Distributed Physics Simulation in Multiplayer Video Games

CSC8599 – Project and Dissertation for Computer Game Engineering MSc – Dissertation

Salih Eren Degirmenci  
Computer Game Engineering MSc  
 Newcastle University  
 Newcastle upon Tyne / UK  
S.E.Degirmenci2@newcastle.ac.uk

ABSTRACT

Multiplayer games typically rely on a centralized server to handle physics simulations. However, as the number of simulated objects increases, the computational demands on this server can become overwhelming, leading to performance degradation and negatively impacting gameplay. This project addresses this challenge by investigating distributed computing techniques to offload the physics simulation workload across multiple machines. By distributing the computational burden, we aim to enhance game performance and scalability, enabling the creation of exceptionally large and complex virtual worlds.

The primary objective of this research is to develop a distributed physics simulation architecture for multiplayer games. This involves designing and implementing a system that can effectively partition the physics workload among multiple servers, ensuring low latency and high fidelity in simulating large-scale environments. The project will explore different partitioning strategies, communication protocols, and synchronization mechanisms to optimize performance and scalability. Additionally, the research will evaluate the proposed architecture through rigorous testing and benchmarking to assess its impact on gameplay experience.

The expected outcomes of this research include a novel distributed physics simulation architecture tailored for multiplayer games, along with a comprehensive evaluation of its performance and scalability. By successfully distributing the physics workload, this project aims to contribute to the advancement of multiplayer game development by enabling the creation of more immersive and interactive virtual worlds. The findings of this research can potentially benefit the gaming industry by providing a scalable solution for handling complex physics simulations, thereby improving gameplay experiences for millions of players.

This research will employ a combination of theoretical analysis, simulation, and experimental evaluation. Initially, a thorough investigation of existing distributed computing techniques and physics simulation algorithms will be conducted to identify suitable approaches. Based on this analysis, a distributed physics simulation architecture will be designed and implemented. The performance and scalability of the proposed architecture will be assessed through rigorous testing and benchmarking using various simulation scenarios. Finally, the results will be analyzed to evaluate the effectiveness of the distributed approach and identify potential areas for improvement.

**KEYWORDS**

Networking, Physics Engines, Distributed Computing, Game server architecture

1 Introduction

Physics engines provide video games to simulate physics events occurring in the game in real-time. With networking, video games that can be played by multiple players(users) can be made. With increasing object count and interaction between those objects in simulated world, orthodox ways of multiplayer video game management can like single server to simulate physics can fail to provide the smooth outcome needed for decent experience.

To produce a proper solution, understanding how physics engines and networking work is an essential part of this process. The physics engines used in video game engines are mostly **real-time** physics engines. Physics engines depend on the speed of the processors in case of accuracy of the simulation. When too many events happen at the same time in simulation, it is possible to compute each event at the same time and proper accuracy.

Networking is another problem that might occur when the scale of the game increases. Increasing player count can cause too much input for one server to manage at the same time. Distributing responsibility of dealing client inputs by separating areas of the video game world might bring more performance to the solution.

By reducing the cost of computing required from physics simulations and traffic from one server, the idea of distributing physic engine to multiple machines and connecting them with a proper communication system can be a solution to this problem.

The main challenge is to overcome the synchronization of the data between different game servers in a distributed environment. Data that needs to be synchronized by distribution system contains transform data of the game objects that are dynamically changeable in the simulated environment, scripted data like player class data, object managers and more.

The data structures like object transforms, states need to be sent from servers very quickly. When the sender server count is increased, unwanted situations like packet losses, not correct results from physics simulation, reading packets with wrong order may occur.

1.1 Physics

Physics has emerged as a cornerstone in contemporary video game development, serving as the computational backbone for simulating the behavior of objects and characters within virtual environments. By applying principles derived from Newtonian mechanics, game developers can create immersive and believable gameplay experiences. From the trajectory of a projectile to the intricate dynamics of character movement, physics engines underpin the realism and interactivity of modern games. The integration of physics extends beyond mere simulation, influencing game design, player engagement, and the overall aesthetic experience. However, it's crucial to note that game physics often represents an approximation of real-world physics, tailored to optimize performance and gameplay [10].

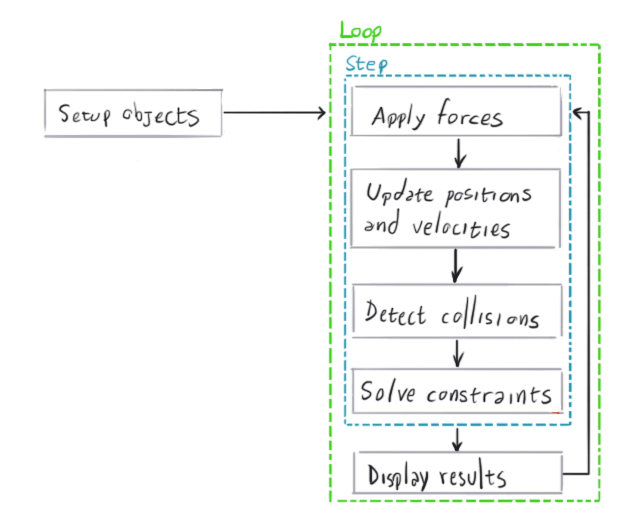


Figure 13: High level overview of the general procedure of physics engine

With the help of the networking, video games can be played together by multiple people in real-time. To achieve realistic physical events, every machine that player uses(clients) needs to visualize the virtual world same with other machines and correctly. If the architecture is designed on clients simulating physics by themselves, every client can calculate different results, and the smooth simulation can’t be achieved. For this reason, physics engines in simulations are usually simulated by one component and send their current state with using network messages using different protocols.

1.2 Networking

Networking is a critical component of modern video games, facilitating multiplayer experiences and enabling real-time interaction between players. It involves the transmission and synchronization of game data across networks, requiring careful consideration of factors like latency, packet loss, and server architecture. Networking technologies have evolved significantly, with options ranging from peer-to-peer connections to dedicated server-based systems. The choice of network architecture significantly impacts gameplay experience, scalability, and security. Moreover, networking has expanded beyond traditional multiplayer modes, influencing aspects such as matchmaking, leaderboards, and cloud saves. [11]

*1.2.1 Networking Protocols in Multiplayer Games*. Networking protocols underpin the communication infrastructure of multiplayer games, dictating how data is exchanged between clients and servers. Protocols such as TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) form the foundation, offering varying degrees of reliability and speed. TCP guarantees ordered and reliable data delivery but incurs higher latency, making it suitable for turn-based or slower-paced games. UDP prioritizes speed and low latency, often preferred for real-time action games where responsiveness is paramount. To optimize performance and address specific game requirements, developers often implement custom protocols built on top of TCP or UDP, incorporating features like data compression, encryption, and error handling. The selection of appropriate protocols significantly impacts gameplay experience, matchmaking, and overall game stability. [12]

