Methods to Accelerate Rendering

GDEV60001 GAMES DEVELOPMENT PROJECT

WRIGHT Jack P E | W012423L
STAFFORDSHIRE UNIVERSITY

SUPERVISOR: KIERAN OSBORNE
SECOND SUPERVISOR: DAVID WHITE

Table of Contents

Glossary

- AIB – Add-in Board, extends the functionality of a computer through the motherboard.
- Instanced / Instanced Rendering – rendering many of the same mesh rapidly, through distributing each instance across the GPU.
- Draw Calls – A call to draw a specific mesh, outputting the results on to the back buffer.
- Buffer – a bunch of data stored in together, such as the render output, or instance specific data.
- PCIe – PCI Express, an expanded version of the Peripheral Component Interconnect.
- VRAM – Video Random Access Memory, like RAM but on the graphics card.
- LOD – Level of detail, used to indicate the level of detail in a model for the sake of graphics optimisation.

Keywords

Instancing, pixel overdraw, bandwidth, rendering, graphics, buffer, optimisation.

Introduction

Rendering techniques have advanced significantly over the last 40 years, along with this, graphics cards have also dramatically changed (Amiri, 2024).

The exploration of efficient rendering is imperative as games keep getting bigger – with graphical fidelity being a key factor for player immersion (Gerling *et al*., 2013). One example of optimisation to rendering can be shown through the abundant use of foliage – many large open world games are filled with foliage such as trees, bushes, and grass. Szijártó and Koloszár (2003) says that rendering foliage is required, especially in outdoor scenes and Karmaker (2016) says that foliage brings life to environments. Karmaker (2016) goes on to show that many games, such as the Far Cry franchise, have foliage that can account for up to 70% of the screen – meaning that players are seeing foliage for most of their playtime.

If every piece of foliage were its own draw call, the performance of the renderer would suffer a significant performance overhead as Wloka (2003) demonstrates, showing that the best way to use the graphic cards performance is through the number of batches, not the number of triangles, rendered. One way in which this is achieved is using instancing. Instancing is a fundamental technology in speeding up rendering of large scenes by reducing the need for the CPU to build and dispatch draw calls. As the CPU must dispatch a draw call (Wloka, 2003), which can change the data involved, the graphics card cannot predict or optimise for the next call. Instancing is achieved due to the massive parallelisation power of modern graphics cards, allowing distribution of work among many cores (Wynters, 2011, pp. 58-66).

Graphic card bandwidth is another factor in the rendering of foliage. Two things that must be considered when dealing with graphic card bandwidth are the transport of data from the CPU to the GPU and the GPUs ability to process that data quickly. Firstly, the CPU must provide the data to the GPU over a bus, freezing the GPU until the data is passed. Instancing reduces the data in transport to the GPU, allowing the graphics card to run uninterrupted. This optimisation is especially critical when rendering large, open world games – such as at a AAA level. Secondly, the graphics card may not be able to process all the data at once (Hovland, 2008), stalling the graphics card until it is all eventually copied across. If the data set is large enough, it may even make a game unplayable.

Another factor in the rendering of foliage is pixel overdraw. This is where there is more than one element being shaded and rendered in the same output pixel as another. Assuming there is no optimisation strategy in place to reduce overdraw, the graphics card can be doing work that is not visible in the final output (Corbalán-Navarro, *et al*., 2021). Overdraw is not a problem that can be solved completely – some overdraw is expected, such as when a smaller object is in front of a larger object.

This paper explores how the rendering of large open worlds are achieved and how this is performed in an efficient manner. This paper covers from the early days of 3D graphics and rendering though to modern day rendering technology; exploring how graphics card architecture evolved to exploit parallelism, allowing massive amounts of work to be calculated in a fraction of the time.

Literature Review

In the beginning of PC gaming, there were no dedicated Graphics Processing Units (GPUs). Games had to use the CPU to render. An early example of this is Quake, on the idTech 2 Engine (Amiri, 2024), which originally rendered in a software only mode. However, a software renderer is limited in terms of performance as CPUs have much more limited parallelism capabilities to modern graphics cards, this is further reinforced that, during the era in which software renderers were dominant, CPUs had very few cores to use, compared to today. As Caulfield (2009) explains, graphics cards can split a large problem into millions of smaller tasks and execute those tasks in parallel, whereas a CPU is designed to manage tasks in serial efficiently.

History of Early Graphics Cards

Add-in graphics acceleration units were only just becoming available for consumers at an affordable price in the late nineties, where the SST-1 card costing 300 US dollars (USD) new, which equates to about 600 USD today (Peddie, 2023, p. 252). This price allowed a mostly accessible graphics acceleration card in a time that they were primarily for research or industry, and not the consumer. This seeded a new market for graphics cards dedicated to personal and home use. To contrast the SST-1 card, the IBM XGA was released in 1990 which had a minimum cost of 1095 USD – about 2650 USD now (Peddie, 2023, p. 140), which is more than the highest end graphic card being release at the time of writing – the RTX 5090, with a launch price of 1999 USD (Nvidia, 2025). The entry price for the IBM XGA, therefore, was far higher than a vast majority of people could afford.

This inability to purchase add-in graphics cards at an affordable price was a driving factor in software rendering's prominence at the time. Quake (1996)'s two rendering modes, software, and hardware through GLQuake. The CPU would manage the entire render pipeline when using the software render mode, whereas an add-in board (AIB) would manage parts of the render pipeline when using OpenGL, this allowed the CPU to offload work to the AIB. An AIB is purpose built to manage computationally expensive parts of the graphics pipeline.

One of the most successful (Peddie, 2023, p. 252) early graphics cards was the 3dfx Voodoo1, which released in 1996 (Peddie, 2023, p. 252). Peddie (2003, p. 252) also states that 'the SST-1… was the first video subsystem that enabled PCs and low-cost video game platforms to host 3D entertainment applications', indicating that, before this, graphics acceleration chips were either too expensive for a typical consumer, or were simply unavailable for use within a standard computer – with the former being back up by the aforementioned IBM XGA card (Peddie, 2023, p. 140).

Another early attempt at a commercial graphics card at this time was the Nvidia NV1 chipset (Peddie, 2023, p. 245) which prioritised using quadrilaterals instead of triangles for rendering, to reduce the amount of work the CPU must do processing polygons – in this case, one quad instead of two triangles. The NV1 chipset, however, was a commercial failure due to a multitude of factors, such as DirectX specification requiring triangle-based rendering as well as Sega backing out of a deal with Nvidia (Peddie, 2023, p. 246). As technology was becoming increasingly complex at this time, software – such as DirectX – that utilised graphic acceleration cards determined where the

development of these add-in boards went, such that when Microsoft announced that DirectX would be utilising a triangle polygon rendering approach (Peddie, 2023, p. 246), this all but ended the Nvidia NV1 chipset. The NV1 had only one option to be a real choice for consumers and that was relying on emulation and the fact that the NV1 chip supported the whole Direct3D stack (Glatter, no date).

Since the SST-1, which became the commercial 3dfx Voodoo1 graphics card, was a dominant choice for consumers, a look at how it works and why early graphics cards rendered the way they did.

The SST-1 (Peddie, 2023, p. 249) can support up to three texture mapping units, also known as 'TREX', which then has each texture from the prior TREX get combined with the next. Once the textures have been processed, the combined result is then passed to the Frame Buffer Interface ('FBI'). This is where the pixel processing occurs, the FBI performs some important, but fixed, functions, such as scan-line conversion, Z-buffer, blending and texturing (Glatter, no date). The pixel is then put into the frame buffer, which is also used as one input to the blending stage.

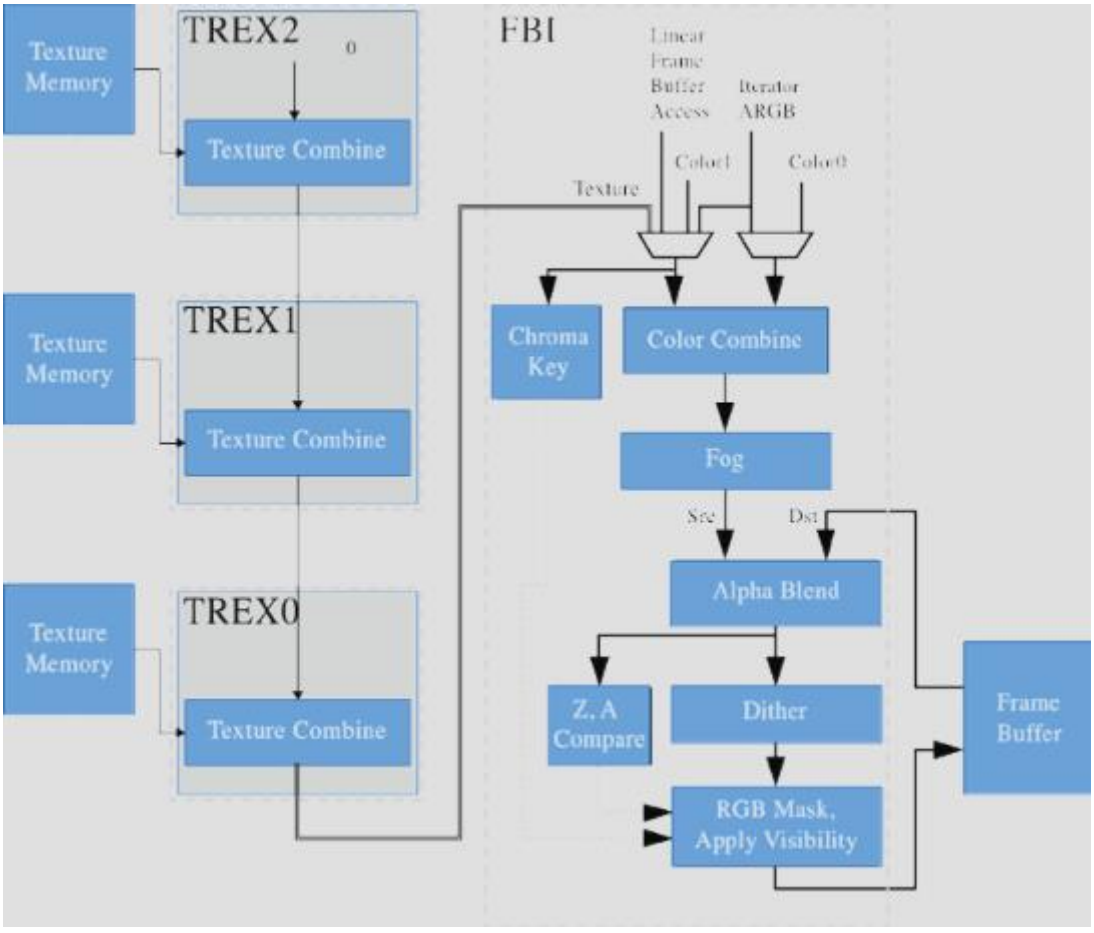


Figure 1 - SST-1 Diagram (Peddie, 2023, Fig. 6.17)

As shown by *Figure 1*, the graphics pipeline is very fixed – there is no programmer input on the processes that happen on the chip. A programmer submits a command and data, and all the work happens in the background. One on hand, the programmer does not need to worry about what is happening on the card, but on the other, there is no way to change what is being done on the card.

Early Graphics APIs and Traditional Rendering

Software Verses Hardware Rendering

Many games in the 1990s and early 2000s came with two (or more) modes of rendering, depending on the hardware present in the computer that the game is running. Quake, when it originally released, only supported a software renderer (Amari, 2024), meaning that the whole game was rendered using the CPU present within the computer. This has the benefit of being able to be ran, whether it is at a smooth and consistent framerates or not, on any machine. However, the downside of this is that the CPU is not designed to manage large volumes of repetitive maths calculations efficiently (Ghorpade *et al*., 2012). This is especially true when compared to modern day graphics cards – which are designed to process a lot of maths very quickly in parallel on the same data, with the advent of general-purpose graphics processing units (GPGPUs) (Ghorpade *et al*., 2012). Id Software ported quake officially to use OpenGL and released in 1997 (Peddie, 2023, p. 158).

Figure 2 - (Rejhon, no date)

As seen with *Figure 2*, the improvements provided by a graphics accelerator cannot be understated at this time – from the squaring of the number of colours, to the near five times higher resolution, all whilst running at the same framerate. The bonuses came with the downside of requiring an extra, expensive, piece of hardware that many home computers at the time would not have had.

