Accurate Real-time Physics Simulation for Large Worlds

Lorenzo Schwertner Kaufmann, Flavio Paulus Franzin, Roberto Menegais and Cesar Tadeu Pozzer
Universidade Federal de Santa Maria, Santa Maria, Brazil

Keywords: Real-time Physics Simulation, Physics Engines, Large-scale Simulations, Floating-point Imprecision.

Abstract: Physics simulation provides a means to simulate and animate entities in graphics applications. For large environments, physics simulation poses significant challenges due to the inherent known limitations and problems of real number arithmetic operations. Most real-time physics engines use single-precision floating-point for performance reasons, limiting simulation with a lack of precision that causes collision artifacts and positioning errors on large-scale scenarios. Double-precision floating-point physics engines can be used as an alternative, but few exist, and fewer are supported in game engines. In this paper, we propose an efficient solution capable of delivering precise real-time physics simulation in large-scale worlds, regardless of the underlying numeric representation. It implements a layer between high-level applications and physics engines. This layer subdivides the world into dynamically allocated sectors which are simulated independently. Objects are grouped into sectors based on their positions. Redundant copies are created for objects crossing sectors' boundaries, providing seamless simulation across sector edges. We compare the proposed technique performance and precision with standard simulations, demonstrating that our approach can achieve precision for arbitrary scale worlds while maintaining the computational costs compatible with real-time applications.

1 INTRODUCTION

Physics simulation plays a fundamental part in many virtual scenarios across a broad range of applications, including games, films, and simulations. Physics are simulated using physics engines —low-level middleware that provides support to physical behaviors, such as fluids, soft and rigid bodies simulation. Games and simulations rely on real-time physics engines to provide dynamic, immersive scenarios. For performance reasons, most of these engines operate with single floating-point precision, enough to cover a wide range of applications. However, for large-scale scenarios, these engines present a limitation due to the low accuracy of the floating-point numbers, especially when representing big numbers. Simulations for objects far from origin might use imprecise values, causing failure in the collision detection algorithms, varying from small intersections that do not compromise the overall simulation to big intersections that spoil the simulation (e.g., vehicles falling off a terrain). Moreover, nowadays, it is noticeable that 3D games and graphical simulations, used for training or amusement, are dealing with constantly expanding scenarios, thus requiring solutions to enable proper visualization and simulation of moving entities.

Floating-point imprecision occurs because compressing infinite real numbers into a limited amount of bytes requires an approximate representation (Goldberg, 1991). IEEE-754 floating-point represents real numbers as $\delta \times \beta^{\varepsilon}$, where δ is a significand, β is a base, and ε is an exponent, thus rounding error increases proportionally to value magnitude. Most arithmetic operations using values represented with floating-point precision produce results that cannot be represented using a fixed amount of bits, thus requiring rounding.

In physics simulations, the rounding error increases proportionally to objects' distance from the origin on which the simulation is continuously performed, and error accumulates over error. Empirical experiments suggest that single-precision floating-point limit is 10^4 units (Unity Community, 2018); while others define it at 5×10^3 units (Unity Community, 2017), or within the range of 4×10^3 to 8×10^3 units (Burk et al., 2016). Thus, applications with scenarios larger than these dimensions are susceptible to numerical inaccuracy and may present problems throughout the physics simulation.

In terms of accuracy, some physics engines are designed to support double (e.g., Bullet and Newton Dynamics) instead of single-precision to represent the physics components and provide more accuracy in calculations. Double-precision math (and thus

Kaufmann, L., Franzin, F., Menegais, R. and Pozzer, C.
Accurate Real-time Physics Simulation for Large Worlds.
DOI: 10.5220/0010194501350142
In *Proceedings of the 16th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2021) - Volume 1: GRAPP*, pages 135-142
ISBN: 978-989-758-488-6

double-precision matrix and vector APIs) cannot take as much advantage of SIMD instruction sets (Ericson, 2005), which are generally sized to support and vectors that have 32-bit wide components. Few physics engines support double floating-point precision and game engines, such as Unity, Unreal Engine 4, Cry Engine 3 and Godot do not natively support them.

