Implementing GJK as a plugin for Unity

DH2323 Project Report June 2022

Christian K. K. Lindberg
KTH Royal Institute of Technology
Stockholm Sweden
ckkli@kth.se

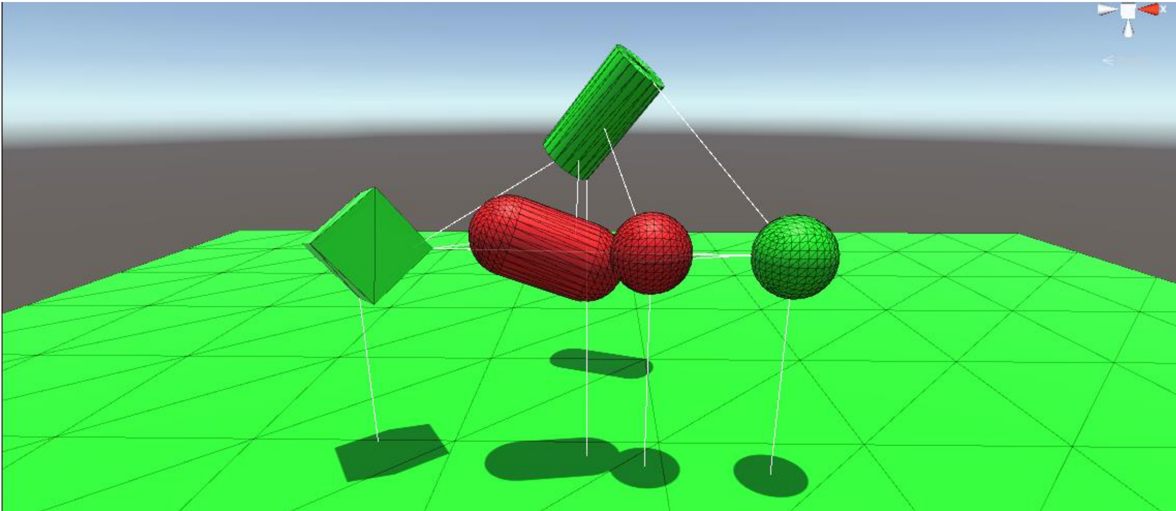

Figure 1: The implemented GJK algorithm visualized in a Unity scene. Objects intersecting turn red and lines are drawn between the closest points of non-intersecting objects.

ABSTRACT

The Gilbert, Johnson and Keerthi (GJK) algorithm is as fundamental algorithm in today's computations of collision detection. It allows for detection of intersection between convex objects as well as computation of closest points between non-intersecting convex objects. Many online tutorials and examples of the algorithm simplify their implementation by using only two dimensions and omit the computation of closest points. This report provides an example of how to implement this in 3D as a plugin for the game engine Unity. A scene in the provided project offers a visualization of the algorithm with colors and lines. A technical evaluation showed that the implementation of the algorithm, with 190 calls in a single frame on 20 objects with 24 vertices each resulted in an average execution time of 21 milliseconds. Similarly, with 20 objects with 515 vertices the average execution time was 120 milliseconds. It was concluded to not be performant enough, as is, for real time applications. The results also identified a single function as the main culprit, allowing future work to easily improve the algorithm.

CCS CONCEPTS

• Computing methodologies ~ Computer graphics ~ Animation ~ Collision detection

KEYWORDS

GJK algorithm, computer interaction, collision detection, unity

ACM Reference format:

Christian K. K. Lindberg. 2022. Implementing GJK as a plugin for Unity: DH2323 Project Report June 2022. *Stockholm, Sweden, 6 pages.*

1 Introduction

Collision detection in real-time applications rely on efficient algorithms and one of the most used, and built upon, is the Gilbert-Johnson-Keerthi (GJK) algorithm [11]. Collision detection in real-time applications such as games, can be split up into two phases [3]. The first *broad phase* is meant to be a fast screening for identifying objects that are *potentially* colliding. In the second, *narrow phase,* more accurate computations are made to determine whether these potential collisions are happening and how they should be resolved. Each phase can make use of several combining techniques and algorithms. The GJK would in general belong in the narrow phase. In the original paper, Gilbert et al. [11] show how it gives the minimum Euclidean distance between two convex polytopes, defined by the closest points in each polytope. A polytope is a geometric object with flat sides, i.e.,

faces, in *n* dimensions. With a distance above zero the polytopes can be determined as not intersecting each other. The computation time of the algorithm is said to be nearly linear in the total number of vertices of both polytopes.

