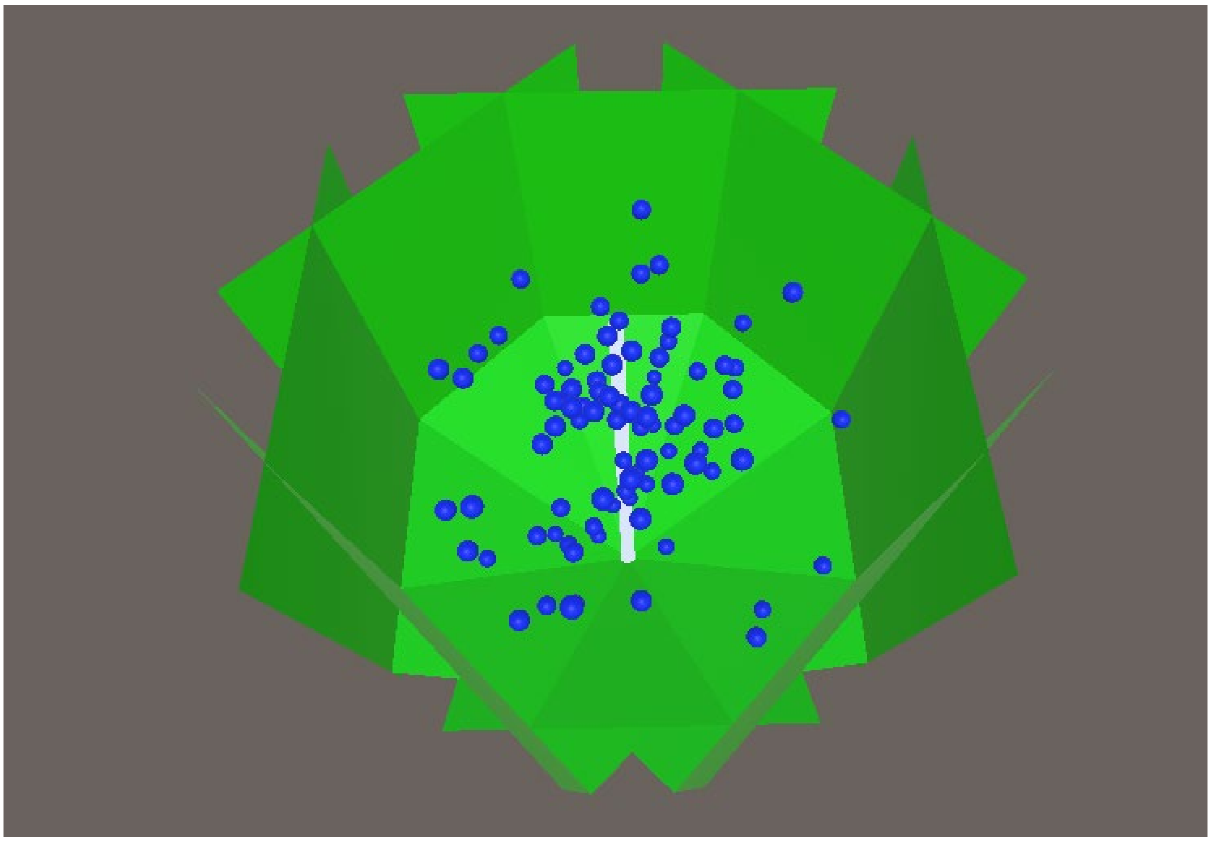

TTT Evaluation Report for Collision Detection in Unity

Author: G.S. van den Ham

Date: 23-02-2025

Module: 2.3 Advanced Tools

Saxion University of Applied Sciences

1. Introduction

Collision detection is a crucial component in real-time simulations and game development. In this report, I analyse the impact of different collider types on **performance (FPS) and active objects** in **Unity's default physics engine**. This study is conducted using a controlled spawning system, measuring FPS across different object counts.

---

2. Research Question

Evaluate the performance of different collision detection methods in Unity by spawning spheres using a fixed random seed and comparing sphere colliders, mesh colliders and capsule colliders.