An alternative to increase numerical precision is the origin shifting approach. This approach establishes a real-time center point (e.g., the observer position), and 1) physics simulation considers the center point as the Cartesian origin or 2) the objects are shifted to the Cartesian origin, having a center point as a reference, maintaining the same layouts of objects' position. In an efficient and precise way, this approach guarantees the physical simulation for any dimension of the scenario. However, physics objects must be close to an arbitrary center point. Otherwise, inaccuracy errors remain. For sparsely distributed simulations, e.g., multiplayer games, origin shifting does not work.

We present a solution compatible with real-time simulations that expands state-of-the-art physics engines, ensuring physics simulations enough precision, regardless of the world scale and underlying numeric representation. Our solution is implemented as a layer between high-level applications and physics engines. This layer split the world in sectors and adds redundancy to each sector, ensuring each part has all the required information to, independently, be simulated without inconsistencies close to the Cartesian origin. Furthermore, our solution benefits from how large-scale worlds application must implement world streaming due to memory and processing limitations, amortizing space partitioning cost. The main contribution of our work is a novel solution that provides arbitrariness in sparse simulations precision, enabling floating-point engines to increase precision and/or scale.

This paper is organized as follows: Section 2 provides background and related works on the subject. Section 3 discusses all the steps of the proposed solution in-depth. Section 4 discusses the effects of the sector size choice. Results are evaluated and discussed in Section 5. Finally, Section 6 presents conclusion and directions for future work.

2 RELATED WORK

Many works explored the distribution and parallelization of simulations to provide simulation at an individual level, i.e., without relying on group-generalized properties. Each agent must have the required information for its correct simulation —this requirement is termed awareness. Different techniques focus on partitioning their agents into groups while keeping a low number of overlapping between regions of interest of objects in different partitions —this problem is termed communication level. Besides it, the distribution of simulations introduces other problems, such as well-balancing the simulation load across the available processing nodes and how to operate efficiently since the partitioning algorithms themselves might not scale well.

(Lozano et al., 2007) explores a complete framework for scalable large-scale crowd simulations. Their solution successfully provides scalability while providing awareness and time-space consistency. This is achieved using rectangular grids and dividing the agents according to their position in the grid.

(Vigueras et al., 2010) improves efficiency of previous methods by employing the use of irregular shapes regions (convex hulls). Their results show that irregular shapes outperform techniques with regular ones, regardless of the crowd simulation.

(Wang et al., 2009) proposes a technique for partitioning crowds into clusters, minimizing communication overhead. Their work achieves great scalability through the application of an adapted K-means clustering to efficiently partition agent-based crowd simulations.

Newer techniques explore further performance improvement, either minimizing communication costs, equalizing balancing, or both, while efficiently doing so. For example, (Petkova et al., 2016) uses community structures to group related agents among processors, reducing communication overhead. (Wang et al., 2018) explores parallelizing the simulation per agent using the Power Law in CUDA architecture to ensure behavior synchronization across threads. Their solution explores minute model details while improving efficiency.

(Brown et al., 2019) propose a solution applying a partitioning solution for physics servers with horizontal scalability. In their work, the scenario is partitioned in regions using Distributed Virtual Environments modeling, and each partition is assigned to a server for physics simulation. Objects interacting across partition boundaries are handled by projecting objects in overlapping regions to guarantee a seamless simulation.

These solutions propose and improve the simulation of a massive number of elements through partitioning and parallelization techniques while solving problems such as communications between agents in different partitions. (Brown et al., 2019) apply said methods showing that physics simulations per-

formance could benefit from partitioning.

However, none of the solutions consider scalability in terms of virtual world size. In fact, to the best of our knowledge, no paper explores the physics simulation of large-scale environments - prone to error because the inherited physics simulation floating-point precision issues increased by the types of arithmetic a physics engine must calculate.

