Volume-based Ambient Occlusion
with Voxel Fragmentation

PROJECT IN TECHNOLOGY FOR ADVANCED COMPUTER GAMES
TSBK03

Christopher Birger  chrbi049@student.liu.se
Erik Englesson  erien139@student.liu.se
Anders Hedblom  andhe893@student.liu.se
Abstract

Ambient occlusion is an effect for approximating the difficulty for light to reach tight areas in a 3D scene. Its purpose is to enhance realism in computer graphics without the necessity of using physically based methods that are often very computationally heavy. Ambient occlusion also increases our ability to perceive 3D objects. The effect can be achieved with several different methods but the basic idea is that the more close geometry that can be seen from a point, the darker it will be. For this project, as a part of the course Technology for Advanced Video Games - TSBK03, we created a volume-based ambient occlusion pass renderer. We voxelize the scene and store it as volume data in a 3D texture, which is sent into the graphics pipeline. A Sparse Voxel Octree was also implemented on the CPU. This can be visualized during runtime.

1 Introduction

In real time rendering applications there is not much room for accurate lighting calculations and therefore many tricks must be used to achieve believable results. If the lack of available computation time was not the case, the obvious choice would be to calculate a full global illumination model. Global illumination can be calculated using ray tracing methods where rays are sent from the camera and are then followed through the scene, to any of the light sources. At each surface hit, the irradiance, i.e, the incoming light is calculated by integrating all possible directions over the hemisphere, with respect to the normal of the surface. Practically, this means that a set number of rays are spawned at each surface hit and ray traced further into the scene. Because of this recursive nature, ray traced global illumination can’t be used in real time application. Compared to a local illumination model where each point is treated without regards to any other point, global illumination depends on the scene complexity.

When objects are close together and the hemisphere is partially occluded, less light will fall onto such points. This is exactly what ambient occlusion tries to approximate, i.e, how much of the hemisphere is occluded and how close other objects are. Ambient occlusion have been shown to increase the perception of scene geometry and aid in the apprehension of three dimensional objects or depth. Ambient occlusion can easily be solved with ray tracing techniques but in most real-time applications this is far too expensive to compute. In the standard graphics pipeline today there is no natural way of accessing all of the scene geometry. Therefore, there has been much research about ambient occlusion using much faster methods. Ambient occlusion was made popular in real-time applications with Screen Space Ambient Occlusion (SSAO) [1], which is a really fast method that can be calculated purely in a fragment shader.

Ambient occlusion is an approximation of light behavior due to the fact that it manipulates ambient light, which is also just a fake component in lighting. Ambient light is a very crude approximation for how light bounces around in a scene with diffuse surfaces. This causes objects to seem lit somewhat from all directions, not just the directions of the light sources. Ambient light ignores the shape of all geometry in the scene and does not therefore contribute to any realism. Setting the ambient intensity in a scene implies that all fragments get a lower illumination threshold, that all other light is added to. As the name states, ambient occlusion reduces this threshold for tight areas by subtracting an occlusion value from the ambient intensity.

Even though SSAO is a very effective way of creating the AO effect in real-time, it definitely has some quality flaws. There are other fast methods for creating AO, such as different volume-based approaches. By storing all geometry as volume data in a texture, it is possible to create ambient occlusion on the GPU, yet still having it view-independent and converging towards a correct result. In this project, we have implemented a volume-based ambient occlusion pass renderer that uses this principle.
2 Ambient Occlusion

In global illumination light is considered to reflect in directions over the hemisphere at each point in the scene. How much and which directions the light reflects in, is modeled through a bidirectional reflectance distribution function, \( f \), which depends on the incoming light direction, the viewing direction and the surface normal. If the surface is a perfectly diffuse, the BRDF is constant, since light is reflected uniformly over the hemisphere. Ambient occlusion stems from the fact that in narrow spaces such as corners, the hemisphere over these points will be partially covered. This will result in that such areas will not receive as much light as “open” areas. If we consider the rendering equation that calculates the excitant radiance in a direction \( \hat{\omega}_o \), from a point, \( \vec{x} \), as in equation 1.

