Advanced Tech Report on Mesh Destruction

Alessandro Bufalino
19017120

University of the West of England

May 22, 2023

This technical report presents a potential method for implementing a destructible wall system in the Unity game engine, specifically designed for shooting games. The system utilizes a combination of the marching squares algorithm to construct the wall mesh, and Voronoi and convex hull algorithms to generate debris resulting from wall destruction.

1 Introduction

The gaming industry has seen remarkable advancements recently, resulting in more interactive and immersive experiences, with a key aspect being mesh destructibility. Unity (Unity Technologies, 2023), a popular game engine, is known for its versatility, ease of use, and accessible mesh formation components, making it ideal for this project. The project is inspired by the destruction method in Rainbow Six Siege (Ubisoft, 2015).

In this technical report, we introduce a method for implementing a destructible wall system in Unity, tailored for shooting games. The system combines algorithms such as marching squares (Foley et al., 1996) for wall mesh construction, Voronoi (Aurenhammer, 1991) and convex hull (Barber, Dobkin, and Huhdanpaa, 1996) algorithms for debris generation from the damaged wall and ways the designers can tailor this implementation to its desired level. The report aims to provide a comprehensive understanding of the method, its implementation, underlying algorithms, and relevant optimizations and challenges.

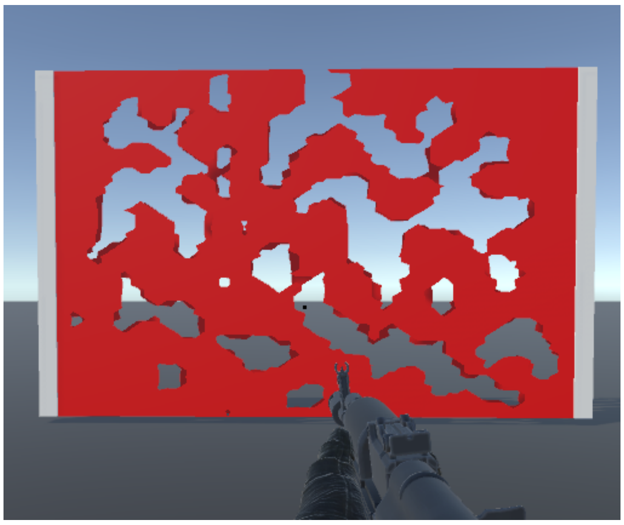


Figure 1: *Showing the wall after it has been shot by the player multiple times.*

2 Related Work

Destructible objects are common features in many modern games, providing users with intractability in levels that were previously impossible due to expensive operations. This section reviews previous studies related to mesh destruction and games that have released variants of such technology.

The most common method for destructible objects in games involves pre-fracturing and model switching of an object. This method uses modelling software to create an undamaged version of the object and a fractured version using built-in plugins found in various modelling software, such as the cell fracture option in Blender (Blender Foundation, 2002). When the object in the game is interacted with, the undamaged object is swapped with the fractured variant. This approach saves the computational cost of calculating a new mesh at runtime and is useful when multiple objects need to be destroyed at once, such as in dungeon crawler games like Minecraft Dungeons (Mojang Studios and Double Eleven, 2020).

On the other hand, recent studies have explored real-time mesh destruction techniques for use in games. For instance, Zhang et al. (2022) proposed a method for real-time mesh cutting and fracturing based on Voronoi diagrams. Their approach demonstrated the potential for creating dynamic destructible environments. Another study by Parker and O'Brien (2009) focused on the corotational tetrahedral finite element (CTFE) method for real-time fracture and deformation. CTFE is a computationally efficient numerical technique used for simulating deformable objects by discretizing them into tetrahedral elements and separating global rotations from local deformations to simplify calculations.

Additionally, a previous work that substantially impacted the functionality and original idea for this implementation was Rainbow Six Siege (Ubisoft, 2015), with its destructible walls that can be destroyed in real-time and in an online environment. Rainbow Six is a tactical FPS game that allows players to create small holes in designated walls to gain a tactical advantage over the enemy player as shown in figure 2. This ability to shape the map to suit team strategies creates an endless loop of possible tactics (L'Heureux, 2016).

