Ambient occlusion and Image-Based Lighting Extensions for RTPara Distributed Ray-Casting Application

for Hewlett-Packard Scalable Visualization Array

August, 2008

Budapest University of Technology and Economics

Department of Control Engineering and Information Technology

Copyright 2008 Budapest University of Technology and Economics,
Department of Control Engineering and Information Technology.
http://www.iit.bme.hu/

Contents

Chapter 1

Introduction

This document gives an overview of the extensions designed for the parallel isosurface ray-casting application called *RTPara* developed by *BME IT2* in 2007 [2]. The application allows the visualization of volume datasets in a distributed environment by rendering isosurfaces [1] using screen space or object space decomposition approaches at interactive frame rates. Rendering parameters like the viewing direction, isosurface threshold, etc. can be interactively modified, the program provides an immediate visual feedback. In the available documentation the implemented isosurface ray-casting algorithm was presented in detail including the parallelization approaches (screen space and object space), the *ParaComp Library* [3] bindings, the installation and usage manual, and also the relevant implementation aspects were covered.

The original application supports basic ray-casting rendering with *Phong-Blinn* and *Lambert* BRDF models. This extension adds new features to the software's rendering capabilities. A fake global illumination method called *ambient occlusion* has been integrated with a cube map based image based lighting technique.

In the second Chapter of this document the algorithmic background of *ambient occlusion* is introduced.

Generated images and measured frame rates are reported in the last chapter.

Chapter 2

Algorithmic Background

Ambient occlusion is a shading technique in 3D computer graphics which helps add more realism to scenes with local illumination models. Local illumination reflection models compute a physically inaccurate, simplified lighting without taking into account multiple light bounces. Unlike local methods like *Phong shading*, *ambient occlusion* is a global method, the illumination at each point is affected by the surrounding geometry, however it is only an approximation of the full global illumination.

Ambient occlusion is most often calculated by casting rays in every direction from the surface. The surface is darker if most of the rays hit an object in the scene, because the surface is occluded. Rays which reach the background (no collision detected in a specific direction) increase the brightness of the surface.

Ambient occlusion has been popularized in production animation, video games and other areas - where interactive feedback is essential - due to its relative simplicity and efficiency. In the industry, ambient occlusion is often referred to as *sky light*. The algorithm has the property of offering a better quality perception of the 3d shape which is important when huge volumetric data sets are rendered.

The occlusion A_p at a point p on a surface with normal $\vec{N}$ can be computed by integrating the *visibility function* over the hemisphere Ω.

In addition to the ambient occlusion calculation, a *bent normal* $\vec{N_b}$ can be also generated. This vector points in the average direction of the unoccluded samples. The

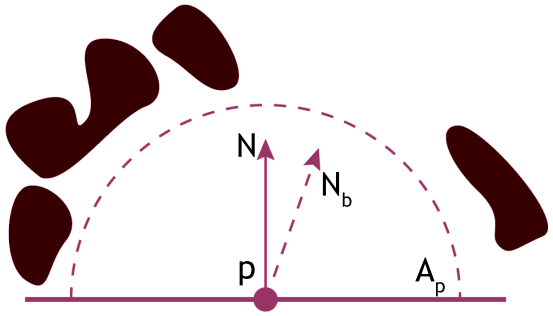

Figure 2.1: *Ambient occlusion A_P at surface point P with normal vector $\vec{N}$ over the hemisphere Ω*

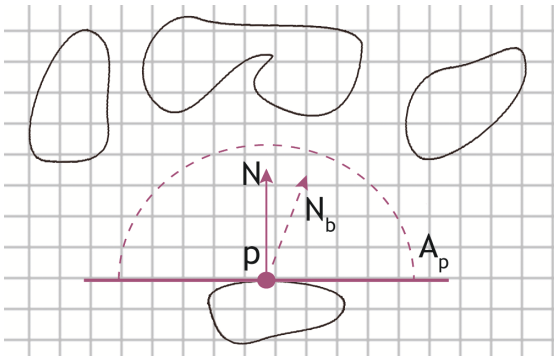

Figure 2.2: *Ambient occlusion for volume rendering. Voxels are sampled along random ray direction. If the density of the tested voxel is equal with the iso value, intersection is found, surface point P is occluded from this direction*

bent normal can be used to look up incident radiance from an environment map to approximate image-based lighting. In our implementation we use dynamic ray-traced ambient occlusion together with environment mapping for approximate *IBL* lighting.

