DiVE: a Scalable Networking Framework for Distributed Virtual Environments

Tristan Imbert, João Oliveira, Marconi Madruga, Helmut Prendinger

National Institute of Informatics

2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan[1]

Abstract

We present DiVE, a networking framework for the rapid development of massively multiuser three-dimensional (3D) virtual world applications. The design aims at polymorphism and simplicity. DiVE tries to strike a balance between fully-featured virtual world application servers such as Second Life or OpenSimulator and minimal networking engines used for massively multiuser scenarios. DiVE has already been used successfully in multiuser applications in the traffic and evacuation domains. In this paper, we provide a technical description and evaluation of the DiVE framework. Specifically, we present an 'area of interest' technique that is based on dividing the world into regions and a 'double' (inner/outer) area of interest concept. The empirical evaluation of the technique shows promising results on four metrics: bandwidth, frames per second, CPU, and RAM.

Keywords: Distributed virtual environments, area of interest management, empirical evaluation

1. Introduction

Popular 3D virtual world application servers like the Second Life[1], OpenSimulator[2] or realXtend Tundra[3] offer content creators powerful features. Objects can be created and modified 'in-world' through extensive high-level Application Programming Interfaces (APIs) and behaviors can be programmed through simple scripting languages.

Email address: timbert@live.fr, jjoaoliveira@gmail.com, koni.kun@gmail.com, helmut@nii.ac.jp (2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan)

URL: http://www.nii.ac.jp/~prendinger/ (2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan)

[1] http://secondlife.com/

[2] http://opensimulator.org/

[3] http://realxtend.org/

This content author friendly approach comes at a price though. Since these platforms synchronize every aspect of shared objects over the network to enable the user-driven content creation and rely on inefficient protocols, (1) the number of simultaneous users per server is typically limited to a few dozen, (2) login and movement latency are high, and (3) individual scenario regions are limited in size. Furthermore, adoption of existing single-user applications to the Second Life network paradigm or the addition of features to existing Second Life viewer applications tends to be cumbersome.

Game-centric network frameworks, Jenkins Software's RakNet[4] or OpenTNL [5], on the other hand, are optimized for speed and scalability. However, despite their generic and flexible APIs, they are too low-level, hence confronting developers with a high degree of complexity for virtual world applications.

Therefore, we developed our original Distributed Virtual Environment (DiVE) technology[6] as an attempt to combine the best of both approaches:

- Scalability in terms of the number of networked clients
- Simplicity of API optimized for the demands of virtual world application developers

Regarding scalability, there is a growing interest in scalable virtual worlds that can accommodate large numbers of concurrent users and support massively multiuser applications such as games or social simulations. The term 'state melding' was proposed to name a technology that creates the illusion of a shared reality [2]. State melding consists of two parts, consistency maintenance and state update dissemination. Consistency maintenance refers to client-side prediction models such as dead reckoning. State update dissemination investigates methods to reduce number of updates that each client receives. If one entity of a client moves (e.g., an avatar), each other $n-1$ client has to receive the update of the moved entity's position. In case all entities move, $n(n-1)$ updates have to be transmitted over the network.

To address this problem, we will propose, and test, a region-based area of interest technique.

The paper is organized as follows. Section 2 discusses related work on distributed virtual environments. Section 3 first presents the features of the DiVE framework. In particular, we describe how to set the region-based area of interest technique and empirically use different kind of movement to get an image of the computational load. Next, we describe the DiVE architecture, its features and highlight the ease developers can have to handle it. In Section 5, we evaluate our approach empirically by varying the area of interest and testing its effect on metrics such as bandwidth and frames per second. Section 6 summarizes and concludes the paper.

[4] http://www.jenkinssoftware.com/

[5] http://www.opentnl.org/

[6] We are aware of a framework with a similar name (DIVE) [1].

2. Related Work

In this section we discuss related works on scalability in Distributed Virtual Environments (DVEs) and on other networking frameworks. While commercial virtual world platforms such as Linden Lab's Second Life have fallen short of early expectations regarding their adoption rate in education and business collaboration, many research-oriented DVEs follow their design philosophy.

2.1. Scalability in DVEs

Distributed virtual environments address the problem of scalability in different ways. As one solution, techniques of multi-server simulation is introduced. Research work such as [3] propose a partitioning algorithm to share a same virtual world among several servers. Handling connections and distributing calculation among several servers makes it possible to significantly scale virtual worlds. However, deploying a multi-server solution for a given application is costly in hardware and complexity.

Some large scale projects, such as OpenSim, implemented this approach. Generally, the design philosophy of OpenSim consists of two primary components: world simulators and viewers. Simulators are hosted on servers, whereas viewers are run on user (client) machines. A virtual world in Second Life or OpenSim is statically partitioned into equally sized, interconnected "regions", each of which is managed by one and only one simulator process. One region can only accommodate a limited numbers of individual users (clients), as its simulator is solely responsible for all of that region's computing workload, including client connection handling, object manipulation, and physics simulation.

For geographical scaling, OpenSim relies on federation, i.e. distributing regions and simulators over multiple servers. The Hypergrid architecture [4] allows simulators to share their authentication and inventory services and enables single sign-ons spanning large (and inter-institutional) deployments. When passing region boundaries, users are automatically 'teleported' to the server and simulator hosting the respective adjacent region. While this design facilitates arbitrarily large worlds, it does not address the problem of over-loading single, popular regions with user connections. Over-loading has been identified as a critical bottleneck in simulator-centric approaches, as they cannot dynamically allocate hardware to scale operations [5].

Recently, researchers at Stanford University introduced a way to distribute the world across several servers (federation) and to order the information to ensure a natural and quick message propagation among servers [6, 7, 8]. This method improves the visibility range in a world that is geographically distributed across several servers. Here, tall distant buildings are better visible than medium range small buildings. The ultimate goal of this work is to make the DVE scalable by lowering the information that is forwarded to each connected entity.

In our work, we are taking another solution to address the scalability problem. We are focusing on the interest management for a single server and polymorphism of our DVE middleware called DiVE. When the number of users in a DVE becomes large, the server (or servers) is not able to inform each user about

everything happening in the simulation. That is why an interest management is required. Here, users are being made aware of what is going on nearby – in the user's 'area of interest' – and are not be flooded by irrelevant information, such as events happening far away in the virtual environment. So this paper address a 'one server' scalability problem.

A lot of work has been carried out to efficiency handle the area of interest. We can divide the different techniques into two main families, the entity based methods (also called classic method), and the region based methods. Entity based methods consist in defining, for each entity a 'surrounding', i.e. an area with the entity in its center 'following' the entity. This surrounding is called 'aura' in [9].

Some research work aims at increasing the efficiency of this method (1) by modifying the shape of the area of interest or (2) by having entities be aware of each other not only when they enter the area of interest, but also when their *nimbus* intersect [9]. Although these methods are easy to implement, they suffer from a lack of scalability. The complexity of such a global check grows quickly with the number of entities. Below, we will demonstrate the complexity in $O(n^2)$ empirically. If there are n entities on the server, for each entity, the server must perform $n - 1$ checks.

