Dual-Cone View Culling for Virtual Reality Applications

Bachelorarbeit

vorgelegt von

Jonathan W. Hale

an der

Universität Konstanz

Sektion Computergraphik

Fachbereich Informatik und Informationswissenschaft

1.Gutachter: Prof. Dr. Oliver Deussen
2.Gutachter: Prof. Dr. Bastian Goldlücke

Konstanz, 2018

Abstract

Topic:	Dual-Cone View Culling for Virtual Reality Applications
Bachelorcandidate:	Jonathan W. Hale
Supervisors:	Prof. Dr. Oliver Deussen Prof. Dr. Bastian Goldlücke Julian Kratt
Submission date:	April 10, 2018
Keywords:	View Culling, Optimization, Virtual Reality

English

Assessment and mathematics of a novel view culling methods optimized for virtual reality.

In the current renaissance of Virtual Reality (VR), new versatile fields are emerging: Desktop VR, Mobile VR, WebVR. All of these fields require highly performant applications to run on current hardware.

In this work we look at a popular method to improve performance in real-time 3D rendering applications – view culling – and explore how it can be improved to better suit VR applications. We focus on using conal instead of frustum-shaped volumes and investigate their performance compared to current solutions for stereo view culling.

Deutsch

Bewertung und mathetmatische Grundlagen einer neuen für Rendering in der Virtuellen Realität optimierte Sichtfeld Culling Methode.

In der aktuellen Renaissance der Virtuellen Realität (VR) entstehen neue Felder: Desktop VR, Mobile VR, WebVR. All diese Felder erfordern hoch performante Appikationen um den Anforderungen auf gängiger Hardware gerecht zu werden.

In dieser Arbeit betrachten wir eine populäre Methode um die Leistung von Echtzeit 3D Rendering Anwendungen zu verbessern – Sichtfeld ("view") culling – und wie sie möglicherweise besser an Anwendungen in der Virtuellen Realität angepasst werden kann. Wir konzentrieren uns dabei auf die Verwendung von Kegel- statt Frustum Sichtfeld Volumen und untersuchen ihre Performance im Vergleich zu herkömmlichen Methoden für Stereo-Sichtfeld Culling.

Table of Contents

1 Introduction

When we render a 3D object using a Graphics Processing Unit (GPU), we have to issue a so-called draw call. This draw call is a unit of communication of the Central Processing Unit (CPU) to the graphics hardware. The data for the 3D object is loaded and processed by a set of specialized GPU programs. These may be *tesselation shader*, *geometry shader* and *vertex shader*. The output of this final program is then clipped; essentially removing parts of the object outside of the screen. If the entire object is removed in the process of clipping, all the stages previous to that, have gone to waste: no pixels resulted from the draw call.

To prevent this from happening, we would like to know in advance, whether an object is visible to a virtual camera. Frustum culling has been around since before 1997 [AM00] and does exactly this. In 3D space the visible volume is a capped pyramid shape, a so-called *frustum*. To find out whether an object will be visible or not, we can test its bounds against this volume and prevent submission of a draw call for it, if it does not intersect it.

Virtual Reality has also been around for a while [Sut68]. It just recently started to grow in popularity since five years ago the Oculus Rift Kickstarter Campaign[1] gained a lot of attention.

With it, new displays are emerging and rendering requirements are increasing. Optimizations for VR rendering are sought and proven rendering techniques are revisited and adapted to VR. So is frustum culling with in [Qui17] by Íñigo Quílez, who suggests to accommodate for the per-eye rendering required for stereo rendering by using one frustum around both views instead of having to test every object twice.

[1]`https://www.kickstarter.com/projects/1523379957/oculus-rift-step-into-the-game` [16.01.2018]

In this work I look at frustum culling in VR in a different way. As in reality, our view is not rectangular as on a computer screen, but rounded.

Even though the screens inside head mounted displays are usually squared, the lenses used to warp the image around a viewers eye to achieve a higher field of view, are round and therefore allow a round area on the screen to be seen.

A interesting new approach to frustum culling could therefore be using a conal (cone-like) shape rather than a frustum shape to test against. Consequently, the name “frustum”-culling is no longer appropriate and we will from now on call it “view”-culling to accommodate both.

As view culling is likely never a bottleneck in rendering, especially in VR, it is not likely to be revisited by the industry in the near future, as expected improvements are low.

Nevertheless, the hope is that the result of culling is not only more acurate – i.e. objects that are not visible in the round areas on the HMD’s screens do not result in a draw call – but may be even faster compute.

Since I had this idea[2], I was unable to find other work related to culling with conal views or testing its viability in real world applications. Consequently I did so in this thesis.

Have fun reading!

1.1 Overview

This work is structured into four parts:

VR and Performance explains the performance requirements of VR and explores current ways to optimize applications to meet them.

Dual Conal View Culling dives into the mathematical foundations of testing intersection of various primitives and cones, as well as into their implementation, followed by

Results of benchmarks thereof and a cone based view culling implementation in Unreal Engine 4.

Conclusion finally summarizes the results and mentions some important restrictions as well as potential for future work.

[2]I was not the only one, though: `https://www.reddit.com/r/oculus/comments/3wxd7i/dual_cones_for_view_frustum_culling/` [22.01.2018]

2 VR and Performance

A classic video game runs at around 30-60 frames per second (fps) with HD or full HD resolution (1920x1080 pixels). The application therefore has 1̃6.66 milliseconds to render every frame and fill approximately 2 million pixels with meaningful color.

In contrast: a virtual reality application has to render between 90-120 fps [Dem16] with 1080x1200 pixels per eye (Oculus Rift/HTC Vive) or even up to 5120x2880 pixels (StarVR [Cor17]). That is about 11ms to render every frame and fill roughly 2.94 million pixels – or even up to 14.7 million in the case of StarVR.

Especially stereo rendering has therefore been a popular target for optimization in the recent years.

This chapter gives a broad overview of performance requirements of different VR platform types and describes a couple of optimization methods which are specific to VR rendering.

2.1 Performance Requirements of VR

2.1.1 Latency

VR applications have an additional performance requirement; when rendering an image, the virtual camera needs to match the head pose of the headset user. The duration between the user moving his head and an image to appear on screen, according to the updated head pose, is called "motion-to-photon latency", short latency.

The shorter this latency, the more immersed the user will feel when using the application. For the user to feel "presence" in the application, which is the term for the feeling of "being there" in the virtual world, latency needs to be lower than 20 ms. If above, the visuals can be noticeably dragging behind of head motions and even cause motion sickness as a result. [Car13]

Concluding, a virtual reality application does not only have high requirements caused by rendering high resolution images at high framerates, but also needs to keep the latency – the duration between receiving a new head pose from the HMD sensors to displaying the image according to this new head pose – as minimal as possible.

2.1.2 Desktop VR

The minimal system requirements for desktop virtual reality usually include either the NVIDIA GeForce 970 or AMD Radeon R9 290 [Inc17]. But even with this rather powerful hardware, VR Games still suffer from subpar graphics compared to classic video games.

Additionally, desktop virtual reality applications often display a "mirror" on classic monitors. Mirroring refers to presenting a third view on a classical computer screen for people outside of VR to be able to watch what a headset user is doing. For performance reasons this view is often a copy of the undistorted left or right view.

Comparable to desktop VR are high-end VR devices used in enterprise virtual reality experiences and theme parks. These headsets usually come with higher resolutions and are commonly backed by powerful backpack computers to allow the user to walk about more freely. Examples of these are the VOID Rapture VR headset[1], Vive Pro[2] and StarVR by Starbreeze[Cor17].

2.1.3 Mobile VR

Current mobile VR platforms run mostly on Google's Android operating system. Apart from the less powerful hardware compared to desktop VR, apps struggle with battery lifetimes and overheating of the mobile devices, which need to run on high loads for long durations, usually encapsulated in a VR headset like Daydream View or Samsung Gear VR.

Related to mobile VR headsets are standalone VR devices such as Oculus Go[3] or Vive Focus[4]. These are headsets that do not rely on a secondary device, such as a smartphone or computer. While their performance is comparable with smartphones, their operating systems are more slim and optimized for VR, with strongly reduced amount of services and other apps running alongside the VR application [Car17].

[1]https://www.sizescreens.com/void-rapture-vr-specifications/ [16.01.2018]
[2]https://www.vive.com/eu/product/vive-pro/ [16.01.2018]
[3]https://www.oculus.com/go/ [16.01.2018]
[4]https://www.vive.com/cn/product/vive-focus-en/ [16.01.2018]

2.1.4 WebVR

While WebVR is still in its early development phases, there are already a couple of browsers which support it: Firefox, Chrome Open Source, Chrome on Android, Microsoft Edge, Oculus Carmel (an Android browser by Oculus Inc.) and partial support in Samsung Internet (also for Android). [Dev18]

WebVR applications are generally simpler, being required to run on both mobile and desktop platforms and in addition being restricted by the performance of sandboxed JavaScript environments.

2.2 VR Specific Optimizations

Optimizing VR rendering starts at the choice of rendering approach. Often *forward shading* is chosen over *deferred shading*. This is due to deferred shading requiring alot of texture reads from the GBuffers at the high rendering resolutions – causing this to be the main GPU bottleneck while rendering. Additionally forward shading allows use of MSAA (Multi-Sample Anti-Aliasing), which results in crisper images than other multisampling techniques. [Dem16]

2.2.1 Multi-Res/Lens-Matched Shading

VR headsets use lenses to stretch the images around the users vision, which allows them to use smaller and flat screens. Since these lenses distort the image, on the HMDs screen it needs to be inversely distorted before presentation.

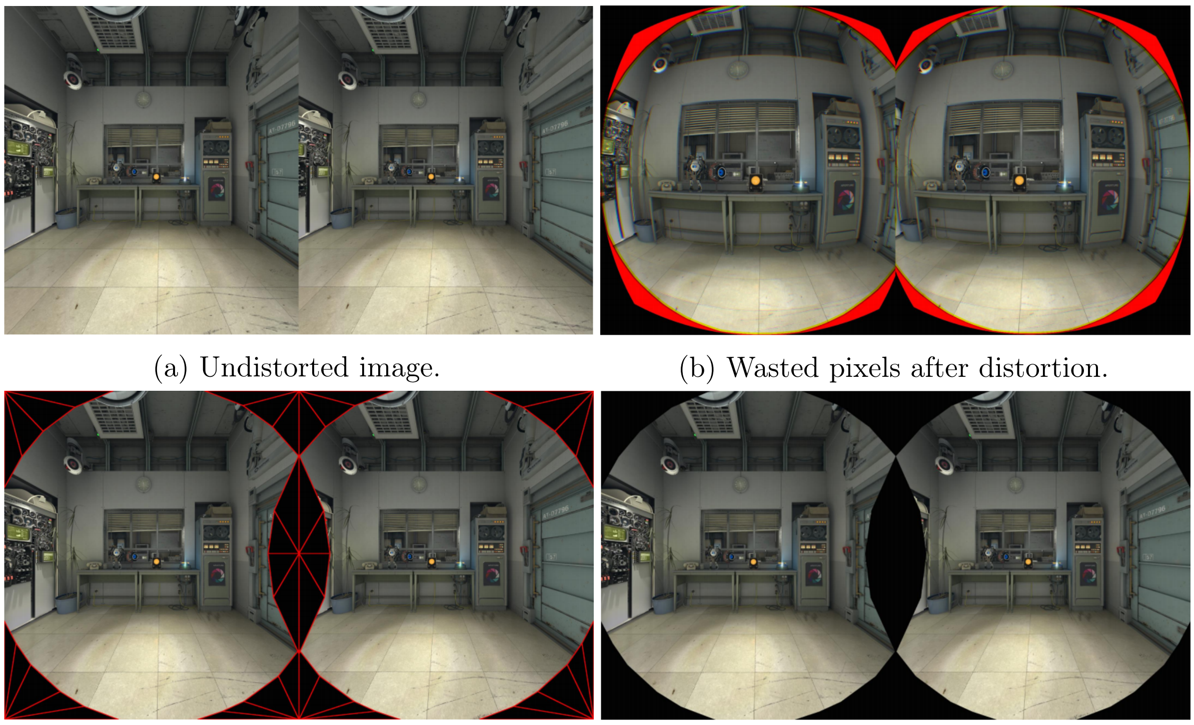
When distorting the result of scene rendering, the center of the image gets stretched (up to 1.4x) while the edges of the image get compressed [Pal17]. The idea of lens matched, or multi-res, shading is to adaptively shade only every other pixel in areas where these get compressed to one later. As we shade less pixels, this can save time in the fragment shader.

