Aura Projection for Scalable Real-Time Physics

Alexander Brown

Department of Computer Science
Newcastle University

This dissertation is submitted for the degree of
Doctor of Philosophy

August 2019

I would like to dedicate this thesis to my loving parents …

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 80,000 words including appendices, bibliography, footnotes, tables and equations.

Alexander Brown
August 2019

Acknowledgements

And I would like to acknowledge ...

Abstract

In this paper we propose a solution to delivering scalable real-time physics simulations. Although high performance computing simulations of physics related problems do exist, these are not real-time and do not model the real-time intricate interactions of rigid bodies for visual effect common in video games (favouring accuracy over real-time). As such, this paper presents the first approach to real-time delivery of scalable, commercial grade, video game quality physics. This is achieved by taking the physics engine out of the player's machine and deploying it across standard cloud based infrastructures. The simulation world is then divided into sections that are then allocated to servers. A server maintains the physics for all simulated objects in its section. Our contribution is the ability to maintain a scalable simulation by allowing object interaction across section boundaries using predictive migration techniques. We allow each object to project an aura that is used to determine object migration across servers to ensure seamless physics interactions between objects. The validity of our results is demonstrated through experimentation and benchmarking. Our approach allows player interaction at any point in real-time (influencing the simulation) in the same manner as any video game. We believe that this is the first successful demonstration of scalable real-time physics.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Video games rely on real-time physics engines to provide realistic environments. This is noticeable in 3D gaming where players expect fast paced interaction. This requirement has produced significant commercial activity resulting in physics engines for real-time simulation for rigid body dynamics. Although realism is convincing, a degree of mathematical accuracy is usually sacrificed (causing error) in order to attain real-time delivery within constrained resource settings. However, with significant commercial effort, the mathematical error has reduced over the years and commercial game engines are finding their use in other industries (e.g., [26, 12, 21]).

Combining physics engines with game engines (e.g., Unity, Unreal), presents a cost-efficient development platform for developing realistic environments. These gaming environments may be presented to geographically distant players using server-side scalable technologies. This allows scalable gaming platforms, presenting players with large online worlds to explore whilst maintaining the visually highly realistic environments generated by the player's local machine. Research into delivering scalability in such online worlds is focused on balancing real-time and consistency issues for enabling player-player interaction. The evolution of such research can be traced back to the earliest work on game area subdivision [13] through player-focused game area sub-division (e.g., [16, 7]) and eventually to the many commercial cloud based solutions as described by [25].

The problem of scalable player-player interaction is not a solved problem, as the numerous interaction possibilities across gaming genres and the latency in networks provides many different possible bespoke solutions. Player interaction can be measured in milliseconds, which current networking technology cannot model for all gaming genres in a scalable manner without giving rise to inconsistency. Therefore, the notion of trying to model real-time physics, which requires solvers working in iterations measured in microseconds, would be considered a near impossible task. However, achieving this would allow the physics engine element of the gaming console to be transplanted to the server-side (cloud), freeing up resources for game-play and rendering on a player's local machine.

Streamed gaming is an active area of research (e.g., [6]), where the game is executed on the server-side and the player interacts remotely (e.g., [22, 19]). The existence of such services suggests the possibility of real-time interaction for all gaming genres across cloud

infrastructures. Therefore, our problem is concentrated on distributing the physics engine element of such technologies across the server-side while maintaining the real-time streamed service to players.

Recent developments have demonstrated the importance of software defined networks (SDN) for presenting timely cloud solutions [24, 9]. The SDN manages the distribution of messages across servers in the most economical way possible (e.g., resource usage). The fact that SDNs can produce timely messaging across tightly coupled server-side deployments (the cloud) suggests the possibility that timely requirements for scalable physics delivery can be fulfilled.

In the context of current research, we can place ourselves at the juncture of balancing physics calculations across servers in the cloud for game streaming services. This should provide improved economic use of resources, as current game streaming services simply mimic the machine requirements of the player without distributing any of the component parts of the gaming engine. In addition, streamed games may benefit from increased numbers of artefacts (way beyond what is currently possible in a player's machine) as other servers may be utilised to solve the physics problems.

To achieve our research goal while remaining relevant to current state-of-the-art physics simulations, this study uses PhysX (from Nvidia) as the physics engine [20]. This ensures a simulation equal in detail to commercial video game titles. The challenge is to deploy instances of PhysX into a cloud infrastructure (on multiple servers) and allow objects within a simulation to be seamlessly passed across PhysX instances. We construct messaging services to enable PhysX instances on different servers to communicate and divide the simulation geographically across such servers. The challenge of this research is ensuring real-time fidelity can be satisfied when objects migrate across PhysX instances (from server to server). This problem of object migration is complicated by the possibility of object interaction occurring during such migration. Considering the assumption of commercial level fidelities of fast paced objects and frame rates of 60 frames per-second, any anomalous behaviour of objects during migration would be easily discernible to a player.

1.1 Purpose of Study

1.2 Outline

Chapter 2

Background and Related Work

2.1 Online Gaming

Although there has been research utilising multiple servers to distribute the task of solving physics based problems (e.g., [14]), to the best of our knowledge there is no literature describing real-time interactive physics exploiting the addition of servers to gain scalability. The closest work to our research is that carried out to seek scalability in terms of player numbers in online gaming in the field of Distributed Virtual Environments (DVEs).

There are primarily two ways in which server-side resources can provide scalability in online gaming (e.g., DVEs): (1) Migratory; (2) Non-migratory. In migratory approaches, a server will assume responsibility for handling in-simulation objects within a region. When objects traverse region boundaries into a region that is the responsibility of another server, they will be handed over to the other server. In a non-migratory approach, in-simulation objects are allocated to the responsibility of a particular server at instantiation time and stay with that server until they are deleted.

The benefit of a migratory approach is that tightly coupled objects (interacting frequently) can be co-located on the same server, reducing interaction latencies. However, the act of moving such objects may be costly in terms of time required to resolve the hosting requirements of an object. The benefit of a non-migratory approach is that servers are rarely exhausted but network traffic will result in higher latencies that will inhibit the fidelity of interaction between objects.

Migratory and non-migratory approaches are now described in greater detail.

