Dynamic Shadows of Symmetric Shapes in a 2D Environment
A case study in the field of Computer Graphics

Jesper Glas
Abstract

With modern computer graphics it is easy to think that the computer has some inherent understanding of the 3D world around it. How else could a game let you experience a dynamic world so close to reality in real time. With waves distorting the light hitting the sand in shallow water, or a traffic light illuminating the exhaust fumes from the car in front of you in a dampened red hue. In reality the computer has no clue, all it does is to handle vast streams of binary data in a way that it has been instructed to. What is even more impressive is the ingenuity of some of these instructions that is tailored towards the strengths of the computer.

Let's imagine a table with a candle and a box on it. For us it is obvious that the box will obstruct the light from the candle and cast a shadow on the table, we have seen this behavior with our own eyes. For the computer this is nonsense. It does not know what a candle is, or how light or shadow behaves, it doesn't even know what space is. We need to tell it all of these things in a common language, and that common language is math. We can instead think of the room as its dimensions, let's say 3x3x3 meters. This is something the computer can understand. Furthermore we can tell it where in this newly defined space we want the table to be and how it looks. When we have given it enough information about where everything is in the scene we can start to instruct it on how these things affect each other. Candles cast light, boxes blocks light etc. This is where the ingenuity comes in to play. How do light really behaves mathematically and how can we simulate this efficiently in the computer.

This project will to define one such instruction. The behaviour it simulates is dynamic shadows of symmetrical shapes in a 2D environment. Furthermore it does this in constant time for a variety of shapes that satisfies certain constraints.

With additional such instructions for various effects, each as efficient as possible, the final render can hold quite a bit of realism. This methodology has for a long time been the most common approach in computer graphics, and is still used in a lot of applications today.
# Contents

## 1 Introduction
1.1 The Problem .......................................................... 2
1.2 Terminology ............................................................ 2

## 2 Background
2.1 Object Representation ................................................. 3
2.2 Light ................................................................. 3
  2.2.1 Ambient Light .................................................. 3
  2.2.2 Directional Light .............................................. 3
  2.2.3 Point Light ...................................................... 4
2.3 Shadows ............................................................... 4
  2.3.1 Shadow Volume .................................................. 4
  2.3.2 Silhouette ....................................................... 4
2.4 Rendering Speed ...................................................... 4
  2.4.1 Offline Rendering .............................................. 4
  2.4.2 Real-Time Rendering ......................................... 4
2.5 Limitations ........................................................... 5

## 3 Method
3.1 Circle ................................................................. 5
  3.1.1 Circle Under Directional Light ............................... 5
  3.1.2 Circle Under Point Light ..................................... 6
3.2 Development Process ............................................... 6

## 4 Requirements
4.1 Math Component ..................................................... 7
4.2 Vector Class .......................................................... 7
4.3 Line Class (Edge) ..................................................... 7
4.4 General Shape ......................................................... 7
4.5 Fixed Shapes .......................................................... 7
  4.5.1 Fixed Square .................................................... 7
  4.5.2 Fixed Triangle ................................................... 7
  4.5.3 Trigonometric Algorithm .................................... 7
  4.5.4 Naive Algorithm ............................................... 7

## 5 Design
5.1 Math Component ..................................................... 8
5.2 Vector Class .......................................................... 8
  5.2.1 Addition and Subtraction ..................................... 8
  5.2.2 Scalar Multiplication and Division ......................... 8
  5.2.3 Magnitude ....................................................... 8
  5.2.4 Dot Product ...................................................... 8
  5.2.5 Angle Between Vector and X-Axis .......................... 8
  5.2.6 Angle Between Two Vectors ................................. 9
  5.2.7 Rotate Points .................................................... 9

April 21, 2023
5.2.8 Point-to-Line Distance ........................................ 9
5.3 Line Class .......................................................... 9
5.4 General Shapes ....................................................... 9
5.5 Fixed Shapes .......................................................... 10
  5.5.1 Fixed Square Properties ...................................... 10
  5.5.2 Fixed Triangle Properties .................................. 10
5.6 Fixed Square Algorithm .......................................... 11
  5.6.1 Base Case (Near Edge) ...................................... 11
  5.6.2 Extended Case (Far Edge) .................................. 11
  5.6.3 Algorithm .................................................... 12
5.7 Fixed Triangle Algorithm ....................................... 12
  5.7.1 Base Case (Near Edge) ...................................... 12
  5.7.2 Extended Case (Far Edge) .................................. 13
5.8 Fixed Shape to General Shape ................................ 14
  5.8.1 Near Edge .................................................... 14
  5.8.2 Far Edge ..................................................... 14
5.9 General Shape Algorithm ....................................... 15
  5.9.1 Tangent Point ................................................ 15
  5.9.2 Edge Face Direction ....................................... 16
  5.9.3 Trigonometric Algorithm ................................ 16
  5.9.4 Naive Algorithm ............................................ 16

6 Implementation ....................................................... 17
  6.1 Vector ............................................................. 17
    6.1.1 Operator Output Stream .................................. 17
    6.1.2 Operator Addition/Subtraction ............................. 17
    6.1.3 Operator Scalar Multiplication/Division .................. 17
    6.1.4 Magnitude .................................................. 17
    6.1.5 Unit ......................................................... 17
    6.1.6 Angle ....................................................... 18
    6.1.7 Angular Distance ......................................... 18
    6.1.8 Rotate ...................................................... 18
    6.1.9 Line Point Distance ..................................... 18
    6.1.10 Flip ....................................................... 18
  6.2 Line Class ........................................................ 18
  6.3 Shape Class ....................................................... 18
    6.3.1 Properties ................................................ 18
  6.4 Tangent Point .................................................... 19
  6.5 Facing Edge ...................................................... 19
  6.6 Blocking Edge (Trigonometric) ................................. 19
  6.7 Blocking Edge (Naive) ......................................... 19

7 Evaluation ............................................................. 20
  7.1 Note About Run Times .......................................... 20
  7.2 Measurements ..................................................... 20
    7.2.1 Total Run Time of Variation .............................. 20
    7.2.2 Log Math Function Calls ................................ 20

April 21, 2023
## Dynamic Shadows of Symmetric Shapes in a 2D Environment

### CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2.3 Math Function Run Time</td>
<td>21</td>
</tr>
<tr>
<td>7.2.4 Compound Score</td>
<td>21</td>
</tr>
<tr>
<td>7.2.5 Comparison</td>
<td>21</td>
</tr>
<tr>
<td>8 Result</td>
<td>21</td>
</tr>
<tr>
<td>8.1 Total Run Time</td>
<td>21</td>
</tr>
<tr>
<td>8.2 Compound Score</td>
<td>21</td>
</tr>
<tr>
<td>8.2.1 Detailed Analysis</td>
<td>22</td>
</tr>
<tr>
<td>8.3 Visual Results</td>
<td>23</td>
</tr>
<tr>
<td>9 Conclusion</td>
<td>23</td>
</tr>
<tr>
<td>9.1 Improvements</td>
<td>23</td>
</tr>
<tr>
<td>9.2 Final Thoughts</td>
<td>23</td>
</tr>
<tr>
<td>10 Future Work</td>
<td>24</td>
</tr>
<tr>
<td>11 Appendix A (Visualization)</td>
<td>25</td>
</tr>
<tr>
<td>11.1 Requirements</td>
<td>25</td>
</tr>
<tr>
<td>11.2 Design</td>
<td>25</td>
</tr>
<tr>
<td>11.3 Implementation</td>
<td>25</td>
</tr>
<tr>
<td>12 Appendix B (Math Component/Benchmark)</td>
<td>26</td>
</tr>
<tr>
<td>12.1 Requirements</td>
<td>26</td>
</tr>
<tr>
<td>12.2 Design</td>
<td>26</td>
</tr>
<tr>
<td>13 Appendix C (Code)</td>
<td>27</td>
</tr>
<tr>
<td>13.1 Environment</td>
<td>27</td>
</tr>
<tr>
<td>13.2 Testing</td>
<td>27</td>
</tr>
<tr>
<td>14 Appendix D (Demo Software)</td>
<td>28</td>
</tr>
<tr>
<td>14.1 Requirements</td>
<td>28</td>
</tr>
<tr>
<td>14.2 Design</td>
<td>28</td>
</tr>
<tr>
<td>15 Appendix E (Test Machines)</td>
<td>29</td>
</tr>
<tr>
<td>15.1 Test Machine 1 (Ubuntu 20.04 LTS)</td>
<td>29</td>
</tr>
<tr>
<td>15.2 Test Machine 2 (MacOS)</td>
<td>30</td>
</tr>
</tbody>
</table>

April 21, 2023
1 Introduction

Data stored on a computer is stored in binary. We can tell the computer how to interpret the data. An example would be text file, where the computer interprets the sequence of bits as letters of the alphabet. Programmers have told the computer how to interpret the binary data in the context of a text file.