Other approaches render to a set of 8 frame buffers (per eye) with different resolutions. With tiled forward shading the resolution can even be determined per tile, which is then called *scaled-bin multires* [Pal17].

2.2.2 Stencil Mesh

As the lenses of VR devices are rounded, in some headsets only a circular cutout of the image displayed on the screen is actually visible through the lenses. Since the barrel distortion does not result in a perfectly round image, we may unnecessarily draw pixels that are not visible to the user as shown in figure 2.1b.

A method presented by Alexander Vlachos from Valve Studios [Vla15] uses stencil tests to mask out pixels that we do not need to render. A mesh as shown in figure 2.1c which represents the invisible area on the screen is drawn into the stencil buffer once


(a) Undistorted image. (b) Wasted pixels after distortion.

(c) Stencil mesh overlayed on undistorted image. (d) Final image.

Figure 2.1: Stencil mesh optimization [Vla15].

and then used as a mask every frame. As invisible pixels fail the stencil test, the cost of fragment shading and filling the pixel is saved.

This technique does not remove draw calls that only result in invisible pixels, though, which is one of the main motivators for dual conal view culling.

2.2.3 Foveated Rendering

Foveated rendering is only possible with eye tracking as this technique exploits the fact that we only really see detailed and clearly at the center of our vision, the so-called *fovea*.

Level of detail or adaptive resolution can be used to have clearer detail in the center of the users vision, while having less and less detail at the peripheral, greatly saving performance. Attention needs to be given to how the detail is reduced at the periphery, though: A simple blur on top of reduced rendering resolution will not suffice as we still perceive more prominent features like sharper edges even outside of the fovea. [Rig17]

2.2.4 Single Pass Stereo Rendering

An easy to implement naïve solution for rendering a scene in stereo would be to render the entire scene twice: once for the left eye and then again for the right eye. This would result in a doubled amount of draw calls per frame compared to mono rendering. Instead, a common technique to prevent the additional draw calls is instanced stereo rendering.

Instead of drawing a single mesh, we use instancing with an instance count of two and use the instance id in the vertex shader to transform the mesh instance with the right or left eye projection and view matrices appropriately. [Qui17] This has the disadvantage that the left and right eye views need to be rendered onto either the same texture or use the geometry shader to choose the right slice on a texture array.

Oculus Inc. submitted an extension to OpenGL for single pass stereo in 2014 [CFR+14] which is a high-level description of stereo rendering and enables GPU manufacturers to optimize stereo rendering futher using multipe GPUs or interlaced fragment shading for example.

For multi-GPU rendering, vendors additionally describe their own extensions to rendering APIs: Nvidia calls theirs “VR SLI”[5] while AMD refers to theirs as “Affinity Multi-GPU”[6].

2.2.5 Timewarp

Timewarp is a technique used to reduce latency while the previously mentioned techniques focus on reducing render time.

The distortion shaders of the VR headset drivers use *late latching* to retrieve more up-to-date sensor data just before presenting an image to the screen of the HMD.

The shader is able to use this data to reproject the image according to the new head transformation.

2.3 View Culling

View culling describes methods to remove those objects from a list, which are outside of a view volume. While it is commonly referred to as *frustum culling*, as classical rendering deals with frustum shaped views, we will instead refer to it as view culling, as we acknoledge other shapes for the view volume.

In computer graphics view culling is used to remove invisible objects from a list of objects to be rendered to the screen, which alleviates the need to submit a draw call for this object.

In classical rendering, the view volume is a frustum shape, which is tested using 4-6 planes. The near plane is often omitted as the additional computation cost often outweights the performance saved [Wih17].

View culling can be performed either on the CPU or the GPU, which has different use cases each.

[5]`https://developer.nvidia.com/vrworks/graphics/vrsli` [12.01.2018]

[6]`https://gpuopen.com/implementing-liquidvr-affinity-multigpu-in-serious-samvr/` [12.01.2018]

When view culling on the CPU, we can either cull instances from an instance buffer – which can decrease the size of a draw call – or cull entire draw calls.

When culling instances, we have to keep in mind, that the instance buffer has to be transferred to the GPU, which may take longer than simply letting the GPU draw the invisible objects.

Culling on the GPU can be used for culling instances, culling indirect draw calls or culling on a primitive level (i.e. triangles or particles). This can be achieved either with compute or geometry shaders.

For culling primitives, a corse culling step is prepended to the fine-grained culling on triangle level. In this step, batches of triangles are culled by a bounding volume and only those which partially intersect the view (not fully outside or inside the view volume) are then refined.[Wih17]

3 Dual Conal View Culling

In virtual reality and reality our view is more conical than frustum-shaped for each eye. And due to rendering and seeing in stereo, we need two cones to represent the volume of a person's visible space.

3.1 Motivation

Stencil meshes (see section 2.2.2) are a method of avoiding shading of pixels which are outside the visible volume. We do not need to render pixels outside of the circular areas that are visible through the headsets lenses. Some objects may not even result in any pixels at all, when entirely outside of the view cone. We would optimally like to prevent submitting draw calls for them entirely.

The hope is that either testing objects against a cone is more efficient than testing against a frustum, or the performance gain of not drawing the objects because of the increased accuracy outweighs a potential cost increase of intersection testing.

The following sections will describe experiments and the mathematical foundation in order to find out whether this method is indeed faster than the conventional frustum culling method when rendering for VR.

3.2 Mathematical Foundation

In case the mathematical notation used in this chapter is unclear, please refer to appendix A.

The mathematics for cone intersection mostly consist of trigonometric math and dot products. The most important concept is the relation of dot product of to vectors to

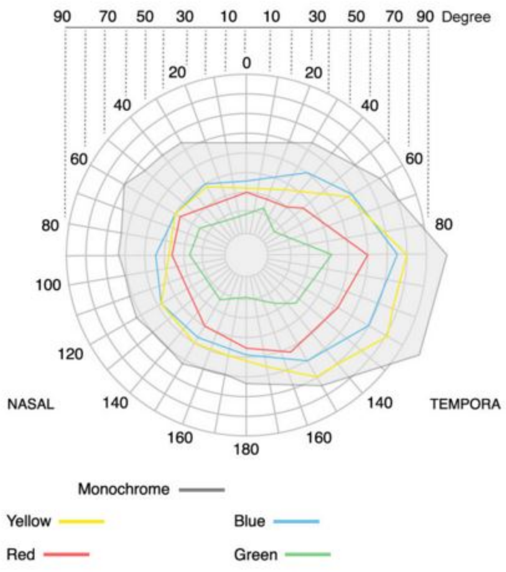


Figure 3.1: Field of view of a human eye [Uhd18].

the cosine of the angle θ between them:

$$\vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}| \cos\theta$$

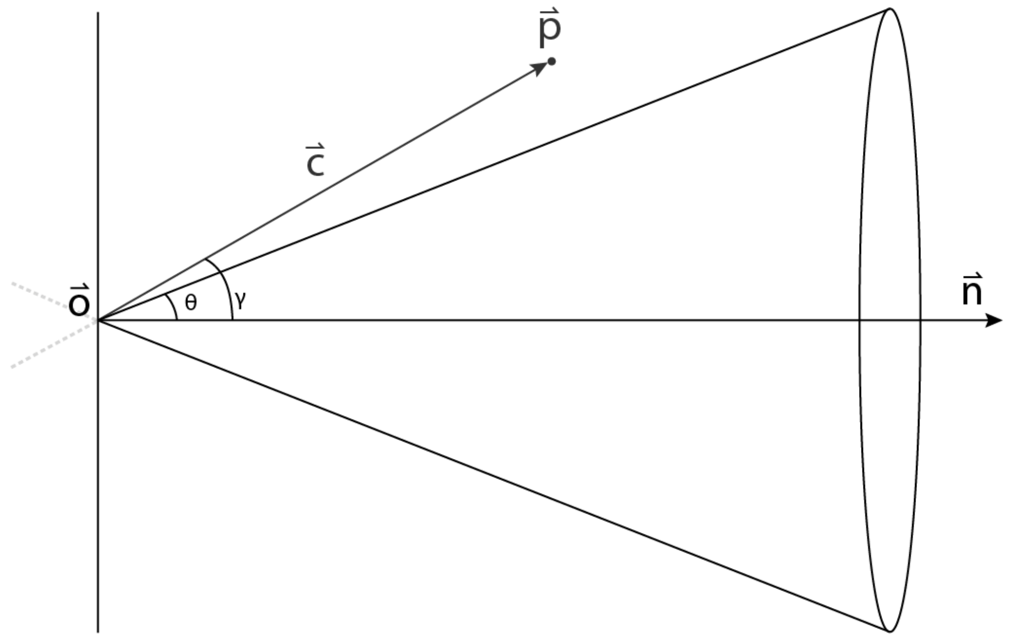
I derived the definition myself first, to get a better understanding of the material. The section “Through Tangens” is what I came up with, while “Through Cosine” is taken from other sources.