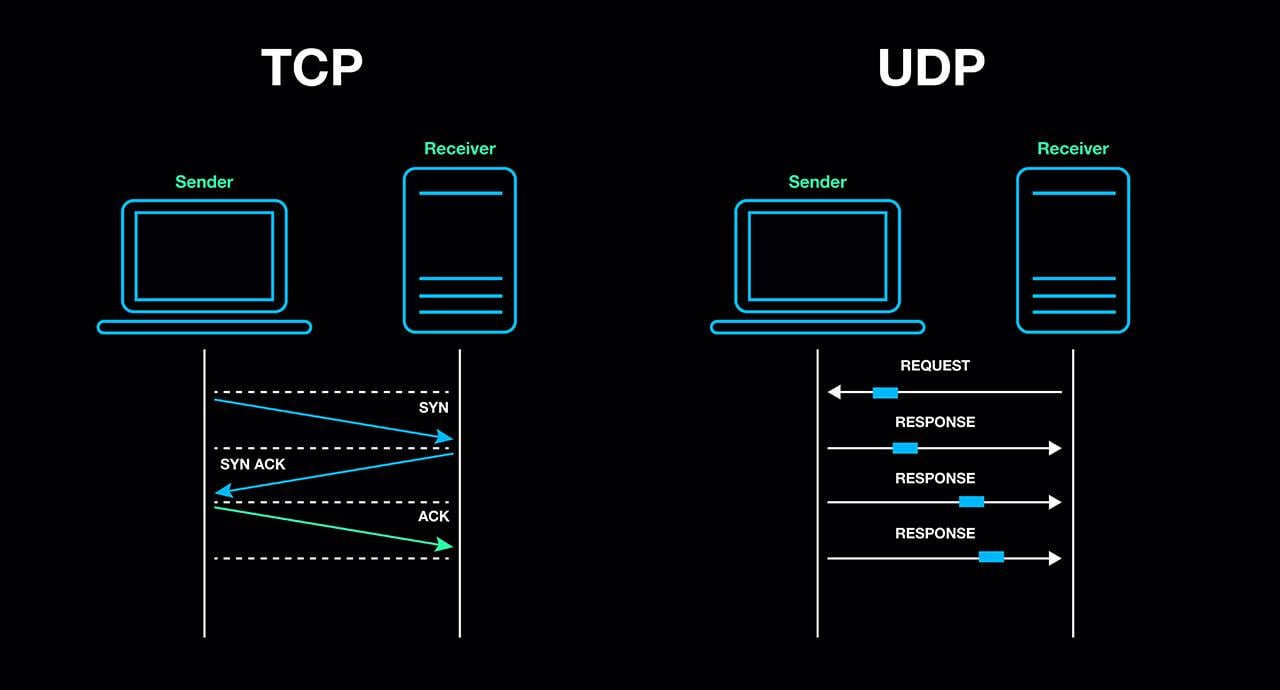


Figure 14: Visualizations of TCP and UDP network protocols

2 Background And Related Work

There has been much research and work done in both academy and professional industry related to improving multiplayer games and getting more accurate physics simulation. A proprietary commercial software called SpatialOS[1] tried distributing real-time physics simulation for large complex virtual worlds. SpatialOS is a distributed operating system designed for building large-scale concurrent applications. It leverages a spatial computing paradigm, where applications are decomposed into independent entities existing in a shared virtual space.

A diagram of a city

Description automatically generated

Figure 1: Visualization of SpatialOS architecture

(Brown et al. 2019) came up with a solution for separating responsibilities of physics engines by separating the areas of the virtual world and how to determine the transitions between distributed server areas by a system called *Aura Projection.*

To be understand every aspect of the work required for implementing distributed physics system, the below topics need to be overviewed:

1. Game Engine Architecture
2. Real Time Physics Simulation
3. Online Multiplayer Games

2.1 Game Engine Architecture Overview

Video game engines are software frameworks that combine multiple programs to simulate essential elements like physics and graphics and provide tools for building the desired video game. Video game engines are mostly based on these components:

1. Physics engine
2. Graphics engine
3. Networking
4. Artificial intelligence
5. Sound engine

These components in the framework get input from player(user) and process that input according to program written and shows the result to user with proper graphical interface.

A diagram of a software development

Description automatically generated

Figure 2: Detailed game engine architecture

2.2 Real-time Physics Simulation

Real-time physics simulations provide essential abilities for creating dynamic and interactive experiences across broad spectrum of disciplines, encompassing not only the realm of video games but also engineering design, architectural modeling, and even scientific research [2]. The key component of these simulations are physics engines. Physics engines are software systems that attempt to simulate the real world’s physical behavior within a virtual environment. This emulation relies on established principles of physics, such as Newton’s laws of motion, for calculating forces, motion, and collisions between objects [3].

In multiplayer games, physics engines mostly run on server and clients visualize the objects without calculating physics related operations to prevent players are not getting different representations of the game. This creates a smooth experience for everyone.

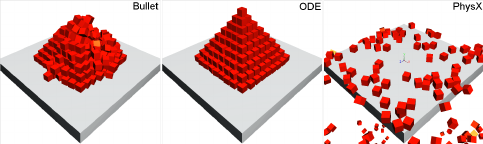


Figure 3: Virtual runtime experiments used different physics engines

The three main functionalities of physics engines are listed below:

1. Integration
2. Collision Detection
3. Collision Response

*2.2.3 Integration.* Change in objects position over time is described by the velocity property, and the rate of change of velocity as acceleration. [20] Integration is a fundamental operation within physics engines, used to predict the future state of physical systems based on their current state and applied forces. Numerical integration techniques, such as Euler's method, Verlet integration, and Runge-Kutta methods, are employed to approximate the solutions to differential equations governing the motion of objects. These methods iteratively update the positions and velocities of objects over time steps. The choice of integration method significantly impacts the accuracy, stability, and performance of the physics simulation. Factors like time step size, numerical stability, and energy conservation must be carefully considered when selecting an integration scheme. [19]

A diagram of a graph

Description automatically generated

Figure 3: Virtual runtime experiments used different physics engines

*2.2.3.1 Linear Motion.* Linear motion describes an object moving in a straight line, characterized by its displacement, velocity, and acceleration. [22] Displacement refers to the change in an object's position relative to a reference point, while velocity is the rate of change of displacement over time. [21] Acceleration, in turn, measures the rate of change of velocity. Linear motion can be categorized into uniform motion, where an object covers equal distances in equal time intervals, and non-uniform motion, exhibiting varying velocity. Grasping the fundamentals of linear motion is crucial for understanding more complex motion patterns in physics.

