TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Computer Science
Chair of General Informatics

Analytic Layered Anti-Aliasing

Bachelor's Thesis

Student:	Georgi Gerassimov
Student code:	123682
Supervisor:	Marko Kääramees

Tallinn
2015

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

(*kuupäev*) (*allkiri*)

Annotatsioon

Aliasing on veel lahendamata probleem arvutigraafikas. Välja on pakutud suur hulk algoritme, kuid enamik neist kas lahendavad probleemi osaliselt või on välja mõeldud spetsiifilisele sisendile ja ei tööta erijuhtudel, näiteks ristumise puhul.

Käesolevas töös palutakse välja analüütilise *anti-aliasingu* rasterdamise algoritmi, mis töötab läbipaistvate objektidega ning lahendab kattuvuse ning ristumise erijuhud. Algoritm kasutab standardset graafika rasterdamise *pipeline-i*. Ta lahendab nähtavuse probleemi kasutades analüütilist mitmetasandilist lähenemisviisi, mis võimaldab teha arvutusi paralleelselt. Algoritmi võrreldakse standardse *anti-aliasing* tehnikaga.

Tulemused näitavad, et analüütilised meetodid annavad parema pildi kui *brute-force sampling* meetodid, mis püüavad lihtsalt suurendada diskreetimissagedust. Analüütiliste meetodite piiranguks on ujukomaarvutuste täpsus ja ajamahukus.

Abstract

Aliasing is still an unresolved problem in computer graphics. A huge number of algorithms have been proposed, however, most of them either partially solve the issue or they are designed for a particular input and do not handle special cases, like intersections.

This work proposes an algorithm for analytical *anti-aliasing rasterization*, which handles transparent objects as well as primitive overlapping and intersections. The algorithm is an extended version of a standard graphics pipeline adopted for analytic *anti-aliasing*. It solves *the visibility problem* using an analytical layered approach, allowing to do calculations in separate threads. The algorithm is compared with standard *sampling anti-aliasing* technique.

Results show that an analytic anti-aliasing has better quality than brute-force sampling methods, which try to increase the sampling rate. However, analytic methods are still limited to computer precision and cost a lot of computational time.

Table of Contents

1. Introduction

1.1 Aliasing in computer graphics

Modern monitors are able to display only raster images, since they consist of pixels. Raster image (Fig. 2) is basically just a set of finite fragments, from which the final image is constructed. While raster images lack quality, they are faster and simpler to render. [1] In contrast to raster, vector image (Fig. 2) holds original shapes, usually described as some equations. In terms of the raster image, that would mean infinite amount of fragments. This allows rendering shapes correctly at any scaling, whereas raster images will lose quality when zooming or transforming them. Moreover, in computer graphics, 3D models are usually represented as vector information. That way they occupy less disk space and also they can be rendered at any scaling without losing precision.

To be able to display vector images, they have to be transformed into raster images. The process of transforming vector data to raster is called *rasterization* (this term also refers to rendering algorithm, described later). This is *discretization* process, since continuous data is sampled and converted into discrete. Data is usually sampled at different rates, depending on screen resolution. Sampling means taking value from certain image point, which usually corresponds to a pixel center location.

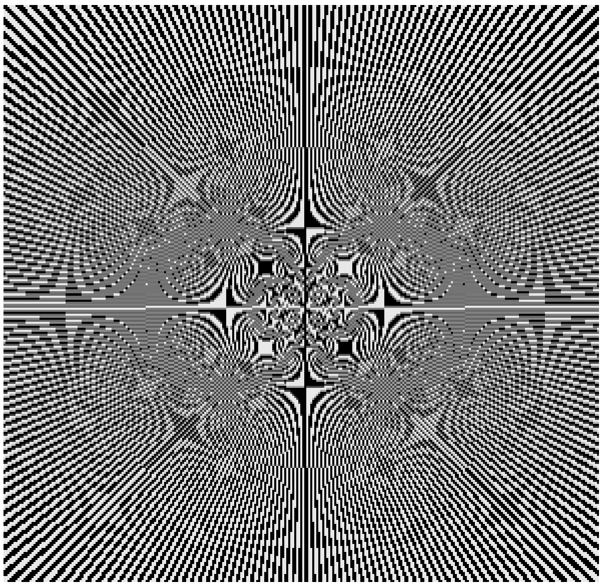


Figure 1 (Moiré pattern)

However, *discretization* has its consequences. If the signal is sampled at frequencies lower than $f_s/2 > f$, which is called the Nyquist frequency [2], distortions, inaccuracies and artifacts can occur, like Moiré patterns (Fig. 1), jagged or pixelated edges (Fig. 2 leftmost). Such effect is called aliasing. [3]. This happens when point sampled value is far different from whole pixel exact area value.

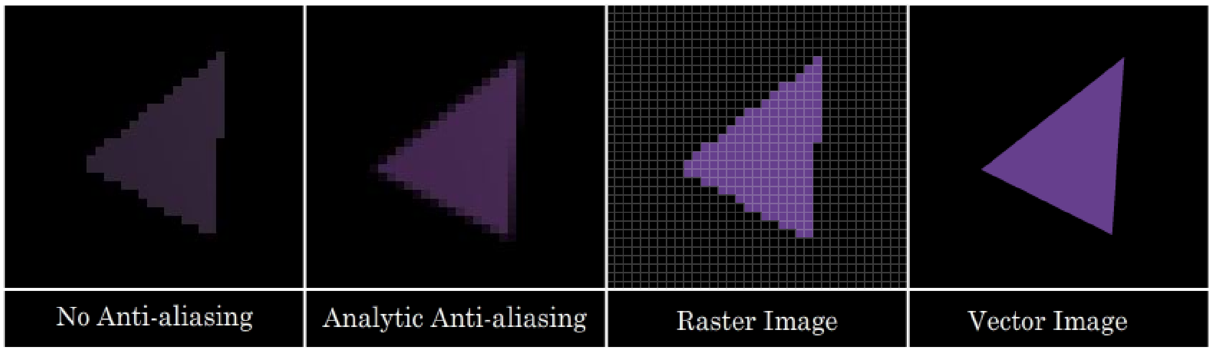


Figure 2 (left: difference between aliased & anti-aliased image, right: difference between raster & vector images)

1.2 Related work

Different anti-aliasing techniques were proposed to solve this problem. Modern sampling algorithms could be divided into two main types: pre-processing and post-processing, which solve the issue before rendering and after respectively. Post-processing methods try to detect edges after rendering, by comparing pixel neighbor values and averaging them to soften edges or applying post-filters. [4] Though this method is fast and parallel, it tends to blur the overall image and it does not solve inaccuracy problem. Pre-processing methods try to reduce original signal frequency or increase sampling frequency.

Increasing the sample rate is called *supersampling* or Full-Scene Anti-Aliasing (FSAA). [5] Instead of one sample per pixel it takes more and averages result by combining values. $P_{total} = \sum_{i=0}^{n} \frac{p_i}{n}$ Rendering resolution could be 2x, 4x, 8x or 16x higher. All pre-filtering approaches inherit *supersampling* idea, trying to approximate and optimize algorithms, for example via compressing rendering buffer [6]. *Supersampling* is still considered to be the best quality sampling anti-aliasing and it is being used for high-quality rendering [7].

