Advanced Technologies: Old-School FPS Game

William Whitehouse (19019239)
University of the West of England

February 27, 2023

1 Introduction

At first glance, First Person Shooter (FPS) games look fairly trivial, but behind the scenes, some complex algorithms and systems ensure the game runs smoothly. This project attempts to recreate a small playable slice of *Doom*, with a focus on exploring and understanding the modern technologies that underpin these systems.

2 Related Work

2.1 Data-Oriented Design

Data-Oriented Design is a programming paradigm that focuses on designing software around the data rather than what logic is performed on it (Fabian, 2018). The goal of this paradigm is to maximise software efficiency by improving cache locality and reducing memory fragmentation and cache misses. Real-time applications, such as games, require maximum efficiency as they involve complex and computationally expensive algorithms; which is why data-oriented design is so important.

Data-oriented design structures code around data rather than objects, using "components of data" instead of traditional Object-Oriented Programming (OOP) objects. Components alone don't hold much meaning, but when combined, they can create complex objects that are easily expandable. For instance, in traditional OOP programming, a player class might include every possible option, even those needed only in rare scenarios (Fabian, 2018). With data-oriented design, a player class doesn't exist, but its functionality is built from a range of components. This approach offers greater flexibility, as components can be added or removed as needed, allowing for more efficient use of resources.

Data organisation can have a significant impact on the performance of a program. By arranging data to be cache-friendly, a computer can make predictable memory fetches, resulting in faster processing times. This is achieved by storing relevant data in adjacent memory chunks, eliminating the need to fetch data from different memory locations (Nystrom, 2014).

2.2 Physics

Physics is a significant system involved in various aspects of gameplay development. Collision detection and resolution are among the primary areas, using a collision volume to identify intersections and calculating the Minimum Translation Vector (MTV), which indicates the minimum object movement required to no longer intersect the original volume (Ericson, 2004).

The Axis-Aligned Bounding Box (AABB) is arguably the simplest volume, defined by its minimum and maximum corner points (Ericson, 2004). The 3D intersection test for AABB is shown in Equation 1, with A_1 and A_2 representing the min and max corner points for object A and B_1 and B_2 representing the min and max points for object B. To optimise collision detection further, one can utilise the GPU (Cai et al., 2014) or implement SIMD instructions (Ericson, 2004)

$$\begin{aligned} \textit{intersection} = &(\min(A_{1x}, A_{2x}) \leq \max(B_{1x}, B_{2x})) \,\&\& \\ &(\min(A_{1y}, A_{2y}) \leq \max(B_{1y}, B_{2y})) \,\&\& \\ &(\min(A_{1z}, A_{2z}) \leq \max(B_{1z}, B_{2z})) \end{aligned} \quad (1)$$

The AABB volume's drawback is that it is always aligned to the axis, making accurate tests impossible for rotated objects. To overcome this issue, an Oriented-Bounding-Box (OBB) can be used. An OBB is characterised by a centre point ($\vec{c}$), a local rotation for each axis ($u_{\vec{x}\vec{y}\vec{z}}$), and halfwidth extents for each axis ($\vec{e}$). The simplest OBB intersection test employs the Separating Axis Theorem (SAT), which projects objects onto an axis L and determines whether they are separated by comparing the sum of their projected radii to the distance between the projection of their centre points (Ericson, 2004). Simply put, the SAT test places a separating hyper-plane between objects. If this can be done without intersection, the objects do not intersect, as illustrated in Figure 1.

