2010

# XPU: A Distributed Architecture for Metaverses

Francis Chang
*Portland State University*, chang.francis@gmail.com

C. Mic Bowman
*Intel*

Wu-chi Feng
*Portland State University*

# XPU: A Distributed Architecture for Metaverses

Francis Chang
Department of Computer Science
Portland State University
Portland, OR
francis@francischang.com

C. Mic Bowman
Intel Labs
Intel Corporation
Hillsboro, OR
mic.bowman@intel.com

Wu-chi Feng
Department of Computer Science
Portland State University
Portland, OR
wuchi@cs.pdx.edu

## ABSTRACT

*A significant problem of designing 3D virtual worlds (such as metaverses) is developing a scalable architecture that can manage millions of simultaneous users in an interactive 3D environment. This paper presents XPU (Extremely Partitioned Universe), a hierarchical client-server architecture for developing highly scalable metaverses. This design addresses the problem of dynamically partitioning the world to manage network and computing resources.*.

**Keywords**: Metaverse architecture, 3D Virtual Worlds, Extremely Partitioned Universes, Massively Multiplayer

## 1. INTRODUCTION

Virtual reality systems [aw,croquet,sl,os] have risen in popularity with readily available high-speed networking and affordable consumer computer graphics processing hardware.

This paper focuses on metaverses – a shared 3D virtual space in which people can interact and communicate through virtual avatars. Unlike massively multiplayer online games (MMOGs) which strive to simplify their universe to optimize their implementation for a specific game environment, metaverses are characterized by a generalized approach to the problem of 3D worlds. These designs seek to promote unconstrained user-generated content for services such as social networking and collaboration, scientific experimentation, e-commerce, marketing and gaming.

The unconstrained nature of metaverses requires a different style of architecture to manage computing and networking resources than online gaming.

Content in today's virtual worlds is designed to accommodate the limitations of the architecture. (For instance, this might mean limiting the maximum number of active users in a single virtual location.) Future virtual world architectures should be able to dynamically adapt to the content in the world and the workload of the world, so that virtual experiences can be designed without arbitrary architectural limitations.

This paper introduces XPU (Extremely Partitioned Universe), an architecture designed with the goals of managing 3D virtual space and content in a client-server situation, using a spatial subdivision algorithm to manage a distributed architecture. This paper focuses on the server components of managing objects in a virtual world simulation.

In the following section, we describe the goals of our architecture. Related virtual world architectures are presented in Section 3. Section 4 outlines the basic structure of our proposed virtual world. Section 5 introduces different algorithms that can be used to support the architecture. Section 6 examines how these algorithms perform in an experimental setting. Section 7 discusses future work related to XPU, while Section 8 concludes this paper.

## 2. DESIGN GOALS

The following are design considerations for our architecture:

- The design must be a client/server architecture. In this way, the service provider can guarantee security, availability and adequate resource provisioning.
- Storage and computing power is large, but no single computer can handle the computing load.
- The population is large, and unpredictable. The architecture must accommodate flash crowds as well as vast unused or sparsely populated spaces.

It is the goal of XPU to be an architecture for metaverse-like entities and to be a foundation for all types of MMO virtual simulations including online gaming and 3D social networks.

### 2.1 Elements Not Part of XPU

In designing any large multi-user system there are many architecture constructs that are only weakly tied to the problem of managing 3D virtual spaces. Components such as asset storage, user profiles, authentication, exploit detection, domain administration and instant messaging are not discussed in this paper. These problems can be addressed in by more general system solutions that are not encumbered by the constraints of managing a metaverse-style universe.

The XPU architecture only addresses the problems of managing 3D virtual space and the objects contained within.

## 3. RELATED WORK

There are many examples of massively multiplayer virtual spaces that each have distinct solutions to the problem of managing vast virtual spaces that need to service a high number of simultaneous clients.

In MMOGs, sharding is a popular approach to broadly partition the user base into disjoint copies of the world. In this model, replication is easy because users belonging to one shard cannot interact with users in other shards [uo,wow]. Load balancing is accomplished by restricting the number of simultaneous users in a shard. In these environments, only a minimal amount of functionality is placed at the server to allow them to scale up. For instance, generalized physics and dynamic content are usually omitted.

Croquet [croquet] is a decentralized approach to the problem of virtual spaces relying on a peer-to-peer synchronization protocol to distribute the contents of the virtual space. A single croquet instance can become congested with many simultaneous users since there is no mechanism to subdivide existing space.