Rather than sampling a signal, analytic methods try to calculate exact area, solving the convolution of filter kernel *W* and continuous image *I* for each pixel at position *(i, j)*.

$$P(i,j) = \iint I(i + x, j + y) W(x,y) dxdy$$

That just means to find exact area, which certain polygon occupies inside pixel area. Sampling is approximation of this convolution. Since the signal is not actually linearly distributed [2] different filters are used for value estimation. Filters simply tell which part of the area has more weight. Many different analytic methods were proposed and implemented, for instance methods using scanlines [8], calculating actual volume [9], improving filter usage [10], special algorithms for terrain rendering [11] or half-analytic approaches like line-sampling [12]. Not all of algorithms are mentioned, however, from those, which try to achieve the best quality, most are quite limited and they render only certain geometry or do not handle intersections and transparency.

When rendering 3D scenes, anti-aliasing always comes along with so called “hidden surface elimination problem”, to determine which parts of 3D geometry are actually visible. Many early methods tried to just sort polygons in correct order [13]. More complex solutions were found later, like clipping polygons before sorting [14], using a grid to sort them [15], solving visibility issues for each pixel [16]. Further algorithms were optimized to solve only actually overlapping polygons [17], decomposing polygon methods were used to handle intersections [18], algorithms were optimized and parallelized even more [19]. Algorithms for efficient polygon clipping were also developed to solve this issue [20]. Because of their complexity and computational time they are rarely used in real-time rendering.

In modern graphic cards *z-buffer* algorithm is used [21]. It can be easily combined with sampling technique and it follows the idea of fast approximation algorithms. The main idea of this method is to calculate the distance for each pixel fragment of each polygon and simply discards those, which are further away, taking nearest ones. This distance is saved along with the color in *z-buffer*, where *z* – is the distance from polygon point to view point. Unfortunately, it is not well combinable with analytic methods.

1.3 Standard rendering pipeline overview

In order to display 3D scene on screen, several steps should be made (Fig. 3): [22]

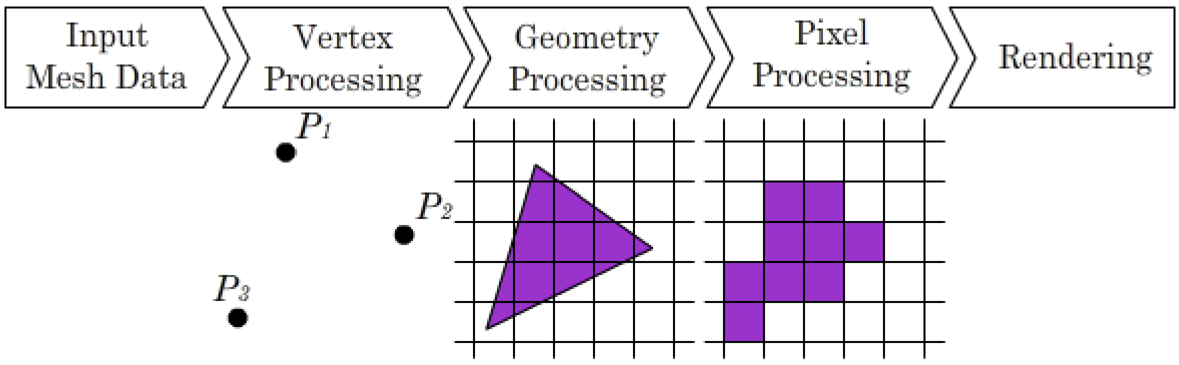


Figure 3 (Diagram of the rendering pipeline)

1. **Input Mesh Data.** Firstly, objects should be defined. 3D Geometry is usually prescribed as a list of vertexes, which form polygons. Polygons could be considered to be built from triangles, since they are simplest polygons, thus easier to render. Those simplest polygons are called primitives.
2. **Vertex Processing.** Then, their position on screen should be found. Objects are declared in so called world space and view position is defined by camera object. Vertexes are transformed from world to viewport and then to window space by multiplying with model-view and projection matrixes. Perspective projection matrix projects points onto screen plane.
3. **Geometry Processing.** Polygons are formed by connecting vertexes. Then vector data is transformed into raster. The program produces a set of pixel fragments for each polygon, linearly interpolating different vertex parameters, like color and *z-values* (distance to screen plane) [23]. When sampling, the program checks whereas point lies within the triangle boundaries or not.
4. **Pixel Processing.** Each pixel fragment receives its color and *z-value*. By simple check, nearest fragment for each pixel is found. This solves the visibility issue, though precision is depending on rendering resolution. Shading (how the object is lit) is computed per pixel. Post-processing also happens on this stage, since the resolution is fixed, applying post-effects is more efficient, than processing the geometry itself.
5. **Rendering.** Lastly, data from the buffer is copied to monitor screen and displayed.

Modern graphic cards render images using *rasterization* technique. As opposed to ray tracing algorithm [24], where pixel iteration is in the outer loop, and primitive iteration in inner, in *rasterization* - pixel iteration is done per primitive. In other words, in ray tracing, program casts a ray from pixel position and then starts looking through all primitives, detecting ray-polygon intersection. In *rasterization*, program projects each polygon onto screen plane, and then starts checking, which pixels are within this polygon area. Since the screen resolution does not change, as opposed to polygon number, *rasterization* is easier to optimize, since for each polygon there will be a fixed amount of pixels to check, using for example scanline method [25]. Moreover, ray intersections have to be performed in 3D space, while in *rasterization* polygons are already mapped to 2D screen plane.

Because of speed and simplicity, this method is widely used in modern computer graphics applications, especially in real-time rendering, where speed is crucial component.

Advantages:

- Fast. The algorithm is parallel, which allows accelerating it by improving hardware. The algorithm is discrete, thus it is easier for computers to work with it.
  In terms of Big-O Notation (a standard way of expressing how efficient an algorithm is, depending on argument size) *rasterization* is *O(n*h*w)*, where *n* is number of primitives, *h* and *w* are respectively height and width of rendering screen. Since screen resolution is constant, it becomes just *O(n)*.
- Requires small amount of memory. Usually it is enough to have only z-buffer and color buffer.

Disadvantages:

- Aliasing.
- When rendering transparent objects, they have to be rendered separately at the end.

2. Analytic layered approach

In this thesis, one of the possible solutions for analytic anti-aliasing is described. It uses standard rendering pipeline, which was previously described, as a basis [26], changing only *rasterization* and visibility solving part, that means that it could be easily combined with other modern techniques. It is assumed that the input consists only of polygons, and then complex polygons are subdivided into triangles.

To further optimize this method, the exact area of each pixel is calculated only if $p < 1$, where p is coverage percent. That means - if triangle covers pixel only partially. For all pixels where all four vertices of pixel box lie within the triangle, $p = 1$ is always true.

In this work simple box filter was used for calculating area convolution, however better filters exist, like gauss or tent (reconstruction) [27]. For each pixel, triangles are clipped by pixel box boundaries. Then the coverage area is calculated. $p = \left|\int_l^r f_1(x)dx - \int_l^r f_2(x)dx\right|$, where l is pixel left edge, r is pixel right edge and f_1, f_2 are equations of lines, which form a triangle [28].

