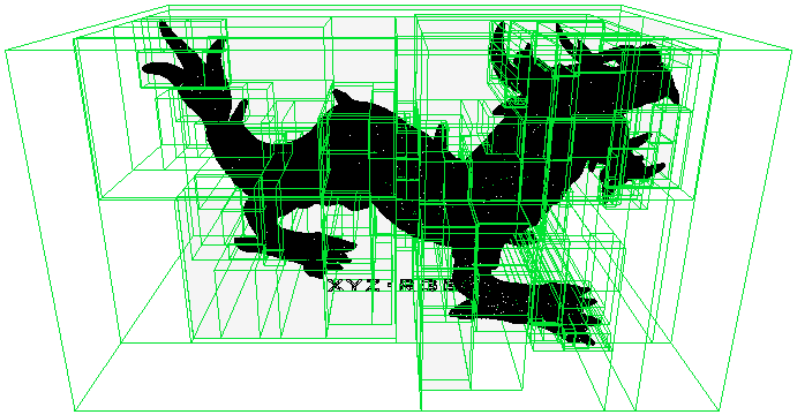
DH2323 Computer Graphics and Interaction
Accelerating Raycast in the Unity Game Engine

Viktor Meyer viktorme@kth.se
Robin Dahlqvist robindah@kth.se

June 15, 2020



1 Abstract

An accelerated raycast operation is proposed for the Unity game engine. High performance bounding volume hierarchies and efficient ray-box intersection tests are implemented in C++. A native plug-in with an easy-to-use API is constructed and integrated into Unity. Benchmarks are conducted that compare our implementation to Unity's built-in reference solution.

2 Introduction

In 1972, the video game *PONG* was released to the public in the form of arcades [Wol08]. The video game consisted of two players, each controlling a paddle at opposite sides of the screen. Centered at the screen was a ball that bounced from side to side. The objective was to score a goal by skillfully bouncing the ball to the other players side. PONG was not the first video game to ever be released, however, it was the first video game to become a major hit. This is why PONG is significant to this day, it introduced the world to an alternative world, the world of video games.

It is safe to say that since then, video games have progressed at a rapid pace, becoming more advanced and sophisticated [Wol08]. Decades of progression has seen video games transform from an electronic novelty into what it is today, a major worldwide industry. The current video games are almost if not completely indistinguishable from their ancestors. An excellent example of this is *World of Warcraft*, a Massively Multi-Player Online Role-Playing Game (MMORPG).

World of Warcraft contains a massive world where players can interact inside an expertly crafted three-dimensional environment [Duc+06]. The world hosts many features such as creatures and quests for its players to enjoy. Nearly everything inside World of Warcraft occurs in real-time, this means that players from all over the globe can freely interact and communicate with each other.

In 2006 the total active player count for World of Warcraft surpassed 6 million [Duc+06]. The game became the largest and arguably most complex video game in the world. From a technical standpoint, the differences between PONG and World of Warcraft are night and day. There are a multitude of challenges that need to be solved in order to facilitate all the features in modern video games, especially from an engineering standpoint.

3 Problem description

With player counts reaching previously unseen numbers, an important focus within the games industry has become to support as many concurrent players as possible [AB04]. To achieve this goal it is essential that every aspect of modern games are optimized and achieve high frame rates.

The *raycast* operation is commonly found in most major game engines [Tec20b; Gam20]. Raycasting provides the functionality of testing whether a ray is intersected by other primitives. This particular functionality is frequently required when building complex games and can require a lot of computational power.

This project explores the problem of optimizing and potentially improving the raycast operation within a context of the Unity game engine [Tec20d].

4 Raycast

As briefly stated in Section 3, the raycast operation provides the functionality of testing whether a ray is intersected by other primitives. This section introduces common definitions and a more in-depth description of raycasting. A *ray* originates from a point $\overrightarrow{O}$ in space and travels in an arbitrary direction $\overrightarrow{D}$ [Tec20b]. This implies that a ray can be defined by its pair of *origin* and *direction* vectors.

Figure 1: Ray definition

The raycast operation accepts a ray as an argument and tests if the ray intersects any primitives along its direction vector [Tec20b]. If no intersections are found, the operation simply returns $false$.