The other family of methods is called 'region based method'. These methods first divide up the world into regions, and then the entities receive updates from the server according to the regions they can see. In this approach, each entity has an aura, as in the entity based method, but this aura is not used to check which entities are nearby, but which regions are nearby. It is important to note that an entity belongs to a unique region, but its aura makes it possible to see several regions.

These region based approaches are currently the most widespread method to achieve scalability. These methods are suited to the distribution of a virtual world among several servers. The research focuses mainly on the definition of optimal ways to divide the world into regions. This is done by varying the size and shape of regions [10].

The trade-off consists in specifying a region small enough to ensure the entity will not receive useless information from too distant entities, and a region large enough

- to obtain all the relevant information,
- to avoid entities moving in or out the region too often, and
- to avoid having too much regions. Indeed, the complexity of the region based method grows with the number of regions.

To address this trade off, [11] try to define the area of interest as small as possible by limiting it to the vision field. By doing so, we ensure the user only receives relevant updates. To make this approach work, the server should be aware of the characteristics of the graphical renderer. [10] tries to partition the world by using triangulation of the space and a tiles visibility algorithm. The

performance results obtained are good, however, this algorithm requires detailed knowledge about the layout of the virtual world.

Our approach in DiVE is to make the interface of the network middleware as independent as possible from any consideration about the type of application it will be used for. That is why we cannot assume anything about the graphical renderer, or the kind of maps that will be used, obstacles and wall occlusion, and so on. Therefore, we propose a method to optimize a square tile region according to a limited number of parameters the developer can choose from: (1) the area of interest of each entity, i.e., depending on the type of application, the vicinity an entity has interest in may change; (2) the maximum speed of an entity, which can also vary with the type of usage of DiVE; (3) the estimated movement pattern, e.g., in a racing game with defined tracks like *Need for Speed Underground*[7], the move pattern of entities is not the same as in a social game like Second Life, where entities are moving in a less predictable fashion. We considered these move patterns as two important categories that cover most move patterns found in simulation/video games.

2.2. Networking Frameworks

The Open Wonderland [12] DVE provides a similar feature set as Second Life and OpenSim. In addition, it offers a plugin API, and thus enables developers to easily extend the system with additional features. However, it is tied to a proprietary viewer component that lacks the visual fidelity of other recent developments. Open Wonderland gives developers more control over where physics calculations and behaviors of user objects are processed. For instance, physics that requires low-latency response can be processed client-side, whereas calculations with a high demand for synchronization can be run server-side. In this way, this approach can lower CPU load on the server.

The server-side component of Open Wonderland is a fork of Sun Research's Project Darkstar [13], dubbed RedDwarf Server[8]. It was created to continue the research and development effort after Darkstar was discontinued due to the Sun/Oracle merger in early 2010. RedDwarf is intended as a generic server infrastructure and supports dynamic allocation of hardware to scale server operations. Instead of following a simulator-centric architecture (as Second Life or OpenSim), in which each server process owns both the physics simulation work and state of a region, RedDwarf conceives of a virtual world as a collection of tasks that operate on the world state. Accordingly, RedDwarf models virtual world operations as an event-based system in which events, e.g. a client's input, trigger tasks, and tasks simulate world operations that evolve the world state. Upon receiving input from a client, a RedDwarf 'game server' (a machine that runs RedDwarf stack and the game logic) sets off a task in response to that event. The world state is stored on the data store, where a backend database service is used to keep track of individual objects. Because of the requirement

[7] http://www.ea.com/nfs

[8] http://sourceforge.net/apps/trac/reddwarf/

that all data is kept in the data store and all tasks access data through the data service, tasks become portable among game servers as the data store can be accessed by any game server. As a result, tasks can be dynamically assigned to different game servers to scale the system's operations by adding or removing resources (e.g. additional hardware).

Open Croquet [14] is an open-source DVE that focuses on collaborative applications. In contrast to the previously discussed DVEs, Open Croquet adopts a decentralized peer-to-peer architecture, where each participating host maintains a replicated copy of individual objects and applies operations to the objects to simulate and evolve the world. Croquet objects reside in islands (object containers), and the islands are replicated across hosts.

Recently, with the quick development of mobile internet devices, the networking and region partitioning problems also started to reach the physical world, with research on efficient ways to cover areas with sensors [15]. In [16], the challenge is to ensure regions are covered with a sufficient number of sensors, so that the network will be able to establish a connection even if sensors are damaged. In our case, we should split up the world into regions to maintain the connection of each users. This type of cyber-physical 'geographical sorting' is also used in smartphone mobile application like eShadow [17], making it possible to obtain information about your nearby friends. Here it is also important to devise a method to choose and dismiss other smartphone users according to their geographical position.

3. DiVE Framework

DiVE aims to achieve high scalability while remaining flexible in terms of choice of operating system and compatibility with different development platforms. For efficient low-level network operations and type serialization, DiVE relies on ExitGames' Photon network middleware.[9] Photon consists of a server-side component and a Standard Development Kit (SDK) to allow client applications communicate with the server.

The main purpose of DiVE is to efficiently share properties among defined network objects, i.e., objects that are shared across the network. In this way, DiVE maintains consistency between users in a multiuser environment. Note that an application based on DiVE must ensure that each client is using the same network objects.

DiVE has been used on several applications [18, 19, 20, 21] running with Unity3D game engine,[10]. DiVE is compatible with virtually any platform supported by the .NET and Mono common language[11] runtimes due to Photon's multiplatform capability. DiVE supports industry standard encryption of all transmissions.

[9] http://www.exitgames.com/
[10] http://unity3d.com/
[11] http://www.mono-project.com/

DiVE was initially developed to conduct transport studies with multiple drivers and traffic simulation [22]. Preliminary systematic tests with DiVE are reported in [23]. In this paper, for the first time, we conduct an major theoretical study of the area of interest method and provide a detailed technical description of DiVE's architecture and features for application developers, including some code examples.

3.1. Area of Interest Technique

In Section 2 the two main approaches were described. The first approach is the entity based method, where an area of interest (AOI), a square area (aura), is set for all entities (see Fig. 1). An entity receives updates on another entity whenever it is in its aura (area of interest). This method is easy to implement, but grows in $O(n^2)$ with the number of entities in the world.

The second approach is the region based approach, which is used in DiVE. Figure 2) shows how an entity sees the world with this method. The entity receives updates from each entity located in nearby regions, whereby "nearby" is determined by the area of interest. Each time an entity enters a region it 'subscribes' to this region's channel, i.e., it will receive all events from the region.

We implemented a Matlab code to demonstrate the difference between the two techniques.[12] In Section 3.1, we used an optimized version of the code to compare the complexity of these methods.