There are many useful variants and extensions of the GJK algorithm described in the field [2,4,16,22], it is therefore imperative to have a good understanding of the GJK for anyone interested in simulated collisions, or physics, of objects.

The original paper on GJK [11] as well as books covering the algorithm [3,9] are heavily numerically/mathematically inclined in their explanations. This report therefore also looked to sources outside the scientific realm that explains the GJK algorithm more intuitively [5,6,17,18,23]. However, these sources simplify the algorithm by either skipping the computation of the closest points and/or use a simpler 2D environment. This project therefore aims to contribute with an example of how the GJK algorithm can be implemented in 3D with computation of the closest points of two convex polyhedra (3D polytope). Furthermore, no readily available implementation of the algorithm was found to be written in C# and exemplified in a readily available game engine project. This project therefore also aims to fill this void, by implementing the GJK algorithm as a C# plugin for the game engine Unity [19] and provide it as an open resource.

This project is guided by the Research Question: "How can the GJK algorithm determine if two convex polyhedra are intersecting, computing their closest points if they are not and be implemented as a plugin for Unity?"

2 Background

This section briefly introduces some important concepts related to the GJK. See references for more details.

2.1 Minkowski Addition

To get an understanding of how the GJK algorithm works, this report studied several sources [3,5,6,9,17,18,23]. One of the main concepts which the GJK takes advantage of is the Minkowski addition, which is the *sum* or *difference* of two sets of position vectors, A and B, in Euclidean space [3,9]. The sum $A \oplus B$ is defined as:

$$A \oplus B = \{P_A + P_B \mid P_A \in A, P_B \in B\} \quad (1)$$

Figure 2 and Figure 3 illustrate how the Minkowski sum looks like geometrically.

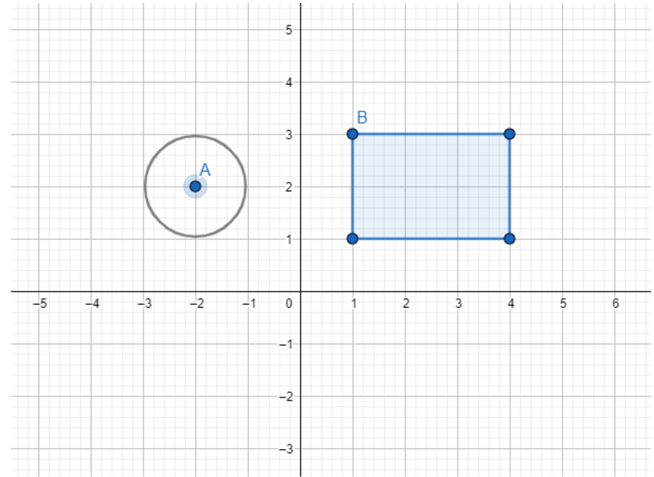

Figure 2: The individual objects A and B in two dimensions.