Graphics APIs and the Route to Flexibility

Graphics APIs in the early days of hardware accelerated games, and even today, dictated much of how graphics cards were (and are) designed. As previously mentioned, the NV1 card from Nvidia was not suitable for market with the release of Direct3D and its specific requirement of using triangles for rendering, at a time where using quads for rendering would have sped up CPU processing of geometry due to the relatively limited performance of CPUs at the time.

The SST-1 graphics engine from 3dfx Interactive came out about a year before Microsoft's Direct3D was released and could not use other graphics APIs due to licencing requirements (Peddie, 2023, p. 253). It required its own graphics API, which came to be called Glide. Glide was in competition with a few pre-existing APIs, such as OpenGL and 3DR. At this time in Graphics APIs, support for an API on a graphics card was all over the place. As an example, Glide was exclusively supported only by 3dfx cards as it was purpose-made for the architecture (Peddie, 2023, p. 253). However, support for OpenGL, which had dominance in games at the time, with Quake 2 and Half Life being the main games with support, came later with the release of the 'MiniGL' driver released for 3dfx cards (Shimpi, 1999) which was an incomplete implementation of OpenGL 1.1. Support was later added for Direct3D which came about with a driver update in 1997. This originally only added support for DirectX 2 and 3 which was then followed up later that year with DirectX 5 support (FalconFly, 2004).

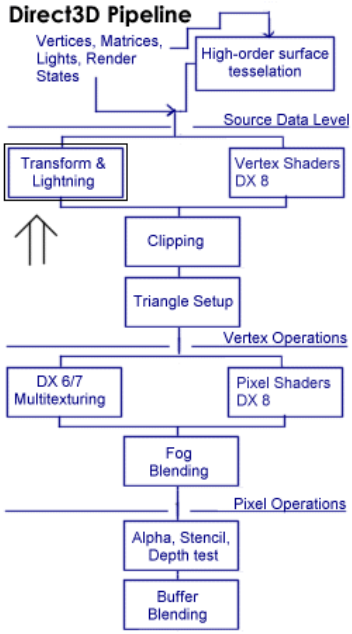


Figure 3 - Direct3D Pipeline (Engel, no date)

Not all parts of the traditional graphics pipeline were accelerated by hardware at this time, either. DirectX 7 introduced hardware accelerated transformation and lighting – which are two of the major steps in the graphics pipeline (Nvidia, 1999). This process, however, is managed directly by the graphics card with no input from programmers – beyond passing the required data.

As seen by *Figure 3*, the fixed-function, or 'immediate mode', pipeline for Direct3D – not including the DirectX 8 components (shaders) – the Transform and Lighting components in the pipeline are managed entirely by the graphics card, as well as clipping and triangle setup. A programmer at this point did not have much control over what was happening on the graphics card, with the only real change being done through the input data to the graphics card. Graphics cards at this point were still ASIC in nature, specifically made to perform a set action very efficiently and quickly.

This finally changed with the introduction of programmable shaders in DirectX 8.

The Advent of Programmable Shaders

DirectX 8 introduced programmable shaders in its first iteration, which allowed programmers to write their own vertex and pixel shaders – giving control of the primary visual method of rendering. However, support for programmable shaders was not at the level seen today yet, as DirectX 8 shaders required writing them in shader assembly (Dietrich, no date).

The move towards programmable shaders also brought about a new era of supporting feature-sets available in graphics APIs. This was due to the need for arbitrary shader program support, which, must be supported directly by the graphics card (Dietrich, no date) – which contrasts with graphics cards being required to support transform and lighting – but not how it is implemented. This means that the transform and lighting calculations can be different between different cards and card manufacturers – which could impact a developer's graphical intent.

Nvidia (no date) postulates that DirectX 8 started a 'compliance verses compatibility' argument, wherein a graphics card may not fully support the latest features and may not be compatible with the latest games. Vertex shaders have three versions possible within DirectX 8 – 0, 1.0 and 1.1 and Pixel shaders only support a version 1.0 and 1.1. Version 0 is DirectX 7 hardware, where the vertex shader is executed on the CPU, 1.0 is the most basic DirectX 8 hardware, with the minimum support for programmable shaders and 1.1 is an extension of that support (Nvidia, no date).

As well as the addition of programmable shaders, DirectX 8 introduced the beginnings of a far more generic graphics API, as some of the limitations of DirectX up until this point had been lifted, such as the number of lights – which was limited to 8 due to the fixed-function nature of DirectX 7 (Nvidia, no date). Following on from the programmable shaders of DirectX 8, DirectX 9 introduced the High-Level Shader Language – HLSL. This allows programmers to write shaders easier in a more natural way (White *et al*., 2020).

These advances in graphical APIs, specifically here – DirectX – paved the way for how games were to be rendered today, as well as graphics cards as general-purpose computing hardware – vastly different from where graphics cards came from, where they were purpose built to do one task.

The Transition from Fixed-Function to Programmable Pipelines

The two major graphics APIs have gone over the same path regarding a fixed-function pipeline transitioning into a programmable pipeline. DirectX 8 introduced programmable shaders (Dietrich, no date) as a concept and OpenGL 2.0 introduced its own version a few years later, in the form of GLSLang (Segal and Akeley, 2004, p. 341). The addition of programmable shaders was the first along a few major steps to a truly programmable graphics pipeline. Since shaders are one of the most important components within the graphics pipeline, as they are purpose made for transforming vertices as well as drawing each pixel to the screen (specifically for vertex and pixel shaders), the technology behind how this is executed becomes the deciding factor in graphic card performance.

Kirk (no date) states that, at the time of the release of the GeForce3 graphics card, that in the future, to expect a highly parallelised and programmable graphics pipeline. The GeForce3 was also the first card to actively support the whole of the DirectX 8 graphics pipeline (Dietrich, no date), as well as this, it supports a small amount of parallelisation in regards to pixel shaders – where the card has four pixel shaders, eight texture mapping units and four render outputs (ROPs) (TechPowerUp, no date) where each pixel shader can read from two texture mapping units each and this combined with the four render outputs. This parallelisation here is the foundation for what will become shading units.

As graphics cards progressed and advanced at that point in time, the ability for more parallelisation to take place occurred. For example, the Nvidia GeForce4 Ti 4200 has support for two vertex shaders and four pixel shaders (TechPowerUp, no date) and the Nvidia GeForce FX 5950 Ultra has support for

three vertex shaders and four pixel shaders (TechPowerUp, no date), both with the same amount of render outputs and texture mapping units, four and eight respectively.

At this point, there were still some fixed-function components remaining in graphics APIs. DirectX 10 was the point in which all fixed-function components were removed and OpenGL 3.1, the same. DirectX 10 also introduced the concept of a unified shader model (Nvidia, 2006). This changed how shaders are executed on the graphics card fundamentally. Instead of each shader having its own dedicated execution unit, as with previously mentioned graphics cards, each type of shader can be executed on the same unit, with the same instruction set. As well as this, the Nvidia GeForce 8800 supports up-to 128 shader/stream processors (Nvidia, 2006), which is an increase of four times over the previous generations highest combined vertex and pixel shader unit count (TechPowerUp, no date). This graphics card also introduced CUDA cores to the world, which allow for generalised processing of data intensive problems (Nvidia, 2006). The power of being such a generalised processor ushered in an era of graphics card accelerated calculations, for example, with physics – allowing the CPU more time to calculate more complicated problem, such as AI.

Whilst parallelisation existed prior to DirectX 10, DirectX 10 introduced the DrawInstanced and DrawInstancedIndexed commands, which reuses per vertex and per instance vertex buffers to render the same mesh many times (Microsoft, 2024). Graphics cards achieve this by distributing the workload across all its stream processors – allowing efficient use of graphic card resources.

If it was possible to batch every draw call to fill each stream processor, functionally saturating the graphics card, the most effective use of graphics card processing power would be utilised, whereas if each object was rendered with its own draw call, it would be the least efficient use of the graphics card (Nvidia, 2006).

Rendering Efficiently

Rendering a scene efficiently in any graphics API is an art that requires several factors to be considered. Each of these factors has different requirements to implement them correctly. These factors are efficient use of graphic card stream processors and load balancing, pixel overdraw, where there is either too many pixels being rendered in the same place or a texel is too small to be mapped directly to a pixel and the bandwidth of the graphics card itself, the ability for it to handle insanely large amounts of data at once.

Instanced Rendering

Instanced rendering is the process of drawing a set of vertices many times. (Khronos Group, no date) The Khronos Group, which manages OpenGL and Vulkan Graphics APIs, states that multiple draw commands for many meshes can be a performance problem. Nvidia (Cebenoyan, 2004, under '28.3.1 Optimizing on the CPU') states, in their documentation, that each draw call has a cost attached to it. This indicates that the less the amount of draw calls the faster the rendering will be.

Modern graphics cards contain many cores for processing a lot of data, the best way to use these is by distributing work across them all at once. A significant use of this parallel processing is by rendering many of the same mesh at the same time across these cores (Carucci, 2005). Examples of objects that can be instanced easily are ones that can be predetermined before being passed off to the graphics card – using the same shader but using different data. This could be any object that shares the same mesh but is repeated in many places across the world, such as grass, where a few different models make up each variation of grass.

Grass can also be rendered in such a way that visually each blade of grass is unique in an overly complex grass simulation. Fan, *et al*., (2015) provides a mechanism to render millions of blades of grass, with each blade being simulated. Based on the distance from the camera, each patch of grass has a different number of blades. Between level of detail patches of grass and graphics card accelerated collision responses, a fast simulation of a large volume of grass was achieved.

Another method for grass that does not need to all be uniquely simulated is to use several quads to draw a grass texture on each face in clusters – to give the effect that there is a lot of detail in a small area (Kelzer, 2004). This method also has the benefit of being able to animate the corner vertices of each plane of grass to give a wind effect on the grass, through distorting the base texture as the planes are no longer sized correctly for the texture.

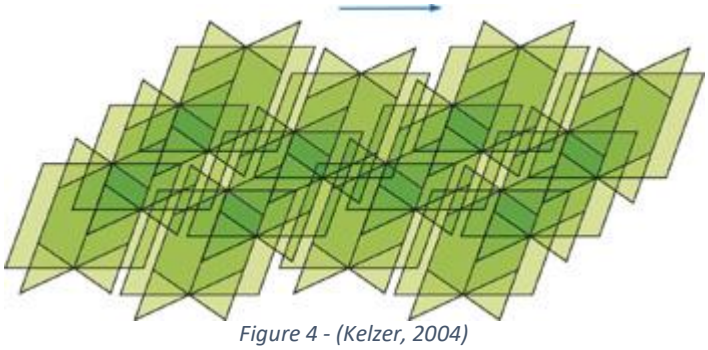


Figure 4 - (Kelzer, 2004)

As seen in *Figure 4* (Kelzer, 2004), the top vertices of each cluster of grass have been offset to the right, which is the cause of the distortion of the texture, creating the wind effect previously mentioned efficiently.

Another example is in the rendering of crowds, which is a far more complex problem to instance than grass. Gosselin, Sander, and Mitchell (2005) explain a highly effective method of randomizing a crowd without creating additional draw calls. In this paper, at a base level, a group of four characters are drawn together in one batch, each group is assigned an identifier which determines the colours that gets used to make the crowd more varied. These colours are then linearly interpolated against another texture to determine the final base colour of that group of characters. This results in a varied crowd without bogging down the graphics card with inefficient rendering of the crowd.

Figure 5 - (Gosselin, Sander, and Mitchell, 2005)

As seen in *Figure 5*, Gosselin, Sander, and Mitchell (2005) managed to create a diverse crowd that all simultaneously fit in with the world around them, as well as look distinct enough to portray a realistic crowd. The crowd would not be as effective if each character looked the same.

Pixel Overdraw

Graphics cards, on their own, do not know what to render or what order to render anything in. This could mean that if there are several objects in front of another, that the ones at the back could be rendered before the front one, causing the pixel to be fully transformed and shaded potentially several times before the final pixel is rendered. This is a waste of graphic card resources and, if it

happens too much, could affect the performance of renderer drastically. In highly detailed environments this could be a major concern. A city, for example, could have a building obscure many different things, but this can even occur with the same model. Sander, Nehab and Barczak (2007) presented an algorithm that reordered the vertices in a way that reduces overdraw, independent of the viewing angle relative to the model. It achieves this by drawing the polygons that are most likely to occlude others first, thus stopping any other pixels from being rendered in its place.