It must be noted that when we mention large scale physics simulations, we are not referring to a 'large number of objects' simulation, but rather to the simulation of an arbitrary number of objects with large scale positions.

Several commercial examples managed to address the physics simulation precision limitations. Kerbal Space Program is a space flight exploration and simulation game praised for accurate orbital mechanics. Internally it uses Unity-PhysX (single floating-point precision) and an origin shifting approach for near player simulations combined with numerical models for distant orbit calculations. This solution is simple and efficient but is not capable of simulating far objects using the physics engine. Therefore, the solution works for applications centered in one entity, and which objects far from this entity does not require simulation. For sparse simulations, e.g., multiplayer games and military simulators, the solution is insufficient.

Outerra (Outerra, 2012) is a 3D planetary engine which uses the JSBSim Flight Dynamics Model library for high fidelity simulation of aircraft, and Bullet physics engine for simulation of vehicle physics. The Bullet physics engine supports single or double-precision floating-point arithmetic (Guo, 2013). With the double precision option, simulation calculations are more precise and can be performed in a larger simulated volume.

Star Citizen is an upcoming multiplayer space trading and combat simulator with explorable star systems. It has been developed in an extended version of CryEngine 3 (Star Engine) with 64-bit arithmetic (Burk et al., 2016) to support a vast playable volume of $200 \times 10^9 Km^3$ (Burk et al., 2016).

These solutions are limited to specific use cases. Origin shifting only supports large-scale simulations if the physics simulation is concentrated around a unique center point (e.g., camera position). Also, physics simulations far from this center point must tolerate low precision simulations or be disabled. Double precision requires physics engine support while few engines support it and fewer game engines. Furthermore, except for Outerra-Bullet, these solutions are proprietary, and the used techniques are not described in scientific papers.

We use a partitioning system, along with objects' relocation relative to their partition center (origin shifting), and a cloning system similar to Brown's Aura projection to achieve large-scale precise real-time physics simulations for sparsely distributed objects.

3 ACCURATE PHYSICS SIMULATION

This section describes the Large Physics Sectorization System (LPSS), our layer-based proposal to perform physics simulation on large-scale scenarios with sparse objects, ensuring numerical accuracy (Figure 1). Our solution can be used with any physics engine that supports distinct concurrent simulations, such as PhysX, Bullet and, Open Dynamics Engine.

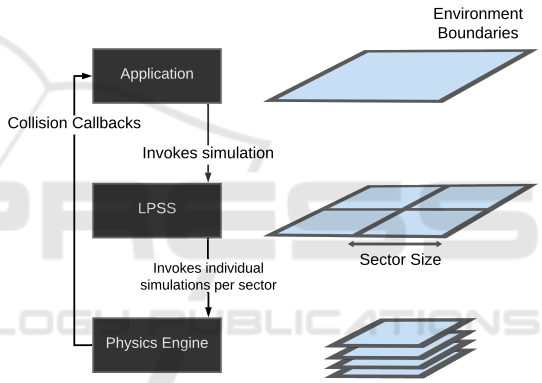

Figure 1: Overview of the architecture. Sector size is defined by the application prior to the simulation start. The physics engine runs the sector simulations concurrently.

The world is subdivided into regular partitions, similarly to (Wang et al., 2009). During the execution, before each physics simulation, the *Sectorization* process (Sec. 3.1) assigns each physics object to one sector based on its center position. Each object is relocated relative to its sector. In the sequence, the *Cloning Manager* (Sec. 3.2) filters potential collisions between objects in distinct sectors and creates *Clones* with redundant physics information only for these objects, similarly to (Brown et al., 2019). Finally, the *Simulation* process invokes one physics simulation instance per sector.

