Proceedings of the Third International Conference on Smart Electronics and Communication (ICOSEC 2022)
IEEE Xplore Part Number: CFP22V90-ART; ISBN: 978-1-6654-9764-0

2022 3rd International Conference on Smart Electronics and Communication (ICOSEC) | 978-1-6654-9764-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICOSEC54921.2022.9952025

Performance Analysis and Optimization Techniques in Unity 3D

Narinder Pal Singh
Immersive and Interactive Technology Lab,
Chitkara University Institute of Engineering and Technology
Chitkara University
Punjab, India
narinder.singh@chitkara.edu.in

Bhanu Sharma
Immersive and Interactive Technology Lab,
Chitkara University Institute of Engineering and Technology
Chitkara University
Punjab, India
bhanu.sharma@chitkara.edu.in

Avinash Sharma
Department of computer Science and Engineering,MM, Engeneering College,Maharishi Markandeshwar(Deemed to be University), Mullana-Ambala, Haryana,India 133207
asharma@mmumullana.org

***Abstract*—Unity3d is a powerful toolset that can be used to build 3D interactive applications and technology like games, simulations, augmented reality, immersive reality, etc. This game engine can execute the graphics, simulate physics, play audio, provide interactions, facilitate photorealistic rendering, and is excellent for cross-platform development. Although everything that users perceive is super-optimized, still an application suffers from poor performance, because of inefficient or poorly organized code, disorganized memory management, unnecessary use of overly complex physics simulations, etc. Poor performance of the application can break the user experience. Another side of an application properly optimized provides a smooth and responsive experience. To provide a good experience to the users, advanced knowledge of engines is required. Even if each application required a unique approach for optimization, Still there are key techniques we can adopt in most situations. We are aiming at filling this gap, by evaluating the best techniques while performing tests and analyses with Unity3d tools. We started by explaining game engine architecture. Then we performed tests while comparing different techniques and came up with observations and findings. As Unity3d, aggressively contribute to developing innovative applications related with 3D computer graphics to promote and support industry and Manufacturing. This work is contribution to the Unity3D community to develop good performance applications to provide a good user experience.**

***Keywords*—Computer Graphics, Unity3d, Optimization, Garbage Collection, Innovation, Research, Technology, technological progress**

I. INTRODUCTION

Doom game engine was released in 1993, and tech has evolved over 30 years since then. Initially, these game engines were possible to generate pixel art based 2d games, but now they are capable of realistic 3d visuals, enormous multiplayer games, realistic physics, and the development of immersive reality applications for cross platforms. Popular game engines are Unity3d [1], Unreal Engine [2], CryENGINE [3], Shiva [4], Game maker [5]. A detailed performance analysis of these game engines is performed [6]. The underpinning of developing the game is the game engine. Unity3d is the most popular game engine in the world. It was announced at Apples Worldwide Developers Conference 2005 [7]. Unity3D is at the heart of game development. Offers a primary foundation and common functionalities for game development. Unity3D allows the creation of games for a variety of platforms, such as the web and mobile devices. Unity3d engine has been used in Augmented Reality based teaching environments [8], [9]. The primary parts of software for modern games are created by game engines. By providing accessible interfaces for the devices and operating systems that the game plays on, a game engine makes the job of programmers easier. A game engine's main objective is to make the most of the home machines' resources to give users the most realistic gaming experience possible. Although performance optimization for improving the coverage standards using hybrid methods has been done [10], [11]. The scientific community needs publications that analyses and assess the state-of-the-art, or the latest game engines, despite how important it is to revisit gaming engines. Considering the importance of the gaming industries, the volume of literature on game engines is surprisingly thin. The architecture and performance of contemporary gaming engines have never been fully examined, despite two position papers calling for research on this topic [12], [13]. In this paper according to what we know, related work focuses on using game engines in particular fields, like research [14], serious games [15], and serious VR applications [16], [17], [18], [19].

II. BACKROUND

When an application is executed on a device, the device's central processing unit (CPU) executes instructions. In the case of a 3d computer graphics application, a frame required the execution of millions of these CPU commands. The CPU must complete its instructions within a certain period to sustain a steady frame rate. Application may slow down,

jitter, or freeze if the CPU cannot complete all its tasks in a timely manner. There may be many reasons for which the CPU has to do much work, like calculating physics collisions, updating animations, executing codes, and managing memory.