\[
L_o(\vec{x}, \hat{\omega}_o) = \int_\Omega L_i(\vec{x}, \hat{\omega}_i) f(\vec{x}, \hat{\omega}_i, \hat{\omega}_o) V(\vec{x}, \hat{\omega}_i)(\hat{\omega}_i \cdot \hat{n}) d\hat{\omega}_i
\] (1)

Where \( L_i(\vec{x}, \hat{\omega}_i) \) is the incoming irradiance from direction \( \hat{\omega}_i \), \( V(\vec{x}, \hat{\omega}_i) \) is the visibility function which is either 0 or 1. The last term, \( (\hat{\omega}_i \cdot \hat{n}) \), models the fact that incoming light is projected onto a greater surface when the angle between the surface normal and incoming direction increases. Again, if we now consider that all the surface are perfectly diffuse and that we replace the incoming irradiance, \( L_i \), with a constant ambient light source, \( L_a \), we get the following equation.

\[
L_o(\vec{x}, \hat{\omega}_o) = L_a \int_\Omega V(\vec{x}, \hat{\omega}_i)(\hat{\omega}_i \cdot \hat{n}) d\hat{\omega}_i
\] (2)

The ambient light term is an approximation to the indirect lighting in the scene, which is often used in local lighting models where the position of the light source is not taken into account. It is now clear that when the hemisphere is occluded, the point will receive less light and vice versa. This happens to be very close to the ambient occlusion equation but we also need to add an attenuation function. Consider that we are looking inside an infinitely large room. Corners in the room should become darker since parts of the hemisphere will become occluded. However, in equation 2 there is nothing that models how close a surface is when evaluating the visibility function. Therefore, all points in this box would evaluate to the same ambient occlusion value. Using this intuition and the approximation made, we end up with a final ambient occlusion integral.

\[
A_E = \frac{1}{\pi} \int_\Omega V(\vec{x}, \hat{\omega}_i)\tau(\vec{x}, \vec{x}_i)(\hat{\omega}_i \cdot \hat{n}) d\hat{\omega}_i
\] (3)

Where \( \tau(\vec{x}, \vec{x}_i) \) is the attenuation function and \( \vec{x}_i \) is the intersection point if there is any. The most accurate solution to this integral is to use some sort of ray tracing method. In ray tracing, the integral is discretized into a sum over the hemisphere. The hemisphere is divided into set number of regions which corresponds to multiple directions, \( \hat{\omega}_i \). For each direction a ray is sent out in the scene. If the ray intersects the scene the visibility is evaluated to one and the ray is terminated. Figure 1 depicts two different surface points in the scene.

![Figure 1: Visualization of ambient occlusion at two surface points A and B.](image)

In figure 1, The AO value in point A is low since only two rays hit another surface far away at a high angle between the normal and the ray. The ambient occlusion at point B is high though, since many of the rays find close surfaces for rays with directions close to the normal.

Ray traced ambient occlusion converges to a correct result when more and more rays are used. This method is too slow for real-time applications, but is commonly used as a reference to the ground truth when comparing approximating real-time algorithms. Some of the most common real-time approaches are detailed in the following sections.
2.1 Different approaches to AO

We can conclude that ray casting the scene is a correct way of achieving the ambient occlusion effect, however, as it is so computationally heavy, the method is not feasible in real-time applications. In static scenes, it is possible to pre-render an ambient occlusion pass once, and apply this during runtime as a texture. This can give really nice and smooth shadows for most times, but immediately becomes a problem when objects start to move around. In todays games, physics and object manipulation is often as important part as the graphics, making it necessary to use other methods than ray casting if AO should be a part of the game.

Listed below are some well used illumination methods, for offline renderings, that do not really require the application of additional AO as they already tries to simulate the behaviour of light. Listed are also some ways of doing fast AO, mostly for real-time usage, where the effect has to be calculated in every single frame.

