

Evaluating Tessellation and Screen-Space Ambient Occlusion  
in WebGL-Based Real-Time Application

by

Chenyang Li

A Thesis Presented in Partial Fulfillment  
of the Requirements for the Degree  
Master of Science

Approved March 2017 by the  
Graduate Supervisory Committee:

Ashish Amresh, Co-Chair  
Yalin Wang, Co-Chair  
Yoshihiro Kobayashi

ARIZONA STATE UNIVERSITY

May 2017

## ABSTRACT

Tessellation and Screen-Space Ambient Occlusion are algorithms which have been widely-used in real-time rendering in the past decade. They aim to enhance the details of the mesh, cast better shadow effects and improve the quality of the rendered images in real time. WebGL is a web-based graphics library derived from OpenGL ES used for rendering in web applications. It is relatively new and has been rapidly evolving, this has resulted in it supporting a subset of rendering features normally supported by desktop applications. In this thesis, the research is focusing on evaluating Curved PN-Triangles tessellation with Screen Space Ambient Occlusion (SSAO), Horizon-Based Ambient Occlusion (HBAO) and Horizon-Based Ambient Occlusion Plus (HBAO+) in WebGL-based real-time application and comparing its performance to desktop based application and to discuss the capabilities, limitations and bottlenecks of WebGL 1.0.

## DEDICATION

To my Parents and my Friends.

## ACKNOWLEDGMENTS

This research would have never been possible without the help of:

My committee, Dr. Ashish Amresh, Dr. Yalin Wang and Dr. Yoshihiro Kobayashi.

My beloved Mother and Father.

My friends who have contributions: Jiadong Xia, Tong Zhou, Xiaoyu Zhang, Rongyu Lin, Qiwei Wu, Xian Luo, Yicong Chen, Jing Li, Kewei Cheng, Weili Yi

# TABLE OF CONTENTS

	Page
LIST OF TABLES .....	vii
LIST OF FIGURES .....	viii
CHAPTER	
1 INTRODUCTION .....	1
2 RELATED WORK .....	3
Ambient Occlusion .....	3
Object-Space Methods .....	4
Screen Space Methods .....	5
SSAO .....	5
HBAO .....	7
HBAO+ .....	7
Surface Smoothing .....	8
Subdivision .....	8
Tessellation .....	8
Curved Point-Normal Triangles Tessellation .....	9
OpenGL .....	9
WebGL .....	10
3 METHODOLOGY .....	11
Screen-Space Ambient Occlusion .....	11
Horizon-Based Ambient Occlusion .....	15
Horizon-Based Ambient Occlusion Plus .....	18

CHAPTER	Page
PN-Triangles Tessellation.....	19
4 IMPLEMENTATION .....	25
OpenGL Pipeline.....	25
WebGL Pipeline.....	27
5 EVALUATION .....	31
Performance Evaluation.....	31
Quality Evaluation .....	31
Parameter Settings.....	32
Results .....	33
Quality Comparison .....	36
Quality Score.....	37
Performance .....	39
6 PROBLEMS AND BOTTLENECKS .....	42
Problems.....	42
Tessellation.....	42
Ambient Occlusion .....	43
Other Problems.....	44
Bottlenecks .....	44
Possible Solutions .....	46
7 CONCLUSIONS AND FUTURE .....	49
Conclusions .....	49
Future.....	50

CHAPTER	Page
REFERENCES.....	53
APPENDIX	
A CONTRIBUTIONS.....	58

## LIST OF TABLES

Table		Page
1.	Visual Outputs .....	34
2.	Quality Score .....	38
3.	GTX 970 Framerates .....	39
4.	GTX 970 WebGL Additional Evaluation.....	39
5.	GTX 960M Framerates .....	40
6.	GTX 960M WebGL Additional Evaluation.....	39



## LIST OF FIGURES

Figure		Page
1.	SSAO Sampling .....	13
2.	SSAO in Different Situations.....	14
3.	HBAO .....	15
4.	HBAO Ray-Marching .....	17
5.	Hardware Tessellation.....	21
6.	Bezier Patch .....	22
7.	SSAO 'Bleeding' .....	36
8.	HBAO and HBAO+ .....	36
9.	Before centralized VS After .....	47

## CHAPTER 1

### INTRODUCTION

Tessellation [1] and Ambient Occlusion [2] have been implemented in many real-time interactive 3D applications. They can provide better-detailed meshes with realistic soft-shadows which is able to improve the quality of the visual effect.

OpenGL [3] has provided many incredible real-time rendered applications in many platforms and areas. However, one of the limitation in latest OpenGL is that some of the new features are not supported on all platforms. WebGL 1.0 [4], based on OpenGL ES 2.0 [5], is a plugin-free web standard for low level Web-based computer graphics application which solves the problem for mobile and web. OpenGL ES 2.0 retains the major features from OpenGL 2.0, but it replaces most of the fixed functions with programmable ones in order to better fit embedded systems. As a result, some predefined functions in OpenGL is not compatible in OpenGL ES 2.0 and WebGL which leads to some issues.

One of the problems is that WebGL 1.0 does not support GPU Tessellation. Tessellating on GPU enables the usage of Level of Details. Without it on WebGL 1.0, this limits the quality and performance for rendering some models such as terrain, water, etc. Also, GPU Tessellation allows real-time tessellation which has a lot of potential for rendering advanced graphics features.

Meanwhile, Ambient Occlusion algorithms in WebGL are not as popular as that in OpenGL due to performance and relatively low-level 3D output. Another problem is that, Horizon-Based Ambient Occlusion Plus, the current state of the art Dynamic

Screen-Space Ambient Occlusion, uses many new features in D3D11 and OpenGL 4.x. These features are not available to WebGL 1.0 or are only available as extensions.

In this research, the goal is to mainly focus on real time rendering on WebGL, implement and evaluate Tessellation with Curved Point-Normal Triangles and all the popular Screen-Space Ambient Occlusion techniques: SSAO, HBAO, HBAO+. The implementation is completed on both OpenGL and WebGL with the same pipeline, while OpenGL's visual output and performance are served as standards for WebGL to compare with. Implementation on WebGL aims to provide reasonable quality and real-time rendering performance by following the OpenGL's core ideas and replacing the unavailable API features with WebGL available ones. After evaluation, bottlenecks and possible solutions are discussed and a future steps towards the rendering of Ambient Occlusion algorithms on WebGL are presented.

## CHAPTER 2

### RELATED WORK

#### Ambient Occlusion

Light surrounds the environment in the real world instead of simple countable light sources cast on surfaces. The light illuminates evenly from all directions is called ambient light which shades the softest shadows when it covers a large solid angle. Objects become flat and unrealistic when the importance of the ambient light shadows also known as ambient occlusion is ignored. Ambient Occlusion is not a direction-dependent type of shadowing effect, so precomputation provides decent results for static objects. However, for animating or deforming objects, dynamic ambient occlusion techniques is more useful [6].

As a result, real-time computer graphics community started to develop algorithms to capture the exposure of objects to the ambient lighting. The algorithms are categorized as Ambient Occlusion, also known as AO [7]. AO is generally used to portray the diffuse and non-directional shading effects around closed or crossed objects obscuring each other. It can solve the unrealistic shadow effects in the scene and improves the unclear presentation in corners, gaps and detailed objects etc. Over all, AO improves the perception, dimension and realism of physical space, the contrast and the artistry of the scene.

AO was originally introduced by Hayden Landis and his colleagues at ILM (Industrial Light & Magic) in Siggraph 2002 [8]. This method tries to address global illumination and ray tracing with less expense. AO is not a physically correct method, but it generally provides visual-satisfying results. The key to ILM's algorithm is to separate a

special pass which casts a single ray to trace occlusion from the final rendering. This technique was embedded in many ILM's productions [9], efficiently used in many movies and won the Scientific and Technical Academy Award in 2010 [10], however it was most commonly using on a non-real-time renderer which is not ideal for computer games or other applications require real-time rendering.

### Object-Space Methods

Bunnell proposed an algorithm to compute ambient occlusion by reforming surfaces into a set of disks by reference to mesh vertices instead of casting costly rays [11]. Bunnell uses a two-pass method to reduce the double-shadowing due to the unnecessary computation of overlapped disks. This approximation method runs relative expensive and yields some good results, but it is efficient and has an overall high-quality output.

Hoberock modifies Bunnell's algorithm to improve the quality at the expense of performance [12]. Ren et al. used spheres instead of disks for the computation [13]. The method can be simply projected onto a spherical harmonics basis. By using a logarithm and exponential transformation, it bypasses the expensiveness of multiplying spherical harmonic functions. Although the results are good, the spherical approximation does not take creases and smaller details into considerations.

Hegeman et al. count the blockers between the occluded point and the boundary of a volume consisting of small simple objects along the surface's normal direction [14]. It is a very fast algorithm which suits perfectly for grouping elements like trees, grass etc. It is also limited to this certain type of object which lacks universal usability.

Based on a scalar function of spatial position known as the distance field, Evans provided a different and interesting ambient occlusion method [15]. This method is suited for small scene and is nonphysical, but it produces satisfying results.

## Screen Space Methods

The major problem of object-space is that scene complexity is always required and becomes the major expense of the computation. Using only the easily extractable information like depth buffer, screen-space methods are independent from the scene complexity and require simpler data structure than storing spatial information. It is generally faster and better for use real-time applications at present.

### SSAO

In Computer games, interactive websites and other real-time applications, Screen Space Ambient Occlusion methods are a set of algorithms used to calculate the approximation of the real self-shadowing effects. The first algorithm used in video games named SSAO (Screen Space Ambient Occlusion) was developed by Vladimir Kajalin at Crytek and was applied in Crysis in 2007 [16]. It first extracts the scene depth buffer from a stored texture and calculates the information in a fragment shader allowing execution on GPU. For a given pixel on the screen, the occlusion is computed by getting the depth variance between the given one and the samples around it. SSAO uses a randomly rotated kernel as a smarter solution than a costly brute force method. The kernel changes directions periodically to generate only high-frequency noise which will be removed by a certain blurring pass before rendering the final image. With

approximately 16 samples per pixel comparing to the large sampling numbers required in a brute force one, such method provides a decent result as well as real-time rendering performance. Based on the image information, the performance of SSAO is uncorrelated with the scene complexity and good in dynamic scenes with only GPU usage required. However, the method might cause over-occluding and bleeding effect due to the locality limitation and the post-process blurring [17].

Luft et al. perform an unsharp mask filter on the Z-buffer in screen-space which is simpler and at a much lower cost [18]. By subtracting a blurred version from the original image, the result appears similar to the ambient occlusion effect with some variables tweaking.