There are a few primary methods to deal with overdraw, each with their own downsides. The first, and most simple of these is the 'Early Z Test,' where the depth buffer is used to determine if the current pixel being rendered is behind one that already has been rendered (Corbalán-Navarro, *et al*., 2021). The worst-case scenario for this is if all primitives arrive from furthest to closest order, where every pixel will overwrite the prior one – causing no pixels to be culled. To effectively use this simple method, the vertices need to be provided to the graphics card in a front to back order (Corbalán-Navarro, *et al*., 2021) to allow the maximum number of pixels to be culled.

Another method for this is by using a 'Z Pre-pass,' which can be used with a deferred renderer. The pre-pass calculated the depth buffer before any shading takes place – specifically dispatching draw calls to the graphics card without a pixel shader attached (Sander, Nehab and Barczak, 2021). Then this depth buffer is used in the same way as the prior 'Early Z Test' but with the bonus of all pixels that will not be visible being clipped guaranteed. However, if a scene is overly complex and has a lot of geometry, the need to draw the whole scene twice may outweigh any benefits gained from this approach (Sander, Nehab and Barczak, 2021).

The final of the primary methods laid out in this paper is using occlusion culling. At a base level, occlusion culling is the practice of determining if a mesh is visible or not and not submitting it to the render pipeline if it is not (Hey and Purgathofer, 2001).

A common strategy for implementing occlusion culling is using a hierarchical representation of the scene. Hey and Purgathofer (2001) says that, through this method, if a building is occluded from view, then all the objects within the building will also be occluded. This saves on checking every object in the scene, vastly reducing the calculations required. However, Hey and Purgathofer (2001) also state that this is impractical for large scenes due to the solving calculation, in general, being expensive to perform in real time.

One strategy that games engines employ is using binary space partition (BSP) trees; early idTech engines and its derivatives use or did use this method. As previously mentioned, this calculation is expensive to perform in real time, so, to get around this, visibility is precalculated when building the game or map – this is true for when building the map in the case of the Source engine, based on GoldScr, which in turn is based on idTech 2 (Valve, no date). During the development of a map in engines that use a 'VIS (visibility) brush' method, wherein a map designer can setup visibility for a level, optimisation for expensive parts of a 'visleaf' (a BSP tree node) can be used to 'hint' at the engine to render the more expensive part of the node less (Valve, no date). One major downside for this approach is that the map must remain static or else the BSP tree node must be much larger, to account for any dynamic objects that would change the visibility of the scene if it were static. On the other hand, many games do not have massively dynamic environments, and ones that do tend to be on the smaller or more technical demo side, such as Teardown (Tuxedo Labs, 2022), which is a game with a fully destructible environment.

Bandwidth

Graphic card bandwidth is how much data at once the card can process. This is a fundamental part of the graphics pipeline as copying any data between the CPU and GPU is determined by the bandwidth of the GPU. Hovland (2008) states that solving problems with a graphics card can accelerate the process by up to seventy times. Depending on the context of the problem, this may involve having to send a large amount of data to and from the graphics card, such as simulating a complex scene's physics on a graphics card.

There are two major factors that determine the functional bandwidth between the CPU and the graphics card. The PCIe lane that the graphics card is physically plugged into the motherboard on and the internal bandwidth of the graphics card itself. The CPU itself is not a factor in the bandwidth equation as it is only active at the start of the transfer of data, affecting the latency and not the bandwidth (Hovland, 2008).

The bandwidth available on a PCIe connection is dependent on the generation of PCIe connection and the number of buses on the connection. Each generation of PCIe connection doubles in bandwidth, such that the bandwidth per lane per direction for PCIe generations 1.x, 2.x and 3.x is ~250MB/s, ~500MB/s and 1GB/s respectively (PCISIG, no date). On top of each generation of PCIe connection is divided into five sizes, with each containing double the PCIe bus lanes than the previous one. These are '1x', '2x', '4x', '8x' and '16x' with each also being double the bandwidth of the prior one too (Glawion, 2022).

PCIe bandwidth itself is dependent on both the PCIe input and connection being the same generation and size – to use the full bandwidth available. PCIe connections are backwards compatible so it is possible that a generation four PCIe device is connected to a generation three PCIe slot. PCISIG (no date) goes on to explain that the PCIe ecosystem will be running at the lowest supported PCIe speeds, such that in the example above, the generation four PCIe device will be limited to PCIe generation three speeds, bus count notwithstanding.

Graphics cards are designed with a particular PCIe generation and bus count in mind, with the Nvidia 5090 using a PCIe Generation 5 connection, with a '16x' PCIe connection type (Nvidia, 2025). This means that this specific graphics card requires the highest generation and bus count on the consumer market currently – this may mean that a consumer purchases a new graphics card and installs it onto a motherboard that is not the same PCIe generation as the new graphics card – which on its own may cause a major bandwidth bottleneck alone. As well as this, it is possible that the motherboard in question may have a '16x' slot but may only be wiring to be an '8x' slot (Glawion, 2022), bandwidth wise. This will have the same effect as a mismatched PCIe generation between card and motherboard – where the bandwidth is reduced.

Assuming the connection between the graphics card and the motherboard matches up and using PCIe generation five with sixteen bus lanes as an example, then the transfer rate is 32 GT/s (giga transfers per second, which also translates to 32 Gb/s) per lane (Jones, 2025). In this case, if the target framerate, specifically from a bandwidth perspective, is sixty frames per second, then the data transfer per lane per frame needs to be about 0.5 GT, or 0.5 Gb. Converting this to 62.5 MB per frame per lane, and if each vertex contains a position, normal and texture coordinate, all of which are floats, which is thirty-two bytes. This would mean that, in this case, the PCIe lane would support about approximately two million vertices, assuming no other data is transferred, which will not be the case in a real application. Since this example is using a PCIe connection with sixteen bus lanes,

then the total vertex count would be approximately 31.25 million, this is nearly four times the number of pixels on a 4K (3840x2160) monitor – which would cause massive pixel overdraw with many vertices overlapping per pixel. This is far beyond what is required to render a scene in a realistic scenario if accounting for other performance impacts previously mentioned.

Comparing the PCIe generation five with sixteen bus lanes maximum bandwidth with the memory bandwidth of graphic cards that supports the PCIe combination type, shows the PCIe is the primary bottleneck and overall, the bandwidth is not affected by the graphics card. The memory bandwidth of the Nvidia RTX 5060, which is an entry level graphics card with a known memory bandwidth, is 448 GB/s (TechPowerUp, no date), as well as this, the Nvidia RTX 5090, the current most powerful consumer graphics card, has a memory bandwidth of 1792 GB/s (Nvidia, 2025). Both are far higher than the PCIe bandwidth, of which is ~63 GB/s. Whilst the graphics card must use its memory for much more than copying data to and from the CPU, the PCIe will, in most cases, be the bottleneck, this is also backed up by Hovland (2008).

Results

To determine the effectiveness of the explored topics, instanced rendering, pixel overdraw and bandwidth, an artefact was created to explore each of these.

Result Capture Method

All performance metrics were captured using Optick (Slyusarev, *et al*., 2022) using a Ryzen 7 5800X, 32GB DDR4 RAM, Nvidia RTX 3080 and a X570 motherboard, conditionally compiling only the code necessary to run the specific renderer. As well as this, only 5 seconds of frames were captured for each test and an average taken from every frame recorded. This does mean for the slowest tests that there are fewer results than would be preferred for accuracy, however in many cases there are hundreds, if not thousands of samples taken – especially so in the case of the tests using fewer objects.

Due to this variance in performance, ranging from sub millisecond frame times to that of over fifty milliseconds, using frame counts as the defining factor would make some tests take a significant amount of time to produce. Other tests could also take, near enough, no time at all, depending on the tuning of the frame count, so that any timing related issues that could resolve themselves after a few seconds of running. This specific problem can be seen in the Optick capture where the first frames would have a far higher time than that of ones later.

Instancing

To demonstrate the differences between not instancing and instancing, multiple build configurations were added to conditionally compile the code required for each configuration to run without any extra overhead. Three configurations were added, firstly to demonstrate a no instancing – with each model being its own draw call, then secondly to demonstrate two different methods of instancing the objects in the scene.

The difference between the two different methods of instancing is between storing the world data into a Structured Buffer or by passing it per instance through the input assembler. The primary distinction between the two instancing methods is the difference between memory access levels. The first method will store the buffer in VRAM, which acts in an analogous way to how the CPU access RAM – it is one of the slowest memory accesses in the card. Whereas, through the input assembly, the data is passed into the streaming processor register, which will be accessed significantly faster than the VRAM.

The primary method of finding out what is the best approach is through direct comparison with each type of build configuration, with more granularity, through CPU and graphics card times and the percentages each take of the total frame time. The comparison will be controlled by the number of objects being rendered.

Pixel Overdraw

Demonstrating the performance impact of pixel overdraw is as simple as flipping the draw order. The artefact, by default, draws each set of models in order of distance from the camera – such that the lowest level of detail models are drawn last and the highest level of detail models are drawn first. By drawing the objects closest to furthest, the best-case scenario can take place. If a pixel is already shaded and drawn to the screen and is closer to the camera than another pixel that is about to be shaded it is discarded, skipping the pixel shader entirely, a process known as the Early Z Test, which is performed automatically on the graphics card. On the other hand, drawing from furthest to

closest will make this test fail almost always, since the next pixel rendered will be in front of the prior rendered pixel – causing a potentially massive performance blackhole.

Bandwidth

Bandwidth can be easily demonstrated through pushing far more vertices through to the graphics card. How this is achieved in the artefact is through several different variables, such as increasing the number of objects to render, increasing the distance at which the models swap to lower quality or having more objects per row, which in turn increases the number of full quality models in the scene.

To push the bandwidth, the highest level of detail (LOD) model has 7432 vertices, or 14848 triangles. Followed up by the next LOD model, with 164 vertices, or 314 triangles and is visually the same, and finally, the final LOD model is a quad, so, 4 vertices, or 2 triangles.

To overload the graphics card bandwidth, using many of the highest detail model will absolutely overload the graphic card's ability to process all the vertices.

The Artefact

The artefact can load in any number of objects in a grid defined by a width count and distance between each object. As well as, swapping between LOD models, based off the Z distance from the camera – so that each row will swap at the same time. The distance where each LOD model takes over can be changed manually, however, for each test conducted, it remained at a fixed distance, that which being one hundred units from the camera for the first LOD model and two hundred and fifty units from the camera for the second LOD model.

The artefact, by default, also sorts the draw calls in order of closeness to the camera, so that pixel overdraw is not an factor in many of the tests captured, as if this was not factored in, the performance could swing drastic from run to run and test to test as the position of each object could be random within the buffer – which could cause a worst case scenario one run and the best case scenario in another.

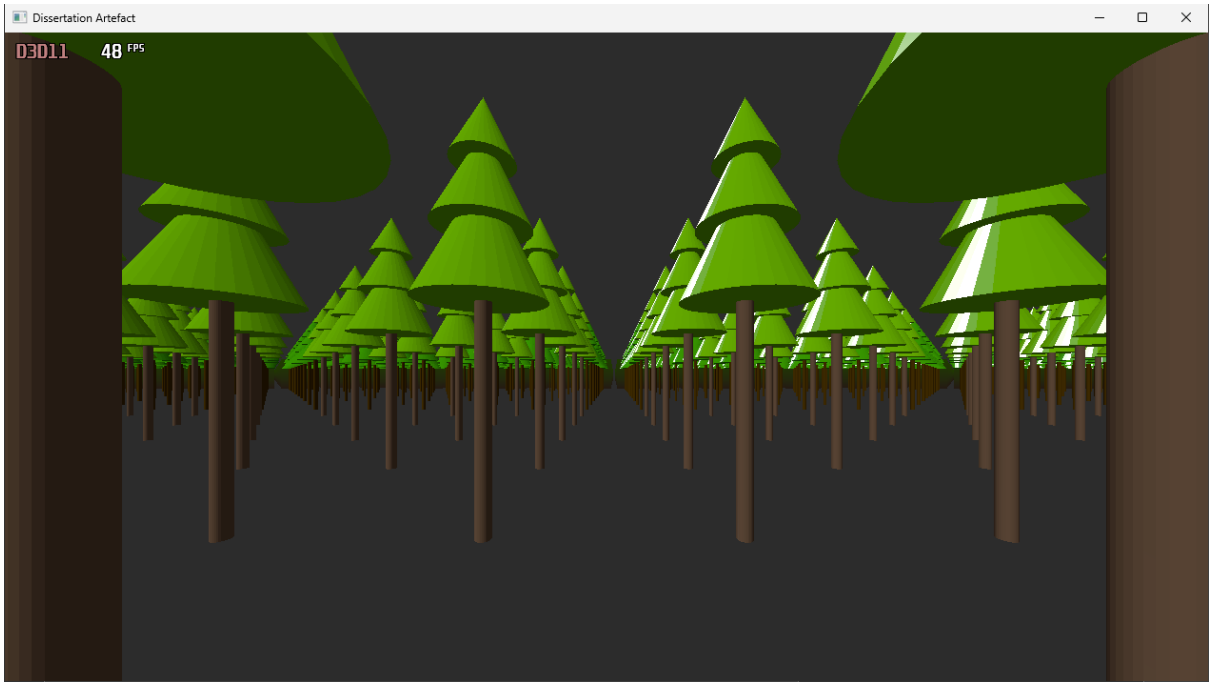


Figure 6 - Screenshot of the artefact running.