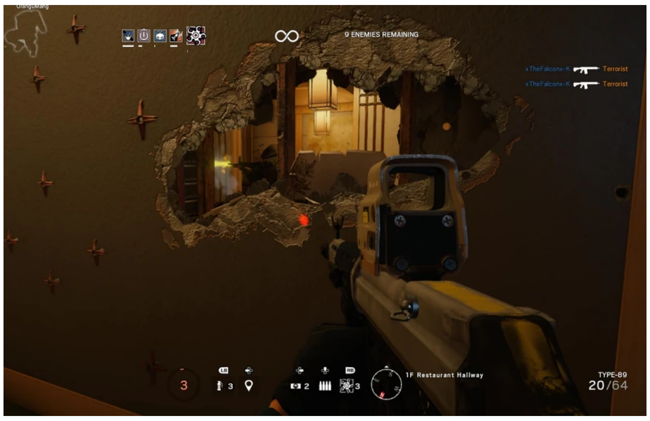
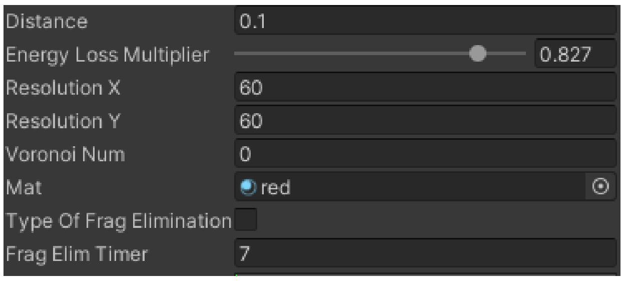


Figure 2: *Bullet penetration system in Rainbow Six Siege. Source: Rainbow Six Wiki, 2023*

3 Method

3.1 Designer section

An essential implementation aspect was enabling the level designer to adjust the destructible wall's settings based on diverse needs. The wall characteristics include:



Distance	0.1
Energy Loss Multiplier	0.827
Resolution X	60
Resolution Y	60
Voronoi Num	0
Mat	● red
Type Of Frag Elimination	☐
Frag Elim Timer	7

Figure 3: *Showing the different parameters the designer can change to make the wall abide the current context.*

1. **Wall position and size:** The designer can drag the guiding corners to set the wall to the desired size and shape. The corners can also be set in irregular positions to form more interesting shapes, as shown in Figure 4.
2. **Distance:** The distance between the two faces of the walls.
3. **Resolution of Destruction:** The number of overall vertices that a wall will have. The more resolution a wall has, the more performance heavy it will be, but will also have a more realistic look.
4. **Voronoi Division:** Different materials when broken have different behaviours. The Voronoi number dictates how many pieces the broken-off piece will break into.
5. **Energy Loss Multiplier:** Determines the loss of energy of the bullet after it has gone through the wall. denoted as a percentage, the higher it is the more fragile the wall will be.

6. **Impact strength:** Using the animator graph provided by unity, the designer is able to graph the damage to the impact done by the bullet to each vertex with respect to the distance from the point of impact, as shown in Figure 5. The graph is unique to each weapon, and is stored in the player script and passed with the function call for the damage to the wall.

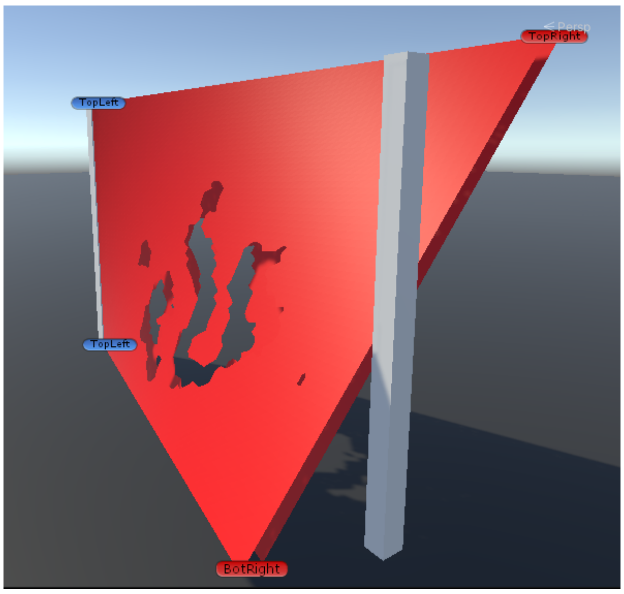


Figure 4: *Showing a bending wall due to the vertices placements following the 4 guiding corners.*

After setting variables, the CreateMSQPlane class generates 4 plane objects, divided into 2 groups for each wall side. Each group has an inner and outer side-facing plane to handle collisions from all directions.

