Collision Detection Evaluation Report  
~ by Δtime and Collision Check Count ~

Maria Cosneanu

Table of Contents

[Evaluation Proposal 3](#_Toc202055970)

[1. Introduction 3](#_Toc202055971)

[2. Testing Environment 3](#_Toc202055972)

[3. Scripts 4](#_Toc202055973)

[3.1 TestManager.cs 4](#_Toc202055974)

[3.2 CollisionLogger.cs 5](#_Toc202055975)

[4. Experiments Analysis 6](#_Toc202055976)

[4.1 Static-Environment 6](#_Toc202055977)

[Static Mesh Collider 6](#_Toc202055978)

[Kinematic Mesh Collider 6](#_Toc202055979)

[Continuous Dynamic Mesh Collider 7](#_Toc202055980)

[4.2 Dynamic Prefab Comparison 8](#_Toc202055981)

[Dynamic Convex Mesh Collider 8](#_Toc202055982)

[Dynamic Compound Collider 8](#_Toc202055983)

[5. Discussion & Conclusions 10](#_Toc202055984)

[Repo link 10](#_Toc202055985)

Evaluation Proposal  
“In Unity, I want to compare per-frame deltaTime and collision-check counts for concave mesh colliders (static vs kinematic-default vs kinematic-continuous-dynamic) and for convex vs compound colliders on moving prefabs (using a fixed random seed and varying object counts)”

# 1. Introduction

This project was made to compare different colliders/rigidbody modes by logging Δtime per-frame and counting collision checks against our target collider. Doing that isolates how much CPU work each setup costs.

This report presents two sets of experiments:

* **A static environment**: a single mesh collider tested in static, kinematic-discrete, and kinematic-continuous-dynamic modes against thousands of spheres
* **Dynamic prefab comparison**: an ever increasing number of “tree” prefabs, each using either a convex mesh or a compound collider, colliding with walls and each other

# 2. Testing Environment

* **Unity version**: 6000.0.35f1
* **Scene**: six static walls forming a closed box (BoxCollider, no Rigidbody)
* **Random seed**: same seed at the start of every test run (Random.InitState(12345))
* **Warm-up-Measure Loop**: 3s warm-up (with no logging), then 10s measurement window
* **Spawn counts**
  + Static test: 250 to 5000 spheres in increments of 250, plus extra steps at 6000, 7000, 8000, 9000, 10000
  + Dynamic test: 100 to 2000 tree prefabs in increments of 100, plus extra steps at 2500 and 3000
* **Data recorded**:
  + Δtime per frame / average Δtime (seconds/frame)
  + Collision checks per frame: total contact points against the target collider

# 3. Scripts

## 3.1 TestManager.cs

TestManager spawns prefabs according to specific parameters and logs collision counts in a csv file.

**A screen shot of a computer

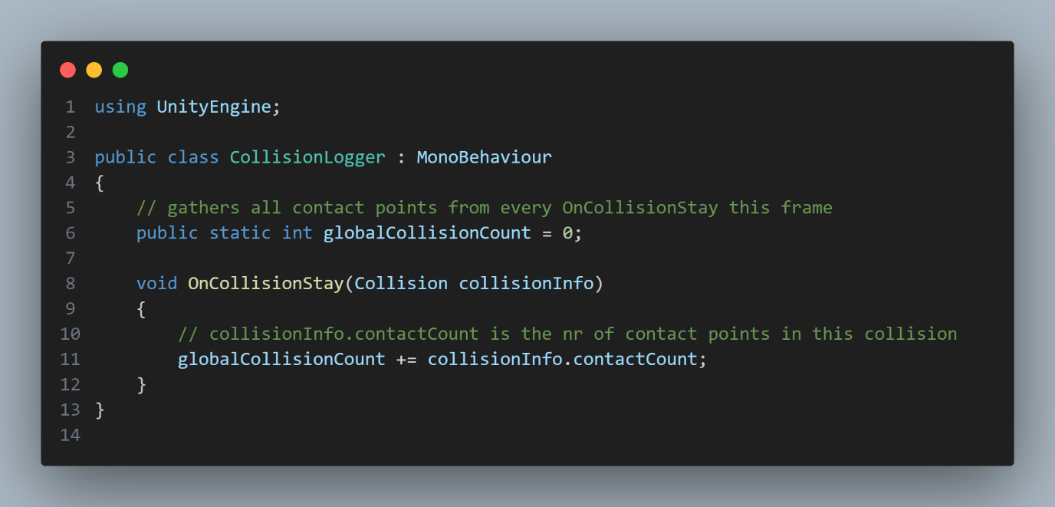
AI-generated content may be incorrect.**

**A screen shot of a computer

AI-generated content may be incorrect.**

## 3.2 CollisionLogger.cs

A very basic script that counts the number of collisions with this specific gameobject.