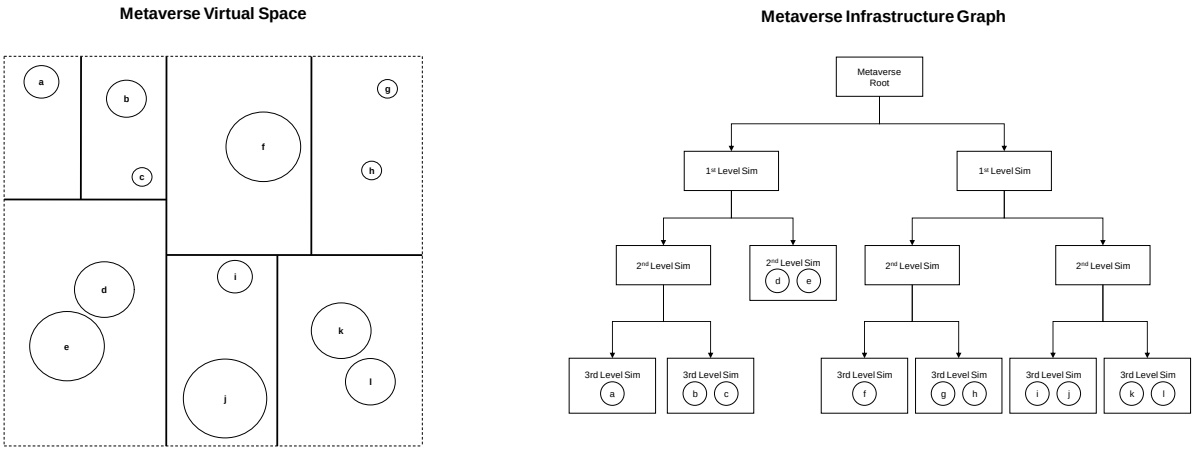


**Figure 1: Recursive XPU partitioning of the virtual space, as well as a graph representing the partitioning. Each circle represents an object in virtual space, while each box represents a separate server process managing the space.**

Active Worlds [aw] is another metaverse-like virtual world that allows dynamic content creation, including a simplified scripting interface. The Active World universe hosts hundreds of worlds which can be traversed by users, where each world is hosted on a single server.

Second Life [sl,kumar] and its open-source counterpart OpenSimulator [os] are metaverse-like worlds that allow users to explore and create a dynamic 3 dimensional space. This space is partitioned into square 256x256m regions, each managed by a separate simulator process. Each simulator is tied to a specific region of land, and cannot be repartitioned to react to a changing workload. This is the primary reason that scaling up is such a difficult problem in this architecture. Larger spaces are created by placing simulators adjacent to one another. Shards or instancing are not required.

Different topologies of fixed grid spatial subdivision have been explored, such as triangular, square, hexagonal and brickworks [presetya]. These systems are not as scalable as spatial subdivision approaches using hierarchical grids. Either dynamic resource allocation is not present, or it involves moving server processes around so that unloaded servers can time-share a single CPU.

The Project Darkstar (Sun Gamer Server Technology framework) approach to accommodating massive world state avoids spatial subdivision in favour of storing object and world state in a massive database. Actions on objects are performed through the database.

[lee] Introduces an architecture based around dividing the world space into a rectangular grid, with the cells representing virtual spaces. The cells are allocated to servers based on a graph partitioning algorithm. This approach is suitable when the scale of the simulation is known but is difficult to adapt to scale-free simulations. (For instance, when an ant colony and a galaxy are both simulated.)

ALVIC approaches metaverse design by using quad-tree subdivision for partitioning logic servers and employing many proxy servers to hide the network topology from clients [quax].

## 3.1 Algorithms from Computer Graphics

XPU borrows fundamental tree data structures from computer graphics. All modern ray-tracers rely on acceleration structures to manage scene and world data to minimize computationally expensive collision and lighting calculations.

One classic approach to this problem is to divide space into hierarchical bounding volumes (HBV) [rubin]. In this approach, the 3D space is divided into rectangular prism hierarchies and arranged in a tree structure. Child nodes represent space encompassed by the parent, with leaves being atomic renderable objects such as triangles and spheres.