It is important to note that the term 'region' refers to different concepts in DiVE and OpenSim. First, in OpenSim, a region refers to an actual server, and a grid is a federation of several regions. This, for instance, implies a slight loading time when you move from one region to another. In Opensim, the region determines to which server a networks object such as 'prim' (primitive object) or an avatar (graphical self-representation of a user) belongs to. In DiVE, by contrast, the concept of region is purely a server concept. Here, the clients are not aware of the regions they belong to.

Second, in OpenSim, the client loads objects (collections of prims) within its vision range, possibly including objects from an adjacent region. Differently in DiVE, the area of interest is used to determine which region should be loaded, as shown in Fig. 2.

In Fig. 2, the current entity's area of interest intersects with Regions 5, 6, 9 and 10. Therefore, it receives updates from all entities located in these regions. We can say this crosshatched group of regions is a *dynamic area of interest* of the current entity as it changes when entity is entering/leaving (subscribing/unsubscribing) regions. The strong aspect of this implementation is that while an entity belongs to a unique region, the entity's area of interest allows it to see several regions. In this way, we keep the number of checks small, while avoiding

[12] https://www.dropbox.com/s/2b3ncmhdftihvw3/area_of_interest.zip?v=0mcn

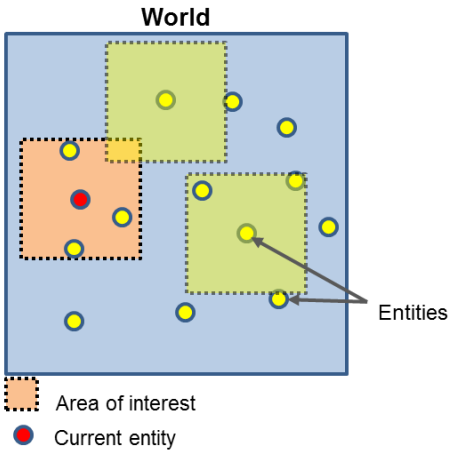

Figure 1: Basic area of interest technique. Here, each entity 'sees' entities within a specified, indicated by a dotted square [23].

the situation, where an entity is at the region border and sees other entities suddenly appear or disappear within very close range.

To illustrate the difference of complexity with entity based and region based approaches, we implemented both methods in MATLAB.[13]. The number of regions has been fixed to sixteen. A run consists of moving the entities pseudo randomly for 10,000 iterations and for each frame, the interest management algorithm is activated. We used the total time of a run to compare the efficiency (as dependent variable). Pseudo random refers to a move pattern where (1) the entity is moving according to a randomly chosen direction and deviates slightly in a random fashion (according to a Gaussian centered on 0 deviation), and (2) a new direction is forced if the entity reaches the border of the world. This move pattern is close to the Brownian move with a fixed speed.

The number of entities was varied from 10 to 1300. Fig. 4 indicates the quadratic complexity of the basic technique as opposed to linear complexity of the region-based approach.

One possible issue of this approach is the overload that is generated if an entity is oscillating between two regions. In this case, the entity enters and leaves the region repeatedly within a short time, thereby subscribing and unsubscribing to the region channel. The repeated forwarding of updates to the entities constitutes a redundancy in communication.

We addressed this problem by adding a hysteresis mechanism, i.e., a trigger with a moving threshold, comparable to a Schmitt trigger in electronics [24]. A Schmitt trigger is used to protect a mechanical actuator. If an electronic command oscillates too much, the demand on an actuator can require it to go

[13]http://www.mathworks.co.jp/

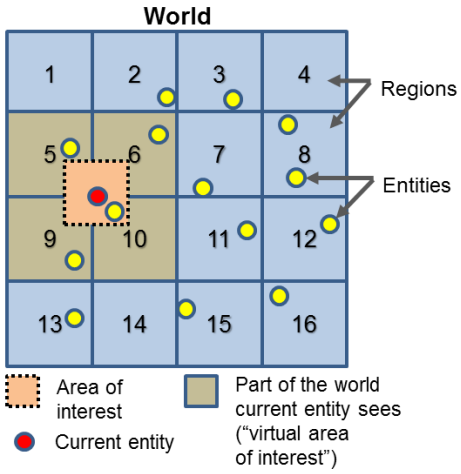

Figure 2: Region-based area of interest technique. Here, each entity 'sees' entities of all regions its interest area (indicted by dotted square) intersect [23].

back and forth too fast and consequently damage it. The Schmitt trigger can eliminate such undesirable effect of small oscillations.

Our specification of area of interest follows a similar principle. Here, an entity has two areas or interest, the 'inner' and the 'outer' area. Fig. 4 shows the demand on updating the server with a standard area of interest (Case (a)) , and with the double area method (Case (b)). In Case (a), the entity enters (subscribes to) or leaves (unsubscribes from) a region each time the area of interest intersects the region. In Case (b), the entity's inner area of interest has to enter the region to belong (subscribe) to the region, whereas the entity's outer area of interest has to exit the region to notify the server that the entity left. This method prevents from flooding the server with subscribe/unsubscribe region requests. To set the overlap size of the inner and outer area of interest, one has to consider the maximum speed of an entity. DiVE's update rate is 10 times per second (every 100ms). To avoid subscription events at every DiVE tick, the overlap should be larger than 0.1 $\times MaximumSpeed$.

3.2. Region size

The region based approach enables to make the number of AOI checks independent from the number of entities on the server. However, now the complexity has been moved to the regions; the more regions we have, the higher the complexity is.

The challenge of a DVE is two-fold. First, to limit the number of messages exchanged by entities; an entity should receive the updates of entity from a relevant but confined area. Second, to limit the complexity of research method; here the region based model makes it possible to sort entities geographically. In this case, limiting complexity means limiting the number of regions.

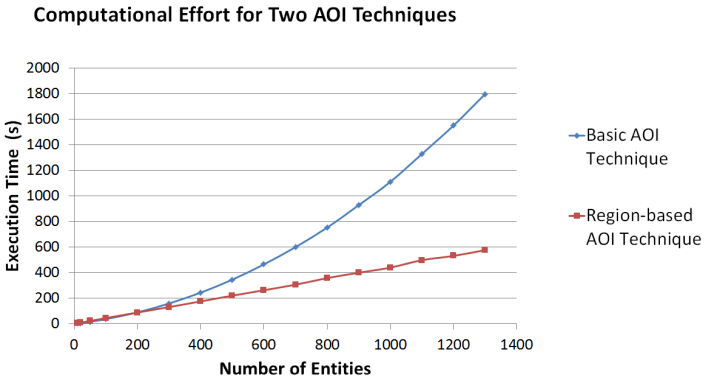

Figure 3: Comparison of basic vs. region-based area of interest techniques.