3.2.1 Definition of a Cone

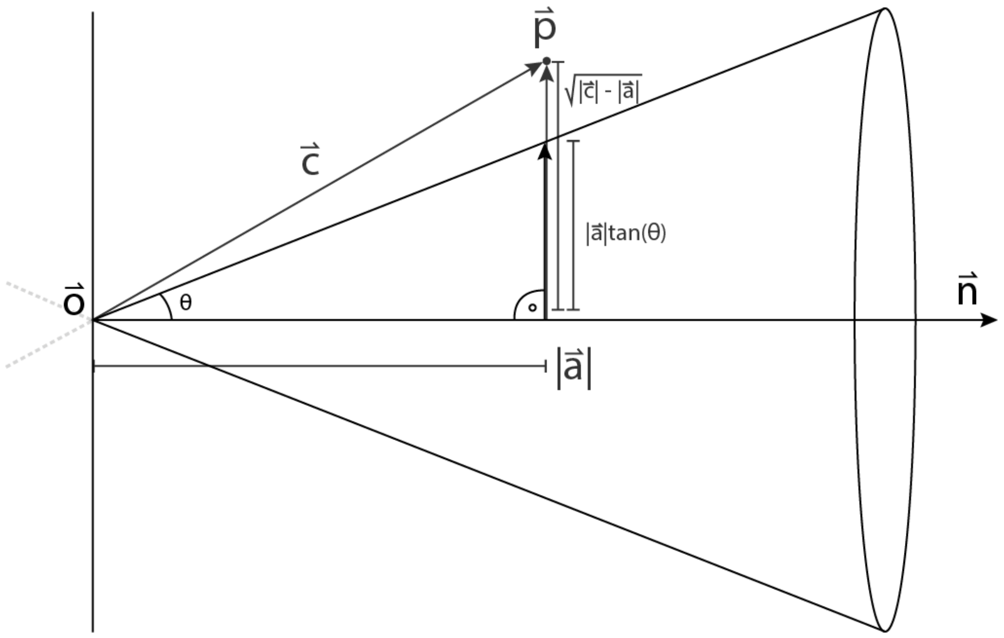
A three dimensional cone $\boldsymbol{C}$ has an **origin $\vec{o}$**, **normal $\vec{n}$** and opening **angle 2θ**. As used more frequently, θ will be called the cone’s **half-angle**, the angle between normal and cone surface. These quantities are illustrated in figure 3.2a.

The plane which shares the normal and origin of the cone shall be called the **cone plane** with *cone-side* (in front of the plane) and *double-cone-side* (behind the plane). A colloquial statement like “The point lies behind the cone.” refers to the point lying behind the cone plane.

We can restrict ourselves to *acute* cones – those with a half-angle of $0° < \theta < 90°$ – as in virtual reality applications we will never render views with a per eye field of view larger than than that of a single human eye (max. 167° horizontal, 150° vertical [Hen15]).



(a) A cone with labels for definition with cosine.



(b) A cone with labels for definition with tangens.

Figure 3.2: Schematic representation of a cone.

Through Tangens

With that, a *double-cone* C_d can be defined as the set of points whose smallest distance from the cones axis is less than or equal to the cones radius at that point:

$$\begin{aligned}
\forall \vec{p} \in \mathbb{R}^3 : \vec{p} \in C_d \Leftrightarrow \quad & \sqrt{|\vec{c}|^2 - |\vec{a}|^2} \leq |\vec{a}| \tan\theta \\
\Leftrightarrow \quad & |\vec{c}|^2 - |\vec{a}|^2 \leq |\vec{a}|^2 \tan^2\theta \\
\Leftrightarrow \quad & |\vec{c}|^2 = \vec{c}^2 \leq |\vec{a}|^2 \tan^2\theta + |\vec{a}|^2 \\
& = |\vec{a}|^2(\tan^2\theta + 1) \\
& = |\vec{a}|^2\sigma, \forall \vec{p} \in \mathbb{R}^3
\end{aligned}$$

$$\begin{aligned}
\vec{c} &= \vec{p} - \vec{o} \\
\sigma &= (\tan^2\theta + 1) \\
|\vec{a}| &= \vec{c} \cdot \vec{n}
\end{aligned}$$

To retrieve a single acute cone from this definition, we add a condition that holds only for those points that are in front of the plane composed by the origin and normal of the cone:

$$C = \{\vec{p} \in C_d | \vec{c} \cdot \vec{n} > 0\}$$

Note that the factor σ is independent of $\vec{p}$ and can be precomputed. Also noteworthy is that $|\vec{x}|^2$ for a vector $\vec{x} \in \mathbb{R}^3$ is simply the dot product $\vec{x}^2$ of the vector with itself and is easier to compute than the length of the vector (which requires an additional square root to compute). For axis aligned cones, this equation can be heavily reduced. For example with $\vec{n} = (0,0,1)^T$:

$$\begin{aligned}
\vec{c} &= \vec{p} - \vec{o} \\
|\vec{a}| &= \vec{c} \cdot \vec{n} = \vec{c}_z = \vec{p}_z - \vec{o}_z \\
\forall \vec{p} \in \mathbb{R}^3 : \vec{p} \in C &\Leftrightarrow \\
\vec{c}^2 - \vec{c}_z^2 &= \\
\vec{c}_x^2 + \vec{c}_y^2 + \vec{c}_z^2 - \vec{c}_z^2 &= \\
\boldsymbol{\vec{c}_x^2 + \vec{c}_y^2} &\boldsymbol{\leq \vec{c}_z^2(\tan^2\theta)} \\
&= |\vec{a}|^2 \cdot \tan^2\theta, \forall \vec{p} \in \mathbb{R}^3
\end{aligned}$$

The reduction here is not in size of the equation, but in its complexity: all multiplications are scalar multiplications instead of dot products.

This will be of great use later on, when culling primitives in view space or in normalized device coordinate space. The view transformation does not only align the view cone to the z axis, but also sets the origin of the cone to the origin of the coordinate system, which further reduces the equation with $\vec{c}_z = \vec{p}_z$ to:

$$\begin{pmatrix} \vec{p}_x \\ \vec{p}_y \\ \vec{p}_z \cdot \tan\theta \end{pmatrix}^2 \leq 0, \vec{p}_z \geq 0 \tag{3.1}$$

Through Cosine

Eberly [Ebe00] defines a three dimensional acute single cone as:

$$\forall \vec{p} \in \mathbb{R}^3 : \vec{p} \in C_d \Leftrightarrow \vec{n} \cdot \left(\frac{\vec{p} - \vec{o}}{|\vec{p} - \vec{o}|} \right) \geq \cos\theta$$

which in the case of $|\vec{p} - \vec{o}| \neq 0$ is equivalent to:

$$\forall \vec{p} \in \mathbb{R}^3 : \vec{p} \in C_d \Leftrightarrow \vec{n} \cdot \vec{p} - \vec{o} \geq |\vec{p} - \vec{o}| \cos\theta$$

Intuitively this can be read as those points whose angle of the vector $\vec{c} = \vec{p} - \vec{o}$ to $\vec{n}$ is smaller or equal to the cones half-angle θ. As with the previous definition, if we square both sides, we avoid computing the square root required for the length of $\vec{c}$. This gives us the double cone:

$$(\vec{n} \cdot \vec{c})^2 \geq (\cos^2\theta)|\vec{c}|^2$$

As before, to obtain the single acute cone again, we add the condition that the points need to lie in front of the plane given by $\vec{o}$ and $\vec{n}$:

$$\forall \vec{p} \in \mathbb{R}^3 : \vec{p} \in C \Leftrightarrow \vec{c} \cdot \vec{n} > 0 \wedge (\vec{n} \cdot \vec{c})^2 \geq \sigma|\vec{c}|^2 \tag{3.2}$$

with precomputable factor $\rho = (\cos^2\theta)$.

3.2.2 Intersections

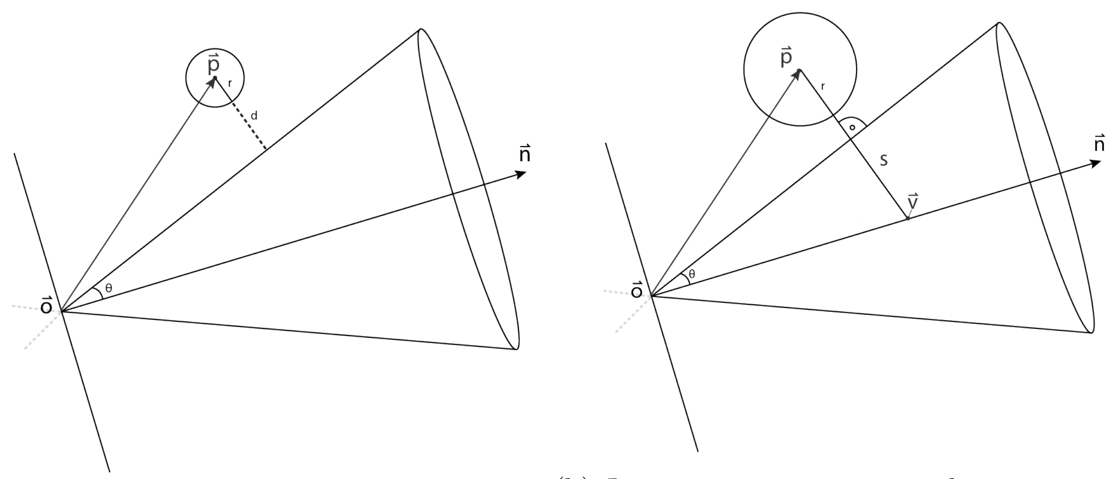
For view culling, we are interested in whether different primitives intersect our view cone. We are not interested in where or how, which should allow for shortcuts that physics and ray tracing applications are not able to make. The following subsections will give an overview over intersection a cones with various other primitives.

Point - Cone

A subtask of Triangle - Cone intersection is testing for each vertex whether it is inside or outside of the cone. This is the simplest intersection and can be directly derived from equation 3.2.

The condition of a point intersecting the cone is:

$$\vec{c} \cdot \vec{n} > 0 \wedge \vec{c} \cdot \vec{c} \leq \sigma(\vec{n} \cdot \vec{c})^2 \tag{3.3}$$



(a) Distance of sphere to cone.

(b) Intersection cone axis and cone tangent-plane normal.

Figure 3.3: Cone sphere intersection overview.

Or expressed with cosine:

$$\vec{c} \cdot \vec{n} > 0 \wedge \rho \vec{c} \cdot \vec{c} \leq (\vec{n} \cdot \vec{c})^2 \tag{3.4}$$

Sphere - Cone

Figure 3.3a shows an overview of the situation and labels some important quantities.

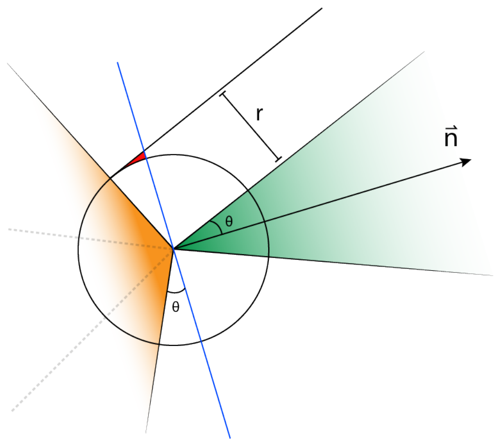
There are many sides from which this problem can be tackled: comparing angles, comparing their cosines, calculating the actual distance, trying to approximate certain quantities, calculating the surface normal towards $\vec{p}$ or computing the closest point on the cone.

Many of these somehow either involve a non-precomputable trigonometric function, inverse trigonometric function or square root, which we want to avoid as they are expensive to compute and for view culling this test is run per object. Our baseline, the sphere-frustum intersection test, does not require any, either.