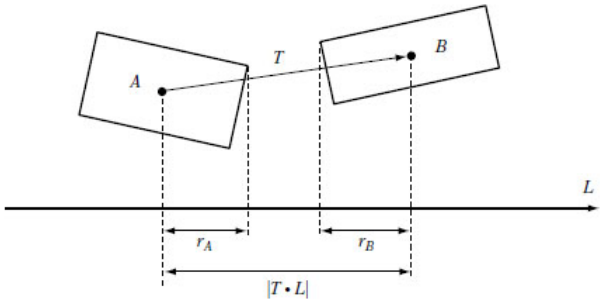


Figure 1: *Two OBBs projected onto axis L, they are only separated if the sum of their projected radii is less than the distance between their projected centres (Ericson, 2004).*

OBB intersection tests require 15 axes to be tested (Huynh, 2009), which include:

$$\underbrace{\begin{matrix} A_x & A_y & A_z \\ B_x & B_y & B_z \end{matrix}}_{\text{6 box faces}} \qquad \underbrace{\begin{matrix} A_x \times B_x & A_x \times B_y & A_x \times B_z \\ A_y \times B_x & A_y \times B_y & A_y \times B_z \\ A_z \times B_x & A_z \times B_y & A_z \times B_z \end{matrix}}_{\text{9 cross-products between faces}} \tag{2}$$

The *Gilbert-Johnson-Keerthi* (GJK) algorithm is capable of testing for intersections between any convex object with varying vertices. It calculates the Euclidean distance between two polyhedra by using the Minkowski Difference (equation 3) and determining the separation distance, which is the distance between M and the origin O (Gilbert, Johnson, and Keerthi, 1988). If the origin is inside the Minkowski Difference, the objects intersect. However, the algorithm doesn't explicitly compute the difference; it samples the difference point set using a *support mapping function* defined in equation 4, which returns the furthest point in a specified direction. The GJK algorithm creates a convex simplex Q from a total combination of $dimension + 1$ points from M on each iteration. It updates the simplex on each iteration and terminates when the origin is inside, indicating an intersection (Van Den Bergen, 2001).

$$M = A \oplus B = \{a + b : a \in A, b \in B\} \tag{3}$$

$$s(M, d) = \max(v \cdot d : v \in M) \tag{4}$$

The algorithm proceeds as follows:

1. Initialise Q with $s(M, -d)$, with d being the direction vector pointing from A to B.
2. Repeat (normally for a fixed number of iterations):
   (a) Compute the support point p of M in the direction d using $s(M, d)$.
   (b) If $d \cdot p \leq 0$, terminate and return no intersection.
   (c) Add v to Q.
   (d) Perform a reduction step to ensure that Q has at most $dimension + 1$ points.
   (e) If Q contains O, terminate and return intersection.
   (f) Compute a new search direction d towards O from Q.

A subroutine is used to check for the position of the origin, depending on the number of points in the simplex. This subroutine employs a line (2 points), triangle (3 points), or tetrahedron test (4 points) that only uses dot products and does not require vector normalisation, making it a very fast process (Muratori, 2006). The subroutine also updates the direction, meaning fewer steps in the final implementation. Note that it's impossible to have just one point in the simplex as before this subroutine is invoked a minimum of two must be added.

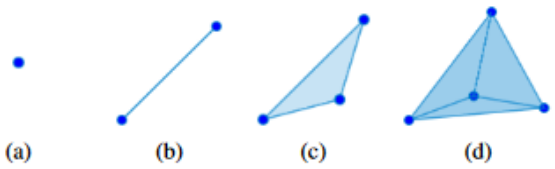


Figure 2: *The four states of the GJK simplex (Montanari, Petrinic, and Barbieri, 2017). 2a being a point, 2b a line segment, 2c a triangle, and 2d a tetrahedron.*

The *Expanding Polytope Algorithm* (EPA) is used to calculate the penetration depth and MTV of the intersection (Van Den Bergen, 2001 and Cameron, 1997). It uses the final simplex generated by GJK and finds the closest point on its boundary to the origin. EPA does this by expanding the simplex and subdividing its edges with new vertices using the following steps:

1. Find the edge of the simplex that is closest to O.
2. Compute the support point p of M in the direction d that is normal to the edge (i.e. perpendicular to it and pointing outside the polygon), using $s(M, d)$.
3. The edge is split by adding this support point to it, which forms a new tetrahedron.
4. Repeat these steps until the dot product of the vector with the last search direction is greater than a threshold, which indicates very little change.
5. Once the algorithm converges, the MTV and the penetration depth can be obtained.