2.1 Hidden surface elimination

To correctly display anti-aliased edges method has to divide polygons, if they are covered by others, and select only visible parts. So, the final image may be considered as a vector image consisting of non-overlapping triangles.

Two ways of achieving that, are described here:

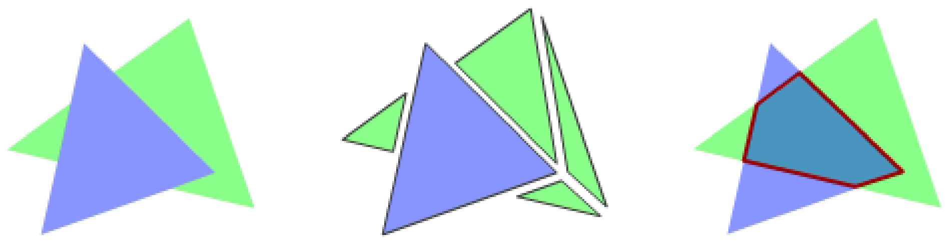


Figure 4 (left: final image; center: method of rendering only actually visible parts; right: method of rendering original triangles and then subtracting invisible parts, unseen part of the green triangle is outlined)

- **Drawing triangles and then subtracting unseen parts.** (Fig. 4 right) Hidden surface elimination is performed after triangles are drawn, that ensures that only if triangles overlap at the same pixel, calculations are performed. After triangles are rendered, parts which are not supposed to be seen are found and subtracted.
  This method is adaptive, since it will use original triangles to solve visibility where possible and only if more than two triangles overlap in the same area complexity will increase (Fig. 5 right).
  The idea of this method is to render original triangles on separate layers, find intersection parts and subtract them from triangles, which are overlapped by others.
  Unfortunately $W = O(2^n)$. Each time when polygon is subtracted, deleted part should also be saved, so it can be re-added next time. (Fig. 5 right) That happens due to the fact that after subtraction original triangles are not changed and intersection part of two subtracting polygons is being subtracted twice. For instance, the addition formula for three triangles will be: $A + B + C - (A \cap B) - (A \cap C) - (B \cap C) + (A \cap B \cap C)$, where A,B and C are representing triangle areas and the last term is exactly the

part, which is re-added. While for opaque objects that does not make any difference, transparent objects will be rendered incorrectly.

Since each new triangle can overlap with already existing ones, it can create the same amount of subtracting polygons. That means: if there were N polygons, there would be $2 * N + 1$ polygons. (+1 is a new triangle itself) This leads to recurrence relation:

$T_n = 2 * T_{n-1} + 1$, where T_n is number of polygons;

characteristic equation for which is: $T_n = 2^n - 1$, which is $O(2^n)$.

- **Drawing only visible parts.** (Fig. 4 center and Fig. 5 left) $O(n^2)$ is for evaluating checks, since it has to be done for all pairs of polygons. That step can be optimized using broad phase, where program finds possibly overlapping polygons using for example axis aligned boxes to represent complex polygons or via mapping polygons to grid. Actual calculations are performed after that in narrow phase.

  However, each time polygon overlaps or intersects it has to be divided. Moreover, some polygons have to be divided in order to maintain only primitive polygons, as working with concave shapes, which can be formed, is harder. Assuming that each new polygon can divide already existing geometry by two or more (including transparent polygons): $W \geq O(2^n)$.

  Hidden surface elimination is done before rendering triangles and only actually visible parts are being rendered.

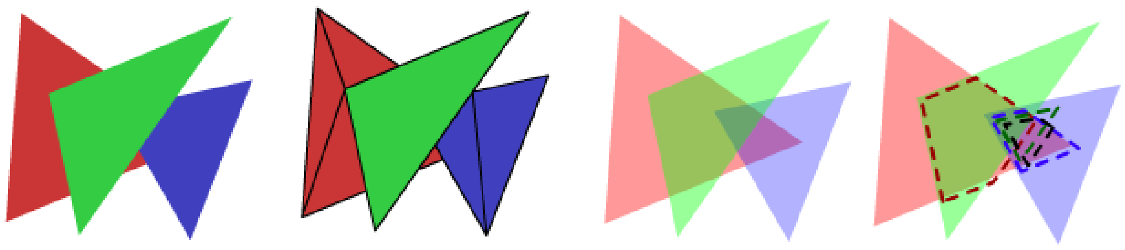


Figure 5 (more complex scenario with three triangles. From left to right: final image; visible parts are outlined; triangles are transparent to show how they overlap; subtractive polygons for those triangles are outlined.)

For this work, the first method was chosen, since it generates overall less geometry, and intersection parts are always going to be convex polygons. It is easier to work with convex polygons, rather than with concave. Moreover, with the second method to actually start rendering polygon, we have to solve the visibility issue first, as opposed to the first method, where rendering one polygon does not interrupt the calculations to find visible parts of the same polygon. Since those actions could be performed in parallel, it could decrease computation time. In addition, the first method is better suited for rendering transparent objects.

To find out overlapping or not overlapping part of each polygon, it has to be cut three times for 2D (Fig. 6) (since the triangle is an area of plane limited by three lines) and four times for 3D to solve intersections (line is formed by the intersection of two triangle planes).

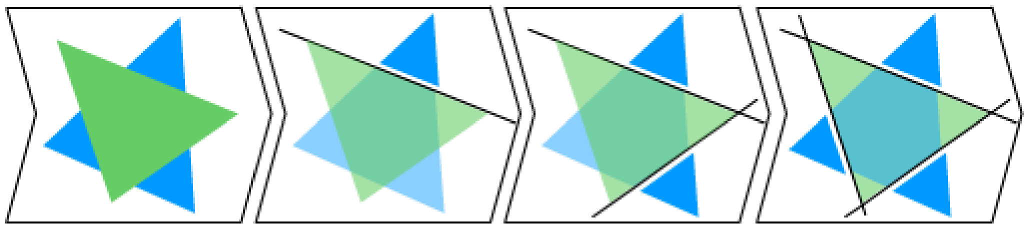


Figure 6 (the process of finding the intersection part: blue triangle is being iteratively cut with lines, which form the green triangle)

Assuming that triangle vertices are declared in winding order, edge function method [29] can be used to determine on which side of the line a certain vertex lies. When the line intersects it forms an intersection point, which belongs to both parts. By connecting all vertexes, which lie on certain side polygons are formed. That way, the triangle is cut into two parts: one, which is outside the cutting triangle and inside one. Outside polygon composes the visible part of cut triangle, which does not intersect with cutting triangle. Inside polygon lies on the other side and intersects with cutting triangle, moreover, it is always going to be a convex polygon. This polygon is the part, which is subtracted afterwards. It is subtracted from triangle which is behind the other one. That can easily be determined knowing plane equations, on which those triangles lie. If planes intersect, intersection line will divide the screen into two regions: where the first triangle is in front of the second one and where the first triangle is covered by the second one.

2.2 Layered rendering

The process of layered *rasterization* is described below. It should be noted that the buffer implementation may be different and only one of possible variants is described here.

Firstly, the original polygon is rasterized into a separate layer. Color values are computed only for original polygons. Whenever a new polygon falls into pixel, already containing another polygon, a candidate polygon is marked into a temporary buffer, to check later whether they overlap or not. Basically, the program finds approximately possibly overlapping polygons with pixel precision. Buffer (Fig. 7) contains all drawn triangle *ID-s* and their coverage percent per pixel. Subtractive triangles are drawn on triangle layer, from which they are subtracted. Only at the end values are combined to get the final color for each pixel. That means, that correct coverage percent for each triangle in pixel is calculated and then values are merged together.