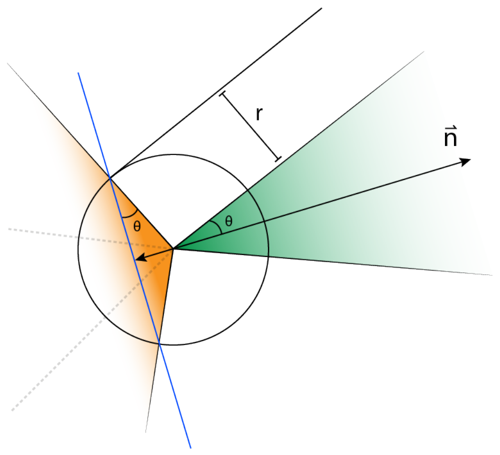
Again Eberly provides a solution for cone-sphere intersection in [Ebe02]. As shown in their paper, Eberly's method can be optimized to be free of square root, and trigonometric functions are only applied to the cone and are therefore easily precomputable, yet there is a way to further improve it.

As we are dealing with a single cone, for point-cone intersection we check whether the center is in front or behind the cone plane. This results in a simpler case for when the center of the sphere lies bedind or on the this plane. Further, this allows us to assume $\cos(\vec{n} \cdot \vec{c}) > 0$ for when the point is fully in front of the plane.

When intersecting with a sphere, this is not quite the case, though, as there a points behind the cone that are closer to the surface of the cone than to its origin, which keeps us from just simply testing the origin with the sphere. This is illustrated in figure 3.4a,



(a) Sphere-Origin test.



(b) Sphere-Orgin test with offset.

where the orange area is the cone in which a simple test of sphere with origin is sufficient to know whether the sphere intersects with the cone. The blue line indicates the plane with the normal of our cone and finally the small red area shows where the origin-sphere test would result in a negative result because the origin is actually closer than the radius to the cones surface, but not to the cones origin/apex.

To solve this, we offset the cone plane from the origin by $-\sin(\theta) \cdot \vec{n}$, splitting the space into behind the plane, where it is sufficient to test wheter the origin is within the sphere:

$$\vec{c} \cdot \vec{c} \leq r^2 \tag{3.5}$$

and in front of the plane, where we need a full test using the distance from the surface. This results in two cases to handle, where Eberly's method requires three cases to be handled.

In this latter case we know that the distance of the point to the cone is at most r in the case that the sphere overlaps or touches the cone.

Trivially we could just subtract the radius from the point-cone distance to retrieve the distance to the sphere. Figure 3.3b visualizes that the distance from $\vec{v} = |\vec{c}|$ to $\vec{p}$ can be compared to the maximal distance $s = \sin(\frac{\theta}{2}) \cdot |\vec{c}|$ when subtracting r. Because of its simplicity, this method was implemented as baseline for benchmarking and ground truth for testing other implementations, see section 3.3.2. Less trivial is squaring the equation, as we need to get rid of the square roots required for distances between points: we would instead like to compare squared distances, but $(d - r)^2 \neq d^2 - r^2$.

The key to solving this is extending the cone so that the distance labeled s in figure 3.3b is extended by r. The distance s is the extension of the line from the sphere center that hits the surface of the cone at a 90° angle (and therefore is the smallest distance from sphere to cone surface) to the cone axis. This extension by r is equivalent to a negative offset of the cones origin in normal direction as shown in figure 3.5, resulting

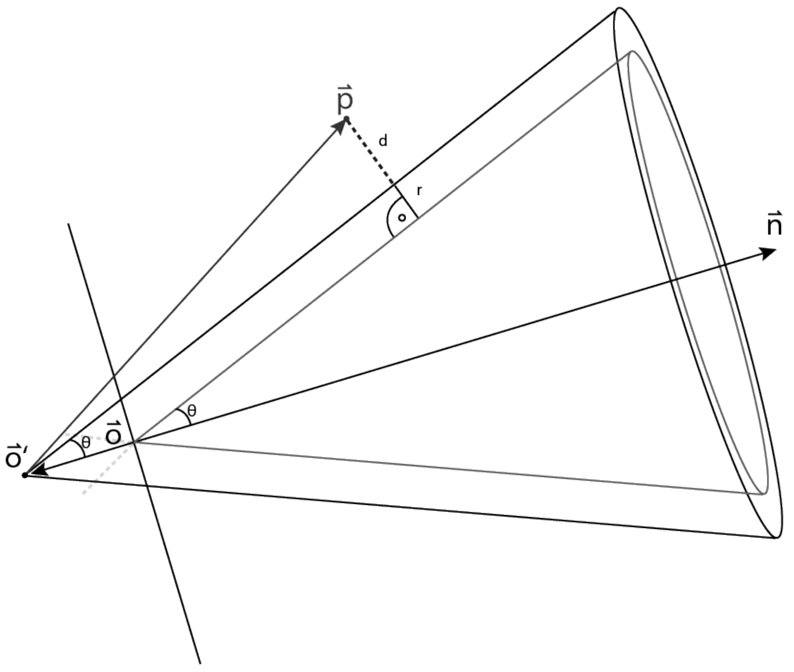


Figure 3.5: Cone with offset.

in a cone with translated origin $\vec{o}'$:

$$\vec{o}' = \vec{o} - \vec{n}\frac{r}{\sin(\theta)} \tag{3.6}$$

We can now simply test the sphere's origin $\vec{p}$ against this new cone. Combining equations 3.5, 3.6 and 3.3 we obtain the final equation:

$$C \cap S \neq \varnothing \Leftrightarrow \begin{cases} \vec{c}\cdot\vec{c} \leq r^2 & \vec{n}\cdot(\vec{c} - \sin\theta\cdot\vec{n}) \leq 0 \\ (\vec{n}\cdot\vec{c}')^2 \geq \rho\vec{c}'\cdot\vec{c}' & \text{otherwise} \end{cases}$$
$$\text{with } \vec{c}' = (\vec{o} - \vec{n}\frac{r}{\sin\theta} - \vec{p})$$

The division can be optimized out as described in section 3.3.2. (σ is defined in section 3.2.1.)

AABB - Cone

A very common shape to describe the bounds of 3D objects is the *Axis-Aligned Bounding Box* (AABB), a box whose normals are all parallel to one of the coordinate axis. This shape consists of a center $\vec{c}$ and (half-)extents $\vec{e}$.

As intersection of AABBs with cones has not been pratical yet – in physics engines, you require rotations which result in non-axis-aligned cones for example – it is even

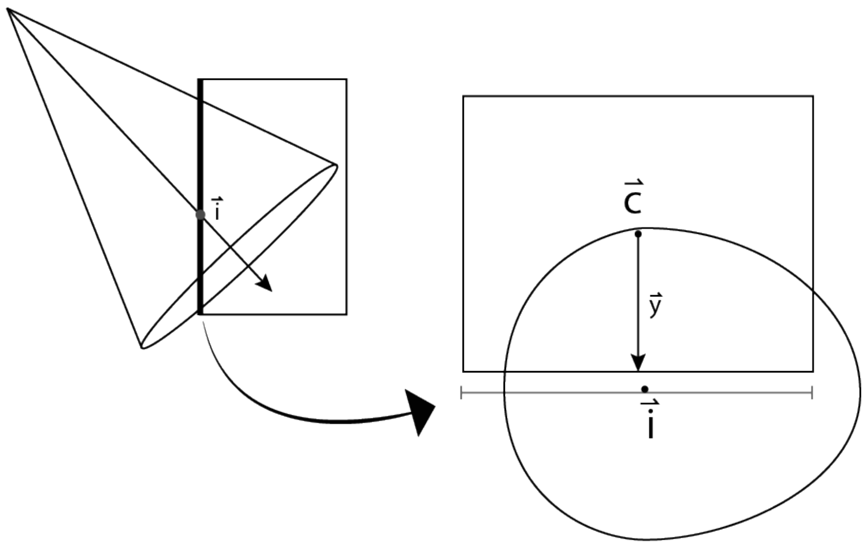


Figure 3.6: Determining the closest point to the cone axis on an AABB face.

missing from the overview table of object intersections on the website accompanying the Realtime Rendering book [TAMH17].

I did, however, find a method to do so, which is even efficient enough to at least compete with a naïve frustum-box intersection implementation, see section 4.1.1.

The method finds the closest point of the AABB to the cone axis on each of the six planes to then test this point against the cone (described in section 3.2.2), until a point is found.

If the cone intersects the box, it must intersect at least one of the planes. (Except if it is cut to a certain height, in which case the origin lies in the box itself, so we can exclude this case by testing for that first.) We can differentiate three types of closest points in case of intersection:

1. the closest point is one of the boxes vertices (a corner),
2. otherwise the closest point is on an edge
3. or, lastly, the cone intersects with neither edges or vertices, in which case the axis of the cone intersects with the plane of a face in the bounds of the box.

To find the closest point, we intersect the cone axis with one of the AABB planes. The intersection point with a plane with normal along axis $a \in \{0, 1, 2\}$ is given by

$$\vec{i} = \vec{n} \cdot \frac{(\vec{c}_a - \vec{o}_a) \pm \vec{e}_a}{n_a}. \tag{3.7}$$

Given that $\vec{n}_i$ is not zero, otherwise the axis does not intersect the plane of that face.

We proceed to find the value on the boxes face that is closest to this intersection (as it represents our cone axis) on the other two axis. We can distinguish two cases: Either the intersection's component for our axis is within the bounds of the AABB on that axis, or outside. In the former case, the closest point is the one closest to the closest edge, in the latter the closest point's component is the same as the intersection's.

By determining all tree components for the must-be closest point on a face, we can then test this point as described in section 3.2.2. If it does not, we need to check the other faces.

This is illustrated in figure 3.6: Intersecting the plane of the bold face with the cone axis gives us point $\vec{i}$, which is clearly in the bounds on the axis from left to right, but for the axis from down to up it is outside, so we use the lower edge (shown by $\vec{y}$) as the component for the closest point. In this case, the closest point is clearly inside the cone.

Triangle - Cone

Testing intersection with a triangle is necessary when culling primitives on the GPU. In most cases whether a triangle intersects can already be decided by checking its vertices, exiting early as soon as one does. In some cases, though, the edges or even the triangles plane needs to be checked.

Eberly [Ebe00] describes a method to do this: First check the vertices, if that is insufficient, check the edges and if that is insufficient, check the plane. This results in having to check all edges and the plane for all triangles whose vertices are outside of the cone. This is still efficient under the assumption that we are culling primitives that are mostly inside the cone. This is indeed the case in GPU culling, as we usually first cull batches of triangles and then refine culling for the visible batches [Wih17].

As I was not able to find a more efficient method, I will outline Eberly's method here.

The point-cone test has been described in section 3.2.2 and can be used to test the vertices. If any is inside the cone, the triangle intersects and we are done. If none are inside the cone, we will test all three edges next. For the edge intersection test, we clip each of the line segments to the cone plane to be able to test intersection with the double cone instead. An edge fully behind the plane will never intersect the cone, while and edge fully in front of the plane requires no clipping.

The points $\vec{p}_0$ and $\vec{p}_1$ form the edge $\vec{e}(t) = \vec{p}_0 + t\vec{d}, \vec{d} = \vec{p}_1 - \vec{p}_0, t \in [0,1]$. We combine the line with equation 3.4 to obtain:

$$\begin{aligned} \vec{n} \cdot (\vec{e}(t) - \vec{o}) &\geq 0 \\ (\vec{n} \cdot (\vec{e}(t) - \vec{o}))^2 - \rho^2 |\vec{e}(t) - \vec{o}|^2 &= 0 \end{aligned}$$

To solve the second, we find the quadratic equation $q(t) = at^2 + 2bt + c$ with

$$\begin{aligned}
a &= (\vec{n} \cdot \vec{d})^2 - \rho^2 |\vec{d}|^2,\\
b &= (\vec{n} \cdot \vec{d})(\vec{n} \cdot \vec{c}) - \rho^2 \vec{d} \cdot \vec{c},\\
c &= (\vec{n} \cdot \vec{c})^2 - \rho^2 |\vec{c}|^2,\\
\vec{c} &= \vec{p}_0 - \vec{o}.
\end{aligned}$$

We now try to find a $t \in [0, 1]$ for either the full edge or the clipped edge:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{3.8}$$

It is sufficient to test whether there is a solution instead of computing the solution itself.

