SCREEN SPACE AMBIENT OCCLUSION USING PARTIAL SCENE REPRESENTATION

By

CHAITANYA TIMMARAJU

Bachelor of Technology in Information Technology

Jawaharlal Nehru Technological University

Hyderabad, Telangana

2015

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 2017

SCREEN SPACE AMBIENT OCCLUSION

USING

PARTIAL SCENE REPRESENTATION

Thesis Approved:

Dr. Douglas R. Heisterkamp

Thesis Adviser

Dr. David Cline

Dr. Blayne Mayfield

Name: CHAITANY TIMMARAJU

Date of Degree: DECEMBER 2017

Title of Study: SCREEN SPACE AMBIENT OCCLUSION USING PARTIAL SCENE REPRESENTATION

Major Field: COMPUTER SCIENCE

Abstract: Screen space ambient occlusion (SSAO) is a technique in real-time rendering for approximating amount by which a point on a surface is occluded by surrounding geometry, which helps in adding soft shadows to diffuse objects. Most of the current methods use the depth buffer as an approximation to scene geometry to sample the occlusion factor. We introduce a novel technique which uses a partial representation of the scene (here triangle information in screen space) using compact triangle storage[1, 2] and a ray-marching approach to find a better approximation of the occlusion factor.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Motivation

Shadows play a key role in perceiving the world around us. Without shadows, objects look like they are dull and flat. They give us a visual representation of how objects relate to one another in the space. In order to simulate anything in the virtual world, we need to compute it, and the same is for shadows. Shadows can be primarily divided into two types, large scale and small scale or soft shadows. For example, shadows that are present at room corners, at the focal point of convex surfaces, between clock and wall comes under soft shadows, and shadows that cover over a certain area comes under large-scale shadows. Ambient occlusion is a lighting model used to approximate soft shadows in the virtual world (Figure1.1).

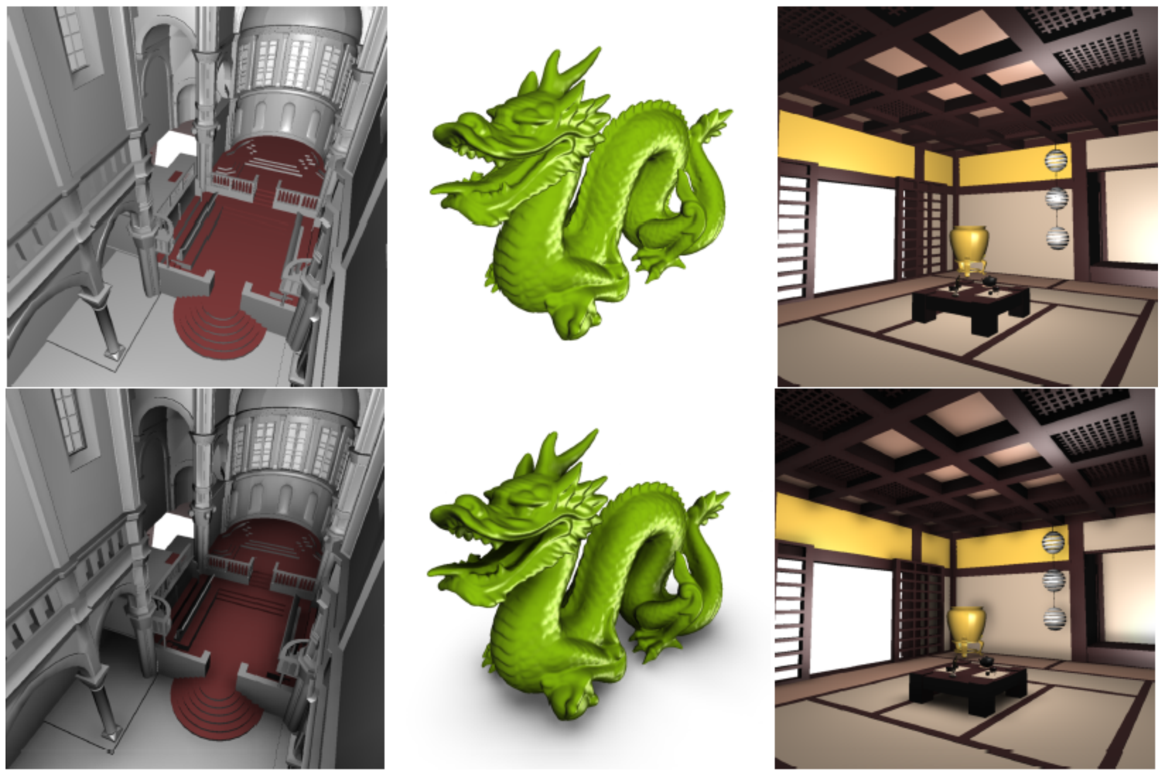


Figure1.1: The top row shows images rendered with phong shading alone and bottom row with ambient occlusion[3].

1.2 Background

In this section, we provide a brief theory and mathematical description of the rendering equation and ambient occlusion and will present some common terms that will make the basis for later discussion.

1.2.1 Rendering equation and Ambient Occlusion (AO)

Radiometry is the underlying concept behind the rendering equation[4] which is described as a flow of radiant energy(here light energy) in space. Many assumptions are made while describing the model such as light being considered as a particle, un-polarized, no interference, etc. For this discussion, it suffices to consider light as particles. The following list is a quick reference and not a detailed explanation of the definitions that occur in the later part.

Solid angle (*ω*): In 2D, angle is considered as a section of unit circle and size of the angle is the length of an arc. In 3D, solid angle is a section on the unit sphere and size is the area for a set of directions.

Power or Flux (*Φ*): Each light particle contains a certain amount of energy (*Q*). The Number of particles crossing per unit time (*t*) is considered as flux.

Irradiance (*E*): It is the flux per unit area (*A*) i.e. a number of particles hitting a surface per unit time.

Radiance (*L*): It tells us how bright a surface is or measurement of intensity in a particular direction at a particular point(x) on the surface. Formally, it is the flux per unit area per solid angle and defined as *L*(x, *ω*). Usually the incoming radiance is denoted as L_i and the outgoing radiance is considered as L_o.

$$L(x,\omega) = \frac{d\phi}{dA(x) \cdot cos(\theta) \cdot d\omega} \tag{1.1}$$

The Bidirectional reflectance distribution function (BRDF) defines the ratio of incident irradiance along an incoming direction ω_i to outgoing radiance along *ω* at point x. It is denoted as *f*.

Physically based lighting models are computationally difficult to simulate in real time applications so they are approximated. There are many popular models that approximate, such as the Cook Torrance[5] model which splits the light into different components such as ambient, diffuse and specular. The light that comes directly from a source is diffuse and specular (direct light) and all the other light that has gone reflections is considered as ambient light (indirect light). From this, we can see that the ambient term we want is the L_o and the equation to calculate this is given in [4] as follows:

$$L_o(x,\omega) = \int_\Omega f \cdot L_i(x,\omega) \cdot (n \cdot \omega) \cdot d\omega \tag{1.2}$$

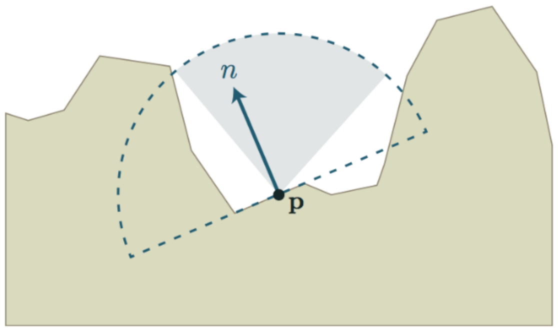


Figure 1.2: Unoccluded region over the hemisphere at a point [6].

here is, the integral all over the directions in the unoccluded region of the hemisphere at a particular point x defined along the direction of normal *n* (Figure 1.2). For this integral a few assumptions[5] are made to make it a feasible quantity. The assumptions are incoming radiance is constant and surfaces are Lambertian[7]. From these assumptions f and L_i are constant. Now the integral reduces to the visible part of the hemisphere above at point x of the hemisphere.

$$A(x) = \frac{1}{\pi} \int_{\Omega} (n \cdot \omega) \cdot d\omega \tag{1.3}$$

A mathematical definition is used in order to evaluate the integral. Consider a ray shot from the point x in the direction of ω and the visibility function[8] V(x, ω) gives the result 0 if the ray hits anything and 1 if it misses. Now the same A(x) can be found using:

$$A(x) = \frac{1}{\pi} \int_{\Omega} V(x, \omega) \cdot (n \cdot \omega) \cdot d\omega \tag{1.4}$$

This integral can be easy evaluated using the Monte Carlo integration[8] by shooting rays in different directions ω.

$$A(x) = \frac{1}{N} \sum_{n=1}^{N} V(x, \omega_n) \cdot (n \cdot \omega_n) \tag{1.5}$$

The final result, A(x) value will be in the range of [0-1], 0 means no occlusion and 1 means full occlusion. We invert this value and multiply it to the final color. This makes the areas darker when there is more occlusion and brighter when there is less occlusion.

CHAPTER 2

LITERATURE SURVEY

AO was introduced to rendering community in real-time applications by Landis[8]. Most of the earlier algorithms for AO were limited to ray-tracers, due to the reason that equation (1.5) can be easily implemented using ray-casting approach. However, in real time applications, ray-tracing is not the ultimate solution because it requires the entire scene representation to get the work done. So the solution was to use rasterization, which requires only a part of the scene at a time. A significant amount of research had been done to bring AO to real-time applications with interactive frame rates. We can categorize the number of methods in two ways: screen space (using depth buffer as scene representation) or geometry-based (using actual geometry) AO. As a topic of the discussion is screen-space, we will just outline a few of the geometry based methods.