In this section, we are defining a few concepts that are commonly used in Unity3d development and are essential for understanding performance analysis and optimization. Below are some important points are listed when beginner developer starts his journey towards development

- **Caching:** Storing and reusing returned results or data from the function is called Caching.
- **Garbage Collection (GC):** GC is a process through which Unity3d performs automatic memory management. The garbage collector seeks to reclaim memory that was allocated by the application but is no longer referenced.
- **Object Pooling (OP):** OP is a technique where objects are created once, temporarily deactivated, and reactivated when required.
- **Occlusion culling (OC):** OC is the mechanism by which Unity avoids rendering computations for GameObjects that are fully hidden from view by other GameObjects.
- **Draw Call:** Draw Call is a call to graphics API to render objects. And each draw call contains all information about the objects that are going to be drawn.
- **Batching:** Combining meshes into the same draw call is called Batching.
- **Static batching (SB):** SB is a method that groups mesh that stay static in the runtime to reduce the draw calls.
- **Dynamic batching (DB):** DB is a method that group meshes that are static or moving in the runtime to reduce the draw calls.
- **SetPass:** SetPass is a command sent to the GPU to change several variables or settings required to render a mesh.
- **Fill Rate:** Fill rate is the number of pixels the GPU can draw on the screen each second.
- **Vertex Processing (VP):** VP is a task that the GPU must complete by drawing each vertex in a mesh.
- **Rendering Time:** Time taken by Unity3d to process or execute a single frame and measured in milliseconds.

III. UNITY3D ENGINE

Unity3d is component-based game engine, where without writing complex code, developer can quickly develop and integrate a feature while adding components. Component models provide scalable programming architecture. Every entity in the scene is called GameObject. According to the need multiple components can be added to the GameObject. Scripts, Physics, UI elements are examples of components attached with GameObjects. Performance analysis and optimization of a 3d application can be done mainly three ways, These are Unity Scripting API, Memory management and Graphics Rendering.

A. UNITY SCRIPTING API

Scripts are the part of the source code that determines the content and structure of compiled code. Well-structured source code produces a well-structured compiled code. But some simple-looking operations in source code can be very complex. Knowledge of native code helps us to produce more efficient compiled code. As some native codes take more time than others. For example, instructions responsible to calculate square roots are more time-consuming than the multiplication of two numbers. Although the time difference between fast and slow instruction may be very low, the resultant of many instructions can be noticeable.

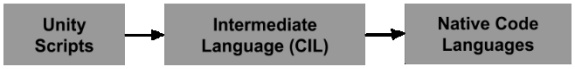

Figure 1: Unity Scripts compiling process

CIL compiled code also called managed runtime, and native code called managed runtime. To eliminate device crashing, managed runtime and do safety checks. CPUs continually switch between running main engine code and managed code and pass data from managed code to back engine code. It needs to do some conversion to make it available in the format required by managed runtime code. Inefficient or poorly organized code can be a reason for poor performance Like a function calls several times however a single call is enough. In this paper, we will discuss poorly organized alternative solutions. Code is well organized; however, it makes excessively costly calls to other routines, which results in unnecessary calls among managed and engine code. We will discuss some expensive Unity API calls and alternatives that are more efficient. Code is efficient, but it's called when it is not required. We will discuss some techniques to implement a code that will run only when required. Code is required to run many instances, and we will discuss making code less demanding. If a performance issue is with our code, then we must plan to solve it. Like if we have demanding code, it's easy to start to work on it but on another hand, we may also have a code that is applied to many game objects and do minor improvements that can give a good performance increase. So, in this paper, we are not proving any steps to have good optimization, but findings and suggestions for improving our code. Efficient and well-Structured code can improve the game performance.