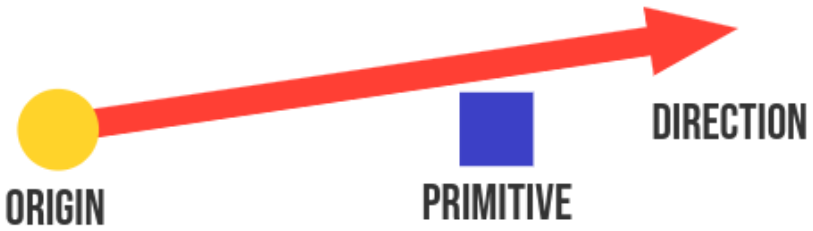


Figure 2: A raycast without intersection returns false

If an intersection is found, the operation returns *true*. It is worth noting that most raycast implementations also return details about the intersected primitive such as distance and point of intersection [Tec20c].

Figure 3: A raycast with intersection returns true

5 Hierarchical Spatial Data Structures

Data structures are used to store and work with different types data [Sam89]. Spatial data structures are specifically concerned with data that is more than one dimensional. Hierarchical spatial data structures are spatial data structures that are based on recursive decomposition. Examples of spatial data structures include bounding volume hierarchies (BVH), octrees and uniform grids [STL14].

Bounding volume hierarchies partition primitives into disjoint sets and are traditionally implemented using binary trees [STL14]. At root level, a tree contains all primitives. Each child node in the tree partitions its parents primitives into two disjoint sets.

Octrees work by subdividing all primitives into octants and are traditionally implemented with octonary trees [STL14]. In comparison to the BVH, the children in an octree usually occupy equal amounts of volume.

Uniform grids splits all primitives into equally spaced axis aligned bounding boxes [STL14]. Each cell contains the corresponding primitives that are encompassed by its bounding box.

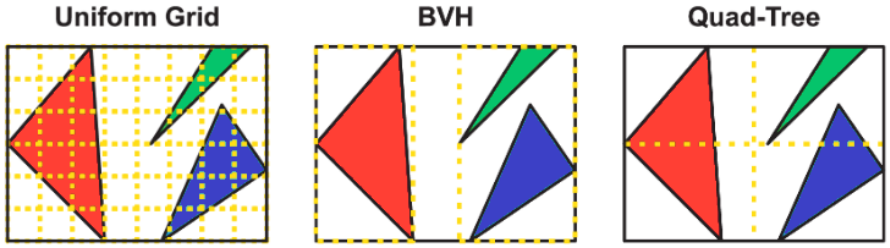


Figure 4: Spatial data structures illustrated in 2D, adapted from [STL14]

Spatial data structures can be used to improve operations such as raycasting [STL14]. The idea is to exploit the spatial partitioning inherent in the data structures to effectively avoid testing ray-intersection for all primitives.

Octrees can be effective for raycasting, however, a critical downside is the considerable time it takes to generate the structure [STL14]. This makes octrees a bad choice for the real-time applications that this project is concerned with. Uniform grids are fast to generate and can provide solid performance under select circumstances. Bounding volume hierarchies offer high performance while still being relatively fast to generate.

Due to limited time this project focuses exclusively on bounding volume hierarchies in the real-time context of games.

6 Implementation

6.1 Bounding volume hierarchy construction

T. Karras published an excellent paper in 2012: *Maximizing parallelism in the construction of BVHs, octrees, and k-d trees* [Kar12]. The paper described ideas that improved construction times for bounding volume hierarchies through parallelism. As such, this project implements the ideas presented in the 2012 paper and integrates them into the Unity game engine.

The first step is to order all primitives by a space filling curve, specifically through the use of *morton codes* [Kar12]. Morton codes provides a mapping from multi-dimensional space to one-dimensional space [AS97]. This enables domain decomposition and the possibility for parallelization [Kar12]. The *libmorton* C++ library was selected for efficient encoding of morton codes [Bae18].

After assigning morton codes to all primitives the codes need to be sorted in ascending order [Kar12]. To speed up this process a parallel sorting algorithm is used from the C++ *Parallelism TS* [Cpp18].

