Runtime Efficiency of Entity Component System Architectures

Alexander Kyriacou, 103059830
Object Oriented Programming - COS20007
Swinburne University of Technology

Abstract – **Entity Component Systems (ECS) have become an industry standard in video game development design patterns. This research project investigates the runtime efficiency of various ECS implementations in addition to an inheritance based method of a simple visual simulation. Performance variances within the observed data was not found to be within a statistically significant range such that no conclusions can be drawn. It was found that for simple and lightweight entities such as the ones used in this research, performance differences were negligible. This result hints at an external confounding variable, standardizing and limiting performance. As such, further testing is required to investigate possible performance bottlenecks negating differences in various entity architecture runtime efficiencies.**

I. Introduction

Traditionally, games have been made using an object oriented (OO) decomposition into a large class hierarchy of game entities. This includes creating deep inheritance hierarchy's that break down each required entity into tiers of functionality that branch from each other. Each entity represents a tangible game object that encapsulates data and functionality that serves some purpose within a game world. Adding a new entity requires it to either implement its own functionality or inherit it from a base class. While this may work for simple games with less complex functionality, this method of development is soon vulnerable to many problems. This can often lead to entities inheriting unneeded or excess functionality that breach encapsulation principals [1]. It is this problem that design anti-patterns such as 'The Blob' formulate, which describe a single class that monopolizes a game's processing [2]. Additionally, this inherence tree can cause ambiguity in how an entity inherits its functionality. In the example that a class (D) needs to inherit functionality from multiple branches classes (B and C) that both stem from a single base class (A), ambiguity is created as to how class D inherits A's functionality[1] [3]. These deficiencies amalgamate in an ironic fact. The existence of such an inheritance hierarchy within a game's architecture has the potential of hindering a games scalability and ease of maintenance rather than the contrary.

A. *Composition over Inheritance*

The widely adapted approach to this problem is to 'flatten' inheritance hierarchies into a number of individual components. Each component is represented by a single class that is responsible for the methods and fields needed to add its relevant functionality to an entity. From here new game entities can be created as an aggregation of a given permutation from available components. In the case that an entity requires new functionality, a new component can be added keeping the game architecture uncomplicated. Component aggregation means entities only contain the functionality they require to achieve their responsibility. Consequently, this solves the problem of entities being bulked up by inadvertently inheriting irrelevant functionality.

The purpose of this research is to find the method of aggregation (or inheritance) that benefits from the greatest runtime efficiency within a program.

B. *Entities as a component bucket*

One of the most basic pattern for implementing entity-component aggregation is to use entities as a container for its components. Here entities contain no functionality themselves and simply contain methods that let components be added/retrieved to the aggregation. Additionally, components are derived from a base component class that references the entity containing them and an update method that executes its functionality (Figure 1)[2]. This way, components can reach the information of each other through the entity. E.g. A movement component retrieving an entities coordinate component to change its position.

[1] This is referred to as the Deadly Diamond of Death Problem and is why C# does not support multiple inheritance. For greater detail see [3].

[2] It should be noted that the code described within this report is just one of many implementation methods. (The methods discussed is the way it was approached within this research.)

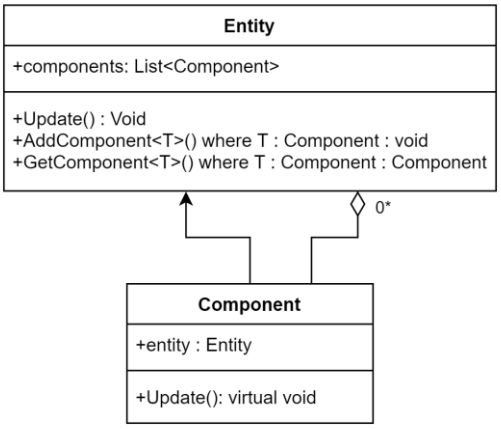

Figure 1: Example implementation of entity and component classes with the entity as a component bucket idea

C. *Entity component system*

Here is where we add the final 'system' part of the entity component system. Essentially, systems extract and separate the functionality of the update method within the previous architecture's component implementation (Figure 2). Here, components become plain old data (POD) types and solely add any new data needed to achieve a specific function. Systems then are able to retrieve the relevant component data stored within an entity and perform its given process. In doing this, systems allow the user to control the order of operations of the game-loop (E.g. allowing all entities to be updated before they are rendered etc.) This removes the complexity issue of inheritance almost completely, giving each class within a program a clear and simple objective/purpose.

D. *CPU cache misses*

Before discussing the final ECS implementation, we first need to discuss CPU caching. While CPU's have benefitted from the exponential performance increase that Moore's law outlines, the increase in RAM performance has not been equivalent. This caused a performance bottleneck given that a CPU can only be as fast as the data being transferred to it. To solve this, CPU's reserve a small amount of memory for caching elements of RAM memory. The CPU cache will store the memory of the RAM that is predicted will be the next required by a process. The CPU cache is filled under the assumption that when a given byte is used, the adjacent byte will be required next and so stores neighboring data into cache memory [4].

If data is required sequentially in the way the CPU predicts, data is retrieved instantly from the cache and a task is able to be performed with little issue. However, if the data required by a task is not found within the CPU cache, the CPU is then required to first load the data from memory [4] [5]. This is referred to as a CPU cache miss and is multiple orders of magnitude slower than a successful cache hit. Cache misses have the potential to shatter a programs performance and are a crucial game design consideration when optimizing performance.

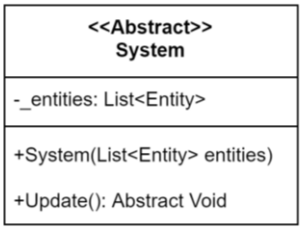

Figure 2: Implementation of base system class within the entity component system architecture

E. *Entities of complete aggregation*

From here we can remove the idea of an Entity entirely and imagine it simple as a sum of loosely linked components. This implementation puts components first within the games architecture with each 'entity' being an interaction grouping between components. For example, if you contained each component type within a list you could treat each index of the list as belonging to a different entity (Figure 3). From here systems only need to request the different component types they require and can simply iterate through to execute the same functionality to each 'entity'. If done correctly this is method is able to avoid the performance concerns that CPU cache misses can cause during a games runtime. This is due to all the required data for of a process being contained within a data structure ordered as it will be needed. Thus, creating a potential for few cache misses. Consequently, an ideal implementation of this ECS method has the theoretical potential to run a game-loop orders of magnitude quicker than its predecessors.