B. MEMORY MANAGEMENT

A computer application required memory to store and access data. When a particular object or variable is not required, memory needs to be released so that it can be used

by another date. If a memory has been used to store data, but data is no longer used in the application then it's called Garbage. And the automatic memory management process that frees unused memory is called Garbage collection. If Garbage collection performs frequently, it can cause poor performance and low frame rates, because Unity performs automatic memory management to run user code. Unity handles memory using Managed memory, C# unmanaged memory, and Native memory layers. Managed memory layer uses a heap and garbage collection to perform automatic memory management, C# unmanaged memory layer can be used to access the native memory layer with Unity Collections namespace and package and doesn't have access to GC so-called unmanaged. Native memory is used to run the engine itself using C++ memory and, in most cases, users don't have access to it. Scripting Virtual machines (Mono and IL2CPPs) implement Managed memory can be further divided into three types, Managed heap, Scripting stack, and Native VM memory. Where Native VM memory holds the memory that is related to the Unity scripting layer, and most of the time we don't need to perform modifications on it. The managed heap is the one section where the VM does automatic memory management with GC and is used for storing long-term storage and large amounts of data. And memory allocated in this section called GC Allocation, the Unity3d profiler can plot and record as a GC.Alloc. A scripting stack is used to store short-term storage of small parts of data. Every time when a variable is declared to store an object or some value, Unity stores the data in either stack or heap. Allocation and deallocation are very fast in the Scripting stack, because it only deals with small pieces of data for a short time, and data is stored strictly in order. However, allocation in a managed heap is very complex, because it stores data for long-term users, and in this way, it can store different sizes and types of data. When a heap variable is declared, Unity needs to check the available memory in the managed heap, if there is no free memory available then a garbage collector is executed to free up unused memory, if still memory is not available then Unity increases the total allocation in the managed heap. This process is very slow as compared to the scripting stack, as it is required to run a garbage collector and may require expansion in memory. Further garbage collection itself is an expensive operation because it is required to search for out-of-scope objects, and need to mark a deletion flag, and after that delete them. In Unity, every variable is allocated on the heap except value-typed local variables.

C. GRAPHICS RENDERING

Rendering a single frame may involve many steps, and there may be many factors that can affect the performance of graphics rendering. It depends on the operating system and hardware platform where the application is running. Like code and memory optimizations rendering optimization also required careful investigation, and analysis to find out the cause behind it.

The rendering process can be divided into basic three parts, in the first step, a central processing unit (CPU) decides what and how it's going to render, in the second step the CPU sends commands to the graphics processing unit (GPU), and in the last step GPU renders objects according to given CPU commands. In more detail, the first CPU makes a list of all objects that are going to be rendered and discards culled objects according to view frustum. CPUs collect rendering information from each object like linked texture, material, etc, and sort the data into instructions, these instructions are called draw calls. Meshes or objects sharing similar data can be combined into a single draw call, and this process is called batching. From each draw call, the CPU creates a data packet called batch to send to the GPU. For each batch the CPU may also need to send extra commands to the GPU called SetPass calls, these commands are used to change the render settings of the mesh. SetPass call is only sent if there are requirements of change in the render state of the next mesh being rendered. Sometimes more than one SetPass call is required for a single mesh if it's divided into sub-parts and each mesh has different materials applied. While the CPU sends commands to GPU, in a parallel manner GPU renders mesh if the draw call command is received and changes the render state of mesh if the SetPass call is received. Further GPU rendering tasks are divided into stages, defined in shaders in sections. Shaders may have some sections, like a vertex shader that is responsible to process the vertices, a fragment shader responsible to draw the pixels.

As graphics rendering involves both CPU and GPU, each hardware should finish the assigned task in a lower time in case it takes a long time then it impacts the performance because it can't finish rendering on time. Low performance because of graphics rendering can be suffering from two main problems, one of them is inefficient pipeline called bottlenecks and the second problem is caused by pushing too much data to the pipeline that it can't handle. The application can be either CPU bound, or GPU bound, to render a single frame if the CPU or GPU takes a long time to complete rendering tasks it's called CPU bound or GPU bound respectively. Similarly, to the performance optimization in other parts of Unity we can also profile to analyze the reason behind the low performance. From the Unity profile window, we can get the time taken by CPU and GPU, if the CPU takes a longer time than GPU, then the application is CPU bound and opposite to GPU bound. However, GPU profile is not supported on some target platforms, in that case, we can also estimate by observing CPU waiting time for GPU in total CPU usages. For that, we can select the CPU profile in Hierarchy mode and can look for Gfx.WaitForPresent function, if it's taking maximum CPU time, then it's GPU bound. We can identify CPU-bound applications from CPU Usage modules in the Profile window if rendering takes maximum time from the rendering function from the Hierarchy section. In Unity3d we can also use Frame Debugger, to get detailed information about draw calls and other rendering commands sent to the