Shanmugam and Arikan combine two techniques to produce the ambient occlusion [19]. The first method uses both depth and visible surface normal information in a full screen. Samples are generated from Z-buffer and represented as spheres, and occlusion is computed with normal vectors. A dark image is presented as a result since double-shadowing is not handled by any means. The second method is a coarse occlusion which shares the idea of the object-space method by Ren et al. which uses spheres to approximate the geometry and adds up in screen space. Double shadowing is not taken care of in this one as well which leads to darker scene.

Sloan et al. combine spherical harmonic exponentiation of Ren et al. with Shanmugam and Arikan's screen space technique [20]. In this technique, double shadowing is taken care of while higher-frequency visibility functions which can be used for mapping and lighting are produced along with ambient occlusion factors. However, this is a coarse occlusion since it does not use actual geometric information.

## HBAO

Louis Bavoil and Miguel Sainz from Nvidia describe an Image-Space Horizon-Based Ambient Occlusion [21], a physically-based technique with the input of Z-buffer and surface normal vectors. Z-buffer is used to compute the heightfield on which samples are marching to calculate the horizon angle, with the given pixel and its normal vector the tangent angle is calculated. By subtracting the sine values and averaging them over two-dimensional directions, the ambient occlusion effect completes and is passed into a blur filter to get the final result. The performance is mainly dependent on the screen and AO texture resolution. As a result, downscaling is often used to preserve reasonable real-time rendering framerates but might cause flickering defects under some circumstances. This defect was mentioned in GDC 2012 [22], however the presented method at the time can only partly fix.

## HBAO+

Louis Bavoil redevelops HBAO with a cache-aware interleaved texturing technique to process images utilizing the new features of Direct3D11 which allows rendering in full-resolution and faster in performance. To increase visual realism, Louis uses a simpler AO approximation inspired by the idea in Scalable Ambient Obscurance by McGuire et al [23]. HBAO+ is currently the best and widely-used image-space ambient occlusion algorithms in games with fully support by Nvidia's graphic cards [24].



## Surface Smoothing

### Subdivision

Subdivision surface is a commonly-used technique in computer graphics to smooth surface by calculating the approximation recursively from a polygon mesh. It was first introduced by Catmull-Clark and Doo-Sabin in 1978 [25]. With a given mesh, a refinement scheme which can be roughly divided into interpolating and approximating is applied to subdivide, generates new vertices of which the positions are calculated with respect to nearby old vertices and forms new faces. The refining process is applied repeatedly to produce a smoother mesh.

### Tessellation

Tessellation is a technique to manipulate polygons and divide them into renderable structures which is in most case presented as triangles in real-time rendering. Tessellation allows dynamic detail controlling in a mesh and silhouette edge managing which is fundamentally limited in former methods.

In OpenGL 4.0 and Direct3D 11, the primitive is the patch instead of element or array with the introduce of the tessellation shader [26]. A tessellator divide the patch into triangles whose degree is controlled by tessellation factors. With the advantage of running on GPU, subdivision surfaces can be rendered in real-time with tessellation.

Instead of a delivering tessellated surface of many triangles from the CPU to the GPU, sending the surface to the GPU to control the tessellation within is faster. This hardware tessellation is more efficient and provides inexpensive geometric data expansion [27]. A tessellator based on a fractional tessellation technique which tessellates a triangle into smaller triangles is added to the standard rendering pipeline. Independent

tessellation factor is available which makes a continuous level of detail possible and avoid defects. After that, vertices are passed into programmable shaders allowing the computation of the precise positions to create a smooth curved surface [28].

#### Curved Point-Normal Triangles Tessellation

PN-Triangles [29] is an advanced smoothing technique originally introduced by Vlachos et al. and then implemented with GPU Tessellation by John McDonald in his presentation in GDC2011 [30]. The idea is to smooth a low polygon mesh by replacing the original triangles with a Bezier surface and make use of the tessellation shader.

#### OpenGL

OpenGL [31], Open Graphics Library, is a cross-language, cross-platform application programming interface(API) for rendering graphics. It was developed by Silicon Graphics Inc. and then released in 1992, it is currently managed by The Khronos Group.

OpenGL is a pipeline-based, hardware-independent and client-server structured API. By specifying the primitives, performing calculations in different shaders, rasterizing and executing the fragment shaders, an image is rendered by OpenGL application through this pipeline while implementing on various hardware or software is available.

OpenGL 4.5 [32] is currently the newest version which contains features such as Tessellation Shader, new Buffer Texture formats, Uniform Buffer Object and explicit shader Uniform locations etc. Utilizing these features makes improvement possible in real-time tessellation and ambient occlusion.

## WebGL

WebGL is a standard 3D graphics API for the Web, on which programmers are able to make full use of the rendering hardware with JavaScript on browser without depending on downloadable plugins or installations. Although WebGL is not officially included in HTML5, many companies have made it as a component or package within their product in order to provide hardware-accelerated 3D experience.

WebGL is a free-to-use API based on OpenGL ES 2.0 which is aimed to create dynamic web applications on multi-platforms [33]. This nature makes it easy to implement and is consistent on different browsers and machines that support WebGL, but also requires developers work harder. However, several open source libraries [34] are available to simplify WebGL development which makes the low-level API more accessible.

Since WebGL is derived from OpenGL ES, the pipeline is the same. WebGL 1.0 is currently most widely used version which is sufficient for development most of the time. Nonetheless based only on OpenGL ES 2.0 [35], WebGL 1.0 does not fully provide all the features from OpenGL. Hardware tessellation, immutable storage or texture array etc. are not available or only served as an extension. As such, some of the OpenGL based applications cannot be ported directly. WebGL 2.0 [36] based on OpenGL ES 3.0 is a newer but less stable version which is experimental and has not yet been fully supported by major browsers.

## CHAPTER 3

### METHODOLOGY

#### Algorithms

##### Screen Space Ambient Occlusion

Generally speaking, a decent real-time AO effect often requires a well-performed ray-tracer, a good texture storage and other components which are able to efficiently handle complex scenes with diverse objects like buildings, characters and trees in a game level or targeting environment. Achieving the effect is time-consuming, resource-costly and requires great programming efforts.

Before Screen Space Ambient Occlusion was introduced, some problems of the former AO methods are addressed. Heavily depending on preprocessing and scene complexity, inconsistent processing between static and dynamic scenes and implementation complexity are some of the main obstacles that time-critical applications with dynamic 3D complex environments need to overcome. To counter these problems, SSAO is a GPU-based approach with a faster approximation of AO shadowing in real-time.

The basis of Screen Space Ambient Occlusion is to compute the AO factor from the depth buffer which samples the surfaces in a discrete manner which allows dynamic computation. The depth values are stored in a texture and can be accessed by shaders on a GPU. For each pixel on the screen, ray-tracing is performed from selected pixel into surrounding and intersection is checked to decide the occlusion factor. This method is generally very expensive and inefficient, but the problem can be solved or avoided by SSAO. SSAO first retrieves scene depth value from a pre-stored texture. Then a full-

screen quad is rendered with the computation result from shaders. In the end, a post-blur is executed to produce the final output with the AO factors.

Human eyes cannot capture the long-distance high-frequency and short-distance low-frequency details and most of time occlusion near the screen border which potentially will theoretically cause problems often looks reasonable. As such, the most important occlusion effects are inside the area of the screen-space allowing computation to ignore the objects outside of the camera frustum but still deliver a satisfying AO shadow.

The information from depth buffer is relatively limited and not fully three dimensional which means the geometry blocked by the visible objects are not taken into account. This lack of information might cause minor visual problems which are fixable by more complex solutions. However, the trade-off between the minor improvement and major performance reduction is not a good decision for hardware. The artifacts are ignorable in general and since the goal is to render in real-time, SSAO only computes the occlusion with the visible information within the screen-space.

According to the basic idea of Ambient Occlusion, in order to achieve a better visual effect, a larger amount of rays, sometimes more than hundreds, have to be cast per-pixel on the screen which is beyond the performance of the current hardware generation. So approximating the occlusion from ray-tracing is the method proposed in SSAO by calculating the ratio between the samples hit the objects and the empty space around a chosen point. As shown in the Figure 1, samples are distributed in a sphere kernel around the point P and the distribution inside or outside objects are the information to be tracked by comparing the sample depth and the stored depth of P. If sample depth is greater, the

sample is considered inside the object, otherwise is outside. The different visual effect is generally categorized into three scenarios: flat, corner or edge (Figure 2) which produces different level of occlusion. The higher the ratio, the darker the effect is. Note that edge usually appears highlighted because the ratio is low, it is physically wrong but is still left in SSAO to improve the spatial perception.