*kd*-trees are a more restrictive type of spatial partitioning, only allowing partitioning planes to subdivide space, perpendicular to the canonical 3-space axis, resulting in a binary space partitioning (BSP) tree. This data structure is successfully used in modern ray-tracing algorithms [reshetov].

# 4. The XPU Tree

The core design motivation of XPU is the assumption that no single computer has enough resources to manage the entire metaverse simulation. XPU provides a convenient load splitting and management mechanism to distribute computation over a set of servers.

At the core of the XPU architecture is the XPU tree. The XPU tree is very similar to a *kd*-tree discussed in Section 3.1. The most significant difference between the XPU and the BIH tree is that leaves in an XPU tree represent virtual 3D spaces instead of objects. Each leaf of the tree is a separate server process that manages all of the objects in that virtual space. Just as in all HBVs, parent nodes must completely encompass the space occupied by child nodes.

The root node in the XPU tree represents a simulation process (termed simulator or sim) managing an entire XPU universe. To distribute the workload of managing the XPU universe, each node can divide its managed space in two, and pass off the processing to two child nodes.

Partition planes must be chosen to balance the load and ensure that all objects are fully enclosed within a child sub-volume.

## 4.1 Load Balancing and Splitting

The most significant motivation to XPU design is the need to divide and distribute processing load of a metaverse over many servers. The two most significant operations in managing XPU systems are node splitting and merging.

When a simulation process is overwhelmed by an implementation-specific definition of load, it can choose to split its workload between two child sims (Figure 1). For this operation, the XPU system will need to assign two servers (from a pool of idle simulators) to the task, and give them each a portion of the simulation state to manage.

The converse operation is much simpler – when two sibling leaf simulators have a small workload, they can choose to simply synchronize state and revert processing to a single simulator. The now vacated child sim can rejoin the pool of idle simulators.

In XPU, the partitioning borders between sims can be dynamic and reactive to the workload. In the case where two neighbouring nodes in an XPU tree have an unbalanced workload, one child can grow while the other shrinks to distribute the workload evenly between the two nodes.

# 5. PARTITIONING ALGORITHMS

The most fundamental algorithm required to develop a stable and reactive XPU implementation is finding an algorithm to manage the server partitions. This problem encompasses deciding when a simulator should be split into two child simulators, when to merge two child simulators back into its parent, and where the boundary exists between two sibling simulators.

Suppose the maximum number of objects that can be computed in real-time by a single simulator is *c*. We define a split/merge operation to determine when a node must be split into two sub-children, and when to merge two child nodes back into a single node.

```
Split_merge( simulator node X)
    if X does not have children
        if X.object_count > s
            split Xin  two, moving X's objects to its children*
            Split_merge( X.leftchild)
            Split_merge( X.rightchild)
    else
        adjust boundary between children as necessary*
        Split_merge( X.leftchild)
        Split_merge( X.rightchild)
        if  X.leftchild.object_count+X.rightchild.object_count  <
m
            merge X's children into one node
```

The values of the split parameter, *s*, and the merge parameter, m, must be determined experimentally, where $s >= m$. If $s > c$, then some nodes in the simulation may become overloaded, and the objects being managed by an overloaded simulator will experience degraded service. While this is not ideal, in some cases it may be preferable for some objects to experience a marginally degraded service rather than incur the cost of transferring objects to new simulators, and using more compute resources.

Lines marked with a * indicate strategy-specific decisions, discussed in Section 5.2.

## 5.1 Fitness Metrics

To evaluate the relative merit of the XPU algorithms, we develop four metrics to quantify that performance of each approach.

We define the metrics:

$\lambda$ = number of simulators
$\delta$ = number of sim crossings (including partition splits & merges)
$\theta$ = overload score
$\omega$ = spatial locality score

$\lambda$ is the number of simulators the solution requires. It is better to use fewer simulators if possible to minimize the amount of compute resources required to manage the virtual world.

$\delta$ is the number of objects that must be transferred between simulators during the simulation. Every object that crosses between simulators incurs a network cost because it must be synchronized, transferred and marshalled over the network. Objects transferred between regions due to sim splitting/merging and the natural movement of objects in the virtual space contribute to $\delta$.

$\theta$, the overload score, reflects the amount and degree of overloaded (lagged) objects in our virtual world.

$$\theta = \sum_{k \in \{k_s > c\}} k \cdot \frac{k-c}{c} \quad \textbf{(1)}$$

where $k_s$ is number of objects in simulator *s*, and *c* is maximum number of objects a simulator can manage before becoming overloaded.