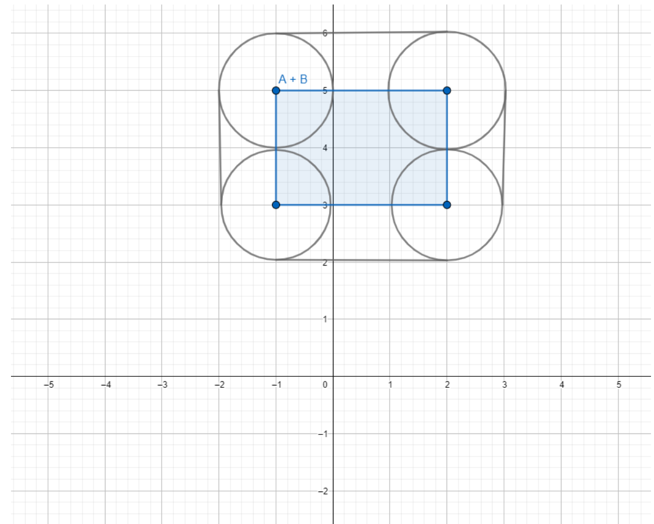

Figure 3: The geometrical Minkowski sum of A + B, one can imagine either object being swept around the other object. With the new center being center A + center B.

The Minkowski difference $A \ominus B$ is similarly defined as:

$$A \ominus B = \{P_A - P_B \mid P_A \in A, P_B \in B\} \quad (2)$$

However, geometrically we obtain the Minkowski difference by adding A to the reflection of B about the origin:

$$A \ominus B = A \oplus (-B) \quad (3)$$

Meaning the subtraction is recast to addition, both terms are therefore often referred to as the Minkowski sum. The Minkowski difference of two objects is often called the Translational Configuration Space Obstacle (TCSO) or Configuration Space Obstacle (CSO) [3,7,9]. This report will refer to is as CSO.

Certain properties of the CSO are fundamental to the GJK and for this report the relevant ones are:

1. if the two objects A and B are convex, their CSO will also have a convex hull;
2. when not intersecting, the distance from the origin to the closest point on the CSO is the distance between object A and B;
3. when the two objects intersect their CSO contains the origin.

Property 2 can be expressed as:

$$d(A,B) = \min\{\|x\| : x \in A - B\} \quad (4)$$

Note that the closest point is not necessarily unique, there can be several points at the same distance.

2.2 Support Point

Computing all points for the CSO would result in a cubic time complexity, with all points *i* in A times all points *j* in B:

$$AB_{ij} = A_i - B_j$$

However, the algorithm is only interested in the hull of the CSO. Meaning the points with maximum distance from its center, in all directions. This is essentially asking for the point with the maximum dot product with a given direction D: max (D • AB_{ij}). This is called a *support point* and is not necessarily unique. As the dot product is distributive we can define:

$$\begin{aligned} \max D \bullet AB_{ij} &= \max\left(D \bullet A_i - D \bullet B_j\right) \\ &= \max D \bullet A_i - \max\left(-D\right) \bullet B_j \end{aligned} \quad (5)$$

Finding the support point is thus a matter of only checking the dot product of all the points A_i relative to direction D and all points B_j relative to -D, giving the process a linear time complexity *i+j*.

2.3 Terminology

Apart from the already detailed concepts, there are some terms to briefly clarify before the next section. A *vertex* is a point in space, usually with additional attributes, in our context belonging to one of the objects A, B or CSO. A *point* is a position in space, not necessarily for a vertex but, a vertex has a point. A *simplex* is the simplest polytope in any given *n*-space, i.e., a 0-simplex is a point, 1-simplex a line, 2-simplex a triangle, 3-simplex a tetrahedron etc. In the context of 3D space of this project, we can thus have between 0- and 3-simplices.

3 Implementation

3.1 GJK Determine Intersection

The most influential sources followed when implementing the GJK algorithm was the explanations of [11] and [12] which builds on the former. These sources could explain in an intuitive way of how intersection could be determined with the GJK.

The algorithm implemented is as follows:

ALGORITHM 1: GJK Intersection Algorithm (Adapted from [17,23])

```
1: procedure bool GJK_intersect(Object a, Object b)
2:      vector D = random direction
3:      vector A = Support(a, b, D)
4:      simplex s = {A}
5:      D = -A
6:      repeat
7:          A = Support(a, b, D)
8:          if dot(A, D) <= 0
9:              return false
10:         s = s ∪ A
11:         if NextSimplex(S, D) == contains_origin
12:             return true
```

The code can be reviewed at [13], script GJK_Muratori.cs. Details of the algorithm can be read at [15], a brief explanation follows.

