Note that the rendering can take several seconds, during which the program may behave as though it had crashed. You can adjust the quality and the look of the AO effect using the "AO rays" and "AO ray length" sliders. The renderer also reports the number of rays shot per second in the screen corner.

1.2 Base solution

Next, you should compile the provided source code using Visual Studio 2010 and run it (preferably from the VS environment, so that the relative data paths will be correctly found). Remember also to switch to the Release configuration unless you need the debug features. The program is otherwise like the sample solution, except the AO rendering produces a black image, and the ray tracer is very slow (you should probably test it only in the very simple Cornell box mesh at first.)

Familiarize yourself with the code of the base solution. Majority of the code in the project manages the real-time viewer, mesh loading and other infrastructure; in addition, we provide a fast ray-triangle intersection routine. You do not need to understand this code in detail (nor modify it), unless you e.g. want to add new buttons for special features you have implemented. The relevant source files are `RayTracer.*` and `Renderer.*`. In particular, the header files give a concise overview of the different components in the renderer.

2 Summary of Requirements

- Bounding volume hierarchy (BVH) acceleration structure builder. The base solution calls the function `RayTracer::constructHierarchy` upon mesh load. The implementation of this function is a stub, which you will need to replace with your own BVH builder.
- Ray traversal using the BVH. The base solution contains a function `RayTracer::rayCast`, which currently implements a naive intersection routine that simply tests the ray against every triangle in the scene. Your task is to replace this with a hierarchical traversal that uses the BVH you constructed. In the leaf node, you may use the existing code.

- Ambient occlusion shading. You will write an implementation for the stub function `Renderer::computeShadingAmbientOcclusion`, which computes the AO shading at a given surface point by casting a randomly chosen set of rays from it, as described below. In addition, you will implement the helper function `Renderer::formBasis`, which forms a transformation matrix to the local surface coordinate system.

The expected output of your renderer on the provided example scenes is shown in the images on the first page.

Completing the BVH assignments according to requirements will give you 8 points and the AO assignment will give you 2 points. For extra credit, you may do additional work detailed in the end of this document.

3 BVH construction and traversal (8p)

The overall ideas of the BVH construction and ray traversal are detailed in the lecture slides.

The BVH construction is implemented in `RayTracer::constructHierarchy`. The function receives a reference to a vector containing all triangles in the scene.

You will need to define a BVH node class. It should contain two pointers to child nodes, so that the nodes form a binary tree. As detailed in the lecture slides, the nodes should not contain explicit lists of triangles or vertices. Instead, the node class should merely store the interval (first and last index) of triangles belonging to the node in the global triangle list. In addition, the node must store its bounding box.

On a high level, `constructHierarchy` has the following pseudocode (adapted from the lecture slides):

```
struct Node
{
   Vec3f bbMin, bbMax;           // axis-aligned BB
   int startPrim, endPrim;       // these are indices in the global list
   Node* leftChild, rightChild;  // these are NULL if node is leaf
}

function constructHierarchy()
{
   Node root = new Node();
   constructTree( 0, num_prims-1, root ); // recursively builds entire tree
}

function constructTree( int startPrim, int endPrim, Node N )
{
   // assign list interval of triangles to current node, compute its BB
   N.startPrim = startPrim;
   N.endPrim = endPrim;
   [N.bbMin, N.bbMax] = computeBB( startPrim, endPrim );

   if ( endPrim-startPrim+1 > MAX_TRIS_PER_LEAF ) // terminate?
   {
      // decide how to split primitives
      S = chooseSplit( startPrim, endPrim );
      // perform actual split: shuffle the indices in the global list
      // so that they're split into two intervals; return the index that
      // separates the two intervals
      splitPrim = partitionPrimitives( S );
```

```
        // create the left child and recursively call this
        // function with the left triangle interval
        N.leftChild = new Node();
        constructTree( startPrim, splitPrim, N.leftChild );
        // similarly for right child
        N.rightChild = new Node();
        constructTree( splitPrim+1, endPrim, N.rightChild );
    }
}
```

Use the spatial median or object median to determine the split. For those familiar with C++ STL algorithms, `partition` and/or `sort`, along with lambdas, may come in handy here.

