Research Article GU J Sci, Part A, 10(3): 310-326 (2023) 10.54287/gujsa.1338594

Gazi University
Journal of Science
PART A: ENGINEERING AND INNOVATION
http://dergipark.org.tr/gujsa

Edge Computing for Computer Games by Offloading Physics Computation

Fatih Mustafa KURT[1] Bahri Atay ÖZGÖVDE[1*]

[1]*Boğaziçi University, Department of Computer Engineering, İstanbul, Türkiye*

Keywords

Edge Gaming
Edge Computing
Physics Simulation
Task Offloading
Computer Games
Game Engine

Abstract

Realistic graphics and smooth experience in computer games come with the cost of increased computational requirements on the end-user devices. Emerging Cloud Gaming that enables executing the games on thin devices comes with its disadvantages such as susceptibility to network latency and the incurred cloud computing cost for the game service provider. The monolithic architecture of the game engines also presents an issue for cloud gaming where scaling efficiency in the cloud turns out to be limited. This paper proposes using edge computing principles to offload a subset of the local computations executed by games to a nearby edge server typically assigned for gaming applications. Specifically, we focus on physics computations since depending on the number of objects and their interactions modes this part may have considerable computational cost. In order to demonstrate the effectiveness of our approach we developed an edge gaming framework called Edge Physics Simulation (EPS) using the open-source game engine Bevy and the Rapier physics engine. We come up with an experiment setup in which a game scene with a high number of objects is executed using both standard local computation approach and using the proposed EPS method. In the experiments up to 8000 objects of varying shape complexities are employed to trigger significant computational load due to the collision detection process. Assessment metrics used are average physics computation time, resource consumption of local device and, the breakdown of the physics duration into its critical components such network time, simulation time and compression time. Our results show that EPS significantly reduces physics time compared to local execution. For the highest number of objects 75% reduction in physics computation time is reported where breakdown of physics time is further analyzed.

Cite

Kurt, F. M., & Özgövde, B. A. (2023). Edge Computing for Computer Games by Offloading Physics Computation. *GU J Sci, Part A, 10*(3), 310-326. doi:10.54287/gujsa.1338594

Author ID (ORCID Number)

0009-0005-0852-310X Fatih Mustafa KURT
0000-0001-9688-766X Bahri Atay ÖZGÖVDE

Article Process

Submission Date 11.08.2023
Revision Date 21.08.2023
Accepted Date 18.09.2023
Published Date 26.09.2023

1. INTRODUCTION

Today's game engines are complex pieces of software encompassing multiple components with unique functionalities to provide immersive gameplay close to real life. For example, the visual appearance of real life is resembled in the games by the rendering component, the audio component produces sounds of the environment and actions, and the physics laws are applied to the objects by the physics engine. Game engines achieve these immersive gameplays by producing continuous or discrete outputs and presenting them to the user, such as displaying frames from the game world, playing sound from the speakers, and vibrating the controller. In addition to producing these outputs, game engines are also required to perform these operations in a limited time. Depending on the exact game universe and mechanics implemented, these requirements can vary across game genres (Efe & Önal, 2020).

Immersive gameplay requires a smooth experience, which means producing image frames at a high rate. In order to accomplish this experience, both game software should be optimized, and the devices that run the games should be computationally powerful enough. This constraint puts pressure on the game engine, the game developer, and the hardware itself. Typically, the limiting factor in this situation becomes the end-user device.

*Corresponding Author, e-mail: ozgovde@boun.edu.tr

Newer computer games, despite efficiency measures taken by the developers, turn out to be resource hungry. Users with insufficient computational capacity, therefore, are refrained from executing their preferred games. This also creates a burden on the game software companies as they can address only a subset of their potential users.

Cloud Computing is a helpful method to let thin devices execute services that require high computing power. There are emerging solutions that provide gaming services under the concept of Cloud Gaming. Cloud gaming enables players to play games on thin devices by receiving the inputs from the device, executing the game in the cloud, and sending the output as a video stream back to the device. However, this method has certain drawbacks. First, the latency between the device and the server makes cloud gaming less applicable outside the advanced internet infrastructures. Secondly, it has a monolithic architecture where all the games run on only one machine and do not attempt to utilize end-user devices.

As a solution to the strict execution time requirements of services, edge computing can minimize the latency substantially compared to cloud computing (Cao et al., 2020). Edge computing allows low latencies between the server and the devices by locating powerful servers at the edge of the network in the proximity of end-users. Edge computing uses the computation offloading technique, which is the delegation of a computational task from the client machine to an edge-enabled server machine. Multiple factors affect the decision of which part of the game engine should be offloaded, such as computational requirements and memory accesses. Offloading lightweight tasks that do not require high computing power may not be worth its communication overhead. In this respect, computationally heavy tasks are good candidates for offloading. However, tasks unavoidably occur in the local software's call graph and may require interacting with other parts of the engine multiple times. Therefore, a component that accesses the core data of the engine frequently may not be a good candidate to offload. However, most physics engines have their context separated from the game context, making them feasible for computation offloading.

There is an immense literature on edge computing and its application domains. Edge computing principles have been successfully applied to a wide range of use cases ranging from IoT to autonomous driving (Cruz et al., 2022). Nevertheless, edge computing for computer games still needs to be explored to its full potential, as most studies focus on cloud gaming. Therefore, the challenges and opportunities of edge gaming require further exploration. This study specifically focuses on offloading the physics process in an edge computing setting. Distributed coordination, state management of game and physics engines, and communication and synchronization approaches are explored.

In this paper, we propose an edge gaming framework named Edge Physics Simulation (EPS) for computer games that involve heavy physics computation. We aim to relieve the computational burden on the end user device and delegate physics tasks to an edge server. To demonstrate the performance of our method, we implement an experiment setup that involves the open-source Bevy game engine and the open-source Rapier physics engine (Bevy, 2023; Rapier, 2023). In the proposed setting, the game engine runs on a thin user device, and the physics engine runs on the edge server. The results of the experiments show that the proposed solution reduces the average calculation time for the physics by up to 75% compared to the solution that executes physics on the thin device.

The contributions of the study can be summarized as:

- A novel gaming framework is proposed in which edge computing principles are applied to the computer games vertical.
- Gamescene is processed in a distributed fashion in which the game context and physics context take place on different hardware environments. An open-source implementation is provided based on Bevy game engine.
- A multi-faceted performance assessment is carried out via extensive experiments where the number of game objects and their shape complexities and over-the-network compression rates are varied.
- Considerable physics execution speed-up is reported, which indicates edge computing as a valid research direction for computer games.

The organization of this paper follows as: Section 2 summarizes a recent selection of the related works about edge gaming and cloud gaming alternatives. Then, Section 3 defines the proposed edge gaming framework explains the components in the solution architecture. In Section 4, the experimental setup and the properties of the testbed are focused on. Section 5 presents the results of the experiments and discusses them. Finally, Section 6 concludes the paper by summarizing the results and listing possible future works.

2. RELATED WORKS

The game engine is a crucial software component in game development and design. Although commercially well-known game engines exist such as Unity, Unreal, and CryEngine, the realm of game engines is much broader (CryEngine, 2023; Unity, 2023; Unreal, 2023). In a recent study, Vagavolu et al. (2021) present a dataset of open-source game engines where the authors analyze the software development activities of 526 game engine projects in the dataset. Similarly, Politowski et al. (2021) discuss the game engines from the perspective of a software framework in which authors compare the characteristics of 282 game engines, including a survey with 124 game engine developers.

Before developing an edge computing framework for computer games, the inner details of a typical game engine need to be analyzed to understand the overall technical challenges better. In this respect, some of the earlier works attempted to dissect the popular game engine Unity3D into its main modules, such as input, rendering, scripting, and physics engine; they analyze CPU consumptions of each module tested with different games with have different resource requirements (Messaoudi et al., 2015). As a common pattern, the rendering module is the most demanding module for the CPU resource. In addition to CPU consumption, the GPU consumption of the rendering's submodules is also analyzed. For a complete discussion of the technical perspectives on game engines, Gregory (2018) provides an excellent reference where all sub-systems of a typical game engine are focused individually and explored in depth.