Triangle intersections are solved once per scene. After the hidden part of triangles has been determined, subtractive triangles are drawn exactly the same way as original ones, with exception of the coverage percent sign (-). For a subtractive triangle of the subtractive triangle the sign is reversed (+), so triangle part will be re-added, otherwise, this area will be just subtracted more than once, as previously explained in description of hidden surface elimination algorithm. Triangles and subtractive triangles can be rendered in any order. Even though this approach takes a lot of memory to hold information about each triangle, computations for each primitive are totally independent and can be done in parallel threads. That way part of a program, which is rendering, does not have to wait for other calculations, it can render without pauses. Furthermore, primitive rendering itself could be already optimized using graphic card features, since calculations for each pixel are independent. The final step of combining layers is independent for each pixel as well. However, parallelizing algorithm does not decrease the total number of operations, which have to be performed in order. That means computation time is depending on the maximum amount of triangles, overlapping in one pixel region. And even though complexity is exponential ($O(2^n)$, as explained previously), it is not likely that too many triangles will overlap in one pixel.

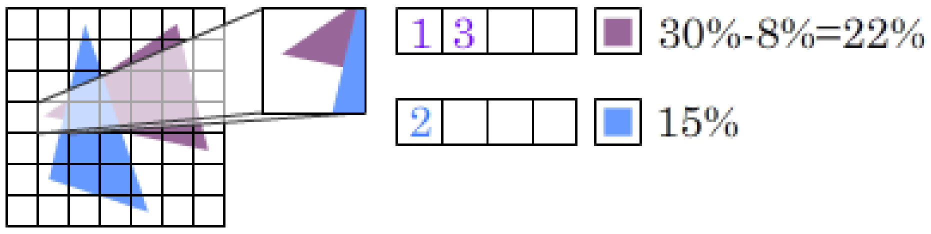


Figure 7 (rendering buffer, each pixel has a dynamic array – a list of triangle or subtractive triangle *ID-s* drawn in that pixel. They all belong to certain layer, which represents an original triangle, from which they are subtracted or to which they are added. When calculating resulting color all values are combined together to find coverage percent of visible part for each triangle)

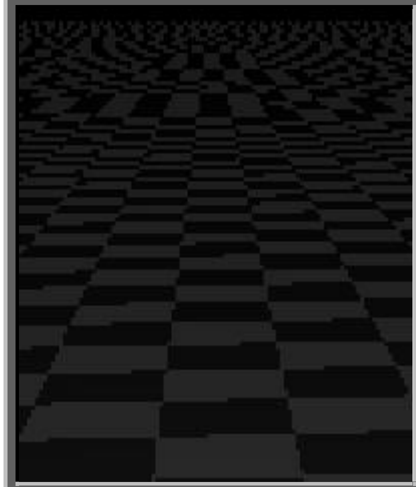
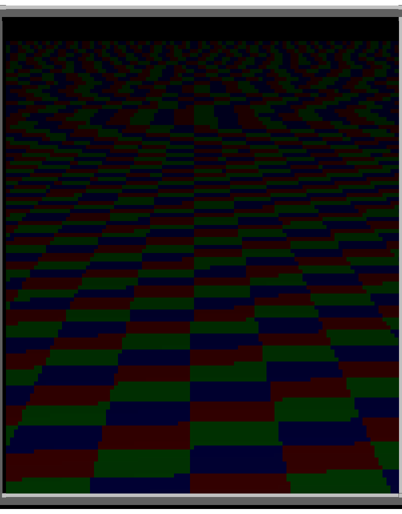
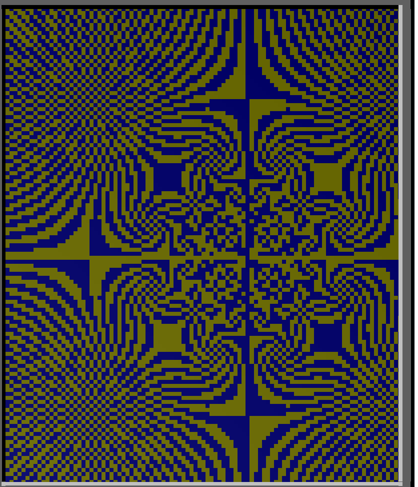
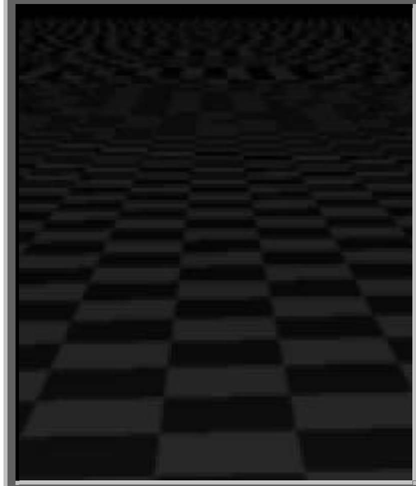
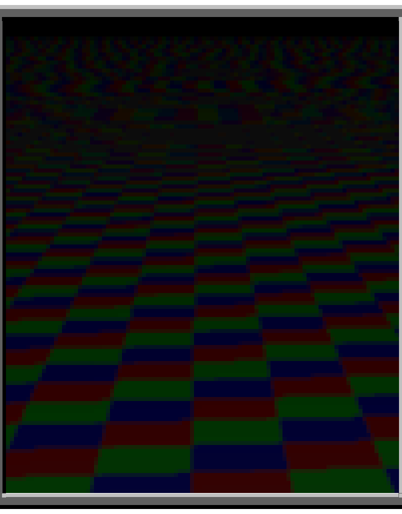
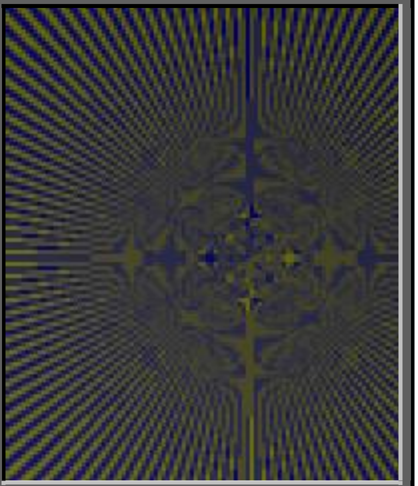
This method supports rendering transparent objects. Transparent triangle will only partially subtract hidden part, depending on transparency degree, which could be possibly calculated for each pixel individually allowing to use more complicated objects. Instead of subtracting whole coverage value, it subtracts less, taking into account transparency.

