# Table of contetents

# 1 Introduction

Collision detection is a vital part of most physical simulations, video games as well as robotics[1]. Therefore, it is in our interest to explore how it is implemented. The Gilbert–Johnson–Keerthi distance algorithm, or GJK, gives a fast and efficient way to calculate the distance between two convex shapes. This makes it a very useful tool in collision detection systems, which is why we chose to make a bow and arrow target practice scene in 3D where focus is on the arrow's collision with the target using GJK.

This project was completed in 5 weeks as part of the course Models and Simulation at KTH. The main motivations for completing the project was to achieve a high grade in the course and to gain knowledge about game technologies. The project was carried out mainly through pair programming where we took turns writing different parts of the code, up until the implementation of 3D GJK where Mårten had a lot of time to spare and decided to implement it during the weekend.

# 2 Gilbert–Johnson–Keerthi

The main aim of this project was to make an implementation of the Gilbert–Johnson–Keerthi distance formula in Unity. The algorithm is used to determine whether two convex shapes are intersecting. It can also be used to determine what points on the outline of two shapes are closest to each other.

## 2.1 How it works

The algorithm can be explained in a very short manner, it simply calculates the Minkowski difference of the two shapes and checks whether the resulting shape contains the origin. If the origin is contained within the Minkowski difference, the two shapes have a point in common and therefore must be intersecting, if not - there are no points in common and they cannot intersect. A more detailed explanation of how GJK works than explained below can be found on our blog https://modsimarrowknee.blogspot.se/.

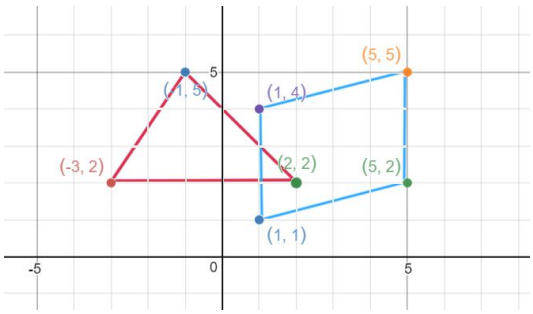
### 2.1.1 Simplex

A simplex is the shape corresponding to a triangle or tetrahedron in the n-dimension.[2] There are four simplexes in 3D GJK, the single point, the line, the triangle and the tetrahedron. Each simplex will be represented in different stages of the algorithm. When the algorithm deals with two points it creates a line between them, when there are three points they form a triangle and with four points - a tetrahedron.

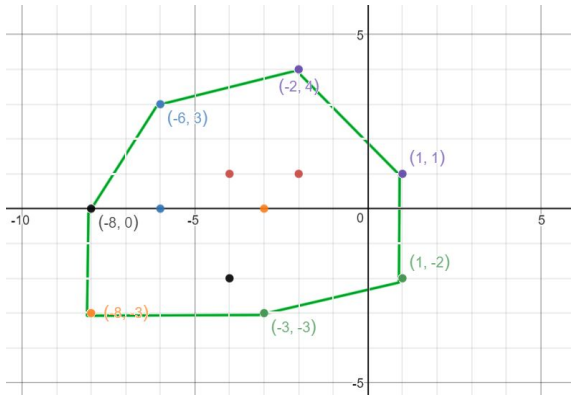
[1] MC Lin, S Gottschalk "Collision detection between geometric models: a survey"
[2] https://en.wikipedia.org/wiki/Simplex

### 2.1.2 Minkowski difference

The Minkowski sum of two objects is the the addition of all the points in the first shape with all the points in the second shape. The Minkowski difference is the same, but with subtraction instead of addition. We would usually only talk about the Minkowski sum, as the Minkowski difference is just the sum with negative terms, but for the sake of comprehension it will be referred to as a difference in this report.




Picture 2.1.2a. Two intersecting shapes.




Picture 2.1.2b. The Minkowski difference.

The resulting outline of the Minkowski difference is what is important in GJK and is therefore highlighted in green in picture 2.1.2b. This is because we only need to know whether the Minkowski shape contains the origin or not.