The next step is to construct a *binary radix tree* where each internal node partitions its keys by first differing key bit [Kar12]. Tree construction is done fully in parallel for each internal node. Detailed description of the binary tree construction algorithm is referred to the original paper for completeness.

Finally the binary radix tree is used to compute bounding volumes [Kar12]. Computing bounding volumes is a parallel procedure that starts at the leaf nodes in the binary radix tree and traverses upwards. At each node in the binary radix tree, a bounding volume that fits its children are calculated. Atomic *fetch_add* instructions are used to ensure that nodes are processed once [Cpp19].

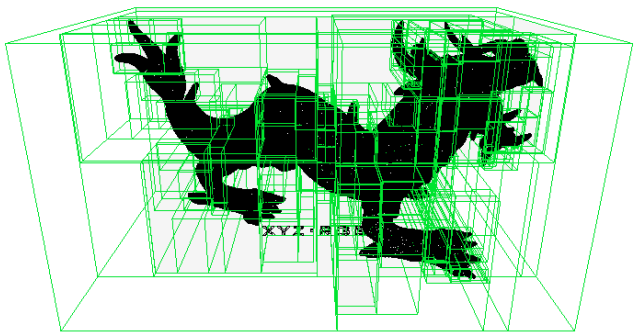


Figure 5: Stanford XYZ RGB Dragon encapsulated by bounding volumes

6.2 Ray-Box intersection

Ray-Box intersection tests are the core of this projects raycasting operation and therefore performance critical. Ray-Box intersection tests can be a bottleneck and computationally expensive when executed at scale [Wil+05]. To achieve a high standard of performance, a variant of the Williams et. al. paper is used: *An Efficient and Robust Ray-Box Intersection Algorithm.*

6.3 Accelerating raycast

The bounding volume hierarchy is used to improve upon the Unity built-in raycast operation. The hierarchy serves as an acceleration structure with the benefit that not all primitives in a scene need to be explicitly tested.

When a raycast operation is executed, traversal starts at the root node in the bounding volume hierarchy [STL14]. From this point, a simple recursive procedure takes place. If no intersection with the current node is present, no further tests are required and the operation returns false. If there is intersection with the current node, the current node's children are recursively tested for intersection until termination. The key advantage is that if a ray does not intersect one of the children's bounding volumes, that entire branch is effectively dismissed.

6.4 Unity integration

Since the majority of code is written in C++ for performance reasons, an integration with the Unity game engine is necessary. This is done through a *Native plug-in* that enables Unity to communicate from its managed C# runtime with unmanaged C++ [Tec20a].

An *API* is exposed from C++ using *extern "C"* to prevent name mangling issues [Tec20a]. This API is then imported to Unity in the form of a dynamically linked library (DLL) and loaded at runtime by the engine.

The API is designed to be as similar to the standard Unity raycast workflow as possible. This design decision was made in an effort to achieve ease of use. However, a notable difference is that this project's API requires the construction of a bounding volume hierarchy before any raycasts can be executed.

```
public class BoundingVolumeHierarchy
{
    /// <summary>
    /// Create a bounding volume hierarchy.
    /// Generates binary radix tree, builds tree structure from bounds.
    /// </summary>
    public BoundingVolumeHierarchy(List<GameObject> objects);

    /// <summary>
    /// Tests a ray against the bounding volume hierarchy
    /// </summary>
    /// <returns>true iff ray intersects an object in the tree</returns>
    public bool Raycast(Ray ray);
}
```

7 Experiments

Different experiments are conducted that try to highlight the differences in efficiency between our implementation and Unity's. The experiments investigate total runtime and breakdowns of critical sections that are computationally heavy. Tested implementations in this experiment are Unity's built-in raycasting and our integrated BVH. Our BVH implementation is tested in two different scenarios: each ray is cast separately (BVHSingle), and another scenario where a batch of rays is cast (BVHBatch). For each experiment, the amount of primitives in the scene ranges from 10 to 500 000 primitives. In an attempt to minimize error, each experiment is run several times with the same variables.

Additional steps are made to try and ensure accuracy of the measurements. The experiments should only measure the computational part of raycasting, and not the other systems contained within Unity. Rendering can impact performance a lot which is why all experiments are conducted without scene cameras to avoid any render overhead. This effectively makes the computational workload more isolated and gives more processing power to the raycast computations that are of primary interest.