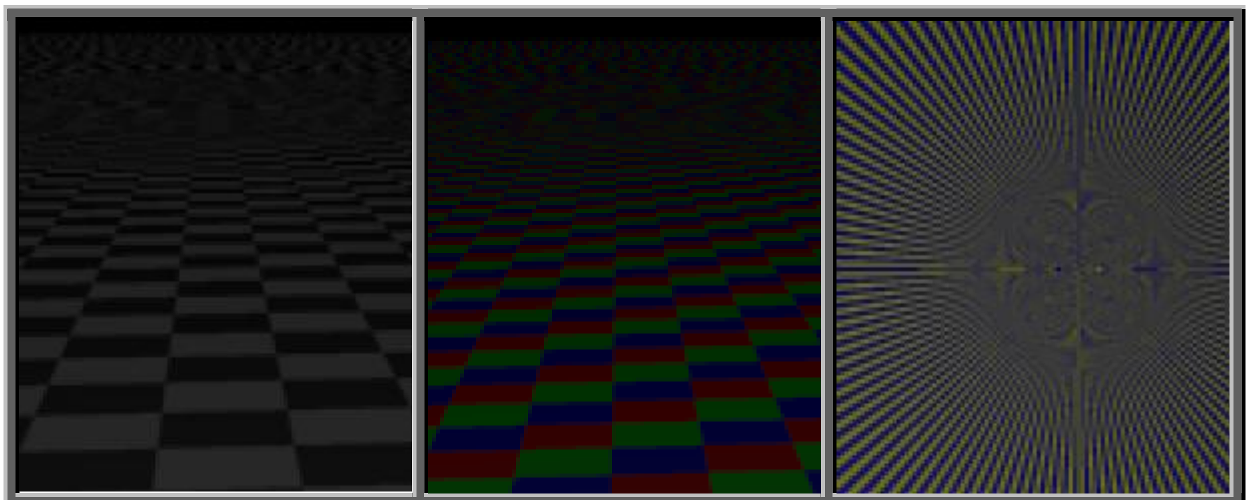
3. Analysis of results

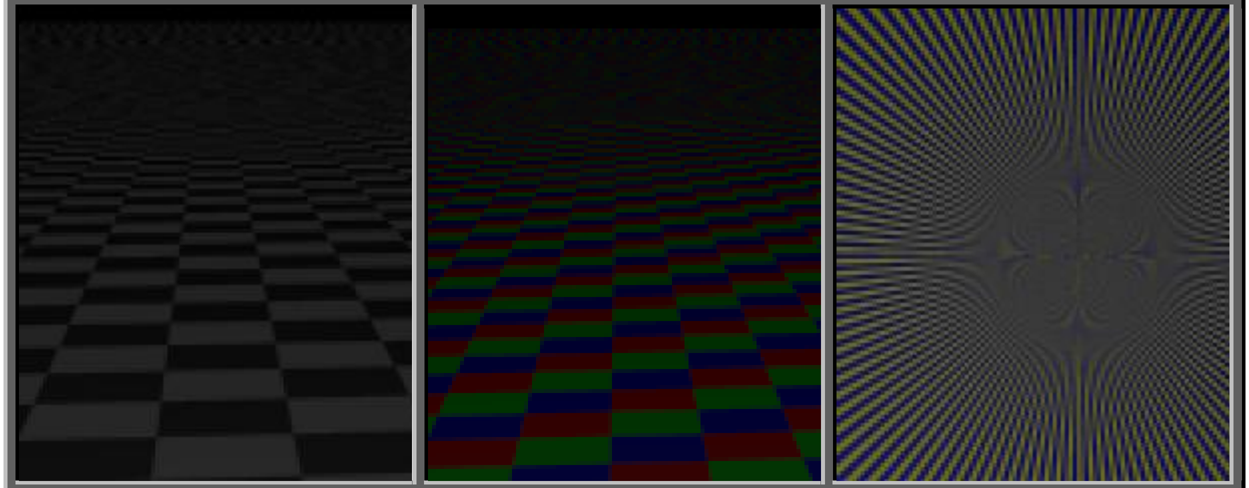
3.1 Comparison

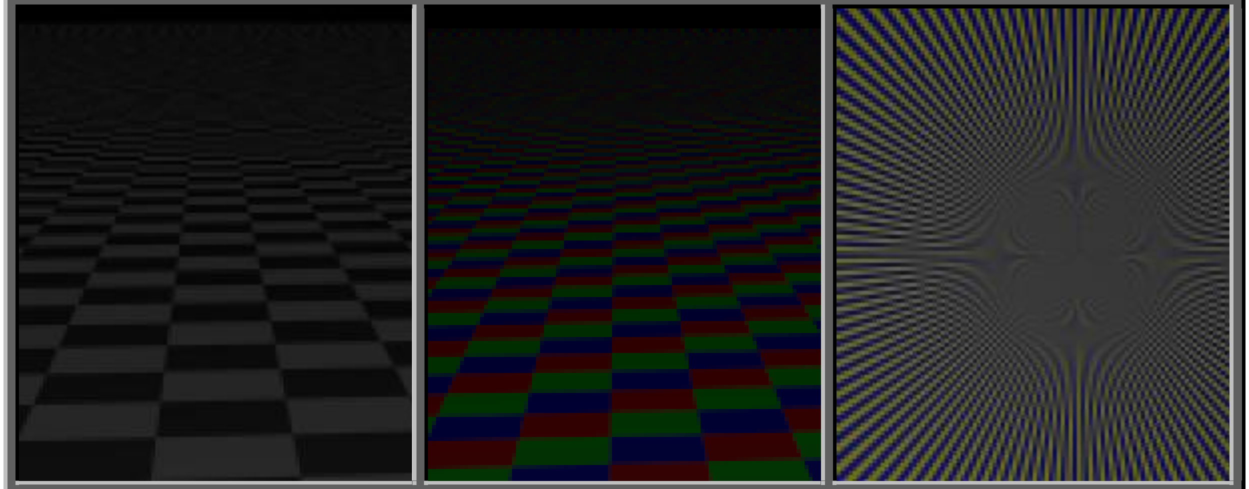
Presented method was compared to multisampling algorithm, since multisampling is considered to give best quality. Since optimization was out of scope of this work, time was not be compared.

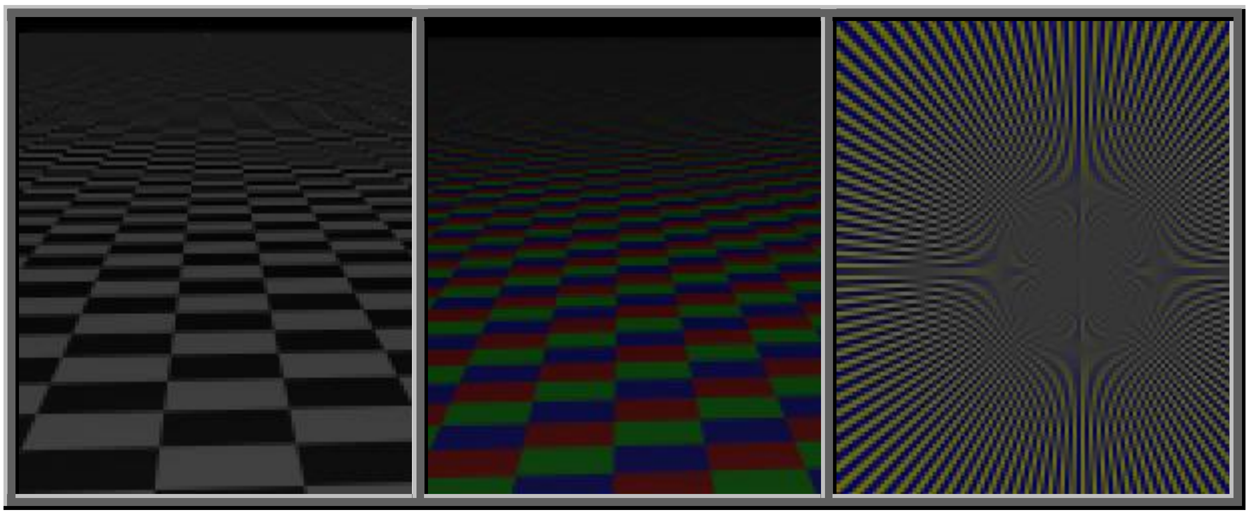
Table 1 (comparison of analytic method and supersampling when rendering high frequency patterns)

No Anti-aliasing		
		
FSAAx2		
		
FSAAx4		


FSAAx8


FSAAx16


Analytic Anti-aliasing



Each chessboard tile was rendered as a separate polygon.