3 Method

3.1 Vulkan & The Rendering Pipeline

The rendering pipeline is set up using the Vulkan graphics API, which allows full control over the GPU. Vulkan has the concept of a *Render Pipeline*, this object holds all the shaders and information on how something should be rendered. The renderer comprises various pipelines to

render different game elements, such as sprites, meshes, and the user interface (UI). Some of these pipelines will be discussed in later sections in further detail.

Render systems create pipeline configuration structures and submit them to the renderer, which sorts them based on Christer Ericson's method, where a 64-bit key is split up to represent multiple render categories and then sorted based on these values (Ericson, 2008). Figure 3 shows the encoded pipeline handle as it's laid out in memory.

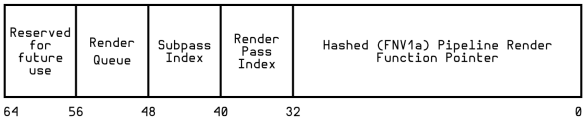


Figure 3: *The encoded 64-bit pipeline handle. Pipelines are sorted by the values present in this handle. The numbers at the bottom represent the number of bits each segment utilises.*

The renderer has four render queues, with the *opaque* and *transparent* ones being the most important. The renderer ensures that the correct pipeline is rendered in the appropriate queue to maintain the order of rendering, for example, transparent objects are rendered on top of opaque ones. Also, pipelines are bound only once to the render pass, which offers a slight performance advantage. Once added, pipelines are sorted only once; therefore, during gameplay, the pipelines are rendered in the most optimal way possible.

Debug gizmos are essential during development to quickly inspect hidden objects in the scene. A pipeline that renders box colliders in the scene was implemented, as shown in Figure 4. This pipeline renders lines instead of triangles.

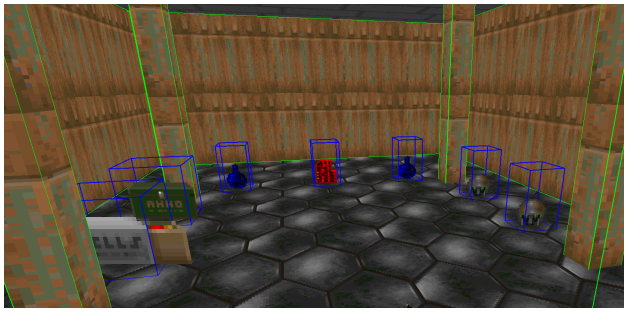


Figure 4: *OBB collider gizmos. Green boxes are colliders, and blue are triggers.*

3.2 User Interface

The UI is rendered to an offscreen texture in a separate render pass which is only submitted when the UI has been changed. This offscreen texture is essentially blit on top of the main render in every frame. This increases the frame rate as the UI is not unnecessarily redrawn when it remains the same.

Unfortunately, there is no automatic way of detecting a change in the UI, the $SetDirty()$ function is invoked manually when there has been a change which triggers a render on the next frame.

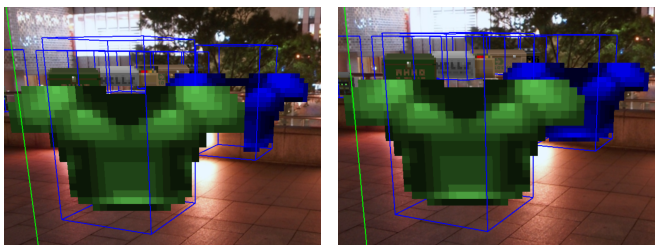
3.3 Sprites

Sprites in the game always face the camera, achieved by updating the view-model matrix before rendering, this is known as billboarding (JEGX, 2014 and Lengyel, 2019). An optimisation to the original method proposed by Jegx is to perform this once per mesh on the CPU rather than on every vertex in the shader stage. The code below computes the *World View Projection* (WVP) matrix that is passed to the vertex shader to transform each vertex; it uses the model matrix (M) of the mesh, and the camera's view (V) and projection (P) matrices.

```
mat4x4 tmpMat = V * M;

if (billboardType != BillboardType::DISABLED)
{
  tmpMat[0][0] = 1.0f;
  tmpMat[0][1] = 0.0f;
  tmpMat[0][2] = 0.0f;

  if (billboardType == BillboardType::SPHERICAL)
  {
    tmpMat[1][0] = 0.0f;
    tmpMat[1][1] = 1.0f;
    tmpMat[1][2] = 0.0f;
  }

  tmpMat[2][0] = 0.0f;
  tmpMat[2][1] = 0.0f;
  tmpMat[2][2] = 1.0f;
}

mat4x4 WVP = P * tmpMat;
```