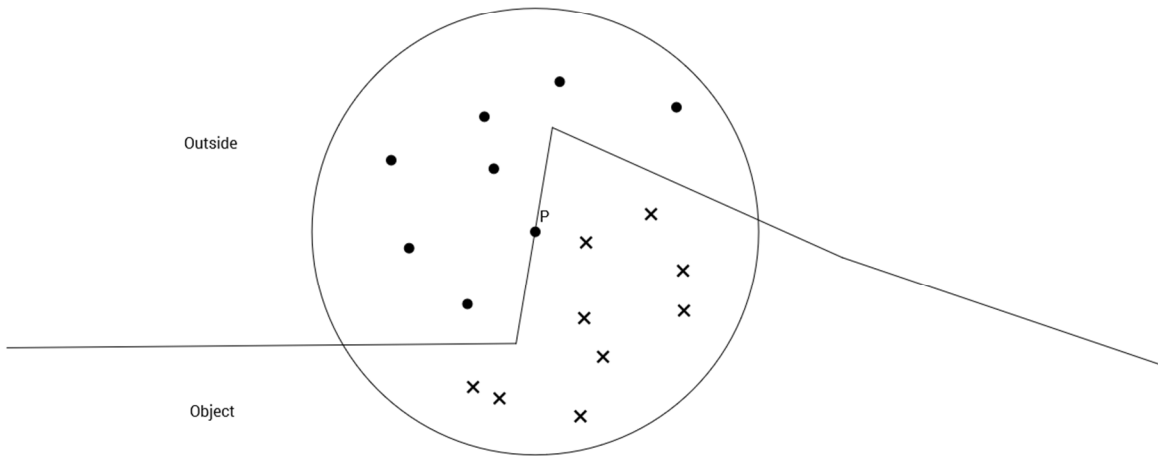


Figure 1. SSAO Sampling

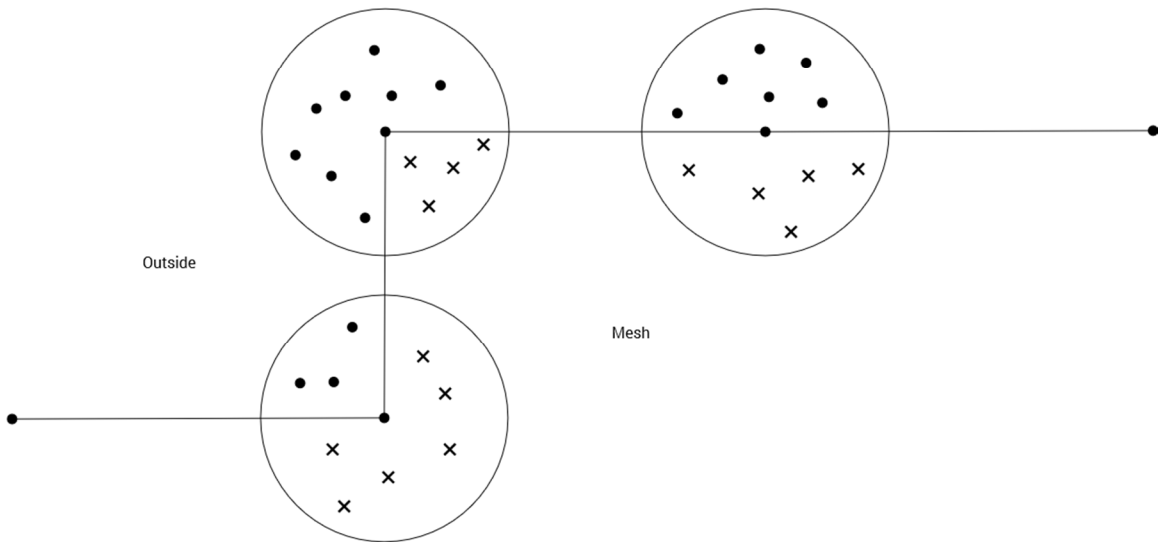


Figure 2. SSAO in Different Situations

Lighting distance attenuation is accounted when sampling the kernel. The sample points in the kernel are distributed in a non-uniform manner where the samples are more densely placed near the center of sphere than the borders. Taking this different weights into consideration, the occlusion effect is denser and more accurate near the chosen point without needing any more computation to achieve the attenuation effect. However, occlusion effect is not always good with respect to distance especially in the case where scene information is not complete due to the disadvantage of depth buffer. Under such circumstance, SSAO does a depth range check and occlusions from samples with large difference are smoothed.

In order to render in real-time, large sample size is not ideal while small size will sacrifice the image quality. In SSAO, each pixel uses a limited number of fetches (16 in most cases) to improve performance and a randomly rotated kernel of small size samples is used to simulate higher size [37]. The kernel rotates and repeats every 4\*4 screen pixels and then a 4\*4 blur step is used to remove the high-frequency noise produced by the kernel rotation. As such, SSAO is able to produce 256 samples for every 4\*4 pattern and remove noise to achieve good results.

### Horizon-Based Ambient Occlusion

Image-Space Horizon-Based Ambient Occlusion, commonly known as HBAO, is an Ambient Occlusion developed by Louis Bavoil and Miguel Sainz at Nvidia in 2008. It is a quality-improved ambient occlusion method that also takes scene normal vectors into account and has a better physical basis.

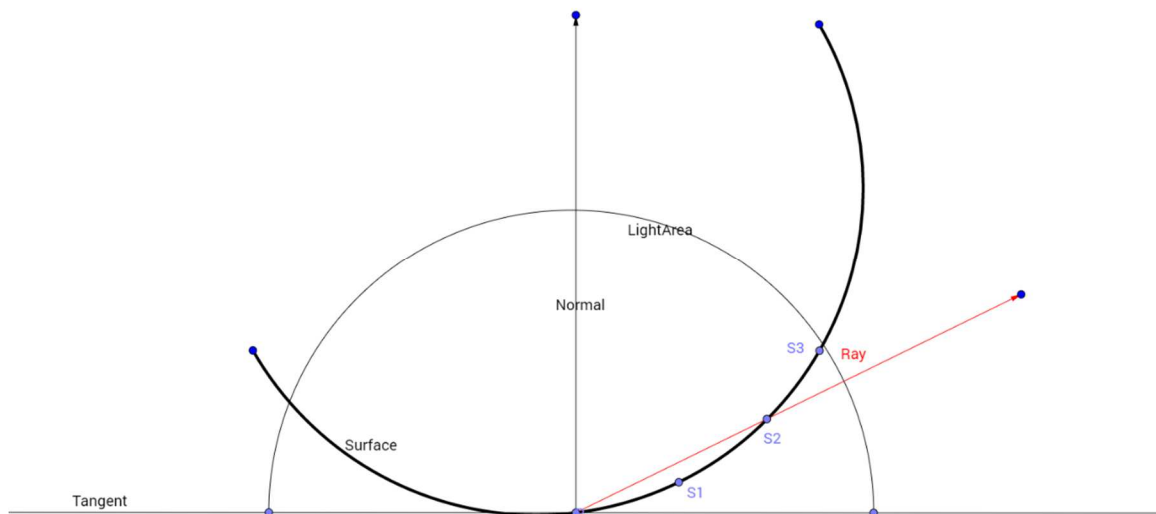


Figure 3. HBAO



The physical basis of HBAO is to use marching rays [38] from a surface point P in a hemisphere oriented around the its normal vector. Rays are casted and marched along a direction to check the intersection with a height field inside a normal-oriented hemisphere (Figure 3). The sample point S1 which is the first one below the current height field of the ray is selected approximately as the intersection point. The height is decided by comparing the depth of the samples and the related depth on the ray. Then the ray marches with a uniform step size to approximate the next intersection point S2. The marching ends when reaching the end point S3 which is at distance R from the starting point. All the samples will be projected into screen space to produce the depth texture. Noted that the algorithm ignores samples with further distance than R to avoid the artifacts of the depth discontinuities. At least three rays per direction is needed to deliver decent results in practice to compute the Monte Carlo Integration (Equation 1) where V is the visibility function returns 1 or 0 and W is an attenuation function to smoothen the ambient occlusion.

$$A = 1 - \frac{1}{2\pi} \int V(\vec{\omega})W(\vec{\omega})d\vec{\omega}$$

Equation 1

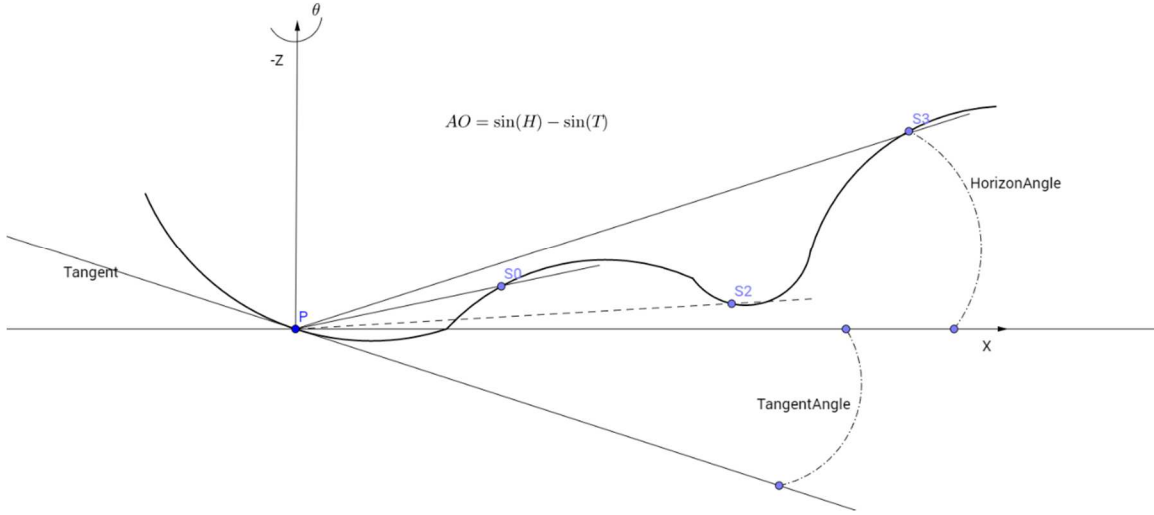


Figure 4. HBAO Ray-Marching

Based on the ray-marching idea, HBAO distributes and steps along 2D directions in image space around the current pixel instead of view space TBN basis. To decide whether a sample contributes to occlusion or not, the elevation angle of it is compared with one of the former sample. If it is greater, the sample is taken into account (Figure 5). For each accounted sample, sine of its Horizon Angle and sine of its associated Tangent Angle is computed and subtracted to calculate the horizon-based integral. As such, the equation (Equation 1) is reformulated into equation (Equation 2) where  $N$  is the number of sample,  $W$  is the attenuation function,  $t$  is the tangent angle,  $\psi$  is the horizon angle and  $\theta$  is the 2D direction. Attenuation function is a quadratic one to attenuate softly than common linear method.

$$A = 1 - \frac{1}{2\pi} \int_{\theta=\pi}^{\pi} \sum_{i=1}^{N_s} W(\vec{\omega}_i) [(\sin \phi_i - \sin t_i) - (\sin \phi_{i-1} - \sin t_{i-1})] d\theta$$

Equation 2

As such, HBAO needs to reconstruct the view-space position of a pixel by using its screen-space coordinate and a stored linear depth buffer while using its surface normal vector to calculate per-sample tangent [39]. A random jitter is stored as a texture for the randomly rotation of the 2D direction in image space and the sample locations are decided by this direction with a computed step size from the projecting of the radius. With samples picked, tangent and angle computed, ambient occlusion is computed accordingly and passed to the noise filter to get the final results.

#### Horizon-Based Ambient Occlusion Plus

After HBAO was introduced to the game industry by Nvidia, some quality and performance problems were found. The major issue performance-wise was that when HBAO renders at full-resolution, the frame rate will be unacceptable for game scenes with complex details. As such, games using HBAO at the time generally rendered at half-resolution which led to the quality-wise problems. Flickering, which often appears in motion on thin objects, is the main quality problem which is difficult to get rid of in every case. Louis Bavoil revamped HBAO to create HBAO+ to tackle the problems mentioned above [40].

With the launch of DirectX 11 in 2009, a fast interleaved sampling method is able to improve HBAO methods with a cache-efficient manner. Before interleaved sampling is used, AO methods often sample with random pattern or jittered pattern. These sampling approaches either lack of spatial locality or lack of efficiency on increased kernel size. The idea of interleaved rendering is to “render each sampling pattern separately using down-sampled input textures” [41].

For an input texture, a depth buffer in ambient occlusion, the first step in interleaved rendering is to de-interleave it by separate it into texture arrays at reduced resolution. For each one of texture array, a jitter-free sampling is applied on it with a constant jitter value per draw. After the sampling process, the texture array fetch is performed on each pixel. Constant value grants better locality and reduced resolution is memory-efficient.

In ambient occlusion,  $4 * 4$  interleaving is the standard. It de-interleaved input into a quarter-resolution texture array with 16 elements and execute 2 draw calls with 8 multiple rendering targets. Then sampling the texture array and applying the AO calculations in 16 draw calls. Finally, the output is interleaved in one draw call and feed into noise filter. With the interleaved rendering method, HBAO+ can render at decent frame rates which is 2 - 3 times faster than HBAO at full-resolution and overcome flickering at the same time.