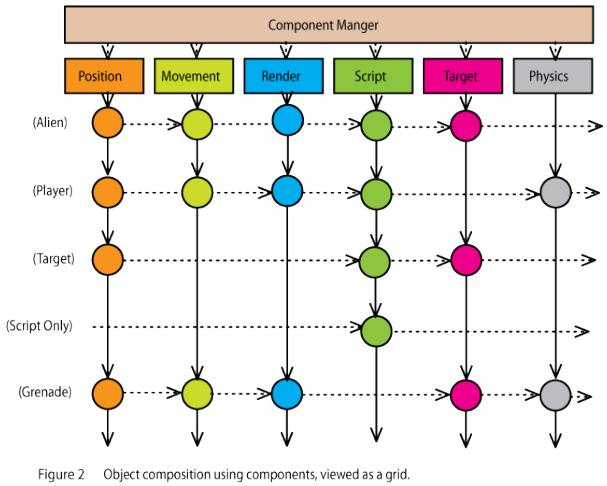

Figure 3: Entities of complete aggregation example. The rows of the grid represent each game entity and the components that make it. The columns of the grid represent each component type. This way, performing a system function requires only iterating through the lists of the components that it requires. Source: [1]

II. Method

A. *Chosen simulation*

In order to compare different implementations of the same model we first needed to create a model. It was chosen to make a program that simulates a number of simple balls bouncing around the borders of a displayed window (Figure 4). This was decided upon for a number of reasons. The first, is that this simulation was simple enough that it can be easily broken

down into its core components. This way it was much easier to create comparable implementations of the same functionality across designs. The second reason was the simulations trivial scalability. By having a single game entity with predictable actions, it refines the number of variables within the simulation. This way, we can increase the entity count consistently across all methods and expect each method to simulate entities of the exact same data. Lastly, this simulation benefits from a great amount of modularity, if extra functionality is required to achieve further complexity, it can be trivially added.

To create this simulation I chose to use the Splashkit library. This decision was less about the functional aspects of the design and more about my familiarity with the library. This allowed for ease of design of the research providing less opportunities for bugs within the code implementation. Using Splashkit allowed me to have a visual element to the simulation which despite not being necessary for the research itself, made it much easier to overview the status of the simulation and detect any problems that may occur.

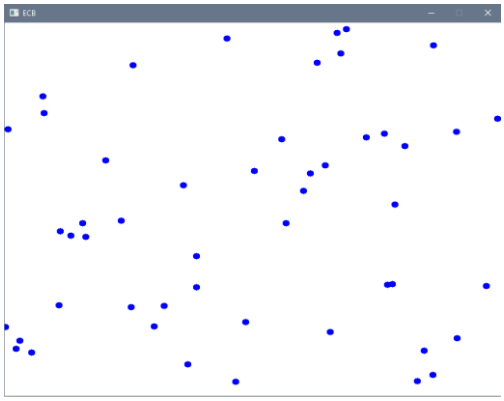

Figure 4: Example of simulation running. Blue balls bounce around the window for a number of seconds before changing the simulation method.

B. Architecture of approaches

Using these considerations I implemented this ball collision model using the 4 different ways described in the introduction of this research. The first method I implemented was an inheritance based model that replicated how a program would be implemented in the case of an inheritance hierarchy based architecture (Figure 5). To do this, a base abstract 'entity' class was created, with simple coordinate fields, constructor and render method. This class was then inherited by a second, abstract, 'dynamicEntity' base class which implemented movement vector fields as well as both a collide and update method. Lastly, a third ball class was created that overrides the methods of the base classes and implements the functionality required of the model. During the simulation runtime, a list of ball objects was initialized and filled with new entities. Meaning that the game loop consisted of a simple iteration through each ball within the list and calling the relevant object functions (Figure 6).

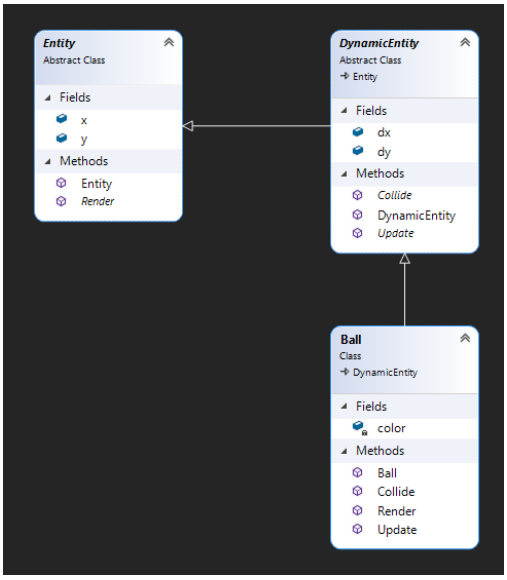

Figure 5: UML outlining the implementation of the entities through inheritance approach

```
//Creation of the simulation entity list
List<Ball> ETI_Entities = new List<Ball>();

//Iterating through the list of balls within the
//relevant game loop
...
foreach (Ball ball in ETI_Entities)
{
    ball.Update();
    ball.Collide();
    ball.Render();
}
...
```

Figure 6: Summary of the simulation process for inheritance based architecture

The second approach, entities as a component bucket, was implemented with 5 different components and an entity structure similar to that described in section 1.A (also seen in Figure 1). The components created for this method are summarized in Figure 7 below. Here the 'entityFactory' class simply contains a single static method, whereby a new instance of each required ball component is created and added to a new entity object.[3] During the simulation runtime, an 'entityManager' class was created and relevant quantity of ball entities were initialized and added to the list. All that was required to update all relevant entities was to call the update method within the 'entityManager' and all components would be updated (Figure 8).

[3] Full code for entire simulation can be found at the link provided in Appendix 2.

Figure 5: UML outlining the implementation of the entities as component buckets approach

```
//Creation of the entity manager
ECB_EntityManager ECB_Entities = new();

//Adding ball entities to the manager
ECB_Entities.AddEntity(ECB_EntityFactory.MakeBall(
                            xLocations[i],
                            yLocations[i],
                            dxValues[i],
                             dyValues[i]));

//Updating of entities during game loop
...
ECB_Entities.Update();
...
```

Figure 8: Summary of the simulation process for component bucket based architecture

The third approach used was a full implementation of an ECS. This involves the 3 POD type components, color, transform and movement. This approach also includes a collision, movement and render system which each request the required components from an entity and perform their relevant operations. The UML for this implementation can be seen in Figure 9. During simulation runtime, this approach used a list of entity object similar to that seen in the inheritance approach. Once the list has been created, the relevant ball objects are added and the entity list is passed to the various ECS systems. Here, the game loop calls each systems update method on each frame of the simulation (Figure 10).

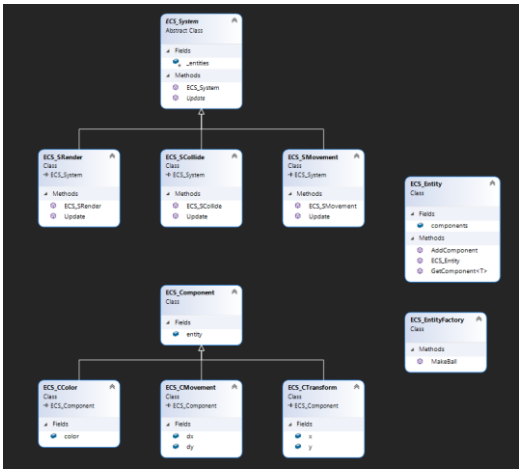

Figure 9: UML outlining the implementation of the full ECS approach

```
//Creation of the list to store entities
List<ECS_Entity> ECS_Entities = new();

//Adding ball entities to the manager
ECS_Entities.Add(ECS_EntityFactory.MakeBall(
                    xLocations[i],
                    yLocations[i],
                    dxValues[i],
                    dyValues[i]));

//Creation of ECS systems
ECS_SCollide eCS_SCollide = new ECS_SCollide(ECS_Entities);
ECS_SMovement eCS_SMovement = new ECS_SMovement(ECS_Entities);
ECS_SRender eCS_SRender = new ECS_SRender(ECS_Entities);

//Updating of entities during game loop
...
eCS_SMovement.Update();
eCS_SCollide.Update();
eCS_SRender.Update();
...
```