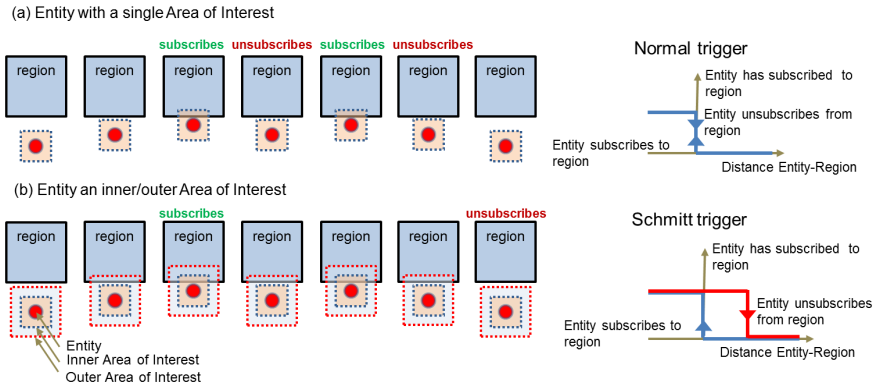

Figure 4: Comparison of event triggering between a single and a double area of interest.

So we are focusing on this part on varying the parameters of a MATLAB simulation, to study the function

$$(t, messages, enter\_leave) = f(AOI\_REG, Move\_pattern)$$

where the output elements are:

- t: the execution time of the simulation (N iterations of M entities, moving and receiving updates from each other every iterations).
- $messages$: the number of exchanged messages per frame.
- $enter\_leave$: average number of times an entity has entered/left a region

We continue with explaining the parameters.

AOI_REG: We decided to proceed as follows. In a given multiuser application, we started with considering what neighborhood (surrounding) an entity

have interest in, e.g. 100m^2. So we chose choose 100m^2 100 as the AOI size. Afterwards, the problem consists in choosing what region size suits this AOI size.

To provide a generic way to choose the region size, we decided to use the ratio "AOI size / region size". So first we defined the AOI size, and then we varied the size of the regions, and consequently the number of regions, to find an optimum.

Move_pattern: We also decided, to reflect different entity behavior, to implement different move pattern. We use a pseudo-random move pattern, and a 'point of interest', or 'checkpoint' move pattern. Pseudo random is the move pattern explained in Section 3.1. The checkpoint move pattern is simpler. All entities share a same checkpoints list, choose randomly one of these points, go to the checkpoint and start over again the operation.

The experiments allowed us to make some interesting observations. Figure 5 shows the execution time of simulations (in seconds) for several ratios of AOI size and region size. If there are few large regions, the AOI will intersect few regions, but still, the entity will see much of the world, and hence will have to update information on many entities. So, too large regions lead to significant complexity in terms of execution time. On the other hand, if we have many small regions, the number of checks on which updates have to be performed increases, and therefore the complexity is growing. The curve is showing a low complexity between 2 and 4. Therefore, if we require to have entities with an AOI size to x, and choose a region size of $x/2$.

We can also observe that the complexity is different depending on the move pattern. The checkpoint move pattern creates a higher load on some parts of the virtual world as the entities tend to be concentrated around the checkpoints, whereas the pseudo random move pattern tends to scatter the entities evenly on the server.

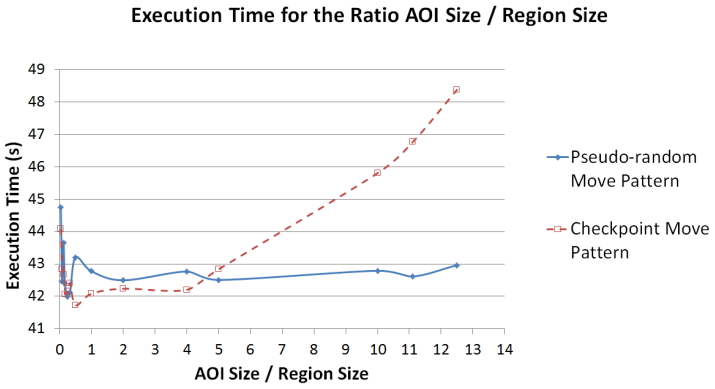

Figure 5: Processing time of the simulation, where entities move around in the virtual environment according to the number of regions (AOI is fixed).

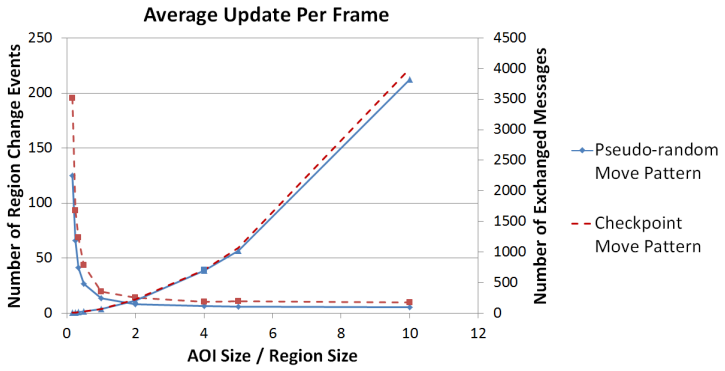

Figure 6: Number of exchanged messages and number of region change events (entering/leaving regions) according to the size of the regions (AOI is fixed).

Figure 6 shows the number of messages exchanged per frame and the number of "entering/leaving region" events for several fractions of AOI size and region size. We see that if there is a few large regions, entities see each other frequently and more information has to be shared than with smaller regions. On the other hand, with a higher number of smaller regions, the number of "entering/leaving region" events is higher, which means more communication on the server. Again, we notice that the a fraction around 2 is a good compromise. Also, the checkpoint move pattern is slightly more computationally demanding than the pseudo random move (because entities concentrate in certain parts of the virtual world). For our experiments in Section 5, we will use the checkpoint move pattern as a reference.

3.3. DiVE Tools

In the following we describe the tools provided by DiVE.

3.3.1. Generic Type Conversion

Data sharing relies on Photon's data serialization mechanism. While the number of types supported by Photon is limited, DiVE lets application developers create and register custom type converters to convert arbitrary data types used in the application. Subsequently these converters are triggered automatically by DiVE whenever an object of that type is synchronized over the network.

3.3.2. Generic Network Event Propagation

In addition to the object-based continuous data sharing mechanism, DiVE also supports application-specific events. Conceptually, events differ from entities in that using an Entity has the goal of synchronizing between clients objects which can be updated frequently, while events are messages sent sporadically. These events encapsulate arbitrary objects courtesy of Google's Protocol

Buffers[14] library and can thus be used to implement simple string forwarders (e.g. for a chat function) or complex RPC components. Efficient serializers are automatically generated at compile time.

3.3.3. Persistency

These serializers can also be used to store generic objects in a database embedded in DiVE's server component, such as user information, settings, virtual world inventories, and so on.

3.3.4. User Authentication

User authentication can be performed with a proprietary backend or arbitrary OAuth 2.0[15] compliant services. It also supports user management operations such as 'kicking' and 'banning'.

3.3.5. Logging

A standalone .NET application functions as a generic logging client for DiVE. In order to log the data from an application, developers must create a setup file that works as a plugin to the logging. This setup determines which entities and events will be logged and how. The logging client creates an entity with maximum area of interest (monitor entity) which can see all entities in the world. From the server point of view, the logging client is just a usual client application.

