An Extensible Plugin for Integrating High-Performance Collision Detection Broad-Phase Algorithms into Unity

Raul Costa Feitosa , Maria Andréia Formico Rodrigues

Programa de Pós-Graduação em Informática Aplicada & GIRA Lab
Universidade de Fortaleza (Unifor)
Fortaleza – CE – Brazil

{raulcostaf,andreia.formico}@gmail.com

Abstract. *This paper presents a plugin architecture for Unity to integrate and benchmark broad-phase collision detection algorithms. The solution connects high-performance C++ algorithms to Unity's C# environment via an external dynamic-link library, allowing developers to test and compare different methods in interactive scenarios. We evaluated three integrated algorithms (Brute Force, Tracy, and KDTree) in tests with up to 1600 objects (uniform and assorted) across ten distinct simulation configurations, assessing both algorithmic performance and the integration bridge's efficiency. Results show a stable architecture, validating the use of C++ DLLs for intensive tasks in Unity and providing a modular framework to extend its native physics capabilities.*

1. Introduction

Collision detection is a cornerstone of digital games and interactive simulations, ensuring physical realism and responsive interactions between virtual objects [Ericson 2005]. It is strategically optimized using a two-stage pipeline. It begins with the *broad-phase*, a fast, initial filter that quickly discards pairs of objects that cannot possibly collide. This allows the subsequent *narrow-phase* to perform slow, geometrically precise calculations only on a small list of potential candidates, preventing the system from wasting expensive computations on irrelevant objects [van den Bergen 2003]. Within this context, the *broad-phase* stage of collision detection is of strategic importance [Gomez 2021].

Major game engines like Unity provide robust, built-in physics solutions, often leveraging integrated libraries such as NVIDIA's PhysX [Unity Technologies 2024]. More specifically, Unity's broad-phase collision detection is a non-customizable, internal component of its physics engines, whether using the traditional NVIDIA PhysX for *GameObjects* or the modern Unity Physics for DOTS. Both systems leverage automatic, highly optimized algorithms—typically based on Bounding Volume Hierarchies (BVH) [Samet 1990]—-but provide no developer API to select, replace, or inspect the underlying method. In practice, the system operates as an efficient yet inflexible "black box." This lack of extensibility prevents developers and researchers from prototyping, benchmarking, or deploying custom broad-phase algorithms tailored to specific game scenarios—a critical limitation, given that performance is highly scene-dependent [Liu et al. 2010, Tracy and Brown 2012], which thereby justifies the need for the modular plugin architecture proposed in this paper.

To address this gap, this paper introduces an extensible plugin architecture that connects validated C++ algorithms to Unity via an external library. Our goal is to provide

a high-performance, modular tool that empowers developers to test and compare different collision detection strategies within scenarios representative of challenges found in real-world games. We present the integration process, the experiments conducted, and the performance results obtained. We discuss the implications of this approach for software engineering in game development and outline future directions for this work.

2. Related Work

Numerous studies have benchmarked the efficiency of collision detection algorithms using synthetic data [Lo et al. 2013, Tracy et al. 2009] and physics simulations [Serpa and Rodrigues 2019b, Liu et al. 2010, Tracy and Brown 2012]. Synthetic benchmarks offer predictability, whereas physics-based scenarios provide more realistic and complex workloads. Most evaluations focus on scalability and performance with varying object counts [Serpa and Rodrigues 2019b, Liu et al. 2010, Tracy and Brown 2012] or non-uniform distributions [Lo et al. 2013]. Other work has analyzed performance in parallel environments, under dynamic object insertion/removal and in static scenes.

Physics engines like Bullet [Coumans 2018] and PhysX [NVIDIA 2019] do not prioritize broad-phase benchmarking as a user-facing feature. Tools such as PEEL [Terdiman 2017] focus more on stability and overall physics engine comparisons. In the academic space, standalone benchmarking frameworks have emerged to provide standardized evaluation of broad-phase collision detection algorithms, supporting various data distributions and parallel implementations.

Our work builds on approaches that integrate Unity with external libraries via DLLs to achieve high performance in real-time applications [Joiner 2019, Lewerentz 2023]. The use of Unity for sophisticated physics simulations, from serious games to virtual environment enhancements [Tarigan et al. 2024], further highlights the relevance of modular, hybrid solutions like the one we propose.