The procedure returns true if object *a* and *b* intersect. Line 2-5 initialize the procedure. The function Support, at line 3 and 7, gets the support point (see Section 2.2), essentially an implementation of Equation 5. If the latest support was not found beyond the origin, line 8, the algorithm will terminate, concluding intersection is false. Otherwise it will add the support point to the simplex *s*. The function NextSimplex, at line 11, checks if the current simplex can enclose the origin. If it cannot, it discards the points that cannot possibly contribute to enclosing the origin and sets a new search direction towards the origin, for the next support function call.

3.2 Closest Points

The algorithm in Section 3.1 is not getting the closest points between two non-intersecting objects. A presentation at the Game Developers Conference, in San Francisco 2010, by Erin Catto [5] was studied to guide this implementation. That presentation explains how to compute the closest points in 2D with a GJK algorithm. However, the method would completely change current logic of the NextSimplex written. Therefore, it was decided to save all the necessary data needed with the current NextSimplex implementation and after termination use the data to compute the closest points. This resulted in the final code that can be reviewed at [13], script GJK.cs.

The logic of finding the closest points is quite simple. We know that when the two objects are not intersecting, we want to find the point on the hull of their CSO closest to the origin. This requires the current simplex building logic in NextSimplex to build the simplex completely down to this point on the hull. The algorithm can no longer terminate as soon as it's determined that intersection is impossible. The new main termination will be when the algorithm detects it is repeatedly getting the same support point. When the GJK algorithm terminates, with intersection being false, it will have the smallest simplex needed to compute

the closest points. At this termination the simplex will always contain one to three vertices, giving us three cases to deal with:

1. The final simplex has just one vertex. That point on the CSO is closest to the origin. This vertex is represented by a single vertex in each of the two individual objects.
2. With more than one vertex we need to compute where on this simplex is the point closest to the origin, as it isn't necessarily one of the vertices. With two vertices the algorithm essentially needs to compute the "closest point on a line to arbitrary point". Computing this, we need to save weights of how "close" to each vertex of the CSO is to the closest point on this line. The weights are then applied to the corresponding vertices of the two individual objects. Figure 4 illustrates this logic.
3. With three vertices the algorithm essentially needs to calculate the "closest point on triangle to arbitrary point". Similarly, as with previous case, the weights need to be saved and applied to the corresponding vertices of the individual shapes.

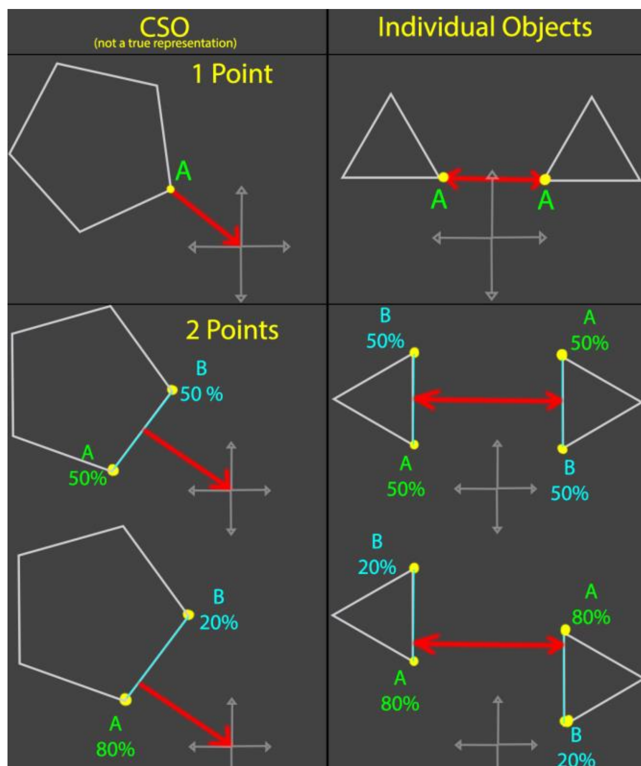

Figure 4: Computing the closest point on CSO to origin by weighted vertices. The weights are applied to the corresponding vertices on the individual objects.