2.2 Distributed Virtual Environments

In the migratory approach, a single game world exists, but is divided into geographical regions. Each region is maintained by a separate server (e.g. [5, 8, 4, 15, 10]). The main drawback of this approach is the complexity of handling interactions between objects in different regions/servers while maintaining consistency [27]. A technique to minimise these issues is to use overlapping regions between spatial partitions. Servers share state information about objects in the overlapping region (examples include 'zoning' as described in [5] or 'sub-regions'

as described in [8]). Examples of games using this technology include [17] and [23], which use the SpatialOS platform [10]. However, the techniques used by SpatialOS are not described in any literature. Their demonstration video exhibits unnatural object "jitter", which is possibly a result of network latency.

In the non-migratory approach, the game world is not divided into geographical regions and players are split between servers in one of two ways: (1) Several instances of the game world run with complete independence from one another (known as shards e.g.[2]) and players have no interaction across shards [27]; (2) Players are distributed amongst servers by some other non-geographical method and interactions with players on other servers requiring servers to share messages [11].

Although shards allow a degree of scalability in the number of players, it is not suitable for use in scaling real-time physics simulations as all entities within a real-time physics simulation, in the same geographical region, may interact with each other.

In the case of architectures not using shards, Interest Management is required to prevent message passing growing polynomially as players increase (e.g. [1] and [11]). [1] proposes the A3 algorithm, an interest management technique for distributed simulations aiming to significantly reduce the necessary bandwidth required between servers. A3 uses a combination of a circular area of interest and field of view combined with a relevance gradient. [11] proposes a Behavioural Interest Management Technique that allocates resources based on player interactions. Auras (an area of interest/influence) are used to determine player message exchanges, reducing message passing while promoting player number scalability.

Despite DVE being a popular area of research, the literature is restricted to modelling player interaction across servers and balancing their support on different servers. Clearly, the interaction patterns of players are significantly less demanding in terms of timeliness than that of interacting physical objects.

2.3 Streamed Gaming

Streamed Gaming (also know as Gaming as a Service) consists of cloud servers streaming to a player's device with player input being returned to the cloud server. The player's device acts as a thin client. The main benefit is that a player does not require expensive, powerful hardware, and games can be played on any operating system (e.g. Android, Linux and Mac). However, these benefits come at the cost of bandwidth and latency requirements [25].

Streamed Gaming services currently available include [19] and [22]. NVIDIA GRID technology is targeted specifically at Streamed Gaming [18].

A drawback to streamed gaming is the requirement for a significantly more powerful machine at the server-side than what would be required if the game was played solely at the client side. This is because the server not only has to run the game, but has to process the video and audio stream into a suitable format for streaming. In addition, real-time player interaction requires low latency and high bandwidth resulting in networking infrastructure more expensive than would be expected for regular streaming services.

In streamed gaming, each game instance resides on a single server. There is no technology to balance the real-time requirements of the game across multiple servers. The core problem is that all gaming technology is built and designed for single console/PC install and the greatest bottleneck is the inability to share physics calculations across machines.

2.4 Cloud Computing

2.5 Scalable Non-Real Time Physics

2.6 Background Summary

In physics simulations, there must be an object present in the solving phase for it to be considered in the overall solution of the scenario. A non-migratory approach requires a "ghost" representation of the object in a remote server to enable interaction. Given the calculations and discretisation steps, a "ghost" object takes up just as many resources in deriving a solution as a real object. This rules out the non-migratory approach for our problem. This leaves migratory.

Migratory approaches are concerned with managing the network traffic for those entities that could possibly exist on two servers but can only be solved on one. Such objects need to be placed with an owning server while minimising the effect of thrashing (where an algorithm frequently transfers objects between servers). In the rest of this paper, we describe our approach to solving this algorithmically, present how a working implementation was achieved, and present results evidencing our work. This is the first presentation of literature that can demonstrate real-time scalable server-side physics modelling and is a significant contribution to reducing the cost of commercialised streamed gaming.

Chapter 3

Problem Definition and Proposed Solution

Our approach allows servers to assume responsibility for objects within a real-time simulation without hindering the fidelity of such a simulation. In order to maintain scalability, the extra processing overhead must be less than that gained by distributing the simulation workload.

3.1 Naive Approach

To provide a comparison and to demonstrate our contribution to the field, a naive approach is described first. The naive approach is used to highlight the challenges faced when solving this problem.

Assuming a migratory approach in which servers are responsible for a geographic area, when one object crosses a boundary between geographic areas, such an object's hosting must be transferred to the appropriate server. It is handling this transfer that is the underlying problem.

Accomplishing a transfer with a simple message from one server to another is charged with many problems: (1) the message may be lost and the object disappears; (2) the message may be delayed and the object disappears and reappears; (3) there may be more than one candidate server for hosting (if object travelling quickly) resulting in duplication of an object; (4) the server an object leaves may be unaware of when it should stop simulating the object (resulting in duplicated objects)[3]; (5) an object may be colliding with another object across the boundary which would be lost if both objects are hosted on different servers (objects may pass through each other).

For additional clarity consider objects A and B in Fig. 3.1. Each server is aware of the objects they host but not other objects. As such, objects A and B would not interact and not collide with each other, creating a time-space inconsistency. A time-space inconsistency is defined as two objects occupying the same space at the same time. If Object B has a velocity directed towards the boundary, it would continue at its initial velocity until it traversed the boundary, at which point it would migrate to server 0. Upon completion of the migration, objects A

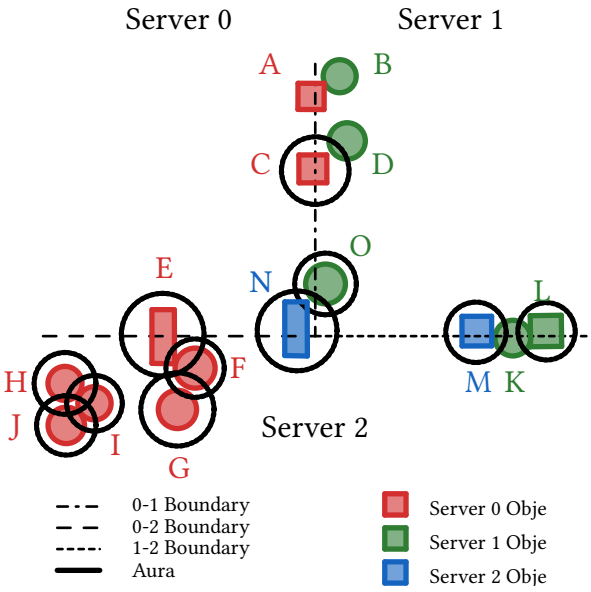


Fig. 3.1 Aura Projection Scenarios

and B would overlap far more than would be expected in a typical simulation, resulting in inappropriate collision response.