3.1 Sectorization

Any system can be employed to partition objects. Using an adapted K-Means algorithm, for instance, would partition objects more efficiently than a grid

partitioning one. Using K-Means, however, would not allow a satisfactory implementation of partition size constraint, and the simulation would suffer from the floating-point arithmetic issues.

The reason above was key in our decision to use a grid partitioning system. Although its performance is far from sophisticated partitioning techniques, it allows us to define hard-boundaries which ensure objects' coordinates are inside a pre-defined volume and, thus, coordinate errors are inside an acceptable range. Specifically, our partitioning system works as follows.

The world is logically subdivided into three-dimensional partitions. Each partition might have a sector assigned to it depending on the existence of objects inside that partition. A sector is an abstract structure in which objects can be assigned and revoked. Since the solution aims for an arbitrary scale, we map each sector into an auto-resizing hash table. Thus, sectors are created when an object is placed within its volume and destroyed when no object is inside. Sectors that contain at least one dynamic object are simulated per physics step. Each sector has an index to localize them in space. During sectorization, physics objects are assigned to one sector based on their position as:

$$(i, j, k) = \left\lfloor \frac{(x, y, z)}{L} \right\rfloor \quad (1)$$

where (i, j, k) is the sector hash index, (x, y, z) is the object position, and L is the sector's edge size.

After sectorization, object position is the sum of its sector center and the displacement to it:

$$(x, y, z) = d_s + (s_i, s_j, s_k) * L \quad (2)$$

where (x, y, z) is the object position, (i, j, k) is the sector hash index, d_s is the object's displacement to its sector center and L is the sector's edge size.

Representing the position in this format guarantees that the object's coordinate is within the sector boundaries, and the associated error has a stipulated limit.

3.2 Cloning Manager

During the *Sectorization* process, each object gets assigned to one sector based on its position. Furthermore, each sector is simulated independently, and objects from different sectors do not interact in the *Simulation* step. Consequently, this can cause inconsistency in the simulation when objects intersect adjacent sectors. As an example, in Figure 2, object B only collides with C when its position is within the sector S_a boundaries. To handle these cases, the *Cloning Manager* identifies probable interactions that would cause inconsistency and creates *Clones* to ensure a correct simulation.

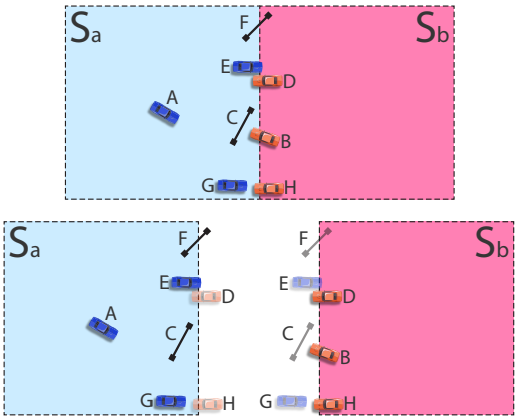

Figure 2: Simulation situations and how cloning manager handles them. Dynamic objects are represented by vehicles, static objects by walls, and Clones are transparent.

A *Clone* is a copy of the physical components of a physics object created to other objects react correctly when assigned to different sectors. The procedure to create a clone is:

1. Create a deep copy of the original object. Game engines already provide methods for this (e.g., Unity's *Instantiate* and Unreal Engine's *DuplicateObject*).
2. Remove all non-physics components related to physics.

Static clones send physics callbacks to their original objects. Dynamic objects physics properties, such as velocity and position, are copied from the original object to their *Clones* before each physics simulation step (Figure 3).

The Cloning Manager considers pairs of objects from different sectors that might collide, using Axis-Aligned Bounding Box (AABB) intersection algorithms. For those pairs, three cases are possible:

- *Static*: static object intersecting with two or more sectors;
- *Dynamic-Static*: a probable collision between static and dynamic objects in different sectors;
- *Dynamic-Dynamic*: a probable collision between dynamic objects in different sectors.