Overall Performance

To compare the overall performance of the artefact the most demanding tests will be used. This means that the tests conducted with five hundred thousand objects will be used and, whilst each test referred to here uses a different width count, they all share almost identical performance results.

To begin with, the run-to-run variance that can occur between each test is shown clearly with the renderer that uses no instancing whatsoever, where *Figure 18* and *Figure 20* have approximately an 5 to 8 millisecond difference to *Figure 19* and *Figure 21*. Due to the complexity of hardware, no one specific effect can be pinpointed to find why such a variance was produced, as well as this, it was consistent through running the same test again. Efforts were made to keep the testing environment consistent between tests, such as having minimal applications open and running the tests automatically through code. The main contributor to this run-to-run variance is the graphics card, as the primarily changing result in these tests is the 'GPU::Draw' result, whereas the 'Engine::Update' result had a variance of approximately 0.2 milliseconds at most – negligible compared to 5 to 8 milliseconds.

Next, comparing no instancing to any form of instancing instantly shows a drastic performance gain in favour of instancing, with an approximate speed up of about 2x – this is however a very shallow look at what is happening in the artefact.

Where the graphics card itself is concerned, there is a speed up of anywhere between 6-9x speedup, however, due to how the CPU must build the buffers for the instanced renderer, the CPU portion of the Draw function can be up to two thousand times slower. This is not a fair assessment, however, as the CPU has functionally nothing to do in the function before the code that runs the drawing itself takes place in the none instanced renderer, whereas, in the instanced renderer, the objects are sorted, and buffers are written to.

As for the 'Engine::Update' results, the CPU markedly takes half the time to run this code, however, the code is the same in all tests. Even beyond the current tests being used for analysis, the CPU takes less time in this function in the none instanced renderer than in the instanced renderers.

Between the instanced renderer where the world data is copied to the VRAM compared to the instanced renderer where the world data is passed to the input assembler, the latter performs faster at this number of objects. The rendering of the objects is between 0.5 milliseconds and 1.9 milliseconds faster and on top of this, the CPU can perform up to one millisecond faster – with both being measured in the Draw function. The Update function takes an approximately equal amount of time on average. This result, however, is not consistent overall, and when comparing this data to data captured on five thousand objects, the former method is faster (*Figure 7*, *Figure 8*, *Figure 9* and *Figure 10*).

Looking at the results for the five thousand object tests (*Figure 7*, *Figure 8*, *Figure 9* and *Figure 10*), the instancing, once again, is almost 2x faster in the best case scenario. There is significant performance scaling that occurs between the none instanced renderer and the instanced renderer when increasing the number of objects per row (width). In the case of both renderer types, there is a two millisecond increase in frame time between a width of 16 and a width of 128 (*Figure 7* and *Figure 10*). This, in turn, shows that, when rendering a relatively simple scene, there is a linear relationship between the number of objects per row and the frame time. This is shown again between the width of 16 and the width of 128, where in the width of 16, there is an approximate 0.5

millisecond difference between the frame times of none instanced render and instanced render – with the same approximate gap appearing at a width of 128.

From this high-level look at the overall performance gained from going from a none instanced renderer to an instanced renderer, there is no functional reason that an application should not use instancing, especially today, where graphic cards have thousands of streaming processes for massively parallel processing. As well as this, considerations should be made on whether to block copy instance specific data to the graphics card VRAM verses handing it off to the input assembler for use as a vertex shader input, as there are upsides and downsides to both methods here.

On the NVIDIA RTX 3080 used for testing, the input assembler can be slower than using the graphic cards VRAM, or more specifically, when using a small amount of data. This is an unexpected result as the input assembler provides near instant memory access as the data will be in the streaming processors register compared to the cache misses required to fetch from the VRAM. However, this result is consistent when the tests are repeated.

Triangle Count Increase and the Effect on Performance

The width values mentioned prior is where the triangle count come from in this artefact. With a higher width value, there are more objects that are using the highest detail models overall, as the LODs are swapped based on the Z distance from the camera, instead of raw distance from the camera. This increase in triangle count will be going up against the number of objects present in the scene overall. With more objects per row, the number of objects in the scene does not change, but instead are, overall, placed closer to the camera at any given time.

The graphics card will discard any pixels not visible on the screen, those that are outside of the view frustum or those that are rendered behind pixels that are closer to the camera, however, the graphics card must still transform the vertices into screen space to determine if this is the case.

Five Thousand Objects

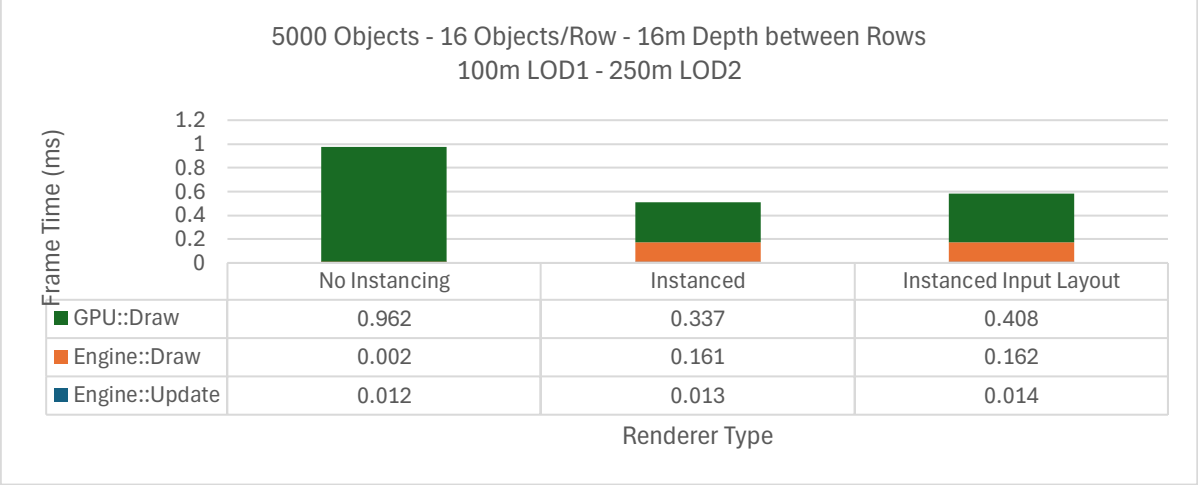


Figure 7 - 5000 Objects - 16 Objects/Row - 16m Depth between Rows 100m LOD1 - 250m LOD2

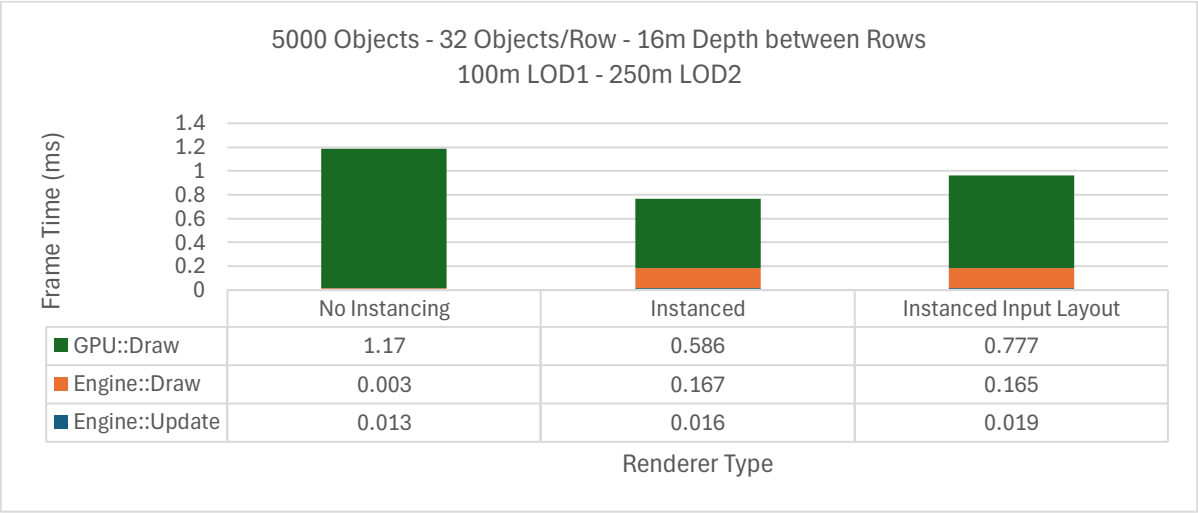


Figure 8 - 5000 Objects - 32 Objects/Row - 16m Depth between Rows 100m LOD1 - 250m LOD2

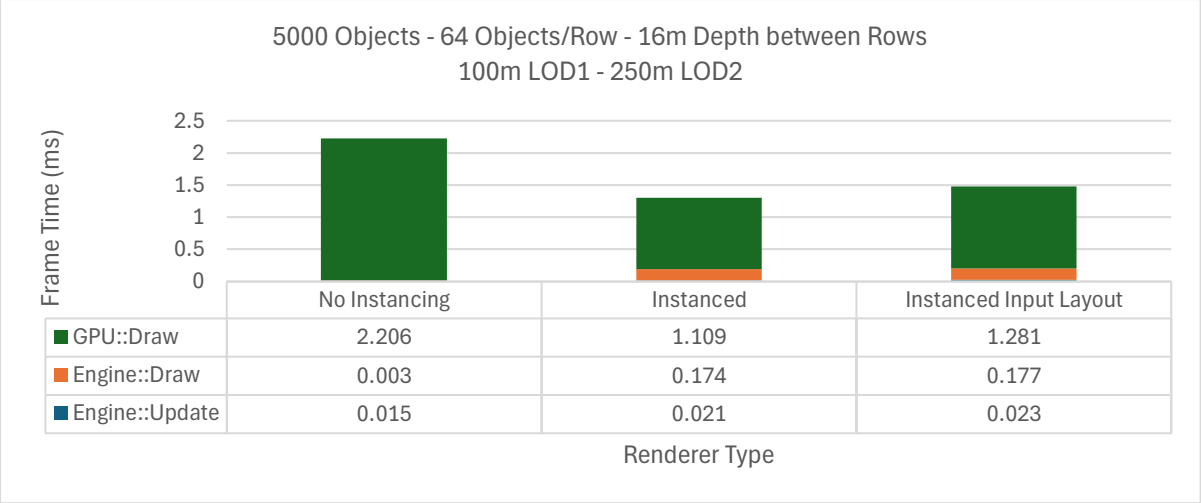


Figure 9 - 5000 Objects - 64 Objects/Row - 16m Depth between Rows 100m LOD1 - 250m LOD2