Figure 10: Summary of the simulation process for ECS architecture

The final approach used within this research is the entities of complete aggregation architecture described in section 1.E. The UML description for this implementation can be seen in Figure 11 and is similar to that of the ECS without a physical entity class (as previously described within the introduction of this report). As can be seen in Figure 11, the 'componentManager' class uses the singleton design pattern such that it can maintain a single point of access for all entities within the game. This way each system can request the list of all active components that they require and iterate through, processing each one individually. During runtime this approach first calls the clear method on the 'componentManager' class to ensure that each simulation starts with identical starting conditions. From here each system object is created, and ball entities are added to the componentManager's lists. The game loop for this implementation simply calls the update method on each system class similar to that of the ECS approach (Figure 12).

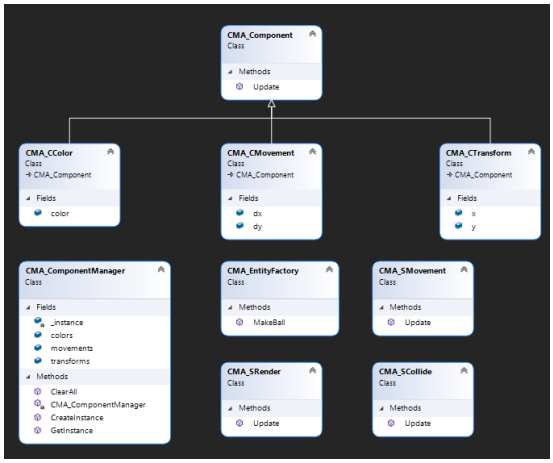

Figure 11: UML outlining the implementation of the entities of complete aggregation approach

```
//Clearing of the component manager objects
CMA_ComponentManager.ClearAll();

//Creation of system objects
CMA_SCollide cMA_SCollide = new();
CMA_SMovement cMA_SMovement = new();
CMA_SRender cMA_SRender = new();

//Adding ball entities to the component manager
//(EntityFactory's make ball method adds created
//entity to the managers component lists)
CMA_EntityFactory.MakeBall(
             xLocations[i],
             yLocations[i],
             dxValues[i],
             dyValues[i]);

//Updating of entities during game loop
...
cMA_SMovement.Update();
cMA_SCollide.Update();
cMA_SRender.Update();
...
```

Figure 12: Summary of the simulation process for pure aggregation architecture

C. *Simulation Metrics*

The metric that will be used to compare the performance of different models is the framerate measured in frames per second. This was chosen as it seemed like the most appropriate measure for a simulation with a visual element such as this one. Using this would seem to expose a level of inconsistency in performance that is contingent on the speed of the computer running. However, while the framerate itself would vary across machines, the relative framerate of each model should stay consistent; that is model 'A' should always have a ±x% difference in perfromance from model 'B'.

D. *Simulation method and data gathering*

To ensure a fair comparison between each approaches, the entities being simulated between each approach needed to contain the exact same data. The way this was achieved was first to generate four arrays of 'n' random numbers where 'n' is the number of balls in the given test. Each of the four arrays represent the initial x, y, dx and dy values used when creating the balls of each simulation. These same arrays would be used when initializing the data in each of the architectures so that the simulations would be modeling identical entities. Each individual test first consisted of the creation of new manager and system objects required for each model including both a stopwatch object and a frame count variable. Secondly, as previously described, a number of random numbers are generated and used as initial values for each simulation. These random numbers are then used to create the relevant entities for each model.

Once everything has been initialized, each game-loop can be run sequentially. This first consists of first resetting the frame count variable to a count of zero so that framerate is measured independent of other tests. Secondly, the start method on a stopwatch object is called that will stop the game loop after a provided interval has elapsed. A loop is then initialized that will perform the current simulation for a given time with the frame count variable incrementing at each integration. Once the stopwatch has reached its given time the loop is broken and the counted frames are recorded in a csv file before repeating with the next architecture (Summarized in Figure 13).

```
frames = 0;
start stopwatch

do
{
   Update model;
   //See Figures 6, 8, 10 and 12 for details on
   //game loop methods of each architecture

   frames++;

} while ( stopwatch count < interval );

AddToCSV( ballcount, time, modelName, frames );

Repeat with each architecture before changing
entity or interval parameters...
```

Figure 13: Summary of the method used to gather data for each model

This process was then run at entity counts of 1, 2, 4, 8, 16, 32, 64, 128, 256 and 512 each at an interval of 5, 10, 20 and 40 seconds. Each parameter set was run 5 times to minimize random variation for a total of 200 simulations per architecture. Due to an inconsistency in results that will be discussed later in the report this process was adjusted such that the C# garbage collectors GC.Collect() method was called between iterations to further ensure consistency between tests. In addition, all other applications were closed and any power-saving or time-out features of the computer were disabled. This adjusted method was run with entity counts in 100 increments of 50 balls starting at 50 and ending at 5000 entities. Each ball increment was run at intervals of 5, 10 and 15 seconds with each parameter set being repeated 5 times to minimize random variations for a total of 1500 simulations per architecture.