**Offline methods**
- Ray-tracing
- Photon mapping
- Radiosity

**Real-time methods**
- SSAO - Screen Space Ambient Occlusion
- HBAO - Horizon Based Ambient Occlusion
- Volume-based AO

2.2 SSAO

Screen-Space Ambient Occlusion [1] is a very fast and effective method for achieving the AO effect in real-time. It was developed at the Crytek company and made its debut in their game Crysis, 2007. The mind behind the technique is Vladimir Kajalin, who came up with the brilliant idea of using the built-in depth buffer, in the rendering pipeline, to calculate the occlusion values. The depth values are compared for objects that are close to each other in the view plane and based on their difference, a final AO intensity can be derived. Due to the fact that SSAO uses the camera depth buffer, it has to be calculated for every frame. Though, thanks to its efficiency this is fully possible without affecting the frame rate too much. SSAO quickly became the standard for AO in games and is still being used today. However, it definitely suffers from a few problems; the method can only take visible geometry into account. Objects that are outside the view plane or blocked by other objects are simply ignored. It also has difficulties in managing object edges. When a foreground object pixel is compared to a pixel belonging to the background, you may get huge differences in depth. This can cause problems when trying to blur the effect, to smooth out noise and aliasing, which can result in strange glorias around objects. Also, areas that are very occluded from the sides (in respect to camera direction) might appear brighter than they should due to lack of variance in depth. Therefore, SSAO will never converge to a correct result, no matter how many samples that are used.

Figure 2: A comparison between different methods for creating AO.
2.2.1 How it works

SSAO has the advantage that it is implemented purely in a fragment shader, which takes a framebuffer copied texture as input. This makes any RAM fetches redundant, thus increases the computing efficiency. The AO dedicated fragment shader reads the depth value for a pixel and compares it to neighboring pixels. How these pixels should be sampled can vary depending on the quality settings from the user. If an adjacent pixel has a smaller depth value, the occlusion value will be incremented for the current pixel in the fragment shader. An illustration of this can be seen in figure 3. Another advantage that the SSAO method has is its independency of scene complexity. As the number of fragments in the scene always are constant, it will take the same time to calculate the AO for a simple plane as for a scene with millions of polygons.

![Diagram](image.png)

Figure 3: Illustration of different depth values in 1D.

2.3 HBAO

Horizon-based Ambient Occlusion is an improvement of the SSAO method. It was released in 2008, presented by Louis Bavoil et al [2]. HBAO uses the angle between the horizon and the vector from an AO point to a point on occluding geometry, to march over all geometry closeby, and thereby achieving the total AO value. This resolves some of the problems from SSAO but the computational cost is also greater.

In many real-time applications, such as games, the difference can be hard to spot and it might not be worth the extra computation time.

3 Volume-Based Ambient Occlusion

Volume-based ambient occlusion, presented by G. Papaioannou et al[3], utilizes the same principles as when ray casting the stochastic hemisphere, as in equation 3. However, here the rays are marched within a volume representation of the scene. The scene geometry can be voxelized and stored inside a 3D texture, and then conveniently be traversed. As all the data from the scene exists in the texture, the method is view-independent. However, it is possible to only derive the ambient occlusion for the volume data that is visible on the screen.

In the implementation presented in this paper we apply a conservative voxelization step which will be described in the following section.

For now, assume that we have a volume representation of a scene, as a 3D texture, filled with ones where there is geometry and zeros everywhere else. When the geometry in the scene is rendered, each pixel fragment can be connected to a voxel fragment in the 3D texture. Given from the geometry is also the normal at that position. From here, rays can be marched out through the texture in directions within the hemisphere of the normal. As we march through the texture we try to linearly interpolate the value at each sampling point. If the ray intersects geometry, the ray will be terminated and the occlusion value for the current pixel will increase. This increment is dependent on how many rays that are casted. The final pixel value will be calculated as the difference between a set ambient value and the sum of all occlusion values.

3.1 Pros and Cons

One of the advantages with volume-based AO is that, as once the scene has been voxelized and stored in a 3D texture, the rest can be done in a fragment shader. In that sense it is possible to utilize the multi parallel computing power of the GPU, which allows for this to be run in real-time.
Furthermore, the technique is independent of scene complexity - AO is only calculated for voxel fragments that are visible on screen.