3.2 Proposed Solution

We term our approach Aura Projection (AP). AP tackles the problem of maintaining consistency across regions while ensuring timeliness of a simulation. In particular, AP provides the building blocks of a scalable solution to server-side physics simulations by handling all configurations of boundary cases in migratory approaches.

In AP, an object maintains an aura, an area of interest around the object, that indicates its possible future location and, therefore, aids in identifying near future interactions. Objects with colliding auras are highlighted as candidates to be co-hosted on the same server in the near future. The presence of an aura allows: (1) the prediction of future hosting requirements allowing time to transfer objects; (2) a narrowing of interest in only considering a subset of objects promoting a scalable solution; (3) an unhindered physics simulation for existing objects; (4) limiting communications requirements between servers based on focusing message passing overhead on interacting objects (promoting scalability). In essence, AP ensures that any two objects that may be interacting are always being simulated on the same server. For example, objects C and D illustrated in Fig. 3.1. Object C projects its aura into Server 1, Object D collides with the aura and is migrated to Server 0, allowing continued interaction to occur.

Messages are sent between servers where aura overlap occurs across a geographic boundary. We term this phase of the algorithm as an object projecting its aura onto another server.

In the context of real-time physics simulations, auras appear on the receiving server as a trigger volume (non-physically interacting volume). For each simulation step that an object is projecting an aura, the object's position and calculated aura are sent. As a network optimisation,

if the boundary object is not moving, no details are sent and the aura on at a receiving server is unaltered.

When an aura is no longer being projected for an object (boundary not overlapped), the server simulating that object sends a message to any servers receiving the aura to remove the aura of that object.

3.3 Aura Calculation

Calculating an aura size is key to creating a scalable solution balanced against consistency: (1) Auras that are too small will result in a faster simulation but with more missed interactions; (2) Auras that are too large will include more calculations that are simply not required, increasing server load and decreasing scalability. Therefore, how auras are calculated will be carefully described and justified.

Spheres are used to represent auras. This bounding volume is computationally efficient as rotational calculations are not required in determining its correct alignment. Auras accommodate the displacement of objects using an estimated network latency based on historic monitoring, in addition to the distance a remote object may be predicted to penetrate an aura (considering velocities).

Fig. 3.2 demonstrates the sequence of events involved when an aura is sent, received, and collided with, resulting in an object migration.

To lower inconsistencies during migration, the following user-defined tolerances are used for the aura calculations: maximum speed; maximum latency time; maximum frame-time. If velocities, latencies, or frame-time are above these tolerances, then stability is no longer guaranteed. This enables AP to deterministically indicate to the overall simulation when latency may be influencing the mutually consistent views of the servers, which in turn manifests as errors in the physics.

To calculate the radius of an aura, the maximum distances travelled by objects within the delay time between simulations must first be calculated. To calculate the maximum distance an object may travel for a given time, the following formula is used: $\triangle s = v \cdot \triangle t$, where $\triangle s$ is distance, v is speed, and $\triangle t$ is time. In this context, $\triangle v$ is substituted with maximum speed tolerance, and $\triangle t$ will be a multiple of the physics step time (to accommodate discrete time step calculations in the physics engine). $\triangle t$ can be calculated using the maximum frame-time and maximum latency tolerance, discussed below.

There is a time delay between the aura being created on a host and being created on a receiving server, which is made up of: up to one frame before sending the aura creation message; inter-server latency; up to one frame from the message being received and acted upon; the time delay between the aura being created and the detection of the collision between a potential remote object and the aura by the physics update step. This last time delay is accounted for by rounding the previous delays up to the nearest physics time step.

Using the time delays mentioned above and substituting the relevant tolerances, the maximum displacement time (Max_{DT}) of a remote object is calculated using the following equation:

$$T_R = \left\lceil \frac{2 \cdot T_F + T_L}{T_P} \right\rceil T_P \tag{3.1}$$

T_R is the Max_{DT} of a remote object, T_F is the frame-time tolerance, T_L is the latency tolerance and T_P is the physics step time.

The aura has to account for the maximum displacement of both the object on the host server and a potential object on the remote server. The total Max_{DT} will be the sum of: a) the Max_{DT} of a remote object, subtracting one physics time step, as the aura only needs to account for the displacement of the host object after the creation of the aura; b) The following time delays: a time delay of up to one frame time between the physics step of the remote server and the remote server update loop sending the migration message from the migration buffer; a time delay of the latency between servers; a time delay of up to one frame time between the message being received by a host and a host acting on the migration message and creating the migrated object in the physics engine; a delay between the migrated object being created and the collision being detected by the physics update step. This last time delay is accounted for by rounding the previous delays up to the nearest physics time step.

The Max_{DT} of the object on the host that is projecting the aura is therefore calculated using the following equation:

$$T_H = T_R - T_P + \left\lceil \frac{2 \cdot T_F + T_L}{T_P} \right\rceil T_P \tag{3.2}$$

T_H is the Max_{DT} of the object on the host.

The total Max_{DT} is therefore $T_R + T_H$, which can be simplified to the following equation:

$$T_T = (3 \left\lceil \frac{2 \cdot T_F + T_L}{T_P} \right\rceil - 1) T_P \tag{3.3}$$

T_T is the total Max_{DT}.

The aura of an object can then be calculated using the following equation:

$$R_a = R_o + (V_t \cdot T_T) \tag{3.4}$$

R_a is the radius of the aura in m, R_o is the bounding sphere of the object, V_t is the speed tolerance in $m \cdot s^{-1}$ and T_T is the total Max_{DT} in s.