Modern computers are powerful machines and can handle large quantities of binary data very fast, this allows for more complex contexts. One such context could be a 3D scene. These are usually defined in a 3D coordinate system, where objects in the 3D space have been given properties such as position, shape, material and movement. The data is still stored in binary form, but the context allows the data to be interpreted differently.

At this point we have a way of interpreting a 3D space stored in binary, but we do not have a way of visualizing it yet. The most common way to visualize data on a computer is through a monitor. The monitor is essentially a 2D grid of pixels where the computer can assign colors to each individual pixel. The bridge between the data stored and interpreted by the computer and what we see on the monitor falls in to the area of Computer Graphics.

A more concrete example would be to render a 3D torus to the monitor. We have provided the computer with a 3D scene in which we’ve placed a torus. To be able to view the torus we need a vantage point in the 3D space, making our monitor a window in to the 3D space we’ve defined. We therefore place a virtual camera in our 3D scene and point it at the torus. This virtual camera can be quite complex, more so in a 3D scene since it has to deal with perspectives, near and far clipping etc [6, p. 142]. These things goes outside of the scope for this report, it’s enough to think of the camera as the observers stationary view port in to the scene.

So far we have a window in to the scene and an object in front of it, but we have no light. The computer inherently doesn’t know how light works. This is another sub area of Computer Graphics, we have to provide the computer with directions of how to light up the object in the scene.

The way we tell the computer how to light up an object is called an Illumination Model. What they essentially do is to determine the color of each point on the surface of our objects based on relevant properties (e.g. shape of the object, position, color and intensity of the light source etc). It does this by combining the effects of various types of light at each point on the surface of the torus, which it then translates to the camera’s point of view before outputting the results on the monitor.

Figure 1 shows a torus shape being lit with various types of light (From left to right: ambient light, diffuse highlights, specular highlights). By combining these light types in various ways we can achieve realistic illumination of the object, from the perspective of the camera, to the monitor.

Figure 2 shows, from left to right, the Lambertian Model [5] and the Phong Model [8] which is the two most common illumination models. The Lambertian model only combines the ambient and diffuse light, giving the impression of a matte surface on the torus. The Phong model combines all three light types, making the torus appear more reflective. Even though the Phong model might produce a more realistic result it carries a heavier computational cost than the Lambert model [6, pp. 268–270]. In certain cases the developer might need to consider the need for that extra realism versus the need for fast calculations which could produce a higher fps in a real-time rendering application for example.
The area of Computer Graphics is vast and the example above only presents a simplified explanation of how a 3D scene is represented on a computer as well as one of many lighting models which together can make binary data appear as a 3D scene on the monitor. The relevant thing to take away from this introduction is that the computer has no inherent knowledge of a 2D/3D space, the shape of objects or how light behaves in a scene. For it to produce a visual output that have similarities to reality, we have to provide it with the means to do so. Furthermore, if we want the rendered scene to be interactive and moving, the solution needs to be fast and efficient as well. Sometimes at the cost of quality.

1.1 The Problem

One subarea of computer graphics is shadow casting. As mentioned in the previous section, the computer doesn’t know what a shadow is or how it behaves, so we have to tell it. Various methods on how to generate shadows exists, a brief overview of some common methods are explained in section 2.3. In this report we’ll dive in to one of these methods to try and solve one of the subproblems associated with the method.

Computer graphics is not limited to 3D but also encompasses 2D, which is the environment we’ll focus on in this report. 3D graphics is a well studied area, maybe because it’s how we can represent the reality we live in. But many of the concepts can be translated to work in a 2D environment as well.

I’ve been tasked to bring realistic and efficient shadows in to a 2D game. Furthermore, due to the nature of the game efficiency will be more important than flexibility and the objects that will cast shadows will be rather simple in terms of symmetry. This gives me freedom to explore solutions based on existing models, while not being tied to their constraints, in the hopes of finding a more effective solution to this specific problem.

1.2 Terminology

- **vertex, point** - Will be used to describe a point (x, y) on the 2D plane.
- **line, edge** - Will be used to describe the direct path between two points on the 2D plane. These will have a direction and a positive angle relative to the x-axis going through its starting point.
- **shape** - Refers to a shape in 2D.
- **object** - Refers to an object (shape) in 3D.
- **silhouette edge** - Refers to the direct line between two vertices in a shape that blocks the light cast from a light source.
- **near edge** - Refers to the edge intersected by the direct line between the center of a shape and the light source illuminating it. Due to our shape constraints there will never be more than one for any given shape/light source relation.
- **vertex separation** - The term will be used frequently in this report and it represents the angular distance between vertices of a shape in relation to its center point.
- **bounding circle** - A perfect circle around a shape that fully encompasses all of its vertices.
- **normal** - Refers to the direction perpendicular to a line or a plane.
- **primitive** - Refers the simplest sub shape of a larger, more complex, shape.
- **naive** - Refers to an iterative solution where a complex problem is solved by breaking it down to sub problems and solving all of these. Aimed at flexibility rather than efficiency.
- * - Represents multiplication or scalar multiplication (Not to be confused with ·).
- · - Represents dot product between two vectors (Not to be confused with *).
2 Background

The goal for this report is to design and implement a library that can be used in a graphics component for 2D games to simulate shadows. The library will achieve this by finding the silhouette edge of geometric shapes which in turn is used to create shadow areas, similar to the shadow volume method or generating shadows.

This report will limit itself to a solution for a dynamic 2D environment. There will be a focus on efficiency rather than flexibility, while still aiming for a pseudo-general solution to encompass a variety of shapes following certain constraints. The limitations of the shapes will be defined so that all share certain properties which allows for a general algorithm to be applied to them all.

2.1 Object Representation

This report will be closely related to geometry and the use of geometry in the context of computer graphics. Therefore it’s useful to know a little bit about how objects are usually represented by the computer.

In this report we’ll take a Boundary-based approach to object representation, meaning that we’ll let the surface of the object be what defines it. Another method that won’t be explored in this report is a Volume-based approach [2, Chapter 3.1].

The most basic building block of a geometric shape is the vertex. The vertex is a data structure in computer graphics. It holds, at the very least, its own coordinates in 2D or 3D space, but can be extended to hold additional information.

Above the vertex in the hierarchy is what’s usually referred to as the primitive. The most common primitive in 3D graphics is the polygon. The polygon holds some properties that makes it excellent as a primitive with which to build more complex objects. Its represented by three vertices, which together holds information about the polygons position, shape and orientation. A polygon is also guaranteed to have a plane that goes through all its vertices, this surface is very useful when determining how light behaves when hitting the polygon.

More advanced shapes can be constructed from a polygon mesh, where the smoothness of the object can be enhanced by introducing more polygons. By constructing objects in this manner more complex operations can be performed on the complex object by performing them on all polygons that make up the shape [2, Chapter 3.2].

The same methodology can be applied to 2D graphics as well. We can imagine the polygon as a line, if we observe it from a position perpendicular to it’s plane. We can use this line as our primitive to build more complex shapes, and use similar methods that’s used in 3D graphics in a 2D environment.

2.2 Light

There is a variety of light sources used in computer graphics. In this report it’s helpful to have a basic understanding of three of the more common ones. These are the ambient light, the directional light and the point light.

2.2.1 Ambient Light

Ambient light can be considered background light or the mix of reflected light in a scene. In computer graphics this is usually represented as a variable that modifies the intensity of colors. Lets say that we multiply all colors in a scene by an ambience value, then 0 would result in complete darkness with no details on the objects in the scene. At a very low ambience we might be able to distinguish details but not much color. The intensity of ambient light is usually very small since a strong light would realistically cast shadows, for this reason it’s usually represented as a small constant value to give some small details about the objects in a scene even in complete darkness. While not technically a light source it functions like one in the scope of this report, where an ambient constant will be used to reveal outlines of objects in darkness.