A popular approach for augmenting the capabilities of end-user gaming devices is cloud gaming. Cloud gaming differs from the edge computing approach in that game software as a whole generally gets executed in the data center (Huang et al., 2014). Bhojan et al. (2020) propose an architecture for cloud gaming and report server resource consumption when number of players in the system varies. Chen et al. (2019) in which authors emphasize the distinction between cloud gaming and video-on-demand (VoD) applications and further propose an adaptive real-time streaming policy using the deep reinforcement learning tool that considers both the quality of service (QoS) and quality of experience (QoE) in cloud gaming scenarios.

A related but technically distinct technical approach for edge gaming is defined in 5G architecture via mobile edge computing (MEC) (Artuñedo Guillen et al., 2020). Nowak et al. (2021) provide a detailed survey on 5G-MEC use cases and a technical summary of research in computer games exploiting 5G to enhance the gamer experience. The authors provide a wide range of examples from the gaming sector and discuss the direction in which 5G can leverage gaming applications from various points. A specific example is reported by Cao et al. (2022) in which authors propose a heuristic algorithm to minimize QoE impairments under given constraints.

Since the rendering consumes most of the CPU resource, offloading the rendering part to a powerful server can help users play the games on resource-limited devices. Bulman and Garraghan (2020) propose a unified Graphics API that is mapped to OpenGL or Vulkan Graphics frameworks depending on the hardware capabilities of the user device (OpenGL, 2023; Vulkan, 2023). With this approach, unified API can offload some of the calls to the cloud when the performance of the device degrades. The authors show that this method can achieve 33% more frames per second when the commands are distributed over the cloud and device with a 50-50 ratio (Bulman & Garraghan, 2020).

Virtual reality (VR) and augmented reality (AR) technologies provide a novel opportunity to enhance computer game interaction modes for the end users. However, the constrained hardware devices involved in AR & VR scenarios necessitate edge computing for a smooth game experience. Nyamtiga et al. (2022) carried out a detailed empirical study on an experimental VR offloading testbed where the performance of the system is evaluated using three different VR games. The authors demonstrate the advantages and the tradeoffs involved in using the reduction in computational load and power consumption on the client device. Another study that focuses on the VR edge computing approach is described by Mehrabi et al. (2021), in which online heuristic

algorithms are proposed for the tradeoff between average video quality and delivery latency. The authors perform a series of simulation-based experiments for performance analysis and report a 22% improvement in video delivery and 8% in video quality.

Another offloading approach, proposed by Messaoudi et al. (2018), partitions the game scene into game objects such as the player character environment; then, in turn, it performs offloading of logic execution and rendering of these game objects to the server depending on the required device resources, data transfer time, and dependencies of the code. As a result of offloading, the server streams OpenGL ES commands back to the device to complete the rendering (Messaoudi et al., 2018).

A similar method for offloading computationally heavy tasks to the cloud for a soft body physics simulator is suggested and implemented by Danevičius et al. (2018) This method defines a task set that can be executed locally or offloaded to the cloud for the simulation program and partitions them as running on the local device or being offloaded to the cloud. This decision is made concerning multiple factors, such as the computation speed of the local device and cloud, the size of the inputs and outputs of the task, and network bandwidth and latency. An intelligent offloading management component uses these factors to decide if the task should be offloaded or not. Even though the study suggests offloading a program that runs a physics simulation, its goals heavily differ from our proposal's goals, such as separating the physics calculations from the game engine completely.

It is also possible to design all the game engine components modularly. The authors of SMASH propose a distributed game engine that is flexible enough to run the game entirely on the local or distribute its components over the network (Maggiorini et al., 2016). The architecture of SMASH resembles microkernels where the engine components interact internally by sending messages over a bus. A component with a SMASH-compatible interface can be added to the engine and can be used by other components. Another distributed game architecture is proposed by Mazzuca (2022) that implements a prototype for the rendering component. The implementation sends scene information to the rendering service over a UDP socket. The rendering service renders the scene and streams encoded video to the device (Mazzuca, 2022).

In order to apply edge computing principles to computer gaming scenarios, computational offloading should be implemented involving the game engine itself. This is not a straightforward task, as existing game engines are typically not designed with edge computing in mind. Our initial exploration with the open-source game engine Godot showed that distributing game engine functionality over a client-server model is cumbersome as the call graph of the overall code does not allow for minimizing communication overhead and latency (Godot, 2023). A suitable game engine for our purposes should have flexible and modular architecture, allowing a re-design with the minimum effort possible. When these requirements are considered, Bevy game engine turned out to be a valid choice for developing a game engine architecture compatible with edge computing.

Bevy promotes itself as a data-driven game engine that is open-source, free to use, and written in Rust programming language (Bevy, 2023; Rust, 2023). It has active development going on and is being developed by the contributions of its community and its members. Being easy to modify and easy to play with its core components are the reasons for choosing Bevy instead of other open-source game engines. The Rust programming language was also a compelling reason to use the Bevy. Rust presents itself as a system-programming language that aims to be both performant, reliable, and productive (Rust, 2023). As a benefit of ownership-based memory management and being memory-safe, developing edge physics was both fun and easy for a network-based application. Another reason for choosing Bevy is the Entity Component System (ECS) paradigm used to represent the objects in the game world.

3. SYSTEM DEFINITION

In order to apply edge computing principles in a gaming scenario, we propose an architecture that offloads physics computation, which is part of the typical local execution of game software. Figure 1 shows the main components of a game engine. Game engines may or may not have a default physics engine included in their software stack. Since the design and implementation of a physics engine is an expertise by itself, there is a variety of third-party physics engines that can be incorporated into the game engines (Bullet, 2023; Havok, 2023; PhysX, 2023). When an independent physics engine is employed within the game engine, there is a clear

separation of physics related computation procedures from the rest of the game engine software, where physics functionality is presented through a well-defined physics application programming interface (API).

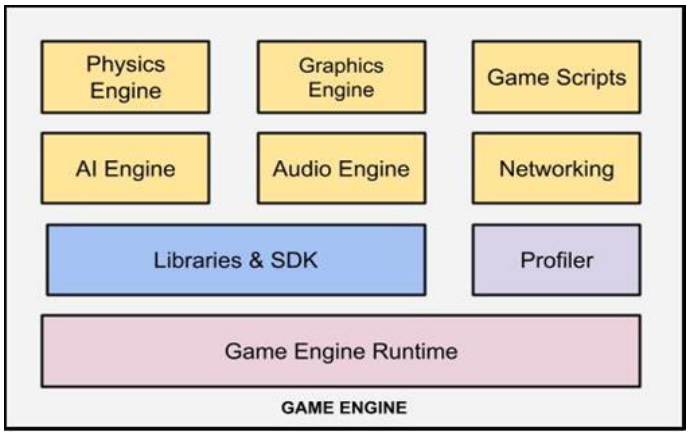


Figure 1. *Components and Modules of a Typical Game Engine*

In this respect, the proposed system architecture contains a distributed flow of game execution in which gamer device (local device) and an edge server (remote device) both take part via a computation offloading sequence. Figure 2 depicts the architecture of Edge Physics Simulation (EPS) framework. As dictated by the edge computing scenarios in general a fast local access networking technology is assumed to connect gamer devices to the edge server. Edge server is not dedicated to client, therefore multiple devices can connect to edge server to take offloading service.