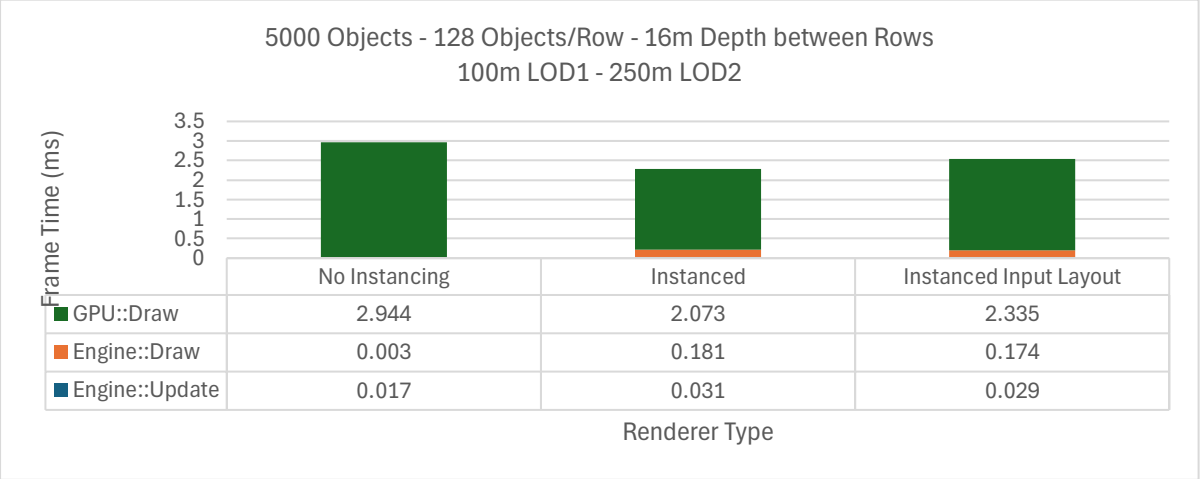


Figure 10 - 5000 Objects - 128 Objects/Row - 16m Depth between Rows 100m LOD1 - 250m LOD2

First will be a look at the performance characteristics when rendering five thousand objects, with respect to vertex count. *Figure 7*, *Figure 8*, *Figure 9* and *Figure 10* show the performance of each different width count for five thousand objects. In order, the width values are 16, 32, 64 and 128. This means that there are six rows of the highest detail model at any given point, as the LOD cutoff for swapping to the next quality down model is one hundred units from the camera, where there is sixteen units between each row – which is the same for every test, this gives test to test consistency, with fewer variables in play for each test, keeping each test fair.

This means that for each width value in ascending order, there are 96, 192, 384 and 768 highest quality models. Each of these models contains 14848 triangles, meaning that, at a 128 width, there are 11.4 million triangles dedicated to just the highest quality model. There is some overhead for extra shaded pixels, however this is negligible in the grand scale of the project. The quality of models are rendered in order of distance from the camera, so that the LOD0 model is rendered first, followed up by the LOD1 then LOD2 models – this means that most pixels are only rendered to once, through early Z pixel rejection.

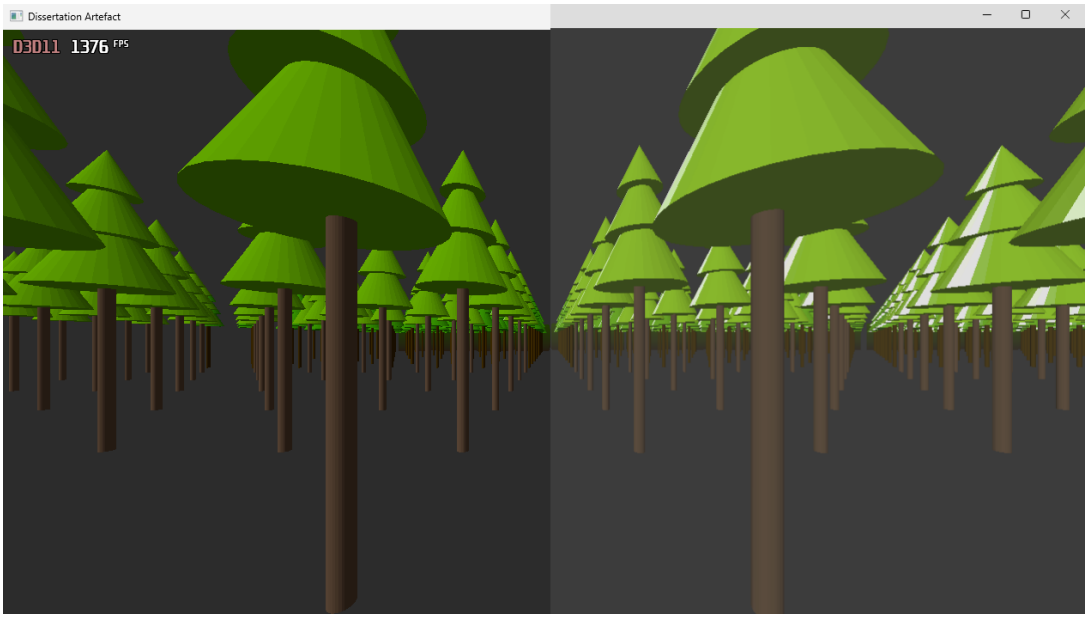


Figure 11 - Left: Width of 16 | Right: Width of 128

Figure 11 shows a comparison between the rendering of a width of 16 and 128. The pixels that get rendered in a width of 128 are only marginally higher than that of the width of 16, where the trees off to the side make up most of the extra shaded pixels.

Comparing the no instancing renderer first across the different widths reveals that there is a none-linear relationship between the number of triangles and the frame time. Going from a width of 16, to a width of 32, which functionally doubles the triangle count overall, only has a frame time gain 0.15 milliseconds. However, going from a width of 32 to a width of 64, has a gain of 1 millisecond to the frame time – this is nearly double the time, but not 4x the frame time of with a width of 16. The frame time is then increased by approximately 0.8 milliseconds when going from a width of 64 to a width of 128.

This overall is an unexpected result, as with each doubling of width, there is an almost doubling of triangles that the graphics card must run the vertex shader on. The graphics card cannot optimise any of these out, as the graphics card does not know where the vertices are until they have been transformed into screen space, requiring them to go through the vertex shader.

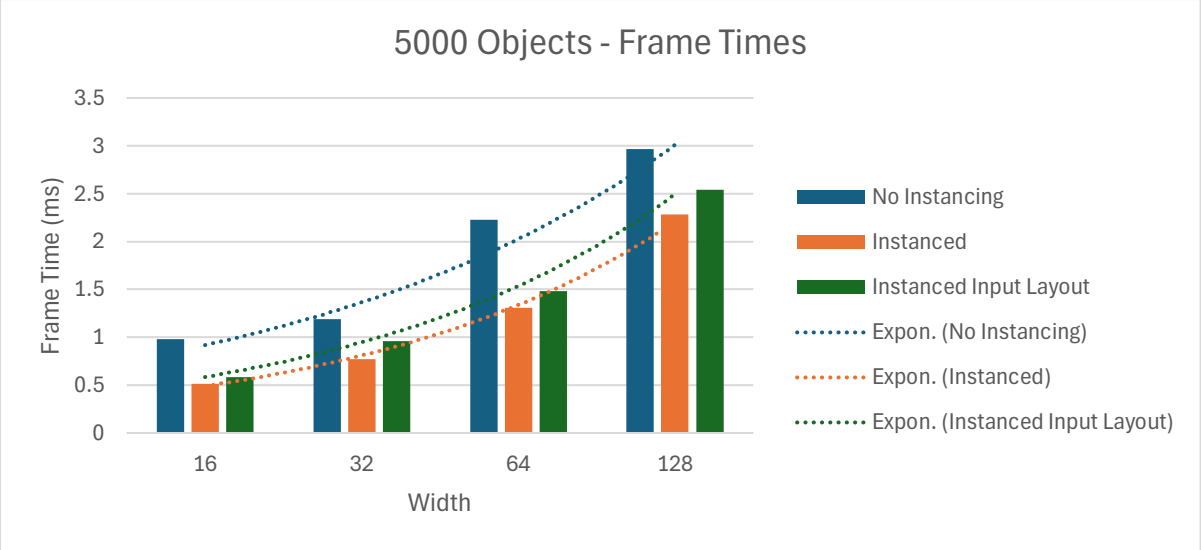
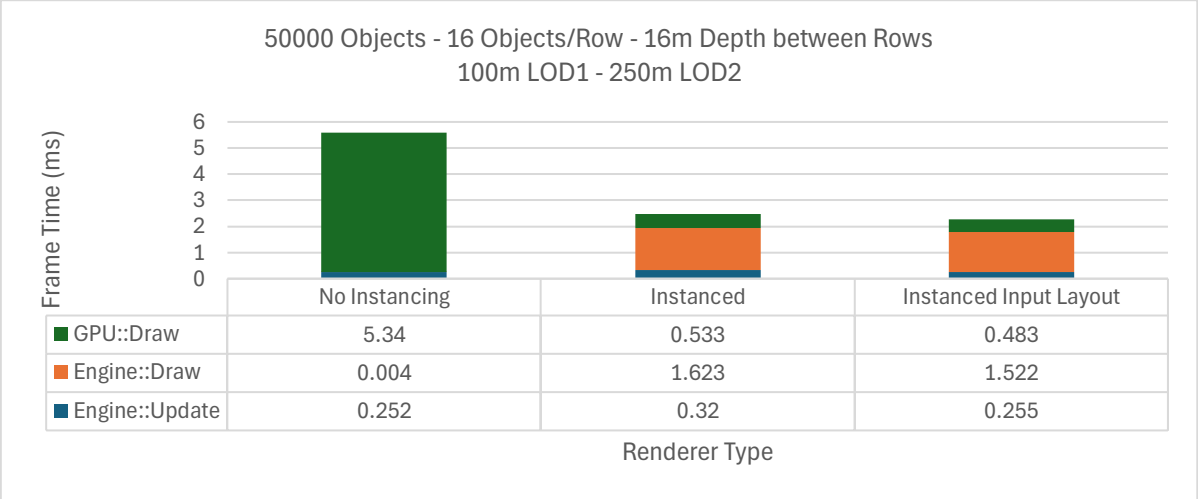


Figure 12 - Frame Times – 5000 Objects – Exponential Relationship

As *Figure 12* shows, there is an exponential relationship between the frame times going up and the widths.

Fifty Thousand Objects



	No Instancing	Instanced	Instanced Input Layout
GPU::Draw	5.34	0.533	0.483
Engine::Draw	0.004	1.623	1.522
Engine::Update	0.252	0.32	0.255

Figure 13 - 50000 Objects - 16 Objects/Row - 16m Depth between Rows 100m LOD1 - 250m LOD2

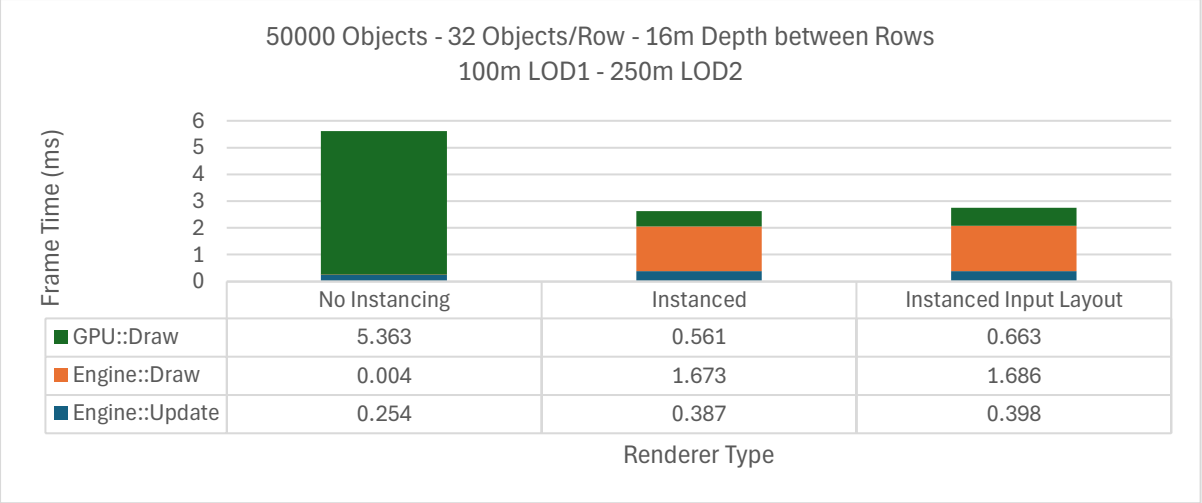


Figure 14 - 50000 Objects - 32 Objects/Row - 16m Depth between Rows 100m LOD1 - 250m LOD2