2.1 Geometry-based AO

Dynamic AO [9] was one of the first papers to introduce AO to rasterization pipeline. This method works by treating vertices as surface elements (Figure 2.1) that can emit or reflect light. For each element, it computes an oriented disk with position, normal and area (precomputed using triangles sharing vertices and taking an average of them). It also preprocesses hierarchal lists based on distance of objects in the scene and efficiently traverses using them at runtime. The main drawbacks of the method are, it entirely depends the on geometric complexity and large preprocessing times. Some methods for example [10] use both object and image in multiple passes, first sampling occlusion by a familiar technique called voxelization and then filtering for AO. Advantages with geometry-based approaches are, they have similar quality when compared to ray-tracing. Most of the geometry-based approaches face problems such as consuming large amounts of time for precomputation, storing more memory or limited availability for movement in scenes.

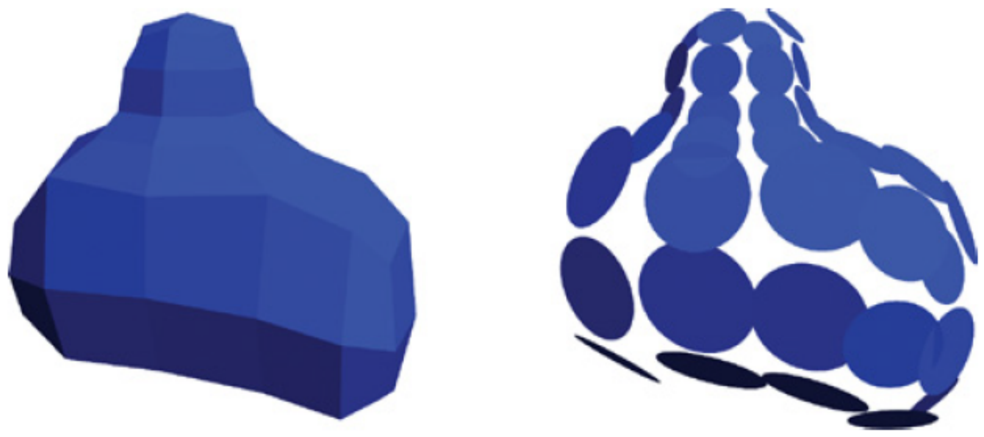


Figure 2.1: Converting of a mesh to surface elements [9].

2.2 Screen-space AO

2.2.1 Depth buffer approaches

In the CryEngine 2 [11] the first SSAO technique was introduced. It takes depth buffer as a scene approximation, as seen from the camera. It uses a deferred rendering pipeline, in the first pass it acquires depth, position (the position can also be reconstructed using depth) and normal information. Next, if uniformly distributes a kernel of N 3D-vectors around the current pixel location, and uses them to sample the depth buffer. This is done by first projecting the random vectors into the depth buffer and comparing the sampled value to the current pixel value using linear depth comparison. This is done for all the N vectors by performing screen space integration using our equation (1.5) and taking the ratio of occluded to un-occluded samples (Figure 2.2). The visibility function in the equation is taken as 1 if sampled depth is greater than current depth value and 0 otherwise. The quality clearly depends on the number of samples taken and with a small fewer samples, banding is introduced. To overcome this banding problem noise (usually a random texture) is introduced, which gives a more visually appealing result. The noise can be removed by using an extra blur pass (for e.g. Gaussian blur).

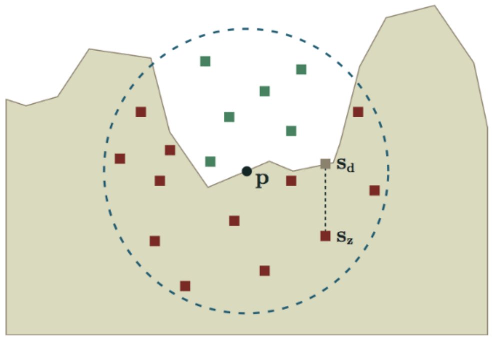


Figure 2.2: Samples projected onto depth field. Red samples are occluded and green are unoccluded [6].

The above method especially produces a different look due to the random sample points distribution on the sphere. Due to this planar surfaces always have half of the samples inside the geometry. A slightly different version [12] is built on top of it. This is addressed by taking a hemisphere oriented in direction of normal (Figure 2.3). Self-occlusion problems are also addressed by taking into account only those samples, which line in the radius of the hemisphere. The author also considers using an attenuation function to avoid over-occlusion with foreground objects.

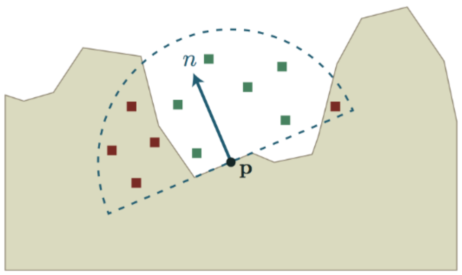


Figure 2.3: Samples taken over the hemisphere oriented along normal n [6].

Similar to the CryEngine 2 SSAO a new method is introduced by [13] which shares common process from distribution of sample points on hemisphere to completion of the first pass. Instead of taking the visibility function into account, the paper introduces a new technique called spherical proxies. After projecting the sample point in the depth buffer, a sphere is constructed around it. Then using that sphere it approximates how much each sample cuts the hemisphere at the current location. All of the construction is done in local coordinate system rather than screen space.

Several extensions [14, 15] are proposed on top of SSAO to improve quality and performance. The former one uses multi resolution depth buffer with enlarged view and sample using them. The latter takes a temporal approach by taking into account previously rendered frames and thereby using them to calculate occlusion factor.

2.2.2 Angle-based approaches

The aforementioned methods use depth checks to calculate the visibility, later Angle-based SSAO was introduced by Epic Games [16]. Instead of using a depth buffer it uses a position buffer and a sample kernel of N symmetrical vector pairs distributed around boundaries of the circle (Figure 2.4). The symmetrical vector pairs are then positioned at around our current location. The angle between each sample vector pair with respect to the eye is calculated. This is done for each whole sample kernel and compared with the previous angles. Later, AO is estimated as mean angle between the pairs:

$$AO \approx 1.0 - \frac{\theta_l + \theta_r}{\Pi} \tag{2.1}$$

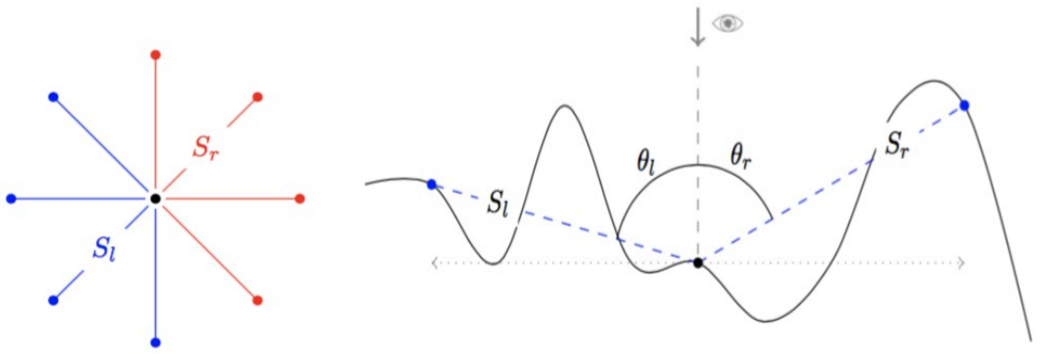


Figure 2.4: On the left sample kernel of N symmetrical vector pairs distributed along the edge of circle. Right is the 2D illustration of symmetrical vectors projected along height field[17].

We can clearly infer that this method has similarities to the familiar technique called parallax occlusion mapping with a ray-marching approach.

Another popular and most widely used approach in the industry was HBAO [18], which is introduced by NVIDIA. It can also be added to already existing games with an NVIDIA graphics driver. The main goal is to find an angle of the visible horizon. This approach primarily makes two assumptions, height field is continuous and other is, rays that cast beyond the horizon angle must be occluded. The paper also uses randomly distributed kernel of vectors along the edge of a circle but the only difference is, they not symmetrical pairs (Figure 2.5). For each random vector, a ray-marching approach is employed in the direction of a random vector and 3D positions are sampled at several points. At every point, angles are calculated and largest of those angles are stored as horizon angle h(θ). Ambient occlusion is approximated as difference between the horizon angle h(θ) and tangent angle t(θ):

$$AO \approx h(\theta) - t(\theta) \tag{2.2}$$

In a more practical way this can be done as follows:

$$AO \approx 1.0 - max(S \cdot N, 0.0) \tag{2.3}$$

Where S is the random vector and N is normal to the current location. The same authors come up with a slight modification to the algorithm called Horizon-Split ambient occlusion (HSAO) [19], by adding few extra computation and increasing the quality and performance. Another paper which is fundamentally similar to HBAO is Alchemy Ambient Occlusion[20] which is developed by Alchemy engine. Currently, it is the most popular and used in a number of games. The only difference is to use a single position buffer sample and a different weighting function:

$$\rho(d) = \frac{u \cdot d}{max(u \cdot d)^2} \tag{2.4}$$

where d is the distance from the random sample point to the current location and other is user parameter for choosing the radius. In terms of quality and performance this is the superior method [6] when compared to all other previously stated approaches.

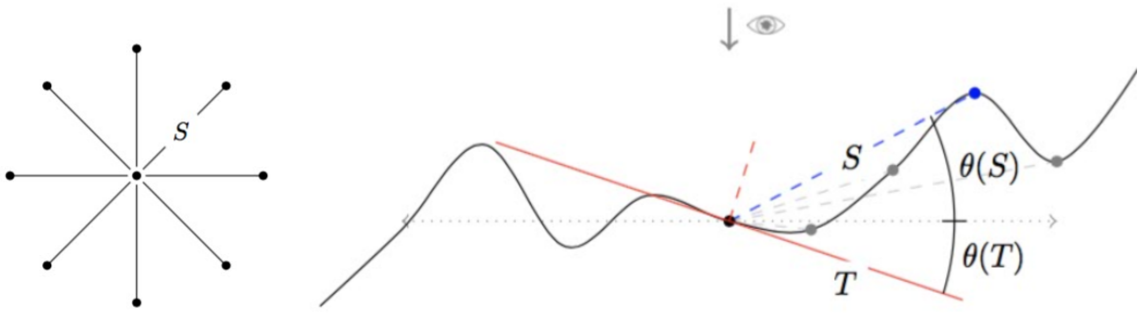


Figure 2.5: On the left, sample kernel of N vector pairs distributed along edge of circle. To the right samples projected (gray rejected and blue accepted) onto height field. S and T are the horizon and tangent angles [17].