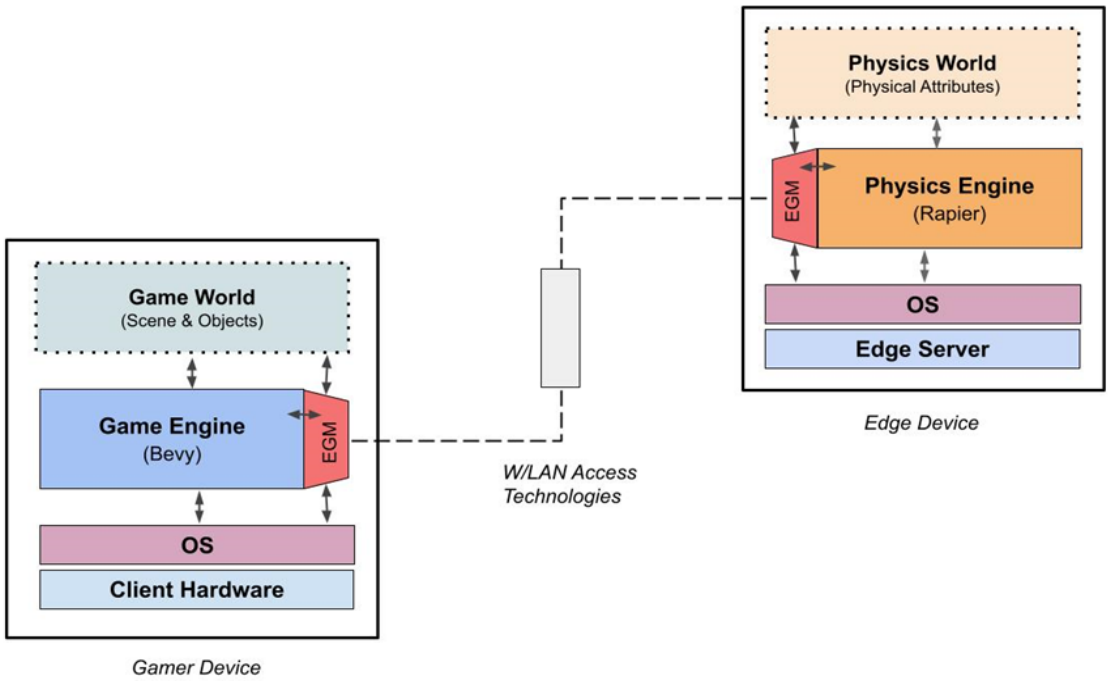


Figure 2. *Edge Physics Simulation (EPS) Architecture (EGM: Edge Gaming Manager)*

In our implementation, we used the Bevy game engine and Rapier physics engine as the core modules. The reasons behind our choice are: (i) both systems are open source and actively maintained by a decent community of developers, (ii) they do not impose any licensing fees, (iii) last but not least, the software architecture of these systems allowed flexible modification opportunities during the development and testing phases of our study. Edge Gaming Manager (EGM) is a special module developed in this study responsible for accessing internal data structures and attributes of the objects, supplying device connectivity for offloading and network payload size control using configurable compression techniques.

3.1. Bevy Game Engine

At its core, Bevy is built around a "game world" and a system scheduler. The game world is a jargon to denote the structure that holds the data that will be used within the game loop. This data includes entities, components of the entities, and resources such as a timer that tracks the past time intervals in the game, meshes to be used in the rendering and game state. On the other hand, the. On the other hand, the system scheduler schedules the systems in the ECS architecture and executes them on the game world. Here, the term "system" refers to a process that acts on "entities" with selected "components". For further technical details of the ECS, Hatledal et al. (2021) give definitions for ECS units and present a simulation framework implementation based on ECS software paradigm.

Systems can be grouped together by putting them into a System Set. Systems in the set can be chained for sequential execution, or the execution order can be left to the scheduler. In addition to chaining the systems, System Sets themselves can further be chained to define an execution order between them. If there is no order given at development time, the scheduler will try to execute systems concurrently to employ parallelism in a way that will avoid modifying a resource by two systems simultaneously. A simplified overview for the System Sets that are available in the Bevy by default and their execution order can be seen in Figure 3. From beginning to end, the execution of these sets represents the game loop

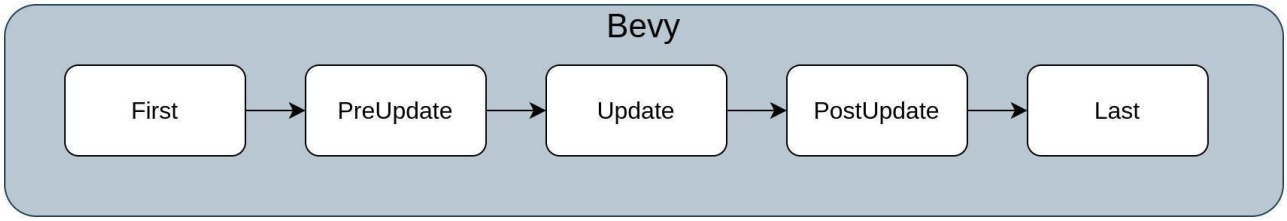


Figure 3. *Bevy's Default System Sets*

Aside from its core units, Bevy provides additional plugins to implement other features, such as rendering and audio. Each plugin modifies the game world by inserting resources and instructs the scheduler on executing its systems. For example, the input plugin schedules its systems to be run at the PreUpdate set in order to let other systems that will be run at the Update set read the user inputs. Plugin-based architecture makes Bevy a modular game engine such that a plugin can be swapped with another implementation of the feature, assuming it provides a compatible interface to the plugins that depend on it. Even the core without any additional plugins is a valuable tool for developing an application that requires a scheduler.

3.2. Rapier Physics Engine

Although Bevy implements fundamental components of a game engine as plugins, it does not have a physics engine that is available out of the box. However, there are certain physics plugins that integrate independently developed physics engines into the Bevy. Bevy Rapier is such a plugin that enables the integration of the Rapier physics engine into the Bevy game engine without much effort (Bevy Rapier, 2023).

As a standalone physics engine, Rapier contains its own world (context) to perform physics simulations. To integrate Rapier into Bevy, the Bevy Rapier plugin defines three main system sets that perform (i) syncing Bevy's game world to Rapier physics world, (ii) simulating the physics for one step, and (iii) transferring the changed values in the physics world to the Bevy's original game world. Here, the word "world" means the context, and Bevy's plugin software interface stores Rapier's context in the game world as a resource. The integration of Bevy's System Sets and the plugin's System Sets can be seen in Figure 4.

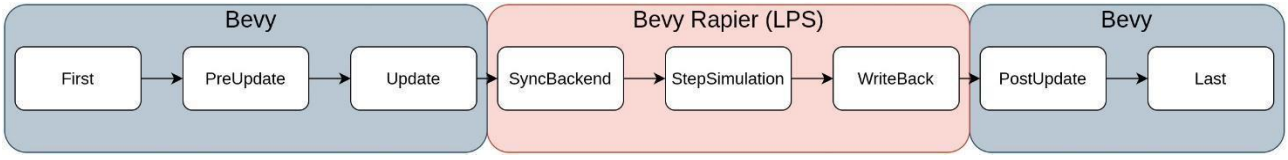


Figure 4. *Integration of Bevy Rapier Plugin's System Sets*

3.3. Proposed Solution

Standard installation and execution procedures let the Rapier plugin, which is named Local Physics Simulation (LPS), execute everything on the local device. This causes heavy physics simulations to become the performance bottleneck of the game loop and results in increased frame times. As a remedy, we propose Edge Physics Simulation (EPS), which offloads physics-based computation to an edge server.

The idea originates from the existence of the two different worlds (contexts) that belong to Rapier and Bevy that are synchronized two times in each frame. This approach has already reduced the number of interactions between the two worlds to a minimum and enabled the separation of these two execution tracks on different computational environments.

In the EPS architecture, a separate controller named Edge Gaming Manager (EGM) separate from the original game, is developed and deployed to an edge server. In the edge server, both the simulation step and the context are moved into EGM. At the startup, EGM listens on a TCP port and waits for a new connection. For the game, it tries to connect to the EGM when it is started. This is achieved by a peer EGM instance on the user device, as shown in Figure 2. When a connection from the game is established, EGM reads the synchronization data over the connection, simulates the physics for one step using rapier functionality, and sends the new positions and the rotations of the objects to the game. These three steps are executed in a loop until the underlying connection gets broken or the game sends a shutdown message. In the game, the execution order of the Bevy Rapier's System Sets is modified to minimize the time that is taken to receive the response. For example, while Edge is performing the StepSimulation step, Client also continues the execution of the PostUpdate and the Last sets. A visual representation of the architecture is shown in Figure 5.