Figure 2 illustrates the cases handled by the Cloning Manager. *Static* overlapping cases (e.g., wall F) are solved when the object is created. For each sector intersected by the object, a clone is created and assigned to it. It remains active during the object's lifetime. For *Dynamic-Static* cases (e.g., B-C), a clone

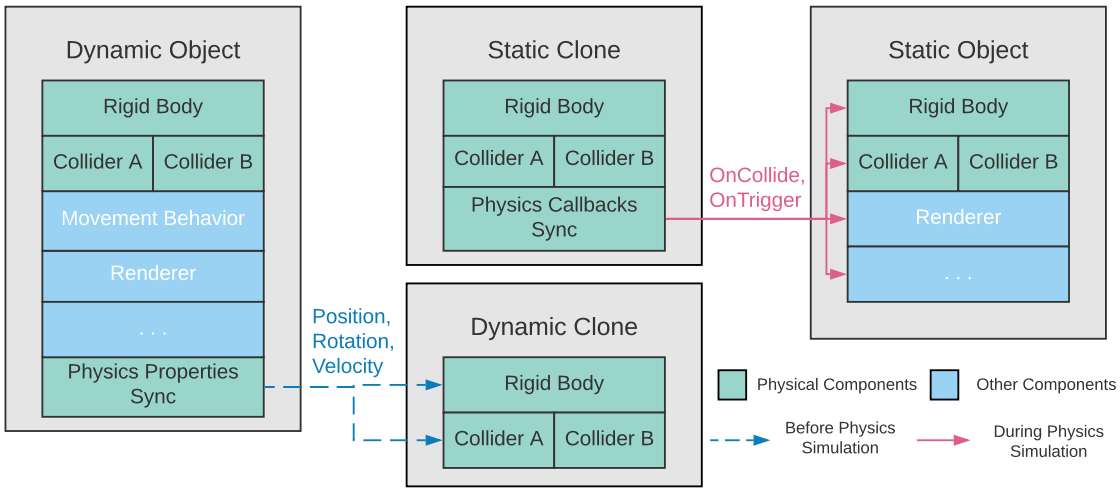

Figure 3: Original objects have miscellaneous components, e.g., renderer and movement behavior, while clones have only physics related ones. Both share a component to sync state (physics properties and collision events).

of the static object is created and assigned to the dynamic object's sector. For *Dynamic-Dynamic* cases (e.g., D-E and G-H), both objects should be cloned. In the D-E case, both objects overlap the adjacent sectors; and, in the G-H case, vehicle G does not overlap sector S_b, but is in imminent collision with vehicle H. Vehicle A is far from the sectors' border and the Cloning Manager ignores it.

For every physics step, the Cloning Manager considers all pairs of objects and generates a collection with required clones. For performance and memory segmentation reasons, each object manages a pool of clones to avoid continuous object creation.

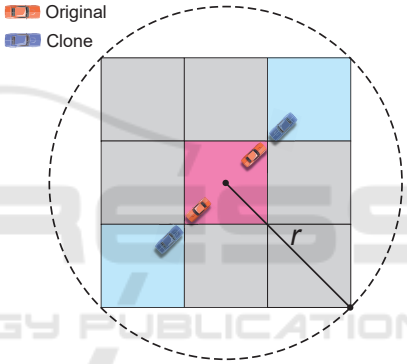

Figure 4: Sector and its neighbors inscribed in precision range (sphere with radius r. For better viewing drawn as a circle).

4 SECTOR SIZE

The sector size is an arbitrary value, but if it is too big, the simulation might be imprecise. In this section, we define the sector size's limit to ensure precision. Our solution has the following design choices:

- The solution does not offer support for objects with AABB larger than the sector size;
- Clones share their original object position while being placed in another sector. Consequently their position is outside their assigned sector.

Consider Figure 4. The pink sector might contain clones from adjacent sectors. To guarantee clones are simulated inside the precision range, sector size must be such that all neighbor sectors stay inside it. The sector edge L must be $L \leq \frac{2r}{3\sqrt{3}}$, where r is the precision limit.