2.2.2 Directional Light

Directional light can be considered light from a light source so far away that its rays appear parallel to each other when they reach our scene. A common example for this is sunlight. As we know sunlight is able to cast shadows, which is why we need a direction of the rays in order to simulate it in computer
graphics. The simplest way to represent directional light is by the direction of its rays as well as an intensity. The direction of the rays can be used to determine if an object is hit by the light or if it's blocked by another object, the intensity can then be used to intensify the colors of everything hit by the light [6, pp. 270–271].

2.2.3 Point Light
Point light can be used to represent the effects of a light bulb in a scene. The light bulb sends out rays of light in all directions, with a falling intensity the further away from the light source it gets. To represent this we need, at minimum, the position of the light source, and the attenuation. The direction of the light will be relative to the lights position [6, pp. 270–271].

2.3 Shadows
In the real world it's hard to imagine light without shadow, they're two sides of the same coin. In Computer Graphics that's not necessarily the case. While there exists methods that handle both illumination and shadows at the same time, such as Ray-Tracing [11], the earlier methods developed worked separately from each other. This meant that we could have a scene where there was only illumination without shadows or vice versa. In this report we'll dive deeper into one of these methods, called shadow volumes [1].

2.3.1 Shadow Volume
To understand the goal of this report it's helpful to have a basic understanding of a method of generating shadows called Shadow Volumes. In the case of a 3D scene each light source will cast rays through each vertex of each objects silhouette towards infinity or a solid object like a wall. We will explore the silhouette more thoroughly in the next section, for now imagine it as the flat silhouette of the object from the lights point of view.

If we imagine all of these rays as lines, between the silhouette of the shadow-casting object and infinity, these rays will encompass a volume. Everything inside this volume is considered to be in shadow.

This method can easily be applied to a 2D scene by downgrading everything by one dimension. The silhouette will be represented in one less dimension making the shadow volume a shadow area instead. This area will make up the shadows for that light source in a 2D scene.

2.3.2 Silhouette
The silhouette, or silhouette edge, is the bounds of an object from the lights point of view. This is represented in one dimension less than the objects itself.

So if we imagine a 3D scene of a globe, illuminated by a single light source. The silhouette edge of the 3D globe will be the 2D area, in this case a circle, that blocks the view of the light source.

If we instead imagine this scene in two dimensions, like we are looking straight down in to the room, we'll see a circle illuminated by a light source. From the lights point of view, positioned on the 2D plane, the silhouette edge is now just a line between the outermost points of the circle.

Finding this silhouette is considered to be one of the bottlenecks for the Shadow Volume algorithm.

2.4 Rendering Speed
As we add more and more effects to our scene, so grows the amount of calculations the computer needs to perform in order to produce a result at each frame. The importance of this varies depending on the use case for the application.

2.4.1 Offline Rendering
Offline rendering can often be characterized as quality over speed. The scene is predefined and the results are not meant to be viewed at the speed its being rendered. This is what's commonly used in CGI effects (Computer Generated Imagery) or computer generated animations. A single frame can take days to produce which allows for heavy and numerous models to be applied to the scene.

2.4.2 Real-Time Rendering
Real-time rendering is not the direct opposite, since quality is always somewhat important, but up to a certain degree favors speed over quality. Real-time
rendering is what's used in computer games, where input from a player results in a dynamic scene which the computer needs to adapt to. For this reason it's important that the computer can compute each frame fast enough to give the illusion of smooth movement.

One way of increasing the quality of the results is to produce more efficient methods used to render each frame. In this report we'll try to improve a method of shadow generation by tailoring it to the needs of our application, and hopefully achieve a more efficient solution without compromising its quality.

2.5 Limitations

There is some additional challenges to shadow generation that won't be addressed in this report. The first being self shadowing, where parts of an object can cast shadows on other parts of the object. The second one is open edges, where the edges or surfaces of an object is not guaranteed to close the object. This could allow light to reach inside the object which makes it a lot more complicated. These problems are not addressed in this report since constraints on the shapes will prevent them from happening.

3 Method

The first part of this section will aim to explain how the circle, and by extension trigonometry can be used to solve this problem. It will also explore how the circle relates to a set of other shapes that can utilize its properties to find its silhouette edge.

The second part will briefly explain the steps this report will take in order to define a general shape with an accompanying algorithm for calculating the silhouette edge.

3.1 Circle

The circle is an interesting shape and will be important for the algorithm developed in this report. As mentioned in the Introduction, complex shapes are represented by a number of interconnected, simpler shapes called primitives. In this report we'll call this primitive an edge, which is represented by the straight line going between two points in 2D space.

To make a circle we'll need a lot of short edges to give the illusion of a round shape. This approach is common in computer graphics and a trade off between performance (Lower amount of primitives means less calculations in each frame resulting in faster performance) and quality (A higher count of smaller primitives results in smoother surfaces at the cost of speed) has to be made. If we instead look at the circle mathematically it's very easy to represent it in 2D space, all we need is a center point and a radius to its border. What's even better is that we can use trigonometry to find all points along it's circumference as long as we know these two properties.

It's this principle that is the starting point of the trigonometric algorithm. As long as we ensure that the shapes all have a certain level of symmetry to it's bounding circle, then we can use trigonometry as a starting point and then adjust the results based on how close our shape is to a circle.

3.1.1 Circle Under Directional Light

When a circle is hit by directional light the silhouette edge will always go through the center point of the circle as long as we assume that the rays run parallel to each other and the circle is perfectly round.
First we find the direction perpendicular to the rays of light. If we then travel in that direction (and the inverted) from the center point by the radius of the circle we’ll find the points that make up the silhouette edge. The angle of the rays will determine the shape of the shadow cast behind the circle.

With this information we can define the span of our algorithm. At one end the silhouette edge is close to running through the center point, but never quite reaching it. On the other end the light is close enough for a single primitive to represent the silhouette edge.

3.1.2 Circle Under Point Light

When subjecting the circle to a point light we are faced with some additional challenges not present under directional light. We can still find symmetry between the outermost points of the silhouette and the angle of the light source. But we can no longer say that the silhouette edge will go through the center point. In fact, we can guarantee that it never will.

The further away the light source is the closer the silhouette edge gets to the center point. This will rarely occur since we often have an outer bound due to attenuation. In other words, the light will fade to the point where it won’t reach the shape long before the silhouette edge gets close to the center point.

We need to find a correlation between the position of the silhouette edge and the distance to the light source. This because as the light source gets closer to the circle the silhouette edge moves further away from the center point. When the light source is close enough to our circle, a single primitive will make up the silhouette edge of the shape. How close this is depends on how many edges we use to approximate the circle shape.

3.2 Development Process

The following three sections of this report (Requirements, Design and Implementation) will detail the build process of the library as well as the algorithm. Any new functionality needed for the library will be developed following a cycle.

1. Define requirements for the functionality or component.
2. Design a solution that fits the requirements.
3. Implement the solution in C++ code.
4. Test and inspect the implementation.
5. Update the Requirement, Design and Implementation sections to reflect changes made.
4 Requirements

The requirements for the library as a whole is that it’s developed to be self contained and not dependent on any other graphics libraries. To ensure compatibility and ease of use, it’s still important that the library is using similar data structures as common graphic libraries such as OpenGL [3] and SFML [10].

4.1 Math Component

The math component will control the implementations and use of all mathematical functions used by the rest of the library. This provides great flexibility in the implementations of these functions as well as the possibility to log mathematical function calls made by the rest of the library.

4.2 Vector Class

The vector class is supposed to be the bottom-layered data structure for the library and serve as the link to other graphic libraries by holding positional data for vertices. It will also be responsible for all linear algebraic functions that can be useful for the rest of the library.

4.3 Line Class (Edge)

The line class will act as the library’s primitives, in a similar manner that polygons work in 3D graphics. It’s main purpose is to represent the silhouette edge as a data structure. In the same manner that three vertices (A polygon) in 3D space is guaranteed to have a plane going through all of them, two vertices (A line) in 2D space is guaranteed to have a line going through them.