2.3 Sub-Pixel Shadow Mapping (SPSM)

All of the methods mentioned up to now use either depth or position buffer as approximate scene representation but not actual geometry into consideration. The new method we present is heavily inspired and also a future work specified in paper Sub-Pixel Shadow Mapping[1, 2], which deals with artifacts that comes with standard shadow mapping[21]. The main problem in the shadow maps is a limited resolution of the depth buffer. A Common solution for this one is to increase the resolution, which has an impact on memory consumption. The authors address the issue in a very clever way by storing the partial representation of the scene geometry by using Conservative rasterization [22] and then again reconstructing it back while rendering. This gives almost a similar result to the ray-tracing approach along with sub-pixel precision (see Figure 2.6 for details) at interactive framerates. It also does not require any precomputation or preprocessing.

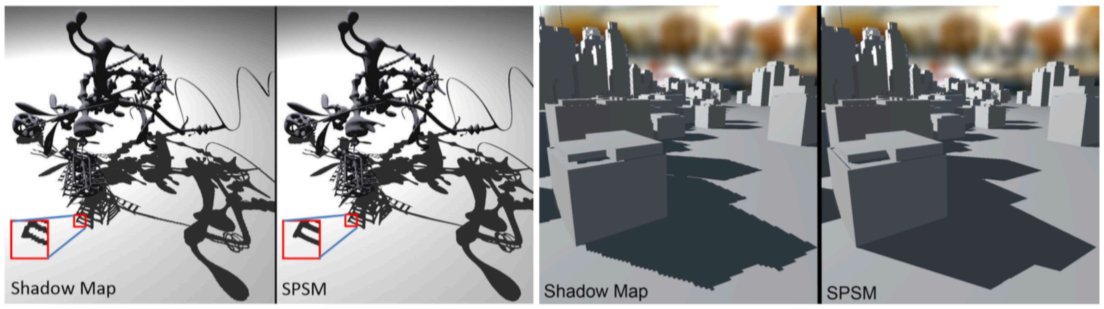


Figure 2.6: Compares the traditional Shadow mapping to the Sub-pixel Shadow Mapping.[1]

CHAPTER 3

METHODOLOGICAL DESIGN AND IMPLEMENTATION

3.1 Overview

As previously stated, our topic is inspired from SPSM [1]. Our methodology is outlined as follows:

• In the first pass save the compact triangle storage using SPSM.

• In the second pass vertex shader, construct Tangent Bi-tangent Normal matrix and output it.

• Convert every randomly generated vector from tangent space to world space using the TBN matrix.

• Using world space position and the world space random vectors, find the new random positions and project them to find the texture coordinates.

• Fetch the triangle information at the texture coordinates.

• Shoot the previous random vector from the world position and check whether it intersects the triangle or not.

• If it intersects, then there is an object occluding it, otherwise not.

The advantages of the current method are, as it uses the actual geometry the calculated occlusion value is much an exact value rather than some general approximations. Unlike HBAO or SSAO, it does not have limitations such as height field is continuous etc.

3.2 Algorithm

All the terminology we use is from the OpenGL API. Refer to OpenGL API for more details. We also use a deferred rendering graphics pipeline for the approach, which is deferring the shading of geometry and storing intermediate output computations to buffers. The sections below are divided into two, one for each pass in the rendering pipeline.

3.2.1 First Pass

❖ First pass pipeline:

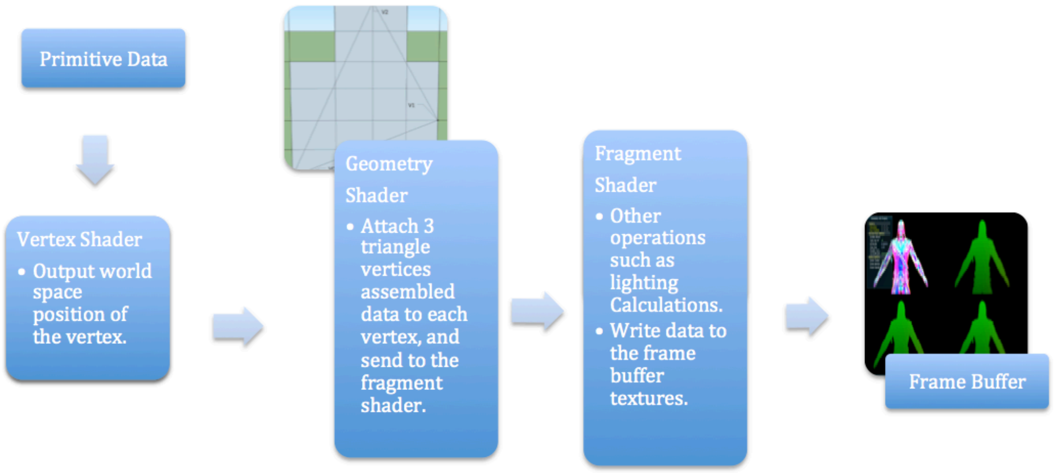

Primitive Data
Vertex Shader
• Output world space position of the vertex.
Geometry Shader
• Attach 3 triangle vertices assembled data to each vertex, and send to the fragment shader.
Fragment Shader
• Other operations such as lighting Calculations.
• Write data to the frame buffer textures.
Frame Buffer

Figure 3.1: First pass pipeline.

In the vertex shader, we convert the object positions to world space and send them to the geometry shader. In the geometry shader, we have the world space positions of the three vertices of the current rendered triangle. Now for each vertex, we send the three vertices data to the rasterization stage and then to fragment shader. In the fragment shader for each pixel, we write these three triangle vertices to three textures (See Figure 3.2). We use three 96-bit floating-point textures for rendering it in high quality. Along with it, we can render the lighting calculations for additional purposes. Each 96-bit floating-point texture takes around 10-12 MB on a 1280 x 720 resolution. For three textures, it is 30-35 MB of storage. Even though the current hardware can process a lot of information, reading and writing back is still a costly memory operation. Instead, we can store all the nine floating-point values (three per each triangle) in a single 96-bit unsigned integer texture. To do this, we convert every vertex from world-space to normalized device coordinate system, which is in the range of [0-1]. Then we divide the 96-bit unsigned integer into 3 equal parts (32 bits for each vertex). Then, each 32 bit is divided into three equal parts each of k-bits (See Figure 3.3). Let R be the 32-bit value in which we store the final compressed result, M be number of bits to be shifted to store a value at correct location in R, V_x be X-axis coordinate value for a vertex in NDC space. We scale V_x value from [0-1] range to previously specified k-bit range i.e. [0-2^k]. After scaling, we place the V_x in the specified location in R by first left shifting V_x with M-bits and masking it with R (see equation 3.1). This nearly increases the storage efficiency by 60% and the main advantage is, that we can have all the data we want in a single texture lookup. While retrieving the data, we fetch each value by bit masking and then un-projecting it from NDC space to world space. We have three bits of precision for the un-projected world space floating point value. As we have less precision, some of the ray-triangle intersections will fail in the second pass, so the quality of the final output will be less. The images Figure 3.4 and Figure 3.5 show the rendered textures in detail.

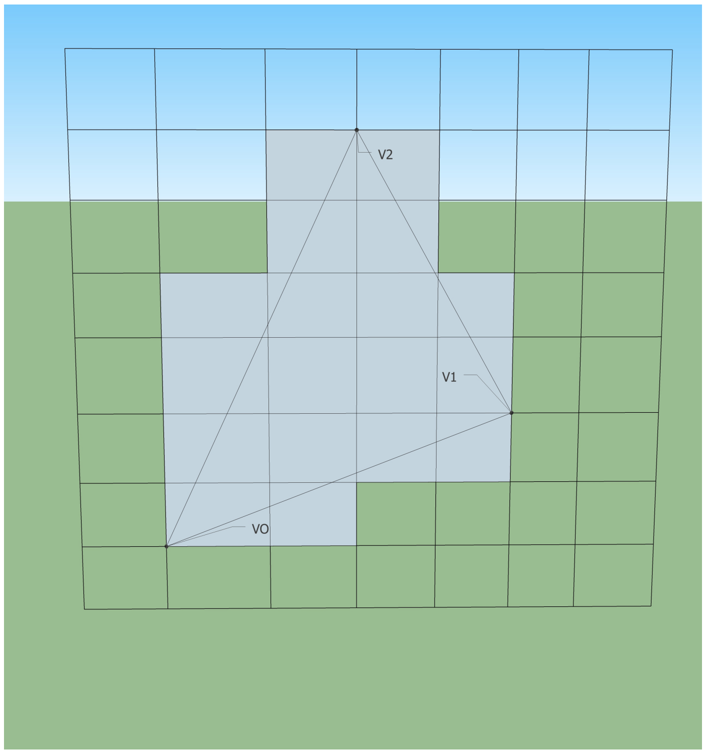
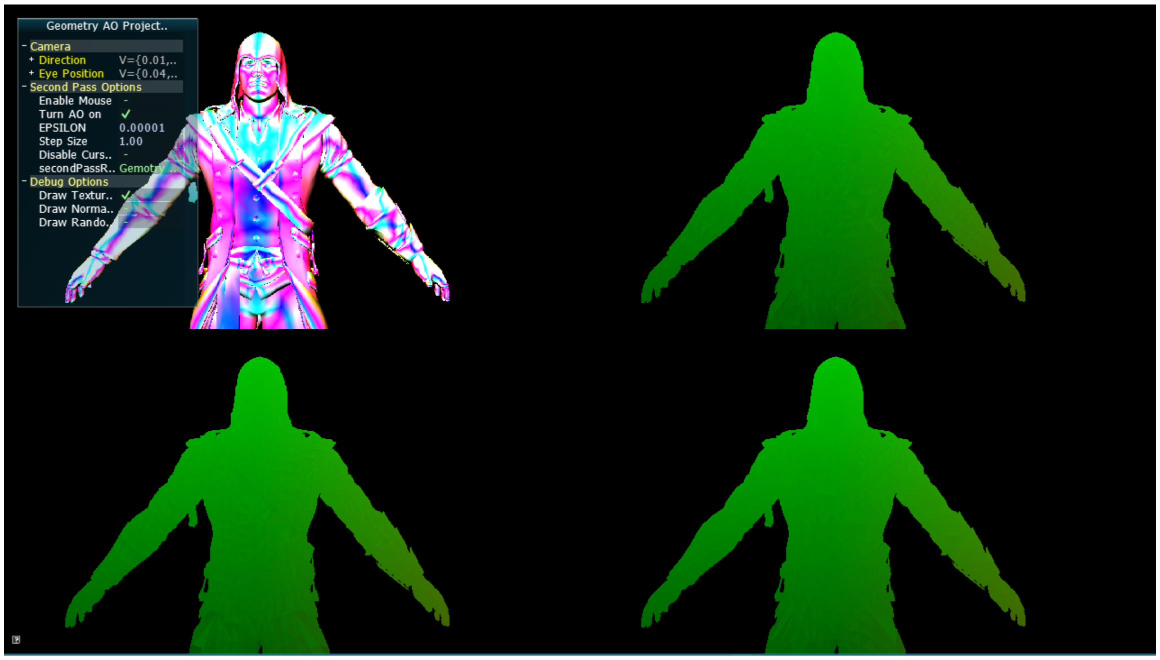