Listing 1: *Update the view-model matrix to be billboarded, depending on the sprites type.*

Sprites with transparency caused image artefacts when rendered out of order (see figure 5a). Sorting the sprites based on distance from the camera and rendering them back-to-front solved the issue, this is formally known as the Painter's Algorithm. Although, this algorithm has drawbacks such as not handling overlapping sections correctly. Fortunately, it didn't affect the game since it only used billboarded sprites.



(a) *Before sorting sprites.* **(b)** *After sorting sprites.*

Figure 5: *A comparison image of before (5a) and after (5b) sorting sprites to solve transparency issues.*

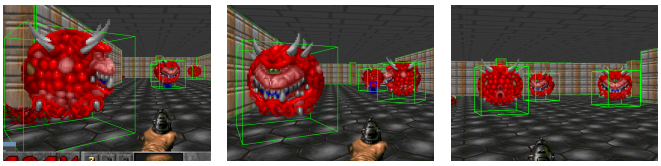
To create the illusion of 3D objects, sprites are stored in a texture array, where the following equation determines

which texture to currently render based on the player's position. Index 0 is chosen when the player is in front of the object, and the last available index for when they are behind. Where I is the chosen texture index, $\vec{D}$ is the direction to the player, $\vec{F}$ is the forward direction of the object and C is the texture count.

$$\begin{aligned} d &= (\vec{D} \cdot \vec{F}) \\ I &= ((d + 1.0)0.5)C \end{aligned} \tag{5}$$

Since the texture index is only computed for one side, the texture must be flipped when the player is on the other side. To achieve this, the x/u texture coordinate is multiplied by either 1 or -1 using the following equation:

$$\begin{aligned} \vec{cross} &= \vec{F} \times \vec{D} \\ u\ multiplier &= \begin{cases} 1.0, & \text{if } (\frac{\vec{cross}}{\|\vec{cross}\|})_y > 0.0 \\ -1.0, & \text{otherwise} \end{cases} \end{aligned} \tag{6}$$



(a) (b) (c)

Figure 6: *The rotating "3D" sprites at different angles.*

3.4 Entity Component System

An *Entity Component System* (ECS) is a performance-oriented system for organising game entities and their data, due to its efficient memory usage, data locality, and cache coherency (Fabian, 2018).

There is *no entity*. Instead, entities are IDs to an implicit collection of components. The entities are never updated, but rather components that are associated with that entity are.

Components are just structures of data that are linked to entities through managers, which use a data structure called a *Sparse Set* consisting of two arrays; one sparsely packed and the other densely packed. The dense array holds components for all entities back-to-back, while the sparse array uses the entity as an index to store the index of its associated component stored in the dense array. A sparse set enables quick $O(1)$ access for adding, removing, looking up, and clearing the set, and $O(n)$ for iteration (Manenko, 2021). Updating each component type at a time, instead of updating all components attached to an entity, improves cache utilisation as components are stored in a densely packed array. The sparse set also maintains random reads/writes from a specific entity's view.

Systems are implemented as loops that perform behaviour and logic on one or a set of components. They define how different components act together and do not hold any data themselves.

3.5 Physics & Collisions

SAT was initially used for OBB intersection tests, which produced a simple yes/no answer for box intersection. However, as mentioned in the Related Work section, more information is needed to resolve the intersection. Therefore, GJK replaced the SAT algorithm, supplemented by EPA to calculate the MTV, which is used in the collision resolution step.