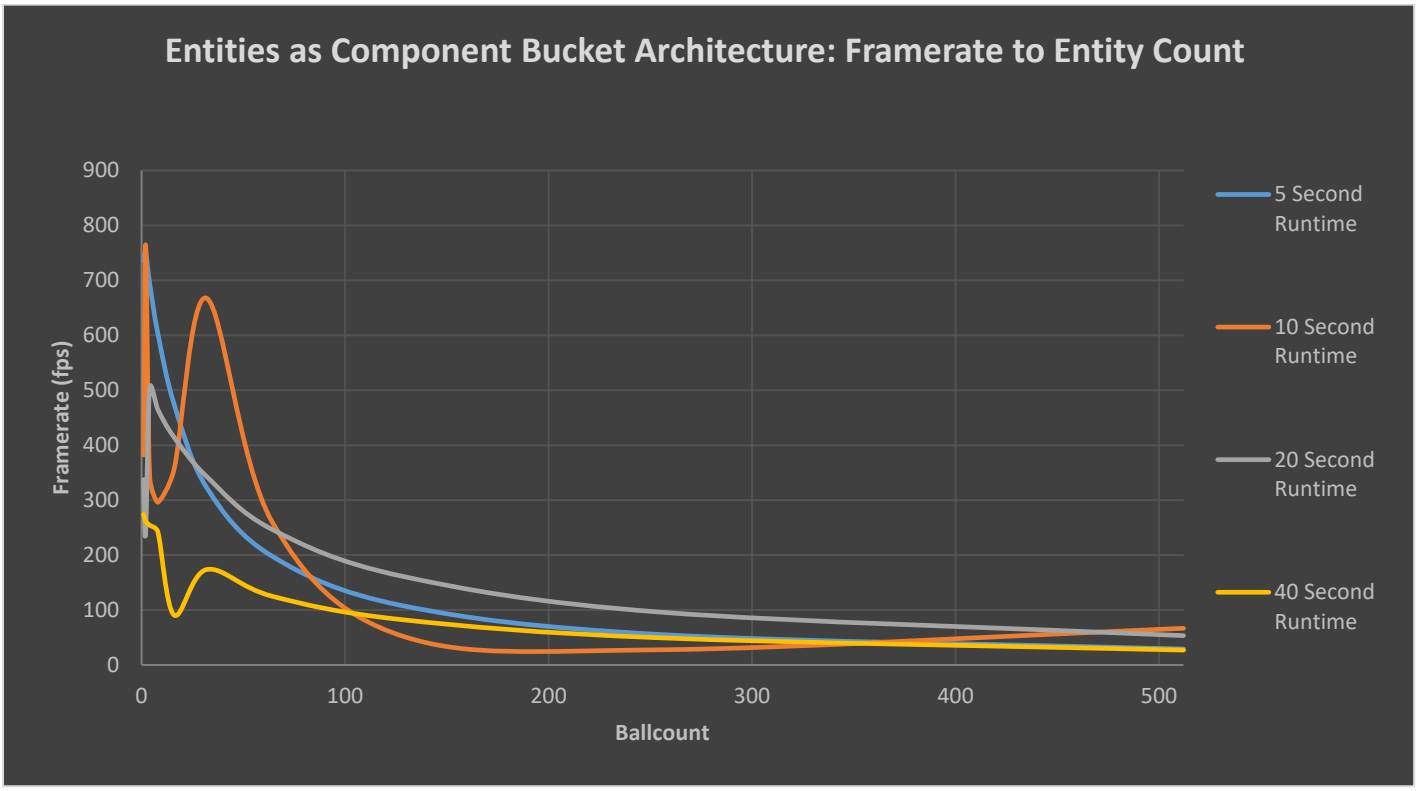

Figure 14: Entity as a component bucket architecture results with different simulation runtimes derived from first round of testing. Note the seemingly random variation in framerate during the tests with low entity counts. In addition, note the difference in framerate despite the architecture between tests being static.

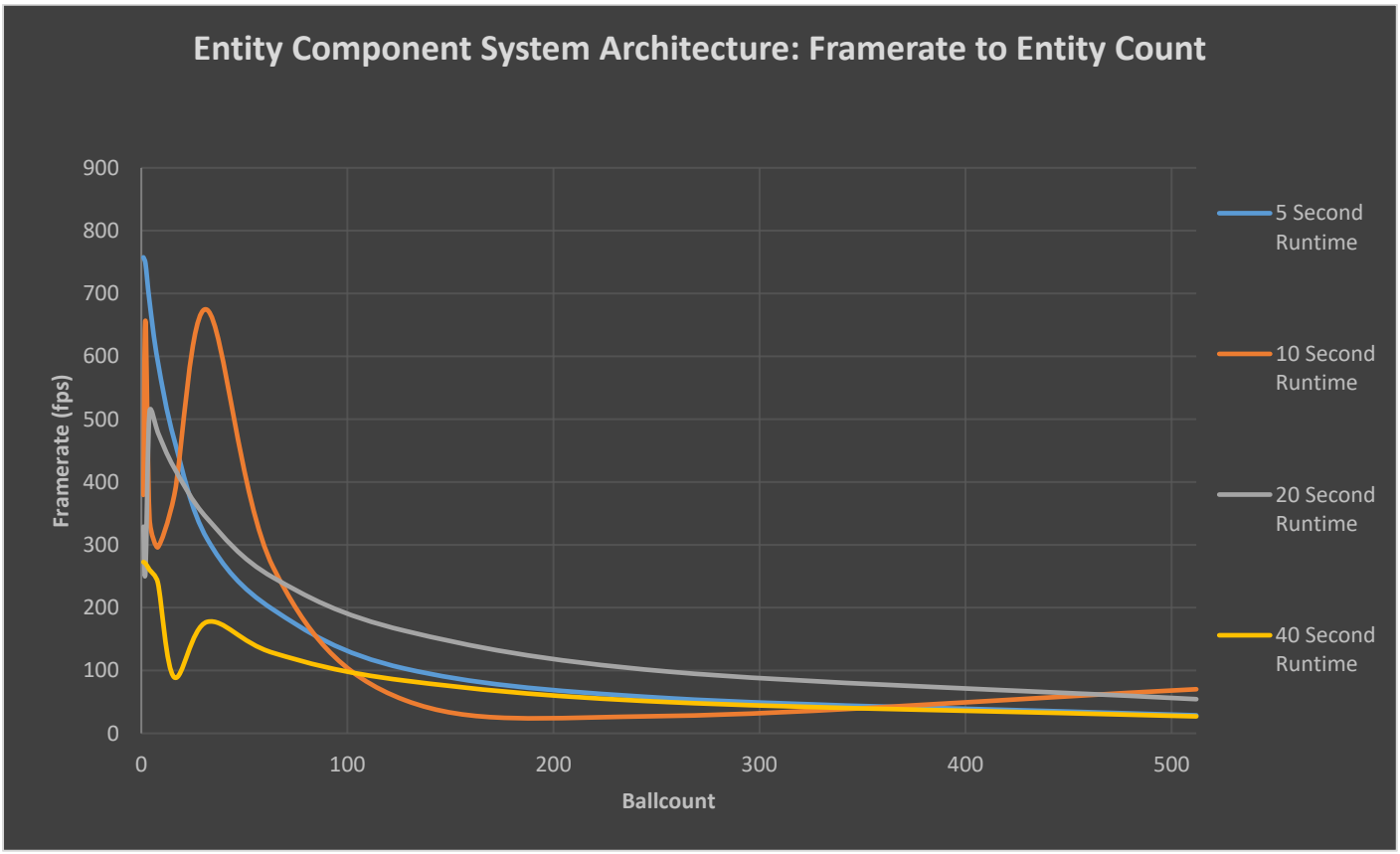

Figure 15: Entity component system architecture results with different simulation runtimes derived from first round of testing. Note how similar to Figure 14 this also suffers from inconsistent results with a different relative framerate between runtimes.