7.1 Time Consumption

The time consumption experiment provides a breakdown of the internal stages in each raycast implementation. More specifically, measurements are provided for: initialization - creating acceleration structure, construction - creating ray objects and work - casting the rays. This breakdown attempts to highlight what parts of the implementations are the most heavy to compute.

7.2 Performance

The performance experiment compares the overall performance of the different implementations. This experiment measures the time it takes to execute each implementation for a specific number of primitives in a scene.

8 Results

Each experiment was conducted with 10, 100, 1000, 10000, 150000, 200000, 250000, 300000 and 500000 primitives. Each experiment was run 5 times, the results represent either the median or average values of an execution. Figures 6-8 visualizes the time spent in each implementations internal stages (Time Consumption experiment). Figures 9-10 visualizes the overall runtime of each implementation (Performance experiment).

8.1 Experiment - Time Consumption

The following charts depict time spent in each implementations internal stage:

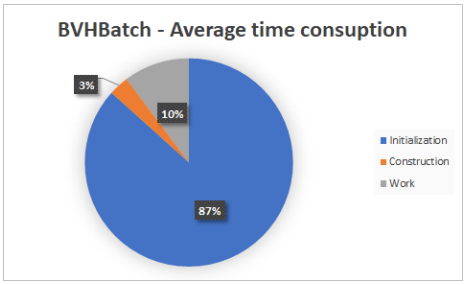


Figure 6: The average time consumption for BVHBatch

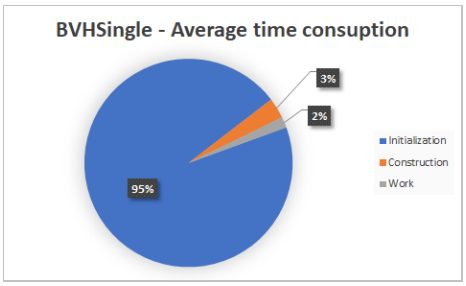


Figure 7: The average time consumption for BVHSingle

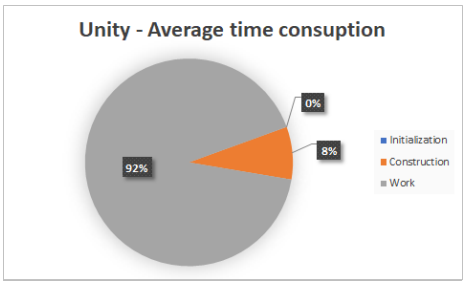


Figure 8: The average time consumption for Unity's built-in raycast

It is evident from Figure 6 and Figure 7 that roughly 90 percent of the total runtime is required for initialization of the bounding volume hierarchy structure. Meanwhile the construction of rays and raycasting itself only amounts to a small fraction of the total runtime.

Figure 8 differs from Figure 6-7 in that the work stage is consuming most time instead of initialization. This is unsurprising since the Unity raycast implemention is unlikely to use an expensive acceleration structure such as a bounding volume hierarchy.

8.2 Experiment - Performance

Figure 9 presents the performance for each of the implementations using a variable amount of primitives. All of the implementations appear to behave according to a similar pattern. Unity's built-in raycast operation show superior runtime performance for almost all primitive counts.

Figure 9: Median runtime for each implementation for specific primitive count

Initialization is a very time consuming stage for the BVH implementations. It is interesting to see what the runtime would look like if initialization could be removed entirely by e.g pre-computation.

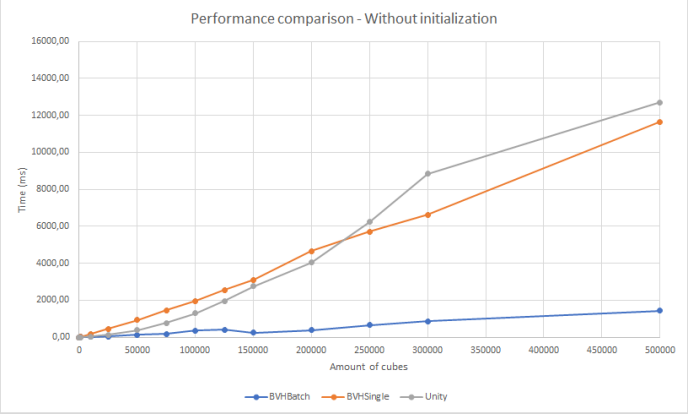


Figure 10: Median runtime for each implementation for specific primitive count, but without the initialization runtime.