However, there are some issues with the implementation of the GJK and EPA algorithms. In rare cases, a fifth vertex is added to the simplex when there should only ever be a maximum of four, the issue is not fatal to the application but means some collisions are ignored. Additionally, the EPA algorithm may fail to converge, meaning there aren't enough iterations to solve for the MTV. However, one potential solution is to increase the number of iterations, which would come at the expense of performance.

The ray-to-AABB method described in *Real Time Collision Detection* was used to implement raycasting, which utilises a test against a Kay-Kajiya slab volume (Ericson, 2004). To support an OBB, modifications were made to transform the ray into the box's local coordinate space and apply the object's rotation matrix to the box extents.

Basic *Physics Layer* support was implemented using a bit flag, which allows colliders/triggers to be tagged as a specific layer. When performing physics tests such as object intersections and raycasting the system can ignore any colliders using the following check:

```
1 #define BITFLAG_HAS_FLAG(bits, flag)
2                 (((bits) & (flag)) != 0)
```

Listing 2: *Macro that checks if a bitfield has a certain bit set.*

3.6 Audio

The in-game sound effects and music are loaded from *wav* files and implemented using the *XAudio2* library. XAudio is a low-level library for interfacing with the audio hardware and managing sound processing and mixing (Microsoft, 2021).

To load *wav* files, the online specification was followed to parse the raw binary and locate the necessary data chunks. However, an endianness problem was encountered, which was resolved by converting everything to big-endian when trying to compare values.

A simple system for playing one-shot sounds and looped audio was implemented. This system loads the audio buffers from the *wav* file, creates virtual voices (which manage audio playback in the *XAudio2* library), and submits the buffers when requested. To loop sounds, the *LoopCount* variable on the buffer options is set before submitting it to the voice.

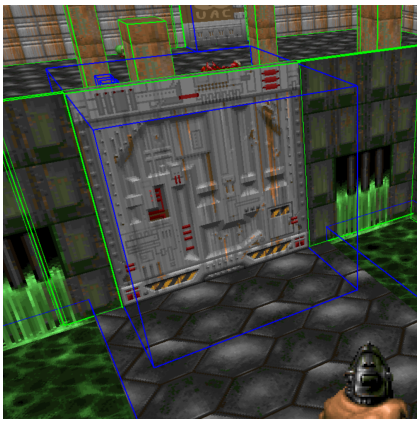


Figure 7: *In-game door with collider and trigger gizmos.*

3.7 Gameplay

Doors utilise raycasting to detect player interaction, when a player interacts with a door it opens using a simple linear interpolation and remains open until they leave the trigger area. See figure 7 where the green wireframe gizmo is the door collider which the player must interact with, and the blue wireframe gizmo is the trigger area.

Doors can be locked and require a keycard of the correct colour to open, this was implemented using bit flags. This approach uses one 32-bit unsigned integer to track which types of keycards the player has and a simple bit check to determine if the player has one of a certain type; which is implemented using the macro in listing 2.

Pickups rely on the collision system described earlier, when the player is colliding with an item it is picked up and updated on the UI.

The project replicates both types of weapon from the original Doom game: hitscan and projectile. Each hitscan weapon has unique features, such as the shotgun firing seven separate rays with a slight offset to simulate a spread, and the chaingun and pistol using one raycast but with different firing rates. The punch also uses raycasting but with a short range, only hitting enemies close to the player. Projectile weapons spawn a new entity that serves as the projectile and moves at a certain speed in the direction the player was facing when fired. When the projectile intersects with an enemy, they take damage. There is very little variation between projectile weapons except for the projectile texture and their firing rate.

4 Future Work

One potential area for improvement and future research involves enabling instancing on meshes, which would result in increased performance. This approach could also be extended to the texture system, whereby the loading of the same texture multiple times would use the already preallocated memory on the graphics card, rather than reallocating memory each time. This approach could make a huge improvement to the UI by reducing the number of textures required for each ammo counter from 30 to 10 and eliminating the need for an astronomical 240 total textures across the 8 counters.