Since we know that both vertices are outside the cone, we know that $q(0) < 0$ and $q(1) < 0$, which means the graph of the quadratic has to be concave (down) in order for equation 3.8 to have a solution between these points. For this to be the case, $a < 0$ must hold.

If the edge intersects, the closest point on it is somewhere in the range we are looking for, which is at $\hat{t} = \frac{-b}{a}$. The division can be avoided, since we are only interested in whether this point is on the edge, $0 < \hat{t} < 1$, and can test this with $0 \le b \le -a$ as $a < 0$. Finally, we need to ensure $b^2 - 4ac > 0$ for the square root to have a solution.

For a clipped edge, we instead find a solution in a more constrained interval for t. This interval is defined by the point that lies in front of the plane ($t = 0$ or $t = 1$) together with the t-value of the intersection with the cone plane $t_i = -\frac{\vec{n}\vec{c}}{\vec{n} \cdot \vec{d}}$. Again, we know that $q(t_i) < 0$ as this point does not intersect. We neglect the case where the intersection point is the origin of the cone, as on the one hand for culling this case is irrelevant (behin the near-z plane).[1]

We can again avoid the division for calculating t_i in the comparisons, which leads us to:

$$(a_2 < 0) \wedge (a(\vec{n} \cdot \vec{c}) \le b(\vec{n} \cdot \vec{d})) \wedge (b^2 \ge ac) \wedge \begin{cases} 0 \le b, \text{if } p_0 \text{ is on the cone side} \\ b \le -a, \text{if } p_1 \text{ is on the cone side} \end{cases} \tag{3.9}$$

If one edge intersects, the triangle intersects and we are done. If none intersects, we have to check the plane. Similarly to the AABB-intersection, in case of intersection, for this final case the axis must be intersecting the triangle, as it does not intersect the edges.

Ray-triangle intersection is a common problem in raytracing and can be done with the Möller-Trumbore algorithm [MT97] for example. We can leave out the calculation of the intersection point, though, as we do not need it.

[1]Eberly explains that this case is even not relevant in general.

Potential in NDC

After vertices are projected through a perspective projection matrix, they end up in *normalized device coordinate* (NDC) space. In this space, a cone is no longer a cone but a cylinder with radius of 1 – analog to a frustum being stretched into the NDC cube.

This allows culling the transformed triangles using a 2D line-unitcircle intersection algorithm using only the x and y coordinates of the triangles vertices.

3.3 Implementation

All of the intersection tests described in 3.2 “Mathematical Foundation” were implemented using the Magnum library to be rigorously tested and benchmarked.

Magnum is an open source “Lightweight and modular C++11/C++14 graphics middleware for games and data visualization”[Von18] with excelent documentation and test coverage, which makes it reliable and dependable for these isolated implementations. For visual verification, I wrote an application to quickly visualize the intersection results and was used to generate figures 3.7, 3.8, 3.9 and 3.10.

On top of that, it comes with a simple unit testing and benchmarking framework, which were both used to validate optimizations and benchmark the various approaches.

Later, these implementations were migrated to the Unreal Engine 4 [Epi18] math API to be used in the actual view culling implementations in “real-world” environments.

3.3.1 Point - Cone

Since this is used by the other intersection tests, I decided to benchmark point-cone intersection separately. We have three methods we would like to test: using the cosine as in equation 3.4, using the tangens in a precomputed factor as in equation 3.3 and finally using the simplicity of equation 3.1 by transforming the point with a 4×4 view matrix first.

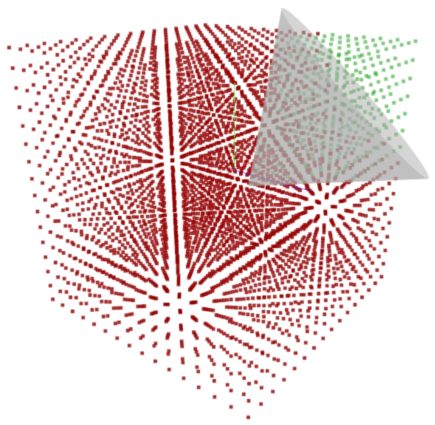


Figure 3.7: Visualization of point-cone intersection tests.

3.3.2 Sphere - Cone

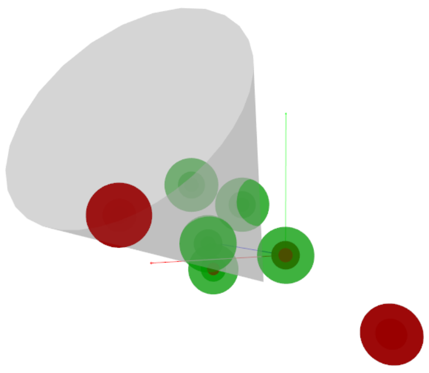


Figure 3.8: Visualization of sphere-cone intersection tests.

As I use own methods for this, I had to ensure that it is valid and carefully test against a fail-proof implementation. As ground truth for testing, I implemented a straight forward version using `acos` and similarly expensive functions and visualized the result in a small custom application for visual verification, as well as testing certain cases in unit tests.

The code for this straight forward implementation is shown in listing 3.1 for your consideration.

Listing 3.1: Ground truth sphere-cone intersection implementation.

```
template<class T> bool sphereConeNotFast(
        const Vector3<T> sCenter, const T radius,
        const Vector3<T>& origin, const Vector3<T>& normal, const Rad<T>
            angle) {
    const Vector3<T> diff = sCenter - origin;
    const Vector3<T> dir = diff.normalized();
    const Rad<T> halfAngle = angle/T(2);

    /* Compute angle between normal and point */
    const Rad<T> actual = Math::acos(dot(normal, dir));

    /* Distance from cone surface */
    const T distanceFromCone = Math::sin(actual - halfAngle)*diff.length();

    /* Either the sphere center lies in cone, or cone is max radius away from
       the cone */
    return actual <= halfAngle || distanceFromCone <= radius;
}
```

I also implemented the method by Eberly [Ebe02] and a couple of own methods, described in section 3.2.2. From the benchmarks shown in section 3.3.2, the most interesting implementation was transforming the sphere using a precomputed 4×4 matrix (which only requires transforming the center vertex, of course), so that the test is equivalent to a z-axis-aligned cone test which is therefore also shown in listing 3.2.

Listing 3.2: Optimized sphere-cone intersection implementation through transform to axis aligned cone test.

```cpp
template<class T> bool sphereConeView(
        const Vector3<T> sphereCenter, const T radius,
        const Matrix4<T> coneView,
        const T sinAngle, const T cosAngle, const T tanAngle) {

    /* Axis align cone */
    const Vector3<T> center = coneView.transformPoint(sphereCenter);

    if (center.z() > -radius*cosAngle) {
        /* Axis aligned point - cone test shifted by radius*/
        const T coneRadius = tanAngle*(center.z()+radius/sinAngle);
        return center.xy().dot() <= coneRadius*coneRadius;
    }

    return false;
}
```

3.3.3 AABB - Cone

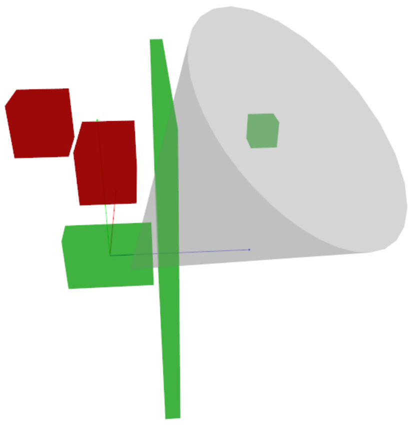


Figure 3.9: Visualization of AABB-cone intersection tests.

With the axis-aligned bounding boxes, we cannot make use of being able to transform into an axis-aligned cone as then the AABB is no longer necessarily axis aligned. We can improve on a straight forward implementation of the method described in section 3.2.2 by always testing the two perpendicular planes of the AABB directly after each other. This gives the compiler a hint that these may be vectorized.

Additionally, the final axis whose planes are checked will not yield much additional information: all vertices and edges are shared with planes that already have been tested and did not contain a closest point. We can conclude that if we reach this final axis, it can only be a face-only intersection which must contain the intersection point of the axis and it is sufficient to test this.

Listing 3.3: AABB-cone intersection implementation.

```
template<class T> bool aabbConeOptimized(
        const Vector3<T>& center, const Vector3<T>& extents,
        const Vector3<T>& origin, const Vector3<T>& normal,
        const T coneHeight, const T tanAngleSquaredPlusOne)
{
    for (int axis = 0; axis < 2; ++axis) {
        const int Z = axis;
        const int X = axis + 1;
        const int Y = (axis + 2) % 3;
        if(normal[Z] != T(0)) {
            const Vector3<T> i0 = normal*((center[Z]-extents[Z])/normal[Z]);
            const Vector3<T> i1 = normal*((center[Z]+extents[Z])/normal[Z]);

            for(auto i : {i0, i1}) {
                Vector3<T> closestPoint = i;

                if(i[X] >= extents[X]) {
                    closestPoint[X] = center[X] + extents[X];
                } else if(i[X] - center[X] <= -extents[X]) {
                    closestPoint[X] = center[X] - extents[X];
                }
                /* Else: normal intersects within X bounds */

                if(i[Y] >= extents[Y]) {
                    closestPoint[Y] = center[Y] + extents[Y];
                } else if(i[Y] <= -extents[Y]) {
                    closestPoint[Y] = center[Y] - extents[Y];
                }
                /* Else: normal intersects within Y bounds */

                if(pointCone<T>(closestPoint, origin, normal,
                    tanAngleSquaredPlusOne)) {
                    /* Found a point in cone and aabb */
                    return true;
                }
            }
        }
        // else: normal will intersect one of the other planes
    }

    if(normal.z() != T(0)) {
        const Vector3<T> i0 = normal*((center.z()-extents.z())/normal.z());
        const Vector3<T> i1 = normal*((center.z()+extents.z())/normal.z());

        const bool b0 = pointCone<T>(i0, origin, normal,
            tanAngleSquaredPlusOne);
        const bool b1 = pointCone<T>(i1, origin, normal,
            tanAngleSquaredPlusOne);

        return b0 || b1;
    }

    return false;
}
```

3.3.4 Triangle - Cone

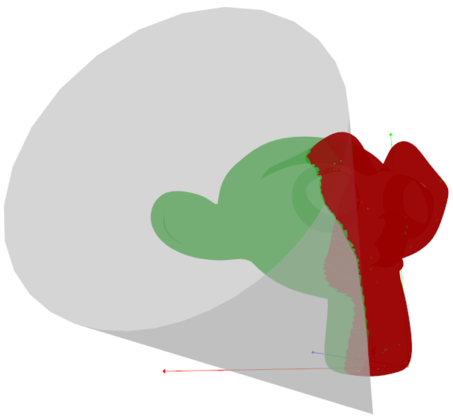


Figure 3.10: Visualization of trianlge-cone intersection tests.