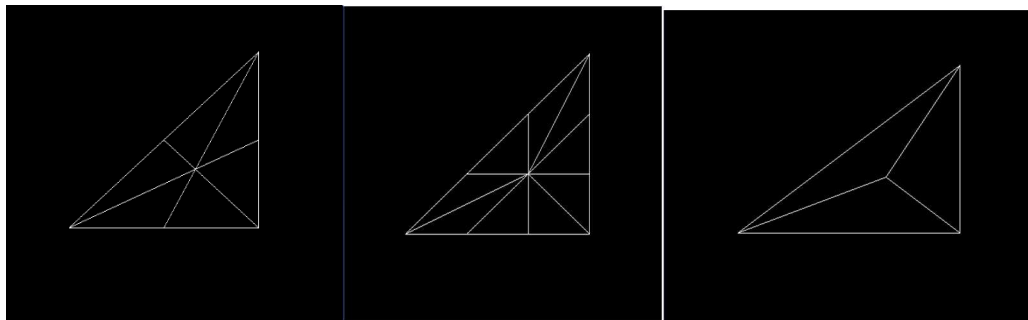
### PN-Triangles Tessellation

Tessellation on GPU is a new feature first introduced in OpenGL 4.0 and used for dynamic surface smoothing on GPU [42]. 3D models are relatively static, so achieving dynamic level of details is often hard and not cost-efficient. In real-time rendering, same model will be rendered with more details when closer to the camera in order to achieve better resource allocation for more efficient computation. However, before hardware tessellation became available, the typical method for distant-based level of detail is to have different polygon counts on the same model and apply them in different situation. This method works in some cases but requires unnecessary modelling process. With

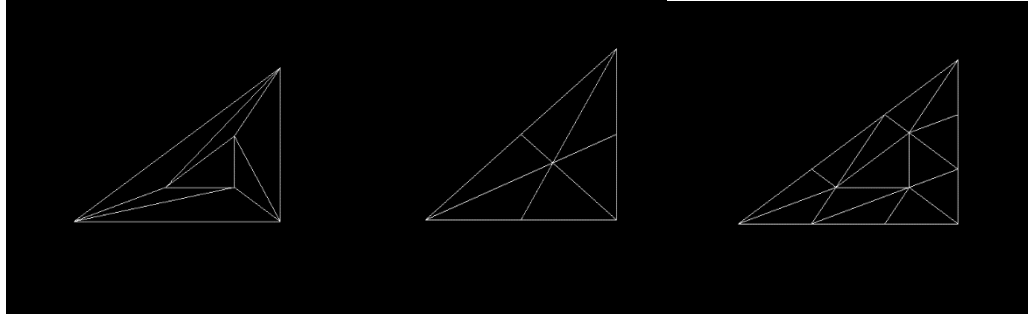
tessellation pipeline, the same level of detail can be achieved by using a low-poly model and subdivide the model accordingly in real-time.

Comparing to the traditional OpenGL pipeline, tessellation has two additional programmable stages: Tessellation Control and Tessellation Evaluation. Between them is a fixed function stage called tessellator or primitive generator which handles the primitives. When tessellation is active, the primitive type can only be `GL_PATCHES` and the subdivide process is done through Tessellation Control shader(TCS) and Tessellation Evaluation shader(TES) [43].

TCS works on a patch often known as control points. By moving control points with the shader, surfaces are defined and shaped. With an input patch and both inner and outer tessellation level (Figure 5), TCS transforms the control points and outputs a patch. In TCS, tessellation levels can be fixed or changing on the fly by applying various algorithms.



(Left to Right: inner 1, outer 2; inner 1 outer 3; inner 2 outer 1)



(Left to Right: inner 3, outer 1; inner 2 outer 2; inner 3 outer 3)

Figure 5. Hardware Tessellation

After TCS, the subdivision is done by the fixed function stage, Tessellator. Tessellator uses the levels to subdivide a domain defined by barycentric coordinates. As such, the output patch from TCS is not actually tessellated in Tessellator which has no access to the patch. Tessellator instead generates points inside the domain and marks them with their unique barycentric coordinates.

Since Tessellator has no access to the patch, TES is needed to read the barycentric coordinates along with the output patch from TCS to generate vertices. In TES, patches can be used to control a surface by using different smoothing algorithms. After TES, comes with the traditional pipeline to rasterize.

With the introduction of hardware tessellation, real-time tessellation with dynamic manipulation becomes possible. In GDC2011, John McDonald presented *Tessellation On Any Budget* using Point-Normal Triangles technique in tessellation shader. This technique was described by Alex Valchos et al. in 2001 and it is a surface smoothing algorithm based on Bezier Surface which basically smooths surface by moving control points with a polynomial function. PN-Triangles method, fits well with the input patch in

shaders, is based on Bezier Triangle where new vertex positions and normal vectors will be calculated.

New vertex positions are computed in the form [44]:

$$\begin{aligned}
 b(u, v, w) &= \sum_{i+j+k=3} b_{ijk} \frac{3!}{i! j! k!} u^i v^j w^k \\
 &= b_{300}w^3 + b_{030}u^3 + b_{003}v^3 + b_{210}3w^2u + b_{120}3\omega u^2 + b_{201}3w^2v \\
 &\quad + b_{021}3u^2v + b_{102}3wu^2 + b_{012}3uv^2 + b_{111}6wvu
 \end{aligned}$$

‘ $uvw$ ’ are barycentric coordinates while  $B_{ijk}$  are control points (Figure 6).

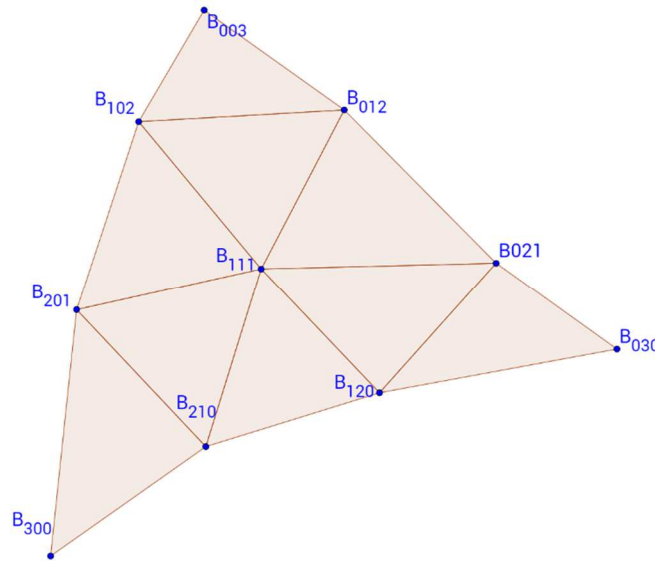


Figure 6. Bezier Patch

Define a triangle with 3 vertices whose position is P and normalized normal vector is N. Then  $B_{ijk}$  can be calculated by:

$$b_{300} = P_1 b_{030} = P_2 b_{003} = P_3$$

$$w_{ij} = (P_j - P_i) \cdot N_i$$

$$b_{210} = \frac{1}{3}(2P_1 + P_2 - w_{12} N_1)$$

$$b_{120} = \frac{1}{3}(2P_2 + P_1 - w_{21} N_2)$$

$$b_{021} = \frac{1}{3}(2P_2 + P_3 - w_{23} N_2)$$

$$b_{012} = \frac{1}{3}(2P_3 + P_2 - w_{32} N_3)$$

$$b_{102} = \frac{1}{3}(2P_3 + P_1 - w_{31} N_3)$$

$$b_{201} = \frac{1}{3}(2P_1 + P_3 - w_{13} N_1)$$

$$b_{111} = E + \frac{1}{2}(E - V)$$

$$E = \frac{1}{6}(b_{210} + b_{120} + b_{021} + b_{012} + b_{102} + b_{201})$$

$$V = \frac{1}{3}(P_1 + P_2 + P_3)$$

The normal vectors are defined as:

$$\begin{aligned} n(u, v) &= \sum_{i+j+k=2} n_{ijk} u^i v^j w^k \\ &= n_{200} w^2 + n_{020} u^2 + n_{002} v^2 + n_{110} wu + n_{011} uv + n_{101} wv \end{aligned}$$



And  $n_{ijk}$  can be defined as:

$$n_{200} = N_1 n_{020} = N_2 n_{002} = N_3$$

$$v_{ij} = 2 \frac{(P_j - P_i) \cdot (N_i + N_j)}{(P_j - P_i) \cdot (P_j - P_i)}$$

$$n_{110} = \frac{(N_1 + N_2) - v_{12}(P_2 - P_1)}{\left| (N_1 + N_2) - v_{12}(P_2 - P_1) \right|}$$

$$n_{011} = \frac{(N_2 + N_3) - v_{23}(P_3 - P_2)}{\left| (N_2 + N_3) - v_{23}(P_3 - P_2) \right|}$$

$$n_{101} = \frac{(N_3 + N_1) - v_{31}(P_1 - P_3)}{\left| (N_3 + N_1) - v_{31}(P_1 - P_3) \right|}$$

With the new P and N computed, the mesh is smoothed.

## CHAPTER 4

### IMPLEMENTATION

#### OpenGL Pipeline

The goal of implementation in OpenGL is to provide both visual and performance results as references for the comparison with WebGL. The pipeline includes hardware tessellation is typical and straightforward. First the program reads the input from a '.OBJ' file and save the data into buffers. Then the data is passed to tessellation shaders and the surface is tessellated on GPU with PN-Triangles algorithm. After that, vertex shader and fragment shader calculates the ambient occlusion effect on the scene with tessellated meshes. Then, the visual output as well as the framerate is ready to be analyzed.

After processing the information from '.OBJ' file, per-vertex normal needs to be reconstructed since some of the shared vertices have different normal values when in different faces. These values are often the original data in the input file and without averaging them, the smoothing result will be wrong for normal vectors play an important role in PN-Triangles method. For each vertex, the normal reconstruction will be executed by averaging the sum of adjacent faces' normal vectors and normalizing the result. The reason of this step is that '.OBJ' file sometimes stores duplicate vertices which have the same position but different normal vectors. For instance, two triangles share one vertex. In reality, the vertex should have one position and one normal value. However, sometimes '.OBJ' stores two face normal values independently which leads to the same vertex has two normal vectors when referring to different triangles. Face normal is always perpendicular to the triangle, so in PN-Triangles tessellation the patch of vertices will be subdivided in 2D space which leads to failure of smoothing. Thus, each vertex

should have a weighted normal value from adjacent faces to preserve 3D spaciousness. This process happens when reading files in the CPU.