3.2 Mesh Destruction

Once one of the wall faces are shot by the player, a sequence of algorithms will run to draw the new face of the wall.

1. Each of the vertices has a weight, when the shoot lands on the created plane the vertices closer to the point of impact will have their weight reduced by an amount specified by the impact strength graph, similar to the inverse square law.

$$F = \frac{k}{r^2} \tag{1}$$

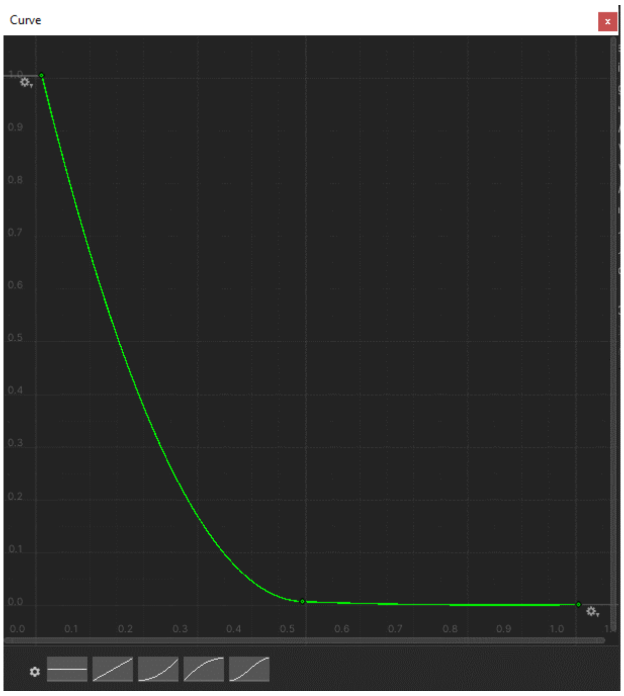


Figure 5: *Graph showing the Damage (y-axis) to the vertices based on the distance (x-axis) of the vertex to the impact point.*

2. The updated vertices and weights are passed through marching squares algorithm to create the hole contours from the “bullet”. If a vertex weight falls below a threshold, it forms a hole. The algorithm divides the lattice into squares and generates triangles based on each square vertex’s status (active or not). A lerping mechanic is included for a more realistic appearance, making triangles converge towards heavier vertices. The wall’s overall shape is formed from the combined squares’ outcomes.
3. The flood fill (O’Callaghan and Mark, 1984) algorithm is applied next to identify regions not connected to anchor points (wall edge vertices). The algorithm uses a recursive function that iterates until the current point fails to meet the minimum weight for an “active” vertex, completing the recursion. The returned list holds a specific region. If it doesn’t include anchor vertices, the region is floating, and a new set of algorithms is executed.

   3.1 To achieve the desired fragmentation count specified by the designer, the Voronoi algorithm is used to divide a plane into regions based on the proximity of a set of points. The algorithm randomly selects a given number of vertices and marks them as seed points. It then iterates through all the points, determining which seed point is the closest. Once all the points have been iterated on, the function returns a list containing a list of vertices that represent the subdivided regions.

   3.2 Each subdivided region is processed by a fi-

nal function that creates the fragmented wall mesh. This function utilizes the Iterative Convex Hull algorithm, which calculates the convex hull of a set of points in 3D space. The convex hull is the smallest shape that can contain all the points, in this case, it will contain all the floating vertices.
The algorithm works by:

3.2.1 A non-visited list of vertices is initialized with all vertices.

3.2.2 Four random vertices from the non-visited list create the first polygon iteration.

3.2.3 The algorithm iterates through the remaining non-visited vertices. If a vertex is inside the current polygon, it's discarded. If outside, a new polygon iteration is drawn.

3.2.4 Faces pointing towards the vertex are deleted and redrawn, including the new vertex in the polygon.

3.2.5 The algorithm runs until the non-visited list is empty or a performance cap is reached. The final triangulation is returned as an ordered list of Unity-type Vector3.

3.3 The list which contains the vertices is then sent to a function which will create a mesh and finally spawn the object.