Figure 3.2: For each pixel, all the three vertices of triangle are stored.

$$R\& = ((V_x * 2^k) << M) \tag{3.1}$$

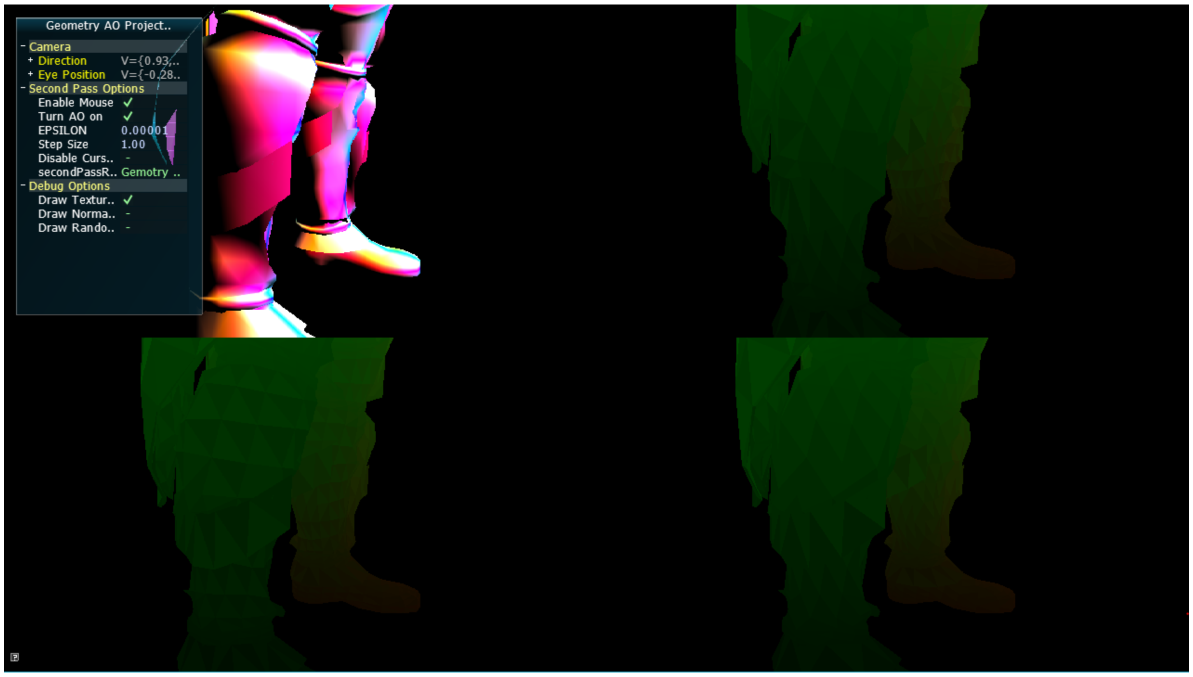
1	0	1	1	1	1	1	1	0	0	0	0	0	1	1	0	0	1	0	1	0	1	0	0	1	0	0	1	0	0	0	1
23-31 bits (v_x)											10-22 bits (v_y)											0-9 bits (v_z)									

Figure 3.3: Packing of three 32-bit floating-point values in NDC space [0-1], to a single unsigned integer.



```
Geometry AO Project..
- Camera
  + Direction        V={0.01,..
  + Eye Position     V={0.04,..
- Second Pass Options
    Enable Mouse     -
    Turn AO on       ✓
    EPSILON          0.00001
    Step Size        1.00
    Disable Curs..   -
    secondPassR..    Gemotry ..
- Debug Options
    Draw Textur..    ✓
    Draw Norma..     -
    Draw Rando..     -
```

Figure 3.4: Each triangle vertex stored in a different texture. Starting counter-clockwise from Vertex0(bottom left), Vertex1(bottom right), Vertex2(top right), Additional lighting calculation texture (top left).

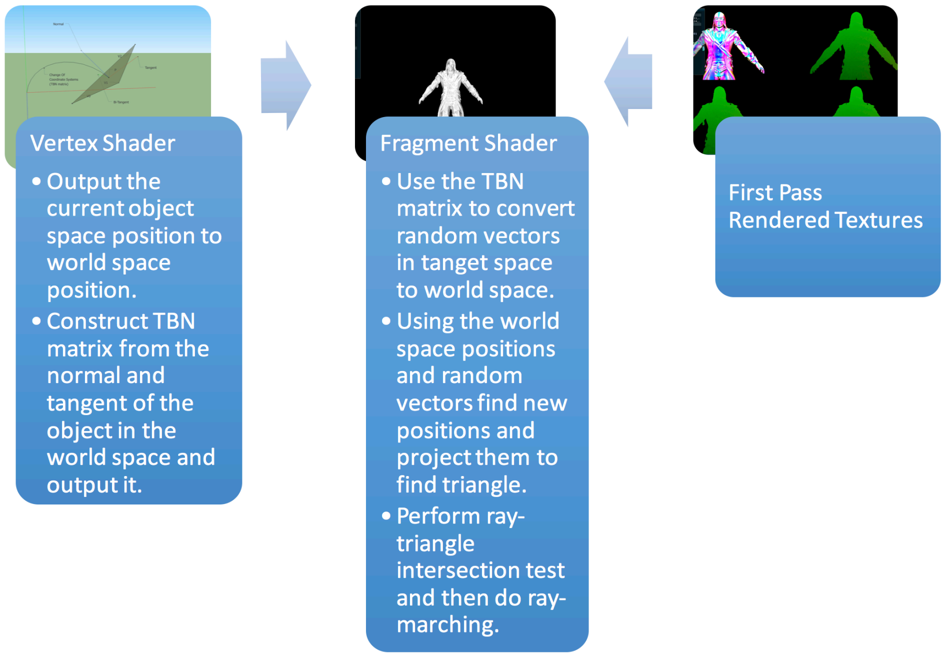


```
Geometry AO Project..
- Camera
  + Direction        V={0.93,..
  + Eye Position     V={-0.28..
- Second Pass Options
    Enable Mouse     ✓
    Turn AO on       ✓
    EPSILON          0.00001
    Step Size        1.00
    Disable Curs..   -
    secondPassR..    Gemotry ..
- Debug Options
    Draw Textur..    ✓
    Draw Norma..     -
    Draw Rando..     -
```

Figure 3.5: Image zoomed in to show detail information.

3.2.2 Second Pass

❖ Second pass pipeline


Vertex Shader

- Output the current object space position to world space position.
- Construct TBN matrix from the normal and tangent of the object in the world space and output it.

Fragment Shader

- Use the TBN matrix to convert random vectors in tanget space to world space.
- Using the world space positions and random vectors find new positions and project them to find triangle.
- Perform ray-triangle intersection test and then do ray-marching.

First Pass Rendered Textures

Figure 3.6: Second pass pipeline.

We generate a random vectors kernel around the hemisphere and send it to the fragment shader. The generated vectors are in tangent space, so we need to convert them to world space before using them. In the vertex shader, we input the tangent and normal of the object in world space and construct a matrix to convert them back to world space. This matrix is called Tangent, Bi-tangent, Normal matrix (TBN). One thing to observe is, in the world space the Up vector is taken as Y-axis and in the tangent-space, the Z-axis is taken as Up vector. Also, the Z-axis is represented every time with the normal of the triangle. This helps us to always shoot the primary rays (random vectors) above the triangle only. Figure 3.7 clearly depicts the local tangent space coordinate system with the tangent, bi-tangent and normal.