### 2.1.3 The algorithm

The basic outlines of our implementation of the algorithm is:

```
d = Vector3.random
p = Support(d)
[ ] ← p
d = -p
loop:
    A ← Support(d)
    if (A.dot(d) < 0) do return false //No intersection
    [ ] ← A
    if (DoSimplex()) do return true //Intersection
    if (iteration > maxIterations) do return true //Looping
```

The `Support()` function returns the point in the Minkowski difference that is furthest away in the given direction `d`.

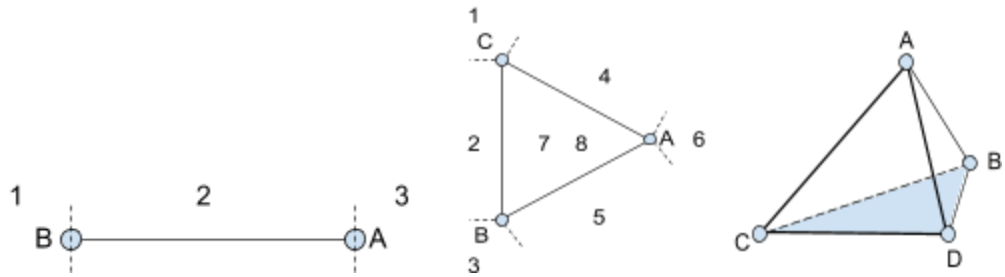
The `DoSimplex()` function handles both updating the simplex to the one that best encapsulate the origin, as well as calculating the direction to search in when the simplex does not yet contain the origin but there is a possibility that it could. It returns true when the simplex encapsulate the origin.

The last line in the loop returns true when the algorithm finds itself going in circles or taking

way too long to complete. Such cases could be when there are many points packed close together or if two corners of the shapes collide. In either case, if the algorithm haven't failed in x number of iterations the two objects are probably colliding or close enough that a collision can be triggered anyway.

#### 2.1.3.1 DoSimplex()

The most simple case (except for the single point) is when the algorithm deals with a line. The origin could be in any of the three regions (1,2 or 3) to the left in picture 2.1.3.1a. In order to know which direction to search for next, the algorithm needs to find in which region the origin is in. By some general intuition we can exclude that it can't be in regions 1 or 3. Similar intuition argument can be said to exclude several regions for each simplex. When the simplex reaches a tetrahedron shape it either returns true (if a collision has occurred) or finds a new point depending on what direction the origin is in.




Picture 2.1.3.1a Different regions origin can be located at in the simplexes

# 3 Process

## 3.1 Project idea

Taking inspiration from the course lectures and from the teacher assistants, the project idea came quite easily. GJK seemed like a project that would be both interesting to implement and give a relatively high grade. After discussing a couple of different sceneries we settled for the arrow-target implementation as described in this report. Most of the planning up until writing the project specification was done through oral conversations within the group. No thorough project plan or similar document was used to schedule the workflow except for the project specification, with the reason being that this was a short project and both team members having lots of free time.

## 3.2 Features

After developing the initial idea the next step was to start looking into the algorithm to get a grasp of what was needed to learn and what requirements there were on the colliding objects. All information about the algorithm came from finding tutorials online. The two most useful tutorials found was a video by Casey Muratori [3] and a blogpost by William Bittle[1]. However, they did differ in a few aspects in the implementation. Firstly the post by Bittle discards some of the checks done in DoSimplex() but also the order in which these checks

were done. Both of the sources only explain the 2D version of GJK but reassures that an understanding of the 2D implementation would be enough to expand it into 3D. This was true for the most part, but when there problems occurred with the 3D implementation a blogpost by Sergiu Craitoiu [2] helped with the problems. In the end the main source that was used was  the video by Muratori [3] which was both thoroughly and more flexible in its pseudocode of the three.

When working on the project specification we came up with an outline of the workflow. The project was split up into the following features:

**Basic Features:**

1. Basic model shapes. Arrow as a rectangle, target as a circle.
2. Gravity on arrow implemented by Unity libraries
3. Collision detection & handling by Unity libraries
4. Some environmental objects, such as boxes and trees.