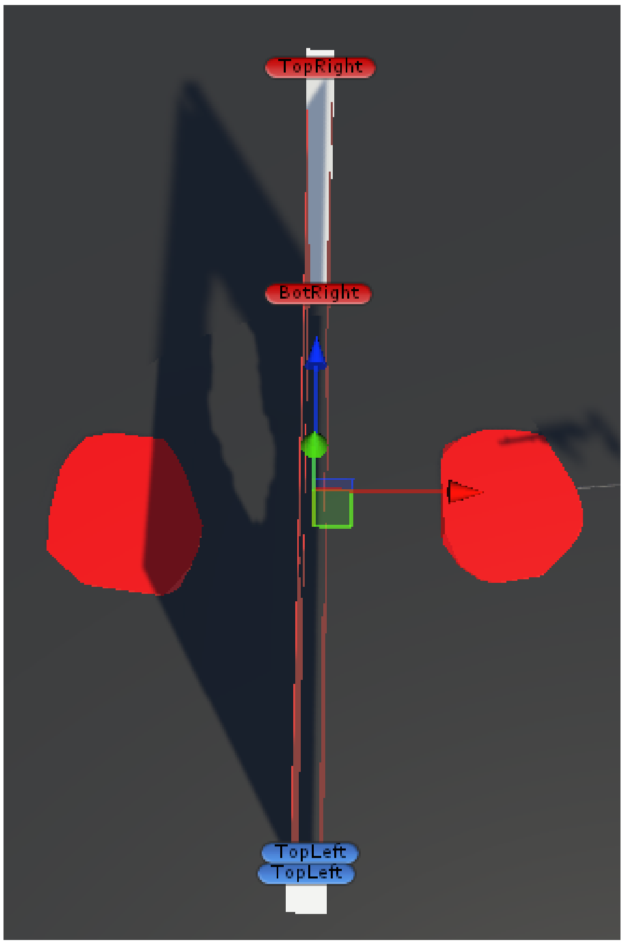


Figure 6: *Image showing the fragment of a wall falling without using Voronoi subdivision.*

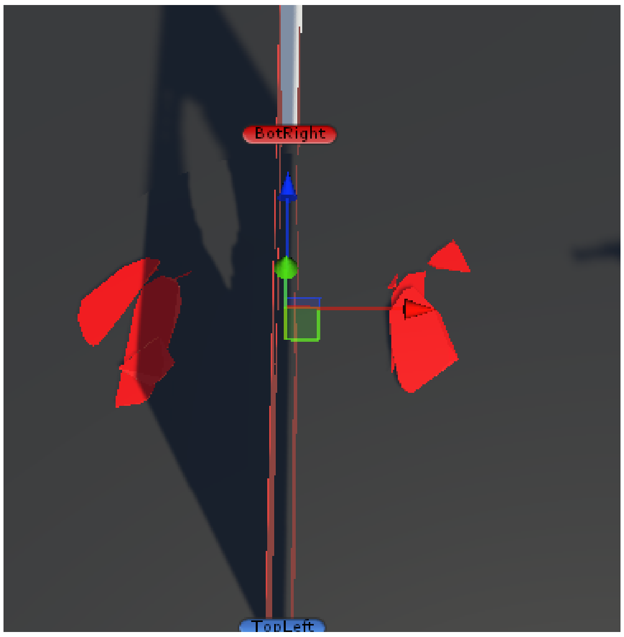


Figure 7: *Image showing the many fragments created from the wall because of the use of Voronoi subdivision.*

4. Lastly, to simulate a bullet passing through the wall, a new raycast is created (see Figure 8) with the same direction and position. This raycast should hit the opposite wall face, and the same algorithms will be applied. The independence of the wall sides allows for more realistic, non-linear shooting patterns. The new raycast has reduced power, simulating energy loss, which can be set by the designer.