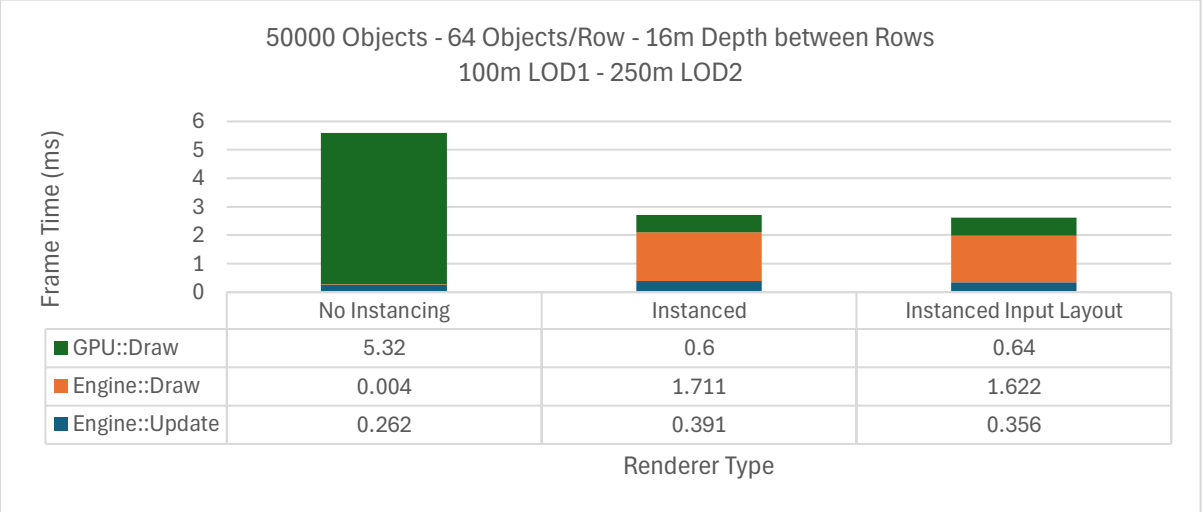


Figure 15 - 50000 Objects - 64 Objects/Row - 16m Depth between Rows 100m LOD1 - 250m LOD2

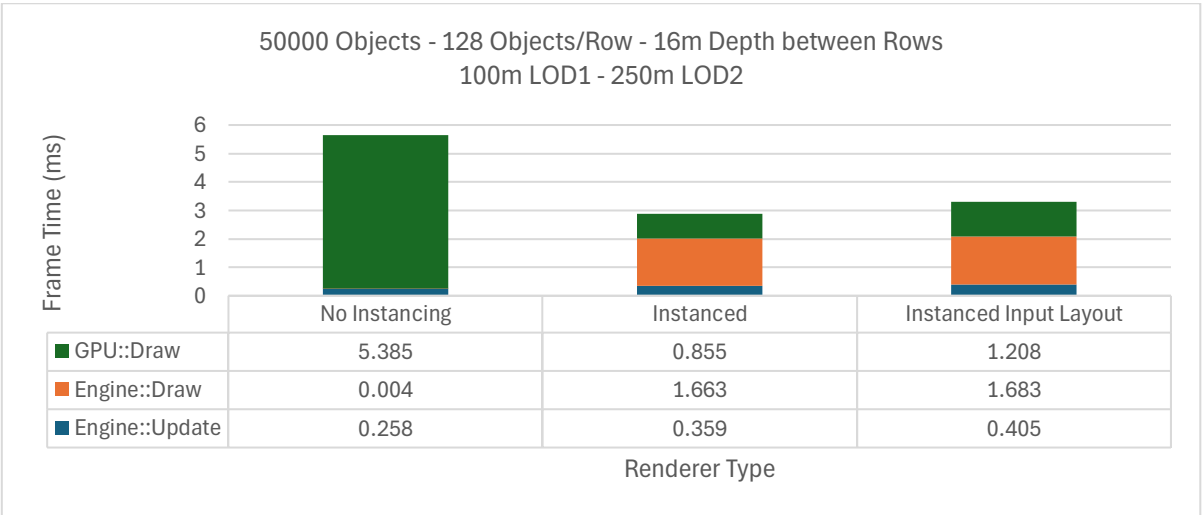


Figure 16 - 50000 Objects - 128 Objects/Row - 16m Depth between Rows 100m LOD1 - 250m LOD2

Looking at the same triangle count increase across different widths, when there are fifty thousand objects tells a different story to that of five thousand.

The no instancing renderer has no scaling present whatsoever across the different widths. There is, at most, a 0.1 millisecond variance across the frame times of all no instancing widths, this is too small a variance to take any real meaning from and is a run-to-run difference. This shows there is far more going on with fifty thousand objects on the graphics card than the number of vertices being processed through the vertex shader.

Scaling is, however, present on both instanced renderers. Both renderers exhibit different scaling, with the instanced renderer that uses a Structured Buffer to store all the world data having a relatively linear relationship with frame time and the instanced renderer that uses the input assembler for the world data having a more exponential relationship, as was present in the five thousand object tests.

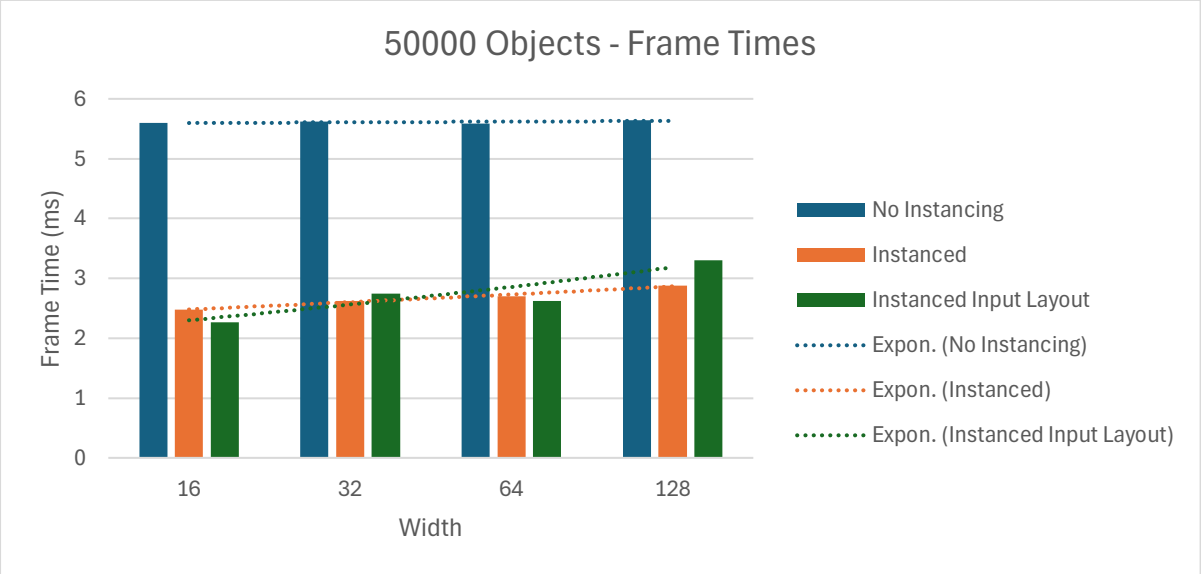


Figure 17 - Frame Time - 50000 Objects - Relationships

Overall *Figure 17* shows the lack of effect on the frame time with a virtually flat trend line for the no instancing render. The instanced renderer that uses the Structured Buffer has a straight line that trends upwards, showing the near linear relationship here. Whereas the instanced renderer that uses the input assembler that trends upwards, with a relatively substantial increase of frame time when going from the width of 64, to the width of 128.

Five Hundred Thousand Objects

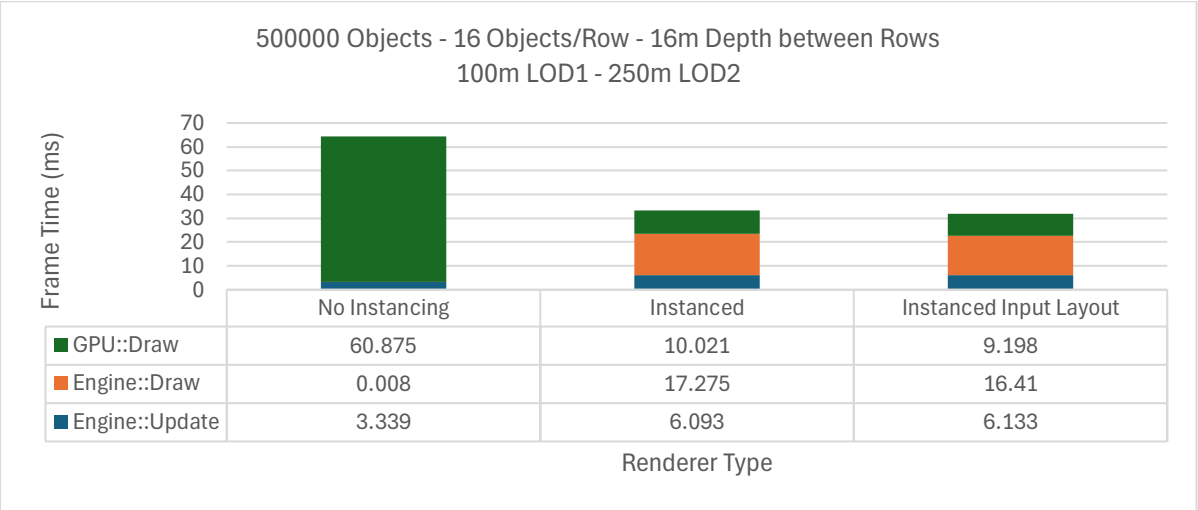


Figure 18 - 500000 Objects - 16 Objects/Row - 16m Depth between Rows 100m LOD1 - 250m LOD2

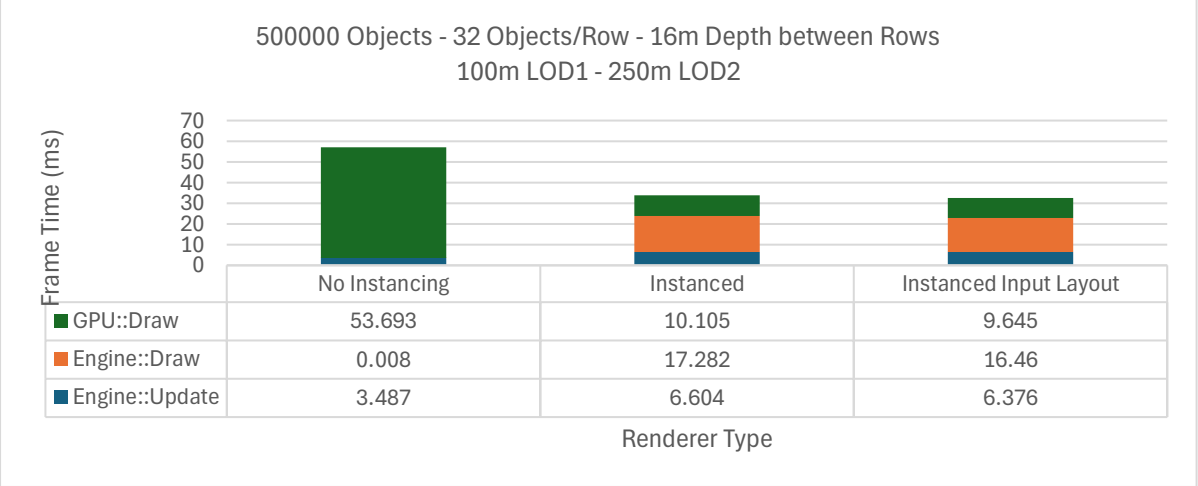


Figure 19 - 500000 Objects - 32 Objects/Row - 16m Depth between Rows 100m LOD1 - 250m LOD2