You will replace the brute force ray intersection routine in `RayTracer::rayCast` with a ray traversal that uses the BVH. In the leaf nodes, you can use the existing implementation to intersect against the few remaining triangles. Use the simple slab intersection methods described in the lecture slides, and pay attention to special cases, such as when the ray origin is inside a given bounding box. Notice also that in our implementation the ray direction R_d is not a unit vector. Rather, its length determines the desired maximum distance of the intersections, so that the intersections will be within the line segment $[R_o, R_o + R_d]$, where R_o is the ray origin.

4 Ambient occlusion (2p)

Ambient occlusion (AO) is a simple and very popular shading effect that has no strict physical basis in reality. One can roughly consider it as an approximation to illumination from a perfectly cloudy sky. It tends to give a smooth shading that nicely accentuates the shape and details of the geometry.

The idea is as follows. The renderer has shot a ray through a pixel, and found an intersection at some point in the model. We then need to determine what color the pixel should receive; this process is called shading. AO is a shading method that, roughly speaking, evaluates how much unblocked "skylight" the point receives from all directions around it. If the point is in a crevice (such as a room corner), it will be dark; if it is in an open, unoccluded space (such as the middle of a floor), it will be bright. In practice, this is done by shooting several (say, 16 or 256) rays from the point to random directions using some maximum hit distance, and computing the percentage of rays that were occluded (i.e. hit another surface).

4.1 Implementation

In practice, the shading will be implemented in the `Renderer::computeShadingAmbientOcclusion` function, which receives pointers to all relevant information in its arguments. The information you will actually need is the surface hit point (see RayTracer::getIntersectionPoint) and the surface normal at that point. The normal can be computed from the triangle vertex coordinates using a cross product (see the implementation of `computeShadingHeadLight`). To ensure that the surface will be shaded correctly when viewed from either side, you should flip the normal if the the camera is in the backside of the triangle. You should also nudge the hit point very slightly (say, 0.001 units) towards the camera, because otherwise the rays leaving the point may hit the surface itself due to rounding errors.

Ray direction generation. The ray directions are easiest to generate in a local coordinate system, from which they can then be rotated into alignment with the surface normal. The procedure for generating a single suitably distributed random direction in local coordinates is as follows:

1. Pick a uniformly distributed random point in the 2-dimensional unit circle on the xy-plane. This is easiest to do using rejection sampling: repeatedly draw two random numbers x and y in the interval $[-1, 1]$, until you hit upon a pair that is inside the unit circle (i.e. $x^2 + y^2 \leq 1$); accept this sample.

2. Lift the point vertically onto the 3-dimensional unit sphere by setting $z := \sqrt{1 - x^2 - y^2}$.

The resulting unit vectors $[x\ y\ z]^\top$ will be distributed on the upper half of the unit sphere according to a cosine distribution (the reason we use this distribution will be explained in future lectures.)

The `Renderer` class has a member variable `Random m_rand` which can be used to generate the uniformly distributed random numbers.

Rotation to normal direction. Finally, the direction must be rotated so that the rays will be distributed around the surface normal instead of the z-axis. You will implement the generation of the rotation matrix R in the function `Renderer::formBasis`. The basic idea is that R must transform the z-axis vector $[0\ 0\ 1]^\top$ into the surface normal vector N, and it must be a rotation matrix. The first requirement is easy to implement: if we put N into the third column of R, then clearly $R\ [0\ 0\ 1]^\top = N$, regardless of the other columns in R.