GPU. The most common CPU-bound reason is sending instructions to the GPU or SetPass call. So, lowering the SetPass calls can improve performance.

TABLE 1 PERFORMANCE OF GRAPHICS RENDER API ANALYSIS IN UNITY 3D

	Parameters	**Slow**	**Fast**
1.	Update Function Instances	Object instances with individual Update()	Object instances shared Update()
	Test	1000 instances populated with individual Update function	1000 instances populated with a custom function, custom function updated by single Update() function
	Time spent on single frame	~ 0.07 ms – 0.14 ms (1000 calls)	~ 0.02 ms – 0.07 ms (1 call)
2.	Loop in Update() Function	Update() for i ← 0 to n: if condition: main code	Update() if condition: for i ← 0 to n: main code
	Test	Running for loop 1000 times in Update() function without any condition	Stopping for loop iteration in Update() with condition
	Time	~ 0.02 ms – 0.1 ms	~ 0.00 ms
3	Use Caching Cache Component	Update() comp ← using GetComponent: comp.changeAttribute	comp Start() comp← using GetComponent: Update() comp.changeAttribute
	Test	Getting Renderer component in for loop that iterates 1000 times on each frame.	Storing component reference in a variable on Starting and reusing in update function.
	Time	~0.32 ms – 0.55 ms	~ 0.25 ms – 0.31 ms
4.	SendMessage()	1000 SendMessage calls on each frame	1000 SendMessage calls on each frame
	Test	A function is called 1000 times from one class to another attached to same object using SendMessage()	A function is called 1000 times from one class to another attached to same object using saved reference
	Time spend on Update()	~ 0.7 ms – 4.92 ms (Found GC spikes)	~ 0.0 ms – 0.02 ms (No GC spikes found)
5.	Find()	1000 Find() calls and setting new position on each frame	Storing object reference in variable and setting position 1000 times on each frame.
	Test	In Update() 1000 Find() calls made to find an object with a particular object, Once found, changing position of that object.	In the Start() function we have one Find() call, found object stored in reference. In Update() 1000 changing position of that stored object.
	Time spend on Update()	~ 0.17 ms – 0.39 ms	~ 0.0 ms – 0.02 ms, No GC spikes found
6.	Position vs localposition	Transform.position	Transform.localposition
	Test	Accessing transform position 1000 times with "position" attribute in Update() function.	Accessing transform local position 1000 times with "localposition" attribute in Update() function.
	Time & Spikes	~ 0.03 ms – 0.11 ms (Frequent spikes)	~ 0.03 ms – 0.08 ms (Less spikes)
7.	Distance vs sqrMagnitude	Vector3.Distance	Vector3.sqrMagnitude
	Test	Calculating distance using Vector3.Distance 1000 times in single frame	Calculating distance using Vector3.sqrMagnitude 1000 times in single frame

	Time & Spikes	~ 0.03 ms – 0.1 ms (Frequent spikes)	~ 0.03 ms – 0.08 ms (Less spikes)

8.	Camera.main vs reference	Camera.main	Camera reference
	Test	Using Camera.main	Using Camera reference
	Time & Spikes	~ 0.09 ms – 0.15 ms & Frequent spikes	~ 0.03 ms – 0.08 ms & Frequent spikes