4.4 General Shape

The general shape will be designed to take an arbitrary number of vertices to create a symmetric shape. This general shape is then supposed to implement a general algorithm that finds the silhouette edge of the shape. It needs to share some properties with the circle so that trigonometric functions can be used in the general algorithm.

4.5 Fixed Shapes

This report specifies two fixed shapes. The purpose of these shapes is to explore a trigonometric solution for a shape with a low vertex count.

4.5.1 Fixed Square

The fixed square has to be designed so that it follows the requirements of the general shape, since we want to able to represent it as a general shape later on. The algorithm for this shape is meant to provide insight in how the silhouette edge behaves on a symmetric shape with a low vertex count.

4.5.2 Fixed Triangle

The fixed triangle has to be designed so that it fits the requirements for the general shape for the same reason as the fixed square. The algorithm for the fixed triangle will examine how the silhouette edge behaves in a shape that doesn’t necessarily have mirrored halves.

4.5.3 Trigonometric Algorithm

The trigonometric algorithm is the name we’ve given to one of the general shape silhouette edge-algorithms. This variation is meant to contrast the naive algorithm and is therefore limited to as little iteration as possible in order to find the silhouette edge. Instead it needs to rely on trigonometry to find the correct edge-points on the shapes bounding circle and then refine the result to find the shapes silhouette edge-vertices.

4.5.4 Naive Algorithm

The naive algorithm is the name we’ve given to the other general shape silhouette-edge algorithm. This will be based on the flexible, and commonly used, method of iterating over the shapes primitives in order to determine the silhouette edge. This algorithm has less strict requirements since the purpose of it is to compare the performance of the trigonometric variation to relevant data. It should however rely on iteration and avoid trigonometry as much as possible.
5 Design

5.1 Math Component

The math header file will have declarations of all mathematical functions used by the rest of the library. The source file will hold all implementations of the mathematical functions. In this project the source file will call the math library for all functions, but this solution gathers all implementations in one place which allows for flexible optimisation if needed.

In the code for this report the math component is extended to log function calls for benchmarking purposes. This extended functionality is not meant for the final version of the library, more information about this extended version can be found in the Appendix B (Section 12).

5.2 Vector Class

The vector is designed to be as simple as possible, holding two public variables x and y. Member functions without additional arguments is implementations of linear algebraic formulas relative to origin. This design makes it easy to implement and test the functions as that's how the formulas are usually defined in literature.

Relevant member functions also have a variation of the function that takes a point2D as an argument. This represents a custom origin point. This design maintains an easy way to implement and test the linear algebraic formulas, while providing the flexibility to use it anywhere in 2D space.

5.2.1 Addition and Subtraction

Vector addition and subtraction will be based on Vector Addition in linear algebra [4, p. 3].

\[ V = (v_1, v_2, ..., v_n) \]
\[ W = (w_1, w_2, ..., w_n) \]
\[ V + W = (v_1 + w_1, v_2 + w_2, ..., v_n + w_n) \]

Subtraction will work the same way but values will be subtracted instead. Note the the order of the variables matter in that case. If any of the arguments is a single float value the operation will assume it's a vector where both values are the float value.

Ex. \((1, 2) - 5 \rightarrow (1, 2) - (5, 5)\).

5.2.2 Scalar Multiplication and Division

Scalar operation on the vector will be based on Scalar Multiplication in linear algebra [4, p. 5].

\[ C = 5 \]
\[ V = (v_1, v_2, ..., v_n) \]
\[ C \cdot V = (c \cdot v_1, c \cdot v_2, ..., c \cdot v_n) \]

Scalar Division works in a similar manner but has to check that the scalar value is not 0.

5.2.3 Magnitude

The vector magnitude is equivalent to the vectors length. To find it in 2D space, which is what we'll need for this project, we use Pythagoras Theorem [4, p. 13].

\[ V = (v_1, v_2) \]
\[ ||V|| = \sqrt{v_1^2 + v_2^2} \]

5.2.4 Dot Product

The dot product of two vectors is based on the formula for dot product [4, p. 10].

\[ V = (v_1, v_2, ..., v_n) \]
\[ W = (w_1, w_2, ..., w_n) \]
\[ V \cdot W = (v_1 \cdot w_1 + v_2 \cdot w_2 + ... + v_n \cdot w_n) \]

5.2.5 Angle Between Vector and X-Axis

We can use the \(\tan\) function to get the angle in a single quadrant.

\[ \tan (\theta) = \frac{y}{x} \]
\[ \theta = \arctan \left( \frac{y}{x} \right) \]
5.2.6 Angle Between Two Vectors

We can use the vectors dot product and magnitude to find the angle between two vectors in 2D space. The method is based on the linear algebraic formula for finding the angle between two vectors [4, p. 17].

\[ \theta = \arccos \frac{\text{dotAngle}(v, w)}{||v|| \times ||w||} \]

5.2.7 Rotate Points

To rotate a point around origin we’ll perform a linear transformation using the standard matrix defined in Introduction to Linear and Matrix Algebra [4, p. 48].

\[
R^\theta = \begin{bmatrix}
\cos \theta & -\sin \theta \\
\sin \theta & \cos \theta
\end{bmatrix}
\]

If we then have a point \( p = (x_p, y_p) \) the rotated point \( p^\theta \) is:

\[
\begin{bmatrix}
x_{p} \\
y_{p}
\end{bmatrix} = \begin{bmatrix}
\cos \theta & -\sin \theta \\
\sin \theta & \cos \theta
\end{bmatrix} \begin{bmatrix}
x_{p} \\
y_{p}
\end{bmatrix}
\]

To rotate points we use a formula that assumes that we rotate the point around the origin point. We then use the \( \sin \) and \( \cos \) functions together with the magnitude function to set the x and y points of the rotated point.

5.2.8 Point-to-Line Distance

If we have the line \( L \) that’s represented as the points \( P_1 \) and \( P_2 \).

\[
L = (P_1, P_2)
\]

\[
P_0 = (x_0, y_0)
\]

\[
P_1 = (x_1, y_1)
\]

\[
P_2 = (x_2, y_2)
\]

Then the signed distance between the line \( L \) and the point \( P_0 \) can be determined by the following formula [9, p. 190].

\[
\frac{(x_2 - x_1)(y_1 - y_0) - (x_1 - x_0)(y_2 - y_1)}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}}
\]

5.3 Line Class

The line class is designed as a pure data structure. It will hold a start and end point in the form of \( \text{ds::point2D} \) (\( \text{ds::vec2f} \)) instances.

5.4 General Shapes

The general shape will be designed around a center point (\( \text{ds::point2D} \), vertex count and a radius which represents the radius of its bounding circle. The bounding circle is the smallest circle that can cover all of the shapes vertices. This design guarantees that the vertices will lie along the circumference of it’s bounding circle which helps us find them using trigonometry. Its vertices will be stored...
in a `std::vector` in the ascending order of angular distance from the x-axis going through its center point. Since the shape will be symmetric it will also store the angular distance between its vertices as a member variable.

5.5 Fixed Shapes

The fixed shapes will be implemented in a similar manner to the general shape but only aim to solve the problems related to its own properties.

Since both shapes have a set number of vertices this will be omitted in the constructor of the fixed shapes. Instead it will only take a center point (`ds::point2D`) and a radius to its bounding circle. The vertex separation will be a constant in these shapes since it’s not dependent on a constructor argument like in the general shape.

The purpose of the fixed shapes is to solve sub problems related to the general shapes trigonometric algorithm, for this reason they’ll only implement their own version of this algorithm. Furthermore this algorithm will only be concerned with problems related to the fixed shape and will not make any effort to be more general the necessary.

5.5.1 Fixed Square Properties

Figure 8 shows the properties of the fixed square. The unusual rotation is to help us by ensuring there is a vertex at angle 0 in relation to the center point.
5.6 Fixed Square Algorithm

5.6.1 Base Case (Near Edge)

The first thing we’ll do is to define a base case for the silhouette edge. We’ll define this base case as the two vertices adjacent to a reference point on the bounding circle that has the same angle as the light source in relation to the squares center point. We can find this point by finding the direction of the light source in relation to the center point, and then walk one radius in that direction from the center point.