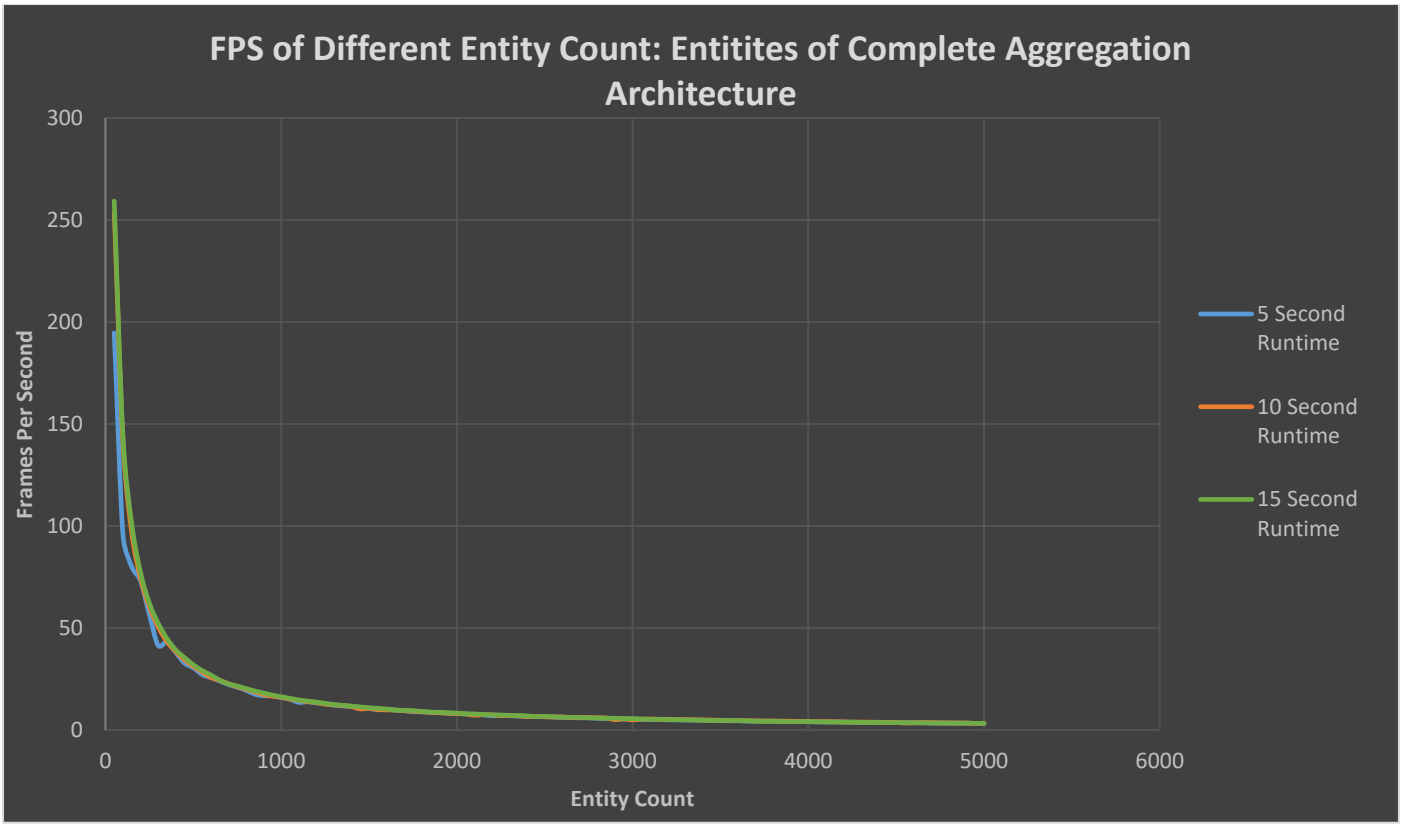

Figure 15: Entity of complete aggregation architecture results with different simulation runtimes derived from second round of testing.

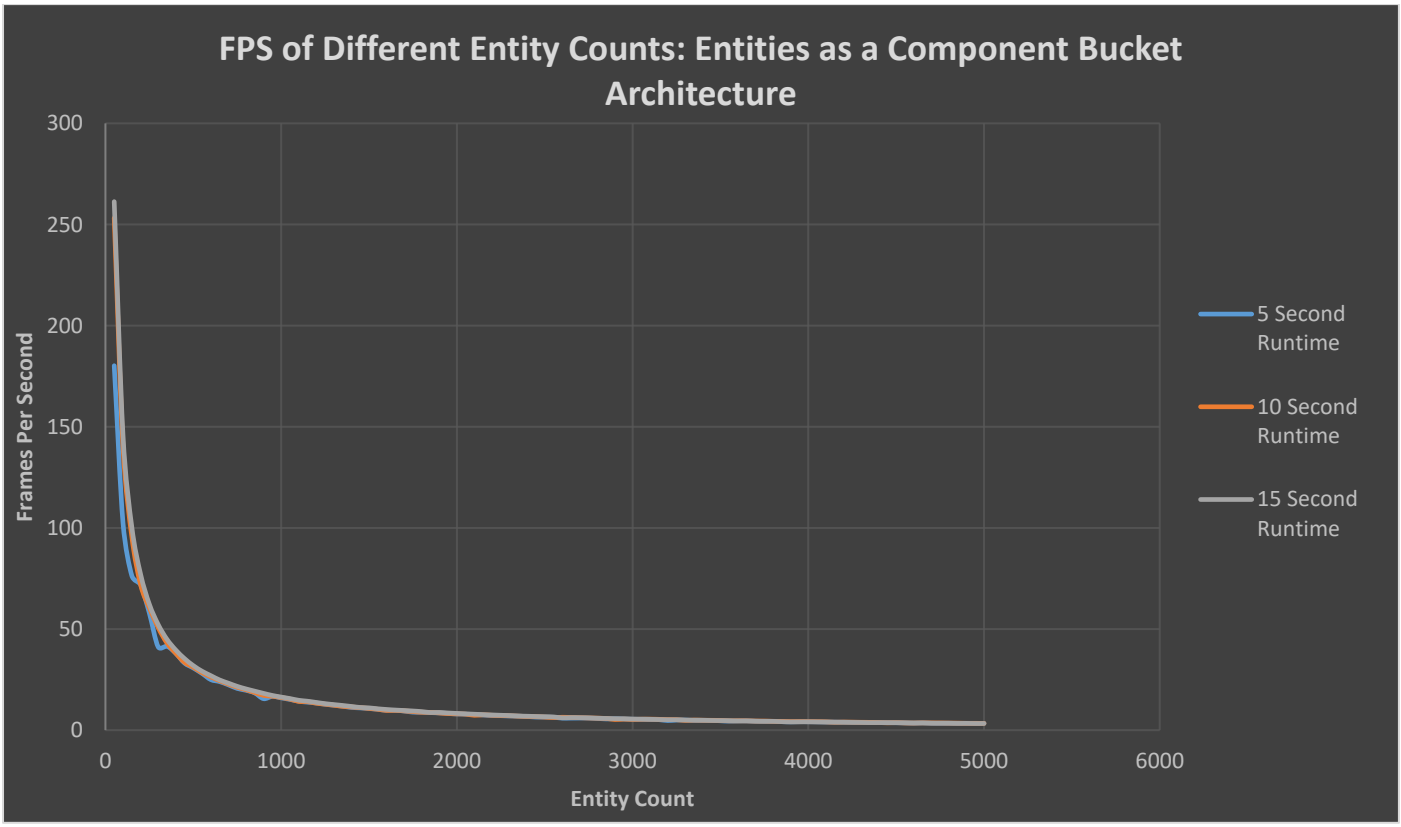

Figure 16: Entity as a component bucket architecture results with different simulation runtimes derived from second round of testing.