3.3 Plugin and Visualizing in Unity

Coding the implementation as a plugin for Unity was straightforward when writing in C#. The documentation by Unity is simple enough to follow [20]. With this implementation as a plugin, a scene that visualizes the intersection test and closest points was created, as illustrated by Figure 1, using Unity version 2020.3.32f1. This Unity project is available at [14].

3.4 Issues

This implementation is not optimized for performance or numerical robustness, and still suffers from at least one big issue. The NextSimplex logic sometimes end up in an endless loop with what seemed to be an edge case between two triangle faces. There was not enough time to investigate this further and is therefore presented as is.

4 Evaluation, Results and Discussion

4.1 Evaluation in Unity and Results

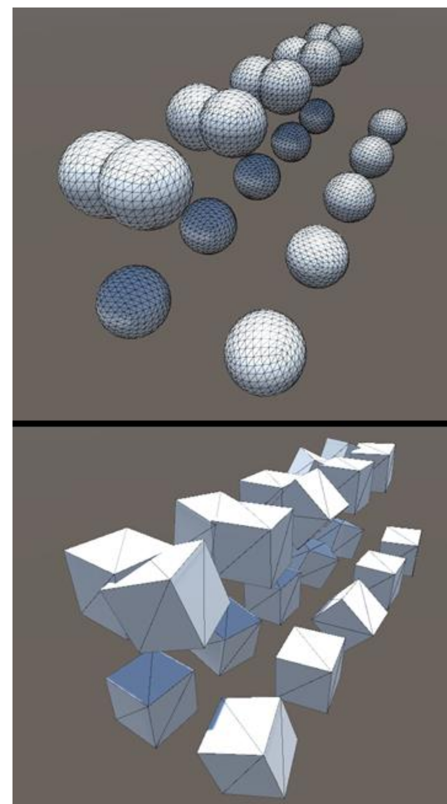

Figure 5: The two tests with 20 objects. Top: 20 spheres with half intersecting each other. Bottom: 20 cubes with half intersecting.

A technical evaluation measured the average execution time, of a frame, with Unity's deep profiler tool [21] and the number of loops within the GJK algorithm per call was coded by hand to output at each termination. Illustrated in Figure 5, the evaluation ran two separate tests, comparing a set of 20 spheres and 20 cubes. A sphere mesh containing 515 vertices and a cube 24 vertices. The objects were given pseudorandom orientations by hand, with half of the objects intersecting each other, in static

positions. Running each test for ten seconds, an average of measures is shown in the two bottom rows in Table 1. The computer running the test had an Intel Core i5-6600K CPU at 3.50GHz.

Table 1: Metrics and measurement results of the evaluation

	Sphere	*Cube*
No. of Objects	*20*	*20*
No. of Mesh Vertices per Obj.	*515*	*24*
No. of GJK Calls per Frame	*190*	*190*
Avg. No. of GJK-loops per Call	*4.15*	*4.05*
Avg. Total Execution Time per Frame (ms)	*120*	*21*

All the objects were being checked against each other with the GJK algorithm at every frame (time step), resulting in 190 calls per frame. The average number of loops within the algorithm before termination for both objects were around 4.1. We can see that the average execution time for a frame were much larger for the spheres, being 120 milliseconds (ms) compared to 21ms for the cubes. This naturally indicates that the larger number of vertices in the spheres contributed to the longer execution time. Unity's deep profiling could confirm this showing the execution times per function within the implementation. The function FindFurthestPoint, which is an implementation of Equation 5, contributed to more than 90% of the execution time for the spheres and more than 60% for the cubes.