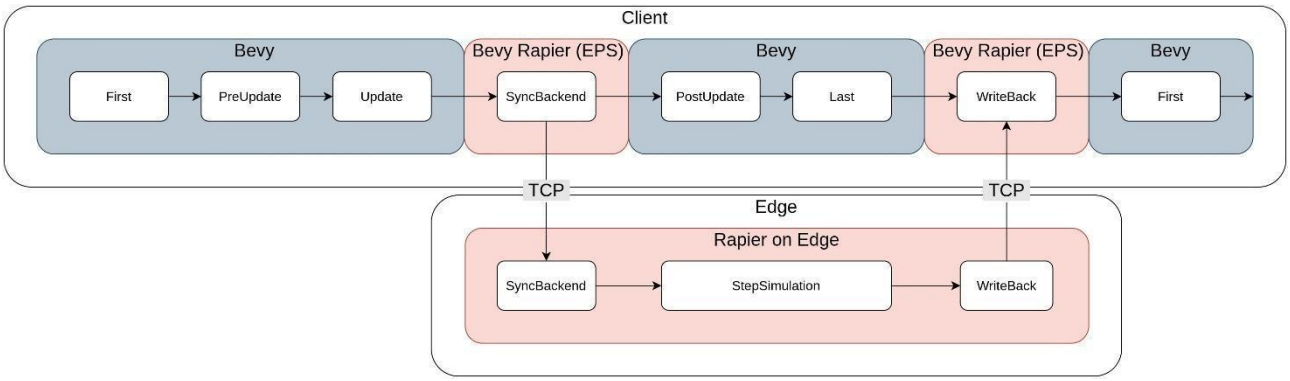


Figure 5. *Proposed Solution's Architecture*

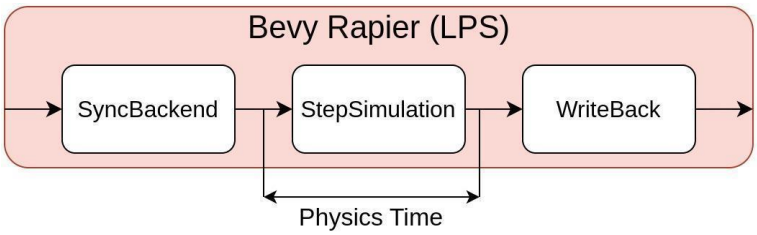
The EPS should provide smooth gameplay even if there is no demanding physics calculations from the game. Moreover, it is also essential to utilize the underlying network connection efficiently. For this reason, reducing the time that is spent in the network should be minimized. EPS uses DEFLATE compression algorithm in order to reduce the size of the data sent over the network (Deutsch, 1996). DEFLATE employs different compression levels that range from 1 to 9 and inform the algorithm to favor faster compression times or smaller compressed sizes, respectively. When applicable, it also disables the buffering algorithm applied to TCP socket by the operating systems, such as Nagle's algorithm, to minimize the latency (Nagle, 1984).

4. EXPERIMENTS

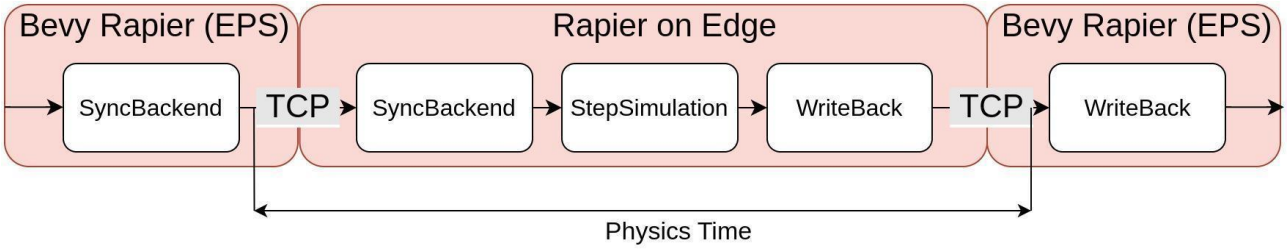
An experiment setup is designed to enable performance evaluation of the two different approaches in a comparative manner. One solution is the Edge Physics Simulation (EPS), in which the client offloads its physics simulation to the edge server for each frame. Another solution is the Local Physics Simulation (LPS), in which all the simulation is executed on the client. For EPS solution, the experiments are conducted on one edge machine and one client machine, whereas LPS contains only the client machine. Details of the implementation platform will be discussed in the following subsection.

During the experiments, the rendering component of the game is disabled due to the client's weak GPU. When the rendering is enabled, the frame time increases and starts fluctuating. Therefore, instead of measuring the

Frames per Second (FPS) for comparison, the physics time taken to perform the StepSimulation step is measured and compared for each solution. For the EPS solution, physics time also contains data transmission time over the network and StepSimulation step. Figure 6 visualizes the physics time measurement for each solution..



(a)



(b)

Figure 6. a) *LPS (Local Physics Simulation) Physics Time,* ***b)*** *EPS (Edge Physics Simulation) Physics Time*

The experiments are based on a scenario that has a multitude of physical objects stacked up on each other to form a grid pattern in a confined space. In this setting, a ball with high velocity initially triggers the objects, and objects fall due to gravity, and considerable inter-object collision takes place. Elasticity and coefficient of friction values are adjusted to maintain the continuous movement of objects. Figure 7 and Figure 8 visually depict snapshots from the experiments in which Figure 7 shows the initial condition.

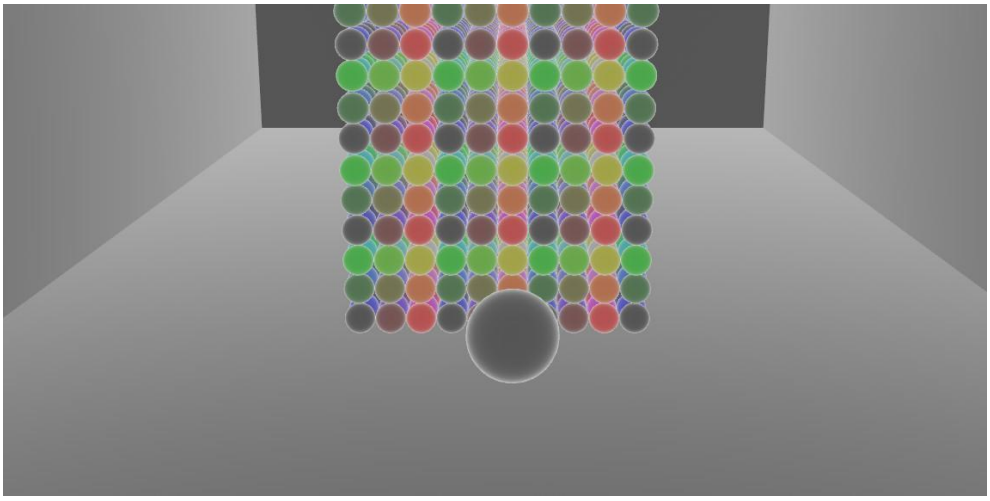


Figure 7. *Appearance of the Scene at Initial State*

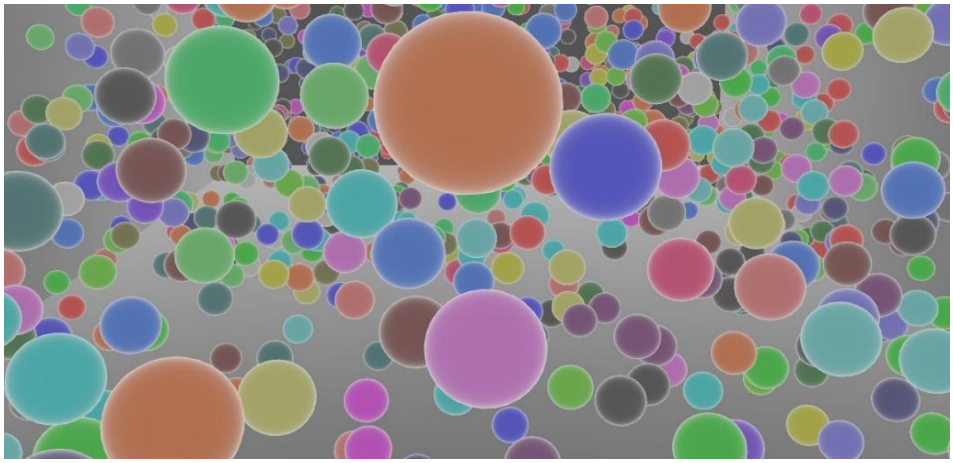


Figure 8. *Appearance of the Scene at the Steady State*

Parameters for each experiment run are defined in terms of the:

- number of objects used for the simulation,
- shape of the objects,
- whether edge gaming is enabled or not (i.e. EPS vs. LPS),
- compression level used in the communication, and
- whether Continuous Collision Detection (CCD) is enabled or not.