4. DiVE Architecture

The DiVE architecture has been designed as a flexible framework that supports easy implementation of multiuser applications. DiVE supports both Microsoft and Mono frameworks, and hence it supports .NET and Mono based applications.

DiVE is divided into two parts: the client dll and the server dll. Fig. 7 shows the DiVE architecture. The server component contains 4451 C# SLOC (Source Lines of Code), distributed among the three dlls: main application logic, persistency and area of interest management. The Client component contains 2342 C# SLOC in a single dll. Both server and client components depend on a 'common' dlls with shared types, which contains 226 C# SLOC.

On the client side, the client dll is responsible for the synchronization of the client application with the server. The developer decides application logic, design and graphics choices. To interact with the client application, the client dll handles three components:

- *NetworkManager*. Exposes DiVE's main API, with functions such as connecting and disconnecting, registering to an entity type, sharing and destroying an entity, and setting up areas of interest.

[14] https://code.google.com/p/protobuf/
[15] http://oauth.net/2/

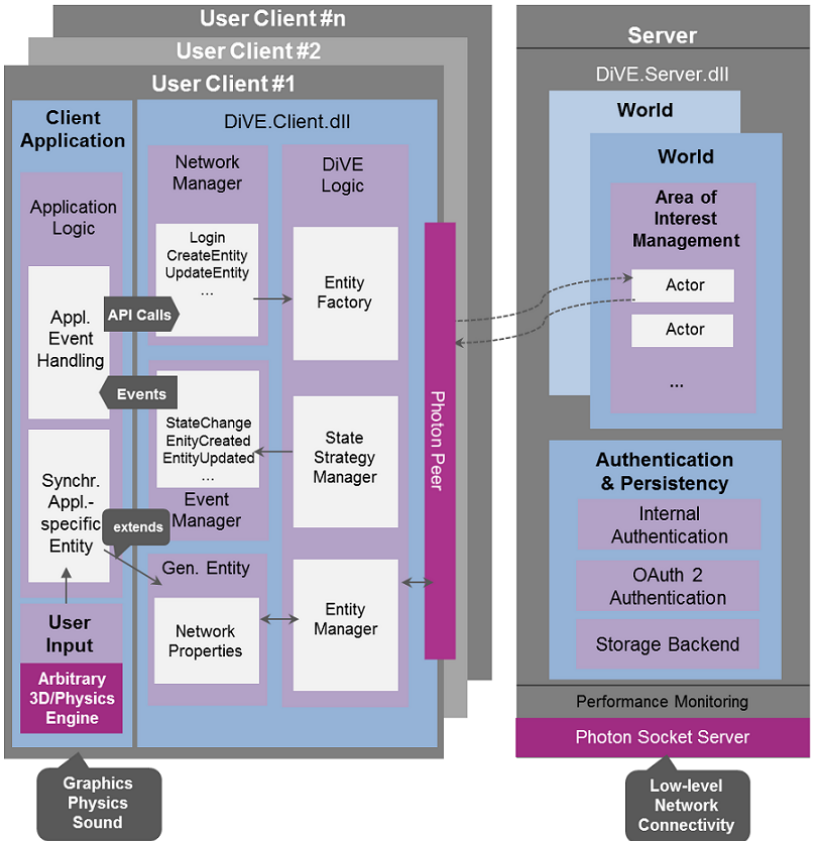

Figure 7: An overview of DiVE's client and server components.

- *EventManager*. Provides a set of built-in events that the client application can listen to and that are triggered by the server. It is through the event manager that a client application knows when a new remote entity is created or destroyed, or when it is out of the sight (outside the interest area).

- *Entity*. Base class of any object that needs to be synchronized over the network. Client applications must define their own entity types, such as a CarEntity or AvatarEntity, and inherit from the *Entity* class. Local entities are shared via the *NetworkManager* and remote ones are received via the *EventManager*.

After establishing a connection and authenticating via the *NetworkManager*, a client application must 'register' each of their own defined entity types with the server. Registering an entity type (e.g. the *CarEntity* type), literally means telling the server that the client knows that type and is interested in sharing and/or receiving instances of that type. After registering, the client is able to

share and receive entities of that type. The basic property of the *Entity* class is the *position* (a 3D vector), which is used by the area of interest algorithms. All other attributes that need to be shared on the server must be declared as *network properties*. After an entity instance is shared, its network properties are automatically shared via the server and updated every 0.1 seconds.

Entities should not only share their properties over the network, they should also be aware of other entities and their surrounding world. For a client application to receive updates of remote entities, it needs to setup an area of interest for its own entities. If a client application owns more than one entity, the areas of interest add up in a union fashion. To know when a remote entity was created or destroyed, a client must subscribe to events defined in the *EventManager*. For example, an 'entity created' event is always triggered by the server and sent to all clients who are registered to that entity type (and according to the area of interest algorithm). It is up to each client application to listen to this event and handle it.

Besides the built-in events from the EventManager, a client application can also define their own custom application-specific events (e.g. a chat message). Fig. 8 shows how clients subscribe to events by registering their custom event type on the server. When a client triggers an event, it is broadcast over the network to all clients who registered to the same event type.

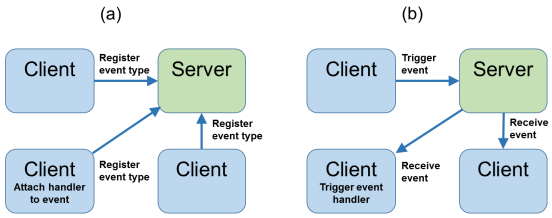

Figure 8: (a) Clients register custom event types and attach handlers to it; (b) a client triggers an event on the server, the server broadcasts it, and the event is captured by other clients.

There is also the possibility for the client to create persistent objects. A persistent object is a serializable type whose state needs to be kept after the client goes offline. An example is inventory items such as virtual money. These objects will be stored on the server database.

The client application is responsible for managing the interaction with the user such as user inputs or graphic rendering, and does not have to handle network synchronization issues. DiVE does not make a difference between a user client and a simulator-type client. That is, DiVE allows to integrate special-purpose simulators, such as traffic or pedestrian simulators, as clients. Those clients may control several entities (e.g. vehicles or pedestrians), while the user controls a single avatar. The DiVE.Client.dll is used for both kinds of clients.

On the server side, the DiVE.Server.dll has to manage the virtual world. First, it has to handle the authentication of the connection request from the DiVE client. In the case of internal authentication, the Server uses a given database of users to allow client applications to connect. When using OAuth

authentication, the client application forwards the OAuth token received by the respective service(e.g. Facebook). Using this token, the server is able to retrieve information such as a user id and name, to be used within the application.