3.4 Thrashing

An object may be overlapping two auras from different servers. Given appropriate velocity, this could result in object migration, followed by aura collision, followed by migration and then repeating the process again. For example, Objects K, L and M in Fig. 3.1. Object K lies inside the auras of both Object L and M. The common term ‘thrashing’ will be used to describe such a scenario.

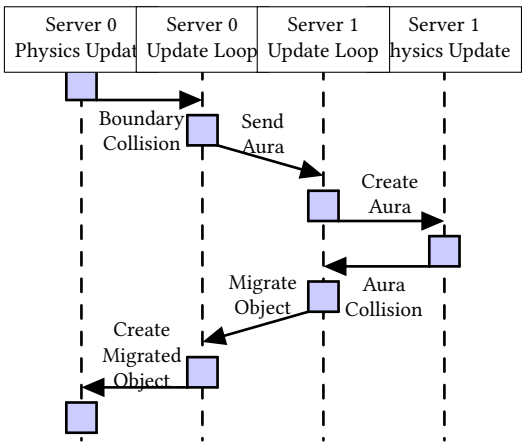


Fig. 3.2 Aura and migration sequence diagram

In order to prevent thrashing, when an object is migrated, it is also migrated with any objects found that lie within the aura of the object or have an existing aura which overlaps the migrating object's aura. This is carried out recursively to ensure all objects that are overlapping are migrated at the same time.

3.5 Islands

When an object traverses a region boundary, the object is migrated only if it does not overlap an aura projected by an object from the same server. In the context of migrations, an island is defined as two or more objects located outside of their host server's region (i.e. have traversed a region boundary) but are each within the aura projected from objects owned by the same host. For example, Objects H, I, and J illustrated in Fig. 3.1. No objects in the island are migrated as each object is within the aura of another object in the island. This should be prevented as it causes processing and networking overhead and is unnecessary as all objects lie within the region of Server 2 and are not interacting with any other objects from Server 0.

To prevent islands, a search is performed at each time step to determine if an object is part of an island. A search is performed to determine if a potential migratory object is within a group of objects with overlapping auras, of which none are positioned within the hosting server region. If the group of objects has no members within the hosting server region then the object is part of an island and the entire island of objects is migrated. Otherwise, no action is taken. For example, in Fig. 3.1 Object I has overlapping auras with Objects H and J. H and J are checked for overlap with the Server 0 - Server 1 region boundary and for overlap with auras from Server 0 that are intersecting the Server 0 - Server 1 boundary. In this scenario, H and J are found to be part of an island, so H, I, and J are all migrated to Server 2. If objects are found to not be part of an island, no action is taken. For example, Objects E, F, and G in Fig. 3.1. Object F is found to have an overlapping aura with Object E, but Object E intersects the Server 0 - Server 1 boundary, so E, F, and G are found to not be part of an island and therefore no action is taken. This solution is shown in Algorithm 2 and the problem of islands can therefore be considered solved.

3.6 Corner Case

This section discusses how AP handles corner cases, i.e., where the boundary of more than two servers meet.

An example of the corner case is illustrated in Fig. 3.1. Object N, hosted on Server 2, sends an aura to Server 0. Object O, on Server 1, sends an aura to Server 0. Object N's aura overlaps with Object O's on server 0. Server 2 is unaware of Object O and Server 1 is unaware of Object N. As the two auras overlap, there is a potential interaction between the objects, yet the two objects remain on separate servers. To solve this problem, auras from boundaries between two neighbouring servers are shared. In the region layout in Fig. 3.1, auras from a boundary are received by interested servers. In the Object N and O example, Server 2 would send Object N's aura to Server 1; when Object O collides with Object N's aura, migration to Server 2 occurs.

A server has to receive the auras of all objects from neighbouring boundaries, as objects being simulated by a server can exist anywhere in a neighbouring server's region. Our assumption is that a region a server simulates is sufficiently large enough to prevent the aura overlap of objects from non-neighbouring servers.

Chapter 4

Implementation

This section will discuss an implementation of AP using PhysX and RakNet.

4.1 System Architecture

The simulation space is partitioned into regions, with each region consisting of its own instance of PhysX running on a dedicated GPU-enabled machine in the cloud. The boundary between regions is defined as a vertical plane, two-dimensionally dividing the simulation space into one region per server. The network library RakNet is used for all message passing between servers. RakNet ensures messages exhibit best effort and are received in sent order.

When objects project auras they are added to a send aura buffer that is sent to all servers associated to the boundary of concern. Each object has a unique identifier (ID). When an aura is received by a server, an aura is created if it does not already exist, otherwise the aura is updated using the data received. When an object is no longer projecting its aura, the ID of that object is added to the delete buffer which is then sent to all servers neighbouring the boundary.

When objects traverse region boundaries, they are added to a migration buffer with all information required to duplicate an object at a neighbouring server. The contents of a migration buffer are sent to a server now responsible for hosting an object. When migration messages are received an object is created within the server's simulation.

Clients may connect to any server and are provided with a streamed visualisation of the simulation in real-time. Clients may also interact with and influence the simulation, providing a comprehensive solution for real-time interactive physics. The client system was built using the Unreal Engine. Once a client is connected, the position and states (replicas) of all objects in the simulation are sent from each server to the client via the RakNet Replica Manager.

4.2 Algorithms

AC is called when an object collides with an aura. A recursive search is performed in order to find all objects that would lie within each object's aura, preventing thrashing as discussed in 3.4. Once the recursive search is complete, all objects are added to the send buffer.

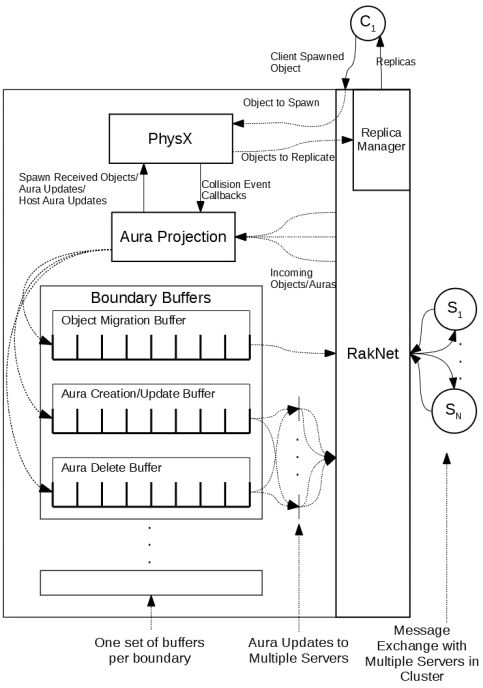


Fig. 4.1 Server Architecture

BT is called when an object traverses a boundary. In order to prevent 'islands' forming (for example Objects H, I and J in Fig. 3.1), a recursive search is carried out to determine is an object is part of an island or not. If an object is found to not be part of an island, the entire cluster of objects is added to the send buffer, otherwise no action is taken.