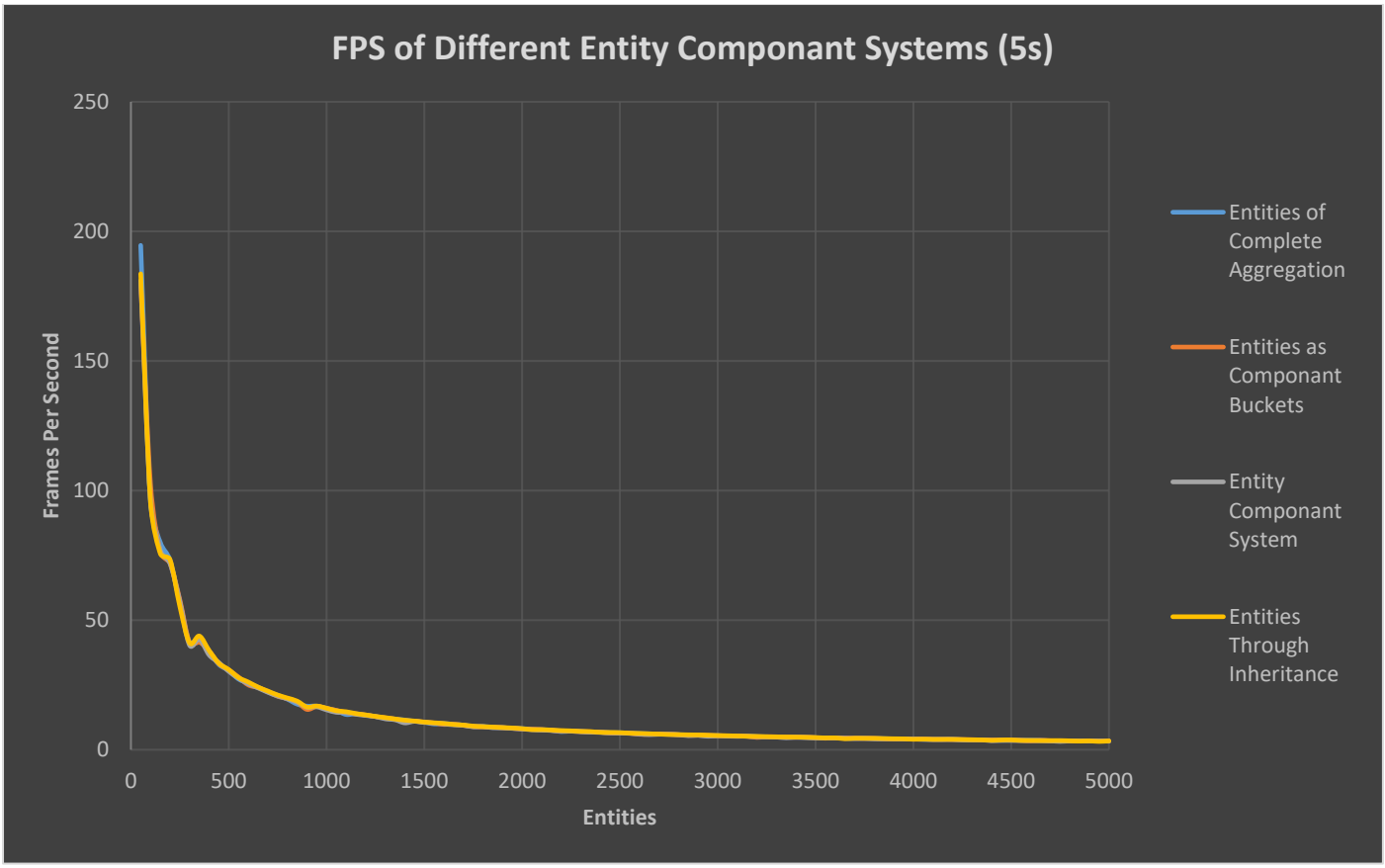

Figure 17: Framerate of different simulation architectures for a duration of 5 seconds. Note that while there is four datasets pictured here they are unable to be clearly distinguished due to their similarity.

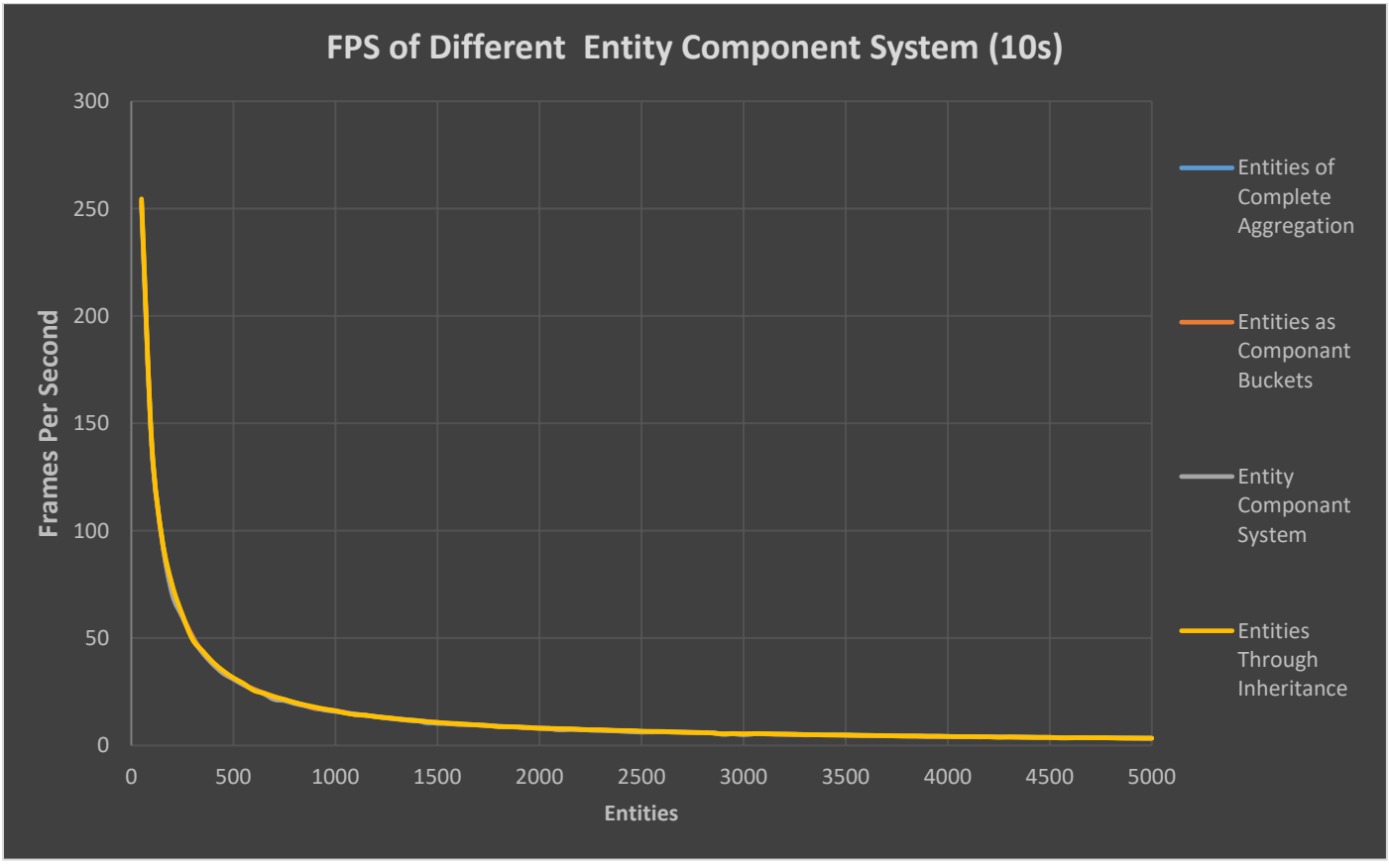

Figure 18: Framerate of different simulation architectures for a duration of 10 seconds. Note that while there is four datasets pictured here they are unable to be clearly distinguished due to their similarity.