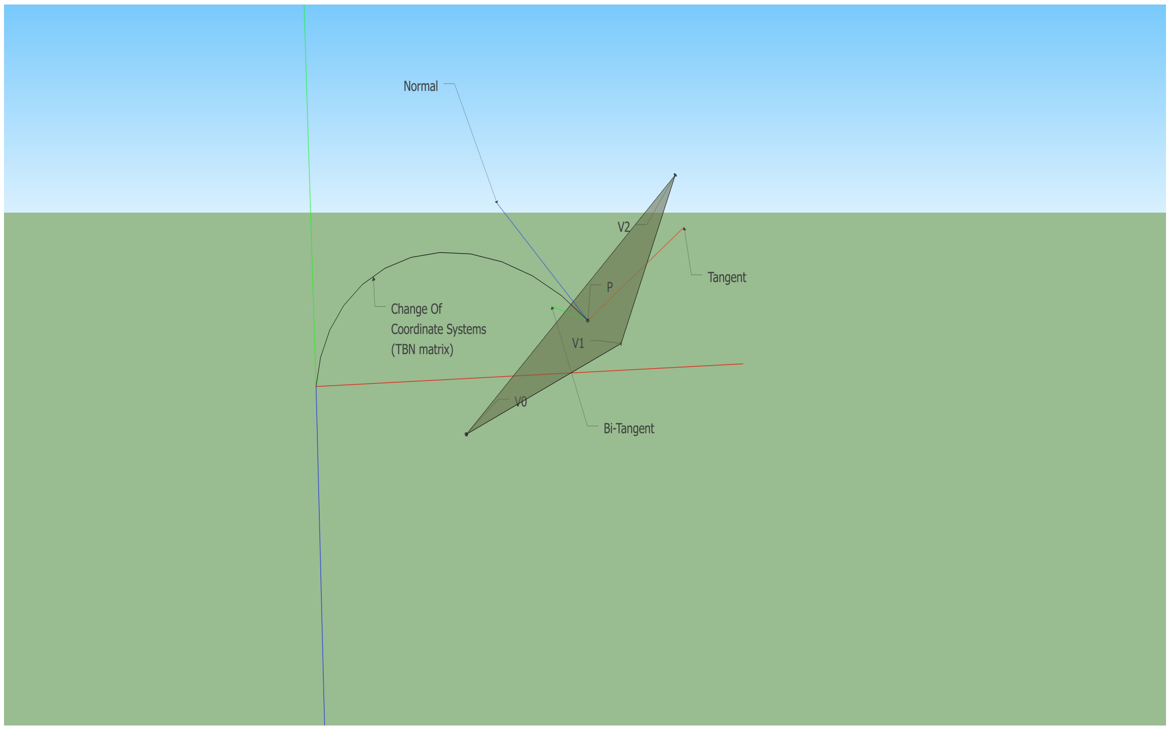


Figure 3.7: Local coordinate system which is constructed with Tangent(red), Bi-tangent(green) and Normal(blue). Note the normal is facing up and it is a left-handed coordinate system.

$$\begin{bmatrix} Tx & Bx & Nx & Px \\ Ty & By & Ny & Py \\ Tz & Bz & Nz & Pz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 3.8: TBN matrix to go from tangent to world space.

In the fragment shader, all the random vectors are converted to world space using the TBN matrix. We use the current world space position and the vector to find a new point in the world space. Then we project the point and fetch the texture coordinates from it and get the triangle information from the textures we previously stored. Using the triangle vertices and the vector direction we check whether the ray in that direction hits the triangle or not. The ray-triangle intersection, which we implemented, is geometric based. It first checks whether the ray intersects the plane or not and the gets the point on the plane. Let A, B, C be any points on the plane and x, y, z be the normal of the plane, then the plane equation is:

$$A * x + B * y + C * z + D = 0 \tag{3.2}$$

Any point lying on the plane must satisfy this equation, the parameter D is the distance from the origin. To find it, we can substitute one of our three vertices of the fetched triangle and normal in it. The equation becomes:

$$D = -(A * x + B * y + C * z) \tag{3.3}$$

Or in vector form

$$D = -(\vec{n} \cdot \vec{p}) \tag{3.4}$$

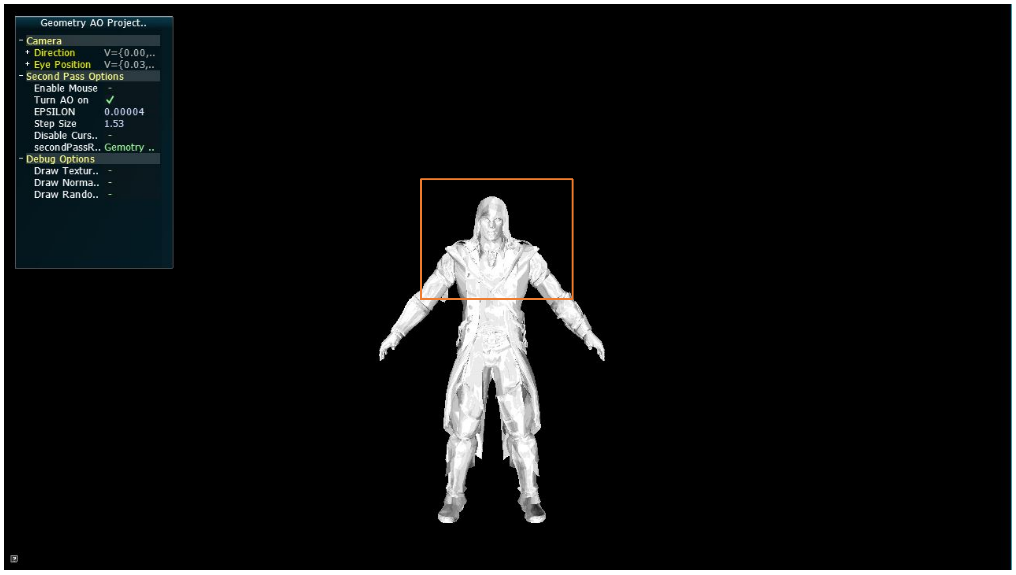
Let Origin 'O' is the current point in the world space and 'v' be our random vector. Then the new point that intersects the plane becomes:

$$P = O + t * \vec{v} \tag{3.5}$$

where 't' is the parameter which shifts the point in the ray direction. If the point is on the plane, then substituting the point in the plane equation must satisfy:

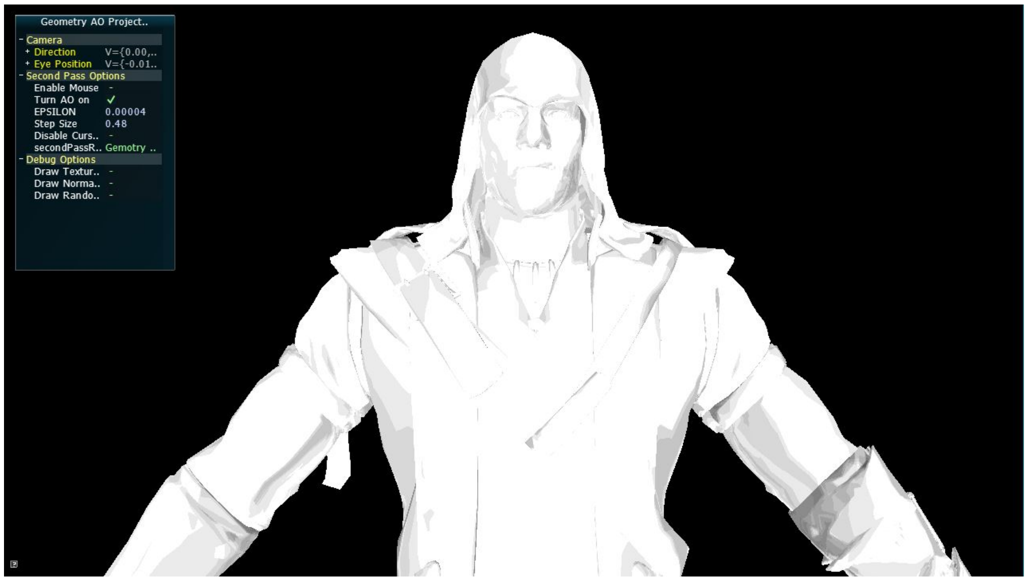
$$D = -\vec{n} \cdot (O + t * \vec{v}) \tag{3.6}$$

By solving this, we get the final ‘t’ parameter, which can be substituted in the equation 3.4 to get the final point on the plane. Note that we need to check when our vector and plane are in parallel. This can be done by checking the dot product between the normal and vector. If the dot product is zero, then they are parallel. Now we need to check whether the point on the plane is on the triangle or not, this check is called inside-out test. For a given edge in the triangle, we can check whether the point is inside or not. To do this we compute the cross product between the vector defined by the edge and the vector defined by the point and start vertex. Then this vector should point above the triangle. To check this, we dot product this vector and normal and see whether the product is greater than zero or not. If the random vector and triangle intersect we can take in that some object is occluding our current point in the scene. Similarly, we calculate for all the random vectors and find the occlusion factor.



```
Geometry AO Project..
- Camera
  + Direction        V={0.00,..
  + Eye Position     V={0.03,..
- Second Pass Options
    Enable Mouse     -
    Turn AO on       ✓
    EPSILON          0.00004
    Step Size        1.53
    Disable Curs..   -
    secondPassR..    Gemotry ..
- Debug Options
    Draw Textur..    -
    Draw Norma..     -
    Draw Rando..     -
```

Figure 3.9: Ambient occlusion image rendered at 1024x768 with a kernel size of 15 points and with ray-marching step size of 5. Darker areas represent more ray-triangle intersections i.e. lesser occlusion factor.



```
Geometry AO Project..
- Camera
  + Direction       V={0.00,..
  + Eye Position    V={-0.01..
- Second Pass Options
    Enable Mouse    -
    Turn AO on      ✓
    EPSILON         0.00004
    Step Size       0.48
    Disable Curs..  -
    secondPassR..   Gemotry ..
- Debug Options
    Draw Textur..   -
    Draw Norma..    -
    Draw Rando..    -
```

Figure 3.10: The orange rectangle in figure 3.9 is our area of interest, which is zoomed to show in detail.

The approach works well but it has one major issue. What if the triangle at our new point is not occluding, but it has some object between? For example, in the figure 3.11, the red point is the world space position which we get from random sample. The triangle information which is there at that position is not occluding it in that ray-direction. Also, there are other objects (triangle and cube) which are in that direction not being considered. In order to fix this, we ray-march from the current position in the ray-direction for a fixed number of steps to find any occlusions. By doing this, we can find the triangle which is within the shortest distance and also this distance can be used as a fall off function. In detail, consider the point equation in 3.5, our current point denotes ‘O’, ‘t’ is our step size and ‘v’ is the random sample vector. We first find the new point with the point equation 3.5, and find the texture coordinates of that point and find whether the ray intersects the triangle at that position or not. If it hits calculate the distance from that point and uses it as fall off, otherwise increase the step size and repeat the process until an occlusion is found.