Once a client is authenticated, a respective 'Actor' is created in the server side. An 'Actor' is the representation of a client in the server and contains the list of entities that belong to its corresponding client. The area of interest algorithm in the server runs through actors and their entities to decide which clients will receive updates for which entities. In sequence, it sends messages to the *EventManagers* in each client with property updates, creation and destruction of entities, which, in turn, triggers events to the applications. Finally, it also manages the storage of persistent objects.

4.1. DiVE Code Examples

We provide code examples for two reasons. First, we want to show that it is quite easy to use DiVE to implement an application. As DiVE handles most of the network issues, the application developer only has to care about which element should be shared on the server. Second, we want to appeal to programmers and provide complementary information to Fig. 7.

The first part of the code example shown in Fig. 9 are the type definitions. In the beginning we define the class *ExampleEntity* that inherits from the DiVE class *Entity*. This means that this application deals with entities of the *ExampleEntity* type and by registering it, the server will forward to this application information from other entities of the same type in the network. *ExampleEntity* has the following properties: *Position* (inherited from *Entity*), *Rotation* and *Name*. The *NetworkProperty* fields are shared with the server every 0.1 seconds. The field *notSharedValue* is local to that client and will not be updated by DiVE or shared with the server as it is not a *NetworkProperty*.

Next, we define the class *CustomEvent*, which holds a custom event defined by the application. This example event contains a string as an argument (e.g. a broadcast chat message) that will be sent to every client that register to the *CustomEvent* type. The argument of the event could be any object serializable by Google's Protocol Buffer.

The code example shown in Fig. 10 relates to application initialization. After instantiating the *EventManager* and the *NetworkManager*, we connect to the server by using the *NetworkManager*'s Connect() method. Since this is asynchronous, we need to subscribe to a built-in event in the *EventManager* to receive the result of the connection attempt. Once we are connected, we register our *ExampleEntity* and *CustomEvent* types. The next step is to create and share our instance of *ExampleEntity*, via the *NetworkManager*'s EntityCreated() method. To listen to our MessageSent event, we attach a local handler to it. This local function will be called by the event manager as soon as another client triggers this event.

```
[Serializable]
public class ExampleEntity : Entity {
 public NetworkedProperty<float[]> Rotation { get; private
     set; }
 public NetworkedProperty<string> Name { get; private set; }
 private int notSharedValue;
 public ExampleEntity()
 {
  Position.Value = Vector3.one;
  Rotation = new NetworkedProperty<Vector3>(new Vector3());
  Name = new NetworkedProperty<string>(string.Empty);
 }
}
public class CustomEvent
{
  public NetworkEvent<string> MessageSent { get; private
      set; }

  public CustomEvent()
  {
      MessageSent = new NetworkEvent<string>();
  }
}
```

Figure 9: Code example showing how to easily create a network entity with DiVE. Each property declared as NetworkProperty will be shared. An example of customized network event is also given.

5. Evaluation

5.1. Informal Testing of DiVE

DiVE has already been successfully deployed in a variety of applications that use DiVE to synchronize multiple users. While those tests have been performed with human users (as clients), there was no intention or opportunity for systematic testing.

- Forty-one students from Hiroo Gakuen Junior and Senior High School participated in a driving behavior study on eco-safe driving practice [18].
- Four user drivers and one hundred traffic simulator controlled cars were used to study traffic congestion caused by the 'rubbernecking' effect [20].
- Up to forty human travelers participated simultaneously in studies on driving and travel behavior under emergency evacuation conditions [19].
- Three human drivers and twenty-seven computer-controlled 'opponent' cars were involved in a multiuser eco-driving training study employing distributed constraint optimization to control the opponent cars [21].

```
public void Main(){
  _evtMgr = new EventManager();
  _netMgr = new NetworkManager(_evtMgr);

  _eventMgr.OnGameStateChange += GameStateChanged;
  _netMgr.Connect("localhost:4040"); // server IP
}

private void GameStateChanged(GameState state){
  if(state == GameState.WorldEntered){
  _netMgr.RegisterEntityType<ExampleEntity>();
  _custom = new CustomEvent();
  _netMgr.RegisterEvents<CustomEvent>(_custom);

  _myEntity = new ExampleEntity();
  _netMgr.EntityCreated(_myEntity);

  _custom.MessageSent.Event += MessageEventHandler;
  }
}
```

Figure 10: Code example showing how to instanciate the network classes. With the 2 code examples given, we created a network entity, sharing its name and rotation on the server.

5.2. Systematic Testing of DiVE

5.2.1. Background

Several metrics can be used to assess the performance of a networking framework like DiVE. Previous work on the architecture of virtual worlds and on scalability used metrics like CPU usage, frames per second, bandwidth, latency and throughput [25, 7, 5]. Varying the number of clients and checking the value of these metrics allows us to determine how well the system handles an increase of load. Importantly, such tests can tell us at what point the user experience gets affected. Since we are dealing with interactive virtual worlds inhabited by users, it is important to identify the number of clients where the system falls below a frame rate of 30, the minimum frame rate for smooth interaction [26].

[25] proposed the Distributed Scene Graph (DSG) architecture, which allows scaling of actuators (processes that change the data in the scene like physics systems and clients) independently of the data (the scene itself). The main idea behind this architecture is that scalability requires that different types of processes need to be handled differently. To demonstrate the validity of their approach, the authors created a test environment with (1) a scene server holding the data of the scene, (2) five client managers, and (3) five machines with multiple clients. The client managers are processes whose main responsibility is to manage clients, which is one possible solution for scaling communication intensive processes.

The authors compared the results by running two versions, without and with

the client managers. In the version using the client managers, the frame rate begins to drop with 400 clients. They showed that because there is more time spent in communication, there is also less time to handle the physics simulation. With client managers they were able to connect 750 clients before the frame rate dropped below 40 FPS. This result indicates (1) that the architecture enables the application of independent scalability techniques, evidenced by the ability to scale the clients without changing the physics simulation and (2) that scaling one part of the system has a positive impact on the remaining parts. This is expected because if we need less time to compute one part of the system (like communication), we have more time available to compute the remaining parts (like physics simulation) even if we do not scale them.

DiVE is not aimed at scaling different classes of processes because in our case, each client is responsible for rendering, physics and application logic. Our solution is targeted at scaling only the communication between processes. For that reason, we will focus on measuring how DiVE affects the rate of communication between clients and server, while keeping in mind the consequences this might have in remaining computer resources.

Using a naive approach, every entity in a virtual environment is updated about every other entity. We will refer to this approach as the 'global model'. This means that whenever an entity generates information it sends a message to the remaining $n-1$ entities ($O(n(n-1))$ complexity). By using DiVE's 'area of interest' notification system, we are able to reduce the complexity and consequently the amount of communication in the network. We will refer to this technique as the 'local model'.

5.2.2. Experimental Setup

We tested the impact of different sizes of areas of interest (with fixed region size) and number of clients on the following metrics (see also [25]):

- Bandwidth as an indicator of the amount of communication;
- Frames per second (FPS) as an indicator of the smoothness of user experience;
- CPU and RAM usage as an indicator of the impact of our algorithm in the server machine's resources.