5 RESULTS

Physics are sensitive to initial conditions, i.e., small changes in one state can result in more significant differences in later ones. As discussed before, floating-point physics simulations are also susceptible to accumulating rounding error, varying the error in many degrees of magnitude with slightly different distances from the Cartesian origin, which increases the sampling error. Thus, computing a physics simulation error is not trivial, and results might differ significantly under slightly different conditions.

We compare the PhysX physics engine and our solution running on top of PhysX, both in the Unity game engine. Our tests run in a Quad-Core i3-9100F, GTX 1050 Ti, and 8GB of DDR4 RAM clocked at 2400 MHz.

Our precision test consists of two spheres, im-

pulsed towards each other to collide. The spheres' radii are 0.5 units, and in the best conditions, a collision is expected when spheres are one unit apart. Running the test at origin (ground truth) results in a collision detection when spheres are 0.479 units apart due to simulation discrete time steps. All tests execute using the same time step value, thus canceling any disturbance on the results. We calculate the error for each object as the Euclidean distance between its final position after the simulation and its final position after the ground-truth simulation. The simulation error is the aggregation of all objects errors using a Least Absolute Deviations loss function. Figure 5 compares collision detection error over origin distance. Since our implementation uses a sector grid centralized at the origin, it offers the same precision until half sector size (with minor differences). After half sector size, PhysX error increases until 8×10^6 distance from the Cartesian origin. After this distance, spheres would not collide due to imprecision even when using sophisticated and expensive collision detection methods available in Unity-PhysX, e.g., Speculative Continuous Collision Detection (Unity, 2020). In contrast, our solution maintains the error in an acceptable range since objects are kept near the origin. When objects are assigned to the next sector, their position becomes local to the new sector, approximating and distancing from the Cartesian origin, configuring an error periodic wave pattern.

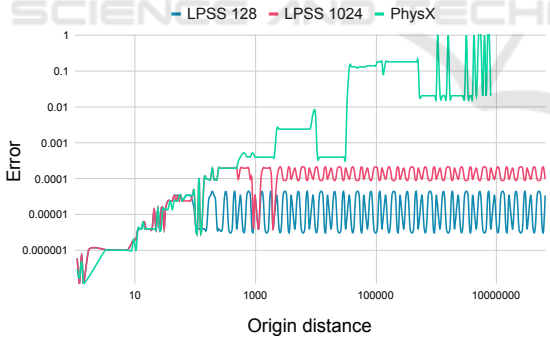

Figure 5: Collision detection error over simulation distance to the Cartesian origin using PhysX and our solution (LPSS) with 128 and 1024 sector size.

The sparse positioning of objects generates more sectors and, consequently, more instances of physics simulation are demanded. As the number of instances increases, more physics invocations are needed, directly impacting the application's performance. Figure 6 demonstrates the time consumption relative to the number of physical instances. The test starts with 128 objects assigned to one sector, and these objects are progressively distributed in more physics instances until reaching one object per sector (ending with 128 sectors). As an example of the impact caused by the number of physics instances, simulating 128 instances with one object is ≈ 80 times more expensive than one instance with 128 objects. Since simulations are independent, the problem can be mitigated by splitting simulations across processing cores.

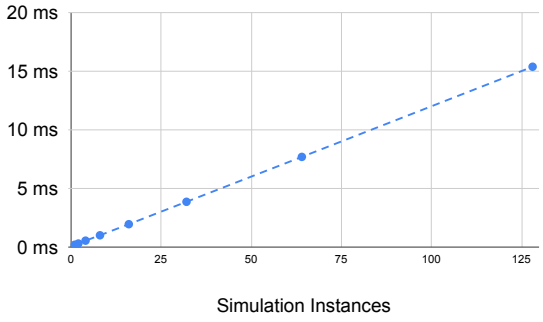

Figure 6: Processing times to simulate 128 physics objects over physics simulations instances.