We can then use modulo calculations on the angle of the light source and the shapes vertex separation to get an overflow value. This value represents the angular distance between the previous vertex and the reference point.

By rotating the reference point counter clockwise we’ll get the first vertex of our near edge. Likewise we can rotate the reference point clockwise by the difference between one vertex separation and the overflow to get the other vertex of our near edge. This process is visualized in figure 10.

This means that we need to expand the area at certain distance/angle combinations. Additionally, by examining Figure 12, we see that this expansion can be achieved in two directions depending on the angle of the light source. Figure 12 shows the scenario when we’ve moved light source resulting in a different silhouette. We can also see that the silhouette edge is going through the center point, something we determined to be impossible for a circle under a point light in Section 3.1.2. This is something we have to keep in mind when designing the general shape trigonometric algorithm.

This means that we need to expand the area at certain distance/angle combinations. Additionally, by examining Figure 12, we see that this expansion can be achieved in two directions depending on the angle of the light source. Figure 12 shows the scenario when we’ve moved light source resulting in a different silhouette. We can also see that the silhouette edge is going through the center point, something we determined to be impossible for a circle under a point light in Section 3.1.2. This is something we have to keep in mind when designing the general shape trigonometric algorithm.
Shifting the focus back to the algorithm at hand we need to determine which point we need to move in case the area needs to be expanded. To do this we'll combine what we've learned so far and start again from the near edge.

**Figure 13** shows a scenario where we've moved all relevant points to the first quadrant using modulo operation on the light source angle. Due to how we've determined the near edge we know that if the light source is in the first quadrant then the starting point of the silhouette edge will be at angle 0, relative to the x-axis going through the squares center point.

We've also added the areas that determines if we need to move a point of the silhouette edge. We've color coded them to their relevant vertex which creates three areas to consider.

The first area is between the green and red line. If the light source is in this area we don't need to move any points of our silhouette edge.

The second area is between the horizontal purple line and the green dotted line. If the light source is in this area we need to move the starting point of the near edge one step clockwise.

The third area is between the red and vertical purple line. If the light source is in this area we need to move the end point of our silhouette edge one step counter clockwise.

### 5.6.3 Algorithm

**Data:** Square and light source position  
**Result:** Silhouette edge initialization; find reference point on bounding circle at same angle as light source relative to center point; use modulo to find overflow between previous vertex and light source angle; rotate reference point clockwise by overflow to find start point; rotate reference point counter clockwise by (vertex separation - overflow) to find end point;

```plaintext
def fixed_square_algorithm(square, light_source):
    # Find reference point on bounding circle
    # Rotate reference point clockwise
    # Rotate reference point counter clockwise
    # Check if in first area
    # Check if in third area
    # Decide if to rotate start or end point
    return silhouette_edge
```

**Algorithm 1:** Pseudocode of the fixed square algorithm

### 5.7 Fixed Triangle Algorithm

#### 5.7.1 Base Case (Near Edge)

The method we used to find the near edge in the fixed square (Section 5.6.1) can be applied to the fixed triangle as well without any modifications.
5.7.2 Extended Case (Far Edge)

If we visualize the near edge case we see that it behaves similar to that of the fixed square. The only difference is the enclosing area that determines if we need to move any of the points of the silhouette edge. The expansion of this area is expected since the angle of the area bounds depend on the angle of the near edge’s adjacent edges, which naturally differs between shapes.

We then assume that the light is positioned so that we need to move one point of the silhouette edge. One scenario for when this could happened is when the light source moves further from the shape, in the direction of the light source (Purple dotted line in Figure 14), until it crosses the green dotted line.

This reveals a scenario which is probably unique, at least among the shapes that we define in this report, where the far edge is far beyond the center point. In fact, it behaves like a near edge, but on the opposite side of the center point. This is visualized in Figure 15.

Although different in shape and areas, the same principle as with the fixed square seems to be true for the fixed triangle as well. We’ll shift the light source by using modulo, just as with did with the fixed square, and inspect the results.
When working with the *fixed square* we called this normalized section a quadrant. This is not accurate when working with the triangle, instead we’ll just call it a *section*. The *section* still holds the properties we wanted by normalizing it, namely that the starting point of the *silhouette edge* is at angle $0^\circ$ relative to the center point. This means that we can use Algorithm 5.6.3 for the *fixed triangle* as well.

### 5.8 Fixed Shape to General Shape

#### 5.8.1 Near Edge

The *fixed shapes* have helped us define a way to find the *near edge* of a shape. We can safely assume that this method is applicable to any shape following the constraints we’ve set up for the *general shape*. As long as the light source is outside of the shape and the length of its edges is $> 0$, then there will always be a span between vertices where the direct line between center point and light source intersects. With this intersection, or rather the point on the bounding circle at the same angle as the intersection, we can find the vertices that make up the *near edge*.

#### 5.8.2 Far Edge

The *far edge* is a bit more complicated. The *fixed shapes* have shown that there is scenarios where the *silhouette edge* is on, and even behind, the center point. This was something that we determined to be impossible for a circle under point light in Section 3.1.2. By combining these insights we can deconstruct the problem and solve it.

In Section 3.1.2 we determined that there is a tangent point on the circle that represents the maximum angle at which a ray of light can hit the circle. If we imagine that circle as a bounding circle for a shape, that point will most likely be between two vertices in the shape. This is similar to the reference point we use to find our *near edge*, only now we have two such reference points, one for each vertex of the *silhouette edge*. Figure 17 is showing the relevant points in finding one of the *silhouette edge* vertices. The same process can be applied on the other half of the shape to find the other vertex of the *silhouette edge*.

The adjustment is easy, we just perform a modulo calculation to get an *overflow* which we then rotate the *reference point* with. The problem is how we determine in which direction to rotate.

If we look at Figure 18 we see that we can define similar areas to those we had when determining the *near edge* for the *fixed shapes*. Unlike with the *fixed shapes* we only have two relevant areas to consider in this case. This is because we know that the light can’t hit beyond the reference point on the bounding circle. If we adjust for this specific shape the limit
becomes the next vertex. Due to the adjustment however we still need to determine if it’s the previous vertex that blocks the light.

We realize that we can redefine the problem to make it easier to understand. What the areas really represents is if the edge that determines them can be hit by light from the light source. So to solve the adjustment problem we have to determine if the relevant edge is hit by light. In this case it’s not. We can see this in two ways. First we can determine that the light source is not in the area enclosed by the green dotted lines. Secondly we can see that a direct ray of light hits the backside face of the relevant edge. This means that the vertex blocking the light, which will make up one of the silhouette edge’s ends, will be the previous vertex.

This leaves us with two subproblems that we need to solve in order for the general algorithm to work. We need to be able to find the maximum tangent point on the bounding circle based on distance and angle of the light source. We also need a way to determine if an edge is facing the light source, that is if a ray of light can hit the outward facing side of an edge.

5.9 General Shape Algorithm

To define an algorithm for the general shapes we first need to solve the subproblems we’ve discovered in the previous section.

5.9.1 Tangent Point

We set up a scenario with as many known variables as possible in the attempt to define an equation for the unknown angle, relative to the center point, that represents the maximum angle of a point on the bounding circle that can be hit by light from the light source. We visualize this in Figure 19.

We are searching for the angle at point C so that the tangent at the unknown point P hits the light source, that will be the outermost point where a ray cast from the light source can hit the circle. We can utilize the law of sine to find this.

\[
\frac{a}{\sin (A)} = \frac{b}{\sin (B)} = \frac{c}{\sin (C)}
\]  

(1)

We need to choose the variables so that the equation only has one unknown. Since we know that a tangent of a circle is always perpendicular to its radius at any point on its circumference we can determine that the angle at point P in the triangle \( CLP \) is 90°. This allows us to set up the law of sine with only one unknown variable.

\[
\frac{r}{\sin (L)} = \frac{d}{\sin (P)}
\]  

(2)

Furthermore we know that \( \sin 90° = 1 \). This makes it even easier to isolate \( \sin L \) on the right side of equal sign which in turn allows us to find \( L \) using arcsin.

\[
L = \arcsin \left( \frac{r}{d} \right)
\]  

(3)

Lastly we use the fact that we can find the last angle of a triangle if we know the other two.

\[
180° = C + L + P
\]

We also know that the angle at point P is 90°.
which gives us the final equation.

\[ C = 90° - L \] (4)

The angle \( C \) is so far normalized, meaning it is calculated as if the light source is at \( 0° \) relative to the x-axis going through the center point. To compensate for the likely case that the light source is not at \( 0° \) we set a reference point on the bounding circle with the same angle as the light source in relation to the center point. We can then rotate this by the angle \( C \), in both directions, to get the start and end point of the blocking edge. The start and end point is now at the maximum angular distance on the bounding circle where the tangent is \(< 90°\).