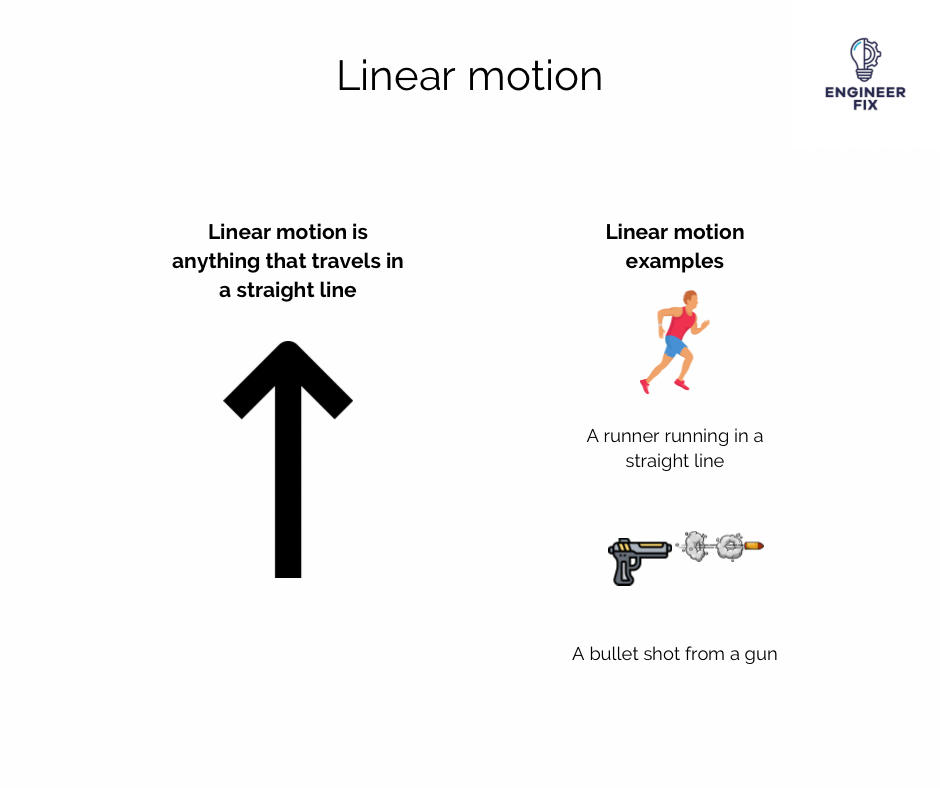


Figure 3: Linear motion examples

Following variables will be used to describe linear motion:

* a-Acceleration
* v- Velocity
* s- Displacement
* n – Current time step

To integrate the linear motion into physics engines, numerical integrator called explicit Euler integration is used:

A mathematical equations with black text

Description automatically generated with medium confidence

This integration might cause instability when used in physics engines when time steps are short, instead much reliable integration called semi-implicit Euler is often used in physics engines. Difference between explicit Euler is updated velocity is used before calculation of new position. It is faster to compute and remains accurate over many iterations and stable when longer time steps are used [23]

A mathematical equations with black text

Description automatically generated with medium confidence

*2.2.3.2 Angular Motion.* Angular motion describes the rotational movement of an object around a fixed axis or point. Key quantities characterizing angular motion include angular displacement, angular velocity, and angular acceleration. Angular displacement measures the change in the object's orientation, while angular velocity represents the rate of change of angular displacement with respect to time. Angular acceleration, in turn, indicates the rate of change of angular velocity. Torque, the rotational equivalent of force, influences angular motion. Understanding angular motion is essential for analyzing the behavior of rigid bodies and rotational systems in various fields, including physics, engineering, and robotics.[22]

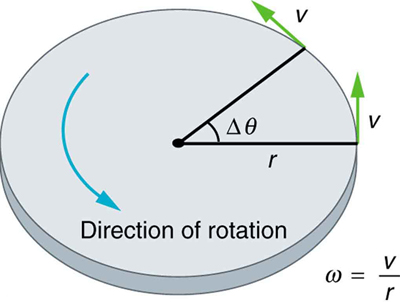


Figure 3: Visualization of calculating angular velocity

*2.2.2 Collision Detection.* Collision detection is a fundamental aspect of physics engines, determining whether objects within a virtual environment are intersecting. The part of objects that will be operated by physics engines are represented by structures called colliders. These structures are mostly created from simple geometric objects like boxes, capsules, spheres and cylinders. For more complex collision volumes for more accurate collision detection, such as convex volumes. Physics engines try to identify potential collision pairs through the collision volumes. Once potential collisions are identified, precise intersection tests are performed using geometric algorithms. The results of collision detection provide critical information for physics simulations, such as contact points, normal, and penetration depths. Efficient collision detection is essential for maintaining real-time performance, especially in games with numerous objects interacting dynamically. [13] Testing every object against every other object in our scene will quickly lead to a lot of wasted computation. To alleviate this, physics engines will split the process of determining and resolving collisions up into two separate phases, the broad phase, and the narrow phase.

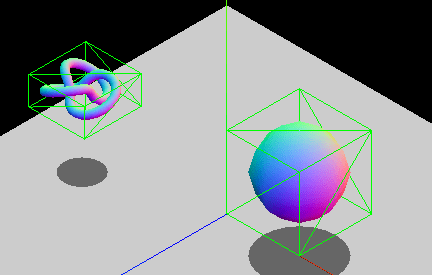


Figure 4: Axis-Aligned Bounding Box (AABB) collision volume

*2.2.3 Collision Response.* After detecting the objects that collides during the Collision Detection phase, collision response phase starts. Collision response determines how objects behave after a collision is detected. This involves calculating the effects of the collision on the objects' velocities, positions, and rotations. Impulse-based methods are commonly employed to instantaneously change object velocities, preserving momentum and energy conservation. Additionally, contact constraints are used to prevent interpenetration and keep object separation. Advanced collision response techniques incorporate friction, restitution (bounciness), and material properties to simulate realistic interactions. Accurate and efficient collision response is crucial for creating believable and engaging gameplay experiences. [13]

There are methods for resolving collisions suitable for different cases. Primary methods are listed like this:

* Projection Method
* Impulse Method
* Penalty Method

*2.2.3.1 Projection Method.* Simple and most intuitive method for resolving a collision. It can be achieved by separating collided objects by changing their positions in opposite directions along the collision normal.[14] By moving objects along the collision normal, penetration is resolved. While simple to implement, this method can introduce inaccuracies, particularly when dealing with objects of significantly different masses or complex geometries. Additionally, it may lead to unstable simulations if not carefully implemented. Despite its limitations, the projection method remains a viable option for certain types of games or when computational efficiency is a primary concern. [13]

A diagram of a collision

Description automatically generated

Figure 5: Resolving collision with projection method.