The third experiments focus on the sectorization and cloning management overhead for each physics simulation step. As seen in Figure 7, objects sectorization cost is proportional to the number of objects and is negligible compared to simulation cost. Cloning management, however, has a considerable cost over object count, which can be minimized using faster broad-phase collision detection algorithms. Clones must be simulated as physics objects, which translates in increasing simulation time.

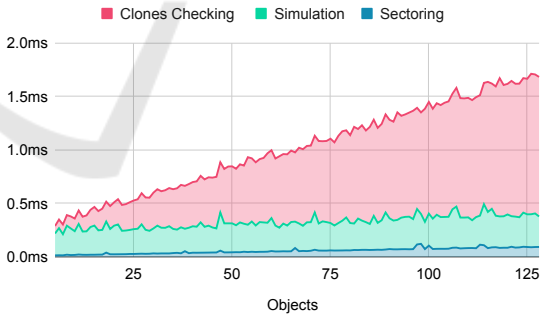

Figure 7: LPSS overhead per physics simulation.

Furthermore, we visually explored the difference between using our proposal and not. Figure 8 demonstrates how our solution scales well with large environments. Right stacks use LPSS while left stacks do not.

Top stacks are simulated at origin and bottom are displaced 2^{24} units from origin. This tests shows that simulation works as expected at the origin regardless if the simulation uses or not or solution, but at high coordinates using PhysX is so inaccurate that the simulation fall apart while our solution remain stable and

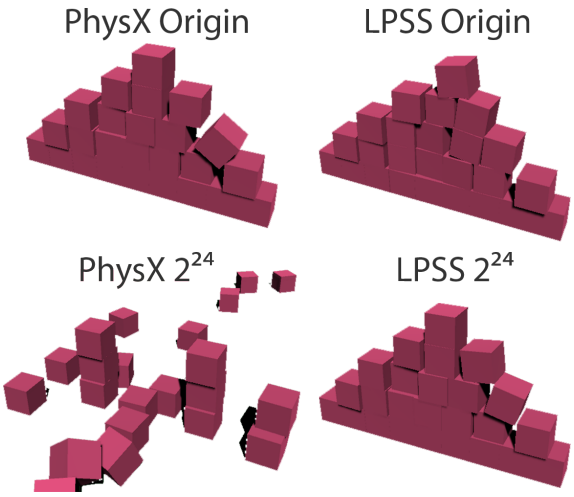

Figure 8: Visual comparison between physics simulations of stacks.

correct. In this test, LPSS sectors are 1000 units in length. A minor difference can be perceive between using or not our solution without displacement from the world origin. This discrepancy is due to a physics frame gap in our system initialization. While the PhysX test is completely disrupted when simulating with big coordinates, LPSS error is kept acceptable.

6 CONCLUSION

In this paper, we present a low computational cost system to increase simulation precision in physics engines, supporting larger or infinity scenarios regardless of the utilized physics engine and its numeric representation. This is achieved by dividing the world into logical sectors and simulating objects with sector local coordinate systems. Coordinates are restricted to half sector size. Ergo, floating-point values have a well-defined max error. Objects redundant copies are created to handle correctly interactions across sector's edges.

Smaller sectors guarantee more precision but cover a small volume, which may require more sectors and physics simulation, compromising the application performance. Hence, a trade-off can be defined between precision and performance without depending on physics engines' implementations.

Our research focuses on achieving arbitrary precision with simulation consistency and low processing cost per simulation step. As future work, we will explore acceleration structures to minimize overhead, especially in cloning management. Furthermore, we will look to merge our solution with (Brown et al., 2019), to provide an efficient cloning system and we will explore more complex partitioning systems to minimize communication overhead while maximizing load balancing, supporting large-scale virtual worlds with horizontally scalable parallelism.