Unfortunately there are also some drawbacks. The texture memory on the GPU sets a fairly early limit on the size of the 3D texture. Less available memory means lower resolution of the voxel grid which inevitably leads to voxel shaped artifacts in the AO shadow contours. The method also scales badly with the number of pixel fragments that are being displayed. Imagine that, for every fragment, nine rays are casted over its hemisphere, where each ray reads from the 3D texture at 10 different positions. It will probably hit a voxel containing geometry before it has traversed all these steps, but as a worst case we have 90 texture fetches for each fragment. Finally, the method suffers from difficulties of creating ambient occlusion in areas that are tighter than the length of a single voxel. If occluding geometry is located in the same voxel fragment as for the pixel we currently calculate AO for, this geometry will be ignored. It is necessary to completely exit the own voxel fragment in the first ray step, otherwise, every ray would report detection of geometry, which would result in every fragment becoming black.

3.2 Bilateral smoothing

The ambient occlusion pass is rendered to a 2D texture, which is sent into another shader in a separate pass. Here, the AO texture is blurred with a Gaussian filter kernel, in order to remove some of the artifacts that arise due to the finite grid resolution. Though, the blur cannot be applied to the whole texture, as that would eliminate all the details in the scene. Only the AO shadows should be smoothed. Here, a technique is used that is much similar to SSAO. Bilateral filters use more information for the weighting of surrounding pixels than just the distance, to preserve sharp edges. Another radiometric value, such as depth can be used to achieve this effect. The Z-buffer depth values are saved in another texture, so that they can be accessed for all pixels in the AO texture. When fetching the pixel values within the filter kernel, from the AO texture, their depth value is also fetched and compared with the current pixel in the blurring fragment shader. The more a pixel’s depth value differs from the center pixel, the less it will be weighted for the final value. This technique is for instance used in the Surface Blur filter in Adobe Photoshop.

Figure 4: Example of an application of a bilateral filter.

4 Volume Rasterization Using The Graphics Pipeline

Volume rasterization or just voxelization is the method of sampling some function into a three dimensional discrete domain. The function can be any type of 3D discrete or continuous function, e.g. a level set, density function or in our case a surface or more precise, a mesh. Since the domain is discrete it has to be divided into some set of smaller regions which is commonly called voxels. Voxels are the three dimensional equivalence to pixels in an image, i.e, they have a position, coverage (size) and of course a value. A fragment is a pixel intersected by a triangle. The 3D equivalent is a voxel intersected by a triangle, called voxel fragment. In this paper we will always assume that the voxels are uniform in size, i.e, they are in the shape of cubes, see figure 5.

It is also important to note that there are at least two different coordinate spaces when dealing with voxel grids. The voxel-space is a local space defined over the number of voxel in each dimension i.e, \((i, j, k) \in [0, 1, 2, \ldots, N_x]\) where \(N_x\) denotes the number of voxels in each axis, \((* = \{i, j, k\})\). However, the voxels are also placed in world space, i.e, voxel \((i, j, k)\) actually represents some world space coordinate \((x, y, z)\). In order to perform this mapping, we must decide the world space size of a voxel, \(\Delta_w\). The mapping can then be performed as follows:
Figure 5: Voxelized version (left) of geometry (right).

\[(x, y, z)_{\text{world}} = \frac{1}{\Delta v} (i, j, k)_{\text{voxel}} \quad (4)\]

\[(i, j, k)_{\text{voxel}} = \lfloor \Delta v (x, y, z)_{\text{world}} \rfloor \quad (5)\]

where \(\lfloor ... \rfloor\) is the floor operation.