**Advanced Features:**

5. Proper Arrow & target models
6. Visible trajectory for the arrow
7. Collision handling with GJK algorithm
8. Ability to look around and shoot

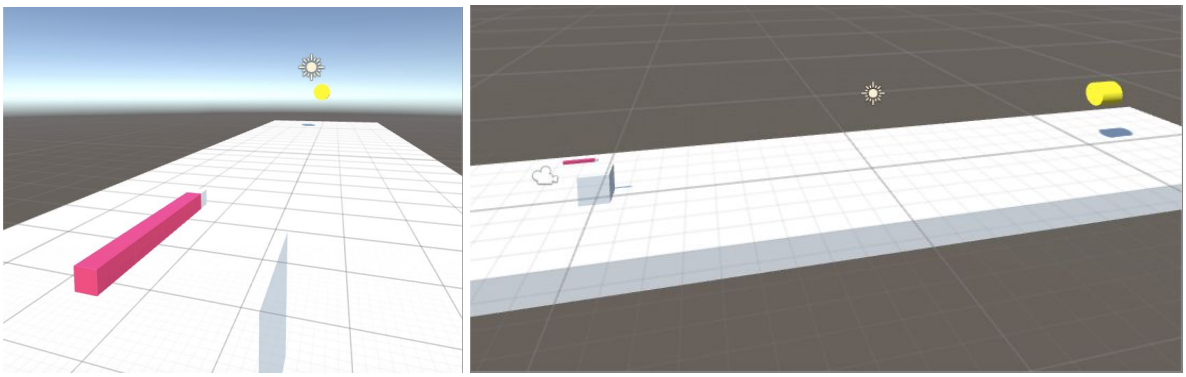
**Stretch Features:**

9. Wind affecting the arrow
10. Penetration depth for the arrow hitting the target
11. A realistic looking scene with animals, grass, trees and sky.

The Basic Features is mostly for setting up the project and to see what the end result might look like and a minimal requirement for a finished deliverable. The Advanced features are what the project aimed to deliver at the deadline and the Stretch Features where in case time allowed it after all the Basic and Advanced features had been added. We also added a feature (feature 8) to look around and shoot in any direction which was not planned at the start. In the end all Basic Features were implemented (except for 4 which we skipped at first), all the Advanced Features as well as number 11 in the Stretch Features. Penetration depth (point 10) came natural with the issues of GJK as discussed in section 5.2.1, but it's not accurate or reliable. Wind affecting the arrow was not implemented at all.

## 3.3 Implementation

The first thing that was done in Unity was to set a basic scene up (see picture 3.2a), using basic shapes to represent the arrow and target, Unity's built-in collision detection and gravity and some simple methods to allow the user to shoot and reset the arrow. The cubes seen in the picture 3.2.a were then replaced with assets from Unity Asset Store.



Picture 3.2a. The first scene in Unity.

After the first scene was set up the implementation of GJK started. This part took the longest time in the project since there was a lot to learn and understand. Naturally, the GJK algorithm was first implemented in 2D. 2D seemed to work great for this project since the projectile was moving so fast and visually it only collided when inside the target board. However in order to increase accuracy the algorithm was later expanded to handle collision detection in 3D. As there was a bit of time left, aiming and shooting in different directions with the arrow was implemented.

After this there was not time to implement any more big features and instead some more environmental objects were added to make the scene more visually pleasing. This process took only about 2 hours to finish since we used premade free models from the Unity asset store. We used the example terrain in Nature Starter Kit 2 and modified it to better fit our scene.

The assets that were used to create the final project are:

- Environmental models - Nature Starter Kit 2 by betasector[3]
- Bow and arrow - Elven Long Bow by Manuel Wieser[4]