With data preprocess completed, tessellation shaders start to take over the tessellation process. Tessellation Evaluation Shader is where the core algorithm of PN-Triangles takes place with the tessellation level gathered from the Tessellation Control Shader. After that, the output is a smoothed mesh with better details.

The tessellation process completes and outputs a scene with our detailed new mesh, then ambient occlusion post-processing will be applied. Despite the difference of the three image-based algorithms mentioned before, scene depth is needed as an input texture. SSAO and HBAO have basically the same pipeline where the AO computation is happening mainly in the fragment shader. HBAO needs a normal texture as well, however in this implementation, normal vectors are reconstructed in the fragment shader using the depth buffer and camera attributes. HBAO+ is slightly different in the pipeline with the addition of the de-interleave stage and re-interleave stage. The de-interleave stage uses fragment shader to get 8 texels around the current pixel from depth texture with fixed texture coordinate offsets. Each group of texels forms a new texture and stores separately in a texture array with 16 elements. Then AO computation is done on each texture element and store in another texture array with the same size. After that, an extra fragment shader handles the re-interleave process by selecting result texel one by one with respect to offset coordinates in de-interleave stage. When AO is done, a blur will be applied on all three algorithms to get the final results.

## WebGL Pipeline

Based on the completed implementation in OpenGL, the pipeline is relatively similar in WebGL. Read in “.OBJ” files first, then reconstruct the normals and process the model with tessellation and different ambient occlusion algorithms. However, WebGL and JavaScript have their own characteristics, thus extended libraries and modifications are very important to achieve the similar output.

The JavaScript libraries used in WebGL implementation is `pregl.js` and `preglxt.js`. The “`pregl.js`” is a lightweight graphics math and WebGL library by Dean McNamee [45] and “`preglxt.js`” is an extension based on `pregl` to do 3D rendering, texturing etc by Marcin Ignac [46]. In order to load models from files, “`webgl-obj-loader.js`” [47] is needed to process vertices, normals and indices.

With all the extensions, modifications are required for the data processing. The first problem is the `obj-loader` library does not consider repeated indices and it also reorders the indices from the files when pass the data to the drawing stage. This is reasonable and easy to do when the goal is to only load and draw the model. However, tessellation requires a more precise and accurate indices array without reordering, otherwise the topology will change and break when tessellated. This will not affect rendering mesh, but will cause tessellated mesh to be in a mess occasionally. The way to handle the vertices in `webgl-obj-loader` is to read the lines contain face information and create new indices every read. For instance, if the first face in the file contains vertex 6/12/50, it will be processed as 1/2/3. If another face contains 6/24/50, it will create 4/5/6 and ignore the sharing vertices. Instead of creating new, the modification passes the original ID for better reference in the tessellation stage.

The data is saved into four arrays: vertices, normals, indices and texture coordinates, but only the first three are used in this implementation. Reconstructing normals is necessary for the same reason as in OpenGL and the idea is the same. The new challenge is how to tessellate the model in WebGL without making too many changes than in OpenGL. As mentioned before, OpenGL provides a hardware tessellation, in other words, shaders can help to tessellate with popular smoothing algorithms in real-time. However, this is a relatively new feature and not currently available in WebGL. Based on the idea of patch, pnTess.js is implemented to simulate the hardware tessellation with CPU in WebGL.

The first thing to do in pnTess is to tessellate triangles. The goal is to subdivide triangles into smaller triangles with the same methods introduced in tessellation shader. After computing and evaluating with different inner and outer levels, a different but easier dividing method is used. For each level, the midpoint of each edge is picked and connected to form a total of four new sub-triangles. This method is the fastest and also highly suitable for barycentric coordinate system for later computation. With the advantage of barycentric coordinate, the edges can be easily tracked, the vertices and normal vectors are arranged into patches and the function of Tessellation Control Shader is completed.

The next step is to implement PN-Triangles. Since the previous step has already provided new triangles and patches, this one is very similar to what happens in Tessellation Evaluation Shader. After completion, the model is now with new positions and normals while smoothed before the rendering stage.

After rendering the tessellated model on screen, Image-Space Ambient Occlusion is processed. SSAO and HBAO both use three pairs of vertex and fragment shaders to compute depth, compute ambient occlusion and post-blur. The core of these shaders are quite similar to OpenGL implementation with slight modifications to be compatible with WebGL. These modifications are mainly on variables types and loop constraints which do not affect algorithms themselves.

The major modifications happen in HBAO+ algorithms. The goal is to make the algorithm works through modification without sacrificing and providing performance and visual output as close to OpenGL as possible. The biggest difference between HBAO and HBAO+ as mentioned previously is the de-interleaved texturing. Based on the OpenGL implementation, a 4\*4 texture array is needed first. OpenGL 4.0 supports array texture which does not exist in current version of WebGL, so it is necessary to store 16 textures instead of one texture array. With these textures, de-interleaving requires binding them to different color attachment instead of one and draw them on a single framebuffer. This can be easily done in OpenGL with calling `glBindFramebuffer` and `glDrawBuffers`. However, draw buffers and multiple color attachments are only available as an experimental extension now on WebGL and needed to be enabled. It is also a D3D11 level API which means it may not be supported in some browsers or machines. Although limited, these extensions are good enough in this implementation. After calling the extensions, multiple color attachments and draw buffers is used in the same way.

Attaching texture to framebuffer in WebGL is of no difference, so all the framebuffers and de-interleaved textures are well-prepared for shaders to process. In the

de-interleaving fragment shader, OpenGL executes pixels with `texelFetchOffset` function to distribute them into texture arrays which again is not in WebGL. Texel is different with texture coordinates for it ranges from 0 to max width/height. This means when calling a texel returns the exact screen position and it is very accurate for offsets. However, WebGL can only use the traditional coordinates ranges from 0 to 1. This might potentially cause inaccuracy in offsets.

At this point, depth textures are de-interleaved and ambient occlusion shaders calculate the AO factors. `glFramebufferTextureLayer` is an OpenGL function to bind array textures to framebuffer. In WebGL, with looping and indexing, same functionality can be achieved. After that, there will be 16 textures containing AO results waiting to be re-interleaved into one. First, a slice ID is computed via screen positions to decide which texture should be read and draw on. WebGL has to take 16 textures while OpenGL takes 1 texture array. Other than using texture coordinates over texels in WebGL, dynamic indexing has to be changed into constants. Under these constraints, a very long if-else has to be written to decide IDs instead of a quick loop. All these modifications make the code redundant and relatively inaccurate in fetching texels.

After re-interleaving, blurring stage is the same and WebGL can provide a reasonable output and real-time runnable performance.

## CHAPTER 5

### EVALUATION

The ultimate goal of real-time rendering is to render at least 60 frame per second. This is also the goal of this implementation. A desktop with Nvidia GTX 970 graphics card and a laptop with Nvidia GTX 960M are used for evaluation. Newest version of Chrome and Firefox are used for testing on WebGL.

#### Performance Evaluation

In OpenGL, the processing time of each frame is recorded for evaluation. Frame per second equals the reciprocal of the recorded time. Noted that the vsync options need to be disabled in Nvidia graphics cards to unlock the 60FPS cap.

Both Chrome and Firefox locks the maximum FPS at 60 and cannot be disabled, thus, evaluating recorded time will not be as effective. For high performance machines, all algorithms will be rendering at 60FPS which cannot find a difference. Based on the idea from WebGL Performance Benchmark, the number of total renderable objects are recorded at a fixed range of frame rate in a fixed space. For instance, if setting a fixed 3D space in front of a camera and the frame rate range sets between 45 to 50, the higher the number of objects, the better the performance. In this way, the FPS lock is bypassed and the performance is represented by the number.

#### Quality Evaluation

The quality evaluation contains two parts. First part is to compare the output image with the certificated output from Nvidia. This part is aiming to decide whether the



algorithms are working properly. Also the details from the corner and crack will be compared.

Second part is a survey for people to give scores on the outputs they saw. The score is between 1 - 5 for the worst and best quality.

### Evaluation Targets

Suzanne monkeys [48] and Sibenik Cathedral [49] are the meshes used for evaluation. Suzanne monkey is a relatively low-poly mesh which is decent for tessellation, it also has some details suit for ambient occlusion. Monkey mesh contains 511 vertices and 968 faces.

On the other hand, Sibenik Cathedral is used for testing ambient occlusion only since it has more details and is more resemble to a scene in an interactive application. Basically, Suzanne focuses on performance testing and Sibenik Church on quality. Sibenik Church mesh contains 40979 vertices and 75283 faces.

### Parameter Settings

#### Tessellation Level

OpenGL's tessellation level starts at 0 and can be increased by user input. However, after certain level the visual quality will not increase but the performance will decrease, thus the cap of the level is set to 200.

Due to the performance limitation, the WebGL's tessellation level range is between 0 - 3.

## Ambient Occlusion

In order to have a better comparison, all the parameters are set the same on OpenGL and WebGL. Some parameters are customizable and some are fixed.

The screen resolution is fixed to 1920 \* 1080. The perspective camera's field of view is 45.0 degree, aspect ratio is the same as the screen, the near plane is at 1.0 and the far plane is at 2000.0.

### SSAO

The customizable parameters are sample size, radius and noise texture scale.

### HBAO

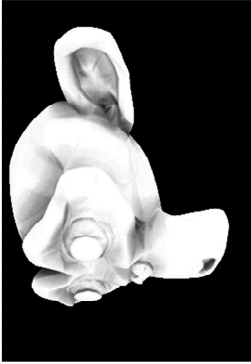
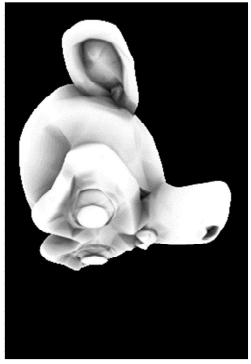
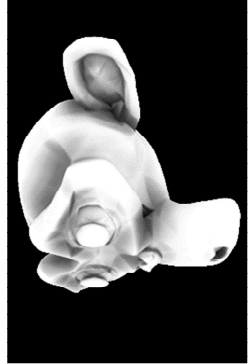


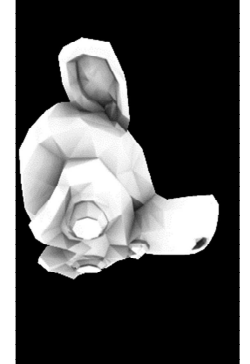
The customizable parameters are number of sample steps, number of sample directions, noise texture scale, radius, angle bias.

### HBAO+

In interleaving process, texture array size is fixed at 16, draw buffer size is 8. The customizable parameters are number of sample steps, number of sample directions, noise texture scale, radius, angle bias etc.