5.9.2 Edge Face Direction

To determine if an edge is facing the light source we’ll use a formula for Point-Line-Distance from Section 5.2.8.

This formula gives us the minimum distance between a point and the closest point on the direct line between two other points. We can modify this to give us a signed result, meaning that we can determine if a point is in front or behind a line by if the value is positive or negative. With this we have all the components we need to define the Trigonometric Algorithm.

5.9.3 Trigonometric Algorithm

**Data:** Shape and light source position  
**Result:** Silhouette edge initialization;  
if distance to light source is LESS than radius then  
    return near edge as silhouette edge;  
else  
    find reference point on bounding circle at same angle as light source;  
    find angle to tangent on bounding circle perpendicular to light source (circle normalized so light source is on x-axis);  
    set start point by rotating reference point counter clockwise by tangent angle;  
    set end point by rotating reference point clockwise by tangent angle;  
    adjust points to correct vertex depending on edge face direction;

**Algorithm 2:** Psudocode of the general shapes Trigonometric Algorithm

5.9.4 Naive Algorithm

The naive algorithm is much simpler. Due to the constraints on our shapes we know that there is a connected path between the start and end point of the silhouette edge. The simple approach we’ll use is to define a variable which holds if the current edge is facing the light source. We then iterate over the edges of the shape until the variable changes, the start point of this edge will make up one of the edge points of our silhouette edge. After that we continue to iterate until the variable changes again, the end point of that edge will make up the other edge point of our silhouette edge. Lastly we perform a check to see if the first edge faced the light source, if so our silhouette edge is flipped and we need to flip it to the correct orientation.
Data: Shape and light source position
Result: Silhouette edge initialization;
for edge in shape do
  if start point AND end point is set then
    break;
  end
  if edge face direction to light source changes then
    if start point is NOT set then
      set start point of silhouette edge;
    else
      set end point of silhouette edge;
    end
  end
if first edge is facing light source then
  flip silhouette edge;
end
Algorithm 3: Pseudocode of the general shapes Naive Algorithm

6 Implementation

6.1 Vector

The vector class is implemented with the header and source code files vec2f.hpp, vec2f.cpp. It’s as simple as possible holding only its 2D coordinates as public member variables.

6.1.1 Operator Output Stream

The operator<< lets us define a way to print the vector to an output stream. The vector will be displayed in the format: \([x, y]\).

6.1.2 Operator Addition/Subtraction

The addition (operator+) and subtraction (operator−) operators just adds or subtracts the corresponding values in each of the vectors passed in to the function. Note that the order of the vectors is important for subtraction but not for addition.

6.1.3 Operator Scalar Multiplication/Division

The scalar multiplication (operator∗) and division (operator/) operators perform a multiplication/division of each of the vectors elements by the scalar value. For division the order of the arguments matter and the scalar must be the last argument. The function will also throw a runtime error on scalar division by 0.

6.1.4 Magnitude

The vec2f::magnitude (Length) function is implemented as a member function of the vec2f class. It has two variations. The main implementation, without arguments, calculates the magnitude in relation to origin. The second implementation normalized the vector in relation to another point and then calls the main implementation of the function.

6.1.5 Unit

The vec2f::unit (Direction) function is implemented as a member function of the vec2f class. It has two implementations similar to that of the vec2f::magnitude function. The unit function utilizes the magnitude functions to get the length of the vector.
6.1.6 Angle

The `vec2f:angle` function calculates the angle relative to the x-axis going through origin. The angle is in radians. It's implemented in a main variation and a normalized variation. The main implementation utilizes the `cmath::atan2` function which allows us to get the correct angle in all four quadrants. It also makes sure to always return a positive angle by offsetting negative results with $2 \cdot \pi$, which is the equivalent to 360°.

6.1.7 Angular Distance

The `vec2f:dotAngle` function calculates the angular distance between two specified points relative to itself. It utilizes the formula from section 5.2.6 using `cmath::acosf` for the arccos calculation.

6.1.8 Rotate

The `vec2f:rotate` function is implemented as a member function of the `vec2f` class. It rotates the point relative to origin using the formula from section 5.2.7.

6.1.9 Line Point Distance

The `vec2d:lineDistance` calculates the signed distance between itself and a line made up of a specified start and end point. It utilizes the formula from section 5.2.8.

6.1.10 Flip

There are three variations of the flip function. The `vec2f:flipX` returns a `vec2f` with flipped x-value relative to origin. The `vec2f:flipY` returns a `vec2f` with flipped y-value relative to origin. The `vec2f:flip` returns a `vec2f` where both x and y-value have been flipped relative to origin.

6.2 Line Class

The line class serves mostly as a data structure to allow a function to return our definition of a silhouette edge. We've chosen to implement this as a class in favor of the struct which is commonly used for data structure definitions in C++. The reason for this is that we might want to extend the functionality for this class in the future. As the library's primitive there might be benefits to be able to add or modify functionality to it. For this project it's only implemented in a header file.

6.3 Shape Class

The `ds:shape2D` class is what we'll call the data structure that holds our general shapes. With the help of the algorithms defined for the fixed square and the fixed triangle we should be able to define a general algorithm that works for shapes with at least three vertices and up.

6.3.1 Properties

Just as the other shapes the `ds:shape2D` will have a variable for its center point as well as the radius of its bounding circle. It also needs to keep track of how many vertices it has, this will be referred to as its size going forward. We have determined in the past algorithms that the vertex separation is useful, so this will also be a variable calculated in the constructor of the shape.

![Figure 21: Visualizing the properties of the Shape.](image)
6.4 Tangent Point

The `shape2D::getMaxRayAngle` is the implementation of the formula defined in Section 5.9.1(4). The variable L is easily obtainable using arcsin on the shapes center point (C) and radius (R) as demonstrated in Section 5.9.1(3).

6.5 Facing Edge

The `shape2D::getFacingEdge` finds the near edge mentioned in Section 5.8.1.

6.6 Blocking Edge (Trigonometric)

The `shape2D::getBlockingEdge` is the implemented version of the trigonometric algorithm defined in Section 5.9.3.

6.7 Blocking Edge (Naive)

The `shape2D::getBlockingEdgeNaive` is the implemented version of the naive algorithm defined in Section 5.9.4.
7 Evaluation

In order to compare the algorithms we'll need a unit that's measurable and relevant to this project. Time is a relevant factor in real time graphics as it determines how many frames per second the program can produce, making motion look smoother. The most basic benchmark we can make is measuring the time it takes for each of the getBlockingEdge variations to run. This will provide us with a performance overview of the different variations but not much else.

To get more detail we'll log how many mathematical function calls, and what specific function is being called, by each algorithm. With this information together with the run time of each of these functions we should get a similar result as the total run time, but with increased detail.

The later information could then be analyzed in order to optimize the algorithm by picking the best of both variations. This data will also show any potential breakpoints where one method might be better than the other.

7.1 Note About Run Times

Comparing runtimes poses some challenges in order for it to be relevant and accurate. For one, the run time will most definitely vary from run to run depending on other processes running on the machine. This can be mitigated by running a series of tests and compiling an average. This approach will be taken with all time related measurements in this report.

Another challenge is the variations between machines. This one is harder to mitigate since it would require an "average" machine which arguably doesn't exist, or at least is very hard to define. So the results of this report is not applicable to all machines. This is not a major problem since the variations of the algorithms will be compared on the same machine. Furthermore we're not really interested in the actual run time, but more of how they compare between the algorithm variations.

One important factor to consider is how the machine handles trigonometric functions. There are various methods on how manufactures solve this both in hardware and software. Since one of the algorithms is centered around trigonometry it's important to keep this in mind.