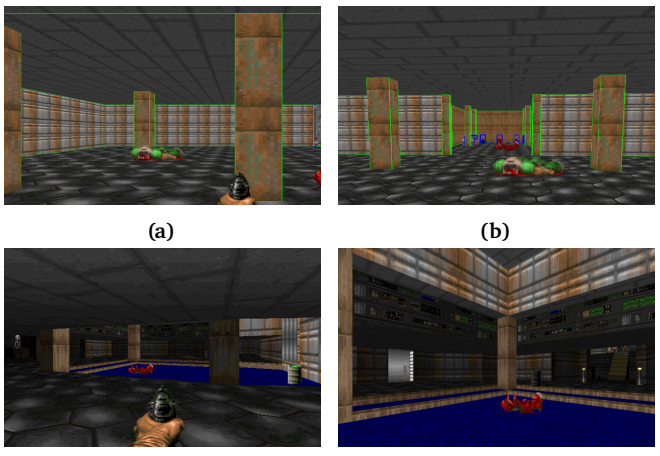
Putting together the level was a very time-consuming task, with the addition of a level editor, the designer could see what changes they were making in real-time and rapidly update them without constantly reloading the application or messing around with numbers in code, which is a huge hassle.

The audio system lacked support for advanced effects, fades, and mixing. To improve the audio system, implementing the ability to blend and fade between clips would be beneficial. Currently, changing music tracks involves stopping the current one and starting the next, which could be improved with this feature.

AI is a major area of game development and wasn't explored in the project, future work could be done on researching how to make enemies traverse the map efficiently, detecting the player and have advanced behaviour.

5 Evaluation

Compared to the original Doom game, this project shows visual discontinuities, such as varying sprite pixel sizes within the game world as they are all scaled differently. However, the UI remains fairly consistent. With the absence of hardware limitations, this project can display a wider range of colours at a time compared to the 256 that the original game could. Figure 8 compares the project with the original Doom; the project lacks lighting while the original game includes basic shading effects.



(a) **(b)** **(c)** **(d)**

Figure 8: *Figures 8a and 8b are screenshots of the final project. Figures 8c and 8d are screenshots from the original Doom game.*

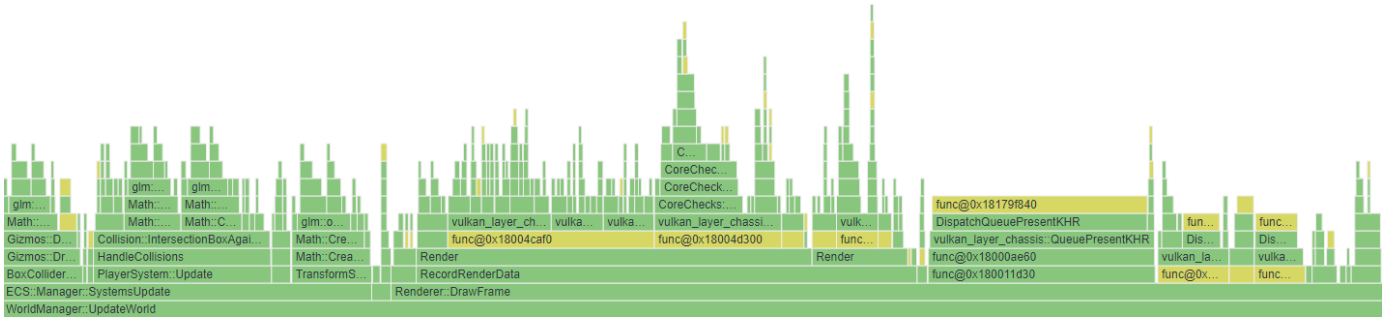


Figure 9: *A single frame call stack graph.*

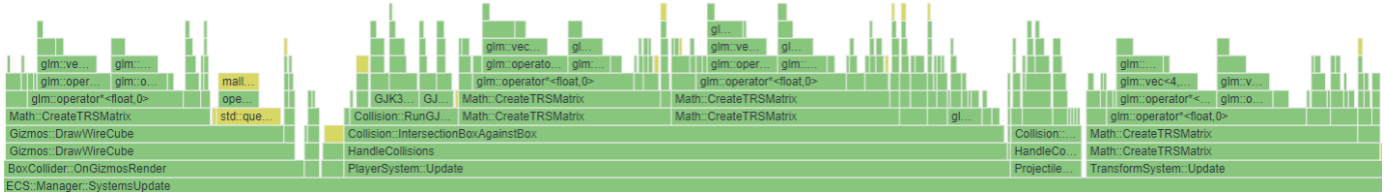


Figure 10: *A call stack graph for updating the ECS during one frame. Zoomed in from figure 9*