---

3. Justification of Chosen TTT

Collision detection is a fundamental aspect of real-time physics simulations, directly influencing the performance and realism of applications such as video games, simulations, and virtual reality. The selection of specific collider types—Sphere, Capsule, and Mesh Colliders—in this study aligns with practices observed in various industry-standard physics engines.

9.1 Industry Examples Utilizing Similar Collision Detection Techniques

1. **Unreal Engine**

Unreal Engine, developed by Epic Games, employs a comprehensive collision system that utilizes both simple and complex collision shapes. Simple collisions are defined using basic geometric primitives such as boxes, spheres, capsules, and convex hulls. These primitives are computationally efficient and are commonly used for dynamic objects to optimize performance. Complex collisions, on the other hand, rely on detailed mesh geometry and are typically reserved for static objects where precise collision detection is necessary. This distinction allows developers to balance accuracy and computational load effectively.

2. **Havok Physics Engine**

The Havok Physics engine, integrated into various game development platforms, offers a robust collision detection system that differentiates between simple and complex collisions. Simple collisions utilize basic shapes like boxes, spheres, and capsules, which are computationally less intensive and suitable for dynamic objects. Complex collisions involve detailed mesh geometries, providing precise collision detection for static objects. This approach enables developers to optimize performance by selecting appropriate collision types based on the object's nature and interaction requirements.

3. **Bullet Physics Library**

The Bullet Physics Library is an open-source physics engine that provides real-time collision detection and multi-physics simulation capabilities. It supports various collider types, including primitive shapes and complex mesh colliders, allowing developers to balance accuracy and performance based on application requirements. Bullet is utilized in diverse applications, ranging from virtual reality simulations to robotics and game development.

4. Experiment Setup

To evaluate the performance impact, I implemented a Spawner system that generates objects at a fixed rate, each equipped with a SphereCollider, CapsuleCollider, or MeshCollider.
The DataCollector script records FPS, active object count, and collisions every 100ms, logging the data for analysis.

To keep the scene in motion I have implemented a moving object in the scene that interacts with the spheres

4.1 Test Cases

I conducted tests under two different object loads:

1. **250 Objects** over **10 seconds**
2. **5000 Objects** over **100 seconds**

---

5. Data Collection Method

The unity project is built to run as a standalone application through the Unity Game Engine. The data is collected by running the exe file with different settings set through a setting.txt file on a PC.

5.1 Technical specs test case

The build was executed on a machine with the following specifications:

CPU: 12th Gen Intel(R) Core(TM) i7-12650H
GPU: NVIDIA GeForce RTX 4050 Laptop GPU
RAM: 16 GB
SSD: NVMe Micron_2450_MTFDKBA1T0TFK

5.2. Workflow build

The build calculates the following data to be gathered:

- **FPS Calculation:** Calculated per frame in **Update()** and logged every 100ms.
- **Active Object Count:** Tracked via a list of all spawned objects.
- **Collision Count:** Tracked using OnCollisionEnter() for all objects.

This is then stored in a CSV file, from there the data can be displayed in graphs. To get accurate data the tests were conducted 5 times per testcase and the averages were used in the charts.

---

6. Results & Analysis

Below, I present a comparison of FPS against the number of active objects for each test case.