$k \bullet (k\text{-}c)/c$ is a measure of the number of objects in the simulator (all of which are experiencing and overload condition) multiplied by amount of overload in a simulator. We sum this over all simulators that are experiencing an overload condition to calculate $\theta$.

ω, the spatial locality score, measures the amount of inter-simulator object-to-object interactions we have in our virtual world. This metric is important because it represents the network cost of supporting interaction in the virtual world.

$$\omega = \sum_{(o_i, o_j) \in O} \frac{1}{1 + \left| o_i - o_j \right|^2} \quad \textbf{(2)}$$

Where $O = \{(o_i,o_j) \mid o_i$ and $o_j$ not in the same simulator$\}$, $o_i$ = location of object *i*, $o_j$ = location of object *j*.

$|o_i - o_j|$ is the distance between two objects. We invert and square the distance plus one to get the probability that the two objects will interact. We observe that nearby objects are more likely to interact with each other than distant objects.

We assume that any two objects that need to interact across simulators will incur a network and communication cost of 1 to interact. The spatial locality score measures the network cost of inter-simulator object-to-object interactions we project to have in our virtual world.

## 5.2 Algorithmic Approaches

This section presents various strategies for placing and moving a partition boundary, as outlined in the pseudo-code framework presented in Section 5.

### 5.2.1 *Balanced Split*

The first algorithm in this study examines is a simple partitioning operation that yields a balanced tree. The partition boundary is always placed or moved so that the number of objects in both children is as equal as possible.

This approach will use the fewest simulators, but requires frequent shuffling of objects between simulators to maintain balance.

### 5.2.2 *Mincross Balanced Split*

In this algorithm, the initial partition is placed so that it divides the objects in a simulator region in half. Afterwards, the partition is continuously moved to minimize $\delta$ (the number of objects that need to be transferred between sims) in each time step. In the event of a tie between two or more choices that minimize $\delta$, we choose the option that will better balance the number of objects in each sibling.

We only allow one sim to contain at most twice as many objects as its neighbour to avoid the construction of a highly unbalanced or degenerate tree.

### 5.2.3 *Cluster Split*

This algorithm attempts to place or move the partition to minimize the $\omega$, spatial locality score. At each time step, at each level in the hierarchy, a simplified *k*-means cluster analysis [kanungo] is performed. This approach partitions the node into two clusters which minimize the mean-squared error of the object locations on each side of the partition, with respect to their centroids. This will have the effect of lowering $\omega$.

### 5.2.4 *Mincross Cluster Split*

This algorithm is similar to *mincross balanced split*, except that tie-breaks are chosen to favour the splits that are closer to the *k*-means derived ideal partitioning.

### 5.2.5 *Mincross Center Split*

This algorithm is identical to the *mincross balanced split* algorithm, except the initial partition is placed to most evenly divide the virtual space in half, and the partition is moved continuously to minimize sim crossings with tie-breaks chosen by dividing the virtual space as evenly as possible.

This approach attempts to favour more square shaped simulator regions, which are favourable because they have a large area, relative to their border length, which lowers the chances of objects having to cross between sims due to regular object movement.

### 5.2.6 *Bintree*

This algorithm is based on a bintree data structure. A bintree is similar to a quadtree, except that at each level it divides space into two congruent rectangles instead of four.

In our implementation, the *bintree* approach simply divides space into two equal parts, regardless of the distribution of objects in the virtual space. Since it does not rely on knowledge of the position of every object in the simulation, this approach is very scalable because it does not rely on global knowledge.

## 6. EXPERIMENTS

To evaluate the fitness of each proposed algorithm, we begin by constructing a test framework. We assume a square world-space, containing objects of different sizes. These objects move under n-body (gravitational) rules with no collision. N-body was chosen because it is a simple, concise way of giving objects in the world motion, as well as having movement properties that are similar to real-world user behaviour such as clustering and flocking.