2.1 Ambient Occlusion in Volume Rendering

Usually, traditional volume rendering approaches use local illumination models for surface shading which do not provide enough support for easy examination of complex data. Therefore, quality of the rendered image should be extended with approximate global illumination model. For the ray-caster application we have reimplemented our GPU shader model to calculate volume ray-traced ambient occlusion over the original shader lit by a cube map environment.

For the occlusion effect random directions need to be generated which are uploaded as an 1D texture. For the random vectors points with uniform distribution are placed inside of a unit circle. The points inside the circle are projected up to the unit hemisphere facing the normal vector. The vector pointing from the current surface point to the projected point is a random direction for the occlusion effect. In the shader code the number of the random samples can be changed for quality and speed balancing purposes.

For each direction the occlusion has to be decided. The algorithm travels the volume grid and checks whether the current density value belongs to the displayed isosurface. If the density test passed for the voxel then the ray intersection test is terminated and the fact of the intersection is registered. From the random rays and intersection results a *bent normal* is produced for each surface point during the rendering. The *bent normal* helps approximate the environment lighting. The *bent normal* is used to determine the main direction of lighting taking into account the surrounding geometry. A cube environment texture is also uploaded and indexed by the *bent normal* vector for more realistic lighting conditions.

2.2 Environment Mapping

In computer graphics *environment mapping* is a commonly used method of simulating complex surrounding surfaces by a precomputed texture image. The texture stores the image of the environment surrounding and can be fetched its color values from the specific directions. There are different ways of storing the environment map. *Cubic environment mapping* unfolds the environment onto the six faces of a cube, therefore it stores six square textures. The *spherical environment mapping* approach uses a single texture which contains the image of the surrounding as reflected on a mirror ball.

The technique is widely used because of its efficiency instead of the classical ray tracing reflections which calculates environment effects by shooting rays and following their optically exact path. However, the method is a crude approximation of the classical ray tracing.

In some cases real-world scenes are extended with computer generated scenes. The advantage of the method is that real-life lighting conditions can be mimiced by environment maps.

2.2.1 Cube Mapping

Our implementation contains the *cube mapping* technique that takes a three dimensional texture coordinate and returns a texel from a given cube map. The texture coordinate is a vector that specifies which way to look from the center of the cube mapped cube to get the desired texel.

Cube mapping is usually preferred, *sphere mapping* yields to more distortion, which needs a higher resolution map. Dynamic cube map generation is also more simple for realtime simulated reflections and lighting effects.

2.3 Ambient Occlusion Shader Implementation

The ambient occlusion extension has been realized as a Nvidia pixel shader. The shader extension gets the environment map texture, the surface iso value, and the value of the occlusion sampling. Uniformly sampled points are chosen inside of a unit circle. Points are projected up onto the hemisphere. Vector direction is generated in tangent-space coordinate system. In the specific direction the algorithm travels with predefined iteration steps. Fragment object positions are stored in the *pos_tex* texture. Rays are emitted from this point fetched from the texture. Occlusion is checked for the generated ray direction starting from the fragment object position. Current voxel density is read via a texture-space conversion of the current voxel from the *vol_tex* volume 3D texture. If the current density value is higher and the previous density value is lower then the *iso_value* than the current point is occluded from that direction based on the current isosurface.