# 4 The final result

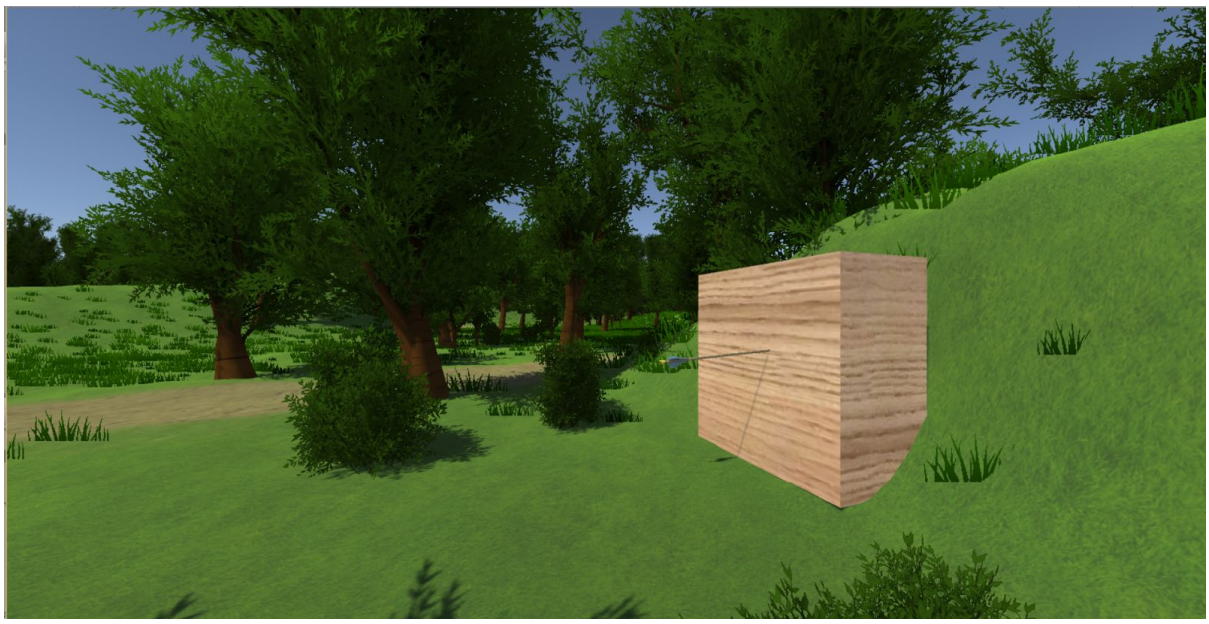
The final result of the project was very satisfactory. The primary goal, to implement GJK in 3D, as well as the feature to be able to aim with and shoot the arrow in different directions were successfully implemented. One of the the Stretch Features was also partially realised, namely to create a more realistic looking scene. The user is now situated in a forest glade, which works well with the simple target and the design of the bow.

[3] Found on https://www.assetstore.unity3d.com/en/#!/content/52977 (2017-03-13)
[4] Found on https://www.assetstore.unity3d.com/en/#!/content/17728 (2017-03-13)

This is the view the thought user has of the scene. The first-person view gives a game-like feel to the simulation, and a next step could for example be to make a moving target to emphasize this aspect. Care has been taken not to place any trees in such a way that they would be a distraction the user or on the path.

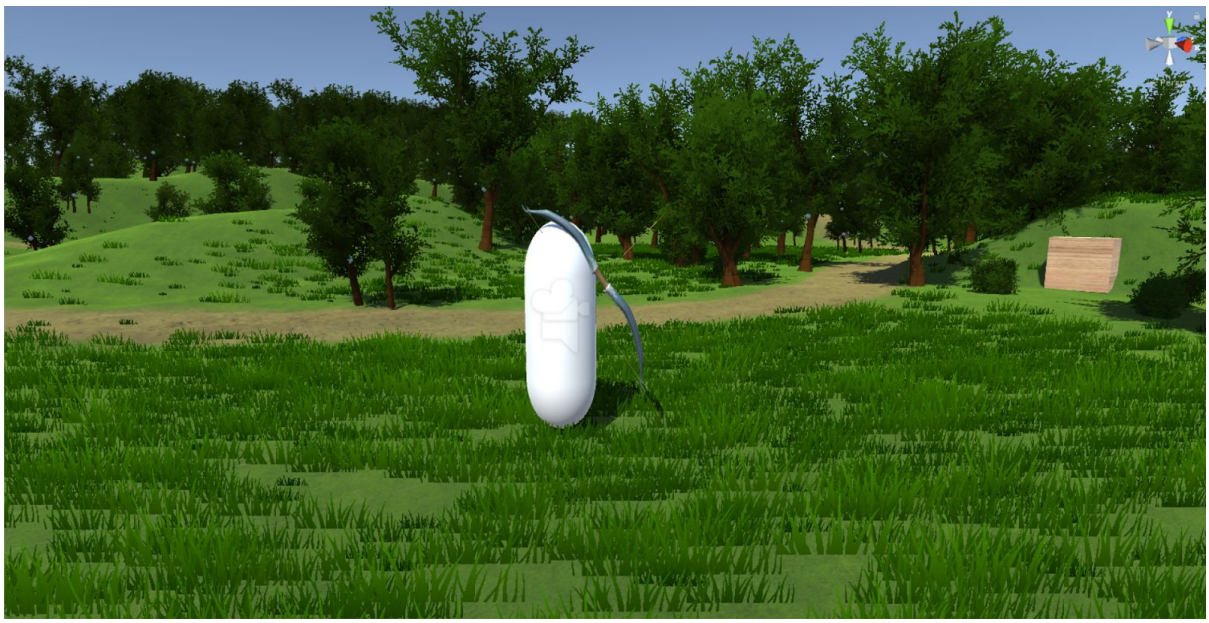
Picture 4a. A first person view of the scene.