## Results

Following table provides the visual outputs of the implementation:

OpenGL					
	Church		SSAO	HBAO	HBAO+
	Monkey + tessellation				
	Monkey no tessellation				

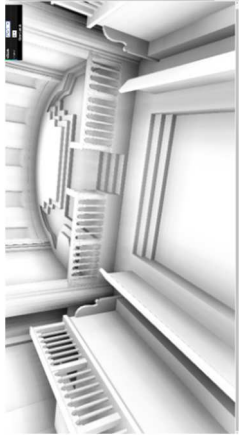
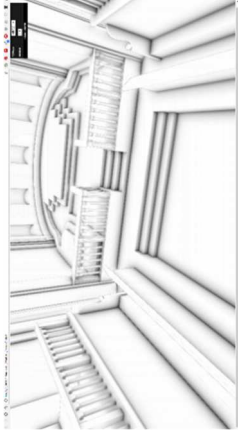
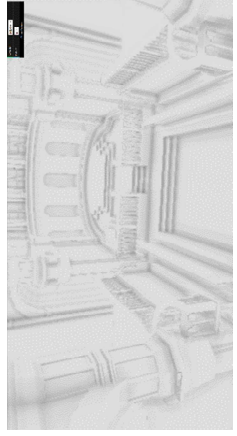
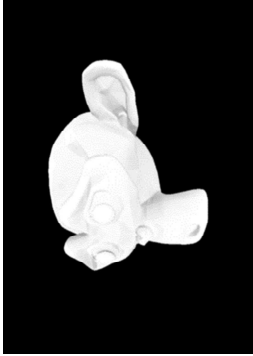
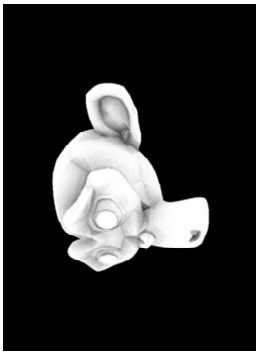
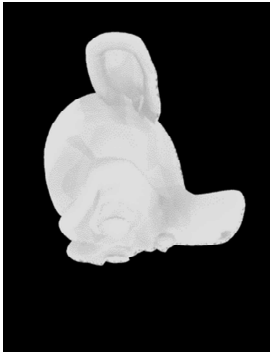
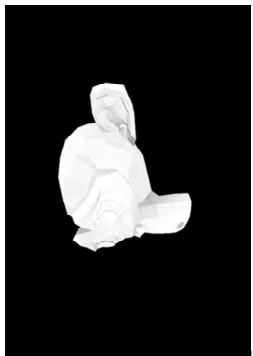
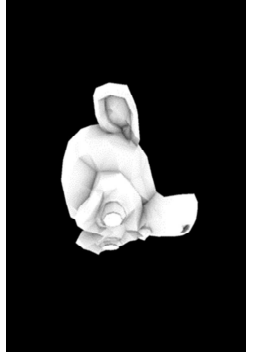

		SSAO	HBAO	HBAO+
WebGL	Church			
	Monkey + tessellation			
	Monkey no tessellation			

Table 1 Visual Outputs

## Quality Comparison

### Before and After Tessellation

As is shown in Table 1, PN-Triangles Tessellation improves the details of the meshes in both OpenGL and WebGL implementations. It is pretty clear that the edges around Monkey's ears, eyes and corners are better smoothed after Tessellation. Since the mesh's details are enhanced, the effect of three Ambient Occlusion gets better.

### SSAO, HBAO and HBAO+

The overall quality of the three algorithms does not have too many distinctions. The edges and corners are satisfying in most cases. However, SSAO does provide an unpleasant 'bleeding' effect as shown in Figure 6.

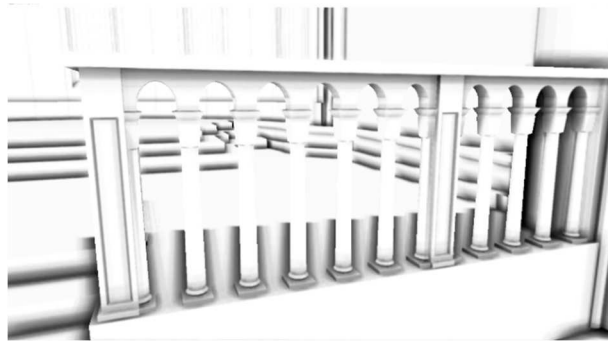


Figure 7 SSAO 'Bleeding'

As can be seen in the image, the pillars' bottoms have shadows which look weird and unrealistic. This kind of effect appears to be softer in HBAO and HBAO+ (Figure 7).

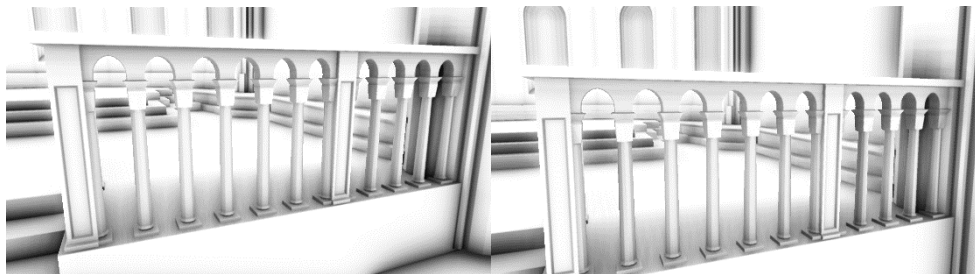


Figure 8 HBAO and HBAO+

## OpenGL and WebGL

The results of WebGL is reasonable and acceptable. SSAO on WebGL is not as good as on OpenGL because the samples are centralized in this implementation. This is solved in later Chapter. HBAO on both are very similar but HBAO+ is much worse due to the limitations and constraints on WebGL.

## Quality Score

The score is ranged from 1 – 10. Participates are from different majors to have a diversity. And this survey is a User-Experience focused than technical.

The scoring standard is not specifically given to the participates, so the score is highly related to personal preferences. Additionally, participates have different level of knowledge on real-time rendering, shadows, smoothing and so on. Based on the fact, participates are not given too much information about the tessellation or ambient occlusion algorithms. This ensures the participates to give scores according to their pure feelings and avoids subliminal influence from additional information.

The surveys were sent to participates separately and were completed independently to make sure their decisions are not affected by others'. The result is in Table 2 which is the average of the total score from their feedbacks. In addition to score, some participates also provided their thoughts of the images and reasons of scoring.

		SSAO	HBAO	HBAO+
OpenGL	Church	7.0	8.1	7.7
	Monkey + tessellation	7.0	8.3	7.9
	Monkey no tessellation	6.6	7.9	7.7
WebGL	Church	7.6	6.6	5.1
	Monkey + tessellation	6.1	8.0	4.4
	Monkey no tessellation	6.0	7.7	4.4

Table 2 Quality Score

The score shows that HBAO and HBAO+ are better than SSAO while Tessellation can improve the quality in general.

HBAO+ on WebGL has the lowest score because the output is blur and it has ‘dirty’ broken pixels on the screen according to the feedbacks from the participates. Also, participates tended to prefer SSAO of Sibenik Cathedral than HBAO on WebGL.

Another interesting feedback from participates of Art majors is that they would prefer the low-poly meshes than the tessellated ones because of the current trend in design.

Performance

		SSAO	HBAO	HBAO+
OpenGL	Church	250FPS	125FPS	270FPS
	Monkey + tessellation	200FPS	121FPS	212FPS
	Monkey no tessellation	256FPS	142FPS	278FPS
WebGL	Church	60FPS	60FPS	60FPS
	Monkey + tessellation	60FPS	60FPS	60FPS
	Monkey no tessellation	60FPS	60FPS	60FPS

Table 3 GTX 970 Framerates

	FPS	SSAO	HBAO	HBAO+
Original	54-60	1374	159	243
	45-50	1669	206	305
	30-34	2376	338	456
Tessellated	54-60	320	93	225
	45-50	392	125	291
	30-34	592	231	438

Table 4 GTX 970 WebGL Additional Evaluation



		SSAO	HBAO	HBAO+
OpenGL	Church	102FPS	80FPS	103FPS
	Monkey + tessellation	83FPS	50FPS	83FPS
	Monkey no tessellation	105FPS	57FPS	106FPS
WebGL	Church	43FPS	47FPS	60FPS
	Monkey + tessellation	58FPS	56FPS	59FPS
	Monkey no tessellation	59FPS	58FPS	60FPS

Table 5 GTX 960M Framerates

	FPS	SSAO	HBAO	HBAO+
Original	54-60	163	61	218
	45-50	792	123	330
	30-34	1325	203	525
Tessellated	54-60	116	26	80
	45-50	153	78	170
	30-34	242	137	291

Table 6 GTX 960M WebGL Additional Evaluation

## Performance Analysis

SSAO has a better performance than HBAO in general. HBAO+ is better than HBAO as well. HBAO+ and SSAO has quite similar performance but SSAO is slightly better.

This performance is reasonable theoretically. And most of the time, the framerate is above 60 or at least above 30 in this parameter setting. This proves that the algorithms are real-time renderable on WebGL.

## CHAPTER 6

### PROBLEMS AND BOTTLENECKS

Since the implementation on OpenGL is served as a standard for WebGL to compare with, this part mainly focuses on the problems and bottlenecks of WebGL implementation.

#### Problems

##### Tessellation

OpenGL uses hardware tessellation which is not currently available on WebGL. The output quality is very similar, however, since WebGL cannot tessellate with GPU, the performance is much worse.

The first problem is that the tessellation level cannot exceed certain level on WebGL. As the result shows, OpenGL tessellation level can reach as high as 200 and can be controlled by user input at real time. But WebGL level is nonetheless at most set to 7 or 8 on testing machine, and changing level will sometimes cause a large amount of framerate drop or crash the program when the level is already high. This is not allowed since the goal is to render at real-time.

Unable to render at real-time is the second problem. One of the goal of real-time tessellation on hardware is to smooth the model according to the level of detail. GPU tessellation implements on shaders which allows multiple advanced accesses, such as the distance between camera and the mesh. If camera changes position or rotation which happens a lot in interactive applications, shaders can decide to tessellate the closer meshes with higher level and lower level for further ones at the same time. This kind of

tessellation saves resources. In contrast, CPU tessellation only allows pre-computation which is not efficient.

### Ambient Occlusion

The first obvious problem is that HBAO+ implementation on WebGL gives a performance improvement but does not provide a very good visual quality. As mentioned before, some APIs like array texture, draw buffers, multi-color attachment and texels are not available on browsers without D3D11 extensions. WebGL implementation maintains the core ideas of HBAO+, however, with the limitations of the current APIs, the reduced quality is reasonable.

The second problem is the extensions have to be enabled on browsers to run HBAO+. It has already been inconvenient to require user to run application only working on Chrome or Firefox, enabling extensions makes matters worse. The current browsers cannot automatically enable extensions, thus, it will cause much trouble for general users to enable extensions themselves.