OBC is called when an object collides with a boundary. The object's aura is calculated and added to the boundary's send aura buffer. A host aura is also created, to allow for the checking of mutual aura overlaps and prevent thrashing, if this is the first boundary intersection.

OBU is called once per frame that an object is intersecting a boundary. If the object is not 'sleeping', a new aura is calculated and added to the boundary's send buffer and the host aura is updated.

The isSleeping() function returns true if an object is sleeping. From the PhysX documentation: An object is considered 'sleeping' when an actor does not move for a period of time. (The default PhysX period of time is $0.4s$ and this is the value used in our approach). Objects are 'woken up' when they are touched by an awake object.

OBE is called when an object is no longer intersecting a boundary. The object is added to the boundary's remove aura buffer. If the object is no longer intersecting any boundaries, the host aura is deleted.

BNU is called once per network connection between servers. It is responsible for sending and receiving object migrations and auras between servers, including the sending and receiving of auras from boundaries between other neighbouring remote servers.

Algorithm 1 Object Migrate - Aura Collision (*AC*)

```
1:  procedure OnAuraEnter                                   ▷ A callback on an object
2:      ▷ Track visited objects to prevent infinite recursion
3:      visited := {}
4:      ▷ Recursively send object with objects that would lie within each object's aura
5:      SendWithOverlaps(thisObject, visited)
6:
7:  procedure SendWithOverlaps(object, visited)
8:      ▷ Get objects whose auras overlap this object's aura
9:      overlaps := GetAuraOverlaps(object)
10:
11:     for each object ∈ overlaps do :
12:         if object ∉ visited then:
13:             visited := visited + object
14:             SendWithOverlaps(object, visited)
15:
16:     AddToSendBuffer(object)
```

Algorithm 2 Object Migrate - Boundary Traverse (*BT*)

```
1: ▷ Update called on each boundary
2: procedure BoundaryUpdate
3:     checked := {}                                  ▷ Used to prevent duplicate checks
4:
5:     ▷ Loop over fully traversed objects from latest update
6:     for each object ∈ traversed do :
7:         if object ∈ checked then:
8:             continue
9:         island := {}
10:        isIsland := IslandQuery(object, island)
11:        ▷ IslandQuery() returns true if object is part of an island and a list of objects in the
   island
12:        if isIsland = true then:
13:            SendGroup(island)
14:        checked := checked + island
15:
16: function IslandQuery(object, visited)
17:    visited := visited + object
18:    if object.overlapsHostRegion = true then
19:        return false
20:
21:    ▷ Get objects whose auras overlap this object's aura
22:    overlaps := GetMutalAuraOverlaps(object)
23:
24:    ▷ If all objects with overlapping auras are islands, then this object is an island
25:    isIsland := true
26:    for each object ∈ overlaps do :
27:        if object ∉ visited then:
28:            isIsland &= IslandQuery(object,visited)
29:    return isIsland
```

Algorithm 3 Create Aura - Object boundary collision (*OBC*)

```
1: ▷ A callback on an object, called when an object collides with a boundary
2: procedure OnBoundaryEnter(boundary)
3:     AddToAuraBuffer(boundary, this)
4:
5:     ▷ Create 'host aura' so GetMutalAuraOverlaps() will detect this object's aura
6:     if boundaryIntersections = 0 then
7:         CreateHostAura()
8:     boundaryIntersections := boundaryIntersections + 1
```

Algorithm 4 Update Aura - Object boundary update (*OBU*)

```
1: ▷ A callback on an object, called per step per boundary the object is colliding with
2: procedure onBoundaryUpdate(boundary)
3:     if this.isSleeping = true then
4:         return
5:     ▷ Send Aura Delta
6:     AddToAuraBuffer(boundary, this)
7:     UpdateHostAura()
```

Algorithm 5 Destroy Aura - Object boundary exit (*OBE*)

```
1: ▷ A callback on an object, called when an object exits a boundary
2: procedure onBoundaryExit(boundary)
3:     AddtoDeleteAuraBuffer()
4:
5:     ▷ If object is no longer sending an aura, no need to keep a 'host aura'
6:     boundaryIntersections := boundaryIntersections - 1
7:     if boundaryIntersections = 0 then
8:         DeleteHostAura()
```

Algorithm 6 Boundary Network Update (*BNU*)

```
1: ▷ Update called once per network connection
2: procedure Network Update
3:     ▷ Exchange migrations with target server
4:     SendObjectsInBuffer
5:     ReceiveObjects
6:
7:     ▷ Send aura state updates to all neighbours
8:     for each neighbour ∈ neighbours do :
9:         SendAurasInBuffer
10:        SendDeleteAurasInBuffer
11:        ReceiveAuras
12:        ReceiveDeleteAuras
```

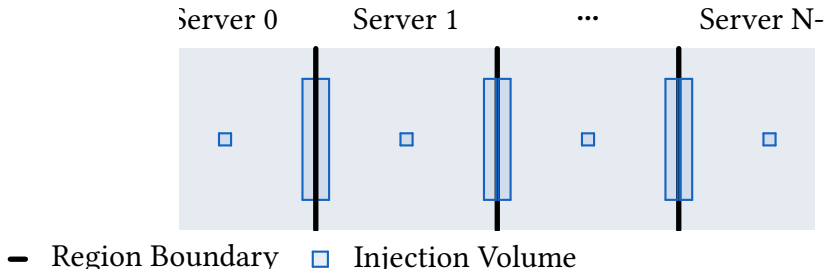


Fig. 5.1 Column Layout

Chapter 5

Experiments and Results

5.1 Scalability Experiments

This study aims to demonstrate scalability. When more servers are added, the timeliness of the simulation improves and more objects may be supported. The performance measure of interest is the maximum frame time of a server as this indicates if the simulation can be maintained when object numbers increase (keeping a low frame-time is the goal). Therefore, an injection rate is used, that spawns moving objects into the simulation as time passes.