The test machines used in this project can in some manner be considered average, see Section 15: Appendix E. They will be ordinary computers with consumer hardware, neither military supercomputers nor microchips.

7.2 Measurements

7.2.1 Total Run Time of Variation

The total run time of each variation will be an average over 1000 runs. The light source will move slightly in an off-centered orbit around the shape between each run. This will subject the algorithm to a varying distances between the light source and the shape.

![Figure 22: Showing the movement of the light source during the test.](Figure 22: Showing the movement of the light source during the test.)

7.2.2 Log Math Function Calls

The math component lets us log what mathematical functions is called between breakpoints in the code. When we benchmark the variations of the algorithm we clear the counters before we run the getBlockingEdge variation and save the results directly after. To make sure that each variation is tested thoroughly we create an average in this case as well. Here we are not interested in run time but instead the amount of function calls to each of the
mathematical functions in the `math` header. We want to subject the algorithms to varying distances and angles, so between each round we rotate the light source in the same manner as we did in the total run time test Figure 22.

7.2.3 Math Function Run Time

The run time of mathematical functions are usually very fast and varies from call to call depending on other processes running on the machine. So for this to be relevant at all we will need an average. The benchmark, see Section 12: Appendix B, runs each function 100,000,000 times with arbitrary, randomized parameters and compiles an average of the run time. The reason for the steep increase in rounds, compared to the total run time test, is because these functions are very fast. This means that small variations has a larger impact on the average run time. We increase the rounds to mitigate this and get an accurate result.

7.2.4 Compound Score

We will define a score that we call compound score for each variation of the algorithm. This will be the results of multiplying the average run time of the mathematical benchmarks with the results of the logged function calls from each variation. The results of this should be similar to the total run time, but with additional details on what components of the algorithm has the largest impact on its total run time. We will gather function call data in the same manner as we gathered total run time data. Running 1000 rounds with a moving light source. This simulates a real world use well enough to be relevant, and keeps the test environment similar between the results we will compare.

7.2.5 Comparison

Lastly we will gather total run time data and compound score data with an increasing amount of vertices. We will start at three vertices since that is the lowest amount of vertices for a shape that we’ve defined. The results will be gathered in a graph where the x-axis is the amount of vertices and the y-axis represents the total run time / compound score.

8 Result

8.1 Total Run Time

The results from the total run time seems to confirm that the trigonometric variation of the algorithm has a more or less constant run time independent of the number of vertices while the run time of the naive version is growing linearly with the amount of vertices. This is the expected behaviour since the naive variation has to check each edge, which is growing at the same rate as the amount of vertices. The trigonometric variation was developed to circumvent this, starting at the circle and then narrowing down its results based on how the shape looked.

![Figure 23: Algorithms compared (1000 rounds each) [Total Run Time/Amount of Vertices]](image)

8.2 Compound Score

The resulting graph of the compound score behaves similarly to the graph of the total run time for each variation. This confirms that the majority of the run time is a result of which mathematical functions are used by each variation. Now that we can trust that both graphs behave similarly we can look in more detail what makes up the run time of each variation.
Dynamic Shadows of Symmetric Shapes in a 2D Environment

8 Result

8.2.1 Detailed Analysis

In Table 1 we can see the average amount of calls each algorithm did to each mathematical function.

<table>
<thead>
<tr>
<th>Vertices</th>
<th>3</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>T</td>
<td>N</td>
</tr>
<tr>
<td>acos</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>add</td>
<td>14</td>
<td>8</td>
</tr>
<tr>
<td>asin</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>atan2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>cos</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>div</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>fmod</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>mult</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>pow</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>sin</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>sqrt</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>sub</td>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>tan</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1: Function calls from each algorithm with arbitrary parameters. (T - Trig, N - Naive)

It’s easy to see that the trigonometric variation is constant as the amount of function calls is the same at different vertex counts. What’s more interesting is the use of trigonometric functions between the variations. The trigonometric variation uses almost all of the trigonometric functions at least once while the naive version only uses atan2 once in a constant manner. Next we will introduce the average run time of each function to see if there is room for improvement in any of the variations.

Table 2: Test machine 1 metrics

<table>
<thead>
<tr>
<th>Function</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sin</td>
<td>31.7599</td>
</tr>
<tr>
<td>cos</td>
<td>30.6941</td>
</tr>
<tr>
<td>asin</td>
<td>40.4035</td>
</tr>
<tr>
<td>acos</td>
<td>46.8141</td>
</tr>
<tr>
<td>tan</td>
<td>54.4701</td>
</tr>
<tr>
<td>atan2</td>
<td>60.4883</td>
</tr>
<tr>
<td>sqrt</td>
<td>22.5253</td>
</tr>
<tr>
<td>pow</td>
<td>40.903</td>
</tr>
<tr>
<td>fmod</td>
<td>41.6389</td>
</tr>
<tr>
<td>add</td>
<td>21.145</td>
</tr>
<tr>
<td>sub</td>
<td>20.8046</td>
</tr>
<tr>
<td>mult</td>
<td>20.8081</td>
</tr>
<tr>
<td>div</td>
<td>22.4972</td>
</tr>
</tbody>
</table>

Table 2: Test machine 1 metrics

The results of the benchmark in Table 2 shows us that the trigonometric functions run slower in general than most other mathematical functions, except for pow and fmod. Given time enough to analyse the variations and find alternative, cheaper solutions to sub problems based on these results could optimize the performance of the algorithms. Further optimizing the algorithm falls out of the scope for this report, but we can safely conclude that we’ve gathered data which would make this process easier.

April 21, 2023
8.3 Visual Results

9 Conclusion

9.1 Improvements

With the results from the mathematical functions benchmark Table 2 we might be able to improve the member functions of the ds::vec2f class. By implementing these with as few and cheap function calls as possible the improvements would influence the total run time of the algorithm.

The algorithm could also be analysed in the hopes of finding alternative, cheaper solutions to sub problems.

If pixel perfect shadows is not a strict requirement then shortcuts in the last adjustment step could be taken. This is especially true if the vertex count is high enough where each individually edge is not visible.

9.2 Final Thoughts

Leaning on the results from the previous section we can conclude that, even though there is room for improvement, it's possible to find a blocking edge in a symmetric shape with at least three vertices in constant time. We can also conclude that this method becomes faster than the naive method when applied to shapes with a relatively low vertex count. We must also conclude that the real world use case for this is limited. Granted that the trigonometric variation is both vertex perfect and more effective than the naive version. But it can only be used on very specific shapes adhering to the symmetrical constraints imposed on the shapes we've defined in this report. It’s also worth noting that the last steps of the algorithm that determines which exact vertex is blocking the light becomes obsolete at a high enough vertex count. When individual edges in no longer visible this last step can be ignored without any noticeable difference at a slight performance increase. Due to it's poor flexibility this algorithm is better suited as a complement to a more flexible solution than being the only solution.
10 Future Work

While the results are satisfying there is a lot of room for improvement and further development. A flaw that became apparent in the later stages of the project is related to one of the earliest decisions made about its design. In Section 4 I set a requirement for the library to be self contained. As a result of this I designed the library to define it’s own data structures, from vertices up to shapes. From a development standpoint this was a good decision since it allowed for self contained development and full control over all data structures used by the library. From a compatibility standpoint this decision was problematic since both this library and most external libraries want to create its own set of vertices, primitives and shapes. A better approach would have been to allow for this libraries vertices to be bound to an external libraries vertices. Using references and pointers this should be fairly easy in C++, albeit time consuming to alter it retroactively.

Another area that I would have liked to explore is extend the library to 3D space. Similar symmetry as a square towards circle should be present when moving from cube to sphere. With the addition of another dimension calculations should be heavier and the use of this libraries functionality might be even more useful. On the other hand a 3D approach would be more complicated to actually use since it would have to pass 3D shadow volumes instead of 2D shadow areas from the CPU to the GPU. In the demo used in this report (Figure 25, 26) the shadow is passed to the GPU as a texture, which is very easy to do. Passing, and using, 3D volumes in the shader is an additional challenge to consider before attempting to extend this library from 2D to 3D.
11 Appendix A (Visualization)

This component is meant to aid in the development of this library by visualizing the various graphical elements and methods defined in it.