3. Background: The Source C++ Framework and Key Algorithms

3.1. A Reference C++ Benchmarking Framework

The algorithms integrated in this work originate from a well-established open-source C++ benchmarking framework [Serpa and Rodrigues 2019a]. This framework was selected for its comprehensive and validated implementations of various broad-phase algorithms, making it an ideal source for our integration efforts. It was created to address the lack of standardized methodologies in collision detection research, enabling comparable results and reproducible experiments. It features a parametric scene generator (for crowds, falling objects, random distributions), implementations of classic and modern algorithms, and tools for measuring performance, scalability, and visual analysis.

3.2. Principal Algorithms

The source framework includes a variety of algorithms, each suited to different scene types and object dynamics. Key examples include: **Sweep and Prune (SAP/iSAP)** [Tracy et al. 2009], **Dynamic Bounding Volume Trees (DBVT)** [Coumans 2018], **Grid** [Lo et al. 2013], **KDTree** [Serpa and Rodrigues 2019b], **Tracy** [Tracy and Brown 2012], and **LBVH** [Liu et al. 2010]. Benchmarks have shown that no single algorithm is universally superior. For instance, **LBVH** and **KDTree** excel in static, dense scenes, while

Tracy is a top performer for high-motion scenarios. For this initial version of our extensible plugin, we chose three algorithms to ensure structural diversity: *Brute Force* (as a baseline for validation), *Tracy* (for its strong performance in dense, dynamic scenarios), and *KDTree* (for its robustness in open-space partitioning).

The *Brute Force* algorithm exhaustively checks all object pairs with $O(n^2)$ complexity. Its inefficiency in large scenes makes it a baseline for comparison, being efficient and often optimal for finding all collisions within small object subsets. *Tracy* is a hybrid algorithm combining a uniform *Grid* [Samet 1990] with per-cell incremental *iSAP* [Tracy et al. 2009]. Each cell updates object lists using temporal coherence [Ericson 2005], avoiding full re-sorting each frame, making it highly efficient in dense scenes with localized motion. *KDTrees* recursively partition space along alternating axes, creating a flexible hierarchy that adapts to object distribution. Tree updates rely on merge, split, or adjust operators triggered by a control algorithm. This adaptability comes at the cost of computationally expensive tree updates as objects move. The tree's height is a critical parameter, balancing the trade-off between creating too many small sub-problems (over-pruning) and leaving them too large to be efficient.

4. The Proposed Plugin Architecture

We developed a Unity plugin that integrates C++ broad-phase collision detection algorithms using an external library (Figure 1). The proposal aims to deliver a modular, extensible, and user-friendly tool for testing and comparing algorithms in interactive simulations. The architecture is split into a C++ backend and a C# frontend in Unity. Communication is handled via serialization. Each frame, the C# layer collects the *Axis-Aligned Bounding Box (AABB)* of every relevant object and sends them to the DLL in an array. An *AABB* is the smallest axis-aligned cuboid that fully encloses an object, and it is used as a fast, $O(1)$ culling mechanism in the *broad-phase* to avoid the expensive calculations of the *narrow-phase*. The C++ library runs the selected algorithm and returns the pairs of overlapping objects.

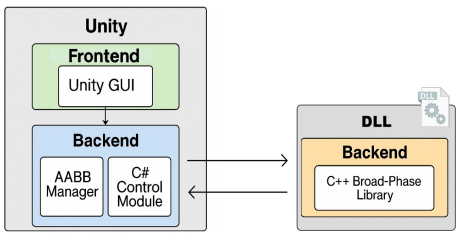

Figure 1. Modular plugin architecture: pluggable collision detectors; modules for control, AABB updates, C++ core, and Unity UI.

4.1. System Requirements

The plugin was designed based on functional and non-functional requirements.

- **Functional Requirements:** Integration of multiple algorithms, dynamic configuration of object counts, continuous *AABB* updates based on object transforms (position, rotation, scale), detection and reporting of overlapping pairs, and visual feedback of *AABBs* and collisions.

- **Non-functional Requirements:** High performance with minimal overhead from Unity-DLL communication, an intuitive GUI for configuration, modular code to simplify the addition of new algorithms, and well-documented C# and C++ codebases for maintainability.

4.2. Motivation

While standalone benchmarking tools provide valuable insights, their algorithms often remain isolated from practical game development environments. This work aims to bridge that gap by integrating these concepts directly into Unity via a plugin. This solution brings advanced *broad-phase* analysis into the real-time development loop, allowing developers and researchers to visualize, test, and compare collision detection algorithms in the engine itself. The project democratizes access to high-performance algorithms, making them easier to integrate into interactive products.