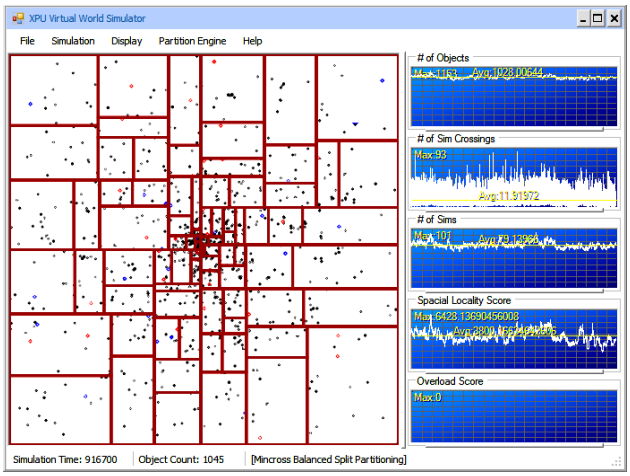


**Figure 2: Screenshot of the XPU Simulator experimental framework. The dots represent objects in the virtual world, while bold lines represent partition boundaries.**

Each object appears and disappears randomly using a power-law distribution, which is similar to observed real-world session lengths and login rates. Objects are given an initial location and velocity based on a Gaussian distribution.

The test framework has been constructed as an interactive C# application that runs on any operating system capable of supporting a windowing environment and a .Net or Mono runtime (Figure 2). A plug-in architecture is in place so that new partitioning algorithms can be added or modified without the recompilation of the framework.

The framework source code has been made freely available [xpusource].

For the reported experiments, the value *c*, the maximum number of objects a simulator can manage is set to 20, with an average of over 1000 objects in the virtual world.

### 6.1.1 *Experimental Results*

The first step in developing a good strategy for our selection of algorithms is to find a good choice of *m*, the merge parameter described in Section 5.

We begin by running all the algorithms using different choices of *m*, while holding the overload score, $\theta$, to zero by setting the split parameter *s* equal to *c*, the maximum number of objects a simulator can manage before becoming overloaded (Figure 3).

If *m* is large, then the algorithms are more aggressive in merging under-utilized sibling simulators. This results in better spatial locality scores, because more objects are distributed among fewer simulators. However, if *m* is too large, then more sim crossings occur, because the system is continuously splitting and merging simulators to achieve a good distribution of objects.

Conversely, if *m* is too small, then there are too many simulators being used, and objects moving around in the virtual space will

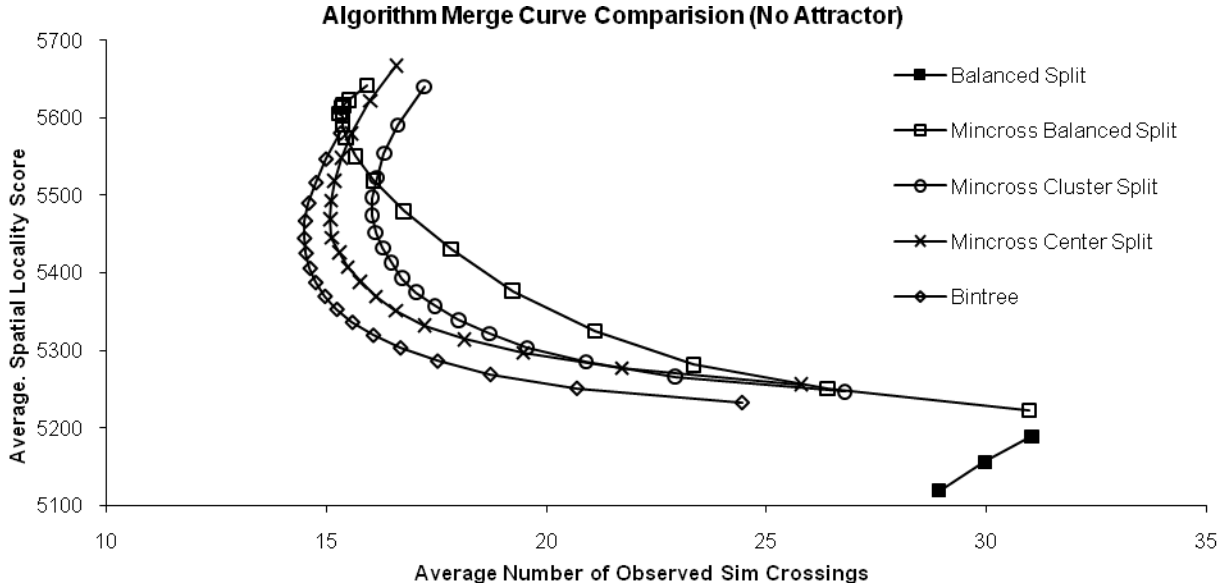


**Figure 3: A plot of the performance of various algorithms in terms of the special locality, $\omega$ versus the number of sim crossings, $\lambda$. The split constant, $s = 20 = c$, the number of objects that a simulator can manage before becoming overloaded. The values of merge parameter, $m$, is chosen from the range [20,3]. Increasing the value of $m$ corresponds to a decrease in the spatial locality score. On this plot, better results are at the bottom-left of the chart.**

incur more sim crossings due to their normal movement because the world has been subdivided into more smaller regions.