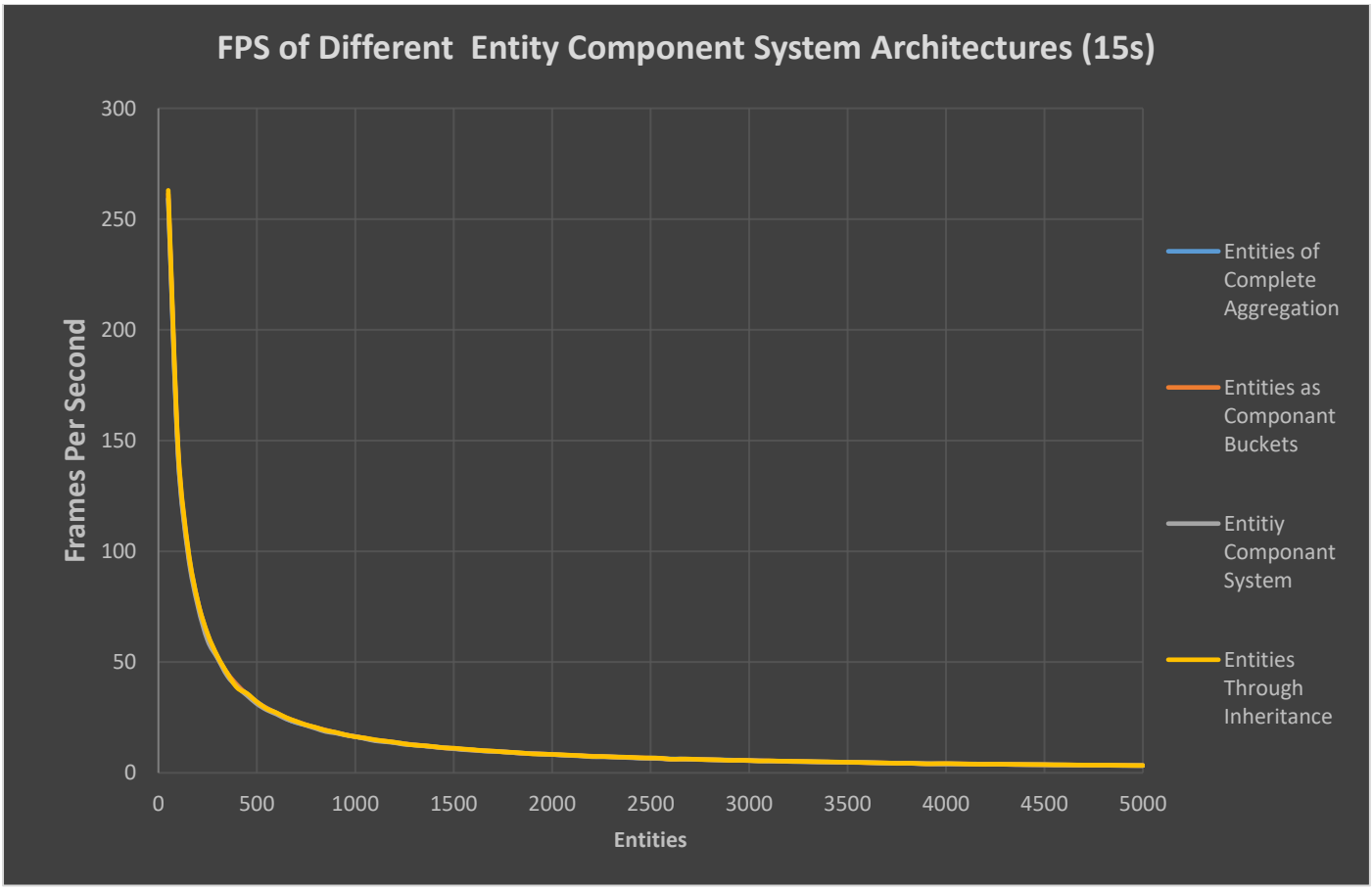

Figure 19: Framerate of different simulation architectures for a duration of 15 seconds. Note that while there is four datasets pictured here they are unable to be clearly distinguished due to their similarity.