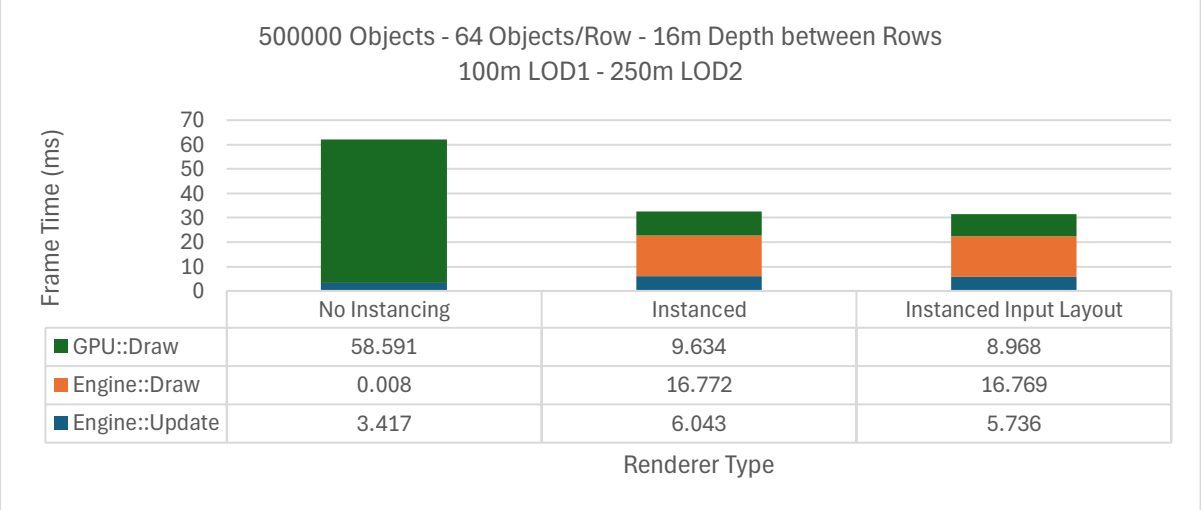


Figure 20 - 500000 Objects - 64 Objects/Row - 16m Depth between Rows 100m LOD1 - 250m LOD2

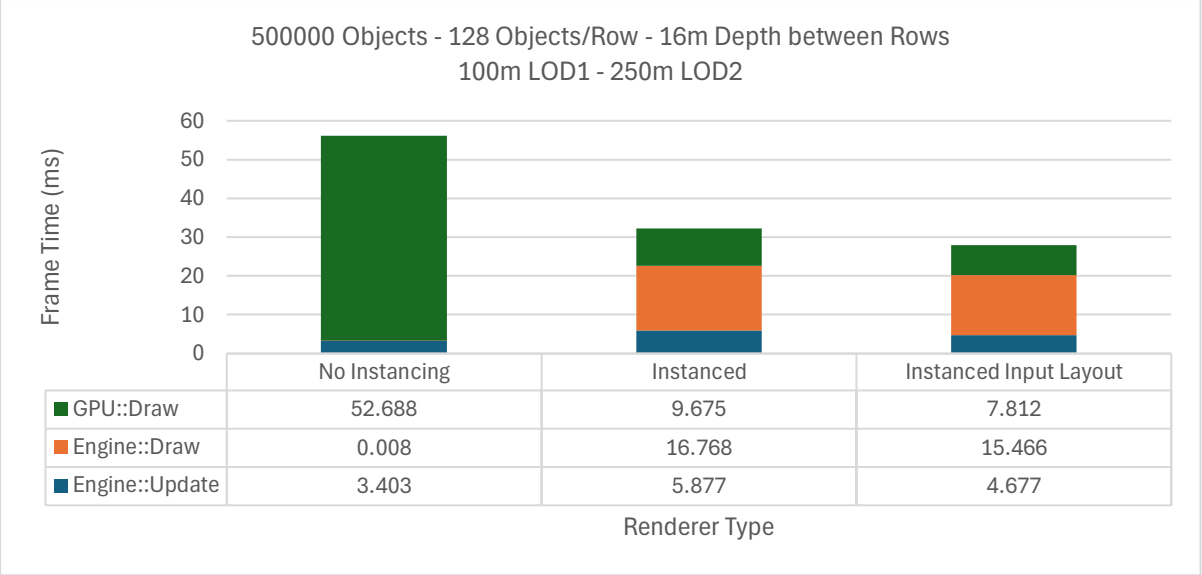


Figure 21 - 500000 Objects - 128 Objects/Row - 16m Depth between Rows 100m LOD1 - 250m LOD2

The results produced for five hundred thousand objects are much different to that of fifty thousand or five thousand. Instead of there being an upwards trend, there is a negative trend in frame time – this is consistent across all renderer times.

This result is entirely unexpected, especially due to the number of vertices the graphics card will be presented with, regardless of the width of the specific test, which is up in the tens of millions per frame.

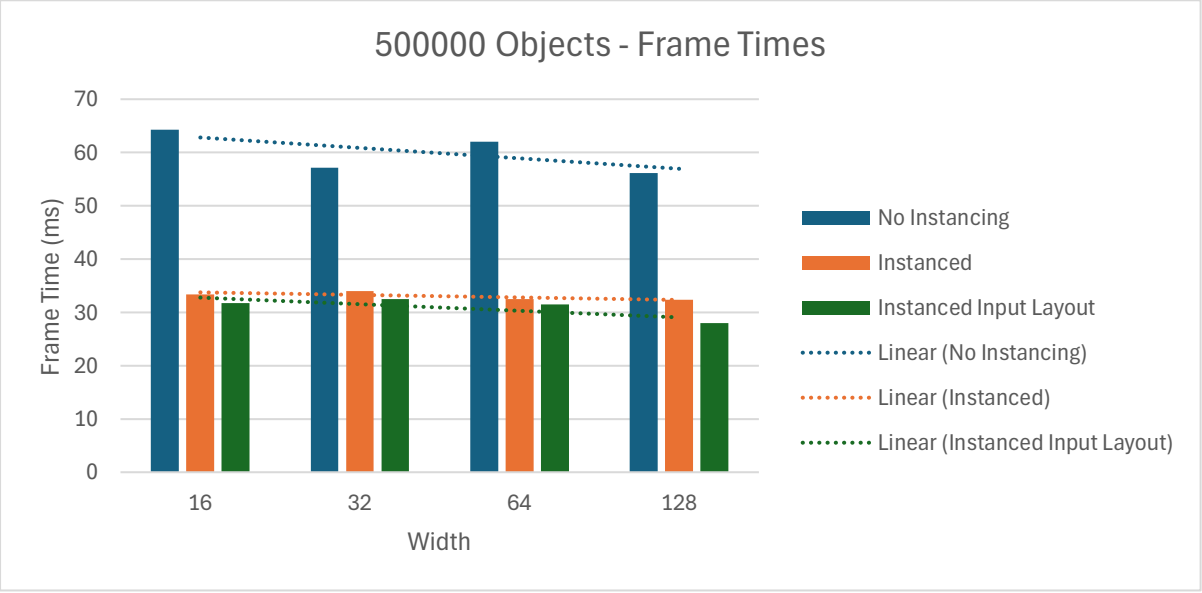
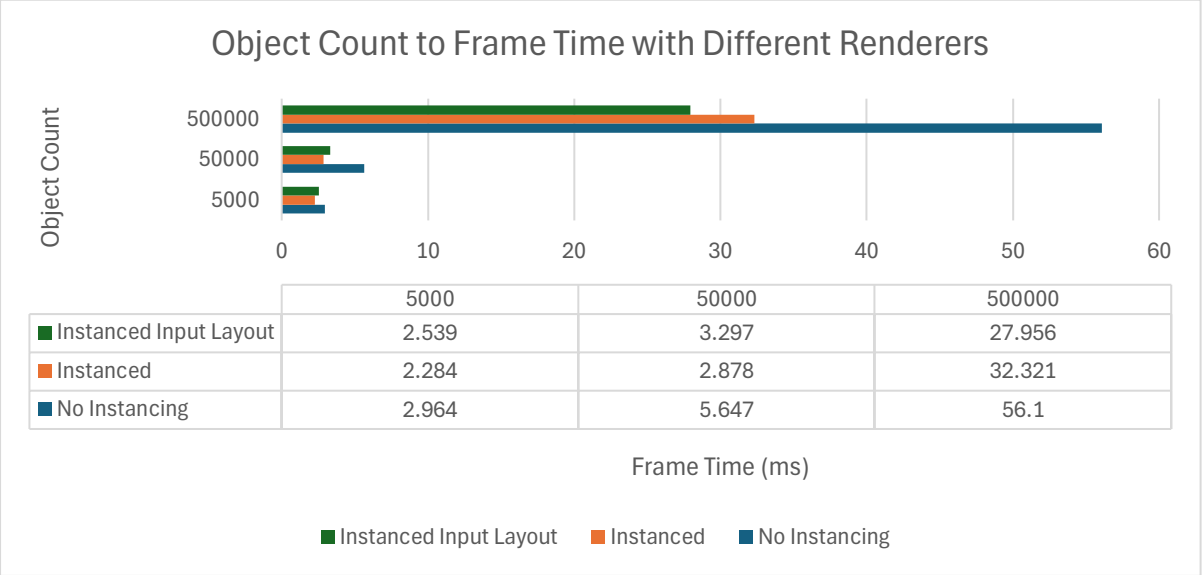


Figure 22 - Frame Times - 500000 - Negative Frame Time Relationship

Only speculations can be drawn from this data as there is no way to be certain what is happening on the graphics card during each of these tests, as the graphics card driver can do unexpected things.

Instancing and its Effect on Frame Time



	5000	50000	500000
Instanced Input Layout	2.539	3.297	27.956
Instanced	2.284	2.878	32.321
No Instancing	2.964	5.647	56.1

Figure 23 – Object Count to Frame Time – Plotted Against Each Renderer Type – Width 128

These results are looked at in a comparable manner to that of the effect of triangle count on frame time, however, instead of comparing by row, these are instead compared by object count. In this case, to keep the results fair, the width used is 128 – which has the highest impact on the graphics card of all the tests presented so far. *Figure 21*, *Figure 16* and *Figure 10* are the graphs used for this comparison, which are collated into *Figure 23*.

This section is dedicated to dissecting what is happening under the hood and how these play into the performance of each renderer type, such as what the CPU or graphics card is doing and why this causes the performance metrics captured.

With the frame times of the five thousand object results being so close, and with the no instancing renderer being marginally slower, the time difference here is to do with the dispatching of draw calls themselves. *Figure 10* shows that the dispatching of draw calls is over 99% of the frame time – with the GPU::Draw result defining this. Comparing this directly to either instanced renderer, there is extra CPU overhead from sorting and building the buffers that are to be rendered by the instanced draw calls. This CPU overhead of approximately 0.2 milliseconds saves the graphics card anywhere between 0.6 to 0.8 milliseconds, saving 0.4 to 0.6 milliseconds overall. With frame times so small in this sample this is a significant percentage – with an approximate 20% decrease in frame time.

As the number of draw calls increases for the none instanced renderer, the amount of time it takes to dispatch draw calls increases exponentially. The frame time increase when going from five thousand to fifty thousand objects is almost double, then going to five hundred thousand objects it increases ten-fold. As each draw call is only submitting one object to be rendered at a time to the graphics card, thirty-two streaming processors are being consumed for something that is being ran on only one of these, as on an Nvidia graphics card, each command consumers at least one block of threads on a streaming multiprocessor (SM) (Nvidia, no date). An RTX 3080, as was used in these tests, has sixty-eight SM cores and each contains 128 streaming processors (TechPowerUp, no date). This results in massively wasted performance on a graphics card, depending on the amount of draw calls that are dispatched to the graphics card – in this case the RTX 3080 only capable of processing 272 objects at any one time.

272 objects are in stark contrast to the potential 8704 objects that an RTX 3080 can manage at once, in the best of circumstances – using every available shading unit/streaming processor available per SM. However, in an unexpected turn of events, the graphics card time for the instanced renderers rendering fifty thousand objects is less than that of five thousand objects. This is completely backwards to what has been discussed, as fifty thousand objects requires the RTX 3080 to use each streaming processor at least five times, with many also being used six times per frame – whereas with five thousand objects, the number of objects does not saturate the streaming processors. The CPU cost, however, is much higher as the CPU must take much longer to sort ten times the amount of objects – resulting in the higher frame time, overall.