The other two columns of R must be unit length and perpendicular to N and each other, because otherwise R is not a rotation matrix. In other words, the columns of R must form an orthogonal basis. We can construct a unit vector T that is perpendicular to N by picking any vector Q that is not parallel to N, taking the cross product $Q \times N$, and normalizing. This is because the cross product always returns a vector that is perpendicular to both of the operands. One way to pick a Q that is guaranteedly never parallel with N is to first set $Q := N$ and then replace its smallest element with 1.

We now have two perpendicular vectors N and T. To get a third one that is perpendicular to these both, we simply take the cross product $B := N \times T$. The matrix R can now be built by stacking the three column vectors side by side, as $R = [T\ B\ N]$.

Putting it together. The ambient occlusion is now easily computed by repeating the following procedure `Renderer::m_aoNumRays` times. Pick a random direction, rotate it to the normal direction, and trace a ray from the surface point to this direction with maximum length `Renderer::m_aoRayLength`. Note that the trace function uses non-normalized vectors, so that the length of the ray vector itself determines the desired maximum distance. Hence you should trace rays `m_aoRayLength` $* \tilde{u}$, where $\tilde{u}$ is a unit direction.

The returned pixel color is the number of rays that returned no hits, divided by `m_aoNumRays`.

5 Extra credit

5.1 Implement the Surface Area Heuristic in BVH construction (3p)

The Surface Area Heuristic (SAH) is a simple and effective heuristic for deciding the split plane placement in BVH construction. Familiarize yourself with the paper (`http://graphics.ucsd.edu/courses/cse168_s06/ucsd/heuristics.pdf`) and implement the method in your BVH builder. Include a toggle to enable and disable the feature, and compare and report the rendering performance against the simpler method (in rays per second).

5.2 Implement proper antialising for primary rays (1-2p)

Let's do it the proper way this time.

This means that each pixel has its own prefilter (you can use a box, Gaussian, or so-called Mitchell-Netravali bicubic filter, see `http://www.cs.utexas.edu/~fussell/courses/cs384g/lectures/mitchell/Mitchell.pdf`)sitting at its center. For non-box filters, the filters of neighboring pixels overlap (this is controlled by the standard deviation of the Gaussian, or the width of the M-N bicubic).

Draw N random samples from each pixel and evaluate the color. The samples are distributed within the pixels in some fashion, each better than the previous: uniformly random, stratified, or low-discrepancy (`http://cg.informatik.uni-freiburg.de/course_notes/graphics2_04_sampling.pdf`). If you do this, we suggest you implement the sample generators in a way that allows you to visualize the results directly, e.g., in Matlab, or a special mode you build in to the viewer itself.

When you've computed the RGB color $C = (C_r, C_g, C_b)$ for a sample at screen position (x, y), you then loop over all the pixels i whose filter overlaps (x, y). For this, you must know the width of the filter, in pixel units. (Note that this is particularly simple for a one-pixel non-overlapping box filter.) Then, you evaluate the filter $f_i(x, y)$ of the i:th pixel to get a weight w_i. Then, you accumulate $w_i * (C_r, C_g, C_b, 1)$ to the ith pixel in the frame buffer (note that the fourth channel keeps track of the total weight accumulated in this pixel).

Once all samples have been drawn and processed this way, you loop over all the pixels and divide by the accumulated weight in the fourth component. Note that you must initialize the fourth channel to zero in the beginning before you start rendering for this to work (starter code doesn't do this, it just overwrites everything).

Note that if you take multiple primary samples per pixel, you will want to decrease the number of AO rays per primary ray to maintain the same number of AO rays per pixel.

Uniform random and stratified sampling give 1 extra point, low discrepancy (doesn't matter which) 1 more.

5.3 Implement better sampling for ambient occlusion (1-4p)

The solution executable uses low-discrepancy Sobol' sequences for generating the samples. It's not doing it in the most sophisticated way possible, but you can notice there is a very significant decrease in the amount of noise when comparing to uniformly random samples.

Sobol samples on the left, uniform random samples on the right.