As I was unable to come up with a better method for testing intersection of triangle and cone, even for axis-aligned cones, I did not investigate much further. While there will definitely be some tricks to get the compiler to produce vectorized code that checks all three vertices at once for example, the proof of concept is there.

Listing 3.4: Triangle-cone intersection implementation.

```
template<class T> bool triangleCone(const Vector3<T>& p0, const Vector3<T>&
    p1, const Vector3<T>& p2, const Vector3<T>& origin, const Vector3<T>&
    normal, const T cosAngleSq) {
    bool inFront[4]{false, false, false, false};
    const Vector3<T> points[4]{p0, p1, p2, p0}; /* Avoid modulos in array
        access */

    for (int i = 0; i < 3; ++i) {
        const Vector3<T> diff = points[i] - origin;
        const T d = dot(normal, diff);
        inFront[i] = d >= T(0);
        if(inFront[i]) {
            if(d*d >= cosAngleSq*diff.dot()) {
                return true;
            }
        } else {
            /* behind the cone */
        }
    }

    if(!inFront[0] && !inFront[1] && !inFront[2]) {
        return false;
    }
    nFront[3] = inFront[1]; /* Avoid modulos in array access */

    /* If any edge intersects, the triangle intersects, therefore test all of
        them */
    for(int i = 0; i < 3; ++i) {
        if(!inFront[i] && !inFront[i+1]) {
            /* Does not intersect as behind the cone plane */
            continue;
        }

        const Vector3<T> dir = points[i+1] - points[i];
        const T d = dot(normal, dir);

        const T c2 = d*d - dir.dot()*cosAngleSq;
        if(c2 > T(0)) {
            continue;
        }

        const Vector3<T> o = points[i] - origin;

        const T dirDotO = dot(dir, o);
        const T normDotO = dot(normal, o);

        const T c1 = d*normDotO - cosAngleSq*dirDotO;
        if(inFront[i] && inFront[i+1]) {
            /* Handle edges fully on the cone side */
            if(T(0) <= c1 && c1 <= -c2) {
                const T c0 = normDotO*normDotO - cosAngleSq*o.dot();
                if(c1*c1 >= c0*c2) {
                    return true;
                }
            }
        } else {
            /* Handle edges that intersect cone plane */
            if(((inFront[i] && T(0) <= c1) || (inFront[i+1] && c1 <= -c2))
                && c2*normDotO <= c1*d) {
                const T c0 = normDotO*normDotO - cosAngleSq*o.dot();
                if(c1*c1 >= c0*c2) {
                    return true;
                }
            }
        }
    }

    /* Plane test */
    const Vector3<T> edge0 = points[1] - points[0];
    const Vector3<T> edge1 = points[2] - points[1];

    const Vector3<T> triangleNormal = cross(edge0, edge1);
    const T dotTriangleConeNormal = dot(triangleNormal, normal);

    const Vector3<T> delta0 = points[0] - origin;

    const Vector3<T> u = dot(triangleNormal, delta0)*normal -
        dotTriangleConeNormal*delta0;
    const Vector3<T> nCrossU =
        Math::sign(dotTriangleConeNormal)*cross(triangleNormal, u);

    if(dot(nCrossU, edge0) <= T(0) && dot(nCrossU, edge1) >= T(0)) {
        const T denom = dotTriangleConeNormal*triangleNormal.dot();
        return dot(nCrossU, edge1) <= denom && dot(nCrossU, edge0) <= denom;
    }

    return false;
}
```

3.3.5 Application to Dual Cones

As we are not only dealing with a single view, but two views in stereo rendering, we require testing a primitive against two cones.

For instanced stereo rendering, a draw call will be issued if an object is in either of the views. We can therefore use a plane in the center between the cones to determine whether to test against the right or left cone.

When submitting the draw calls twice, we can either choose to test all objects again for the second view, or again use a test to find those objects which are in the union of both cones, or even better, do one pass over the all objects and test against both cones, storing which cones the object is visible in (left, right or both). A single pass allows us to take the advantage of the fact that the cone axis are parallel[2] and therefore share the cone plane and usually have the same angles. We can therefore reuse some calculations for the second cone and possibly even get the compiler to vectorize the code to hide the cost of the second cone test entirely.

With axis aligned cones, we can apply an x axis offset for the second cone to avoid the second view transformation (Again under the assumption that the cone normals are parallel).

[2]If they are, which is actually not necessarily the case in more modern headsets.

4 Results

All benchmarks where run on a machine equipped with an Intel i7-4790K processor, 16GB of RAM and an NVidia Geforce 970 GPU on Windows 10. All code was compiled using Microsoft Visual C++ (MSVC) 15[1]. The benchmarks for intersection math were implemented using the Corrade benchmarking framework (which comes along with the Magnum library). Performance of implementations in Unreal Engine was measured with the engine's profiling tools.

4.1 Intersection Tests

As the intersection math is a key element of the view culling implementation it has been benchmarked separately first. Great care was taken to ensure the benchmarks measure what we want measured – ensuring the compiler did not optimize away any code because the result is never used. Data was generated on the fly and every set of benchmarks contains at least one baseline to compare against.

Sections are ordered by sequence of implementation, as certain benchmarks were constructed because of results in previous benchmarks.

4.1.1 Sphere Intersection

The intention for this benchmark was to test if any of the sphere with cone intersection code could be faster than the sphere with frustum intersection test. This would give a clear indication of whether it could be worthwile to check the cones for the extra accuracy when used for VR rendering.

[1]Compiling and benchmarking with different compilers exceeded the scope of this thesis.

Mean Time	Description
26.17 ± 0.27 ms	Sphere - Cone Baseline
11.49 ± 0.15 ms	Sphere - Cone Own
11.29 ± 0.14 ms	Sphere - Cone Eberly
11.03 ± 0.18 ms	Sphere - Cone Eberly (“Optimized”)
10.72 ± 0.12 ms	Sphere - Cone Own (Optimized)
9.14 ± 0.10 ms	Sphere - Frustum Baseline
7.35 ± 0.09 ms	Sphere - Cone View Transform to Axis Aligned Test

Table 4.1: One-by-one intersection tests; Mean of 10 iterations with 10 passes over 64^3 objects per iteration.

Table 4.1 lists benchmark results for which $64 \times 64 \times 64$ spheres with the same radius were generated on-the-fly and tested against a cone with an offset from the origin and non-axis-aligned normal, as well as an equivalent frustum – that is, a frustum with vertical and horizontal “field of view” that matches the cone angle.

All sphere-cone intersection tests were slower than the sphere-frustum. But building a view transformation matrix from the cone normal and origin to transform the origin of the sphere (included in the measurement) to then test against the cone axis-aligned to $+Z$ with the peak at the origin is actually significantly faster than the sphere-frustum test.

This is unexpected as transforming the point requires a matrix-vector multiplication and float division which should outweigh the common sphere-cone intersection test in terms of floating-point operation count. This performance improvement is likely due to the heavy vectorization of the matrix multiplcation code, done by the compiler.

```
; 577  : Vector3<float> transformPoint(const Vector3<float>& vector) const {
    [...]
    movss  xmm1, DWORD PTR [r8+4]
    mov  r10, rdx
    movss  xmm2, DWORD PTR [r8+8]
    mov  rdx, rcx
    movss  DWORD PTR $T3[rsp], xmm0
    [...]
    movaps xmm0, xmm1
    mulss  xmm0, DWORD PTR [rdx+rax]
    addss  xmm0, DWORD PTR [rax]
    movss  DWORD PTR [rax], xmm0
    [...]
```

Listing 4.1: Excerpt of the assembly code generated by MSVC 15 (/O2) for matrix multiplication.

Mean Time	Description
10.03 ± 0.12 ms	Point - Cone View
9.79 ± 0.12 ms	Point - Cone Cos
9.44 ± 0.11 ms	Point - Cone Tan
8.64 ± 0.13 ms	Point - Frustum Baseline (Batched)
8.43 ± 0.13 ms	Point - Frustum Baseline
7.90 ± 0.08 ms	Point - Cone View (Batched)

Table 4.2: Intersection of point with sphere benchmark; Mean of 10 iterations with 10 passes over 64^3 objects per iteration.

Listing 4.1 shows that indeed the generated assembly code contains instructions like `mulss`[2] and `addss`[3]. These instructions are able to multiply up to four floats from memory or an 128 bit register with another 128 bit register in a single instruction cycle.

Point Intersection

Table 4.2 displays results of benchmarking the point-cone intersection code similarly to the sphere-cone benchmark setup. The exceptions are the benchmarks denoted with "(Batched)". Inspired by the result of the cone sphere benchmark that used the matrix transformation, these batch allocate an array of 64 vectors upfront and then fill it with the on the fly data. The test is then performed on this array. In the cone-view case, we iterate over this array to apply the transformation to all the vertices in the batch and then iterate over it again to perform the test against the Z-axis aligned origin cone.

While again, this batched method is slightly faster, there are manually vectorized point frustum test implementations that would likely outperform it.

AABB Intersection

As intersection of axis aligned bounding boxes did not seem very widely used yet, I did not know what to expect here. While table 4.3 shows the results of the benchmarking the AABB-cone test against a AABB-frustum test, do keep in mind, that the frustum test is a *very* naïve implementation.

As frustum culling has been around for a while, there are many implementations that are way more efficient. As Unreal Engine 4 implements their view culling with frustums and AABBs, the cone test will be more fairly matched there.

The results show that the optimization discussed in section 3.3.3 did yield an small improvement over the straight forward implementation. Further optimization attempts, like precomputing certain values in batches, probably either did not vectorize or led to memory reads that did not trade off well with the hoped gain by vectorization.

[2] `http://www.felixcloutier.com/x86/MULSS.html`
[3] `http://www.felixcloutier.com/x86/ADDSS.html`

Mean Time	Description
59.41 ± 0.57 ms	AABB - Frustum (Baseline)
37.06 ± 0.40 ms	AABB - Cone
33.78 ± 0.33 ms	AABB - Cone (Optimized)

Table 4.3: AABB-cone benchmark; Mean of 10 iterations with 10 passes over 64^3 objects per iteration.

While we cannot assume testing intersection against a cone is generally faster, it seems to be at least comparable. This makes testing it in a real-world environment worthwile.

Triangle Intersection

As I was unable to find a simpler method, the only trick I was able implement was transforming the triangle vertices so that the test can instead be performed against a Z-axis aligned cone. This allows precomputation of some dot products. The result table 4.4 shows that the improvement this has over the non-axis-aligned method is minor and still far off from the triangle-frustum intersection test.

As for GPU culling, the vertices may already be view tranformed, there is a benchmark which precomputes the view transformation once and then performs the test against the Z-axis-aligned cone directly to measure without the transformations. It should not be compared to the others, but indicates that the frustum test due to its simplicity is always faster.

Mean Time	Description
15.58 ± 0.12 ms	Triangle - Cone
12.02 ± 0.10 ms	Triangle - Cone View
(8.71 ± 0.08 ms)	(Triangle - Cone View without transformation)
2.19 ± 0.08 ms	Triangle - Frustum (Baseline)

Table 4.4: Triangle-cone benchmark; Mean of 100 iterations with 10 passes over a mesh with 188928 vertices.