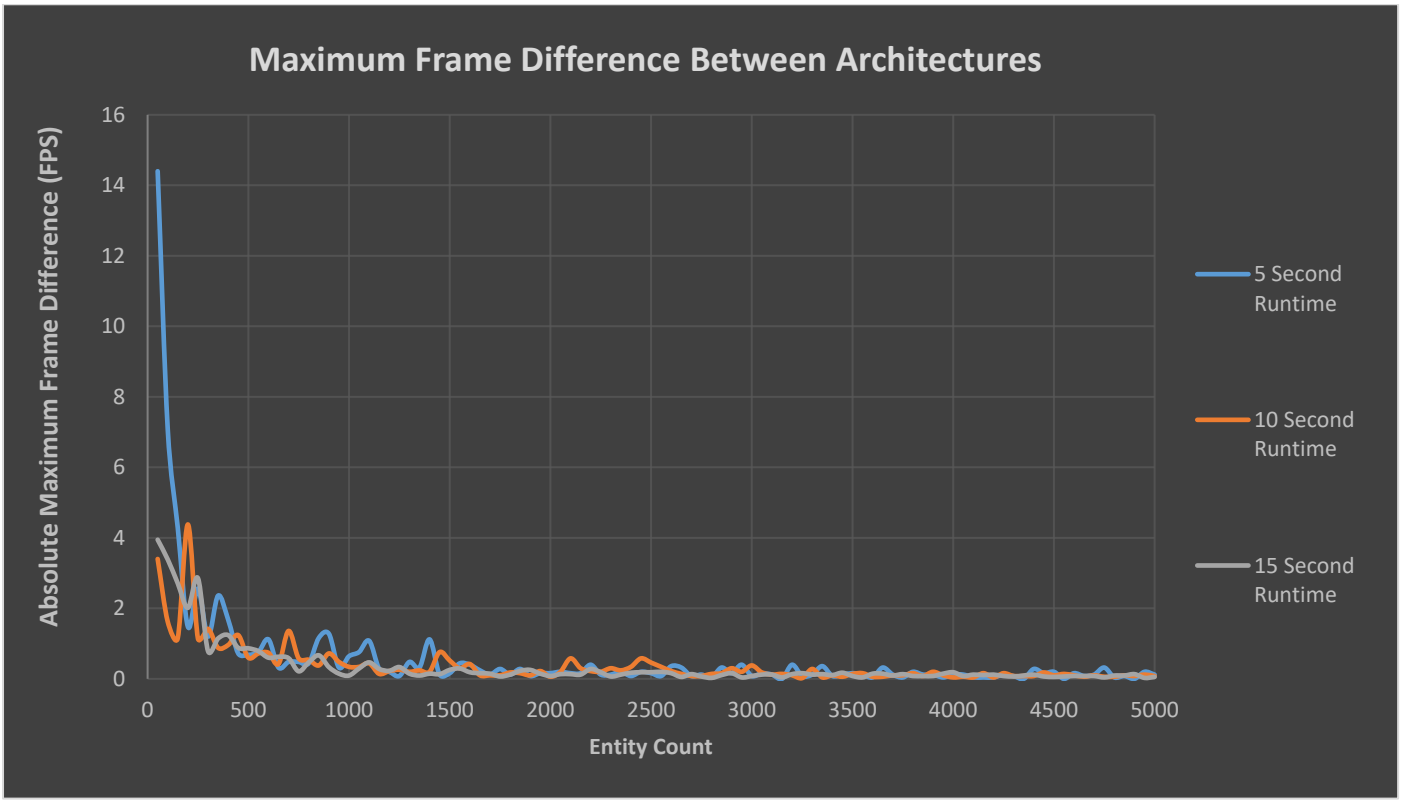

Figure 19: Maximum absolute difference in framerate between different architectures. It should be noted that the average maximum framerate difference was recorded to be 0.43 FPS.

IV. Discussion

After analyzing the results found from the first round of testing it was clear to see that there was a flaw in the methodology of the tests. As can be seen in Figure 14 and Figure 15 there is a large variation in frame rate performance despite the architecture of the test remaining constant. The volatility in framerate despite entity counts strictly increasing hints at a flaw within the testing framework. In addition, this discrepancy was likely due to a predictable source, interpreted from the consistency in volatility between Figure 14 and Figure 15 (Both tests see in increase in frame rate at entity counts between 2 and 32 for a 10 second runtime). In order to address this concern of irregularity within the test and ensure a rigorous testing environment the following changes were made. First, the C# garbage collector was called between each test to ensure that objects from previous tests would not interfere with future measurements. Secondly, all background processes, power saving features and timeouts within the computer running the tests were halted to ensure a more consistent distribution of computer resources between tests. Lastly an adjustment was made to the parameters of the tests themselves. Entity counts would be both more gradually increased as well as reaching a higher maximum entity count (Described in detail within the method section of this report). This way trends within the performance data could be more easily observed and in the case of additional artifacts within measurements, it would be more trivial to isolate the origin. As seen in Figure 15 and Figure 16 these changes minimized volatility within the data fundamentally. Figure 15 and Figure 16 show that the framerate of the simulation did not change significantly with an increase in simulation duration. This serves as a benchmark, proving that the adjustments made in the testing process was able to minimize external resource usage and random performance variances caused by memory management processes.

As can be observed in Figures 17, 18 and 19 the data shows no statistically significant difference in framerate performance between architectures. This can be further observed in Figure 19 where the performance difference between the best performing architecture and the worst performing architecture was recorded. Here we can see that there was a significant difference during low entity and low duration simulations. However, this result can likely be attributed to the same framerate volatility seen in short duration testing also observed in Figure 17. Despite this initial performance difference, the recorded average maximum framerate difference was found to be 0.43 FPS. This result, as well as the clear trend of identical recorded data between architectures (observed in Figures 17, 18 and 19) are evidence of a significant performance bottleneck within the test itself. There appears to be an unconsidered source within the test structure that is constricting the framerate performance of each test to a level below what each architecture is capable of individually. That is, each architecture is being capped at the same performance such that no distinction can be made between them. While it is impossible to make a definite conclusion as to the origin of interference with the available data. It is hypothesized that it is due to the Splashkit libraries rendering capabilities. The reasoning behind this is that Splashkit's rendering functions are one of very few similarities between testing architectures. Consequently, this process having a large resource requirement is the most obvious source of a performance bottleneck that would affect each implementation uniformly. Each architecture used the Splashkit libraries 'FillCircle' function to render their relevant entities to the test window. Therefore, it is likely that rather than examining the performance of various entity component system architecture's, we have rather observed the 'FillCircle' functions execution time per entity. This result means that no conclusions can be made on the data and that further testing is required to investigate confounding variables.

V. Conclusion

This research report set out to compare the performance efficiency of different entity component system architectures. To do this a simple visual ball collision model was created in a number of different ways. These simulations were then run using various entity counts and durations in order to observe the relationship between runtime performance and entity counts between different systems. Initial testing was found to contain a number of unexplained variations in data and so a revised test structure was designed. This secondary round of testing was able to ensure fairness between tests, removing the volatility seen previously. However, this second round of testing revealed a further issue with the testing methodology. A problem that is postulated from the observed similarity in results between simulation methods. This hints at the existence of a performance bottleneck that is standardizing and limiting each architecture such that they appear identical.

Further testing is required in removing any possible performance limiting elements of the simulation. In a continuation of this research, the removal of the dependency of the Splashkit library would be the most logical step. Removing any visual components to the simulation would further isolate the explanation of performance data to the architecture of the simulations themselves. This is due to such a simulation simply consisting of the alteration of data within entities. Consequently, the performance from such a simulation is likely to greatly better reflect the performance of the various entity systems themselves rather than being affected by external processes. In addition, further testing may benefit from additional metrics being recorded such as CPU usage (processes threads and handles) as well as memory usage. This way more objective and statistical measurements of the performance can be observed. Overall, while this research did suffer from confounding variables it serves as a groundwork for future, more rigorous and conclusive investigation.

Appendix I. Raw test data

The raw first round testing data can be found at:

https://docs.google.com/spreadsheets/d/1TzBdB1zuVX5-I6NeH-ksa_6qdP3FSiIj/edit?usp=sharing&ouid=113435264555937625106&rtpof=true&sd=true

The raw second round testing data can be found at:

https://docs.google.com/spreadsheets/d/1FGPR_v93Pn8R2OAlrvdGuWx-Jx3qMidH/edit?usp=sharing&ouid=113435264555937625106&rtpof=true&sd=true

Appendix II: Code used within research

Full code used within this research can be found at the below link. Note what is referred to as "Cache Miss Adjustment" within the code solution is referred to as Entities of Complete Aggregation within this report.

https://github.com/AlexKyriacou/Research-Project-Code

References

[1] M. West, "Evolve Your Hierachy," 5 Jan 2007. [Online]. Available: https://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/. [Accessed 20 10 2021].

[2] Source Making, "The Blob," 2021. [Online]. Available: https://sourcemaking.com/antipatterns/the-blob. [Accessed 20 10 2021].

[3] H. Weerasinghe, "Deadly Diamond of Death," 16 9 2019. [Online]. Available: https://medium.com/@harshamw/deadly-diamond-of-death-e8bb4355c343. [Accessed 20 10 2021].

[4] B. Nystrom, "Bob Nystrom - Is There More to Game Architecture than ECS?," in *Roguelike Celebration*, Virtual, 2018.

[5] M. Hall, "Creating an Entity-Component-System in C#," 25 9 2020. [Online]. Available: https://matthall.codes/blog/ecs/#fn1. [Accessed 20 10 2021].