Ambient occlusion is composited over the original shader. The occlusion pass is a grayscale image which is darker in more occluded areas.

```
struct fragmentIn {
    float2 unit_tc : TEXCOORD0;
    float4 fragment_color : COLOR0;
};
```

```
struct fragmentOut main(
        fragmentIn IN,
        uniform float4x4 mv_mat : state.matrix.modelview,
        uniform float4x4 tt_inv : state.matrix.texture[0],
        uniform samplerCUBE env_tex,
        uniform sampler3D volume_tex,
        uniform sampler2D pos_tex,
        uniform sampler2D grad_tex,
        uniform sampler1D lu_color_tex,
        uniform sampler1D rand_tex,
        uniform float iso_value,
                uniform float samples
        )
{

    float2 unit_tc = IN.unit_tc;
        float4 pos_obj = tex2D(pos_tex, unit_tc);

        // number of intersected rays;
    float hits = 0.f;

    float u, v;
    float theta, phi;
    float3 randVec, O, P, newDir;
    float3 bentNormal = float3(0,0,0);

    // occlusion
    for (float i = 0.0f; i < samples; i=i+1) {

        u = -1 + 2*i/samples;
        v = -1 + 2*i/samples;

        // tangent-space base vectors
        O = cross(N, float3(0,0,1));
        if (length(O) < 0.0001) { O = cross(N, float3(0,1,0)); }
        P = cross(N,O);

        float x0 = 2*u - 1;
        float y0 = 2*v - 1;

        if ((x0*x0+y0*y0)<1) {
            // z from projection onto the hemisphere
            float z0 = sqrt(1-(x0*x0+y0*y0));
            // random direction
            newDir = O * x0 + P * y0 + N * z0;
        } else continue;

        // intersection test
        float4 raydir = float4(newDir.x,newDir.y,newDir.z,1);
        // iteration step
        const float dt = 0.01f;

        // intersected point
        float3 p;
        float4 p4;
        float4 p4text;
        float4 p4ray;
        float s;
        float vp = 0, vpold = 0, t;

        // is intersected ?
        bool found = false;
```

```
        // intersection check
        for(t = 0.01; t <= 0.9f; t += dt) {
            if (t != 0.01) {
                p = pos_obj.xyz + raydir.xyz * (t-dt);
                p4 = float4(p, 1.0);
                p4text = mul(tt_inv, p4);
                vpold = tex3D(volume_tex, p4text).r;
                p = pos_obj.xyz + raydir.xyz * t;
                p4 = float4(p, 1.0);
                p4text = mul(tt_inv, p4);
                vp = tex3D(volume_tex, p4text).r;
            }
            else {
                p = pos_obj.xyz + raydir.xyz * t;
                p4 = float4(p, 1.0);
                p4text = mul(tt_inv, p4);
                vp = vpold = tex3D(volume_tex, p4text).r;
            }


            // isovalue voxel density check
            if (vp > iso_value && vpold < iso_value) {
                    hits+=1.f;
                    found = true;
                    break;
            }

        }
           if (!found) bentNormal+=newDir;
    }

    // occlusion value
    float occlusion = hits / samples;
    // bent normal vector for lighting
    bentNormal = normalize(bentNormal);
}
```

Listing 2.1: Pixel shader for occlusion calculation

Chapter 3

Results

In this section execution results are presented for a four-node HP SVA cluster. Each node had a dual-core AMD Opteron 246 processor and NVIDIA Geforce 8600GT graphics card. The software environment was HP XC V3.2 RC1.

Table 3.1 and Table 3.2 show the results of the distributed running of the volume renderer application. The ambient occlusion calculation time decrease the simulation speed, but scales well on the cluster. Speed and quality can be balanced by parameters like *occlusion samples*. Less samples yield to noisy images but increase the FPS value.

occlusion samples	1 node	2 nodes	3 nodes	4 nodes
32	3.6 fps	6.8 fps	9.8 fps	12.9 fps
64	1.9 fps	3.8 fps	5.4 fps	7.1 fps
128	1 fps	2 fps	2.9 fps	4 fps
256	0.5 fps	1 fps	1.5 fps	2 fps

Table 3.1: *Results of the screen-space distributed ray-caster with ambient occlusion and IBL extensions. The rows show simulation results in FPS for 32, 64, 128, 256 occlusion shadow ray samples per surface point. Simulation is measured with the engine data set (256x256x118 voxels)*

occlusion samples	1 node	2 nodes	3 nodes	4 nodes
32	2.8 fps	5.3 fps	6.8 fps	8.2 fps
64	1.5 fps	2.9 fps	3.6 fps	4.5 fps
128	0.8 fps	1.6 fps	2 fps	2.6 fps
256	0.4 fps	0.85 fps	0.9 fps	1.3 fps