4.2 CPU View Culling

As the geometric overlap tests were only benchmarked in a very controlled environments, but the goal of the thesis was to test for real-world applicability, the view culling was implemented in Unreal Engine 4.

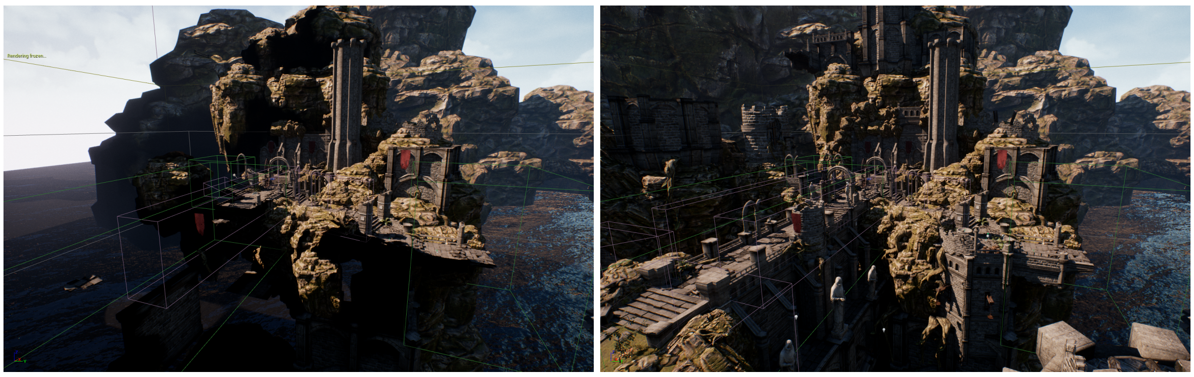

(a) Cone culling turned on. (b) Cone culling turned off.

Figure 4.1: Cone Culling in UE4 (camera for which the scene is culled is in the top-left)

4.2.1 Implementation in Unreal Engine 4

Setup

EpicGames provides access to assets and the "Elven Ruins" level of their game Infinity Blade[4]. I used this level as a real-world example.

The VR headset used during the performance tests was an Oculus Rift CV1. The PC used to for rendering was the same as for the other benchmarks, specified in the introduction to this chapter.

For performance comparison, the camera was unlocked from the HMD and ran through a camera animation, once with frustum culling and once with cone culling. Both times, the game is profiled with the UE4 profiling tool which is described in "Profiling in UE4".

Figure 4.1b shows a view of the level and figure 4.1a the same scene cone-culled for a camera located in the top-left of the image. To keep the culling from updating while repositioning the view, the `FREEZERENDERING` console command was used [Hob17].

Profiling in UE4

Unreal Engine provides a built-in profiling tool, which already gathers information like frame time, number of culled primitives, duration of frustum culling and much more. This tool can be used to collect data and later convert certain values from the data into a CSV-file or view it directly in the editor window.

It is important to note that the animation is running asynchroneously to the frame count; it may be sampled differently in one from the other run, the camera pose may therefore be different in one run on one frame for the Frustum - AABB culling comparison to the Cone - Sphere culling, because of different interpolation. As the output data is per-frame and does not specify an absolute elapsed time per frame, it is necessary to

[4]https://www.unrealengine.com/marketplace/infinity-blade-plain-lands

cumulate the frame times for every frame to later be able to align the data of the runs using elapsed time instead.

The data was then imported into the KNIME Analytics Platform[5] and prepared to finally be input into R[6] via their integration to draw a plot with ggplot 2[7].

Results

The graph for the amount of per-frame culled primitives (4.2a) indicates that cones are way less acurate than frustums for the Oculus Rift. The projection matrix for one of the eyes has a field of view of around 42° to the left, 36° to the right, 47° up and 41° down. To be able to use a circular cone, I used the maximum angle of each direction, which leads to alot less accuracy and is one of the causes for why cone culling performs so much worse here.

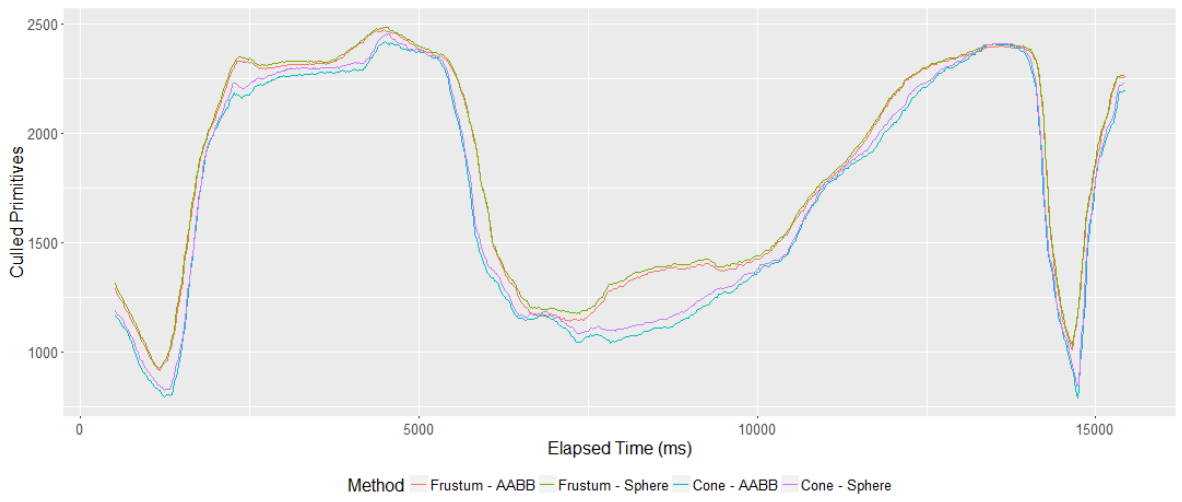
Instead, a solution would be to store scaling factors for each direction (up/down/left-/right) with the cone to apply to the primitives that are culled depending on whether they are below or above the normal or right or left of it. This only really works with axis aligned cones, but as the sphere intersection algorithm does the view space transformation, it would work well there. For non-axis aligned cones, up and right vectors corresponding to up and right of the camera of the cone could be stored. The latter would add significant computation cost to the intersection.

Graph 4.2b which shows the measured frame times does not imply any significant improvements either. With my hardware, all methods overshoot the limit of 13 ms per frame and miss vsync for this scene, which requires the engine to wait for the next frame, causing a frame miss.

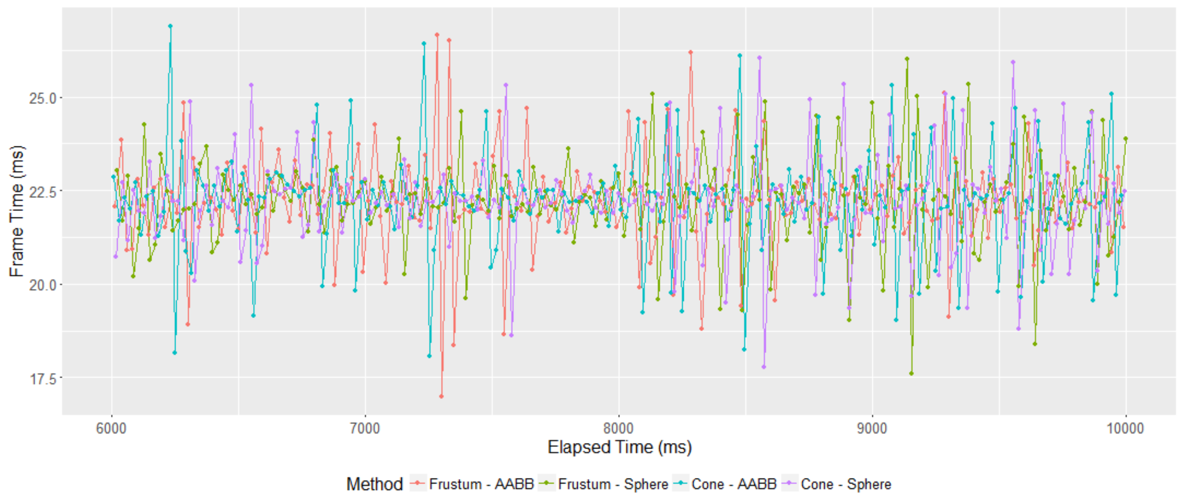
Finally figure 4.2c shows the time spent on view culling with which we can conclude that cone culling is generally more expensive. While individually, the intersection methods were able to hold up against my naïve implementations of the equivalent Frustum intersection methods, they do not stand a chance against a well optimized implementation like the one in Unreal Engine. The culling here could be improved by not testing independent views, but the stereo view with dual cones. While this may be able to yield a factor of 2 of improvement, that would still not make it quite as fast as the frustum culling.

Even with the view cone being specified very conservatively, on the Oculus Rift pop-in was still visible. This seems to be the case because the entire image rendered to the HMD's screen is visible through the lenses as opposed to the HTC Vive, where certain areas are actually not visible.

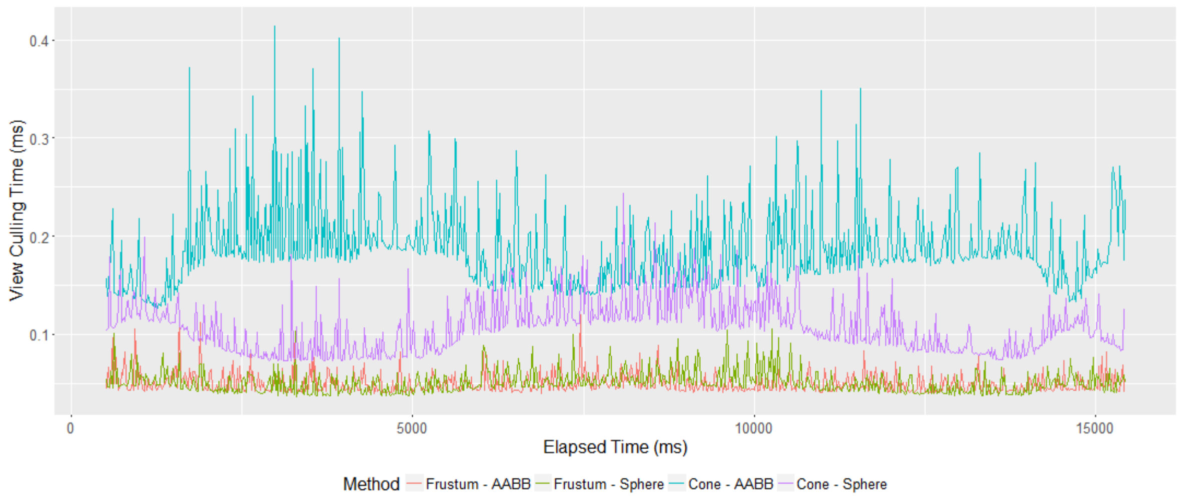
[5]https://www.knime.com/
[6]https://www.r-project.org/
[7]http://ggplot2.org/



(a) Comparison of number of culled primitives.



(b) Comparison of frame times.



(c) Comparison of view culling times.

Figure 4.2: Unreal Engine profiling results.