5. Methodology and Test Environment

5.1. Re-engineering Process for Unity Integration

Adapting the source C++ algorithms for Unity required a careful re-engineering process. We chose to encapsulate the collision detection logic in a DLL rather than rewriting it in C#. This decision preserved the performance of the validated C++ routines while leveraging Unity for scene management and rendering.

To ensure interoperability, we defined intermediate data structures (such as *AABBs* and vectors) with memory layouts and calling conventions compatible between C# and C++. Careful memory management was essential to prevent leaks and inconsistencies at the boundary between managed and unmanaged code. The development followed an incremental methodology to ensure a robust integration between the native C++ library and the Unity engine. GitHub was used for version control. The key stages were:

1. Core Interoperability: We first built and validated the core communication layer, for correct function exports from the DLL and data marshalling between C++ and C#.

2. Unit & Static Testing: The exported library functions were unit-tested in isolation. Concurrently, a static prototype with hardcoded *AABBs* was created in Unity to test the integration logic with non-moving objects.

3. Dynamic Integration Testing: Modular test scenarios with dynamic object spawning and movement were developed. The fully integrated system was then tested to ensure object behaviors and collision detection functioned as expected.

4. Final Validation: The primary *SearchOverlaps*[1] function was tested to confirm the consistency and accuracy of the collision data returned to Unity.

5.2. Test Scenario Modeling

We adapted the original scenarios from the source framework and created new abstractions, shown in Figure 2. All were simplified and parameterized to allow for scalable and

[1]It is the core function of any broad-phase algorithm. Each algorithm provides its own unique implementation of this function, defining its strategy for finding potential overlaps between object bounding volumes.

controlled tests, validating the plugin's performance in dynamics typical of digital games. We evaluated three integrated algorithms (*Brute Force*, *Tracy*, and *KDTree*) across ten distinct simulation configurations (5 dynamic scenarios × 2 distributions, *i.e*, uniform and assorted), with object counts ranging from 200 to 1600: **Free Fall**, where objects fall under constant gravity (Figure 2.a); **Rotating Gravity**, which simulates a rotating environment by changing the gravity vector (Figure 2.b); **Brownian**, which features random particle-like motion (Figure 2.c); **Random Gravity**, where each object is assigned a unique gravity vector (Figure 2.d); and **Hurricane**, with high-velocity rotation of objects around a central point (Figure 2.e).

(a) Free Fall **(b) Brownian** **(c) Rotating Gravity** **(d) Random Gravity** **(e) Hurricane**

Figure 2. Test scenarios implemented in the plugin. Each pair (a)–(e) shows, respectively, uniform object distribution (left) and assorted distribution (right).

5.3. Evaluation Metrics

We defined key performance metrics: average frame time (ms) for fluidity, memory usage for allocation control and leaks, and CPU spikes to identify bottlenecks. Data was gathered via an automated script for reproducibility and visually validated against the Unity Profiler's CPU and Memory modules to guide optimization.

5.4. Tools and Environment

The plugin was developed using Unity Engine (2022.3.3f1)[2], C#, and C++. The IDEs were Visual Studio 2019 and Visual Studio Code. Version control was managed with Git and GitHub. Tests were performed on a Windows 10 Pro (x64) machine with an AMD Ryzen 7 2700 CPU (8 cores/16 threads) and 16 GB of RAM.

6. Implementation Details

6.1. Plugin Architecture

The architecture consists of four main modules. The first is a C# **Control Module** in Unity that manages communication with the DLL. It collects object data, calculates *AABBs*, and sends them to the library. The second C# module, the **AABB Manager**, updates the bounding box of each object based on its transform and provides visual feedback for overlaps. The third module is the **C++ Broad-Phase Library**, which currently implements the *Brute Force*, *Tracy*, and *KDTree* algorithms. Finally, the fourth module is the **Unity GUI**, which provides a graphical interface for selecting algorithms, adjusting parameters, and viewing real-time statistics. This modular, low-coupling design combines the performance of C++ for intensive computations with the flexibility of Unity for configuration and visualization.

[2]Unity 2022 LTS was chosen as the plugin's base for its stability and long-term support, minimizing the risk of API breakage from engine updates.

6.2. Documentation and Maintainability

The project was structured into independent modules with clear coding standards. Each detection algorithm is encapsulated, facilitating the integration of new methods. The separation of concerns, consistent naming conventions, and in-code comments ensure readability and simplify maintenance and expansion.