When removing initialization cost from the BVH implementations the results are noticeably different. Figure 10 shows that BVHSingle has seen little improvement. However, much more interesting is the change in runtime for BVHBatch.

9 Discussion

Is our implementation of BVHs within the context of the Unity game engine better than the built-in raycast operation? The answer is that it highly depends on how the results are interpreted. Referring to Figure 9, it is possible to conclude that the experiments for each implementation did not show any significant differences or improvements. We argue that this conclusion is not fair since the BVH structure is integrated into Unity through a native plug-in. This leads to a number of complications when it comes to initializing the BVH acceleration structure. When BVH is initialized it has to read and copy all primitives from managed (C#) to unmanged (C++). This leads to dynamic memory allocations that are very large and also results in poor use of cache. Our implementation of BVH does not make any significant efforts at reducing interop overhead. We argue that a real world implementation would be able to reduce the interop and initialization overhead significantly. This would likely yield a result that is much closer to Figure 10 than Figure 9.

What also needs to be discussed is accuracy of results. When measuring runtimes for specific stages of each implementation, we observed that runtimes could differ a lot from when measured as a whole. A good example of this is the BVHSingle implementation, referring to Table 1. When measuring total runtime for 500.000 primitives it had a median runtime of 14377 ms. However, when aggregating independent measurements of the three internal stages it amounts to an average runtime of 2866 ms. The independently aggregated value is only 20 percent of the total measured runtime which suggests measurement error. For very small time measurements it is understandable if some milliseconds vanish, but in this particular case there is a whole 11500 milliseconds missing. This is a critical error that has to be solved for future data collection and the unfortunate effect of incorrect timing or low resolution clocks.

Table 1 Runtime for BVHSingle

Cubes	Whole	Total	Initialization	Construction	Work
10	0,00	0,00	0,00	0,00	0,00
100	1,00	0,08	0,08	0,00	0,00
1000	16,00	3,00	2,80	0,00	0,20
10000	211,00	42,92	40,68	1,64	0,60
25000	571,00	113,28	107,24	4,44	1,60
50000	1147,00	232,12	219,80	8,88	3,44
75000	1802,00	363,20	344,16	13,80	5,24
100000	2424,00	494,80	469,04	18,76	7,00
125000	3150,00	626,64	594,08	23,76	8,80
150000	3844,00	781,44	739,16	31,60	10,68
200000	5754,00	1141,16	1087,76	39,24	14,16
250000	7052,00	1420,36	1353,12	49,52	17,72
300000	8171,00	1640,88	1555,08	64,44	21,36
500000	14377,00	2865,76	2720,08	109,76	35,92

10 Future work

An implementation such as our bounding volume hierarchy raycast requires a lot of testing and experimenting in order to obtain good and reliable data. Hence this section is mostly about ideas related to obtaining better future data.

10.1 More samples

It is never a disadvantage to obtain too much benchmark data. In this experiment only specific primitive counts were tested. Looking at the data that was obtained in Figure 9 it appears to scale linearly with regards to primitives. It is possible that data between sample points could yield different results such as spikes in runtime. The reason for having large steps between sample points was time limitation since the runtime of each experiment was very high. A smaller step between experiment sample points would be more ideal.

10.2 More accurate samples

Similarly to the previous improvement, more samples for each tested size would increase confidence in the credibility of each measurement. In the measurements it was noticeable that some values appeared to be outliers. Having many more samples of the same tested size is something that is needed to improve confidence in our results. Due to long runtime, the number of samples was reduced for the data used in the Results section.

10.3 Different type of primitives

In our experiments there were only one type of primitive, cubes. It would be interesting to explore the effect of other primitives, that being said, the bounding volume hierarchy is primarily concerned with cubes.