4.3 GPU View Culling

With GPU culling we can only save the cost of rasterizing primitives that are invisible when seen through the HMD lenses. It would *not* save the cost of the fragment shading, as this is already avoided by techniques such as the stencil mesh, which is described in section 2.2.2.

As the triangle-cone intersection math is significantly more complex than the math for triangle-frustum intersection and – as shown in section 4.1.1 – the method performs worse accordingly, GPU culling with cones seems to not be worth implementing at least on a primitive level.

The chances of success are already very low and as Unreal Engine does not already have a GPU culling pipeline yet, its implementation would have grown out of scope of this thesis.

5 Conclusion

The goal of this work was to test culling with cones for viability in real-world environments, as it promised a more precise view culling result.

Required intersection algorithms were gathered or developed to independently compare their performance against intersection algorithms with conventional frustum volumes. While triangle primitives are more complex to test for intersection, point, sphere and AABB shapes are comparable to frustum tests in terms of efficiency, but my own implementations were only able to stand up to unoptimized versions of the frustum intersection tests.

This does not allow for more efficient object based culling on the CPU with cone views yet as the results of the implementation in Unreal Engine 4 showed.

5.1 Restrictions

When culling objects outside of the view cone, we cannot reuse the left or right eye render for the common rectangular mirror image (see section 2.1.2).

On the HTC Vive, using the distorted and masked view of the right or left eye is not uncommon. Under these circumstances the described method could be used. Similarly, on mobile devices where the screen of a smartphone is used as the VR display, we do not need a mirror view and therefore a circular view is not a problem. With Timewarp on Oculus, additional pixels are required at the edge of the image, as they potentially get warped further towards the center of the view. We can accommodate for this case by increasing the cone angle or applying a negative offset along the direction of the normal to increase the tolerance of culling.

During profiling in Unreal Engine, I made the discovery that the assumption of only seeing a circular cutout cannot be made for all headsets, as in some headsets this circular cutout is larger than the entire distorted image. Here we notice pop-out and pop-in when trying to apply the cone culling, as all rendered pixels are essentially visible.

This and the worse performance in comparsion to frustum view culling restricts the use of this method enough to conclude that it is not real-world applicable, at least not in its current form.

5.2 Optimization Potential

While this work contains solutions and implementations for intersection of cones with all primitives needed for culling, the implementations themselves may be further improved through manual SIMD/vectorized implementations to potentially better compare against the performance of frustums. Additionally, when transforming to Z-axis-aligned cones that have the apex at the origin, we may use dual quaternions instead of a matrix transformation.

Quaternions may even have further use, as they may be able to describe cones, spheres and line segments projected onto a sphere. Both a sphere and a cone would appear as circles when projected onto a sphere with center at the cone's apex. During my research, I came across [MS05], which shows that certain geometrical shapes can be represented by unit quaternions. In such a unit quaternion based geometrical space, there may be an easier way for intersection.

Representing a line segment in such a space would probably require some kind of interpolation between two quaternions, just like a line segment can be seen as interpolating between two points. SLERP (spherical linear interpolation), which is such an interpolation method, requires trigonometric functions, which makes it questionable, whether this approach would be efficient enough, but there may be a way to drop the sine computations as the location of the intersection is not interesting for culling.

Whether something like this is efficient will depend on how costly it is to project into such a space, though.

As the accuracy was worse than with frustum culling, as described in 4.2.1, it will be necessary to find efficient intersection methods for non-circular cones. While cones with oval baseshapes are simple to test in view space (i.e. after transforming the problem so that the cone is axis aligned with apex at the origin), as this is an additional two scalar multiplications, in world space, where culling is usually applied on the CPU, it requires significantly more work.

5.3 Future Work

As no more efficient triangle intersection method was found and Unreal Engine does not implement a GPU culling pipeline yet, GPU culling was not further explored in this work, so that finding a more efficient solution for triangle-cone intersection, especially for axis-aligned cones for view space, or even in NDC space to improve clipping, could be an interesting extension to this work. Even though I was unable to find such an intersection test in this work myself, the assumption of view or NDC space *should* allow making the intersection much more efficient.

Cones may not only be useful for virtual reality. They may be more generally applicable to the "view" volumes of spotlights during rendering of shadow maps for example. Similarly, light-view culling like this could be used to determine whether an object is even affected by a spotlight, which could potentially allow for an interesting new take on clustered shading, where now you could define a per-object light index list of spotlights.

I would also be very interested in seeing if the Cone-AABB test can be used to improve Voxel-Cone-Tracing [CNS+11] by using AABBs to encapsulate denser parts of the sparse voxel octree structure, possibly allowing for easier precomputation of separate meshes that could even be dynamic.

A Notation

A quantity denoted as lowercase letter

$$\vec{x} \in \mathbb{R}^3$$

refers to a **three dimensional vector**, while x refers to a scalar value. As we generally deal with three dimensional space, $\mathbb{R}^3$ is often ommited for brevity and readablity.

$$\vec{x}_x, \vec{x}_y, \vec{x}_z, \vec{x}_i$$

refer to the first, second, third component and the component with 0-based index i of the vector $\vec{x}$.

$$|\vec{x}|$$

refers to the **norm or length of the vector** $\vec{x}$ and

$$\vec{x}^2 = |\vec{x}|^2 = \vec{x} \cdot \vec{x}$$

is a shorthand notation for the **squared length of the vector** $\vec{x}$, or dot product with itself.

Dot products are explicitly indicated with $\cdot$, while the operator for scalar products is implied. Greek letters, e.g. ρ, refer to angles and captial letters, e.g. C to sets or volumes.

Bibliography

[AM00] Assarsson, Ulf ; Moller, Tomas: Optimized View Frustum Culling Algorithms for Bounding Boxes. 5 (2000), 07

[Car13] Carmack, John: *Latency Mitigation Strategies.* https://web.archive.org/web/20131029211812/http://www.altdevblogaday.com/2013/02/22/latency-mitigation-strategies/. Version: Februar 2013

[Car17] Carmack, John: *Oculus Connect 4 Keynote: Carmack Unscripted.* https://youtu.be/vlYL16-NaOw?t=51m50s. Version: Oktober 2017

[CFR+14] Carmack, John ; Forsyth, Tom ; Ribble, Maurice ; Dolan, James ; Kilgard, Mark ; Songy, Michael ; Uralsky, Yury ; Hall, Jesse ; Lottes, Timothy ; Fredriksen, Jan-Harald ; Gustavsson, Jonas ; Holmes, Sam ; Williams, Nigel ; Hector, Tobias ; Koch, Daniel: *OVR_multiview extension specification.* https://www.khronos.org/registry/OpenGL/extensions/OVR/OVR_multiview.txt. Version: Oktober 2014

[CNS+11] Crassin, Cyril ; Neyret, Fabrice ; Sainz, Miguel ; Green, Simon ; Eisemann, Elmar: Interactive Indirect Illumination Using Voxel-based Cone Tracing: An Insight. In: *ACM SIGGRAPH 2011 Talks.* New York, NY, USA : ACM, 2011 (SIGGRAPH '11). – ISBN 978–1–4503–0974–5, 20:1–20:1

[Cor17] Corp., StarVR: *StarVR - Panoramic Virtual Reality Headset.* https://www.starvr.com/. Version: November 2017

[Dem16] Demoreuille, Pete: *Optimizing the Unreal Engine 4 Renderer for VR.* https://developer.oculus.com/blog/introducing-the-oculus-unreal-renderer/. Version: 2016. – [Online; abgerufen 11.07.2017]

[Dev18] Deveria, Alexis: *Can I Use – WebVR.* https://caniuse.com/#feat=webvr. Version: Januar 2018

[Ebe00] Eberly, David: *Intersection of a Triangle and a Cone.* https://www.geometrictools.com/Documentation/IntersectionTriangleCone.pdf. Version: Oktober 2000

[Ebe02] EBERLY, David: *Intersection of a Sphere and a Cone.* https://www.geometrictools.com/Documentation/IntersectionSphereCone.pdf. Version: März 2002

[Epi18] EPICGAMES, Inc.: *Unreal Engine official website.* https://www.unrealengine.com/. Version: Januar 2018

[Hen15] HENNING, Rönne: Zur Theorie und Technik der Bjerrrumschen Gesichtsfelduntersuchung. In: *Archiv für Augenheilkunde* 78 (1915), Nr. 4, 284–301. http://hans-strasburger.userweb.mwn.de/materials/roenne_1915_zur_theorie_und_technik_der_bjerrrumschen_gesichtsfelduntersuchung.pdf

[Hob17] HOBSON, Tim: *Visibility Culling Overview.* http://timhobsonue4.snappages.com/culling-visibilityculling.htm. Version: Januar 2017

[Inc17] INC., Oculus: *Minimum PC specs for Oculus Rift.* https://support.oculus.com/170128916778795/. Version: November 2017

[MS05] MEISTER, L. ; SCHAEBEN, H.: A concise quaternion geometry of rotations. In: *Mathematical Methods in the Applied Sciences* 28 (2005), Nr. 1, 101–126. http://dx.doi.org/10.1002/mma.560. – DOI 10.1002/mma.560. – ISSN 1099–1476

[MT97] MÖLLER, Tomas ; TRUMBORE, Ben: Fast, Minimum Storage Ray-Triangle Intersection. In: *Journal of Graphics Tools* 2 (1997), Nr. 1, 21-28. http://dx.doi.org/10.1080/10867651.1997.10487468. – DOI 10.1080/10867651.1997.10487468

[Pal17] PALANDRI, Remi: *Multi Resolution Rendering.* https://youtu.be/pjg309WSzlM?t=28m10s. Version: Oktober 2017

[Qui17] *Kapitel* V-1. In: QUILEZ, Inigo: *GPU Pro.* Bd. 8: *Efficient Stereo and VR Rendering*. Engel, Wolfgang, 2017, S. 241–252

[Rig17] RIGUER, Guennadi: *Optimizing Graphics Performance: Temporal Foveated Rendering for VR.* https://www.youtube.com/watch?v=gV42w573jGA. Version: Oktober 2017

[Sut68] SUTHERLAND, Ivan E.: A Head-mounted Three Dimensional Display. In: *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I.* New York, NY, USA : ACM, 1968 (AFIPS '68 (Fall, part I)), 757–764

[TAMH17] TOMAS AKENINE-MÖLLER, Eric H. ; HOFFMAN, Naty: *realtimerendering.com - Object/Object Intersection.* http://www.realtimerendering.com/intersections.html. Version: April 2017

[Uhd18] Uhdinger, Jay: *Field of View for Virtual Reality Headsets Explained.* https://vr-lens-lab.com/field-of-view-for-virtual-reality-headsets/. Version: Januar 2018

[Vla15] Vlachos, Alex: *Advanced VR Rendering.* http://media.steampowered.com/apps/valve/2015/Alex_Vlachos_Advanced_VR_Rendering_GDC2015.pdf. Version: März 2015

[Von18] Vondruš, Vladimír: *magnum engine official website.* https://magnum.graphics/. Version: Januar 2018

[Wih17] *Kapitel* VI-1. In: Wihlidal, Graham: *GPU Pro.* Bd. 8: *Optimizing the Graphics Pipeline with Compute*. Engel, Wolfgang, 2017, S. 277–320