*2.2.3.2 Impulse Method.* It’s possible to resolve collision by modifying the first derivate of position; means the velocities of the colliding objects. Calculating the a new velocity for each object that will move them apart in a realistic manner, taking into consideration their relative mass and current velocity. An instantaneous change in velocity, without first integrating from acceleration regarding time, is known as an impulse, and so collision response using these is known as the impulse method. [14] This approach often yields more physically plausible results compared to the projection method, as it directly addresses the cause of the collision—the objects' velocities—rather than simply correcting their positions. By applying appropriate impulse magnitudes based on factors such as relative velocity, mass, and restitution coefficient, the impulse method can simulate a wide range of collision behaviors, from elastic bounces to inelastic impacts.[15]

Both linear and angular velocities of the objects should be taken into account and modified accordingly.

A diagram of a circle and a circle

Description automatically generated

Figure 5: Resolving collision with projection method

To calculate linear impulse, the relative velocity between the objects should be taken into consideration:



To calculate how much of the relative velocity is in the direction of the collusion normal, usage of dot product operator. With combining the coefficient restitution of the colliding objects, total velocity of the collision can be calculated.



To ensure conservation of momentum is done, velocity needs to be scaled so that each object applied gets the correct proportion of the total energy of the collision.

A math equation with numbers and symbols

Description automatically generated

After the impulse J is calculated, velocities of the colliding objects can be adjusted.

A math equations with numbers

Description automatically generated with medium confidence

After collision, not only linear impulse will be generated, but also an angular impulse will be generated. To accurately simulate the rotational effects of a collision, calculating the change in angular velocity is essential. This involves determining the torque generated by the collision impulse. The torque is calculated as the cross product of the collision point relative to the object's center of mass and the applied impulse. The resulting torque is then used to update the object's angular velocity using the inverse inertia tensor. This process ensures that objects respond realistically to collisions, exhibiting appropriate rotational motion based on the impact point and the object's mass distribution. [14] To calculate the angular velocity, cross product between the relative collision point and the collision normal must be calculated. Then crossing it by the relative position on the object once again gives an objects angular velocity.

A close up of a number

Description automatically generated

The modified angular velocity of the colliding objects can be calculated by getting cross product between the local collision position and the applied impulse force, scaling the result by the inverse inertia tensor.

A number and plus one

Description automatically generated with medium confidence

Finally, the combined impulse can be found with taking the dot product of angular velocities between two objects versus the collision normal direction, then combining the result with the divisor of the impulse vector, along with the inverse masses.

A math equation with a line of letters

Description automatically generated with medium confidence

*2.2.3.3 Penalty Method.* The penalty method addresses collision resolution by introducing repulsive forces between overlapping objects. These forces act as virtual springs, pushing objects apart. While conceptually simple, this approach often suffers from stability issues, leading to oscillations and unrealistic behavior. Accurately determining the appropriate force magnitude is challenging, as excessive force can cause numerical instability, while insufficient force may result in object interpenetration. Despite these drawbacks, the penalty method finds applications in specific scenarios, such as soft body simulations or when real-time performance is critical and precise collision handling is less essential. [16, 17]

To address some of the limitations of the basic penalty method, a viscoelastic model can be introduced. This model incorporates both elastic and damping components. Penalty force vector can be calculated by multiplying damping coefficient(c) with relative velocity then subtracting it from stiffness coefficient. [18]

F = -k \* penetrationDepth - c \* relativeVelocity

Diagram of a diagram of a fruit

Description automatically generated

Figure 6: Penalty method collision response visualization

2.3 Online Multiplayer Games

With the power of the internet, video games that can be played with players all across the world and together in many different virtual worlds are made possible. New genres like MMORPG’s(Massively Multiplayer Online Role Playing Game), MOBA(Multiplayer Online Battle Arena), Online FPS’s and more.



Figure 7: Image from MMORPG game World of Warcraft

With the diverse range of online games and genres, developers face the challenge of creating a multitude of new features. Essential elements such as matchmaking systems to connect players, leaderboards to track progress, and intricate progression systems to keep players engaged are just a few examples of the complexities involved in developing a successful online game.

There are various online multiplayer game architectures that have been used by developers in industry. With changing client count and design of the game made developers coming up with these architectures to build their games:

There are various online multiplayer game architectures that have been used by developers in industry. With changing client count and design of the game made developers coming up with these architectures to build their games:

* Client-Server
* Peer-to-peer(P2P)
* Hybrid

*3.1.1 Client-Server.* This architecture is the most common one. Clients connect to a central server that manages the game world, game logic and synchronizes information for all players. It offers strong control and security but relies heavily on robust servers for smooth gameplay. Commonly, the server runs a special copy of the product, that does not include any rendering task. They are running the tasks like physics, AI and broadcast the behaviors of the game objects to all clients to make sure everyone receives the same state and run the product together at the same time. It provides a logic to operate the inputs of the players received from the client machines.

A diagram of a computer

Description automatically generated

Figure 4: Client-server architecture

The client-server architecture is well-suited for multiplayer games that require persistent world, where many real-time interactions are required and where game logic needs to be centralized. [12] Especially genres such as MMORPG’s, Competitive PvP (Person versus person) first-person shooters (FPS) and real-time strategies heavily rely on this architecture. These genres benefit from the persisted worlds, centralized authority for managing game’s state, and due to clients can only communicate with inputs, they benefit being able ensure fair-gameplay and prevent cheating.

This model can introduce latency and scalability challenges as the number of players increases or the simulated world gets bigger and complicated. To overcome the limitations of client-server model, developers came up with various optimizations and enhancements. Load balancing distributes players across multiple servers to improve performance and reduce latency. Sharding partitions the game world into smaller segments, allowing for more efficient management of player data and interactions. Dedicated game servers, optimized for specific game mechanics, can improve performance and scalability. Additionally, techniques like client-side prediction and interpolation help to mitigate network latency and provide a smoother gameplay experience. [12]

Another method to overcome the network latency problem is implementing prediction algorithms to client-side programs or server reconciliation to prevent situations like problems with character movement, interactions with the objects and more game actions. [24] But, making the latency problem between machines that communicate on network protocols completely disappear is impossible.

One of the major problem is risk of single points of failure. With one the games or simulations that run on only one server, integrity depends on that one server. If any critical malfunction happens on hardware or software, whole system will be not operatable. SpatialOS brings a solution for this problem, with being able to distributing game area to multiple servers, if one server is down, other servers in the environment takes the responsibility of simulating and communicating jobs to themselves and simulation lifecycle interruption can be prevented.

*2.3.2 Peer-to-Peer(P2P)*. In this model, players connect directly to each other without a central server. This can be efficient for games that do not require too many players. But due to the host is a player’s machine, this architecture is not so protective against cheating. In a P2P network, each participant, or peer, acts as both a client and a server, sharing resources and communicating directly with others. This decentralized approach can potentially enhance scalability, reduce latency, and lower infrastructure costs compared to traditional client-server setups. However, challenges such as network topology management, data synchronization, and security must be carefully addressed to ensure optimal performance and reliability. [24]

A diagram of a computer network

Description automatically generated

Figure 5: Peer-to-peer architecture

Peer-to-peer architecture finds application in specific game genres that prioritize player interaction and resource sharing. Games that emphasize player-driven economies, such as virtual worlds or trading simulators, can benefit from P2P networks by facilitating direct interactions between players. Additionally, cooperative multiplayer games where players collaborate to achieve common goals can leverage P2P to distribute gameplay responsibilities and reduce reliance on centralized servers. However, competitive multiplayer games often require a centralized authority to ensure fair gameplay and prevent cheating, making client-server architecture a more suitable choice in these cases.