The third problem is that both HBAO and HBAO+ will introduce noise effect occasionally. This is not a consistent problem, it sometimes happens when access the web for the first time, but the noise will often disappear after refreshing the page. Other than that, when camera zooms out, the scene in the far side will also give minor noises. In HBAO+, the noise is more apparent when objects are far from camera. This effect only happens in WebGL and SSAO is not affected. This problem might be in the depth extraction since it is highly related to the distance between objects and camera. It also can

be related to the radius conversion since in HBAO and HBAO+ screen-space radius will be converted into world-space radius where decimal precision might affect.

### Other Problems

SSAO Quality on WebGL is quite discrete at the beginning due to the sampling randomness. After doing some research on SSAO quality improvement, a sample centralization algorithm is implemented [50]. It increases the quality and surprisingly increases the performance at the same time.

An interesting and strange exploration is that SSAO with 64 samples runs faster than 16 samples on OpenGL.

### Bottlenecks

Tessellation and smoothing improves visual quality most of the time, however, it is not always good or necessary to do this. A scene generally contains lots of different type of meshes, yet global tessellation cannot distinguish what to and what not to. It might be good to smooth an animal's mesh, but same process will change a cube into a sphere which is not intended. A method to let hardware decide which mesh to tessellate should be good, but this might be hard for current generation to do in real-time.

Real-time tessellation is a huge step for real-time rendering, however there are still some features or functions missing currently. Although with shaders programmers are able to apply different level of details when tessellating the scene, the LOD is limited to the distance at current state of art. If tessellation can work on more detailed and specific level based on the importance of the area or characteristic features, it will benefit

a lot on interactive applications and games. For instance, when rendering a human face, higher-level tessellation will be applied on important or animation-focused parts like mouth, eyes etc. Other parts will be at a lower level since they will have high possibility to be ignored during the gameplay or interaction.

HBAO+ is currently the best and most widely used AO algorithms for games or other real-time rendering applications. However, since it is based in image-space, HBAO+ still has the common problems happens in other image-space algorithms.

Image-space algorithms generally will miss some information for occlusions. Missing information will often cause over-occlusions or under-occlusions. Over-occlusions often happen on overlaid objects: front object will cast unnecessary occlusions on behind objects. Under-occlusion usually happens on objects of great length in Z-direction, for instance if a car faces the camera, some of the soft shadows on the ground will be missed under the car. These problems are usually minimized by changing the radius. However, higher radius will counter under-occlusions but give over-occlusions at the same time, lower radius is the opposite. So finding a ‘golden’ radius setting is necessary but difficult.

Generally speaking, the biggest bottleneck on WebGL is the limitation of APIs. WebGL does not include tessellation shaders, so hardware tessellation for real-time performance and dynamic level of details is not available or cannot be implemented as easy as on OpenGL 4.0. Other features mentioned previously are not available as well, thus HBAO+ can only be done in a similar way with reduced visual output quality.

The nature of web application is also a concern for real-time application. Web application [51] aims to be fast, easy-accessible and generally no installation required etc.

This is definitely beneficial and cost-effective while it also means web application has to be adaptive to different machines, different browsers and even different users. A web-based multiplayer first person shooting game is possibly being played by a professional developer with a Nvidia GTX 1080 desktop on Chrome and at the same time another user is trying to access it on a tablet's preinstalled browser. How to maintain the benefits of a web application without increasing the minimum requirement or reducing the quality is a bottleneck for web-based games or interactive applications.

### Possible Solutions

This part mainly focuses on how to possibly counter problems or improve the outputs. The ultimate goal of all the possible solutions is to suggest ways to better implement algorithms mentioned before in a real-time web application.

#### Tessellation

The straightforward way is obviously to wait for API updates. Alternatively, tags or layers can be added on all the meshes for categorization. Objects are tagged with different priorities will be given different tessellation level, or even different smoothing algorithms. Alternatively, meshes are layered as background, environment, character etc and each layer uses a unique tessellation.

#### Ambient Occlusion

The result suggests that the quality-performance tradeoff of WebGL HBAO+ does not make it better than other two algorithms in this implementation due to the limitations. Thus, it might not be a good idea currently to choose HBAO+ over other methods for web application.

Due to the constraints and other API-level differences, improving visual quality of HBAO+ might need more effort. Meanwhile, the implementation to some extent diminish some advantages of web since it uses extensions and advanced settings which are not easily accessible for common users. It also requires a D3D11 supported browser and decent hardware, so the most reasonable solution is to improve SSAO or HBAO on WebGL.

Tweaking parameters is a good practice in general, different settings might suit for different situations. However, a global or intelligent solution is always better but hard to achieve at the time. Based on the results, the first thing to look at is how to generate better samples for these algorithms. Applying a sample centralization in SSAO dramatically improves the visual quality (Figure. 8).

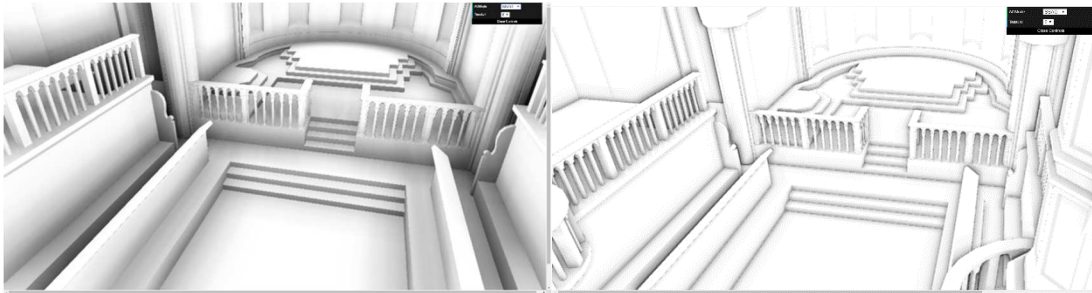


Figure. 9 Before centralized VS After

Additionally, HBAO+ basically improves the sample accuracy and effectiveness via de-interleave texturing which results a better performance than HBAO. Thus, it is a good idea to further research on sampling and sample selection methods [52]. An idea of dynamic samples which is similar to LOD in hardware tessellation might work as well. In lieu of maintaining one sample kernel, maintaining multiple kernels with different sample size and utilize them according to the depth.



Another way is to use multi-view or multi-layer [53] together with multiple parameter settings. As the result shows, the frame rate exceeds 60 even on WebGL which means there are spare executing space to potentially spend. It is also an interesting idea to combine AO algorithms on WebGL, choose HBAO for better quality while SSAO for better performance and balance the power.

To improve the overall performance, JavaScript optimization is also a crucial and practical part in the implementation which is not emphasized in this research.

## CHAPTER 7

### CONCLUSION AND FUTURE

#### Conclusion

In this research, meshes are loaded, tessellated, smoothed and real-time rendered with three Image-Space Ambient Occlusion algorithms on both OpenGL and WebGL. OpenGL is implemented as a standard for WebGL to refer to. The main task is to revamp all the algorithms on WebGL and compare both performance and quality with OpenGL. The goal is to figure out the power and potential of real-time rendering in WebGL by repeating the pipeline as much as we could and also discuss the bottlenecks and possible solutions at current state of art.

WebGL is a relatively new platform deriving from OpenGL ES. It is very useful, lightweight and fast for rendering graphics on web. In this research, WebGL is able to render ambient occlusion on large high-poly meshes and tessellated low-poly meshes at real time. It provides good quality and performance on real-time SSAO and HBAO, it also gives good results in tessellation and smoothing. Although a better hardware is required for better performance, the application runs smoothly on current-gen machines and maintains at least 30 frame per second. Both performance and quality results show that current WebGL version has the potential to render more complex scene for interactive applications or games.

However, WebGL 1.0 limits the hardware tessellation for real-time adaptive smoothing. Without tessellation shader, WebGL's tessellation and smoothing is a pre-processed and static when rendering, thus WebGL tessellation is not real-time and the

level cap is much lower than OpenGL. In other words, tessellation on WebGL is generally convert low-poly meshes with smoothing algorithms into relative high-poly ones. The quality will not be affected much but the performance and diversity is no way near hardware tessellation.

Meanwhile WebGL APIs do not fully support porting HBAO+ from OpenGL to WebGL especially in the de-interleaved texturing stage. During the research, in order to revamp as much as possible, many APIs on OpenGL are replaced by available ones on WebGL, even though this causes some inconsistency and imprecision which results in worse visual quality. On the other hand, HBAO+ has a better performance than HBAO which proves the implementation is successful to some extent and the potential of WebGL is promising.

In conclusion, although some constraints limit WebGL capability and functionality to render real-time applications comparing to OpenGL or other advanced graphics platforms, it is still well-performed and more suitable for relatively small applications like simulators, demonstrations or mini-games. In this research, tessellation and Ambient Occlusion algorithms which are proved to work well on OpenGL, provide decent performance and quality on WebGL as well.

## Future

### Tessellation

The future of Tessellation is to fully utilize the advantage of GPU. Besides the idea of level of detail, some more advanced and complex feature selection ideas are worth digging into. H. Schafer et.al. described a dynamic feature-adaptive subdivision

method in 2015[54] which is suitable for hardware tessellation and theoretically can improve the performance and efficiency.

## AO

Ambient Occlusion is one of Nvidia's ShadowWorks main topics. They improve HBAO+ by adding multiple layers with dual resolutions and creates HBAO+ Ultra for better quality [55]. It was introduced in GDC 2016 and has already been in some AAA games. Moreover, they start to look into world-space than image-space as the power of GPU increases. They introduced VXAO in GDC 2016, a world-space ambient occlusion method [56] produces much better quality with a slower performance. This world-space method has the spatial awareness as well as the locality which are lack in image-space methods. It is integrated into advanced game engine Unreal 4 and used in Rise of Tomb Raider. It has a lot of potential for future AO development.

## WebGL and Web Applications

WebGL 2.0 has been released recently in an unstable state and available in some versions of browsers. It is based on OpenGL ES 3.0 and granted some new features [57]. Multiple render targets, texels etc are in 2.0 version while array texture is still missing. These new features will benefit the improvement in HBAO+ implementation, however, WebGL 2.0 is at an experimental stage which is not fully supported.

Tessellation shader is not yet in OpenGL ES, so it might longer time to fully utilize it in WebGL. The better practice is to study alternative ways to do faster tessellation for web applications.

Virtual reality is very popular nowadays. A website with VR support should be very exciting. There are some available APIs aiming to provide VR support on website [58]. Since VR is a 3D technology, the algorithms used in this research will be useful in many ways. How to integrate these methods respecting VR applications' characteristics is a topic of great value in the future.