Experiments were performed on two layouts of servers: column layout and corner layout. The column layout experiment was performed using an increasing number of servers from 1 to 10. The regions were laid out in a column configuration as shown in Fig. 5.1. Objects are injected at a constant rate, both near and far from boundaries. Injected objects are randomly selected from the following types: sphere (radius: $0.3m$), cuboid ($0.3m \times 0.3m \times 1.0m$) and capsule (radius: $0.3m$, height: $2m$) and start with a random velocity from the uniform distribution of: $(-10 < x < 10, -10 < y < 0, -10 < z < 10) m \cdot s^{-1}$. 50% of objects are injected in a volume of $20m \times 20m \times 150m$ centred $12m$ away from a boundary and $15m$ above the ground plane. 50% of objects are injected in the centre of a server's region in a volume of $20m \times 20m \times 20m$, $15m$ above the ground plane.

The experiments were run for 60 seconds with an injection rate of 160 objects per second. For all experiments, a speed tolerance of $32 m \cdot s^{-1}$ was used, which was the maximum expected speed for any object (based on maximum injection height and velocity, and gravity). The latency tolerance for these experiments was set to $2ms$ (based off measurements of latency between servers) and frame-time tolerance was set to $15ms$ (based off preliminary performance

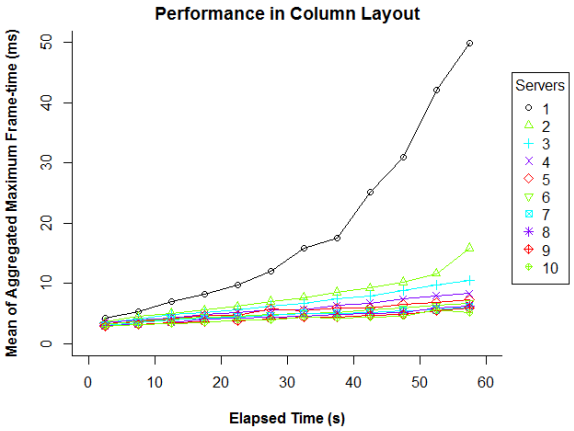


Fig. 5.2 Performance of increasing numbers of servers with an accumulating number of objects (in column layout)

results). A time of 16*ms* was used for the physics time-step. Each experiment was repeated 50 times at various times of day to account for differing performance in cloud resources at different times. For each iteration, the maximum frame time of any server was aggregated for each 5*s* period. The mean of the aggregated maximum frame times of the iterations was then calculated and plotted.

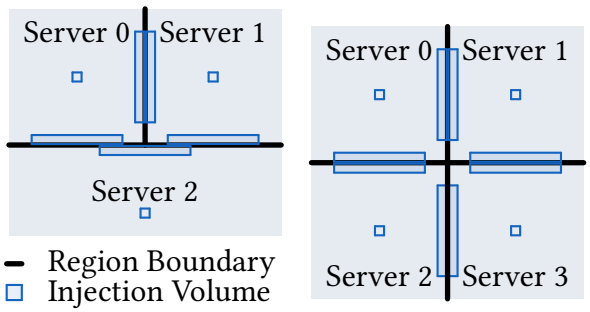
Experiments were conducted using AWS G2.2xlarge servers located within the same geographical region. A G2.2xlarge instance uses a 2.60GHz Intel Xeon E5-2670 CPU with 16GB RAM and an NVIDIA GRID K520 (Kepler) GPU running Amazon Linux AMI 2017.09.

Fig. 5.2 shows the performance of our system with rising server numbers from 1 to 10. The graph clearly shows that the addition of servers lowers the frame time throughout the experiment. This is increasingly noticeable later in the simulation when a greater number of objects are present. For higher server numbers the reduction in maximum frame time is less than for lower numbers, which is as expected as in the ideal case the workload per server would be $1/n$ of the total workload, where n is the number of servers.

From these observations it may be declared that this system is scalable in column configuration as the addition of servers results in increased performance.

Experiments were also carried out using servers in a corner layout. The layouts of 3 and 4 servers are shown in Fig. 5.3, in which the 4 server case is a 2x2 grid with a single corner intersection in the centre. The 9 server case is a 3x3 grid with 4 corner intersections. Injection rates and volume dimensions remain the same as in the column based experiments. Fig. 5.4 shows the graph describing performance. The graph demonstrates that increasing the number of servers lowers the frame time. The additional processing overhead of exchanging more messages with more neighbouring servers in the 9 server experiment is outweighed by the performance gains of additional servers.

From these observations it may be declared that this system is scalable in corner configuration as the addition of servers results in increased performance.



(a) 3 Server Corner Layout (b) 4 Server Corner Layout

Fig. 5.3 Corner Layouts

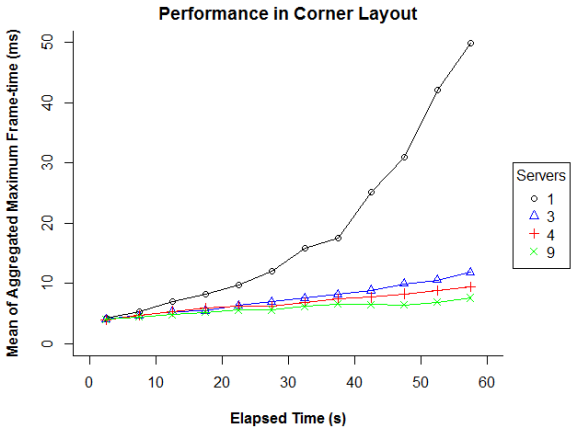


Fig. 5.4 Performance of increasing numbers of servers with an accumulating number of objects (in corner layout)

5.2 Correctness Experiments

In the following section we explore the correctness of collisions between objects, i.e. do objects have the same collision behaviour in a AP as they do in a centralised system. We are concerned with objects that are migrating and/or interacting with an object that has recently migrated. We will now explain how a naive system can lead to erroneous collision behaviour and AP can lead to erroneous collision behaviour if speed, frame-time or latency go above the user-defined tolerances.