10.4 3D space

When conducting experiments, the positions of each primitive was randomly generated and bounded by a certain space, in this case a box with each edge being 200 units long. In effect, all primitives were spawned inside the space of this box. Hence for very large sets of primitives, most of the primitives in the space most likely intersected with other primitives. For future experiments, different space layouts should be tested in order to see their effect with regards to performance.

10.5 Other data structures

Only bounding volume hierarchies (BVH) were experimented with. For future work, comparisons and experiments with other spatial data structures such as octrees and uniform grids could be implemented and tested as well [STL14].

References

[Sam89] Hanan Samet. "Hierarchical spatial data structures". In: *Symposium on Large Spatial Databases*. Springer. 1989, pp. 191–212. URL: http://www.cs.umd.edu/users/hjs/pubs/SametSSD89.pdf.

[AS97] Srinivas Aluru and Fatih Erdogan Sevilgen. "Parallel domain decomposition and load balancing using space-filling curves". In: *Proceedings Fourth International Conference on High-Performance Computing*. IEEE. 1997, pp. 230–235.

[AB04] A. Abdelkhalek and A. Bilas. "Parallelization and performance of interactive multiplayer game servers". In: *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* 2004, pp. 72–. URL: https://ieeexplore.ieee.org/document/1303003.

[Wil+05] Amy Williams et al. "An Efficient and Robust Ray-Box Intersection Algorithm". In: *ACM SIGGRAPH 2005 Courses*. SIGGRAPH '05. Los Angeles, California: Association for Computing Machinery, 2005, 9–es. ISBN: 9781450378338. DOI: 10.1145/1198555.1198748. URL: https://doi.org/10.1145/1198555.1198748.

[Duc+06] Nicolas Ducheneaut et al. "Building an MMO With Mass Appeal: A Look at Gameplay in World of Warcraft". eng. In: *Games and Culture* 1.4 (2006), pp. 281–317. ISSN: 1555-4120. URL: https://journals.sagepub.com/doi/10.1177/1555412006292613.

[Wol08] Mark J. P. Wolf. *The video game explosion: a history from Pong to Playstation and beyond*. Greenwood Press, 2008. URL: https://books.google.se/books?id=XiM0ntMybNwC.

[Kar12] Tero Karras. "Maximizing parallelism in the construction of BVHs, octrees, and k-d trees". In: *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*. 2012, pp. 33–37.

[STL14] Artur Lira dos Santos, Veronica Teichrieb, and Jorge Lindoso. "Review and comparative study of ray traversal algorithms on a modern gpu architecture". In: (2014). URL: https://otik.uk.zcu.cz/bitstream/11025/11931/1/Santos.pdf.

[Bae18] Jeroen Baert. *Libmorton: C++ Morton Encoding/Decoding Library*. https://github.com/Forceflow/libmorton. 2018.

[Cpp18] CppReference. *Extensions for parallelism - cppreference.com*. 2018. URL: https://en.cppreference.com/w/cpp/experimental/parallelism.

[Cpp19] CppReference. *Atomic operations library - cppreference.com*. 2019. URL: https://en.cppreference.com/w/cpp/atomic.

[Gam20] Epic Games. 2020. URL: https://docs.unrealengine.com/en-US/Engine/Physics/Tracing/Overview/index.html.

[Tec20a] Unity Technologies. *Unity - Manual: Native plug-ins*. 2020. URL: https://docs.unity3d.com/Manual/NativePlugins.html.

[Tec20b] Unity Technologies. *Unity - Scripting API: Physics.Raycast*. 2020. URL: https://docs.unity3d.com/ScriptReference/Physics.Raycast.html.

[Tec20c] Unity Technologies. *Unity - Scripting API: RaycastHit*. 2020. URL: https://docs.unity3d.com/ScriptReference/RaycastHit.html.

[Tec20d] Unity Technologies. *Unity - Unity*. 2020. URL: https://unity.com/.

11 Appendix

11.1 Individual contribution

Viktor Meyer

Bounding volume hierarchy (C++)

Ray-box intersection test (C++)

Unity native plug-in (C#, Unity)

Written report (pp. 1-6)

Robin Dahlqvist

Performance evaluation (C#, Unity)

Written report (pp. 7-11)

11.2 Source code

https://github.com/VIGGEEN/VIROBVH