## REFERENCES

1. "Tessellation." *Tessellation - OpenGL Wiki*. N.p., n.d. Web. 12 Mar. 2017
2. "Ambient occlusion." *Wikipedia*. Wikimedia Foundation, 15 Dec. 2016. Web. 12 Mar. 2017.
3. Group, Khronos. "The Industry's Foundation for High Performance Graphics." *OpenGL.org*. N.p., n.d. Web. 12 Mar. 2017.
4. "WebGL." *Mozilla Developer Network*. N.p., n.d. Web. 12 Mar. 2017
5. "OpenGL ES 2\_X - The Standard for Embedded Accelerated 3D Graphics." *The Khronos Group*. N.p., n.d. Web. 12 Mar. 2017.
6. Miller, Gavin (1994). Efficient algorithms for local and global accessibility shading. Proceedings of the 21st annual conference on Computer graphics and interactive techniques. pp. 319–326.
7. Pharr, M., Green, S. Ambient Occlusion. GPU Gems, Chapter 17.
8. Hayden Landis. Production-ready global illumination. In SIGGRAPH 2002 Course Note #16: RenderMan in Production, pages 87–102, 2002.
9. Seymour, M(2011). Ben Snow: the evolution of ILM's lighting tools. (2014, October 07). Retrieved February 12, 2017, from <https://www.fxguide.com/featured/ben-snow-the-evolution-of-ilm-lighting-tools/>
10. Ambient Occlusion Awards. (n.d.). Retrieved February 12, 2017, from <http://www.ilm.com/awards/ambient-occlusion-awards>
11. HOBEROCK J., JIA Y.: High-Quality Ambient Occlusion. Addison-Wesley Professional, 2007, ch. 12
12. Bunnell, M. Dynamic Ambient Occlusion and Indirect Lighting. GPU Gems 2, Chapter 14. Retrieved February 12, 2017, from [http://http.developer.nvidia.com/GPUGems2/gpugems2\\_chapter14.html](http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter14.html)
13. Akenine-Möller, T., Haines, E., & Hoffman, N. (2010). Dynamic Computation of Ambient Occlusion. In Real-time rendering(pp. 381). Wellesley, MA: Peters.
14. Akenine-Möller, T., Haines, E., & Hoffman, N. (2010). Dynamic Computation of Ambient Occlusion. In Real-time rendering(pp. 382). Wellesley, MA: Peters.

15. Akenine-Möller, T., Haines, E., & Hoffman, N. (2010). Dynamic Computation of Ambient Occlusion. In *Real-time rendering*(pp. 382). Wellesley, MA: Peters.
16. Dachsbacher, C. (2009). Global Illumination Effects. In *ShaderX 7: advanced rendering techniques*(pp. 411-412). Boston, MA: Course Technology.
17. Sainz, Miguel.(2008). "Real-Time Depth Buffer Based Ambient Occlusion" Presentation.
18. Luft, T., Colding, C., & Deussen, O. (2006). Image enhancement by unsharp masking the depth buffer. *ACM SIGGRAPH 2006 Papers on - SIGGRAPH '06*. doi:10.1145/1179352.1142016
19. Shanmugam, P., & Arikian, O. (2007). Hardware accelerated ambient occlusion techniques on GPUs. *Proceedings of the 2007 symposium on Interactive 3D graphics and games - I3D '07*. doi:10.1145/1230100.1230113
20. Akenine-Möller, T., Haines, E., & Hoffman, N. (2010). Dynamic Computation of Ambient Occlusion. In *Real-time rendering*(pp. 385). Wellesley, MA: Peters.
21. Bavoil, L., Sainz, M., & Dimitrov, R. (2008). Image-space horizon-based ambient occlusion. *ACM SIGGRAPH 2008 talks on - SIGGRAPH '08*. doi:10.1145/1401032.1401061
22. Bavoil, L & Andersson, J. (2012). "Stable SSAO In Battlefield 3 With Selective Temporal Filtering".Presentation.
23. McGuire, M., Mara, M., Luebke, D. (2012) Scalable Ambient Obscurance. *High Performance Graphics (2012)*
24. NVIDIA. (n.d.). HBAO+: Horizon-Based Ambient Occlusion and Ambient Occlusion (AO). Retrieved February 13, 2017, from <http://www.geforce.com/hardware/technology/hbao-plus>
25. Catmull, E., & Clark, J. (1978). Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design*,10(6), 350-355. doi:10.1016/0010-4485(78)90110-0
26. Segal, M., Akeley, K., Frazier, C., Leech, J., & Brown, P. (2010). *The OpenGL R Graphics System: A Specification (Version 4.0 (Core Profile) )*.
27. Microsoft. Tessellation Stages. Retrieved February 13, 2017, from [https://msdn.microsoft.com/en-us/library/ff476340\(v=VS.85\).aspx](https://msdn.microsoft.com/en-us/library/ff476340(v=VS.85).aspx)
28. Tariq, S. (2009). "D3D11 Tessellation". Presentation

29. Vlachos, A., Peters, J., Boyd, C., & Mitchell, J. L. (2001). Curved PN triangles. Proceedings of the 2001 symposium on Interactive 3D graphics - SI3D '01. doi:10.1145/364338.364387
30. McDonald, J. (2011). "Tessellation on Any Budget". Presentation
31. Kessenich, J. M., Sellers, G., & Shreiner, D. (2017). OpenGL® programming guide: the official guide to learning OpenGL®, version 4.5 with SPIR-V. Boston (Mass.): Addison-Wesley.
32. The Khronos Group. The Industry's Foundation for High Performance Graphics. Retrieved February 13, 2017, from [https://www.opengl.org/documentation/current\\_version](https://www.opengl.org/documentation/current_version)
33. Parisi, T. (2014). Programming 3D applications with HTML5 and WebGL. Beijing: O'Reilly.
34. Three.js - Javascript 3D library. Retrieved February 13, 2017, from <https://threejs.org/>
35. The Khronos Group. WebGL - OpenGL ES 2.0 for the Web. Retrieved February 13, 2017, from <https://www.khronos.org/webgl/>
36. WebGL 2.0 Samples. (n.d.). Retrieved February 13, 2017, from <http://webglSamples.org/WebGL2Samples/>
37. Kajalin, V. (2009). Screen-Space Ambient Occlusion. In ShaderX 7: advanced rendering techniques(pp. 413-424). Boston, MA: Course Technology.
38. Perlin, K., & Hoffert, E. M. (1989). Hypertexture. ACM SIGGRAPH Computer Graphics,23(3), 253-262. doi:10.1145/74334.74359
39. Bavoil, L., & Sainz, M. (2009). Image-Space Horizon-Based Ambient Occlusion. In ShaderX 7: advanced rendering techniques(pp. 425-444). Boston, MA: Course Technology.
40. NVIDIA . (n.d.). HBAO Technology . Retrieved February 13, 2017, from <http://www.geforce.com/hardware/technology/hbao-plus/technology>
41. Bavoil, Louis & Jansen, J. (2013). "Particle Shadows & Cache-Efficient Post-Processing". Presentation
42. The Khronos Group . (n.d.). Tessellation. Retrieved February 13, 2017, from <https://www.khronos.org/opengl/wiki/Tessellation>



43. OGLdev. (n.d.). Basic Tessellation. Retrieved February 13, 2017, from <http://ogldev.atspace.co.uk/www/tutorial30/tutorial30.html>
44. Vlachos, A., Peters, J., Boyd, C., & Mitchell, J. L. (2001). Curved PN triangles. Proceedings of the 2001 symposium on Interactive 3D graphics - SI3D '01. doi:10.1145/364338.364387
45. deanm. (2011, March 26). deanm/pregl. Retrieved February 13, 2017, from <https://github.com/deanm/pregl>
46. Ignac, M. (n.d.). Marcin Ignac : SSAO. Retrieved February 13, 2017, from <http://marcinignac.com/experiments/ssao/>
47. frenchtoast747. (2014, October 26). frenchtoast747/webgl-obj-loader. Retrieved February 13, 2017, from <https://github.com/frenchtoast747/webgl-obj-loader>
48. kivy. (n.d.). Monkey mesh. Retrieved February 13, 2017, from <https://github.com/kivy/kivy/blob/master/examples/3Drendering/monkey.obj>
49. McGuire, Computer Graphics Archive, accessed 2016 Jun 28. <http://graphics.cs.williams.edu/data>
50. Chapman, J. (n.d.). SSAO Tutorial. Retrieved February 13, 2017, from <http://john-chapman-graphics.blogspot.com/2013/01/ssao-tutorial.html>
51. Magic Web Solutions. (n.d.). The benefits of web-based applications. Retrieved February 13, 2017, from <http://www.magicwebsolutions.co.uk/blog/the-benefits-of-web-based-applications.htm>
52. Holden, D., Saito, J., & Komura, T. (2016). Neural network ambient occlusion. SIGGRAPH ASIA 2016 Technical Briefs on - SA '16. doi:10.1145/3005358.3005387
53. Bavoil, L., & Sainz, M. (2009). Multi-layer dual-resolution screen-space ambient occlusion. SIGGRAPH 2009: Talks on - SIGGRAPH '09. doi:10.1145/1597990.1598035
54. Schäfer, H., Raab, J., Keinert, B., Meyer, M., Stamminger, M., & Nießner, M. (2015). Dynamic feature-adaptive subdivision. Proceedings of the 19th Symposium on Interactive 3D Graphics and Games - i3D '15. doi:10.1145/2699276.2699282
55. Tatarinov, A & Panteleev, A. (2016). “Advanced Ambient occlusion Methods for Modern Games”. Presentation

56. Penmatsa, R., Nichols, G., & Wyman, C. (2010). Voxel-space ambient occlusion. Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games - I3D 10. doi:10.1145/1730804.1730989
57. The Khronos Group. (n.d.). WebGL 2 Specification. Retrieved February 13, 2017, from <https://www.khronos.org/registry/webgl/specs/latest/2.0/>
58. WebVR. (n.d.). Retrieved February 13, 2017, from <https://webvr.info/>

APPENDIX A  
CONTRIBUTIONS

The completed research contains DEMO executables, websites and videos which can be found at:

OpenGL implementations: <https://github.com/RadiumP/Tessellation>

WebGL implementations: <https://github.com/RadiumP/WebAO>

SSAO+HBAO WebGL website: <https://radiump.github.io/WebGL/hbao.html>

Videos:

WebGL Tessellation + Ambient Occlusions: [https://youtu.be/pF2im\\_08fKo](https://youtu.be/pF2im_08fKo)

WebGL AO: [https://youtu.be/Rf\\_AQfarBMo](https://youtu.be/Rf_AQfarBMo)

OpenGL Tessellation + Ambient Occlusions: [https://youtu.be/BsRYSHFbE\\_E](https://youtu.be/BsRYSHFbE_E)

OpenGL AO: <https://youtu.be/pdYwSzCpDIU>

OpenGL vs. WebGL AO: <https://youtu.be/SwW-Qp9HCr4>

OpenGL vs. WebGL Tessellation + AO: <https://youtu.be/kD9H117YiSg>

Note that HBAO+ has limited support on browsers now, so it is not included in the website.