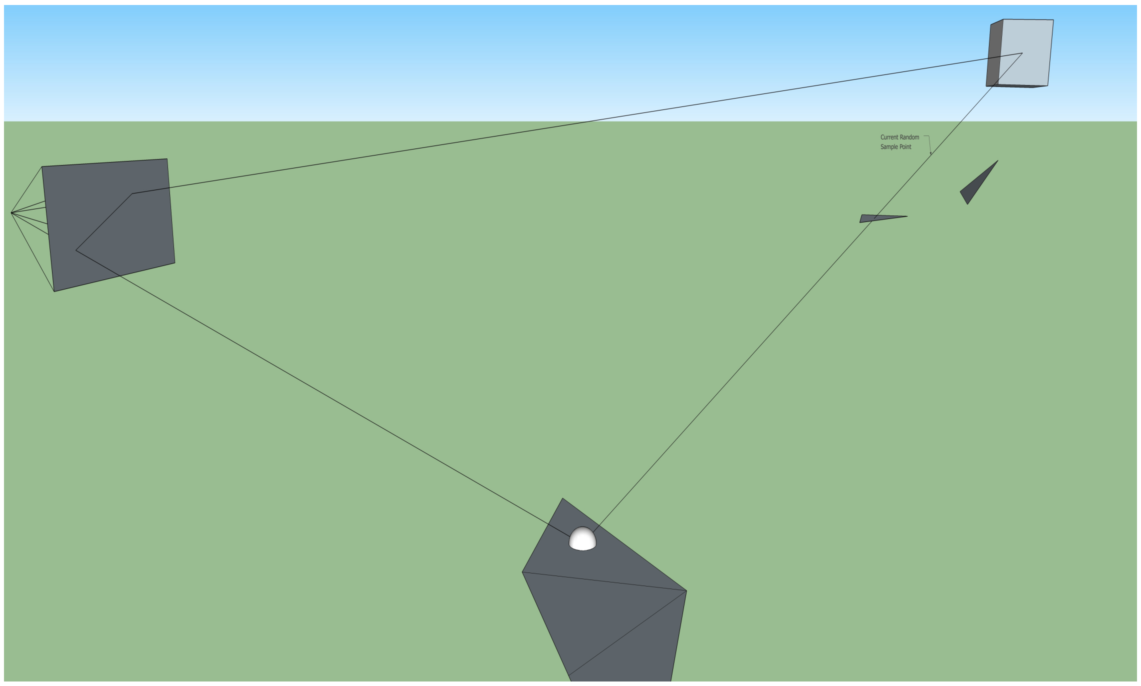


Figure 3.11: The hemisphere on the triangle strip is our current point and the black arrow is the sample position in the ray direction and the triangle data over there is not intersecting. But there are objects (triangle & cube) in the ray direction which can be found by ray-marching from our current point.

Alternatively, we can find our current position in the texture space (in OpenGL we can directly find by dividing gl_Fragcoord.xy with resolution) and the direction vector in texture space. Now we can directly traverse texel by texel in the direction of vector without projecting the new sample position every time. Figure 3.12 depicts the ray marching in texture space.

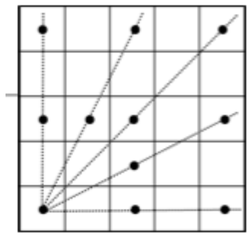
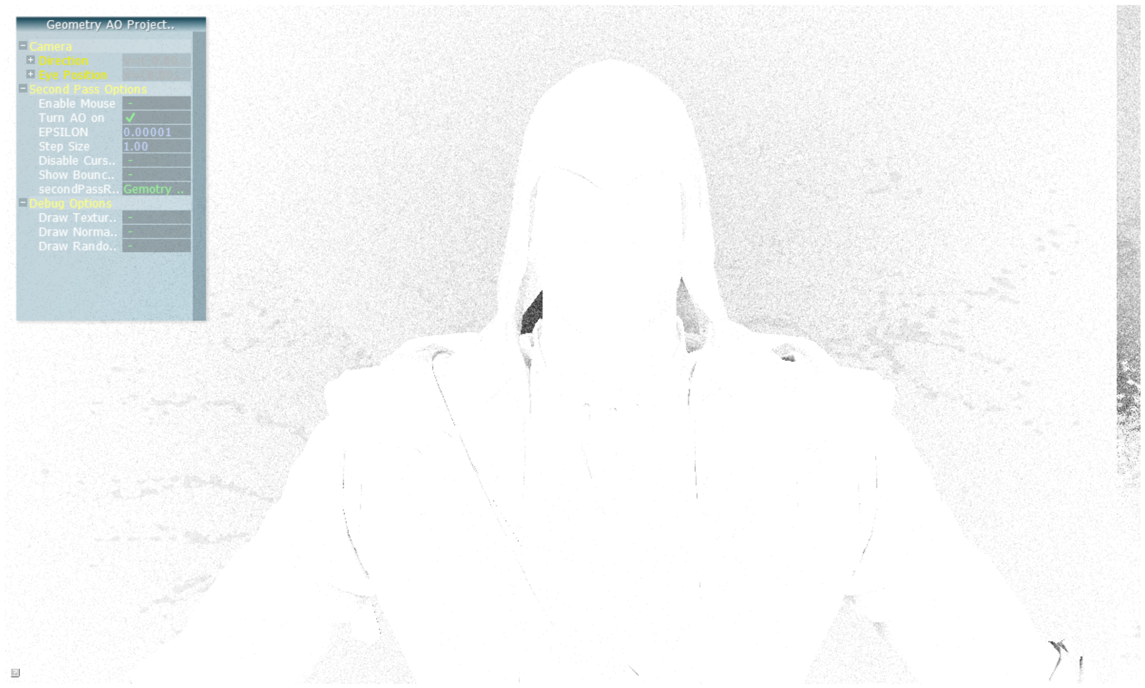


Figure 3.12: Ray-marching in texture with step size of 2 texels.

The ray-marching procedure increases quality but decreases the frame rate drastically. This is because of the cost of the ray-triangle intersection and number of texture look-ups increases with a number of steps. For example, if the step size is 10 and kernel size is 10, in the worst case we have to check 100 ray-triangle intersections and texture lookups for each pixel Figures 3.13 and 3.14 shows the quality of the ambient occlusion with and without ray marching. It clearly shows there is a 50-60% drop in the frame rate.



Geometry AO Project..

Camera	
Direction	
Eye Position	
Second Pass Options	
Enable Mouse	-
Turn AO on	✓
EPSILON	0.00001
Step Size	1.00
Disable Curs..	-
Show Bounc..	-
secondPassR..	Gemotry ..
Debug Options	
Draw Textur..	-
Draw Norma..	-
Draw Rando..	-

Figure 3.13: Without ray marching with a kernel size of 10 and rendered at 50fps.

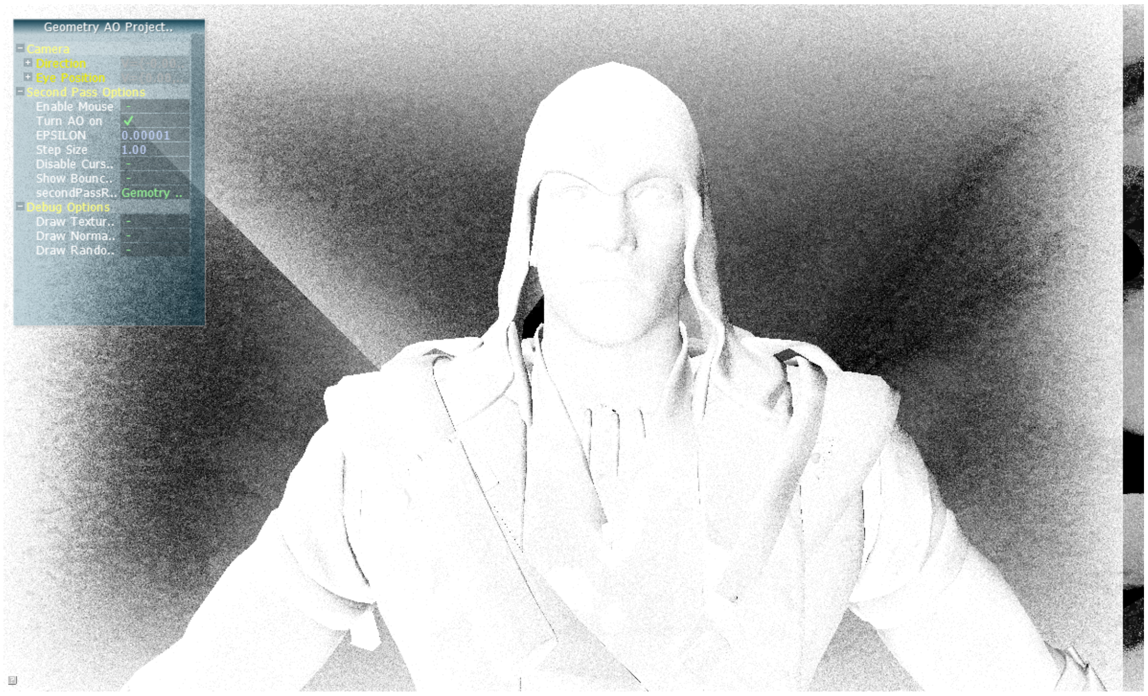

```
Geometry AO Project..
- Camera
  + Direction
  + Eye Position
- Second Pass Options
    Enable Mouse     -
    Turn AO on       ✓
    EPSILON          0.00001
    Step Size        1.00
    Disable Curs..   -
    Show Bounc..     -
    secondPassR..    Gemotry ..
- Debug Options
    Draw Textur..    -
    Draw Norma..     -
    Draw Rando..     -
```

Figure 3.14: With ray-marching with a kernel size of 10 and rendered at 18-22fps.