From these experiments, choosing $m = ¾\ s$ appears to be a good choice in balancing the number of sim crossings, $\delta$ and spatial locality score, $\omega$.

The *balanced split* algorithm achieves a good result for $\omega$, because it is able to use the fewest simulators to manage the simulation, but yields a relatively poor score for $\delta$. This is because the balanced split algorithm is constantly moving objects between simulators to balance the XPU tree, and the overhead of this very strict maintenance dominates the traffic generated by the motion of objects between sims.

The *mincross balanced split* approach mitigates this by placing emphasis on moving partitions based on minimizing the number of sim crossings between time steps. This has the effect of drastically reducing the number of sim crossings. However, since the objects are not as efficiently distributed over the simulators, the spatial locality score, ω, is increased.

The *cluster split* algorithm (not graphed) yields generally good results for ω, in the 5180-5600 range. Surprisingly, in this simulation the *balanced split* approach yields better results for ω. This is due to the distribution of the objects in this particular simulation; all the objects are generally clustered around the center of the world rather than many distinct clumps, and so there are not as many clusters for the algorithm to identify. The results for $\delta$ are much worse than the other algorithms, in the 105-135 range. This is because the algorithm is very aggressive in moving objects between simulators to preserve perceived clusters.

The *mincross cluster split* approach improves upon the *mincross balanced split* approach by involving by analysing more positional information. Because partitions are constructed to minimize frame-to-frame sim crossings and preserving clusters, more objects are placed in the center of a partitioned space. This reduces the amount of sim crossings that are incurred by objects just moving between partitioned simulator regions.

The *mincross center split* approach further improves the results generated by *mincross cluster split*. By favouring partitions that divide space evenly, simulator regions are closer to being square shaped. Square shaped regions are preferable to more elongated rectangular regions, because squares have a smaller perimeter compared to their area, which reduces the amount of sim crossings incurred by objects moving around.

The results for *bintree* are significantly better than *mincross center split*, which is surprising, since it does not incorporate fine grained understanding of object location into its partition placement algorithm. This algorithm benefits greatly by favouring the construction of square-shaped sims, which minimize their perimeter relative to their area. This suggests that having good simulator shapes is more important than using more complex algorithms to attempt to minimize frame to frame sim crossings. As an additional benefit, this algorithm does not require any form of global knowledge to construct its partitions. Since the only knowledge any simulator needs to perform load balancing & shedding is the computing workload of its two children, these decisions can be performed locally. This allows the *bintree* approach to scale to arbitrarily complex virtual world instances.

The average value for $\lambda$ required to support the simulation is presented in Table 1. As expected, the *balanced split* algorithm uses the fewest number of simulators, because it balances the entire tree at every time step. The other algorithms use between 50-60% more computing resources to manage the simulation.

| *Algorithm* | $\lambda_{avg}$ | $\lambda_{max}$ |
|---|---|---|
| *balanced split* | 73.20762 | 78 |
| *mincross balanced split* | 103.39764 | 125 |
| *cluster split* | 97.55792 | 113 |
| *mincross cluster split* | 98.63064 | 111 |
| *mincross center split* | 99.7733 | 114 |
| *bintree* | 100.4572 | 116 |

**Table 1: Average and maximum $\lambda$ (number of sims required to support the simulation) when $m$ = ¾ s**

To summarize these findings, *balanced split* uses the fewest computational resources, while *bintree* uses the fewest network resources to manage the simulation. Additionally, *bintree* is the only algorithm that does not depend on global knowledge, making it a suitable algorithm to base future development on.