Figure 4: Animal Crossing: New Horizons uses peer-to-peer networking for multiplayer mode.

While P2P architecture offers potential advantages, it also presents significant challenges. Network topology management, the process of establishing connections between players, can be complex and prone to disruptions. Player discovery can be difficult, especially in large-scale games. Ensuring data consistency and synchronization across multiple peers is another critical challenge. Cheating and hacking are also more prevalent in P2P environments due to the decentralized nature of the network. Additionally, P2P games often require sophisticated matchmaking systems to connect players with similar skill levels and geographic locations, which can be computationally intensive.

*2.3.3* *Hybrid.* This architecture combines elements of clients-server and peer-to-peer architecture. The server might handle core game logic and world state, while clients handle communication and some other aspects directly with each other.

When creating an online multiplayer game, developers need proper ways deal with these problems, consistency, latency, fault tolerance, cheating, data integrity. These challenges may have different approaches to deal with them in different architectures.

3 Design and Implementation

This section delves into the design and implementation of the system's core component: distributed physics simulation. The fundamental principle involves partitioning the simulated world into discrete segments, each managed by a dedicated server within the system. This distributed approach empowers multiple physics servers to concurrently handle entities within large-scale virtual environments, thereby enhancing simulation efficiency and scalability. To achieve this, enhancements and new abilities need to be implemented to the physics and networking pipeline of the usual multiplayer game building approach.

The requirements to decide which component is needed in the system are listed below:

* Manage components in the system by establishing or helping establishing connections.
* Overwatching the components throughout the simulation lifetime.
* Giving commands to the system components.
* Simulating the virtual environment in the simulation.
* Visualizing the virtual environment that is simulated by the system.
* Provide a user interface for users to interact with the virtual environment that is simulated by the system.

3.1 System Architecture

The proposed distributed physics system is designed with four main components. The first component is the overseer component: Distributed System Manager. This component helps other components to establish connections with the required components in the system. The second component is the main worker: Distributed Physics Server. Physics servers simulate a part in the virtual environment that is assigned to them by distributed manager, they broadcast the states of the entities in their responsible areas and communicate with the other servers for making transitions of the objects that are leaving their borders.

The third component is the visualizers of the system: Game Clients. Game clients are mainly responsible for receiving broadcast data from the game servers and visualize them using renderers powered by graphical API’s. If the product wants to be developed includes interacting with the users, game clients provide a user interface for users to interact with the visual environment and clients send the input data to the physics servers for them to process. The last component in the system is the component that is responsible for managing machines that run physics server instances: Physics Server Middleware. Main responsibility of this program is receiving run physics server instance commands and execute a new physics server program.

A computer screen shot with a computer and a computer diagram

Description automatically generated with medium confidence

Figure 6: System topography

3.2 Used Technologies

To build the components discussed in the previous part and implement the features required for the distributed physics simulation, a strong programming language which can work fast. Due to the system requires multiple components that serves different purposes, programming environment should support a tool that manages building multi-project solution with different parameters.

C++ was selected for its capacity to directly manipulate memory, enabling optimal computational efficiency. Given the system's multifaceted nature, a robust build management tool is required to orchestrate the development of diverse components and configurations. CMake, a cross-platform build system, was employed to streamline this process. Leveraging Newcastle University's NCLGL networking and physics libraries, in conjunction with these core technologies, provided a solid foundation for the development of the distributed physics simulation.

3.3 Physics Server

The physics server is the core component in the system. The distributed system is designed to have a cloud server cluster in it. Each machine in this cluster is connected to the distributed manager component as a client to receive messages related to distributed system commands.

The physics server runs a physics engine instance inside of it. This requires objects that are responsible inside of the server’s area need to be created and simulated during their time inside of the server’s responsible borders. Due to that, when a call for start as a game server from distributed manager comes, servers open a port for other servers and clients connect. On this port, the server streams game object data for real-time simulation. Also, as a requirement of physics engines, other events like collision resolving are calculated inside of the server.

The physics engine has features like resolving collisions between objects by their bounding volumes. It supports volumes like AABB (Axis-Align Bounding Box), OBB (Oriented Bounding), sphere and capsule. The physics system can interact with objects in the world by applying force to them. Also, the physics system checks for collision detection using the broad phase/narrow phase controls. In the broad phase, the objects that have possibility to collide with each other are determined and stored into a possible colliding objects list commonly known as *collision pair*. In narrow phase, physics system iterates over the collision pairs and checks for is there any collision between them. If there is a collision, the system resolves it.

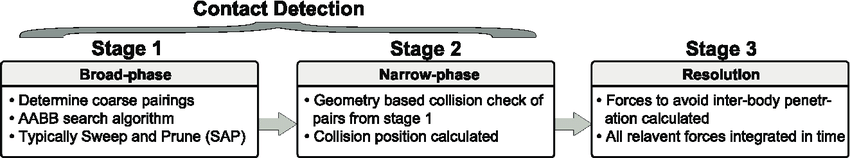


Figure 12: Broad phase – narrow phase physics update stages

Physics server in the distributed system are responsible for simulating objects in an area they are assigned for. Physics objects in these servers can move outside of these borders through other servers. Objects that are going outside of these areas should be handled. To able to handle these operations, servers requires to know which object will be leaving their borders before it actually happens.

*3.3.1 Physics Prediction.* Accurate physics simulation is one of the key challenges of these research. Especially in the cases like collision of the objects that are in the different physics server borders or object which is leaving its current simulated borders. To address this, a physics prediction system is proposed. This research focuses on developing a mechanism to accurately forecast the future position and orientation of objects within a fixed timestep. By anticipating object movement, the system aims to optimize server load, prevent object loss, and enhance overall simulation performance. This dissertation explores the design, implementation, and evaluation of a physics prediction system, with the goal of improving the efficiency and scalability of distributed physics simulations.

|  |
| --- |
| **Algorithm 1:** Predict Future State of Physics Objects |

1: **procedure** PredictFutureState(physicsObject, predictionTimeStep)

2: ▷ Retrieve current state of the physics object

3: currentPosition := physicsObject.GetPosition()

4: currentVelocity := physicsObject.GetVelocity()

5: ▷ inverseMass := physicsObject.GetInverseMass()

6: force := physicsObject.GetForce()

7: applyGravity := physicsObject.ShouldApplyGravity()

8: ▷ Calculate acceleration

9: acceleration := force × inverseMass

10: **if** applyGravity then

11: acceleration := acceleration + GravityVector

12: **end if**

13: ▷ Predict future position

14: futurePosition := currentPosition + (currentVelocity × predictionTimeStep) + (0.5 × acceleration × predictionTimeStep^2)

15: ▷ Predict future velocity

16: futureVelocity := currentVelocity + (acceleration × predictionTimeStep)

20: ▷ Store predicted state in physicsObject

17: physicsObject.SetPredictedPosition(futurePosition)

18: physicsObject.SetPredictedVelocity(futureVelocity)

19: **end procedure**

Using this algorithm to calculate the future positions and orientations, physics servers knows the objects that will leave their borders.