6.3. Engineering Challenges and Solutions

A key engineering decision was to keep the computational cores in a C++ DLL, preserving the performance of the validated routines. Another one was bridging the gap between managed (C#) and native (C++) code. We defined platform-agnostic data structures and used C#'s *StructLayout* attribute to ensure memory layout alignment. Marking arrays as [*In*] or [*Out*] was essential for managing data transfer correctly. Another critical aspect was handling Unity's garbage collector, which can interfere with native memory management. Furthermore, we had to account for threading models. While the C++ algorithms can be parallelized, Unity restricts scene interactions to the main thread. Our architecture keeps heavy computation in the DLL, passing only the consolidated results back to Unity each frame to avoid blocking. The lack of native DLL debugging in Unity was overcome by implementing detailed logging for traceability throughout all execution phases.

7. Performance Analysis and Results

7.1. Algorithms Comparison

The numerical values in Table 1 represent the plugin's performance, based on data gathered by the benchmark script. Although the Unity Profiler allows for detailed timeline analysis—revealing that in most executions, over 70% of the frame time is consumed by the functions responsible for initiating routines and calling the native library—the average frame time remains the most relevant metric for practical comparison. Therefore, the quantitative analysis primarily uses the values exported by the script, while the Profiler serves as a supplementary tool to verify bottlenecks and validate the stability of each algorithm's execution. The results confirm that performance is highly dependent on the chosen algorithm, scenario, and object count. As expected, *Brute Force*'s performance degrades quadratically, becoming unusable real-time applications with 800 objects and computationally prohibitive at 1600 objects. In contrast, *Tracy* and *KDTree* scale more effectively, though both are pushed beyond real-time viability in the most demanding scenarios at 1600 objects. However, *KDTree* solidifies its position as the superior algorithm in nearly every test case, and its performance gap over *Tracy* generally widens at the highest object count. Its advantage becomes overwhelming in chaotic scenarios like **Random Gravity** and **Free Fall**, where it can be up to 10 times faster under high load. The analysis that the "Assorted" distribution boosts *KDTree*'s efficiency holds true, as does *Tracy*'s competitive edge in the **Rotating Gravity** scenario due to its suitability for coherent mass updates. Thus, while *KDTree* proves to be the stronger and more robust generalist for scaling into high-density, dynamic environments, the optimal algorithm choice remains dictated by specific scene dynamics.

7.2. System Performance

Throughout all tests, total memory usage remained below 1.96 GB, with Garbage Collector allocation holding stable at approximately 0.5 GB. This low memory footprint indicates the simulation can run efficiently on machines with modest specifications, avoiding

Table 1: Average Frame Time (ms) by Scenario, Algorithm, Object Count, and Scale Distribution.

Scenario	Algorithm	200 objects		400 objects		800 objects		1600 objects	
		Uniform	Assorted	Uniform	Assorted	Uniform	Assorted	Uniform	Assorted
Brownian	*Brute Force*	115.3	116.6	438.3	444.3	1520.6	1536.7	4572.8	4561.5
	Tracy	3.9	5.4	20.8	26.2	71.5	77.3	197.2	300.7
	KDTree	**2.8**	**3.0**	**12.9**	**13.2**	**64.1**	**67.2**	**173.0**	**281.2**
Rotating Gravity	*Brute Force*	155.8	183.5	545.5	605.6	1676.0	1790.8	4335.7	4547.6
	Tracy	23.3	52.6	114.5	190.5	424.6	636.6	1079.0	1475.6
	KDTree	**16.5**	**41.8**	**95.5**	**141.9**	**294.9**	**493.3**	**814.8**	**1196.8**
Random Gravity	*Brute Force*	120.6	138.9	461.2	499.8	1553.3	1587.6	4550.8	4535.3
	Tracy	11.5	10.8	39.6	40.1	120.4	163.6	341.1	526.9
	KDTree	**2.8**	**3.6**	**4.4**	**7.2**	**16.7**	**23.6**	**202.8**	**234.6**
Free Fall	*Brute Force*	105.3	140.7	409.6	521.9	1424.2	1691.6	4475.3	4536.9
	Tracy	3.6	12.4	23.1	36.9	159.7	216.2	593.1	970.2
	KDTree	**2.6**	**3.8**	**3.7**	**6.4**	**8.3**	**14.9**	**57.3**	**316.0**
Hurricane	*Brute Force*	134.8	133.6	491.5	497.7	2017.0	2056.9	4502.3	4596.0
	Tracy	5.9	6.2	39.1	43.2	244.5	248.1	716.3	917.0
	KDTree	**4.2**	**5.5**	**25.5**	**37.9**	**115.3**	**181.9**	**440.9**	**652.7**

performance degradation or memory swapping from exceeding RAM limits. The performance analysis validated the plugin's architecture as efficient and stable. It introduces minimal CPU overhead and stable memory usage with no leaks; performance spikes were confined to initial scene loading, as expected. The integration framework itself proved robust. Observed hangs under extreme load stemmed from algorithmic complexity, not the plugin architecture. The system scaled roughly linearly, though the object ceiling likely reflected Unity Editor overhead rather than a standalone build.