As can be clearly seen on examples (Table 1), by increasing the sampling rate, high frequency signals are sampled better and aliasing disappears, though it still remains for even higher frequencies. (Distortions are moving towards the horizon, while sampling rate is increased). Even with x16 multisampling distortions remain very far away (Table 2). Using the analytical approach those distortions totally disappear. Though they remain for middle-frequencies. Results do not get much better on middle frequencies for multisampling after x4 as well, that is due to the limitation of using a box filter (FSAA can be considered using box filter, since pixel is interpreted as a box).

Table 2 (image was zoomed in to show the difference between analytical and supersampling anti-aliasing at very high frequencies)

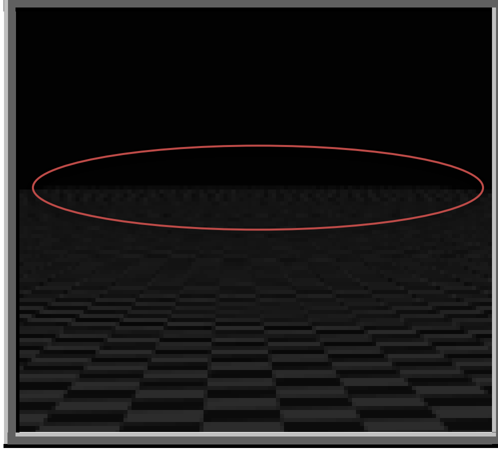
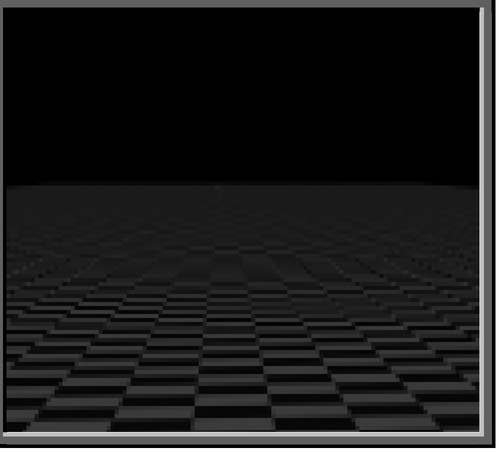
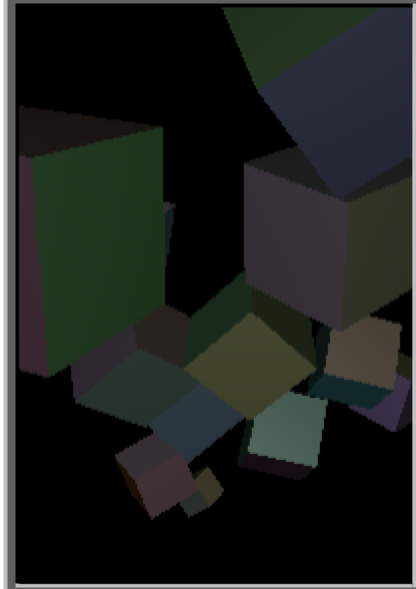
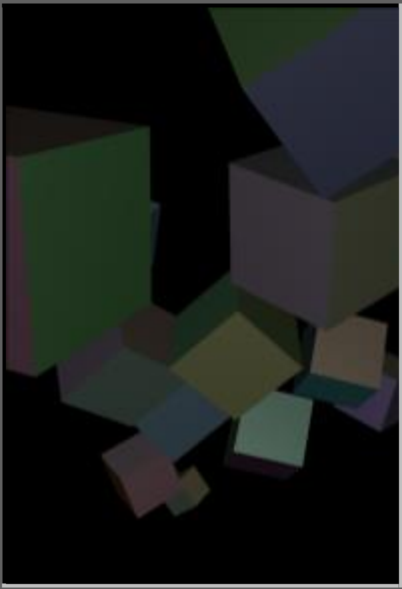
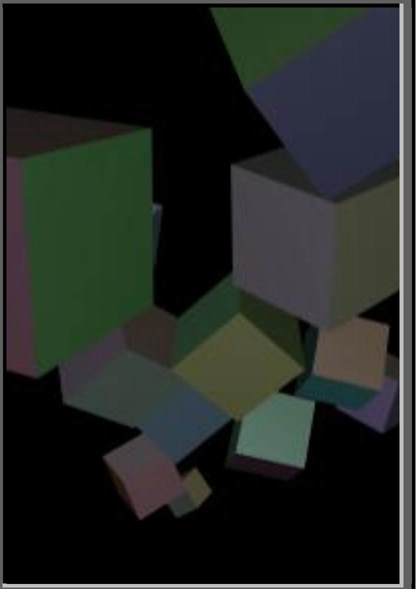
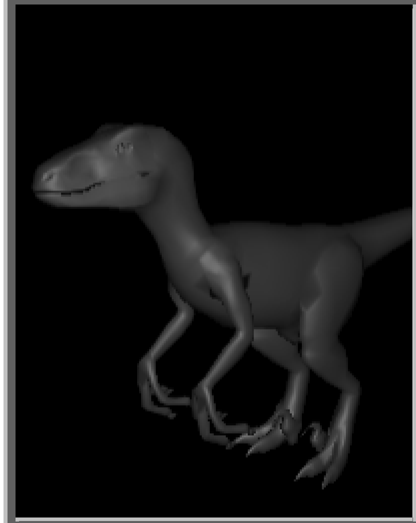
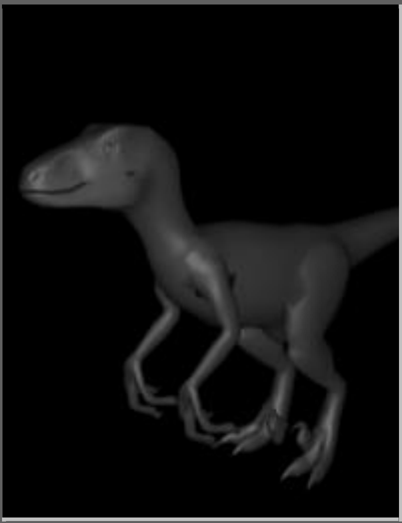
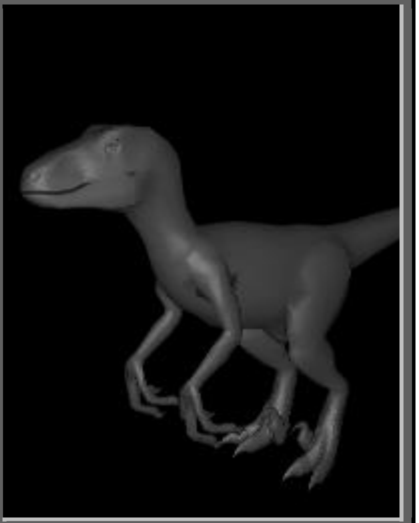
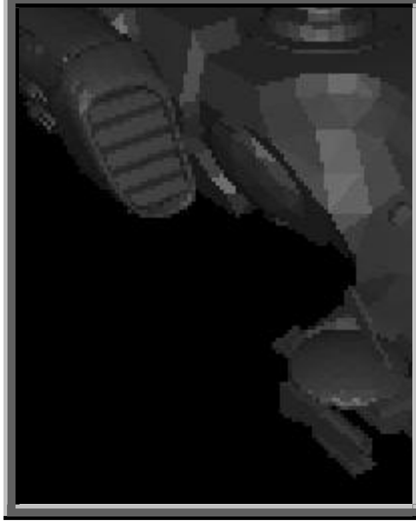
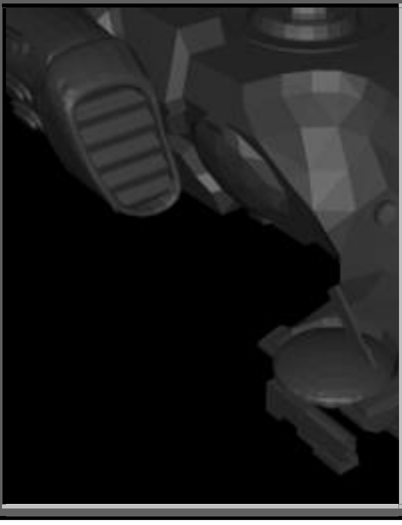
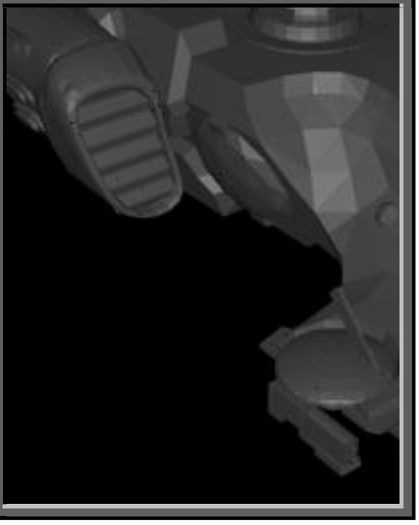
FSAAx16	**Analytic Anti-aliasing**
	