Figure 9 shows a single frame call stack using the Intel VTune profiler during normal gameplay. Multi-threading the renderer could be a significant optimisation as 48.7% of application time is spent recording and submitting render data to a queue, with almost half of that time spent on submission and waiting. By performing this task on another thread, the game logic could continue while waiting for the render data to submit, potentially improving overall performance.

Focusing just on the game logic (figure 10), which took about 18.1% of application time, there are clear bottlenecks for performance. The function *Math::CreateTRSMatrix* is called multiple times per frame and seems to be taking up most of the frame time. In fact, a singular invocation of this function takes up 2.8% of application time which is astounding compared to the 1.4% of time the GJK algorithm takes to perform intersection tests between the **player and every collider in the scene!**

6 Conclusion

To summarise, the created *Doom* slice implements many of the primary gameplay features and technical systems/algorithms, although they are not perfect and could benefit from further bug fixing and refinement, such as the collision resolution system. There is plenty of future work that could be achieved, with many of the engine systems only implemented in a basic state. However, given the substantial scope of the project, it is remarkable to see various systems working together to produce a final playable product.

References

Cai, Panpan et al. (2014). "Collision detection using axis aligned bounding boxes". In: *Simulations, Serious Games and Their Applications*, pp. 1–14.

Cameron, Stephen (1997). "Enhancing GJK: Computing minimum and penetration distances between convex polyhedra". In: *Proceedings of international conference on robotics and automation*. Vol. 4. IEEE, pp. 3112–3117.

Ericson, Christer (2004). *Real-time collision detection*. CRC Press.

– (2008). *Order your graphics draw calls around!* URL: `https://realtimecollisiondetection.net/blog/?p=86`.

Fabian, Richard (2018). "Data-oriented design". In: *framework* 21, pp. 1–7.

Gilbert, Elmer G, Daniel W Johnson, and S Sathiya Keerthi (1988). "A fast procedure for computing the distance between complex objects in three-dimensional space". In: *IEEE Journal on Robotics and Automation* 4.2, pp. 193–203.

Huynh, Johnny (2009). "Separating axis theorem for oriented bounding boxes". In: *URL: jkh. me/files/tutorials/Separating% 20Axis% 20Theorem% 20for% 20Oriented% 20Bounding% 20Boxes. pdf*.

JEGX (2014). *Simple Billboarding Vertex Shader (GLSL)*. URL: `https://www.geeks3d.com/20140807/billboarding-vertex-shader-glsl/`.

Lengyel, Eric (2019). *Foundations of Game Engine Development Volume 2: Rendering*. Terathon Software LLC.

Manenko, Oleksandr (2021). *Sparse Sets*. URL: `https://manenko.com/2021/05/23/sparse-sets.html`.

Microsoft (2021). *XAudio2*. URL: `https://learn.microsoft.com/en-us/windows/win32/xaudio2/xaudio2-introduction`.

Montanari, Mattia, Nik Petrinic, and Ettore Barbieri (2017). "Improving the GJK algorithm for faster and more reliable distance queries between convex objects". In: *ACM Transactions on Graphics (TOG)* 36.3, pp. 1–17.

Muratori, Casey (2006). *Implementing GJK - 2006*. URL: `https://www.youtube.com/watch?v=Qupqu1xe7Io`.

Nystrom, Robert (2014). *Game programming patterns*. Genever Benning.

Van Den Bergen, Gino (2001). "Proximity queries and penetration depth computation on 3d game objects". In: *Game developers conference*. Vol. 170.

Figure 11: *Final screenshots.*