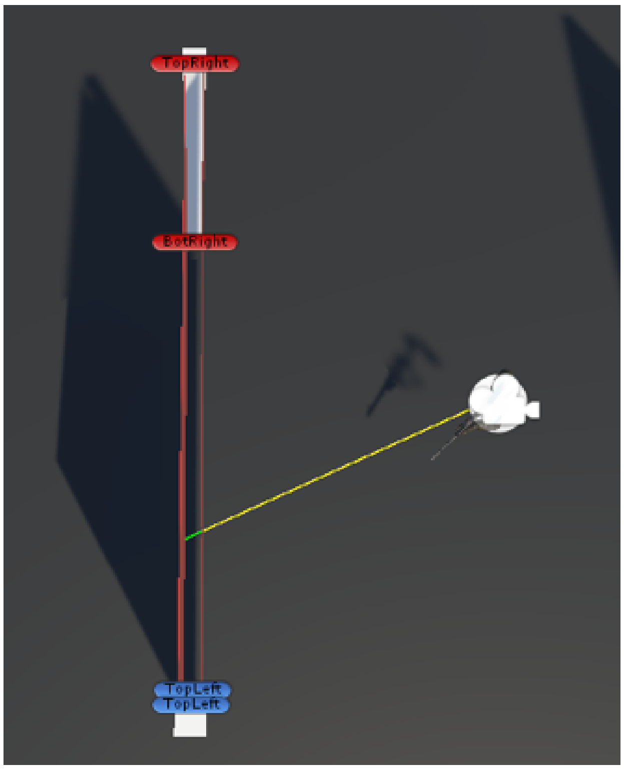


Figure 8: *Yellow raycast showing the projectile from the player. Green raycast is the relayed raycast from the wall with lower power.*

To destroy the fragmented parts of the wall, the Designer can decide whether the fragment shrinks in size, and then after a set size, disappears or the item just fall through the floor after the first collision with another object. The designer can also decide how long of a time the fragments should wait until they start to disappear.

4 Evaluation

4.1 Performance and Efficiency

The destructible wall exhibits adequate processing times with the occasional stutter depending on the size of the vertices set by the designer. This is because the entire mesh is reconstructed every time it gets shoots, which could prove inefficient when some parts are left untouched.

On the other hand, to mitigate the total processing load to successfully run all the algorithms that were listed above in one frame, different methods were used to lighten up the number of processes the script would take to calculate the new mesh:

- As previously mentioned, each face of the wall is made up of an outer and inner portion. When one of the planes is changed due to the marching square algorithm as a result of the player shooting the wall, instead of re-running all the algorithms for the other plane, the unaffected planes will copy their mesh.
- Chunk system: To eliminate needless checking of different points in the mesh when using the marching square algorithm, a chunk system has been added to reduce the number of checked vertices. This works by first checking in which quadrant the shoot has landed and added all its vertices to a list, and then checking all the immediately adjacent quadrants. This means that if the player were to shoot the bottom of the plane, the top row of vertices would not be checked, cutting the time spent on this algorithm in half on average (see Figure 9).

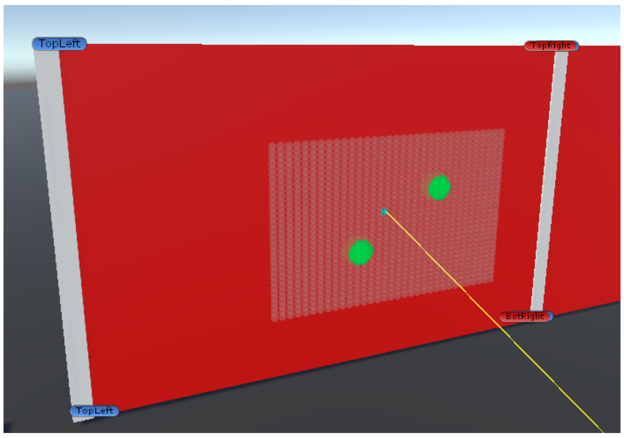
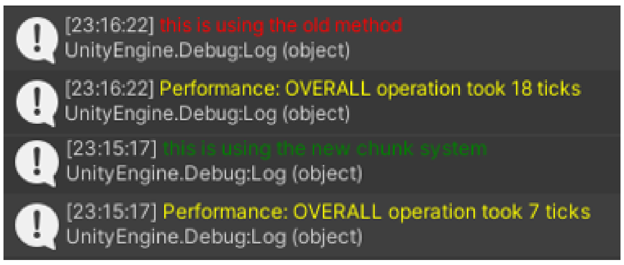