### 4.1 Surface mesh voxelization

Compared to a continuously defined three dimensional function, a surface mesh is quite difficult to rasterize since the surface is not explicitly defined. In this paper we assume that a mesh is a set of triangles and that each triangle is defined by three vertices. The basic idea of rasterizing a mesh, utilizing the graphics pipeline, is to use the hardware rasterizer to generate our voxel positions. First, a cubical frustum is defined in world space which specifies the domain of the volume representation. The trick is then to set the viewport or plane size to the dimensions of the voxel dimensions and orthographically project each triangle. The graphics pipeline rasterizer will then generate fragments that will correspond to voxel fragments. That is, the screen coordinates of the fragments will correspond directly to the voxel-space coordinate. The depth of the fragment can be used to calculate the voxel-space depth by multiplying with the dimensions of the voxel dimensions. The approach is very simple in theory but practically there are several issues that need to be resolved. We will begin by examining how a simple triangle is rasterized on to the screen.

Two dimensional triangle rasterization can be done with some scanline algorithm which will not be covered in this paper, though the result can be seen in figure 4.1. This figure shows two different types of rasterization. In figure (a), only pixels that have their center inside the triangle are rasterized, which is often called standard rasterization. This is not always good since it underestimates the boundary of the surface. Taken to three dimensions, this could result in holes in the surface when the angle between two triangles is very steep. Conservative rasterization, which has been used in figure (b), can be used to fix this problem. This method rasterizes pixels that are both intersected by the triangle and that are contained. How this can be achieved with the graphics pipeline is covered later. As mentioned, if the triangle is orthographically projected onto an image plane with the same resolution as the grid, the screen coordinate of each fragment will be the same as the voxel-space coordinate. However, in three dimensions the depth is also needed. The depth, \(d\), for any given fragment will be in the range \([0, 1]\) so it has to be converted by multiplying it with the dimension of the grid. The problem is that the when the triangle is projected and rasterized, the area of the triangle is decreased. It is therefore possible that the triangle intersects multiple voxels in depth, but the pipeline only generates a single fragment. In the next section we will see how to modify the standard rasterization pipeline to do conservative...
rasterization, that will alleviate the issues just described.

4.2 Conservative Rasterization

As mentioned, conservative rasterization is a technique that rasterizes all fragments that are intersected by a triangle. Conservative rasterization is applied both in screen coordinates and in depth, but these are two separate methods and are applied in different stages of the graphics pipeline. Because of the fact that the rasterization process is not open for modification, the only way to change it is to feed it with a different input. Therefore, to achieve the result depicted in figure 4.1 (b), the triangle must modified before sending it to the raster stage. One way of achieving this is by dilating the triangle, which was proposed in [5]. Their approach for convex polygons like triangles, is to put pixels on each vertex and calculate the convex hull with these pixels included, see figure 7.

![Figure 7: Adjusted over-estimated triangle.](image)

This can be done by enlarging the triangle to be aligned with the convex hull, shown in blue, and creating a bounding box that bounds the convex hull, shown in orange in figure 8 below.

![Figure 8: Illustration of the bounding box, which cuts off parts of the over-estimated triangle.](image)

The enlarging of the triangle and creation of the bounding box is done in a geometry shader. After the standard rasterization, fragments in the convex hull are created as intended, but also fragments in the enlarged triangle outside the bounding box. The first thing that is done in the subsequent fragment shader is to discard these unwanted fragments by checking if the fragment is inside the bounding box or not.

It is also needed to compute a conservative depth value, though conservative in the sense of depth is a bit different than in screen space. As mentioned, the problems stems from the projection step. In figures 9, a triangle is projected and rasterized to the image plane. The black points refer to fragments in the image plane and the grid refers to both pixels and voxels. Also, note that the triangle is seen above the projection dimension, i.e, the depth increases when moving away from the projection plane. In figure 9 (a) it is clear that some voxels are missed due to the fact the the triangle intersects several voxels in depth within the same fragment. Our goal is to produce the results in figure 9 (c).