Picture 4b is a close up on the target. There is an arrow sticking out right from the middle of it but as the arrows are quite thin it is a bit difficult to spot. The height differences in the terrain are also very apparent in this picture. This view could be used for a point counting feature where the user actually can see where the arrow hit the target.



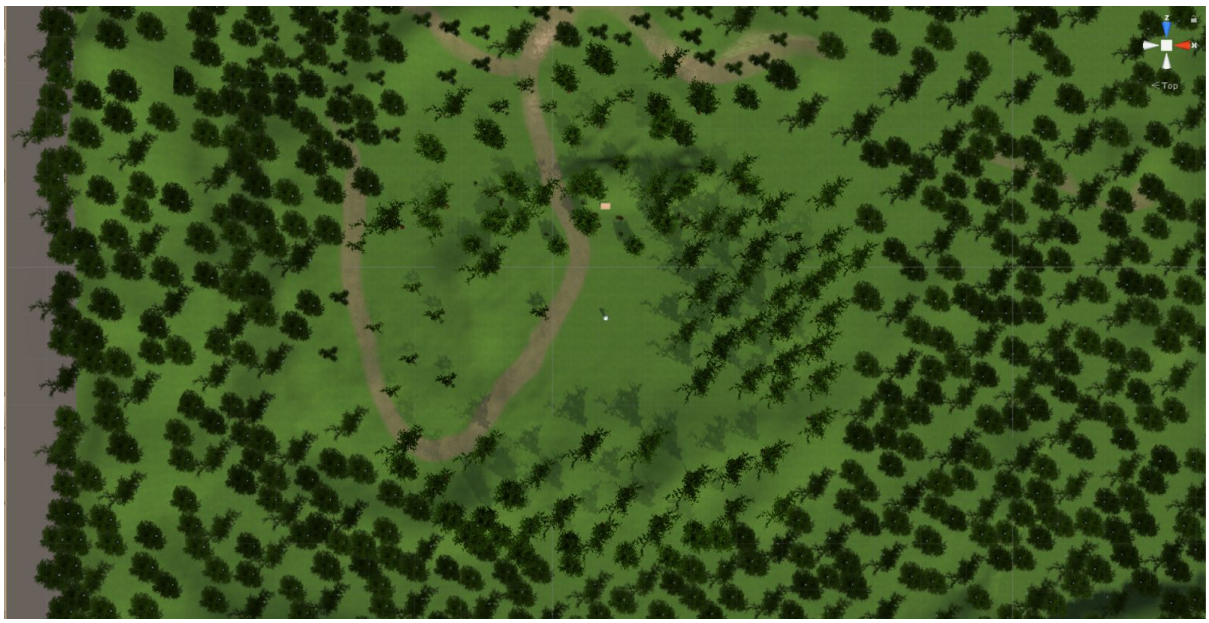
Picture 4b. The target with an arrow in it.

Since the player is not meant to be seen by the user, it is just a simple shape (see picture 4c). This is also something that could be worked on in the future. We decided to still render it since it gave a feeling of something living and it casted shadows on the ground.



Picture 4c. A third person view of the player and scene.

Picture 4d is an overview over the entire terrain and scene. We chose to make the scene a lot bigger than needed to give it more depth and not just look like the scene is floating in space. The terrain and vegetation was modified with the terrain brushes build into Unity, so there were very few manual placements of trees.