Erroneous collisions fall into two categories:

- Missed collisions, collisions that should have taken place but were not detected by the physics engine e.g. Bullet through paper problem.
- Late collisions, collisions that were detected later than they should have been and result in not only different collision results, but also unstable collision response.

To understand how late collisions lead to unstable collision response, we must first understand how in real-time collision detection works. In real-time physics engines, objects move in descrete steps, as a result of this, when two objects collide, they overlap each other i.e. penetrate. The collision is resolved by calculating the forces resulting from the collision and moving the objects so they no longer overlap (in PhysX if penetration is high, the latter may

be done over several physics time-steps). If the penetration is high, such as in the case of a late collision response or if two objects are moving at high speed towards one another, the collision response can no longer guarantee stable results.

It is possible to detect late collisions between objects. Given the relative speed of the two objects and the physics time-step, it possible to calculate the maximum expected penetration distance. If a collision is detected and it is above this value, it means it is a late collision.

Maximum penetration for a given speed occurs when two objects travelling directly towards each other have, in the final time-step before colliding the two objects have no distance between them, but are not overlapping, so not colliding. The next physics time-step, the collision will be detected and will have the maximum penetration possible for that relative speed. Re-arranging s=d/t, to d=s.t and substituting the physics time-step for t, gives us the maximum distance (penetration) for a relative speed, and we are able to plot the maximum expected penetration line.

Experiments were carried out where collision results were recorded and used to detect missed and late collisions of objects. The experiment scenario used only two objects, meaning if there was a missed collision there would be no collision output. The two objects start the scenario with a specified velocity (drag and gravity are disabled), the direction being directly towards the other object and the two objects were given the same speed, which is varied throughout the experiment. Spheres were used, so rotational effects don't need to be accounted for.

A control experiment on a centralised system was performed, in which the scenario was repeated multiple times. The purpose of this experiment was to confirm the predicted maximum expected penetration. Fig. 5.5 shows the penetration of objects with increasing speed in a centralised simulation and demonstrates that penetration is never greater than the maximum expected penetration.

AP can only lead to late or missed collisions when at least one object involved in the collision has just migrated (received since the last physics time-step). In order to test this case, the same scenario as the control was used with two servers. The two spheres were given starting positions so the two would collide at a point on the boundary between servers, thus creating the most likely situation for objects collide with each other in their first time-step after being migrated. The purpose of this experiment is to demonstrate that collisions in AP are always correct if the respective values remain within their tolerances. Only objects that have just migrated are considered in the experiment results. The two servers use the same tolerances used in the performance experiments, including a maximum speed tolerance of $32 m \cdot s^{-1}$.

5.3 Case Study: Suspension Bridge

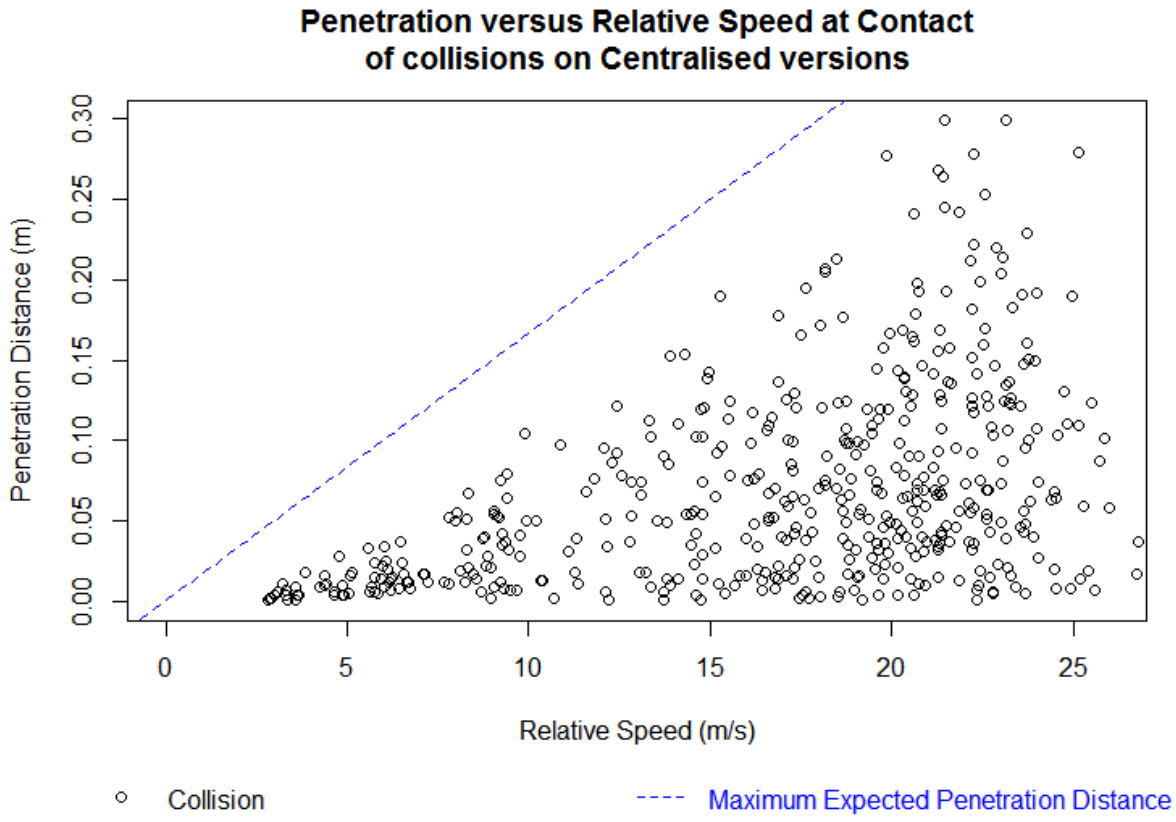


Fig. 5.5 Penetration of objects with increasing speed and maximum expected penetration

Chapter 6

Conclusions and Future Work

An approach to networked real-time physics simulations that is scalable and alleviates the processing limitation of a single server has been presented. Only open-source software has been used in our approach and our algorithm has been developed in a way that is agnostic to any specific application technology.