ACKNOWLEDGEMENTS

We thank the Brazilian Army and its Prg EE ASTROS 2020 for the financial support through the SISASTROS project.

REFERENCES

AdoredTV (2015). Citizen spotlight - Star Citizen - Roberts Space Industries. https://robertsspaceindustries.com/community/citizen-spotlight/1599-Star-Citizen-How-Big.

Baraff, D. (1989). Analytical methods for dynamic simulation of non-penetrating rigid bodies. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1989*, volume 23, pages 223–232.

Boeing, A. and Bräunl, T. (2007). *Evaluation of real-time physics simulation systems*. Elsevier.

Brown, A., Ushaw, G., and Morgan, G. (2019). Aura projection for scalable real-time physics. In *Proceedings - I3D 2019: ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*.

Burk, S., Coleman, A., and Tracy, S. (2016). Sean Tracy on 64-bit Engine Tech & Procedural Edge Blending. https://www.youtube.com/watch?v=OB_AI9ukSp8.

Ericson, C. (2005). *Real-Time Collision Detection*, volume 1. Morgan Kaufmann.

Goldberg, D. (1991). What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1):5–48.

Guo, S. (2013). Double & Single Precision in Bullet. http://guoshihui.net/double-single-precision-in-bullet.

Kavan, L. (2003). Rigid body collision response. https://www.cs.utah.edu/~ladislav/kavan03rigid/kavan03rigid.html.

Liu, H., Bowman, M., Adams, R., Hurliman, J., and Lake, D. (2010). Scaling virtual worlds: Simulation requirements and challenges. In *Proceedings - Winter Simulation Conference*, pages 778–790.

Lozano, M., Morillo, P., Lewis, D., Reiners, D., and Cruz-Neira, C. (2007). A distributed framework for scalable large-scale crowd simulation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4563 LNCS(May 2014):111–121.

Lui, J. C. and Chan, M. F. (2002). An efficient partitioning algorithm for distributed virtual environment systems. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):193–211.

Outerra (2012). Outerra: Outerra Tech Demo released! https://outerra.blogspot.com/2012/02/outerra-tech-demo-released.html.

Petkova, A., Hughes, C., Deo, N., and Dimitrov, M. (2016). Accelerating the distributed simulations of agent-based models using community detection. In *2016 IEEE RIVF International Conference on Computing Communication Technologies, Research, Innovation, and Vision for the Future (RIVF)*, pages 25–30.

Seugling, A. and Rolin, M. (2006). Evaluation of physics engines and implementation of a physics module in a 3d-authoring tool. *Umea University*, page 89.

Tracy, D. J., Buss, S. R., and Woods, B. M. (2009). Efficient Large-Scale Sweep and Prune Methods with AABB Insertion and Removal. Technical report, University of California San Diego.

Unity (2020). Manual: Continuous collision detection (CCD). https://docs.unity3d.com/Manual/ContinuousCollisionDetection.html.

Unity Community (2017). Making an open world map. Could use some input. - Unity Forum. https://forum.unity.com/threads/making-an-open-world-map-could-use-some-input.484643.

Unity Community (2018). Floating Point Errors and Large Scale Worlds - Unity Forum. https://forum.unity.com/threads/floating-point-errors-and-large-scale-worlds.526807/.

Vigueras, G., Lozano, M., Orduña, J. M., and Grimaldo, F. (2010). A comparative study of partitioning methods for crowd simulations. *Applied Soft Computing Journal*, 10(1):225–235.

Wang, J., Mao, T., Song, X., Liu, S., Jiang, H., and Wang, Z. (2018). Parallel crowd simulation based on power law. In *2018 International Conference on Virtual Reality and Visualization (ICVRV)*, pages 78–81.

Wang, Y., Lees, M., Cai, W., Zhou, S., and Low, M. Y. H. (2009). Cluster based partitioning for agent-based crowd simulations. In *Winter Simulation Conference*, WSC '09, page 1047–1058. Winter Simulation Conference.