# 7. FUTURE WORK

One of the drawbacks of the *bintree* algorithm is that it does not adapt well to the situations where the density of objects drops off faster than exponentially – for instance, simulating an ant-hill in an otherwise empty virtual space the size of a galaxy. Since *bintree* can only divide space in half, it lacks the fine-grained control to allocate vast empty regions of space to a single server. This can be addressed by implementing techniques similar to those presented in [presetya]. If multiple unloaded simulators are allowed to share a single server, then many empty regions can be grouped together. As an additional advantage, partially loaded sims can also be grouped together, making a more efficient distributed system.

Another component of XPU that has not been explored in detail is the management of the simulation when there is insufficient compute resources available to manage the simulation. In this case, the system is overloaded, and there are no more idle simulators to bring online. It is necessary to explore how to intelligently distribute the workload so that high priority objects can be processed without delay while lower priority objects will experience the least possible amount of lag.

Another component of analysis that requires deeper exploration is an examination of the relative importance of the various fitness metrics. For example, it is not clear what the relative cost is of a sim crossing is compared to the cost of two objects interacting across a sim boundary. A more comprehensive model of the virtual world or an implementation of a virtual world is required.

# 8. CONCLUSION

The growing domain of metaverse applications uses a variety of "scale-out" mechanisms to make ever larger virtual worlds. While these approaches provide a means to support increasingly large numbers of simultaneous users, they do not accommodate the demand for additional richer, simultaneous interactions. To drive new usages, we need to remove the limitations of current approaches so that the simulation architecture is driven by the content, rather than having the content limited by the architecture.

In this paper we have described XPU, a hierarchical space partitioning architecture used to distribute a simulation workload in infinitely scaling chunks so that any simulation requirements can be met. XPU borrows acceleration structures from modern ray tracing algorithms to maintain a tree that successively divides the virtual space into manageable collections of objects and avatars. By using a *bintree* subdivision method, we are able to construct a scalable load distribution mechanism for arbitrarily complex metaverses.

The unique benefit of the XPU hierarchy is that the simulation scales to accommodate both the limitations of the simulation and the requirements of the application. That is, the XPU architecture enables metaverse interactions to scale arbitrarily to accommodate the requirements of simulation by distributing the simulation across available compute and communication resources.

# 9. REFERENCES


[croquet] Croquet Project http://www.opencroquet.org/

[darkstar] Sun, Game Server Technology White Paper, http://www.sun.com/solutions/documents/white-papers/me_sungameserver.pdf

[kanungo] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. An efficient k-means clustering algorithm: analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Volume 24, Issue 7, July 2002 Page(s):881 - 892

[kumar] Sanjeev Kumar, Jatin Chhugani, Changkyu Kim, Daehyun Kim, Anthony D. Nguyen, Christian Bienia, Youngmin Kim and Pradeep Dubey. Characterization and Analysis of Second Life Virtual World. *IEEE Computer Graphics & Applications*, (March 2008)

[lee] Kyungmin Lee and Dongman Lee. A scalable dynamic load distribution scheme for multi-server distributed virtual environment systems with highly-skewed user distribution, In *Proceedings of the ACM symposium on Virtual reality software and technolog*, (2003)

[os] OpenSimulator, http://opensimulator.org

[presetya] Kusno Prasetya and Zheng da Wu. Performance Analysis of Game World Partitioning Methods for Multiplayer Mobile Gaming. In *Proceedings of the Workshop on Network and Systems Support for Games (NetGames)*, (October 2008).

[quax] Peter Quax, Jeroen Dierckx, Bart Cornelissen, Gert Vansichem and Wim Lamotte. Dynamic server allocation in a real-life deployable communications architecture for networked games. In *Proceedings of the Workshop on Network and Systems Support for Games (NetGames)*, (October 2008).

[reshetov] Alexander Reshetov, Alexei Suoupikov and Jim Hurley. *ACM Transactions on Graphics* 24,3, pp. 1176-1185 (2005)

[rubin] Steven M. Rubin and Turner Whitted. A 3-Dimensional Representation for Fast Rendering of Complex Scenes. In *Computer Graphics (Proceedings of SIGGRAPH 80)* vol. 14, pp.110-116 (1980)

[sl] Second Life, http://secondlife.com

[uo] Ultima Online, http://www.uo.com

[wachter] Carsten Wächter and Alexander Keller. Instant ray tracing: The bounding interval hierarchy. In *Proceedings of the Eurographics Symposium on Rendering*, pages 139--149, 2006.

[wow] World of Warcraft, http://www.worldofwarcraft.com

[xpusource] http://francischang.com/downloads/XPU_Sim.zip