Our experiments establish that the approach is scalable, as demonstrated by the addition of servers improving the performance of the system when simulating an increasingly large number of objects. This study has demonstrated that a standard real-time physics engine (in this case, PhysX) may be incorporated into our scalable real-time physics system and achieve performance that is acceptable for real-time distributed simulations such as networked games.

Our continuing work will be to carry out experiments to determine the effects that increasing latency and packet loss, and varying user defined latency and speed tolerances, have on the scalability and stability of our approach. This will provide data, which may inform a dynamically adapting messaging layer that may manage cloud resources more efficiently depending on the distribution of object interaction within a simulated world.

We are currently seeking to extend our approach to support large jointed objects that could span multiple servers. For example, a suspension bridge could be modelled in fine detail using multiple servers. This requires our approach to be extended to ensure some jointed elements may span section boundaries to connect independent objects on different servers.

Future work will enable boundaries between regions to evolve dynamically allowing load-balancing to occur, further lowering the maximum frame time across servers. In addition to load-balancing, dynamic boundaries can routinely repartition the regions between servers to avoid or reduce the number of objects interacting over region boundaries. This will reduce both network and processing overhead by avoiding the need for auras and migrations to be exchanged between servers, further improving the performance and scalability of our approach.

References

[1] Bezerra, C. E., Cecin, F. R., and Geyer, C. F. R. (2008). A3: A novel interest management algorithm for distributed simulations of mmogs. In *Proceedings of the 2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*, DS-RT '08, pages 35–42, Washington, DC, USA. IEEE Computer Society.

[2] Blizzard (2019). World of warcraft. (Accessed on 22/02/2019).

[3] D'Amora, B., Nanda, A., Magerlein, K., Binstock, A., and Yee, B. (2006). High-performance server systems and the next generation of online games. *IBM Systems Journal*, 45(1):103–118.

[4] de Senna Carneiro, T. G. and Arabe, J. N. C. (1998). Load balancing for distributed virtual reality systems. In *Computer Graphics, Image Processing, and Vision, 1998. Proceedings. SIBGRAPI '98. International Symposium on*, pages 158–165.

[5] Dong, L. and Yue-Long, Z. (2013). An overlapping architecture for roia in cloud. In *2013 Fourth International Conference on Emerging Intelligent Data and Web Technologies*, pages 61–65.

[6] García-Valls, M., Cucinotta, T., and Lu, C. (2014). Challenges in real-time virtualization and predictable cloud computing. *Journal of Systems Architecture*, 60(9):726 – 740.

[7] Greenhalgh, C. and Benford, S. (1995). Massive: a distributed virtual reality system incorporating spatial trading. In *Proceedings of 15th International Conference on Distributed Computing Systems*, pages 27–34.

[8] Hori, M., Iseri, T., Fujikawa, K., Shimojo, S., and Miyahara, H. (2001). Scalability issues of dynamic space management for multiple-server networked virtual environments. In *2001 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (IEEE Cat. No.01CH37233)*, volume 1, pages 200–203 vol.1.

[9] Kumar, R., Hasan, M., Padhy, S., Evchenko, K., Piramanayagam, L., Mohan, S., and Bobba, R. B. (2017). Dependable end-to-end delay constraints for real-time systems using sdns. *CoRR*, abs/1703.01641.

[10] Limited, I. (2019). SpatialOS. (Accessed on 22/02/2019).

[11] Lu, F., Parkin, S., and Morgan, G. (2006). Load balancing for massively multiplayer online games. In *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '06, New York, NY, USA. ACM.

[12] Lu, X. and Guan, H. (2017). *Visualization for Earthquake Disaster Simulation of Urban Buildings*, pages 303–326. Springer Singapore, Singapore.

[13] Macedonia, M., J. Zyda, M., R. Pratt, D., T. Barham, P., and Zeswitz, S. (1994). Npsnet: A network software architecture for large scale virtual environments. 3:265–287.

[14] Mashayekhi, O., Shah, C., Qu, H., Lim, A., and Levis, P. (2018). Automatically distributing eulerian and hybrid fluid simulations in the cloud. *ACM Transactions on Graphics (TOG)*, 37(2):24.

[15] Min, D., Choi, E., Lee, D., and Park, B. (1999). A load balancing algorithm for a distributed multimedia game server architecture. In *Proceedings IEEE International Conference on Multimedia Computing and Systems*, volume 2, pages 882–886 vol.2.

[16] Morgan, G., Lu, F., and Storey, K. (2005). Interest management middleware for networked games. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, I3D '05, pages 57–64, New York, NY, USA. ACM.

[17] NINPO (2019). Vanishing stars: Colony wars by ninpo. http://www.vanishingstars.com/. (Accessed on 22/02/2019).

[18] Nvidia (2019a). Cloud gaming - gaming as a service (gaas) | grid|nvidia. (Accessed on 22/02/2019).

[19] Nvidia (2019b). Game stream. (Accessed on 22/02/2019).

[20] Nvidia (2019c). Physx knowledge base/faq | Nvidia UK. (Accessed on 22/02/2019).

[21] Shah, S., Dey, D., Lovett, C., and Kapoor, A. (2018). *AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles*, pages 621–635. Springer International Publishing, Cham.

[22] Sony (2019). Playstation now. (Accessed on 22/02/2019).

[23] Studios, B. (2019). Worlds adrift. https://www.worldsadrift.com/. (Accessed on 22/02/2019).

[24] Wang, X., Wang, C., Zhang, J., Zhou, M., and Jiang, C. (2017). Improved rule installation for real-time query service in software-defined internet of vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 18(2):225–235.

[25] Wu, D., Xue, Z., and He, J. (2014). icloudaccess: Cost-effective streaming of video games from the cloud with low latency. *IEEE Transactions on Circuits and Systems for Video Technology*, 24(8):1405–1416.

[26] Xu, J., Tang, Z., Wei, X., Nie, Y., Yuan, X., Ma, Z., and Zhang, J. J. (2017). *A VR-Based Crane Training System for Railway Accident Rescues*, pages 207–219. Springer International Publishing, Cham.

[27] Yahyavi, A. and Kemme, B. (2013). Peer-to-peer architectures for massively multiplayer online games: A survey. *ACM Comput. Surv.*, 46(1):9:1–9:51.