6.1 FPS vs. Active Objects (250 Objects)

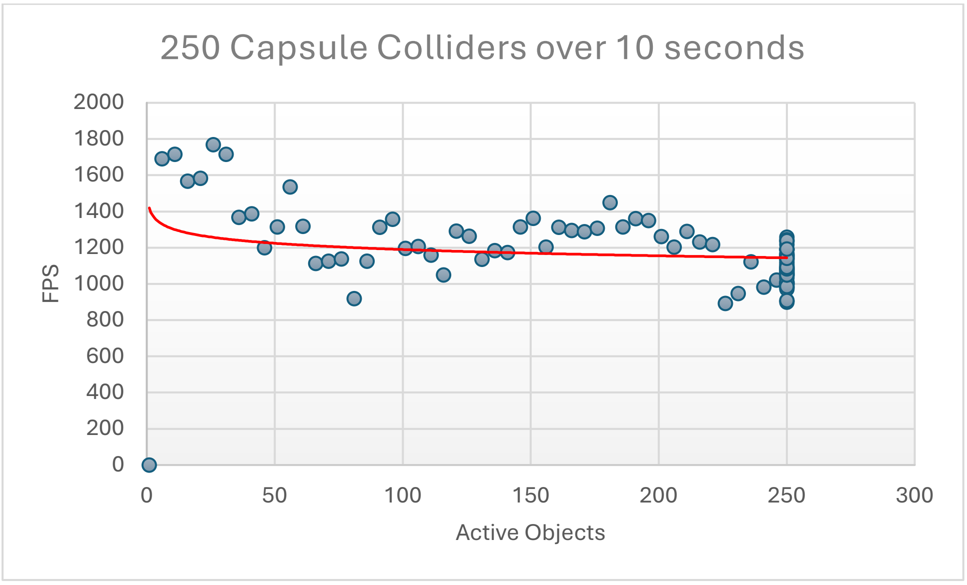

Figure 1: 250 Capsule Colliders over 10 seconds

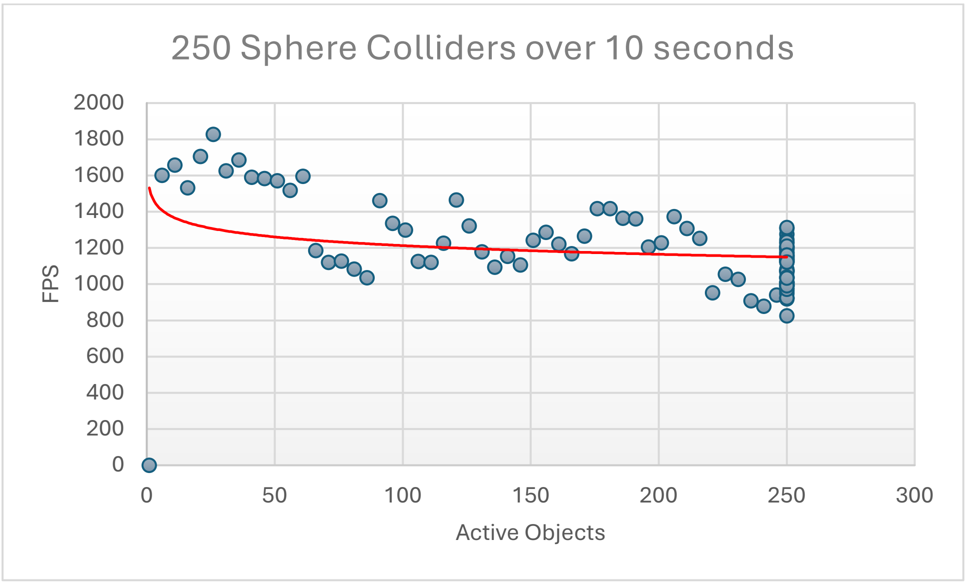

Figure 2: 250 Sphere Colliders over 10 seconds

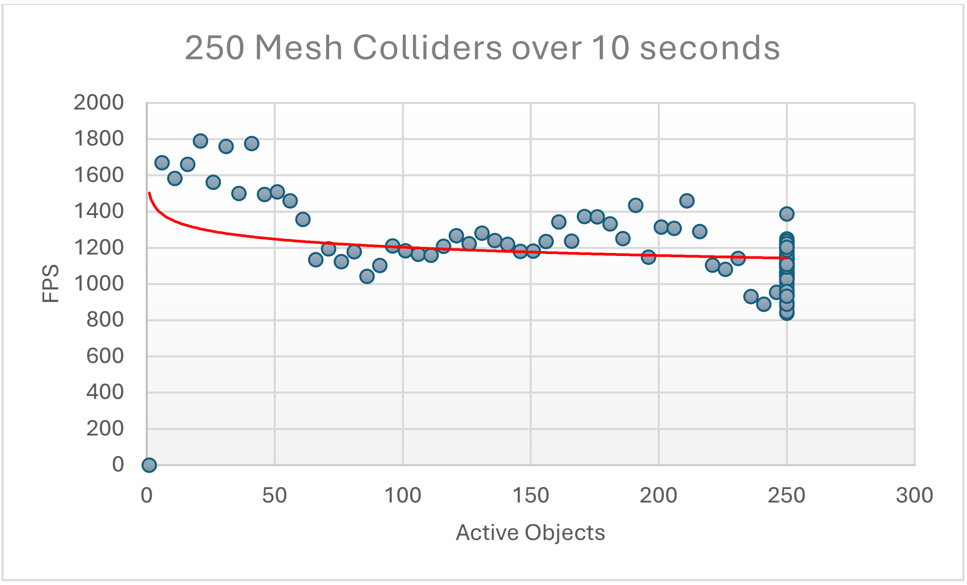

Figure 3: 250 Mesh Colliders over 10 seconds

As seen in the graphs, FPS remains relatively stable initially but drops as more objects interact. The trendlines look about the same leading to the conclusion that no real difference can be noted from such a small amount of active objects.