4.2 Future Perceptual Study

In games the player immersion should not be interrupted by illogical physics. However, things that the player does not perceive one can simplify and cut corners, saving computation resources. It is hard to imagine how this implementation per se would be evaluated, however, a future perceptual study where the project could be used is for a sort of "method of adjustment" [12] study. Measuring the threshold of participants ability to determine how close two virtual convex objects can be positioned before they appear to be colliding. The participants would be asked to position two different virtual objects as close as possible without them touching. The implementation of this project would verify the distance between the objects or if they are colliding. The results could perhaps tell us that some objects are perceived as colliding on larger distances than other shapes. Allowing the system to ignore doing collision test on these objects as the users will not notice them. Alternatively, a simpler but more performant type of collider could be used in situations where the player will not notice the preciseness of a "mesh collider".

4.3 Future Improvements

Section 5 concludes how the issue brought up in Section 3.4 could be addressed and how the algorithm could be sped up. This project could be extended and improved in many ways. Right now, the project just uses a "mesh collider". The "collider" class could be extended with primitive objects that has their own, faster, implementation of FindFurthestPoint, also known as support point. Furthermore, a broad phase technique could be implemented such as the "sort and sweep" [1], also described as "sweep and prune"[8]. A technique that makes use of bounding volumes for every object. The bounding volumes are simple geometrical shapes that can be very quickly checked against each other. The volumes could be axis-aligned bounding boxes or bounding spheres. Spheres have the advantage of being orientation independent. These volumes are stored in a data structure that is iterated every frame, checking the volumes against each other, and updating positions or orientation if needed. If a volume is found intersecting another, the GJK would run on the objects for a definitive collision detection.

5 Conclusion

The aim of this project was to provide an implementation of the GJK in 3D with computation of closest points, written in C#, as a plugin for Unity. The final implementation successfully runs a GJK algorithm with computation of closest points. A Unity project provides a scene that visualize intersection and closest points. The plugin code and visualization scene are available on GitHub [13,14] and written to be understandable by novices with plenty of comments describing the code.

Though it was not the aim of the project to be robust/performant, a technical evaluation showed that the code may not ready-to-use for real-time applications. For example, a real-time game would generally want to run at 60 frames per second, equivalent to an execution time of 16ms per frame. The implementation is well above that, with 20 objects executing at 20-120ms. However, the evaluation also showed that there is one function that can be optimized for better performance, the function FindFurthestPoint. It iterates all the vertices of a mesh, running a dot product computation for each vertex to find the support point. A future alternative method could be the *hill climbing* approach described in [3,4,9]. Furthermore, 20 objects with mesh colliders does not have to be tested every frame with some improvements mentioned in Section 4.3.

The implementation also suffered from an edge case with endless looping. This could be the result of lack of numerical robustness in the NextSimplex function, as this code is adapted from Muratori's [17] tutorial which has been said to lack numerical robustness[10]. Future work could look at [16] which describe a more numerically robust approach for the GJK algorithm.

REFERENCES

[1] David Baraff. 1997. An introduction to physically based modeling: rigid body simulation II—nonpenetration constraints. *SIGGRAPH course notes* (1997), D31–D68.

[2] Gino Van den Bergen. 1999. A Fast and Robust GJK Implementation for Collision Detection of Convex Objects. *Journal of Graphics Tools* 4, 2

(January 1999), 7–25. DOI:https://doi.org/10.1080/10867651.1999.10487502

[3] Gino van den Bergen. 2003. *Collision Detection in Interactive 3D Environments*. CRC Press.