Figure 9: *Green points showing the Top Right and Bottom Left vertex of that quadrant. The Cyan points showing the added vertices to the list for performance gains.*



```
[23:16:22] this is using the old method
UnityEngine.Debug:Log (object)
[23:16:22] Performance: OVERALL operation took 18 ticks
UnityEngine.Debug:Log (object)
[23:15:17] this is using the new chunk system
UnityEngine.Debug:Log (object)
[23:15:17] Performance: OVERALL operation took 7 ticks
UnityEngine.Debug:Log (object)
```

Figure 10: *Showing the difference of time taken between the old system and the new system.*

4.2 Flexibility and scalability

The system demonstrated flexibility in adapting to different game scenarios and environments, as it can be easily integrated into any kind of map using its flexible corner (see figure 4) to create any shape and be oriented in any rotation wanted. Furthermore, the necessary processing can also be scaled thanks to the resolution variable to accommodate lower-power machines.

4.3 Future Work

Future work could include using Unity's compute shader to offload processing to the GPU for better performance and employing the alpha shape Delaunay triangulation algorithm (Edelsbrunner and Mücke, 1994) for more accurate debris contours, resulting in a tighter and more complex boundary around shot-off shapes (see figure 11).

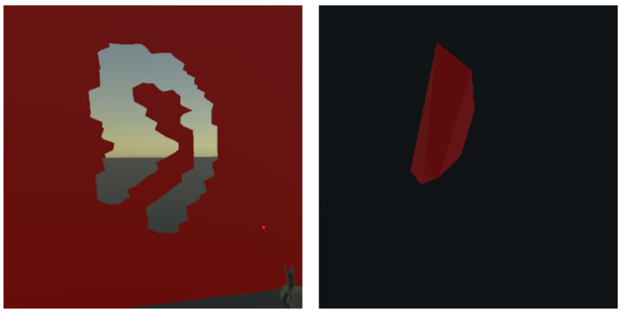


Figure 11: *Showing the difference from expected versus actual outcome.*

5 Conclusion

In conclusion, this report introduces a method for a destructible wall system in Unity, inspired by Rainbow Six Siege. Combining multiple algorithms, it creates realistic, dynamic destructible environments, enhancing player immersion. Future work can focus on optimization and additional features.

Bibliography

Aurenhammer, Franz (1991). "Voronoi diagrams: A survey of a fundamental geometric data structure". In: *ACM Computing Surveys (CSUR)* 23.3, pp. 345–405.
Barber, C. Bradford, David P. Dobkin, and Hannu Huhdanpaa (1996). "The Quickhull Algorithm for Convex Hulls". In: *ACM Transactions on Mathematical Software (TOMS)* 22.4, pp. 469–483.
Blender Foundation (2002). *Blender*. URL: `https://www.blender.org/`.
Edelsbrunner, Herbert and Ernst P Mücke (1994). "Three-dimensional alpha shapes". In: *ACM Transactions on Graphics (TOG)* 13.1, pp. 43–72.
Foley, James D. et al. (1996). *Computer Graphics: Principles and Practice*. 2nd ed. Addison-Wesley Professional.
L'Heureux, Julien (2016). *The Art of Destruction in 'rainbow six: Siege'*. URL: `https://gdcvault.com/play/1023003/The-Art-of-Destruction-in`.
Mojang Studios and Double Eleven (2020). *Minecraft Dungeons*. [Video Game]. URL: `https://www.minecraft.net/en-us/about-dungeons`.
O'Callaghan, L. J. and D. M. Mark (1984). "The Seed Fill Algorithm: A Seed Fill Algorithm for Filling Regions on Raster Devices". In: *IEEE Computer Graphics and Applications* 4.8, pp. 56–65.
Parker, Eric and James O'Brien (Aug. 2009). "Real-time deformation and fracture in a game environment". In: pp. 165–175. URL: `https://www.minecraft.net/en-us/about-dungeons`.
Rainbow Six Wiki (2023). *Bullet Penetration*. URL: `https://rainbowsix.fandom.com/wiki/Bullet_Penetration`.
Thomas, Rachel and Wenshu Zhang (2022). "Real-time fracturing in video games". In: *Multimedia Tools and Applications* 82.3, 4709–4734. DOI: `10.1007/s11042-022-13049-x`.
Ubisoft (2015). *Tom Clancy's Rainbow Six Siege*. Video game. URL: `https://www.ubisoft.com/en-us/game/rainbow-six/siege`.
Unity Technologies (2023). *Unity Game Engine*. `https://unity.com/`.