7.3. Threats to Validity

Threats to validity include limited generalizability from controlled tests that may not reflect the AI-driven unpredictability of commercial games. Results rely on simplified *AABB* geometry and omit costs from complex mesh–mesh collisions. The implementation is tightly bound to Unity's C++ interop and execution order, so engine updates may require rework.

8. Conclusions and Future Work

This work successfully demonstrates the creation of a modular and extensible software architecture for integrating external C++ broad-phase algorithms into Unity. Performance tests confirm this plugin architecture is lightweight, stable, and efficient, with a viable Unity-DLL communication bridge that operates with minimal resource overhead without compromising the host system's stability. The key contribution is that this framework circumvents the "black box" nature of Unity's default physics engine, empowering developers and researchers to select, benchmark, and deploy the optimal broad-phase strategy for their specific needs. Future work will focus on optimization (incremental updates, multi-core parallelization, GPU acceleration), extension (integrating other broad phase collision detection algorithms, and a narrow-phase stage to form a complete pipeline), and validation (applying the framework in real-world game projects).

Artifact Availability

All artifacts from this research are publicly available at `https://rb.gy/p5qhcn`.

Acknowledgments

Maria Andréia F. Rodrigues thanks CNPq (Grant 314776/2023-0).

References

Coumans, E. (2018). Bullet physics library 2.88. https://github.com/bulletphysics/bullet3.

Ericson, C. (2005). *Real-Time Collision Detection*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Gomez, M. (2021). A quick-and-dirty guide to building a collision detection system. *Game Developer*. Accessed: July 12, 2025.

Joiner, D. (2019). DLLs and Unity. https://joinerda.github.io/DLLs-And-Unity/.

Lewerentz, A. t. (2023). Integrating Julia Code into the Unity Game Engine to Dive into Aquatic Plant Growth. In *3rd Int. Conference on Interactive Media, Smart Sys. and Emerging Technologies*, pages 97–100. The Eurographics Association.

Liu, F., Harada, T., Lee, Y., and Kim, Y. (2010). Real-time Collision Culling of a Million Bodies on Graphics Processing Units. *ACM ToG*, 29:154.

Lo, S.-H., Lee, C.-R., Chung, I.-H., and Chung, Y.-C. (2013). Optimizing Pairwise Box Intersection Checking on GPUs for Large-Scale Simulations. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 23.

NVIDIA (2019). Physx 4.1. https://github.com/NVIDIAGameWorks/PhysX.

Samet, H. (1990). *The Design and Analysis of Spatial Data Structures*. Addison-Wesley Longman Publishing Co., Inc., USA.

Serpa, Y. R. and Rodrigues, M. A. F. (2019a). Broadmark. https://github.com/ppgia-unifor/Broadmark/tree/master.

Serpa, Y. R. and Rodrigues, M. A. F. (2019b). Flexible use of temporal and spatial reasoning for fast and scalable CPU broad-phase collision detection using KD-Trees. 38(1):260–273. https://onlinelibrary.wiley.com/doi/10.1111/cgf.13529.

Tarigan, N. P., Darmawan, I., and Nasution, M. N. N. (2024). Implementation of Enhanced Axis Aligned Bounding Box for Object Collision Detection in Distributed Virtual Environment. *ResearchGate Preprint*.

Terdiman, P. (2017). Physics engine evaluation lab (peel). https://github.com/Pierre-Terdiman/PEEL.

Tracy, D. and Brown, S. (2012). Accelerating physics in large, continuous virtual environments. *Concurrency and Computation: Practice and Experience*, 24:125–134.

Tracy, D., Buss, S., and Woods, B. (2009). Efficient Large-Scale Sweep and Prune Methods with AABB Insertion and Removal. *IEEE VR*, pages 191–198.

Unity Technologies (2024). Physics in Unity. https://docs.unity3d.com/Manual/PhysicsSection.html.

van den Bergen, G. (2003). *Collision Detection in Interactive 3D Environments*. Morgan Kaufmann, San Francisco, CA, USA.