CCD is a special collision detection algorithm that considers the positions of the objects between frames. CCD provides much more accurate collision detection behavior, especially when objects are small and have high velocity. However, this comes with an extra computational cost; therefore, game programmers need to switch this feature on only when necessary. Table 1 summarizes the values used in the experiments for the relevant parameters. Visual appearances of the shapes are shown in Figure 9. These parameters affect the computational load taking place in the game scene and allow us to model different game profiles requiring different levels of physics computations. Each experiment run gets executed for 15 seconds. After an exploratory phase of the experiments, it was seen that this duration was sufficient for the system to reach a steady state. Please note that when the local execution is used, there is no compression option since no data is sent over the network.

Table 1. *Experiment Parameters*

Parameter	Values
Solution Approach	Local, Edge
Number of Objects	500, 1000, 2000, 4000, 8000
Collision Detection Algorithm	Discrete CD, Continuous CD
Compression Levels	No compression, Level 1, Level 3
Object Shapes	ball, capsule, cuboid, complex

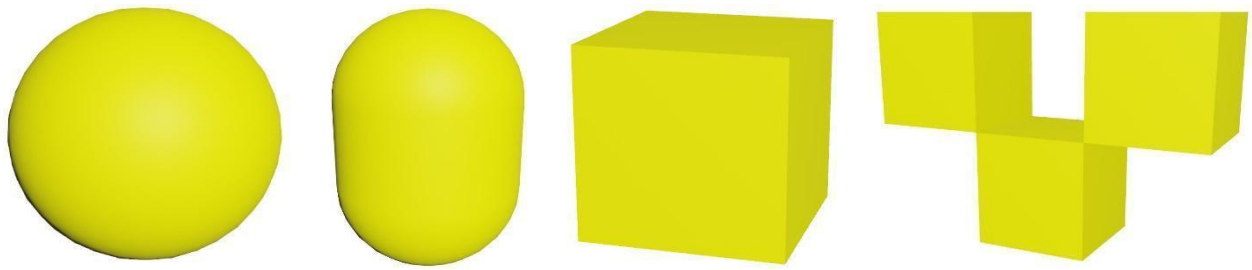


Figure 9. *Visual Appearances of Shapes* ***a)*** *Ball* ***b)*** *Capsule* ***c)*** *Cuboid* ***d)*** *Complex*

There are also other parameters belonging to the physics engine that affect the experiment's behavior and computational load. However, values for these parameters are kept constant for all the experiments. Table 2 summarizes these parameters and the values chosen for them.

Table 2. *Constant Experiment Parameters*

Parameter	Values
Density of Object	0.477 kg/cm^3
Elasticity Coefficient of Object	1.1
Friction Force	0.0 N
Gravity	9.81 m/s^2

When the combinations of different values for each option are considered, there are 160 different configurations to be tested. Moreover, each configuration is tested 2 times to check if they produce similar results. Displaying each possible configuration in this paper is impossible due to space considerations. Instead, a subset of the configurations that gives a clear understanding of the general behavior of the solutions is shown in Section 5.

4.1. Implementation Platform

To implement the experiment platform, two different machines are used as an edge server and a client device. For the EPS solution, these machines are connected over a Local Area Network (LAN) with 100 Mbit ethernet connections using a modem. Figure 10 presents a diagram that describes the platform visually.

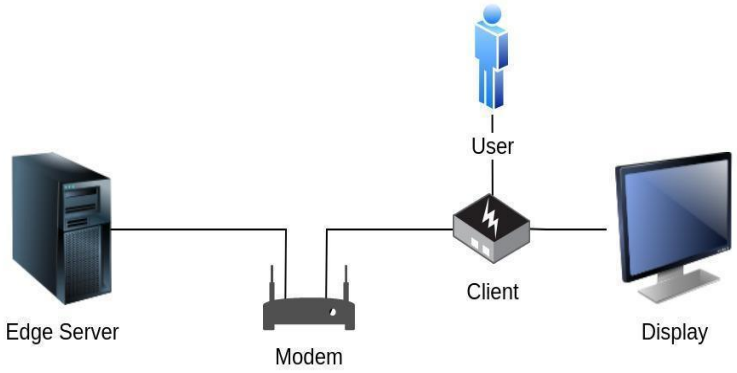


Figure 10. *Diagram of the platform used for the implementation*

The flow in this diagram for generating one frame can be explained as: The client accepts input from the user, processes it, and sends updated components of the entities to the Edge Server. After completion of the physics calculation, the client receives the latest state of the entities from the edge server and outputs the result to the Display. In this platform, the Edge Server is a PC that contains powerful hardware compared to the client device. On the other hand, the Raspberry Pi 4B device is used as the client. The hardware and software configurations of the machines are summarized in Table 3.

Table 3. *Machine Configurations*

Machine	**Hardware**			**Software**	
	CPU	**Memory**	**Connection**	**OS-Arch-Distribution**	**libc**
Client	4 core Cortex-A72@1.8 GHz	4GB LPDDR4@3200 MHz	100 Mbit Ethernet	Linux - aarch64 - Arch Linux	Glibc 2.35
Edge	6 core AMD Ryzen 5600X@3.7 GHz	32GB DDR4@3200 MHz	100 Mbit Ethernet	Linux - x86_64 - Gentoo	Glibc 2.37

4.2. Data Collection from Experiments

Evaluating and analyzing the experiments requires collecting related information from the application and its environment during the experiments. For the application part, some diagnostics are placed into the source code of both the client application and the physics server. These diagnostics collect data from the application state for every frame. Logs are collected and stored inside a file by another application specifically developed for this case. The diagnostics that include information from the physics server are collected in the physics server and sent back to the client for each frame. A summary of the data collected and processed during the experiments is depicted in Table 4. Other than the diagnostics data collected, the CPU usage data is also collected by another process.

Table 4. *Edge Gaming Experiment Data Summary*

Diagnostic	**Description**
Time	Time passed since application startup
Frame Count	Number of frames generated since startup
Frame Time	Time taken to generate current frame
Physics Time	Time taken to complete physics computation
Network Time	Time spent in the network communication
Sent Bytes	Number of bytes sent to edge server
Sent Compressed Bytes	Number of actual bytes that are sent over the network. For **No Compression** case, this is equal to **Sent Bytes**
Received Bytes	Number of bytes expected to be received from edge server
Received Compressed Bytes	Number of actual bytes that are received over the network. For **No Compression** case, this is equal to **Received Bytes**
Compression Time in Client	Time taken to compress payload on the client
Decompression Time in Client	Time taken to decompress payload on the client
Compression Time in Server	Time taken to compress payload on the server
Decompression Time in Server	Time taken to decompress payload on the server

5. RESULTS AND DISCUSSION

The two solutions discussed, namely EPS and LPS, are investigated concerning two different criteria: execution performance and resource consumption. The execution performances of the solutions are identified by their physics time durations, whereas the resource consumptions are identified by their CPU usage. For EPS solution, bandwidth usage of the network is also considered to be a metric for resource consumption.

5.1. Physics Time

In order to obtain a general overview, the performances of EPS and LPS are compared with respect to their physics execution time under different configurations. In the first set of experiment runs, the effect of the number of objects on the scene is explored. In order to underline the physics computation, the object shape is chosen as "complex" so that extra collision geometry is involved. For other experiment parameters, CCD (Continuous Collision Detection) mode is switched off, and for the EPS part, the compression level is set to 1. Figure 11 shows the performances of LPS and EPS comparatively. The performance gap increases for a larger number of objects in the scene in favor of the edge-based solution.