If you perform stratified sampling, you get 1 point. Designing your pixel antialiasing and AO sampling to jointly co-operate on a low-discrepancy sequence will earn 4.

5.4 Multithread your renderer (3p)

Design your BVH and ray tracer code to be thread-safe, such that you can break the loop over scanlines into multiple threads. NOTE that this will require you to think about the ray tracer state; it's not OK to keep just one version of the intersectionPoint member any more, for instance, because it will be overwritten by the concurrent threads. You may want to get rid of the internal state and convert to a Hit structure that gets returned by rayCast() that carries the hitpoint and triangle along.

See base/MulticoreLauncher.hpp in the framework for an easy-to-use way of distributing work along multiple threads. The header contains an example.

This should get you a large speedup.

The solution code is single-threaded only.

5.5 Parallelize your ray tracer using SIMD (5-10p)

This is a large change to the underlying code, both for the actual tracer and the renderer, which is currently based on shooting one ray at a time.

Modify your ray tracer to trace multiple rays at once using the 4- or 8-wide SIMD registers available in modern CPUs. You may want to look into Matt Pharr's ISPC compiler (`http://ispc.github.com/`) for an easy way of writing data-parallel code without the agonizing pain of using the assembly intrinsics. Since you only trace a small number of rays at a time, you can make use of this in the ambient occlusion rendering loop without re-engineering the scanline loop above. **This will give you 5 points.**

In practice, it is best to re-engineer the rendering loop into two interleaved stages: first, you generate a number of rays into a batch, then you trace the batch all at once, after which another pass over the results updates the associated pixels. This is sometimes called "wave-based" ray tracing. Fortunately, the current case is simpler than the fully general recursive path tracer, because each primary ray shoots a fixed number of shadow rays in the ambient occlusion renderer. **Re-engineering your renderer to be wave-based, such that you process large batches of rays at a time, gets you 5 more extra points.**

5.6 Write a GPU ray tracer (10p)

This is an even larger step, which requires you to trace even larger batches of rays at a time. It is crucial that your renderer is wave-based (see above). See Aila and Laine for tips on how to do this efficiently (`http://www.tml.tkk.fi/~timo/HPG2009/index.html`) – and do not hesitate to ask questions!

5.7 (NEW) Visualize your BVH construction (2-5p)

Write code that visualizes the BVH construction using OpenGL.

For instance, this could mean that for each recursive call to the construction function, you briefly visualize the triangles currently under consideration; then, visualize how they are split into left and right subtrees by coloring the triangles in the left side blue and red on the other side (say). When you reach a leaf, display the triangles in another color. You will probably want to draw the entire scene in some translucent color so that the triangles indicated as described above have some context.

Keep this in mind as a guideline: a good visualization is one that you can capture into a video and show on the lecture as a teaching tool.

You will receive a score depending on how informative your visualization is.

5.8 (NEW) Ray tracing in a BVH (2p)

You can also think of visualizing the traversal of a single ray in the BVH. Here, a 2D scene is probably more informative.

Again, ask yourself: how would you visualize the traversal process such that the essential points come across?

6 Submission Instructions

You are to write a README.txt that answers the following questions:

- Your name and student number at the beginning of the document.
- Did you collaborate with anyone in the class? If so, let us know who you talked to and what sort of help you gave or received.
- Were there any references (books, papers, websites, etc.) that you found particularly helpful for completing your assignment? Please provide a list.

- Are there any known problems with your code? If so, please provide a list and, if possible, describe what you think the cause is and how you might fix them if you had more time or motivation. This is very important, as we're much more likely to assign partial credit if you help us understand what's going on.
- Did you do any of the extra credit? If so, let us know how to use the additional features. If there was a substantial amount of work involved, describe what how you did it.
- Got any comments about this assignment that you'd like to share?

You should create a single .zip archive containing:

- Your whole Visual Studio project.
- The README.txt file.

Submit it online in Optima by February 27th at 23:59.
This assignment does not require the submission of an artifact.