Table 3 (comparison of analytic method and supersampling when rendering 3D models [30])

No Anti-aliasing	FSAAx8	Analytic Anti-aliasing
		
		
		

When rendering low-poly models (Table 3) difference is less noticeable. Since no high frequency patterns exist, there is no aliasing in the final image. Even though analytic approach should be more accurate, final precision is still limited by resolution and those details are not distinguishable. That means, edge anti-aliasing does not require such precision, as opposed to high-frequency patterns.

3.2 Advantages and disadvantages

Pros and cons of the proposed method are presented below.

Advantages:

- Can render transparent (Fig. 9) and opaque (Fig.8) primitives in any order.
- Solving intersections (Fig. 10) does not increase complexity much.
- Analytic approach is theoretically the best quality anti-aliasing (Table 2).
- Rendering and hidden surface elimination calculations can be done independently in multiple threads.

Disadvantages:

- Slow. $W=O(2^n)$. Very inefficient for rendering high-poly models.
- Even though it tries to calculate exact area, there is no perfect filter and calculations are still limited by the precision of floating point numbers. Since floating point is by itself an approximation of real number, calculations can be inaccurate.
- Requires a lot of memory to hold all geometry data and buffer layers. Dynamic memory is always slower than static.

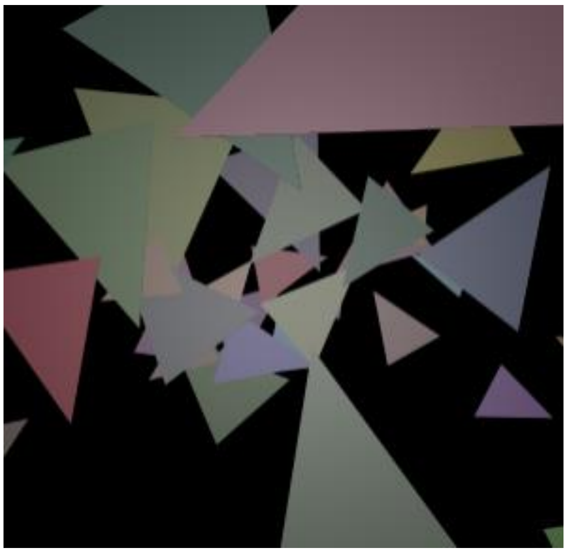


Figure 8 (shows that program solves hidden surface elimination for opaque objects)

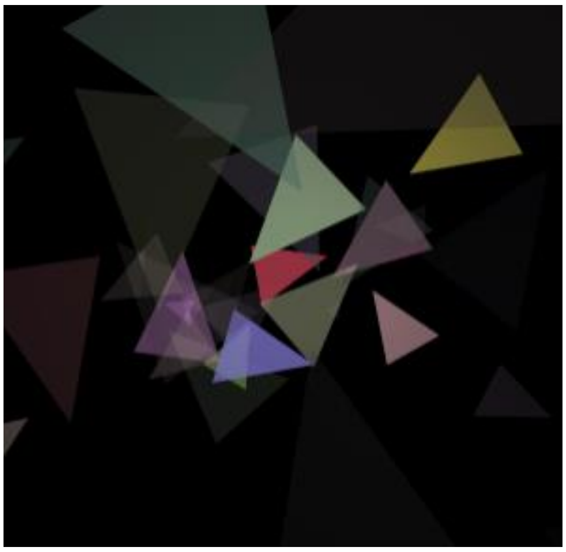


Figure 9 (shows that program solves hidden surface elimination for transparent objects)

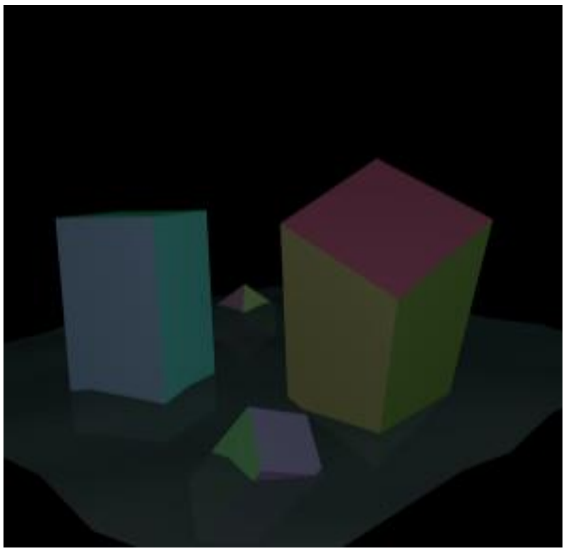


Figure 10 (shows that program correctly handles polygon intersections; boxes are intersecting with transparent terrain)

4. Conclusion

An algorithm for analytical anti-aliasing, which handles transparent objects and intersections was proposed, implemented (Fig. 11) and evaluated.