Table 3.2: *Results of the screen-space distributed ray-caster with ambient occlusion and IBL extensions. The rows show simulation results in FPS for 32, 64, 128, 256 occlusion shadow ray samples per surface point. Simulation is measured with the visible human data set (512x512x512 voxels)*

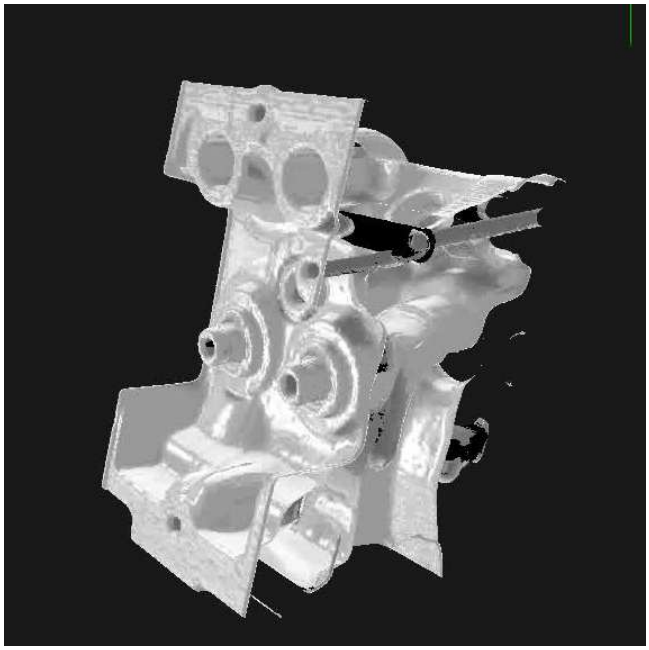

Figure 3.1: *Engine data set visualized by RTPara distributed ray-caster with ambient occlusion and IBL extensions (256x256x118)*

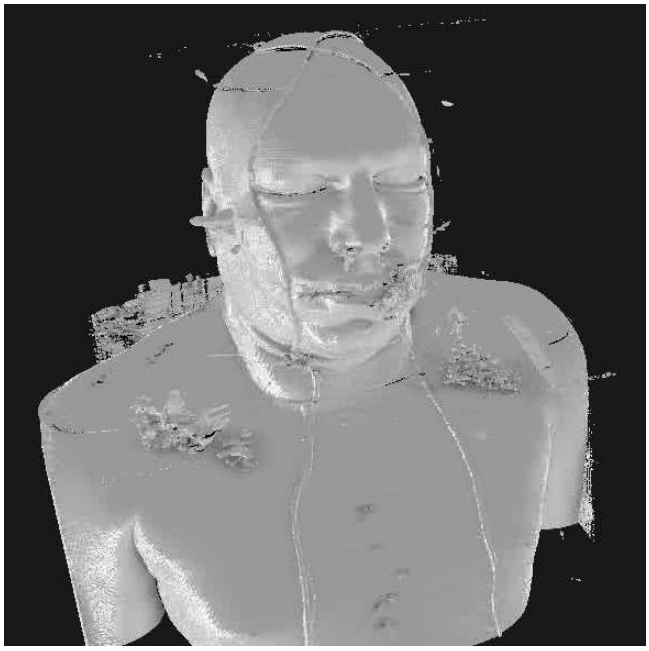

Figure 3.2: *Visible human data set visualized by RTPara distributed ray-caster with ambient occlusion and IBL extensions (256x256x256 slice)*

Bibliography

[1] S. Guthe, M. Wand, J. Gonser, and W. Straßer. Interactive Rendering of Large Volume Data Sets. In *Proceedings of IEEE Visualization 2002*, pages 45–52, 2002.

[2] BME IT2. RTPara - Distributed Ray-Casting Application for Hewlett-Packard Scalabe Visualization Array. *http://amon.ik.bme.hu/rtpara/*, 2007.

[3] Hewlett Packard. *HP Scalable Visualization Array Parallel Compositing Library Reference Guide*, 2007.