Testing a DVE using only numerical metrics might be insufficient. Multiuser applications such as simulators or games have to provide a feeling of smoothness that is not sufficiently captured in the parameters described above. To ensure that our application is not lagging we decided to implement a simple 2D graphical interface that allows to observe the smoothness of the application. Figure **??** shows a screenshot of the test environment. We want to assess DiVE's ability to broadcast information efficiently. Therefore, we set up our experiment in a local network (LAN). If a lag is noticed, it can be attributed to DiVE and not to a network latency.

In summary, our test software consists in (1) simulating clients and (2) rendering the client's position on a map, and (3) rendering the positions of the

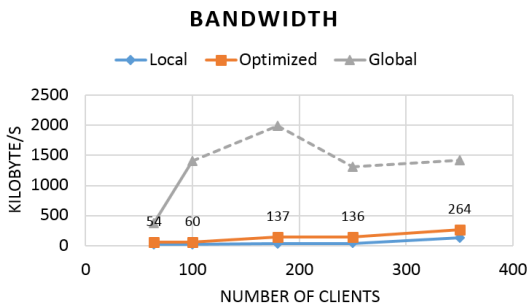

Figure 11: Comparison of bandwidth usage with the local and global models.

other clients in their AOI. To increase the realism of the client, the program simulates some computation (looping 1000 times) for each client.

In our tests we used a world with a 2400m×2400m area with a fixed region size of 60m×60m; so there are $40 \times 40 = 1600$ regions. Then we placed the entities evenly according to their number, and moved them according to the checkpoint pattern, as described in Section 3. The checkpoint move pattern makes entities move inside the virtual world. As entities will typically gather around the checkpoints, we provide a sufficient number of checkpoints (50) to avoid a too high concentration of entities in some parts of the virtual world.

Each client has exactly one entity. As we have seen previously, the checkpoint move pattern is slightly more heavy on the server than a pure random move pattern. However, we have chosen it because we consider this kind of move is more close to the movement pattern in multiuser virtual worlds. With an increasing number of clients or with a larger area of interest, more entities intersect each other's area of interest, which leads to an increased traffic on the network.

An ideal test scenario would use one networked machine for each client. Because this is impractical for a research lab, we devised a different approach. We use 10 different machines, each one able to connect to up to about 50 clients. The machines were running on Windows 7, and were equipped with an Intel core I7. One 11th machine was running the DiVE server while the other ten machines where running clients in parallel. The client load was divided evenly by all machines, e.g., for a scenario with 200 clients, each of the machines would host 20 clients. Our results indicate that even though running several clients in parallel in one machine affects the outcome, it is still a good approximation of the trend we would obtain if the clients were running in different machines.

5.2.3. Tests

We measured the bandwidth in three runs with different sizes of areas of interest and with an increasing number of clients.

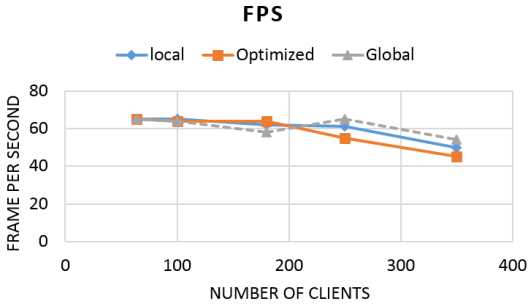

Figure 12: Comparison of FPS (client) with local and global models.

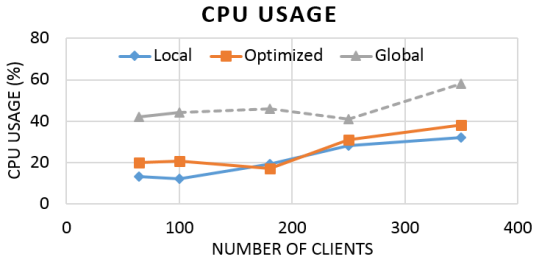

Figure 13: Comparison of CPU usage (server) with local and global models.

- *Local AOI*: we use a 'local' area of interest, 1m×1m.[16] This size is smaller than the region size, so the clients basically see only what is in their region.
- *Optimized AOI*: we use an 'optimized' area of interest according to our proposal in Section 3.
- *Global AOI*: here each client sees all other clients; there is no area of interest management.

Bandwidth: Figure 11 shows the results for bandwidth. As expected, we obtain a weakly ascending line: the required bandwidth increases with the number of clients. Moreover, as we have seen in Section 3, the growth is linear and not quadratic. Finally, the difference between using or not using interest management is obvious here, for 100 clients, we decrease the bandwidth usage from 1200 to 60kb/s. Unfortunately, the dashed part of the 'global' curve is not reliable. Indeed, above 120 clients, we started to observe strange behavior when using a global area of interest, including unexpected client disconnection, and important update loss, resulting in a very obvious latency on our 2D interface.

[16]Note that the client can still see the whole region where the AOI is located in.

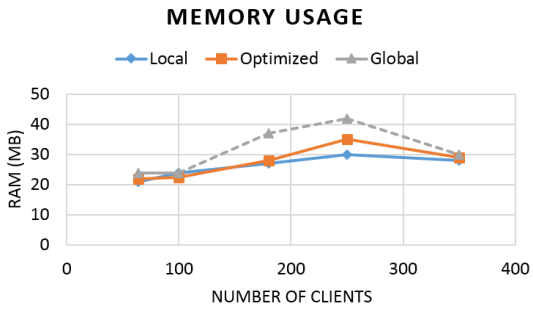

Figure 14: Comparison of RAM usage (server) with local and global models.

Frames per second (FPS) on the clients: We also wanted to test how the client would react to the growth of clients on the server. That is why we assessed the FPS on the CLIENT side. Figure 12 illustrates that we succeed to keep the FPS high enough (above the 30 fps threshold). One more time, we cannot rely on the 'global' curve, insofar as above 120 clients, the connections were not reliable.

CPU and RAM usage on the server: As shown by [25], scaling one component of the system can also have an impact on the remaining ones. In our study we wanted to test whether our algorithm would overuse the computational resources on the server. Fig. 13 and Fig. 14 indicate that CPU and RAM evolution with the number of clients. The memory usage is not affected a lot by the size of the AOI, but more affected by the number of clients.

It is interesting to note that the connection loss we observed above 120 client connections is not related to an CPU or memory overuse, but to a connection overload. The maximum number of clients in one region is the same as the maximum number of clients with a global area of interest. We started to have connection drops above 120 clients.

6. Conclusions

In the paper, we described DiVE, a simple and scalable networking framework for distributed virtual environments. DiVE is a practical framework that has already been used to implement networked multiuser applications in eco-driving [18, 21], traffic accidents [20], and evacuation [19].