*3.3.2 Network Management.* The networking framework has the ability to host a server by opening a port and connecting to the servers as a client by connecting an existing server’s connection information. Game servers has the ability to create connections between clients, distributed manager and with the other physics servers in the system. Connections to the clients include hosting a server for them to receive snapshots of the simulation every time the server ticks. These snapshots of the simulation are sent from server to the clients as data packets that are called full packet. Full packets include every state of an object like objects position, rotation. When clients receive a packet, they send another packet to the server that they are receiving the snapshot data for acknowledgement. But not every object has a change in their data per tick. To prevent sending large data to the clients in every snapshot, a data packet called delta packet is used. Each time clients receive an acknowledgement, rather than sending a full packet, the server compares the last successfully sent and received snapshot data to current snapshot and creates a data packet from the difference and sends it.

The pipeline of game server with starts with Distributed Manager sending a packet that includes port information, ID assigned from the distributed manager, border point coordinates that server is responsible managing for connecting server to the manager. Currently servers are started as another process on the machine that runs the distributed manager program.

When the physics server program has been started and connected to the distribution manager, the program logs information about the data parsed. System also created for logging any system command for understanding the current state of the physics server. Also, currently due to the starting physics server as separate process in same machine with the distributed manager, data is sent by command line arguments to the program rather than data packet. This feature will be kept for debugging purposes.

Connection between the physics servers is essential. Through these connection, physics servers can send the data

A screenshot of a computer

Description automatically generated

Figure 11: Physics server program output when started and received start server packet from the distributed manager successfully.

After starting the physics server successfully, the server creates necessary objects in between responsible areas and positions them accordingly. After that, the physics server hosts another server on a different port for streaming snapshot data for clients. For clients to be able to connect to this server, the physics server sends an acknowledgement packet to the distributed manager for sending the connection data to the clients that connected to distributed manager.

After all physics servers successfully started and all clients connected to the started physics servers and finally received the game start packet from distributed manager, the physics servers start simulating the game world and starts streaming the snapshot data.

3.4 Distributed Manager

The distributed manager is responsible for controlling the systems’ management events. It is connected to cloud machine cluster and able to start physics servers by sending them necessary data related to the game.

The data includes the borders for a game server to be responsible in the virtual world. This border values are calculated before the system begins. The algorithm used to determine border values are dependent on the corner values of the game world area, total game server count.

Algorithm to calculate physics server’s responsible area in virtual environment:

|  |
| --- |
| **Algorithm 2:** Calculate Physics Server Boundaries |

1: **procedure** CalculateServerBorders(serverNum)

2: ▷ Initialize a new GameBorder object

3: border := new GameBorder(0.0, 0.0, 0.0, 0.0)

4:

5: ▷ Calculate number of columns and rows

6: sqrtVal := sqrt(2)

7: ceilVal := ceil(sqrtVal)

8: numCols := cast\_to\_int(ceilVal)

9: numRows := cast\_to\_int(ceil(2 / numCols))

10:

11: ▷ Calculate dimensions of each rectangle

12: rectWidth := (X coordinate maximum value - X coordinate minimum value) / numCols

13: rectHeight := (Z coordinate maximum value – Z coordinate minimum value) /numRows

14:

15: ▷ Determine row and column for the server

16: row := serverNum / numCols

17: col := serverNum % numCols

18:

19: ▷ Set borders for the server

20: border.minX := (X coordinate minimum value) + col \* rectWidth

21: border.minZ := (Z coordinate minimum value) + row \* rectHeight

22: border.maxX := **if** col == numCols - 1 **then** (X coordinate maximum value) **else** border.minX + rectWidth

23: border.maxZ ← **if** row == numRows - 1 **then** (Z coordinate minimum value) **else** border.minZ + rectHeight

24:

25: **return** border

36: **end procedure**

A program created for managing connections between clients and game servers, managing lifecycles of game servers and managing game instances for this component needs to be able to create a communication line between those components. Using the ENet library, necessary toolset for these communications can be gained. “ENet's purpose is to provide a relatively thin, simple and robust network communication layer on top of UDP (User Datagram Protocol). The primary feature it provides is optional reliable, in-order delivery of packets. [3]”. Program hosts a connection for all game servers for to connect at the beginning of the distributed system. When a cluster of machines starts up by manager for a game instance, first they are connecting to this hosting as clients. With being able to have this connection, the manager can send and receive data from servers as system commands or other data.

A screenshot of a computer

Description automatically generated

Figure 8: Distributed Manager Program

Distributed manager program starts the program lifecycle with creating connection with *Physics Server Middleware* programs. With these connections manager program can send start a server instance command to a physics server middleware to trigger it to run a physic server instance.

Distributed physics system is responsible for creating game instances. This instance data contains a unique identifier of the instance called game instance ID, how many clients are going to connect to the instance, and how many servers will be created to simulate the instance world.

After an instance is created, game clients expected to connect to distributed manager as a client. After connection is established, a packet sent from client to the game manager that includes a connection type and game instance ID that it tries to connect. If an instance is not existing managed by the manager, it sends a handshake packet that client cannot join that instance. If an instance is available; manager adds the client to the players of that instance, assign an ID to the game client then sends a handshake packet including the result of connection, player number.

//TODO ADD A FLOWCHART FOR OPERATION ABOVE

All clients need to connect to the manager and receive the game instance data to run the simulation. After all required clients connected to a game instance, manager sends packet to physics server middleware’s to start needed physics servers. Created physics server instances connects to the distributed manager as a client first.

A screenshot of a computer

Description automatically generated

Figure 9: Physics server successfully connected to the distributed manager

After a new physics server successfully connected to the distributed manager, physics server sends a packet with opened port number of data distribution server for all clients to connect to the physics server. When distributed manager received the packet, sends another packet to all connected clients with information included combined with IP address in the physics servers’ packet. Clients use the IP address and port information to connect to the physics server. Then the distributed manager needs the confirmation packets from every connected client and physics server that required connections are completed. After every required connection is made and handshakes are completed, the distributed manager sends game start packets to start the simulation.

//TODO ADD A FLOWCHART FOR OPERATION ABOVE

A screenshot of a computer program

Description automatically generated

Figure 10: Distributed Manager started simulation

3.5 Game Client

The game client is responsible for visualizing the simulated world and doing various interactions with the user. To visualize the simulated world, clients create the objects by combining assets like meshes, textures, animations and place them in the world by using parsed level data. For interactions, game uses platform specific tools to perform platform specific tasks such as interacting with the controller for PS5 or communicating with the platform hardware on lower-level communication.