![Figure 9: Going from (a) to (c) by using the z-min and z-max in (b).](image)

In order to achieve this, we must calculate a minimum and maximum depth value within the fragment, which can be seen in figure 9 (b). However, this requires a known depth change, $\Delta z$, within the fragment, which can be approximated by the partial derivatives of the depth, given by the following equation [6].

$$\Delta z = \frac{1}{2} \left( |\frac{\partial z}{\partial x}| + |\frac{\partial z}{\partial y}| \right)$$ (6)

The minimum and maximum depth value is then given by $z_{min} = z_c - \Delta z$ and $z_{max} = z_c + \Delta z$ where $z_c$ is the depth in the center of the fragment. Partial derivatives are often built in shader function in several shader languages e.g., the function $fwidth(\_\_)$ in GLSL. The final step in the rasterization process is to write to the 3d texture. At least in OpenGL this has been made possible since the addition of load/store methods in the release of OpenGL 4.2.
5 Sparse Voxel Octree

When a 3D texture is used to represent the scene most of the data will be unused, since the actual surface is just a fraction of the total volume. Sparse voxel octree (SVO) is a voxel data structure that only stores information in voxels intersected by the surface. It is also a tree structure where each node can have 8 children. Each node represent a voxel in space, e.g. the first node represents the whole scene as a voxel and its children nodes represents an 8th of its parents volume etc. The sparse nature of SVO is that each node does not have to have children. The subdivision rule can vary between different applications of the data structure. In our implementation it is based on the voxel fragments. See figure 10 for a visualization of our octree.

Figure 10: Octree visualization on leaf-node level.

5.1 Implementation

In our implementation the actual building of the octree is only done once at startup. This limits the current implementation to static scenes since moving objects will not be updated in the octree. Dynamic objects should be rasterized separately and stored in a way that it is easy to remove at the end of each frame. A simple but brute force solution could be to voxelize the whole scene, static and dynamic parts, at each frame.

The actual building of the octree is done in separate passes in a breadth-first manner, i.e. each level of the octree is done in order, which is described in [7]. Pseudo code for the building of our octree is listed below:

```java
for each level l
    for each voxel fragment vf
        markNode(getVoxelCoordinate(vf)); //Marks node based on voxel coordinate of vf
    end for
    for all nodes n in l
        if(n.isMarked())
            n.createChildren();
        end if
    end for
end for
```

This is more illustrated in figure 11. Our octree implementation is made in C++ but it is parallel in nature. Each voxel fragment can mark nodes independently and each node can be subdivided in parallel. When creating each level one by one, all nodes of the same level get stored close in memory which is good for cache since neighboring accesses are common.

5.2 Pros and cons

The SVO reduces the memory requirement significantly when compared to 3D textures. The 3D texture on the other hand can make use of the fast hardware interpolation. Crassin et al. [8] solved this problem by having each node store the 3x3x3 voxel neighbourhood in a local 3D texture instead of just storing the one voxel. There are problems with this as well, e.g. the most obvious problem is the higher memory requirement.

Both the 3D texture and SVO are spatial data structures which makes it easy to find neighbouring scene information. The SVO is more efficient when there is a lot of empty space between surfaces, since it can check large empty areas quickly by traversing the octree. Finding neighbours can also be optimized by storing pointers to neighbours in each node at the cost of more memory.
6 Visualization

6.1 Geometry

To clarify, the actual triangle scene is rendered every frame and is lit by using the information in the voxels. This makes the surface quality independent of the quality of the lighting, i.e. the actual scene geometry will never get blocky by grid artifacts. If artifacts from the grid appears it shows up in the lighting.

6.2 Grid - geometry shader

The SVO is visualized by sending the octree level to be shown and the positions of the voxels in the SVO to the GPU. The geometry shader then creates wireframe boxes centered on these positions scaled appropriately for the specified octree level.

6.3 Phong shading and AO

A scene rendered with ambient occlusion only, can look very appealing. However, when nothing else than ambient occlusion exists, it is impossible to judge its contribution to the final rendering, compared to a scene rendered without AO. Therefore, a basic Phong shader is also implemented. We are allowed to load any model into the program and rasterize it according to described method. The Sparse Voxel Octree is done on the CPU and remains there, thus it is not used for the AO render pass, but rather during the visualization of the geometry filled volume data. For this, wireframe cubes are created within a geometry shader, and can be turned on and off, as well as scaled in aspect to octree level, during runtime. We are also able to adjust the parameters for the AO shader, such as number of rays and the amount of blur.