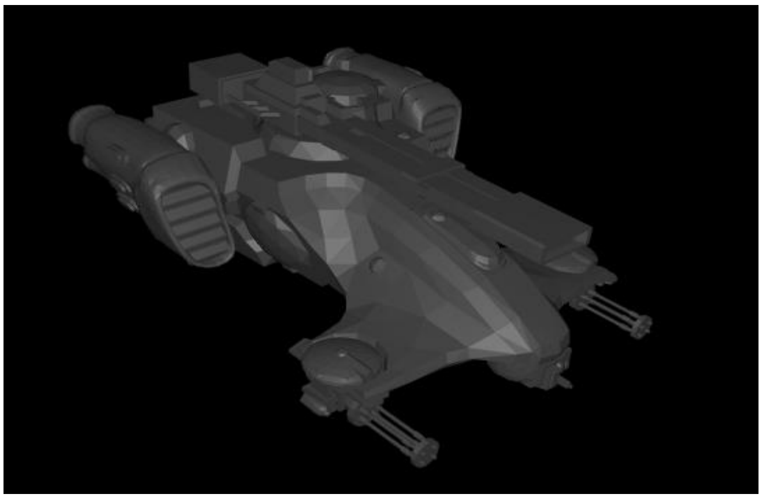


Figure 11 (shows that method is capable of rendering 3D models, correctly resolving visibility problem, intersections and applying anti-aliasing)

The main limitations of this approach are:

- Complex computations take a lot of time.
- It is hard to make it even more parallel and optimize further.
- Floating point errors can create aliasing and other inaccuracies.
- Filters are not ideal, since they approximate actual distribution of vector image.

Analytical methods are obviously not suited for fast real-time rendering, though for some applications where the scene is pre-rendered they can give better results, both in quality and time, compared to computational extensive high resolution *supersampling*. Presented method could be further optimized or combined with other techniques to achieve better results.

However, analytical methods give practically no benefit, when rendered scene does not have the aliasing effect of its own. This may lead to focusing future works on creating some hybrid approaches, which can detect high-frequency regions and apply corresponding type of anti-aliasing.

5. References

[1] Foley J., Sibert J., Wenner P., Acquah J., "A conceptual model of raster graphics systems," in *Proc. of SIGGRAPH '82*, New York, NY, USA, 1982.

[2] Grenader U., Probability and Statistics: The Harald Cramér Volume, Wiley: Alqvist & Wiksell, 1959.

[3] Crow F., "The aliasing problem in computergenerated shaded images," *Communications of the ACM,* vol. 20, no. 11, p. 799–805, November 1977.

[4] Echevarria J. I., Sousa T., Gutierrez D., Jimenez J., "SMAA: Enhanced Subpixel Morphological Antialiasing," *Computer Graphics Forum,* vol. 31, no. 2pt1, pp. 355-364, May 2012.

[5] Haines E., Hoffman N., Akenine-Möller T., Real-Time Rendering 3rd Edition, Natick, MA, USA: A. K. Peters, Ltd., 2008.

[6] Poulin P., Beaudoin P., "Compressed multisampling for efficient hardware edge antialiasing," in *Proc. of GI '04,* Waterloo, Ontario, Canada, 2004.

[7] Enderton E., Wexler D., "High-Quality Antialiased Rasterization," in *GPU Gems 2,* Addison-Wesley Professional, 2005.

[8] Kallio K., "Scanline edge-flag algorithm for antialiasing," in *Proc. of TPCG07,* Bangor, Wales, 2007.

[9] Guthe M., Jeschke S., Auzinger T., "Analytic antialiasing of linear functions on polytopes," *Comp. Graph. Forum 31, 2pt1,* p. 335–344, May 2012.

[10] Musialski P., Preiner R., Wimmer M., Auzinger T., "Non-Sampled Anti-Aliasing," in *Proc. of the 18th International Workshop on Vision, Modeling and Visualization,* Lugano, Switzerland, September 2013.

[11] Cohen-Or D., "Exact antialiasing of textured terrain models," *The Visual Computer,* vol. 13, no. 4, pp. 184-199, June 1997.

[12] Perry R. N, Jones T.R., "Antialiasing with Line Samples," in *Proceedings of the Eurographics Workshop on Rendering Techniques 2000,* London, UK, 2000.

[13] Sproull R. F., Schumacker. R. A., Sutherland I. E., "Sorting and the hidden-surface problem," in *Proc. of AFIPS '73,* New York, NY, USA, 1973.

[14] Atherton P., Weiler K., "Hidden surface removal using polygon area sorting," in *Proc. of SIGGRAPH ’77*, New York, NY, USA, 1977.

[15] Franklin W. R., "A linear time exact hidden surface algorithm," in *Proc. of SIGGRAPH ’80,* New York, NY, USA, 1980.

[16] Catmull E., "An analytic visible surface algorithm for independent pixel processing," in *Proc. of SIGGRAPH ’84*, New York, NY, USA, 1984.

[17] Mulmuley K., "An efficient algorithm for hidden surface removal," in *Proc. of SIGGRAPH ’89*, New York, NY, USA, 1989.

[18] Doan K., "Antialiased Rendering of Self-Intersecting Polygons using Polygon Decomposition," in *Proc. of PG'4*, 2004.

[19] Dévai F., "An optimal hidden-surface algorithm and its parallelization," in *Proc. of*

ICCSA'11, Berlin, Heidelberg, 2011.

[20] Hormann K., Greiner G., "Efficient clipping of arbitrary polygons," *TOG 17, 2,* pp. 71-83, 1998.

[21] Catmull E., *A subdivision algorithm for computer display of curved surfaces,* 1974.

[22] "Rendering Pipeline Overview," OpenGL.org, 12 April 2015. [Online]. Available: http://www.opengl.org/wiki/Rendering_Pipeline_Overview. [Accessed 28 April 2015].

[23] Rosen P., Popescu V., "Forward rasterization," *TOG 25, 2,* pp. 375-411, April 2006.

[24] Appel A., "Some techniques for shading machine renderings of solids," in *Proc. of AFIPS '68,* New York, NY, USA, 1968.

[25] Romney G. W., Evans D. C. Erdahl A., Wylie C., "Halftone Perspective Drawings by Computer," in *Proc. of AFIPS '67,* New York, NY, USA, 1967.

[26] Scratchapixel, "Rasterization: a Practical Implementation," Scratchapixel, [Online]. Available: http://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation . [Accessed 28 April 2015].

[27] Reed N., "Antialiasing: To Splat Or Not," 15 November 2014. [Online]. Available: http://www.reedbeta.com/blog/2014/11/15/antialiasing-to-splat-or-not/. [Accessed 28 April 2015].

[28] Levoy M., "Practical analytical antialiasing," 17 October 2002. [Online]. Available: http://graphics.stanford.edu/courses/cs248-02/scan/scan2.html. [Accessed 28 April 2015].

[29] Pineda J., "A Parallel Algorithm for Polygon Rasterization," in *Proc. of SIGGRAPH '88,* New York, NY, USA, 1988.

[30] tf3dm.com, "tf3dm," 2015. [Online]. Available: http://tf3dm.com/. [Accessed 2015].