****

# 4. Experiments Analysis

## 4.1 Static-Environment

### Static Mesh ColliderA screenshot of a graph AI-generated content may be incorrect.

**Behaviour:**

* Treated as an immovable, non-convex collider. Only broad-phase checks against moving spheres

**Performance:**

* Small scale: lowest Δtime and checks, least overhead
* Large scale: growth identical to other modes so overhead advantage disappears

**Best use:**

* Mesh never moves
* Total colliders remain low ( less than 2000)

### Kinematic Mesh Collider A screenshot of a graph AI-generated content may be incorrect.

**Behaviour:**

* Moves under script control, but still only broad-phase checks against moving spheres

**Performance:**

* Small scale: slightly heavier at very low counts (some extra transform updates)
* At mid-range (2000-6000) performs slightly better

**Best use:**

* When you need movement through scripts

### Continuous Dynamic Mesh Collider

A screenshot of a graph

AI-generated content may be incorrect.

**Behaviour:**

* Continuous collision detection enabled

**Performance:**

* Looks like highest overhead at low counts
* No advantage in this static situation

**Best use:**

* When mesh moves fast enough to tunnel through other colliders

All three collider types have very similar average delta-time curves, difference is less than 0.002ms even at 10000 spheres spawned.

A graph with lines and numbers

AI-generated content may be incorrect.A graph showing a graph of a graph

AI-generated content may be incorrect.The static mesh collider is maybe slightly faster at low counts (250–2000 spheres) but it joins with kinematic and dynamic continuous at higher counts. Total collision checks also track very closely, with dynamic continuous slightly fewer checks around 6000 spheres, but ending up higher at 10000.

## 4.2 Dynamic Prefab Comparison

### Dynamic Convex Mesh Collider

A screenshot of a graph

AI-generated content may be incorrect.

**Behaviour:**

* Treats the entire mesh as a single convex hull

**Performance:**

* Low counts: very cheap, with around 0.02 ms Δtime at 1000 colliders with 5.7M checks
* Mid counts: Δtime jumps rapidly as collision volume rises.
* High counts(over 5000): Δtime stagnates

**Best use:**

* When object is truly convex
* You need really low overhead

### Dynamic Compound Collider

A graph of a function

AI-generated content may be incorrect.

**Behaviour:**

* Breaks down a complex concave mesh into multiple convex submeshes with primitive colliders
* Each child collider participates in broad-phase and narrow-phase tests every frame

**Performance:**

* Low counts: heavy overhead, 4 times higher than convex at 1000 objects
* Mid to high counts: gets to Unity’s FPS cap way faster

**Best use:**

* When you must represent concave shapes
* Really low object counts

Compound colliders have a startup penalty at low counts. Both types converge after 3000 colliders because the collision-check load greatly exceeds any structural overhead.

Use a convex mesh collider when your object is truly convex, since it has the lowest overhead per frame for any number of objects. Only use a compound collider for meshes that are naturally concave and need convex shapes. At low counts, the initialization cost per object is substantial, but after about 3000 instances, it becomes insignificant.

A graph with a line and a graph

AI-generated content may be incorrect.A graph of a graph with blue and orange lines

AI-generated content may be incorrect.

# 5. Discussion & Conclusions

**This project confirmed** that collider mode and mesh composition have a real impact on Unity’s physics performance, but once you get past a few thousand objects, raw collision volume (the O(N²) growth in paired checks) becomes the main factor.

**Purpose**

* Quantify the CPU toll each collider shape and rigidbody mode imposes on Unity’s physics pipeline by measuring per-frame Δtime and collision-check counts

**Key Lessons**

1. **Static Mesh (Concave)**
   * It has minimal per-frame cost. It is ideal for immovable level geometry
2. **Kinematic (Discrete)**
   * It adds about 30% overhead compared to static because of per-frame tree updates
3. **Kinematic (Continuous-Dynamic, CCD)**
   * It adds another 20% or so for sweep tests to prevent tunneling. Only use it when you really need to avoid tunneling
4. **Convex Mesh Colliders**
   * They are the best choice for dynamic objects. They are fast and accurate enough for most moving actors
5. **Compound Colliders**
   * They give a tighter fit for concave shapes but cost about 10–15% more CPU at low object counts (that overhead becomes negligible past about 3 000 instances)

**Scaling Insights**

* Delta time grows roughly linearly with object count (a constant cost per object)
* Collision checks grow on the order of N squared, but each check costs the same
* Use your measured slopes and trendline coefficients to plan how many objects you can afford and pick collider settings that match your performance targets

# Repo link

<https://github.com/morqp/Advanced-Tools-REPO>