TABLE 2 PERFORMANCE OF GRAPHICS RENDERING ANALYSIS IN UNITY 3D

	Parameters	**Slow**	**Fast**
1	Texture Atlas	Separate material and texture on objects	Texture atlas and single material
	Test	10000 cubes rendered with 5 material and textures	10000 cubes rendered with 1 material and 1 Texture atlas
	SetPass Calls	32	4
	Rendering Time	Spikes between ~ 25ms - ~30 ms	Spikes between ~20.56 ms - ~ 24.74 ms
2	Static Batching	Static batching off	Static batching on
	Test	10000 cubes rendered having single material without enabling static batching	10000 cubes rendered having single material with enabling static batching
	Batches	20025	3304
	Rendering Time	Spikes between ~20.56 ms - ~ 24.74 ms	Spikes between ~18.56 ms - ~ 16.74 ms
3	UI Atlas	UI Sprites form individual image files	UI Sprites from single UI atlas map
	Test	5 Individual sprites	5 Sprites from UI atlas map

IV. PERFORMANCE OPTIMIZATION

A. API OPTIMIZATION TECHNIQUES

API	**Optimization Techniques**
Update() / FixedUpdate()	Lower the use of Update() or FixedUpdate() methods, Shared methods can the lower CPU overhead
Loop in Update()	Calling loops per frame lowers the performance, Conditions can limit loop frequent calls
Use Caching	Caching and reusing stored component and game objects can be more efficient
SendMessage()	SendMessage is a extremely expensive to use, instead a class reference can be used
Find()	Find() is a extremely expensive to use, instead a class reference can be used
Transform.localposition	Transform localposition is less expensive then Transform position, because position is recalculated each time, consider using local position if possible
Vector3.sqrMagnitude	Square root calculations in Vector.Distance can be avoided using Vector3.sqrMagnitude

Camera.main	Camera.main work like FindWithTag(), Instead we can store the reference if possible

B. MEMORY OPTIMIZATION TECHNIQUES

Component	**Optimization Technique**
GC trigger frequency	Minimize the GC single run time and the frequency with which GC run
GC trigger time	Can trigger GC at the time when there is no critical performance
Caching	Use caching for components and objects to lower the heap allocation
Per frame allocation	Avoid the per frame allocation in Update or LateUpdate Mono Behavior, instead we can use time to limit per frame calls
New collection	Instead of creating new repeatedly Clear collection using Clear()
Object Pool	Object pooling should be used, Instead of creating and destroying objects at runtime.
StringBuilder	Use StringBuilder class to build string, it build string without allocation
String Concatenate	Instead of frequently concatenate strings, use separate two or components if possible
Debug.Log()	Remove Debug.Log() calls, after completing debugging
GameObject.CompareTag()	Use GameOBject.CompareTag() instead of comparing GameObject.tag with string

C. GRAPHICS RENDERING OPTIMIZATION TECHNIQUES

Rendering Processes	**Optimization Technique**	**Optimization Examples**
Batching	Share a common render state, reduce the batches	Share the material among different objects
		Try to use identical material settings for
		Use static batching for non moving objects, but be careful with high memory usage
		Use dynamic batching for moving objects, but be careful on CPU usage that can cause it to cost more.
		Make UI atlas instead of keeping each element separate
		Use Texture atlas if possible to share a batch
		Disable renderers and camera those are not in use
SetPass call & Batching	Lower rendered objects	Reduce number of visible objects
		Decrease camera draw distance to minimize objects
		Use occlusion culling to disable objects that are hidden by other objects
		Only animated object should use SkinnedMeshRenderer component
SetPass call	Decrease render calls for	Avoid dynamic lights

	single object	Bake lighting and shadows
		Tweak quality settings of real time shadow
		Avoid or optimize Reflection Probes
Fill rate	Lower the number of pixels that GPU can render on each second	Decrease the display resolution
		Avoid complex fragment shaders, instead use simple one
		Minimize the overdraw, with investigating Draw Mode
		Optimize Image Effects with different settings
Memory Bandwidth	Need to reduce memory used by texture, so that the rate to read and write to its memory by GPU should not low	Reduce the Texture Quality Settings
		Lower the texture size
		Use Texture compression to reduce the size of the textures.
		Use mipmaps for objects those are far from camera
Vertex processing	Lower the number of vertices or operation performed per vertice	Lower the number of vertex's
		Use normal maps to create illusion of detail
		In case not using normal mapping, then disable vertex tangents for the mesh
		Use LOD for objects those are far from Camera
		Reduce vertex shader complexity