11.1 Requirements

This component is responsible for visualizing all of the library's data structure and their related member functions in 2D space. It should also be able to generate the results from multiple scenarios which we call visual test cases.

11.2 Design

The component aims to be as simple as possible. It will have a header file which holds methods to render all of the graphics related classes defined in the library. It will also contain files where "visual test cases" can be defined and generated.

11.3 Implementation

The component utilizes the C++ library matplotlibcpp which is a C++ implementation of the Python library matplotlib [7]. The library functions a lot like Matlab but accepts C++ code for its variables.

The header file visualUtil.hpp contains overrides to the matplotlibcpp::plot function for all of the library's graphics related classes. It also contains methods which define a "default" plot environment and one which saves the plot as an image to a specified location.

Due to the nature of this project two "main" files are defined. One is meant for testing where "visual test cases" can be defined and generated. This is meant to serve as a tool to control the functionality of the library.

The other one is meant for documentation. The scenarios defined here is meant to generate images explaining results and concepts in this report.
12 Appendix B (Math Component/Benchmark)

The math component used in this project has some functionality which is beyond the scope of what is needed for a real world use of this library. The requirement section details what the library needs to function. This appendix will contain information about how the component is extended to handle benchmarks and reveal additional details about the library’s performance.

The decision to extend the math component instead of creating a separate solution is that such a solution would fall outside the scope of this project. It’s more important to have all math related functions in a single component to avoid ambiguity, and said component is also the easiest place to gather function call data from since it’s used by the rest of the library.

12.1 Requirements

To get a more detailed look in to how the library operates and where potential bottlenecks occur this component needs to be able to log function calls to all functions declared in its header file. It will also need to be able to output the data in a useful way.

12.2 Design

To keep it as simple as possible, both to implement and to remove for real world use, additional functions related to the benchmark will be declared and implemented in the math.hpp and math.cpp files. A variable will be declared which can hold function call data.
13 Appendix C (Code)

13.1 Environment

This library is written in C++ as a component for a potential game engine. As such it’s important that it’s easy to build and integrate in to other projects. CMake is used to automate the build process. It’s configured to handle specific dependencies using CMakes FetchContent function which fetches the dependencies from Github during the install process. It’s also configured to build the project as a CMake subproject, allowing it to be integrated in other projects in the same way using the FetchContent function with a reference to its Github repository. Advanced use of CMake falls out of the scope of this project and therefore additional dependencies might have to be installed manually.

13.2 Testing

The project supports unit tests using the GoogleTest library. Each header file has a corresponding unit test file where all relevant methods are tested. Unit tests is a great tool when troubleshooting simple methods and formulas, but quickly becomes tedious when dealing with visual geometry. For this reason a custom visual testing component was implemented, see Section 11: Appendix A. Results are saved as images, with names tied to their corresponding test. These images proved invaluable for inspecting the behaviours of geometry-related methods.

![testSquareShadowConcept](image)

*Figure 27: The result of a visual test case showing the concept of the blocking edge on a square.*
14 Appendix D (Demo Software)

The demo software was developed at the end of the project and serves as a bare-minimum, real-world use case of the library. Accidentally, and a bit late, it showed one of the biggest flaws of the library, namely it’s own creation of vertices instead of the ability to use references to already defined vertices. The demo is built using SFML which has support for shader programming and custom shapes using a data structure called vertex arrays. The shadows are projected on a texture by the CPU and passed to the GPU which uses it to apply light and shadow to the scene.

14.1 Requirements

The demo did not have set requirements other than to showcase the functionality of the library in a real-world use case. It will not win any prizes for functionality but serves it’s purpose is a sufficient way.

14.2 Design

As mentioned it’s designed to showcase the library in a bare-minimum way. It uses a shadow texture on which all shadow areas are projected in each frame and then sent to the shader which applies it as a light multiplier to each pixel of the view.

Figure 28: Demo visualization
15 Appendix E (Test Machines)

15.1 Test Machine 1 (Ubuntu 20.04 LTS)

- **Architecture:** x86_64
- **CPU op-mode(s):** 32-bit, 64-bit
- **Byte Order:** Little Endian
- **Address sizes:** 39 bits physical, 48 bits virtual
- **CPU(s):** 4
- **On-line CPU(s) list:** 0-3
- **Thread(s) per core:** 2
- **Core(s) per socket:** 2
- **Socket(s):** 1
- **NUMA node(s):** 1
- **Vendor ID:** GenuineIntel
- **CPU family:** 6
- **Model:** 78
- **Model name:** Intel(R) Core(TM) i7-6600U CPU @ 2.60GHz
- **Stepping:** 3
- **CPU MHz:** 1871.573
- **CPU max MHz:** 3400,000
- **CPU min MHz:** 400,000
- **BogoMIPS:** 5599.85
- **Virtualization:** VT-x
- **L1d cache:** 64 KiB
- **L1i cache:** 64 KiB
- **L2 cache:** 512 KiB
- **L3 cache:** 4 MiB

<table>
<thead>
<tr>
<th>Function</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>sin</td>
<td>31.7599</td>
</tr>
<tr>
<td>cos</td>
<td>30.6941</td>
</tr>
<tr>
<td>asin</td>
<td>40.4035</td>
</tr>
<tr>
<td>acos</td>
<td>46.8141</td>
</tr>
<tr>
<td>tan</td>
<td>54.4701</td>
</tr>
<tr>
<td>atan2</td>
<td>60.4883</td>
</tr>
<tr>
<td>sqrt</td>
<td>22.5253</td>
</tr>
<tr>
<td>pow</td>
<td>40.903</td>
</tr>
<tr>
<td>fmod</td>
<td>41.6389</td>
</tr>
<tr>
<td>add</td>
<td>21.145</td>
</tr>
<tr>
<td>sub</td>
<td>20.8046</td>
</tr>
<tr>
<td>mult</td>
<td>20.8081</td>
</tr>
<tr>
<td>div</td>
<td>22.4972</td>
</tr>
</tbody>
</table>

*Table 3: Test machine 1 metrics*

Run time of arithmetic functions taken as an average of 100 000 000 runs with randomized parameters.
15.2 Test Machine 2 (MacOS)

machdep.cpu.brand_string: Apple M1 Pro
machdep.cpu.core_count: 8
machdep.cpu.cores_per_package: 8
machdep.cpu.logical_per_package: 8
machdep.cpu.thread_count: 8
hw.ncpu: 8
hw.activecpu: 8
hw.perflevel1.cpusperl2: 2
hw.perflevel1.logicalcpu: 2
hw.perflevel1.logicalcpu_max: 2
hw.perflevel1.physicalcpu: 2
hw.perflevel1.physicalcpu_max: 2
hw.perflevel0.cpusperl2: 3
hw.perflevel0.logicalcpu: 6
hw.perflevel0.logicalcpu_max: 6
hw.perflevel0.physicalcpu: 6
hw.perflevel0.physicalcpu_max: 6
hw.cpu64bit_capable: 1
hw.cpusubfamily: 4
hw.cpusubtype: 2
hw.cputype: 16777228
hw.logicalcpu: 8
hw.logicalcpu_max: 8
hw.physicalcpu: 8
hw.physicalcpu_max: 8

<table>
<thead>
<tr>
<th>Function</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>sin</td>
<td>19.525</td>
</tr>
<tr>
<td>cos</td>
<td>19.3505</td>
</tr>
<tr>
<td>asin</td>
<td>19.5069</td>
</tr>
<tr>
<td>acos</td>
<td>19.4785</td>
</tr>
<tr>
<td>tan</td>
<td>19.5291</td>
</tr>
<tr>
<td>atan</td>
<td>19.4959</td>
</tr>
<tr>
<td>sqrt</td>
<td>18.7334</td>
</tr>
<tr>
<td>pow</td>
<td>20.2852</td>
</tr>
<tr>
<td>fmod</td>
<td>19.6966</td>
</tr>
<tr>
<td>add</td>
<td>18.9726</td>
</tr>
<tr>
<td>sub</td>
<td>18.8856</td>
</tr>
<tr>
<td>mult</td>
<td>18.8898</td>
</tr>
<tr>
<td>div</td>
<td>18.9356</td>
</tr>
</tbody>
</table>

Run time of arithmetic functions taken as an average of 100 000 000 runs with randomized parameters.
References