6.2 FPS vs. Active Objects (1000 Objects)

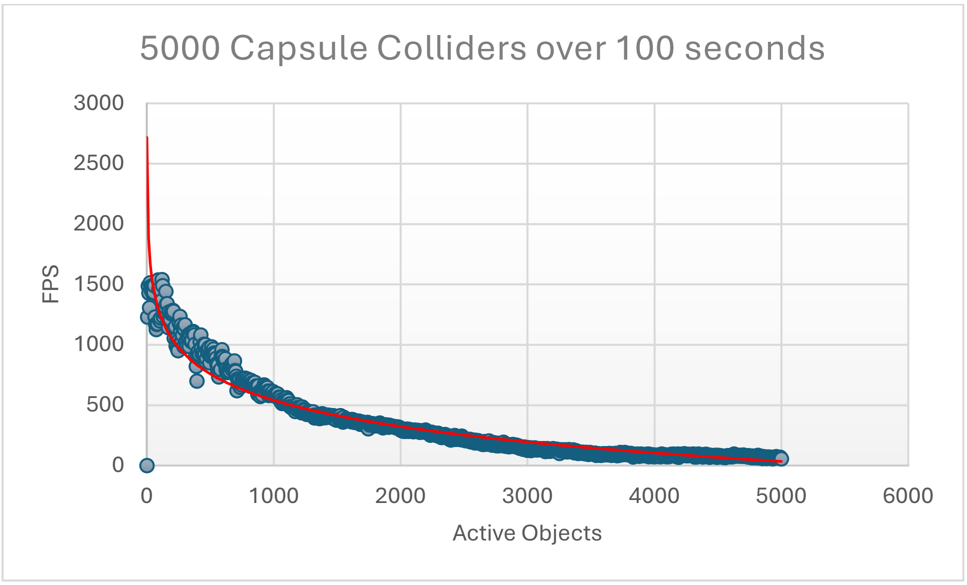

Figure 4: 5000 Capsule Colliders over 100 seconds

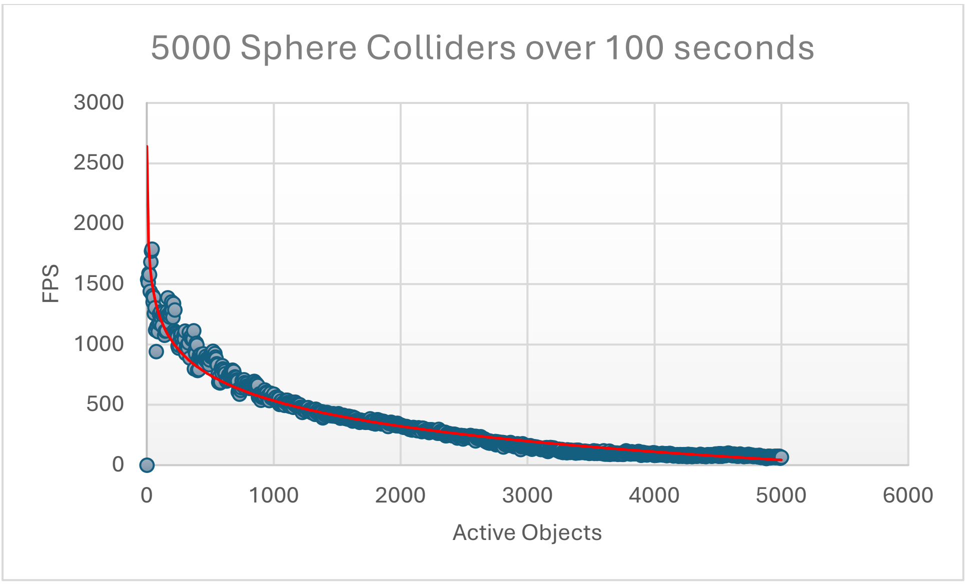

Figure 5: 5000 Sphere Colliders over 100 seconds

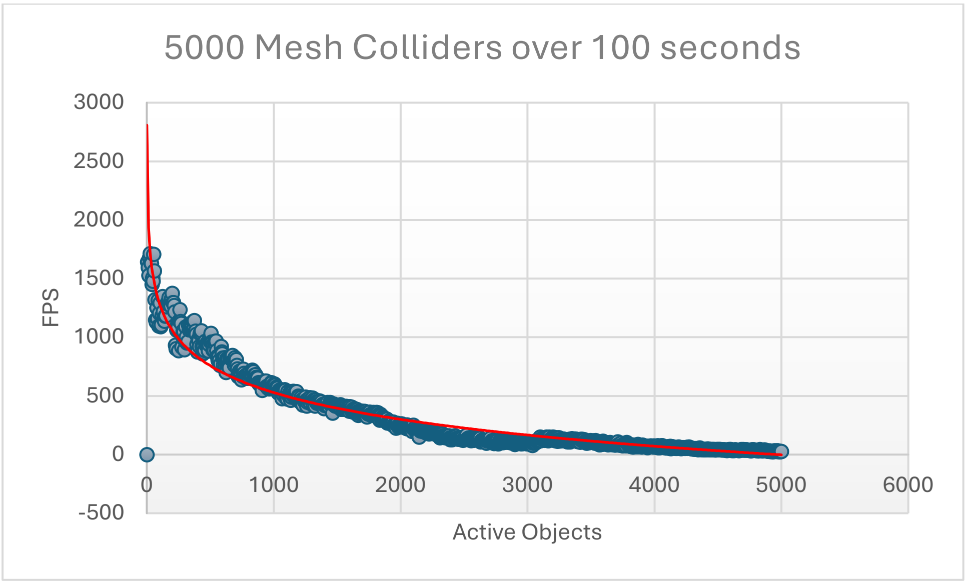

Figure 6: 5000 Mesh Colliders over 100 seconds

The FPS drop is more noticeable as Unity processes significantly more collision checks. Although the charted trendline looks about the same for all three colliders, there is a difference when it comes to the data. The data shows differences in frame rate from about 10 to 15 FPS. When the FPS is above 120 this is not noticeable but when the FPS is beneath 60, which it is in all three testcases around the 3000 objects mark, this is a significant difference. The Mesh, Sphere and Capsule colliders perform with about the same efficiency with a smaller number of objects but when the object counter reaches about 1000 you already see more efficiency in the sphere and capsule collider with the capsule collider being the most efficient.

7. Conclusion

From the results, I observe the following trends:

- When using **small amounts of objects,** the type of collider does not seem to have significant impact on performance.
- **Higher object counts lead to exponential FPS drops.**
- **Mesh Colliders** significantly reduce performance due to complex shape calculations.
- **Capsule Colliders** are the most efficient due to their simpler mathematical calculations.
- **Collision detection in Unity scales poorly when object numbers exceed 1000+**, suggesting the need for optimizations.

---

8. Recommendations

- **Use primitive colliders (Sphere, Capsule) whenever possible** to optimize performance.
- **Optimize collision layers and physics settings in Unity** to reduce unnecessary calculations.

---

9. Sources

Collision. (n.d.). Retrieved from Epic Games: https://dev.epicgames.com/documentation/en-us/unreal-engine/collision-in-unreal-engine

Unreal Engine Support for Simple Collisions. (n.d.). Retrieved from Github: https://weisl.github.io/collider_ue_guidline/

Appendices

Appendix A: Raw Data (CSV Logs)

See data folder for .csv files and combined Excel files