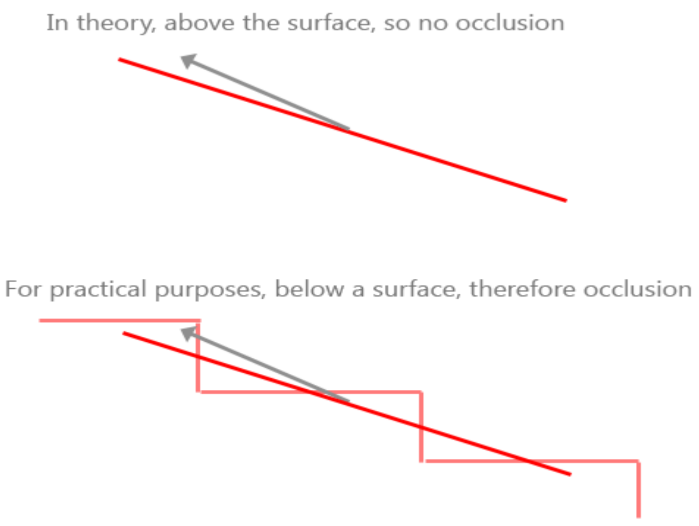

In theory, above the surface, so no occlusion
For practical purposes, below a surface, therefore occlusion

Figure 3.15: Depth resolution artifacts.

3.3 Common Observations

Most of the SSAO algorithms face two main artifacts. One is due to depth resolution and other is due to the size of the hemisphere. Current depth buffers use 24 or 32 bits to store depth and values are quantized. Also, the precision depends on whether the object is nearer or farther from the viewer. Problems occur when the random sample lies closer to the surface, because of limited resolution and quantization we can have false occlusion (Figure 3.15). Since we are not using the depth buffer for approximation, this is avoided. The other artifact is also called AO radius artifact, which produces unrealistic darkening for some objects. In practice, if the sample is far away from the current point a falloff function is used as a workaround. This is to limit for not checking sample with an unrelated object, which is away in the scene, usually done by taking the radius of hemisphere into account. We use ray-marching technique to check the first intersection with triangle geometry, which avoids this issue. Another problem is aliasing, that comes with using of fixed random sample kernel. This can be easily resolved by using noise. We used 2D Perlin noise to solve this, alternatively, we can use noise textures and repeat all over the screen.

3.4 Limitations

The major limitation of our approach is when the geometry rendered is far away and the resulting output of the first pass textures renders more triangles geometry to closer texels. In these cases, the ray-triangle intersection passes for many points and the output is darker than it should be. Also, the generation of random sample kernel vectors should be evenly distributed over the hemisphere, otherwise it will discard occlusion in some directions.

CHAPTER 4

RESULTS AND DISCUSSION

All of our tests are run on Intel i5 PC, equipped with 16GB RAM and an NVidia GeForce 940M graphics card. Our first test demonstrates performance vs quality trade off at different kernel sample sizes. From the graph 4.1, we can see that with 5-10 samples the image can be rendered at 40-25 FPS with decent quality. The rendered images can be seen in the top row of figure 4.1.

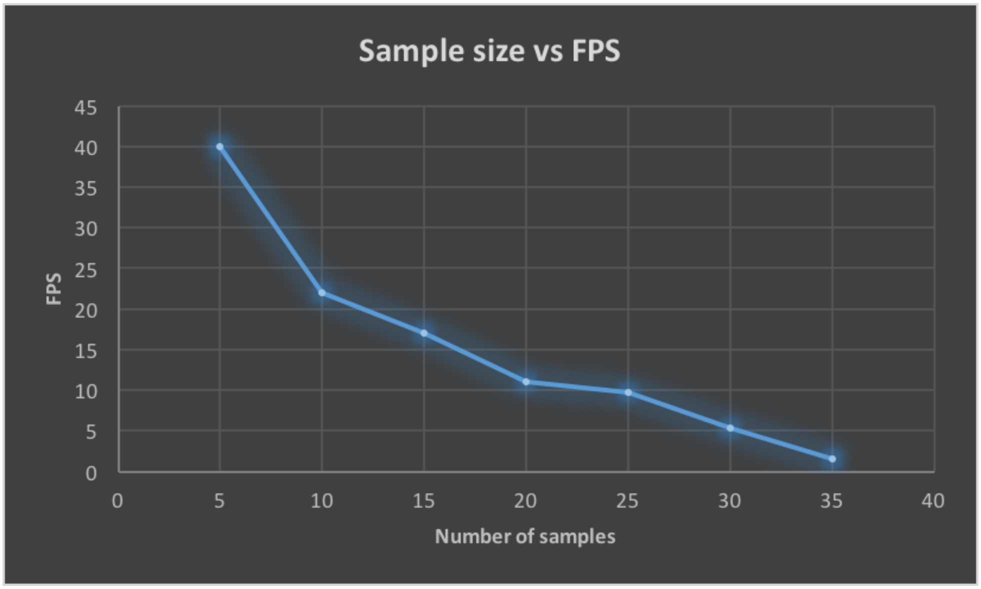


Figure 4.1: The graph demonstrates the frame rate decrease as the sample size increases. The textures are rendered at 1024x768 resolution and 10 steps are used for ray-marching.

As the sample size increases the noise in the images decreases. There is not much a difference in images rendered with 5 samples versus 20 samples, rather than image get smoothed and increased accuracy. Instead, we can render with less number of samples and use a blur filter to get a smoother output.

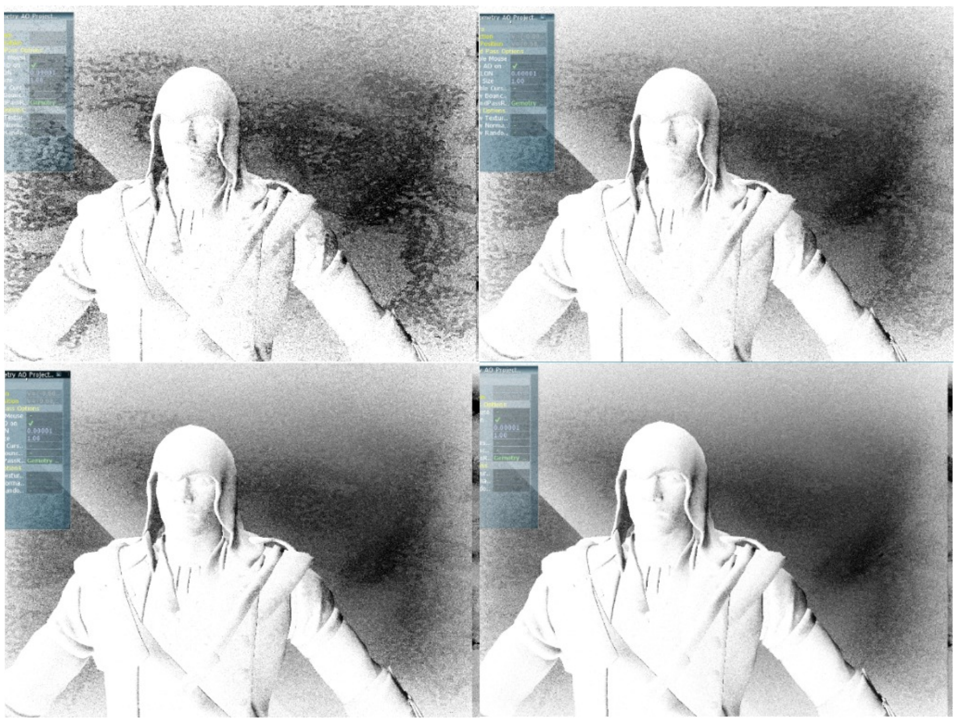


Figure 4.2: Each image is rendered at 1024x768 resolution with a step size of 10. Starting clock-wise, images are rendered with 5 samples(top-left), 10 samples(top-right), 15 samples(bottom-left) and 25(bottom-right).

Our second test demonstrates the size of data vs performance. Table 4.1 shows the models on which we have tested. Figure 4.2 show outputs for two of the models we have tested. We have tested with optimization being turned off i.e. rendered three different textures for each vertex instead of storing all in one texture. For large meshes, we can improve frame rate by decreasing the rendered texture size in the first pass and storing them in a single texture.

Number of triangles (5 samples)	FPS	Model Name
19058	55	Nano suit
28501	39	Altier
69451	21	Bunny Model
>100000	17	Stanford Dragon

Table 4.1: Models rendered at 1024x768 with 5 samples and 10 steps for ray-marching.

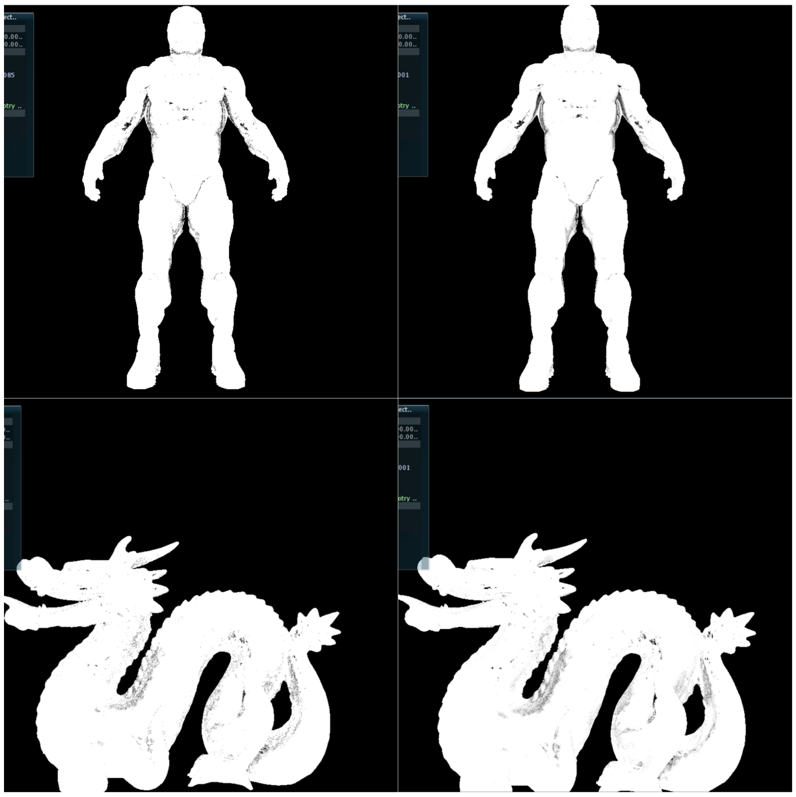


Figure 4.3: Left column shows Nano suit and Stanford Dragon models rendered at 1024x768 with 5 samples and right column with 25 samples.