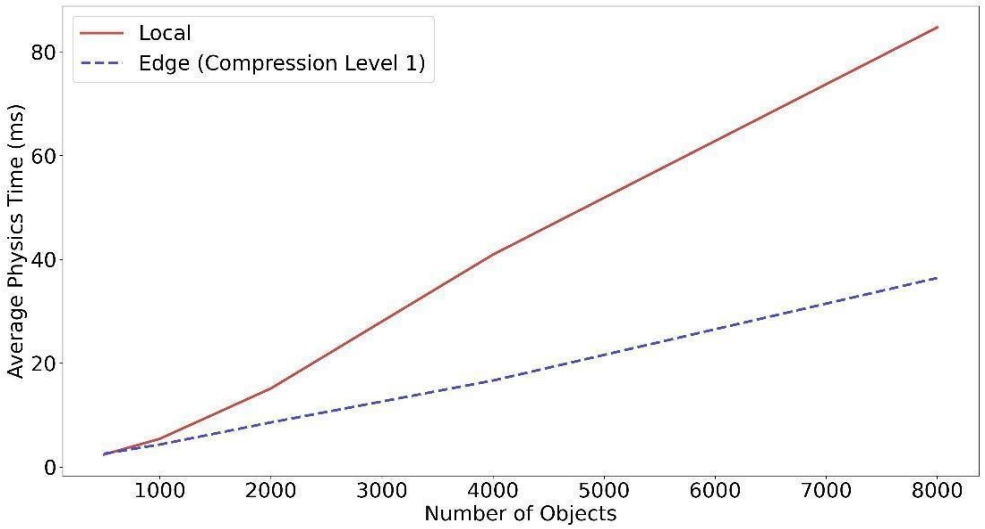


Figure 11. *Average Physics Time for solutions, shape = complex, CCD = off*

Apart from reporting the mean physics time values, it is also meaningful to see how physics time in every frame evolves during the experiments. In Figure 12, the characteristics of the scenario can be seen. Initially, physics time is small since all the objects are idle. When the ball on the ground hits the objects, a spike in physics time is observed. After a small amount of time from the collision, the physics time converges to a steady state with small variations over time. This convergence is because the number of collisions that happen in one step becomes steady due to gas molecules like the perpetual movement of the objects. As expected from Figure 11, Figure 12 reveals that edge-based proposed solution (EPS) outperforms local computing (LPS) at all times.

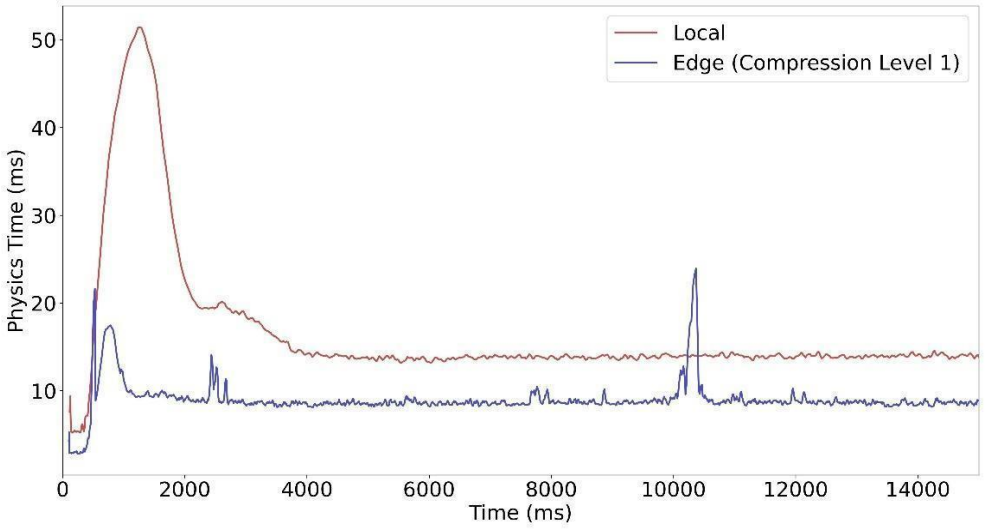


Figure 12. *Time series visualization of physics computation time. Each data point belongs to successive frames over time. (Objects=2000, Shape Type = Complex, CCD=off)*

To explore the EPS model in more detail, the extra computational load incurred by collision detection modality is experimented by turning CCD mode on and off. These two configurations illustrate the characteristics of the solution under different computational requirements for the physics calculations. Figure 13 clearly shows the effect of the CCD where turning on the CCD mode changes the linear behavior of the solution under different number of objects. The non-linearity of Average Physics Time where CCD mode is turned on indicates that the behavior of the EPS solution is mostly dependent on the physics calculation performed under the hood.

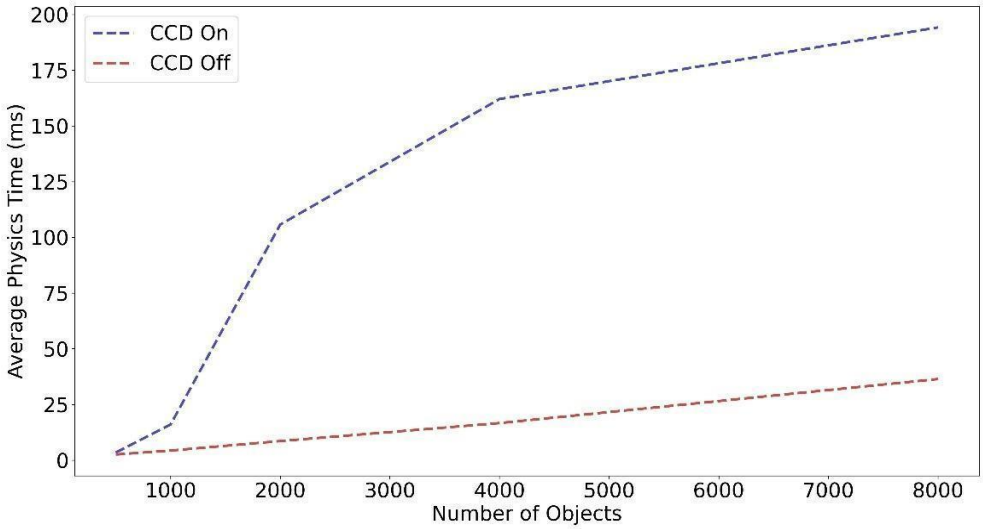


Figure 13. *Average Physics Time for EPS where CCD is toggled for the configurations*

The performance of edge computing approaches is determined not only by the computation time on the server but also by the time spent over the network. In this respect, our proposed solution employs data compression techniques further to minimize the data context transfer time for physics offloading. Compression, however, comes with a new trade-off where higher compression levels result in lower data size at the expense of increased computation overhead. The compression algorithm used in the experiments allows for ten different compression levels where level=1 has the fastest computation time, and level=10 has the maximum compression rate (Deutsch, 1996).

To explore the trade-off brought by compression, a series of experiments is executed with different compression rates for over-the-network transfer. Figure 14 depicts how local solution (LPS) compares to edge solution (EPS) with different compression rates. For a clear visual representation, the figure presents normalized values for physics time. For a very low number of game objects, edge computing turns out to be not effective. However, as the number of game objects increased, all variants of edge computing solutions outperformed local computation based approach. It is apparent that compression enhances performance. The best-performing compression rate occurs at level=1, meaning that extra time required for more complex compression does not pay off. Results belonging to all possible compression configurations are not included in Figure 14 to avoid a cluttered view where compression levels 1 and 3 are sufficient to understand the general trend.

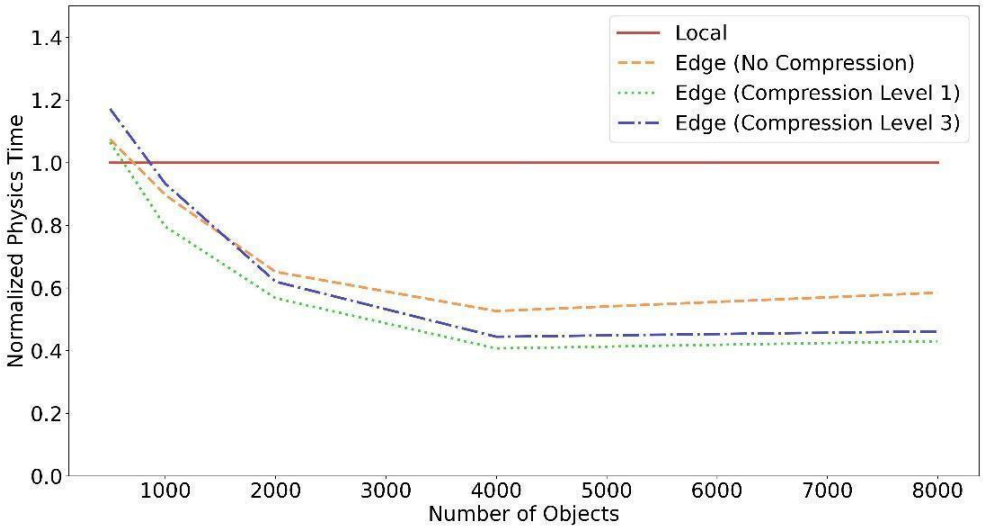


Figure 14. *Normalized physics computation time to depict the effect of compression rates. (Local physics computation is normalized to 1)*

To better understand the reported physics time for the edge-based solution, Figure 15 shows the breakdown of the overall physics time in terms of network transfer time, physics simulation time, and data compression time. Timings are reported for one physics simulation step of EPS where level 1 compression and complex shape are used, as CCD is turned off. The figure shows that network time takes most of the physics time at the lower number of objects. This network overhead can be eliminated using higher-speed connections such as gigabit ethernet. As the number of objects increases, the simulation and compression times increase to a point where network time takes less than half of the physics time.