[4] Stephen Cameron. 1997. Enhancing GJK: computing minimum and penetration distances between convex polyhedra. In *Proceedings of International Conference on Robotics and Automation*, 3112–3117 vol.4. DOI:https://doi.org/10.1109/ROBOT.1997.606761

[5] Erin Catto. 2010. Computing Distance using GJK — GDC 2010. In *Game Developers Conference 2010*. Retrieved June 3, 2022 from https://box2d.org/publications/

[6] Ming-Lun Chou. 2013. Game Physics: Collision Detection – GJK | Ming-Lun “Allen” Chou | 周明倫. Retrieved May 1, 2022 from https://allenchou.net/2013/12/game-physics-collision-detection-gjk/

[7] Ming-Lun Chou. 2013. Game Physics: Collision Detection – CSO & Support Function | Ming-Lun “Allen” Chou | 周明倫. Retrieved June 9, 2022 from https://allenchou.net/2013/12/game-physics-collision-detection-csos-support-functions/

[8] Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha, and Madhav Ponamgi. 1995. I-COLLIDE: an interactive and exact collision detection system for large-scale environments. In *Proceedings of the 1995 symposium on Interactive 3D graphics* (I3D ’95), Association for Computing Machinery, New York, NY, USA, 189-ff. DOI:https://doi.org/10.1145/199404.199437

[9] Christer Ericson. 2004. *Real-Time Collision Detection*. CRC Press.

[10] Randy Gaul. 2017. Gilbert-Johnson-Keerthi (GJK) collision detection algorithm in 200 lines of clean plain C. *r/gamedev*. Retrieved June 11, 2022 from www.reddit.com/r/gamedev/comments/6wivay/gilbertjohnsonkeerthi_gjk_collision_detection/dm9g3mk/

[11] E.G. Gilbert, D.W. Johnson, and S.S. Keerthi. 1988. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal on Robotics and Automation* 4, 2 (April 1988), 193–203. DOI:https://doi.org/10.1109/56.2083

[12] E. Bruce Goldstein and Laura Cacciamani. 2021. *Sensation and Perception*. Cengage Learning.

[13] Christian Lindberg. 2022. *frilel/CLCollision*. Retrieved June 11, 2022 from https://github.com/frilel/CLCollision

[14] Christian Lindberg. 2022. *frilel/GJK_tester*. Retrieved June 11, 2022 from https://github.com/frilel/GJK_tester

[15] Christian Lindberg. 2022. Researching GJK. *Christian’s Project Blog*. Retrieved June 11, 2022 from https://dh2323christianlindberg.wordpress.com/2022/05/11/researching-gjk/

[16] Mattia Montanari, Nik Petrinic, and Ettore Barbieri. 2017. Improving the GJK Algorithm for Faster and More Reliable Distance Queries Between Convex Objects. *ACM Trans. Graph.* 36, 3 (June 2017), 30:1-30:17. DOI:https://doi.org/10.1145/3083724

[17] Casey Muratori. 2006. Implementing GJK (2006). Retrieved May 1, 2022 from https://caseymuratori.com/blog_0003

[18] Reducible. 2021. *A Strange But Elegant Approach to a Surprisingly Hard Problem (GJK Algorithm)*. Retrieved June 7, 2022 from https://www.youtube.com/watch?v=ajv46BSqcK4

[19] Unity Technologies. 2022. Unity Real-Time Development Platform | 3D, 2D VR & AR Engine. Retrieved June 7, 2022 from https://unity.com/

[20] Unity Technologies. 2022. Unity - Manual: Managed plug-ins. *Unity Documentation*. Retrieved June 11, 2022 from https://docs.unity3d.com/Manual/UsingDLL.html

[21] Unity Technologies. 2022. Unity - Manual: The Profiler window. *Unity Documentation*. Retrieved June 11, 2022 from https://docs.unity3d.com/Manual/ProfilerWindow.html#deep-profiling

[22] Gino Van Den Bergen. 2001. Proximity queries and penetration depth computation on 3d game objects. In *Game developers conference*.

[23] Iain Winter. 2020. Winter’s Blog. *GJK: Collision detection algorithm in 2D/3D*. Retrieved June 3, 2022 from https://blog.winter.dev/2020/gjk-algorithm/