The third test depends on the resolution of the image rendered in both the passes. From figure 4.3 we see that with lower resolution, we can achieve higher interactive frame rates. We can also improve performance by rendering the first pass textures at a lower resolution and then rendering the second pass at a higher resolution. The only problem will be unwanted occlusions, as many triangles will be rendered at same location.

All of our previous tests are done on NVidia GeForce 940M, which is a mobile graphics card and older GPU. For the last test, we tested on one high-end mobile and one desktop GPU hardware, and there is a significant performance increase. Figure 4.4 show on NVidia GTX 760 (which is a bit older than mobile GPU we tested) can render around 52 FPS.

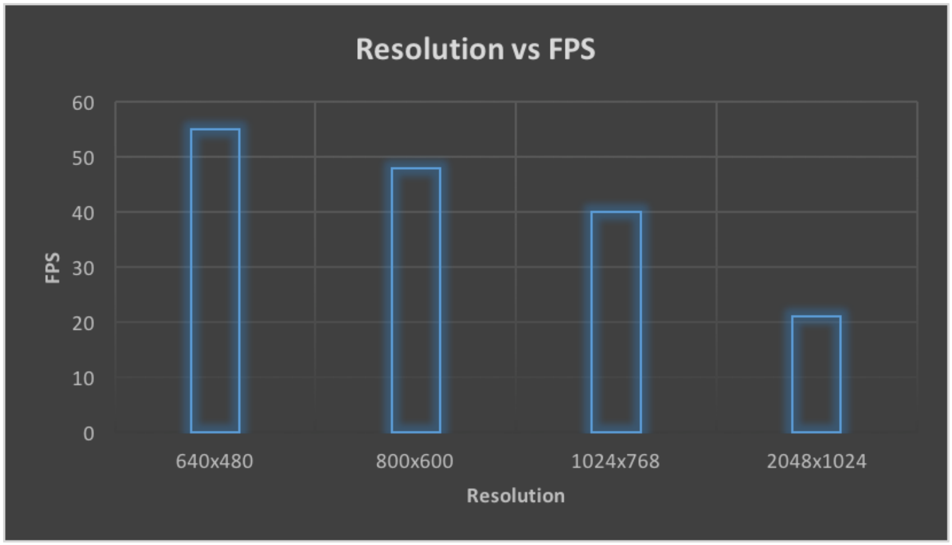


Figure 4.4: Each image is rendered with 5 samples and a step size of 10 on NVidia GeForce 940M with Altier model .

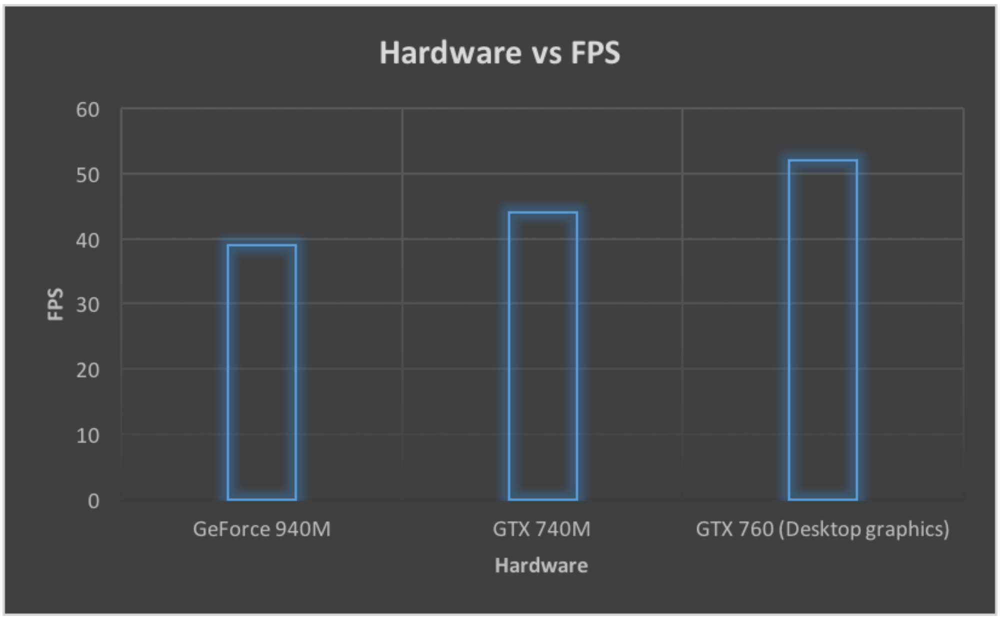


Figure 4.5: Each image is rendered at 1024x768 resolution with 5 samples and a step size of 10 with Altier model.

CHAPTER 5

CONCLUSION AND FUTURE WORK

We have presented a new ambient occlusion method based on the triangle-based geometric information. Unlike previously proposed methods, our approach gives a better approximation at interactive frame rates with no preprocessing. A major advantage of our technique is, it can be easily integrated with a single geometry map. Major artifacts, which occur due to the limited resolution of the depth buffer, can be avoided. In the future we want to investigate how to efficiently pack the triangle information using the barycentric coordinates or derivatives without losing the precision in a single texture. This will enable us to render our textures at much higher resolutions, which help us to overcome limitations based on resolutions. Another area where it would be better to investigate is to check on different ray-triangle intersection approaches and find which one suits better for the current approach. Lastly, we can limit the number of texture fetches, by improving the ray-marching technique using other searches, instead of linear search, will be a major performance improvement.

REFERENCES

1. Lecocq, P., et al. Sub-pixel shadow mapping. in *Proceedings of the 18th meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. San Francisco,March 2014. p. 103-110.

2. Dai, Q., B. Yang, and J. Feng. Reconstructable geometry shadow maps. in *Proceedings of the 2008 symposium on Interactive 3D graphics and games*. Redwood Shores,February 2008. doi: 10.1145/1342250.1357016.

3. Reinbothe, C.K., T. Boubekeur, and M. Alexa. Hybrid Ambient Occlusion. in *Eurographics (Areas Papers)*.Munich, March 2009. p. 51-57.

4. Kajiya, J.T. The rendering equation. in *ACM Siggraph Computer Graphics*. New York, February 1986. p. 143-150.

5. Cook, R.L. and K.E. Torrance, A reflectance model for computer graphics. ACM Transactions on Graphics (TOG), New York, January1982. **1**(1): p. 7-24.

6. Aalund, F.P., A Comparitive Study of Screen-Space Ambient Occlusion Methods. 2015.

7. Whitted, T. An improved illumination model for shaded display. in *ACM Siggraph 2005 Courses*. Los Angeles,July 2005. doi: 10.1145/1198555.1198743.

8. Landis, H., Production-ready global illumination. Siggraph course notes, January 2002. **16**(2002). p. 11.

9. Bunnell, M., Dynamic ambient occlusion and indirect lighting. Gpu gems, 2005. **2**(2): p. 223-233.

10. Reinbothe, C., T. Boubekeur, and M. Alexa. Hybrid ambient occlusion. in *Proceedings of the Eurographics Symposium on Rendering*. Munich, March 2009. p. 51-57.

11. Mittring, M. Finding next gen: Cryengine 2. in *ACM SIGGRAPH 2007 courses*. San Diego, August 2007. p. 97-121.

12. Filion, D. and R. McNaughton. Effects & techniques. in *ACM SIGGRAPH 2008 Games*. Los Angeles, August 2008. p. 133-164.

13. Shanmugam, P. and O. Arikan. Hardware accelerated ambient occlusion techniques on GPUs. in *Proceedings of the 2007 symposium on Interactive 3D graphics and games*. Seattle, April 2007. p. 73-80 .

14. Bavoil, L. and M. Sainz. Multi-layer dual-resolution screen-space ambient occlusion. in *SIGGRAPH 2009: Talks*. New Orleans, August 2009. doi: 10.1145/1597990.1598035.

15. Smedberg, N. and D. Wright. Rendering techniques in gears of war 2. in *Game Developer Conference*. 2009.

16. Mittring, M., The technology behind the unreal engine 4 elemental demo. part of Advances in Real-Time Rendering in 3D Graphics and Games, SIGGRAPH. Los Angeles,August 2012.

17. Hardeman, S., A Guide to SSAO. 2014.

18. Bavoil, L., M. Sainz, and R. Dimitrov. Image-space horizon-based ambient occlusion. in *ACM SIGGRAPH 2008 talks*. Los Angeles, August 2008.doi: 10.1145/1401032.1401061.

19. Dimitrov, R., L. Bavoil, and M. Sainz. Horizon-split ambient occlusion. in *Proceedings of the 2008 symposium on Interactive 3D graphics and games*. Redwood Shores,February 2008. doi:10.1145/1401032.1401062.

20. McGuire, M., et al. The alchemy screen-space ambient obscurance algorithm. in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*. Vancouver, August 2011. p. 25-32.

21. Williams, L. Casting curved shadows on curved surfaces. in *ACM Siggraph Computer Graphics*. New York, August 1978. p. 270-274.

22. Hasselgren, J., T. Akenine-Möller, and L. Ohlsson, Conservative rasterization. GPU Gems, 2005. **2**: p. 677-690.

VITA

Sai Vamsheekrishna Chaitanya Timmaraju

Candidate for the Degree of

Master of Science

Thesis: SCREEN SPACE AMBIENT OCCLUSION USING PARTIAL SCENE REPRESENTATION

Major Field: Computer Science

Biographical:

Education:

Completed the requirements for the Master of Science in Computer Science at Oklahoma State University, Stillwater, Oklahoma in December, 2017.

Experience:

Graduate Teaching Assistant, Department of Computer Science, Oklahoma State University, January 2017 – May 2017.

Graduate Research Assistant, App-center, HBRC, Oklahoma State University, August 2017 – December 2017.