The focus of the paper is scalability on a single-server basis. For that purpose, a region-based area of interest technique was used and tested. First, we could demonstrate the gain from using a region-based area of interest such that the weight cost increases linearly with the number of entities, rather than quadratically. Importantly, we determined an optimized (relative) value for region size and area of interest size. The application developer can select the optimum region setting by defining the area of interest size as x, and then having the size of the region as $x/2$.

Second, the empirical evaluation showed most promising results for the bandwidth metric. At the level of 100 users, our method outperforms a system without area of interest (AOI) by 20 times. The results also indicate that using an AOI reduces the server CPU usage about 50%. Finally, the implementation of a simple 2D interface made it possible to visually assess the latency of remote clients. Our (non-systematic) observations indicate that the AOI technique supports smooth moving of remote entities.

Currently in DiVE, the size of the regions is fixed. To further improve scalability, we could change the number and size of regions on the server according to the number of connected clients. One solution could be to develop an algorithm spotting automatically where the checkpoints area are, and partition the world accordingly.

Acknowledgments

We would like to thank Martin Lindner for implementing several components of DiVE. This work was partly supported by the 'Global Lab' NII Grand Challenge grant, a Kiban Kenkyu B grant from the Japan Society of the Promotion of Science (JSPS), and by FCT (INESC-ID multiannual funding) through the PIDDAC Program PEst- OE/EEI/LA0021/2011.

[1] E. Frecon, M. Stenius, Dive: a scalable network architecture for distributed virtual environments, Distributed Systems Engineering 5 (3) (1998) 91–100.

[2] H. Liu, M. Bowman, F. Chang, Survey of state melding in virtual worlds, ACM Computing Surveys 44 (4) (2012) 1–25.
URL http://dl.acm.org/citation.cfm?doid=2333112.2333116

[3] J. Lui, M. Chan, A efficient partitioning algorithm for distributed virtual environments, IEEE Transactions on Parallel and Distributed Systems 13 (3) (2002) 193–211.

[4] C. Lopes, Hypergrid: Architecture and protocol for virtual world interoperability, IEEE Internet Computing 15 (5) (2011) 22–29.
URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5871570

[5] H. Liu, M. Bowman, Scaling virtual worlds: Simulation requirements and challenges, Proceedings of the Winter Simulation Conference (2010) 778–790.
URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5679112

[6] D. Horn, E. Cheslack-Postava, T. Azim, M. J. Freedman, P. Levis, Scaling virtual worlds with a physical metaphor, Pervasive Computing (2009) 50–54.

[7] D. Horn, E. Cheslack-Postava, B. F. Mistree, T. Azim, J. Terrace, M. J. Freedman, L. Philip, To infinity and not beyond: Scaling communication in virtual worlds with Meru, Tech. rep., Technical Report, Stanford University (2010).

[8] E. Cheslack-Postava, T. Azim, B. F. Mistree, D. R. Horn, J. Terrace, P. Levis, M. J. Freedman, A scalable server for 3D metaverses, in: Proceedings of USENIX Annual Technical Conference (ATC'12), 2012, p. 20.

[9] S. Benford, L. Fahlén, A spacial model of interaction in large virtual environments, in: Proceedings 3rd European Conference on Computer-Supported Cooperative Work, 1993, pp. 109–124.

[10] J.-S. Boulanger, J. Kienzle, C. Verbrugge, Comparing interest management algorithms for massively multiplayer games, in: Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games (NetGames'06), ACM Press, 2006, p. 6.

[11] N. D. G. Mojtaba Hosseini, Steve Pettifer, Visibility-based interest management in collaborative virtual environments, in: Proceedings of the 4th International Conference on Collaborative Virtual Environments, ACM Press, 2002, pp. 143–144.

[12] J. Kaplan, N. Yankelovich, Open Wonderland : An extensible virtual world architecture, IEEE Internet Computing 15 (5) (2011) 38–45.

[13] J. Waldo, Scaling in games and virtual worlds, Communications of the ACM 51 (8) (2008) 38–44.

[14] D. A. Smith, A. Kay, A. Raab, D. P. Reed, Croquet – a collaboration system architecture, in: Proceedings of the First Conference on Creating Connecting and Collaborating Through Computing, ACM, 2003, pp. 2–9.

[15] Y. Gu, Y. Ji, J. Li, B. Zhao, Covering targets in sensor networks: From time domain to space domain, IEEE Transactions on Parallel and Distributed Systems 23 (9) (2012) 1643–1656.

[16] S. He, J. Chen, P. Cheng, Y. Gu, T. He, Y. Sun, Quality of sensing, sensor allocation, sensor relocation, distributed algorithms, IEEE Transactions on Parallel and Distributed Systems 23 (9) (2012) 1657–1667.

[17] J. Teng, B. Zhang, X. Li, X. Bai, D. Xuan, S-Shadow: Lubricating social interaction using mobile phones, IEEE Transactions on ComputersPreprint.

[18] M. Madruga, H. Prendinger, T. Tilma, M. Lindner, E. Santos, A. Nakasone, Practicing eco-safe driving at scale, in: Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI'12), 2012, pp. 2147–2152.

[19] M. van den Berg, E. Doirado, R. van Nes, H. van Lint, H. Prendinger, S. Hoogendoorn, Application of a 3d multi-user environment for research on travel choice behavior. the case of a tsunami, in: Proceedings 13th International Conference of the International Association for Travel Behaviour Research (IATBR'12), 2012.

[20] K. Gajananan, A. Nantes, M. Miska, A. Nakasone, H. Prendinger, An experimental space for conducting controlled driving behavior studies based on a multiuser networked 3d virtual environment and the scenario markup language, IEEE Transactions on Human-Machine Systems 43 (4) (2013) 345–358, the paper was recommended for publication in the former IEEE Transactions on Systems, Man, and Cybernetics – Part A: Systems and Humans.

[21] M. Madruga, H. Prendinger, iCO2: Multi-user eco-driving training environment based on distributed constraint optimization, in: Proceedings 12th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'13), 2013, pp. 925–932.

[22] H. Prendinger, M. Miska, K. Gajananan, A. Nantes, A Cyber-Physical System simulator for risk free transport studies, Computer-Aided Civil and Infrastructure EngineeringIn press.

[23] H. Prendinger, J. Oliveira, J. Catarino, K. Puntumapon, M. Madruga, R. Prada, iCO_2: a networked game for collecting large-scale eco-driving behavior data, IEEE Internet ComputingIn press.

[24] O. H. Schmitt, A thermionic trigger, Journal of Scientific Instrument 5 (1938) 24–26.

[25] D. Lake, M. Bowman, H. Liu, Distributed scene graph to enable thousands of interacting users in a virtual environment, in: Proceedings of the 9th Annual Workshop on Network and Systems Support for Games (NetGames'10), 2010, pp. 1–6.

[26] S. Kumar, J. Chhugani, C. Kim, D. Kim, A. Nguyen, P. Dubey, C. Bienia, Y. Kim, Second Life and the new generation of virtual worlds, IEEE Internet Computing 41 (9) (2008) 46–53.