Game clients can connect to the Distributed Manager using their join a multiplayer game functionality. After joining a multiplayer game, they are redirected to the waiting screen. After distributed manager starts the necessary physics servers, they receive packets for them to connect opened physics server snapshot data distribution ports. After connecting successfully to the both of the servers, the client will wait until the game start packet comes from the distributed manager until all of the connections are created and acknowledgements are made in the system both client and server-side.

A screenshot of a computer

Description automatically generated

Figure 7: Distributed system client main menu

*3.3.1 Graphic API’s.* Graphics APIs (Application Programming Interfaces) function as a bridge between software applications and the Graphics Processing Unit (GPU) on a computer. These APIs provide a set of functions and tools that allow programmers to manipulate and render graphics on the screen [4]. They function as a translator, taking high-level commands from the application and converting them into instructions the GPU can understand. To be able to show graphics, the program needs to have implementation of the required graphic API of the required by the wanted platform. The client program currently implements two graphics API’s: OpenGL for running graphics tasks on Windows platform, GNM implementations for running graphics tasks on PS5 platform.

*3.3.2 Platform API’s.* Platform APIs, also known as native APIs or system APIs, provide applications with programmatic access to the core functionalities of a particular operating system or platform [5]. Windows API(Win32 API) provides necessary structures for communicating with the operating system and the hardware on Windows platform to be able to perform tasks like opening program as a window.

A screenshot of a computer

Description automatically generated

Figure 12: Game client successfully connected to game server packet sender port

3.6 Physics Server Middleware

Physics server middleware program is the overseer node for creating new instances of physics servers as new processes in the machine they are responsible for. When a new machine wanted to be added in to the distributed physics system, this program needs to be run in that machine.

Middleware accepts an IP address and a port number to connect the distributed manager in that system as a client. With this connection established, it can receive commands to create new physics server instances for game instances in the system.

4 Results and Evaluation

The Distributed physics simulation is a complex system with multiple software components with running on multiple machines. Evaluating these components needs different metrics, tools and test scenarios.

The components that will be tested throughout this chapter are listed below:

* Game Client
* Physics Server

To evaluate the system, two different server layouts Physics servers need to achieve these things in different server layouts. For both servers will be tested with two clients(players) connected to the system. For each component in the system, different metrics will be evaluated. The layouts used in the tests are listed below:

1. 2 Server Layout
2. 3 Server Layout

While testing each of these components, a profiler library is developed using ImGui technology to visualize it. The Profiler class has the members of test metrics needed and the programs dynamically change these members by accessing it.

A black and white diagram

Description automatically generated

Figure 4: 2 Server Layout

A black screen with white text

Description automatically generated

Figure 4: 3 Server Layout

For components, profiler will be capturing the metrics related to memory usage listed below:

* Virtual Memory Used by Program
* Total Virtual Memory
* Useable Virtual Memory
* Physical Memory Used by Program
* Useable Physical Memory
* Total Physical Memory

4.1 Profiler

To effectively analyze the performance and behavior of the distributed physics server, a custom profiler tool was developed. This tool was instrumental in gathering detailed performance metrics, identifying bottlenecks, and evaluating the impact of various optimizations. By providing in-depth insights into the attached programs internal workings, the profiler enabled a comprehensive analysis of the system's efficiency and scalability, ultimately informing the design of performance enhancements.

Profiler has two main components, “Profiler” class for managing states and metrics variables, providing interfaces to access and modify these values. The second component is the renderer that requires a window handle and a profiler type (components in the distributed physics simulation) to render appropriate metrics.

4.2 Distributed Physics Server

There are two things’ physics server needs to carry out with performance for having a smooth simulation experience. First, a physics server needs to run physics updates correctly and time efficiently. Second, they need to broadcast the data of game objects that are in their responsible borders. The metrics listed below captured using the profiler:

* Physics Update Time
* Physics Prediction Time
* Full Snapshot Broadcast Time
* Delta Snapshot Broadcast Time

These metrics help test the capability of physics servers built for this research executing tasks required for the distributed system.

*4.1.1 Results In 2 Server Layout.* Two different object count used for the tests. At first, a single object instantiated for each server. Then 100 objects instantiated for each server in the layout.

Captured profiler output for the first scenario shows us the physics servers hardware usage and captured metric values under lightweight.

A screenshot of a computer

Description automatically generated

Figure 4: Physics server profilers for 2 server layout tested with single objects per server

For the second scenario hundred objects per physics servers are instantiated. Captured metrics showed

A black screen with white text

Description automatically generated

Figure 4: 3 Server Layout

*4.1.2 Results In 3 Server Layout.* Two-phase testing scenario that changes the object count described in the other layout used to test this layout too.

4.3 Game Client

Game clients needs to render decent amount of frames per second for smooth visual experience; needs to execute network tasks to receive snapshots and show visualize the game objects correctly and send user inputs to the servers.

* Frames per Second
* Network Time
* Program Update time

*4.3.1 Results In 2 Server Layout.* Two-phase testing scenario that changes the object count described in the other layout used to test this layout too.

*4.3.2 Results In 3 Server Layout.* Two-phase testing scenario that changes the object count described in the other layout used to test this layout too.

5 Conclusion

This research has demonstrated the feasibility of employing multiple physics servers to construct and simulate expansive virtual environments. The successful implementation of the distributed physics server system presented in this dissertation provides a foundation for the development of large-scale multiplayer video games and simulations.

Key to this achievement was the development of a robust distributed manager program. This component effectively oversaw network connections and event management across multiple server instances, establishing a scalable and consistent framework for server-client interactions. By distributing the computational load of physics simulation across multiple machines, the system achieved real-time performance for complex game worlds.

The integration of game client functionality further validated the system's capabilities. The client's ability to establish connections with multiple physics servers and visualize the simulated world in real-time underscores the effectiveness of the distributed architecture. Through the provision of user interfaces and interactive elements, the game client successfully transformed the simulated environment into an engaging user experience.

With the physics server middleware, containers of machines to be used as physics servers can be added to the distributed system, and can be triggered the creation of new instances to simulate virtual environments for a game instance by it.

5.1 Researchers Experience

Designing and implementing the distributed system was a real challange. To be able to coming with the ideas to make the things workout, knowledge of how physics engines are built, how real-time physics simulations are possbile; how networking in multiplayer games are done, which methodologies are used in different sceneraios was requiered. To be able to get this knowledge, I did reading academic papers realated to these fields, building projects about these indivudally and combined.

After understanding and gaining more knowledge about how usual approaches used in physics engines and networking tools, I came up with an abstract requirements of the system and listed solutions to satisfy these requirments.