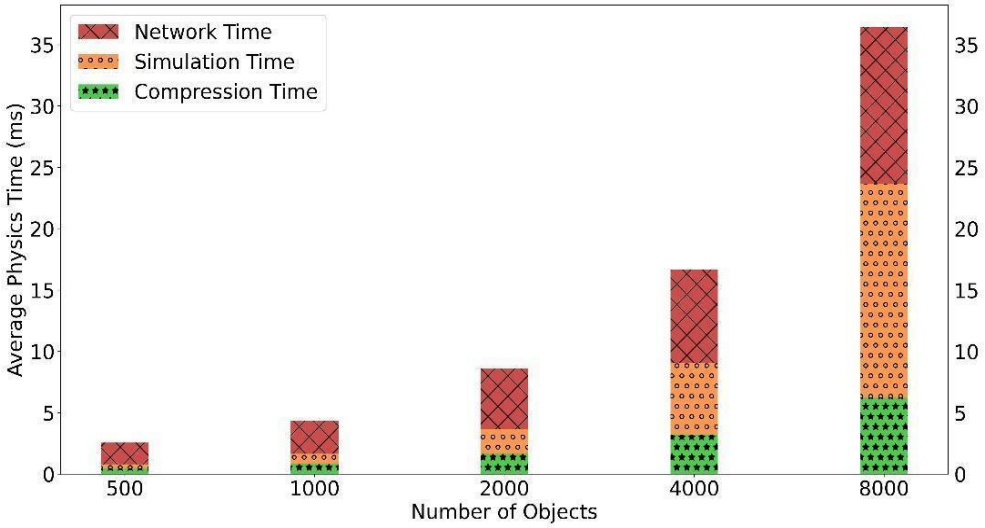


Figure 15. *Breakdown of Physics Time, shape = complex, CCD = off*

5.2. Resource Consumption

In order to compare the resource consumption of the edge and local computation-based solutions, CPU utilization statistics of the game execution processes are recorded. Note that LPS (Local Physics Simulation) involves the execution of both the Bevy game engine and the Rapier physics engine as a bundle. In contrast, EPS (Edge Physics Simulation) includes our proposed modified setup for offloading physics computation.

Figure 16 shows that the edge-based solution lowers the CPU utilization of the client considerably. This observation parallels the general edge computing paradigm, where applications can be run on low-end devices. As the number of objects increases, the usage of the EPS decreases. This behavior is because the time spent on the edge server and on the I/O increases as the number of objects increases. This blocks the client and keeps it idle most of the time.

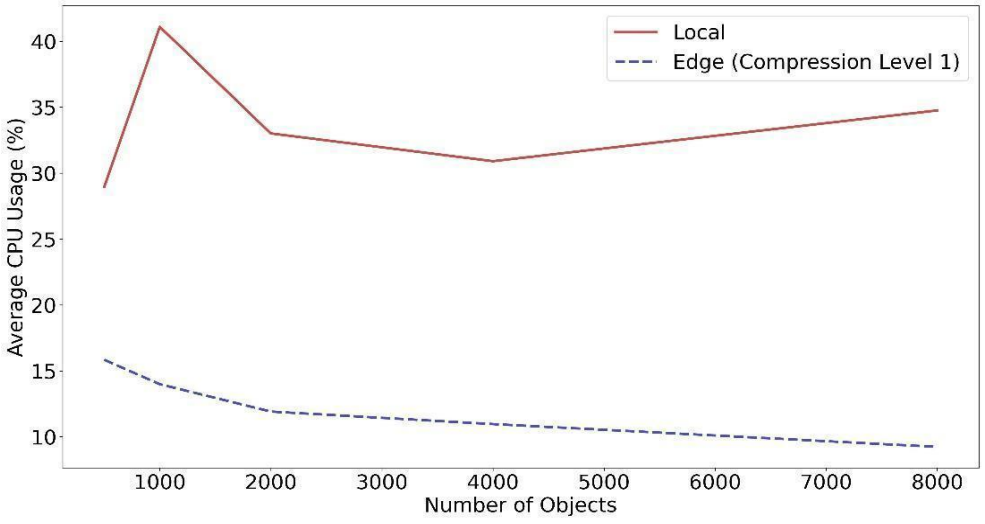


Figure 16. *Average CPU Usage for solutions, shape = complex, CCD = off*

One could imagine that CPU usage for the LPS solution should be similar across the experiments. On the contrary, Figure 16 shows that configurations for *500* and *1000* objects have very different CPU usage. The usages only become similar after the *2000* number of objects. This shows that Rapier's physics engine employs parallelism in its implementation. However, this implementation hits scaling issues under different configurations.

In Figure 17, there are three different metrics for the bandwidth usage. The first, *Uplink*, represents the data transmitted from the client to the edge, whereas the second, *Downlink*, represents the data received from the edge. Since the objects are spawned at the beginning of the experiment, the client uses the Uplink only at that time. Throughout the experiment, Uplink stays close to *0*. For the Downlink, as seen in Figure 12, it becomes steady over time except for some significant drops that also affect the frame time.

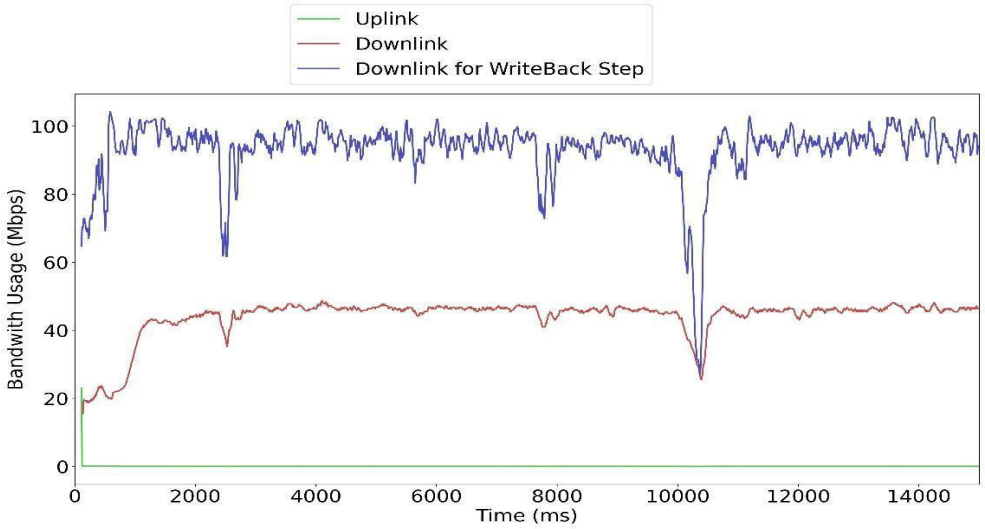


Figure 17. *Bandwidth Usage over time, number of objects = 2000, shape = complex, CCD = off*

The experiments are executed in a network environment with a 100Mbps line rate. Downlink figures stay much below this because communication between two machines does not occur during the whole frame. We include one additional metric, *Downlink for WriteBack Step*, as a correction factor. This metric takes into account the time that is passed in the *WriteBack* step. This clearly shows that the connection speed limit is reached during this step.

After performing multiple comparisons between two solutions under different configurations, it can be said that the proposed solution EPS provides substantially shorter physics simulation time compared to default LPS and reduces CPU usage when a weak client and a powerful edge server are used. The performance of EPS can also be improved further by increasing the bandwidth of the network, such as employing a higher-speed connection.