The figures below show the result in some different aspects.

6.4 GUI

To make it easier to test out different settings for the rendering, we use the AntTweakBar extension for GLFW. This lets the user connect different parameters to a GUI and modify these during runtime. Being able to turn ambient occlusion on and off, adjusting the number of cast rays, how far they should reach, and how much the AO should be blurred are some of the parameters that we can change.

7 Results

The result of this project is an Ambient Occlusion pass renderer based on volume data. To see the difference between the use of ambient occlusion and having it turned off, a Phong shader is also implemented. We are allowed to load any model into the program and rasterize it according to described method. The Sparse Voxel Octree is done on the CPU and remains there, thus it is not used for the AO render pass, but rather during the visualization of the geometry filled volume data. For this, wireframe cubes are created within a geometry shader, and can be turned on and off, as well as scaled in aspect to octree level, during runtime. We are also able to adjust the parameters for the AO shader, such as number of rays and the amount of blur.

The figures below show the result in some different aspects.

Figure 11: The procedure of building the octree.
(a) Assassin’s Creed - Altair: 377,000 triangles.

(b) The Sibenik Cathedral: 1,200,000 triangles.

Figure 12: Two different renderings with ambient occlusion only.
Figure 13: Some different renderings comparing Phong shading only (left) with Phong shading and ambient occlusion combined (right).
Figure 14: *Illustration of some different SVO levels.*

Figure 15: *The SVO, visualized at its deepest level.*
8 Conclusion

In this paper we have described a volume based ambient occlusion model. This approach utilizes the standard graphics pipeline to rasterize the scene geometry into a three dimensional texture as described in [8], [4] and [6]. We have seen that this method produces visually pleasing results but suffers both from grid artifacts and slow computation time. It is however interesting to have seen that the computing power of today’s GPU’s can compute ray traced ambient occlusion in real time. We have also seen that the graphics pipeline can effectively be used to voxelize surface meshes. This has been made possible through OpenGL 4.2’s load/store functionality, which makes it possible to both write and read two arbitrary buffers or textures inside a shader program. The major conclusion we make is that volume based ambient occlusion is a feasible approach but much work has to be done to speed up the ray marching step which is the main bottleneck of the method.

9 Future work

SVO on the GPU To speed up the ambient occlusion calculations and to save a lot of memory, the Sparse Voxel Octree could be implemented on the GPU. This would imply using the SVO instead of ray marching a 3D texture. This would most easily be done in a compute shader available in OpenGL 4.3.

Implement voxel cone tracing Voxel cone tracing [8] is a method that tries to reduce the time to calculate an intensity for a pixel when using rays to gather information. Instead of sending out a lot of rays for each fragment, the idea of voxel cone tracing is to send out fewer rays that approximate the result as good as possible. These fewer rays can be seen as cones with the external point of the cones at the surface. The radius of the cones then determines at what voxel resolution the ray should sample, see figure 16.

Since the rays sample lower and lower resolutions while they get further and further away this method gets automatic LOD rendering. The mipmaps are just averages of lower resolutions which also can be used for blur effects. Voxel cone tracing works very well with ambient occlusion as well. In our implementation each fragment spawns a number of rays that samples a 3D texture. The conversion would be to spawn fewer rays per fragment and sample the appropriate 3D texture instead based on the radius of the cone.

The naive way of implementing this would be to use a high resolution 3D texture and its mipmaps. The problem with this method is of course the memory requirement. The more sophisticated way would be to use the sparse voxel octree data structure, described in section 5. A modification to our SVO would be to store scene information in all nodes instead of only the leaf nodes, where each node gets its values from averaging the values of its children [8].

Shooting a ray from a voxel fragment out in the scene would be the same thing as sampling octree nodes closer and closer to the root, which means that the accuracy of the check for occluding geometry becomes lower and lower. This naturally also contributes to smoothing out voxel artifacts in the AO.

References