VI. CONCLUSION

This paper presented performance analysis and optimization techniques for Unity3d. Where we have used Unity3d inbuilt profiling tools to evaluate the different techniques. We have analysed the performance of Scripting API, memory management and graphics rendering. In the Scripting API performance analysis, we have compared different functions, programming styles and APIs, and classified slow and fast API calls. In memory management we have discussed heap and stack memory and their optimisation techniques. In the Graphics Rendering we have discussed some optimization techniques to lower the GPU and CPU overhead. Presented work added the contribution to the 3d computer graphics development community.

REFERENCES

1. UNITY 3D. 2015. Unity: The Leading Global Game Industry Software (accessed August 2015).
2. https://www.unrealengine.com/en-US
3. https://www.cryengine.com/
4. https://shiva-engine.com/
5. https://gamemaker.io/en/gamemaker
6. Messaoudi, F., Ksentini, A., Simon, G., & Bertin, P. (2017). Performance analysis of game engines on mobile and fixed devices. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, *13*(4), 1-28.
7. Andrade, A., Game engines: a survey, EAI Endorsed Transactions on Serious Games, 15(6): e8, 2015.
8. Sharma, B., Singh, N. P., Mantri, A., Gargrish, S., Tuli, N., & Sharma, S. (2021, October). Save the Earth: Teaching Environment Studies using Augmented Reality. In *2021 6th International Conference on Signal Processing, Computing and Control (ISPCC)* (pp. 336-339). IEEE.A. Meijers, "Unity Advanced," in Immersive Office 365, 2020. doi: 10.1007/978-1-4842-5845-3_5.
9. Sharma, B., & Mantri, A. (2019). Augmented Reality Underpinned Instructional Design (ARUIDs) for Cogno-Orchestrative Load. *Journal of Computational and Theoretical Nanoscience*, *16*(10), 4379-4388.
10. Karuppusamy, Dr P. "Effective Test Suite Optimization for Improving the Coverage Standards Using Hybrid Wrapper Filter Memetic Algorithm." Journal of Soft Computing Paradigm 2, no. 2 (2020): 83-91.
11. Shakya, Subarna, and S. Smys. "Reliable Automated Software Testing Through Hybrid Optimization Algorithm." Journal of Ubiquitous Computing and

Communication Technologies (UCCT) 2, no. 03 (2020): 126-135.

12. Ulbrich, C., & Lehmann, C. (2012, August). A DCC pipeline for native 3D graphics in browsers. In *Proceedings of the 17th International Conference on 3D Web Technology* (pp. 175-178).
13. Evans, A., Romeo, M., Bahrehmand, A., Agenjo, J., & Blat, J. (2014). 3D graphics on the web: A survey. *Computers & graphics*, *41*, 43-61.
14. Ihemedu-Steinke, Q. C., Sirim, D., Erbach, R., Halady, P., & Meixner, G. (2015). Development and evaluation of a virtual reality driving simulator. *Mensch und Computer 2015–Workshopband*.
15. Berdak, P., & Plechawska-Wójcik, M. (2017). Performance analysis of Unity3D engine in the context of applications run in web browsers. *Journal of Computer Sciences Institute*, *5*, 167-173.
16. Shin, I. S., Beirami, M., Cho, S. J., & Yu, Y. H. (2015). Development of 3D terrain visualization for navigation simulation using a Unity 3D development tool. *Journal of Advanced Marine Engineering and Technology*, *39*(5), 570-576.
17. Nandy, A., & Chanda, D. (2016). Introduction to the game engine. In *Beginning Platino Game Engine* (pp. 1-17). Apress, Berkeley, CA.
18. Lin, J. R., Cao, J., Zhang, J. P., van Treeck, C., & Frisch, J. (2019). Visualization of indoor thermal environment on mobile devices based on augmented reality and computational fluid dynamics. *Automation in Construction*, *103*, 26-40.
19. J. D. Bayliss, "Developing Games with Data-Oriented Design," in 2022 IEEE/ACM 6th International Workshop on Games and Software Engineering (GAS), 2022, pp. 30–36.