6. CONCLUSION

Computer games belong to the group of the most resource-demanding software in the modern era. Gamers continuously face a hardware challenge as newer games typically require more computational capacity. As a remedy, this study proposes an edge gaming architecture called Edge Physics Simulation (EPS) that offloads physics computation tasks to the edge server. EPS further enhances task offloading operation by incorporating a compression mechanism that can be tuned. To demonstrate the performance of EPS, an edge gaming setup is implemented by employing an open-source Bevy game engine and Rapier physics engine. A series of experiments are conducted covering various system parameters, including the number of objects on the game scene, the complexity of the object shapes, collusion detection algorithm type, and the level of compression. By looking at the results of the experiments, it is shown that the edge computing based implementation can speed up the physics calculation up to 75% compared to its local-only counterpart. The speed-up figures reported are significant and show the feasibility of edge computing for computer games. As such, EPS will enable users to play their games on hardware with relatively low resources.

As future work, we aim to focus on other critical components of a typical game engine that can potentially be distributed over the network, such as rendering and AI modules. This will enable a full-fledged edge gaming setup in which low-resource devices are able to execute games with high computational requirements. Also, we plan to examine the effect of client hardware resources on the overall system performance. This will enable us to model different user equipment profiles, such as mobile phones, tablets, and older generation PCs.

CONFLICT OF INTEREST

The authors declare no conflict of interest.

REFERENCES

Bevy (2023). A refreshingly simple data-driven game engine built in Rust, Free and Open Source Forever. (Accessed:15/06/2023) URL

Bevy Rapier (2023). Official Rapier plugin for the Bevy game engine. (Accessed:15/06/2023) URL

Bhojan, A., Ng, S. P., Ng, J., & Ooi, W. T. (2020). CloudyGame: Enabling cloud gaming on the edge with dynamic asset streaming and shared game instances. *Multimedia Tools and Applications*, *79*(43-44), 32503-32523. doi:10.1007/s11042-020-09612-z

Bullet (2023). Real-time collision detection and multi-physics simulation for VR, games, visual effects, robotics, machine learning. (Accessed:10/08/2023) URL

Bulman, J., & Garraghan, P. (2020). *A Cloud Gaming Framework for Dynamic Graphical Rendering Towards Achieving Distributed Game Engines*. In: Proceedings of the 12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20), Virtual Event.

Cao, K., Liu Y., Meng G. & Sun, Q. (2020). An Overview on Edge Computing Research. *IEEE Access*, *8*, 85714-85728. doi:10.1109/ACCESS.2020.2991734

Cao, T., Jin, Y., Hu, X., Zhang, S., Qian, Z., Ye, B., & Lu, S. (2022). Adaptive provisioning for mobile cloud gaming at edges. *Computer Networks*, *205*, 108704. doi:10.1016/j.comnet.2021.108704

Chen, H., Zhang, X., Xu, Y., Ren, J., Fan, J., Ma, Z., & Zhang, W. (2019). T-Gaming: A Cost-Efficient Cloud Gaming System at Scale. *IEEE Transactions on Parallel and Distributed Systems*, *30*(12), 2849-2865. doi:10.1109/TPDS.2019.2922205

Cruz, P., Achir, N., & Viana, A. C. (2022). On the Edge of the Deployment: A Survey on Multi-access Edge Computing. *ACM Computing Surveys*, *55*(5), 1-34. doi:10.1145/3529758

CryEngine (2023). The complete solution for next generation game development by Crytek. (Accessed:10/08/2023) URL

Danevičius, E., Maskeliūnas, R., Damaševičius, R., Połap, D., & Woźniak, M (2018). A Soft Body Physics Simulator with Computational Offloading to the Cloud. *Information*, *9*(12), 318. doi:10.3390/info9120318

Deutsch, P. (1996, May). DEFLATE Compressed Data Format Specification version 1.3. Network Working Group. (Accessed:10/08/2023) URL

Efe, A., & Önal, E. (2020). ONLINE Game Security: A Case Study of an MMO Strategy Game. *Gazi University Journal of Science Part A: Engineering and Innovation*, *7*(2), 43-57.

Godot (2023). Free and open source 2D and 3D game engine. (Accessed:31/07/2023) URL

Gregory, J. (2018). *Game Engine Architecture*. CRC Press.

Artuñedo Guillen, D., Sayadi, B., Bisson, P., Wary, J. P., Lonsethagen, H., Antón, C., de la Oliva, A., Kaloxylos, A., & Frascolla, V. (2020). *Edge computing for 5G networks* - white paper. Zenodo. doi:10.5281/zenodo.3698117

Hatledal, L. I., Chu, Y., Styve, A., & Zhang, H. (2021). Vico: An entity-component-system based co-simulation framework. *Simulation Modelling Practice and Theory*, *108*, 102243. doi:10.1016/j.simpat.2020.102243

Havok (2023). Havok Physics, Make game worlds real. (Accessed:10/08/2023) URL

Huang, C.-Y., Chen, K.-T., Chen, D.-Y., Hsu, H.-J., & Hsu, C.-H. (2014). GamingAnywhere: The first open source cloud gaming system. *ACM Transactions on Multimedia Computing, Communications, and Applications*, *10*(1s), 10. doi:10.1145/2537855

Maggiorini, D., Ripamonti, L. A., Zanon, E., Bujari, A., & Palazzi, C. E. (2016). *SMASH: A distributed game engine architecture*. In: IEEE Symposium on Computers and Communication (ISCC 2016), (pp. 196-201), Messina.

Mazzuca, L. (2022). *Distributed Cloud Gaming Pipeline.* MSc Thesis, Universidad Complutense de Madrid.

Mehrabi, A, Siekkinen, M., Kämäräinen, T., & yl¨-J¨¨ski, A. (2021). Multi-Tier CloudVR: Leveraging Edge Computing in Remote Rendered Virtual Reality. *TrACM Transactions on Multimedia Computing, Communications, and Applications*, *17*(2), 49. doi:10.1145/3429441

Messaoudi, F., Ksentini, A., & Simon, G. (2015). *Dissecting games engines: The case of Unity3D.* In: International Workshop on Network and Systems Support for Games (NetGames 2015), (pp. 1-6), Zagreb.

Messaoudi, F., Ksentini, A., & Bertin, P. (2018). *Toward a Mobile Gaming Based-Computation Offloading.* In: IEEE International Conference on Communications (ICC 2018), (pp. 1-7), Kansas City.

Nagle, J. (1984). Congestion Control in IP/TCP Internetworks. Network Working Group. (Accessed:10/08/2023) URL

Nowak, T. W., Sepczuk, M., Kotulski, Z., Niewolski, W., Artych, R., Bocianiak, K., Osko, T., & Wary, J.-P. (2021). Verticals in 5G MEC-Use Cases and Security Challenges. *IEEE Access*, *9*, 87251-87298. doi:10.1109/ACCESS.2021.3088374

Nyamtiga, B. W., Hermawan, A. A., Luckyarno, Y. F., Kim, T.-W., Jung, D.-Y., Kwak, J. S., & Yun, J.-H. (2022). Edge-Computing-Assisted Virtual Reality Computation Offloading: An Empirical Study. *IEEE Access*, *10*, 95892-95907. doi:10.1109/ACCESS.2022.3205120

OpenGL (2023). The Industry's Foundation for High Performance Graphics. (Accessed:15/06/2023) URL

Politowski, C., Petrillo, F., Montandon, J. E., Valente, M. T., & Guéhéneuc, Y.-G. (2021). Are game engines software frameworks? A three-perspective study. *Journal of Systems and Software*, *171*, 110846. doi:10.1016/j.jss.2020.110846

Physx (2023). NVIDIA PhysX® is an open source, scalable, multi-platform physics simulation solution supporting a wide range of devices, from smartphones to high-end multicore CPUs and GPUs. (Accessed:10/08/2023) URL

Rapier (2023). Fast 2D and 3D physics engine for the Rust programming language. (Accessed:15/06/2023) URL

Rust (2023). A language empowering everyone to build reliable and efficient software. (Accessed:15/06/2023) URL

Unity (2023). Unity Real-Time Development Platform 3D, 2D, VR & AR Engine. (Accessed:10/08/2023) URL

Unreal (2023). The world’s most open and advanced real-time 3D creation tool. (Accessed:10/08/2023) URL

Vagavolu, D., Agrahari, V., Chimalakonda, S., & Venigalla, A., S., M. (2021). *GE526: A Dataset of Open-Source Game Engines.* In: IEEE/ACM 18th International Conference on Mining Software Repositories (MSR 2021), (pp. 605-609), Madrid.

Vulkan (2023). Cross platform 3D Graphics. (Accessed:15/06/2023) URL