Picture 4d. An overview of the terrain and scene.

# 5 Discussion

## 5.1 Project related discussion

In this section we will discuss what we believe is the most relevant parts of the project process for this report, such as our thoughts on planning the project and making sure there was something to hand in at the end of the project.

### 5.1.1 Planning and time schedule

The reason for not having a time plan or schedule for this project was mainly due to two reasons. The first one was the size of the project. We reasoned that this was not such a big project that we needed to have a time plan in order to finish it. We had some basic deadlines for the presentations and final submission but since we worked continuously on the project since the start, we only had to consider the deadlines a few days before due. The second reason for not having a time plan was that we had lots of free time on our hands, and since we both were excited about this project, we would not a have problem finding time each week to work on it.

A time plan could however have helped us know how far we had actually come in the project and how many features that could be implemented. We noticed quite late that the 2D implementation took longer time than expected and therefore the 3D expansion was rushed in the end. This also lead to the environmental objects being added very close to the final submission. In retrospect we believe that a time plan might have helped us come further in the project development, but since we managed to implement the features that we thought would be in the final project (except for arrow trajectories) the overall (minimal) planning was a success.

### 5.1.2 Maximizing probability of a deliverable

Since we had little knowledge of how time demanding the GJK implementation would be, we realised that in order to finish the project and have at least one working deliverable in the end we had to prioritise the core features. This is represented in the Basic Features described in section 3.2. Even though the Basic Features would barely be an acceptable deliverable we would still have something to show. This way, if the GJK implementation failed, there would be material to discuss about why it failed or other smaller features that could be implemented instead.

## 5.2 Issues

In this section we discuss what problems we ran into while working on the project and how we approached them.

### 5.2.1 GJK issues

Since this project focused mainly on GJK implementation there were some issues concerning it. The first and most pervasive issue was understanding the core concepts behind GJK. The sources did a great job of explaining the algorithm and concepts in a simple manner, but it took several days before actually understanding it fully. Executing the algorithm by hand and trying different scenarios with physical objects made the understanding come much faster than just reading about it.

Another issue with using GJK is that it's not meant to be used as collision detection on fast moving projectiles. Since the algorithm can only operate on discrete timesteps, if the arrow is being shot with too much speed it could be on one side of the target in one point in time and on the other side the next point in time. This is noticeable when shooting an arrow at a very thin target as the arrow just passes through it. By making the board thick enough we could prevent this and consistently make the arrow stick in the target. GJK does not calculate penetration depth by default, which could be useful in this project. However the discrete points in time made it so that the arrow would sink into the target naturally. Although the GJK algorithm works pretty well, another collision detection algorithm such as Ray casting would have been better.

### 5.2.2 Unity issues

Learning the workings of Unity was also a big part of the project. There were some issues with model rotations and making the arrow face the right direction. While trying to implement the ability to look around and shoot, there were also some problems with getting the arrow to face the angle it was being shot in, which we managed to solve with some help from an older student. Finding assets also proved to be a bit more difficult than we had imagined. Although we did find perfect assets for the arrow and bow fairly quickly, all of the target assets either looked entirely different from what we had imagined or did not work the way we wanted them to. In the end, we went back to our cube and added a wooden texture to make it blend in with the scene.

### 5.2.3 Group work issues

Overall, working in a small group worked well for this project. Being only two people, communication and decisionmaking was easy and the work could be carried out efficiently. As we were in different classes both working on other courses parallel to this one, our schedules would sometimes collide, making it difficult to meet at certain times. This was however not a big issue as we did have a lot of time and could always find some for working on the project.

# 6 References

[1] Bittle, William. "GJK (Gilbert–Johnson–Keerthi)." Dyn4j. April 2010. Accessed March 13, 2017. http://www.dyn4j.org/2010/04/gjk-gilbert-johnson-keerthi/ .

[2] Craitoiu, Sergiu. "GJK Algorithm 3D." In2gpu. June 06, 2014. Accessed March 13, 2017. http://in2gpu.com/2014/05/18/gjk-algorithm-3d/.

[3] Muratori, Casey. “Implementing GJK - 2006”. Mollyrocket. Web. 12 May 2016. Accessed March 13, 2017. https://mollyrocket.com/849