During the designing phase, I used the help of creating flowcharts, abstract topographies and diagrams for visualizing the system. They helped to decide needed components to achieve the system. I wrote features of these components and how to implement them and change them dyanmically throught out the implementation phase to keep everything together because system is includes a lot of components.

I used the help of agile software management methodologies to track my progress. I created a simple backlog and I created simple sprints to track my progress and make sure I was ready for the submission dates.

Before I started implemenations, I researched the technologies I can use for the system after coming up for some like using commercial engines like Unreal Engine or Unity, because they have good implementaitons both in physics engine, render pipeline, networking due to the reasons of source code of many components are not available for these tech stacks and I need to deal with a super big codebase, I decided to carry out implementing the necessary softwares using the NCLGL codebase. Because I was familiar with the codebase and I have the ability to change anything in the source code.

During the implementation, after implementing each component, code base got bigger and more hard to manage. It was a real challange to built the final product and managing the codebase.

Debugging process was another challange because each component needs to compiled, configed and runned.

In the end, every aspect of the implementation was a real challange and I think it is a spectacular case to work on.

5.2 Future Work

The research presented in this dissertation represents a significant step forward in the development of distributed physics simulation systems. By successfully demonstrating the feasibility of creating large-scale virtual environments using multiple physics servers, this work establishes a solid foundation for future advancements in the field. The following sections explore potential avenues for expanding upon and refining the research presented.

First thing to do the bringing this system one step further, algorithm that used for determining future positions and orientations of the physics objects simulated by physics server can be improved to be more accurate or replaced with the more accurate one. By doing this, smoother transactions between objects can be possible.

By developing a serialization system for game instance data across the system, level creation can be centralized, and it can be independent from the clients. Just by receiving data to instantiate in both clients and servers, the system can support multiple client and server types.

Currently, the system is not supporting clients to join a game instance that is already started. With the help of network packets this can be achieved. With this ability, clients can join a game instance that is already started, or they can rejoin a game instance whether they lost a connection or sign out for a period.

A created physics server instance that is simulating a part of a world that belongs to a game instance; can be stopped working reasons like connection loss, power outage for the machine that is running the program, or unhandled software exceptions. This will cause a system to not receive any data regarding the game objects on that server. A logic can be developed for servers to claim the area of a neighbor physics server instance which is down; Then it can simulate it for the consistency of the simulation. And to extend this ability more, physics servers can be added to a game instance which is started and be an addition to the physics engine power for simulating to that virtual world. For implementing this ability, system needs to redistribute the physics servers while the simulation is running and redistribute the game objects to the servers.

REFERENCES

[1] Blayn Parkinson, DigiFoE (2017). SpatialOS – Simutaling complex real world environments. [SpatialOS – Simulating complex real world environments | DigiFoE (wordpress.com)](https://digifoe.wordpress.com/2017/07/06/spartialos-simulating-complex-real-world-environments/). (accessed on 10.06.2024).

[2] Sayed Nasrollahi, Medium (2020). Real-time Physics Simulations and Machine Learning.<https://seyedhn.medium.com/real-time-physics-simulations-and-machine-learning-90e6e1ef1b3d>(accessed on 19.06.2024)

[3] Enet, <http://enet.bespin.org>. (accessed on 15.06.2024)

[4] Unterguggenberger, S., Kaufmann, H., & Bösch, J. (2022, May). The Road to Vulkan: Teaching Modern Low-Level APIs in Introductory Graphics Courses. [Proceedings of the 55th Hawaii International Conference on System Sciences](https://repositum.tuwien.at/handle/20.500.12708/175673) (pp. 1221-1230) (accessed on 17.06.2024). Institute of Electrical and Electronics Engineers (IEEE).Conference Name:ACM Woodstock conference

[5] Flanagan, D. (2012). **Computer Science: A Historical Perspective**. Pearson Education.

[6] Claypool, M., Claypool, K., & Damaa, F. (2006). The Effects of Frame Rate and Resolution on Users Playing First Person Shooter Games. In Proceedings ACM/SPIE Multimedia Computing and Networking (MMCN) Conference (pp. 959–968). San Jose, California, USA.Conference Short Name:WOODSTOCK’18

[7] Feldmann, A., Jain, M., & Chuang, W. (2000, October). Network Game Traffic Modelling: A Game Theoretic Approach. In Proceedings of the ACM SIGCOMM Conference on Internet Measurement (pp. 125-139). Association for Computing Machinery.

[8] Longshaw, Stephen & Turner, Martin & Finch, Emma & R, Gawthorpe. (2009). Physics Engine Based Parallelised Discrete Element Model.

[9] Roennau, Arne & Sutter, Florian & Heppner, Georg & Dillmann, Rüdiger & Oberländer, Jan. (2013). Evaluation of Physics Engines for Robotic Simulations with a Special Focus on the Dynamics of Walking Robots. 10.1109/ICAR.2013.6766527.

[10] Thalmann, D., & Thalmann, N. M. (2004). Game engine architecture. AK Peters/CRC Press.ISBN:978-1-4503-0000-0/18/06

[11] Rollings, A., & Adams, E. (2003). Andrew Rollings and Ernest Adams on game design. New Riders.

[12] Laganà, A., Casadei, G., & Palazzo, S. (2014). Game networking architecture: A survey. Computer Networks, 65, 222-243.

[13] Ericson, C. (2004). Real-time collision detection. Morgan Kaufmann.

[14] Richard Davison. Physics-Collision Response. Newcastle University

[15] Bridson, R. (2015). Fluid simulation for computer graphics. CRC press.

[16] Baraff, D. (1994). Fast contact force computation for nonpenetrating rigid bodies. In Proceedings of the 21st annual conference on Computer graphics and interactive techniques 1 (pp. 23-34). ACM.

[17] Guendelman, E., Bridson, R., Fedkiw, R., & Witkin, A. (2003). Non-convex rigid bodies with stacking. In Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation (pp. 151-158).

[18] Baraff, D. (1994). Fast contact force computation for nonpenetrating rigid bodies. In Proceedings of the 21st annual conference on Computer graphics and interactive techniques (pp. 23-34). ACM.

[19] Murray, D. (2016). Game physics engine development. O'Reilly Media, Inc.

[20] Richard Davison, (?). Physics-Linear Motion. Newcastle University

[21] Halliday, D., Resnick, R., & Walker, J. (2014). Fundamentals of physics. Wiley.

[22] Beer, F. P., Johnston, E. R., DeWolf, J. T., Mazur, D. J., & Russell, J. J. (2016). Vector mechanics for engineers: Dynamics. McGraw-Hill Education.Year:2018Date:June

[23] Brown, A. J. R. (2020). Distributed real-time physics for scalable and streamed games and simulations. PhD thesis, Newcastle University.

[24] Snowden, J. (2018). Network programming for game programmers. CRC Press.

Copyright Year:2018

Copyright Statement:rightsretained

DOI:10.1145/1234567890

RRH: F. Surname et al.

Price:$15.00