Going from fifty thousand objects to five hundred thousand objects tells a similar story, in terms of performance characteristics. The amount of data that is involved in five hundred thousand objects is much higher than can be optimised by the graphics card driver – meaning that the whole buffer is guaranteed to not be in the graphics card cache, as the RTX 3080 has only 5 MB of L2 (TechPowerUp, no date), the largest of the cache available and instead at best being in the graphic card's VRAM. This explains the how, in this case, the instanced renderer using the input assembler has better performance over the Structured Buffer renderer, as the input assembler copies the world data to the streaming processors register, which is the fastest access point for data from a streaming processor, with VRAM being the second slowest, in front of the system RAM. That being the case, however, theoretically the input assembler should be faster in all cases as the access is so immediate.

The overhead of dispatching five hundred thousand draw calls for each object renders the no instancing render unusable for games at this point, with a frame time of approximately fifty-six milliseconds, which equates to under twenty frames per second. The benchmark for games is to target sixty frames per second, with demanding games instead targeting thirty frames per second. The instanced renderers both hit this target of thirty frames per second, marginally, with the Structured Buffer renderer hitting just under the frame time limit of 33.3 milliseconds for thirty frames per second.

Overall, however, the CPU updating each object's current LOD model is slower on the instanced renderers than the no instancing renderer, which is another highly unexpected result. The code that is ran on renderer is the exact same, with no conditional compiling taking place within any update code, as well as this, the renderer is single-threaded so there is no way that the CPU is doing any work in the background, in the context of the program. This result is the same across all object counts and widths too.

The Effect of Pixel Overdraw on Performance

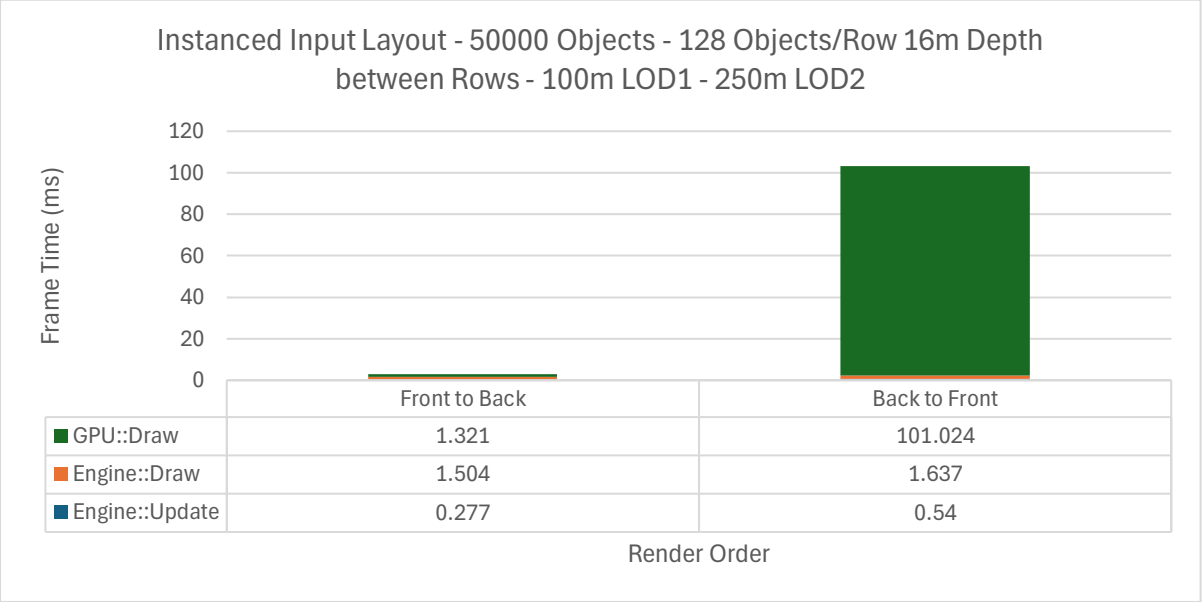


Figure 24 - Instanced Input Layout - 50000 Objects - 128 Objects/Row 16m Depth between Rows 100m LOD1 - 250m LOD2

To determine the effect of pixel overdraw on the performance of a render two tests were done. One being the best-case scenario for a renderer, all objects drawn in order from closest to furthest from the camera. This allows the graphics card to use the early Z test to discard a pixel before it is shaded by the pixel shader. The other is the worst-case scenario for a renderer, all objects being drawn from furthers away to closest from the camera – completely bypassing the early Z test as the depth of the pixel being rendered will almost always be in front of the pixel already shaded, creating a detrimental effect to graphic card performance.

Looking at the results directly, there is a near one hundred times performance cost to the worst-case scenario. As previously mentioned this is due to the inability to perform early Z test and discards on pixels, as in most cases the pixel currently being processed by the graphics card will be in front, in screen space, of the already rendered pixel – thus passing the Z test. This test was performed with fifty thousand objects, with one hundred and twenty-eight per row. This means there is at approximately three hundred and ninety objects per column, meaning that, in some cases a pixel may be rendered that many number of times – this is not accounting for any geometry that may get rendered on the same object behind the final pixel of that object too. With most of the scene being rendered to at any given point, which means a lot of pixels that get rewritten hundreds of times. In contrast, the front to back render may only write to some pixels at worst a few dozen times, as a large majority of the screen is rendered to early on, as objects closer to the camera are larger.

This test was inconclusive when using five hundred thousand objects, trying to render that many objects back to front resulted in a program that seemingly would never finish rendering the next frame, the frame time was beyond five minutes before the test was cancelled and reduced to fifty thousand objects. Testing also was not done with five thousand objects as this would not provide the same performance impact as that of with fifty thousand objects.

Bandwidth and Graphics Card Performance

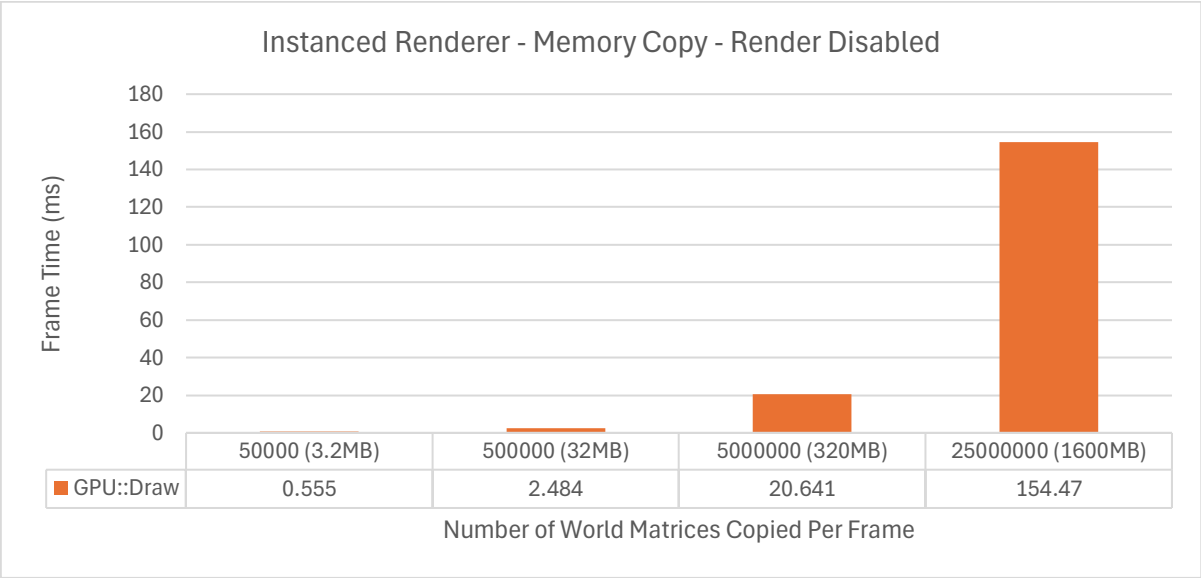


Figure 25 - Instanced Renderer - Memory Copy Benchmark - No Render

Bandwidth is how much data can be transferred between two places, in this case, to and from the CPU and graphics card. As determined previously, the PCIe connection between the motherboard and the graphics card is the bandwidth bottle neck in many cases as graphics cards have a far higher memory bandwidth than the PCIe connection bandwidth.

The RTX 3080 uses a PCIe Gen 4 x16 connector, which is matched by the testing machines motherboard. Using the mathematics calculated previously, the bandwidth per frame, assuming that the target frame rate is sixty frames per second, can be calculated. PCIe Gen 4 x16 connections have a transfer rate of 16 GT/s per lane. This translates to 16 Gb/s per lane which is 2 GB/s. When accounting for a target frame rate of sixty frames per second, this gives a per lane rate of approximately 33.3 MB, or 532.8 MB across all sixteen lanes of the PCIe Gen 4 x16 connector, per sixtieth of a second. The artefact, using the structured buffer copy as the benchmark, which the data in

Figure 25 uses, requires 64 bytes per object – the world matrix of each object. This gives a total throughput of 8.325 million world matrices, assuming no other data is transferred to the graphics card at any given point, however this will never be the case as, at the very least, the frame buffer will need to be copied from the graphics card to the CPU to be placed into the swap chain.

To run a fair test, the graphics card does not render anything during these captures and instead the CPU is copying the buffers over to the graphics card during these tests. The amount of system memory the testing machine has is not enough to saturate the PCIe Gen 4 x16 connector, however. This would require a machine that has over 32 GB free system memory, without resorting to a page file, which would drastically hinder the benchmark, and the machine used has a total system memory of 32 GB. To get around this, an enormous range of memory was used to determine any overheads that may become visible when copying substantial amounts of memory across, this is especially true for the twenty-five million object test, where there is 1.6 GB is copied across.

Starting with fifty thousand objects, the data being copied across is 3.2 MB. Based on the total bandwidth per second of a PCIe Gen 4 x16 being approximately 32 GB/s, the theoretical time is 0.1

milliseconds, however the process took 0.555 milliseconds on average, with 0.455 milliseconds of overhead and approximately five times slower.

Stepping up to five hundred thousand objects, the theoretical time for this is one millisecond, as the data is exactly ten times larger, going from 3.2 MB to 32 MB and 0.1 millisecond to one millisecond. The process took 2.484 milliseconds on average, leaving an approximate 1.5 millisecond overhead to the theoretical time, but only 2.5 times slower, versus the near five times slower of fifty thousand objects.

Multiplying the object count by ten again, going up to five million objects, the theoretical time is ten milliseconds, when copying 320 MB of data. It instead took 20.64 milliseconds on average, with an approximate ten millisecond overhead, with a further decreased multiplier of approximately two times slower.

Finally, going up to 2.5 million objects, five times that of five million objects, the theoretical time is fifty milliseconds – this time copying 1.6 GB of data. The actual result is an average time of 154.5 milliseconds, which has an overhead of approximately one hundred milliseconds and is instead three times slower than the theoretical time, an increase of that of five million objects.

Overall, there is an indeterminate overhead in copying data to the graphics card. With copies of 32 MB happening per frame, there is no major impact to frame time, however, with a copy of 320 MB happening every frame, a significant chunk of a thirty frame per second target is ate from that alone, sixty frames per second is not attainable whatsoever with 320 MB as that is 16.67 milliseconds, and 320 MB took an average of 20.6 milliseconds.

Conclusion

Overall, the artefact was a success, showing multiple ways of increasing a renderers performance, either through fundamental changes to a renderer, like with that of using instanced draw calls to effectively use the parallelisation present in graphics cards, to how sorting draw call in order of distance to screen can provide a substantial performance increase from utilising the graphic cards early Z test and discard for pixel shading.

Regarding the instancing renderers, they alone can accelerate the graphics card drawing from anywhere between three to ten times, with the biggest gains coming from when there are more objects being rendered per draw call.

When rendering from back to front, the performance impact was clearly shown to be potentially completely detrimental to a project – with fifty thousand objects being rendered backwords having an approximate impact of up to one hundred times.

On the other hand, the artefact was weak in demonstrating some topics, such as the effect of memory copying to the graphics card, where the results are not easy to pull a conclusion from. The most relevant data gathered from the test is that, at least, on the RTX 3080 used to perform the test, a copy per frame limited to about 50 MB has only a minor effect on the frame time. The only other conclusion is that the amount of overhead is potentially exponential, however there may well be far more at play in between the CPU and the graphics card, such as the graphics card driver. As well as this, the CPU benchmarks in the instanced renderers are too slow when sorting the draw calls, where the CPU takes anywhere between 1.5 to three times slower than the rendering portion.

Another failing of the artefact was correctly exploring the scaling of the number of objects or triangles overall. There was no notable scaling with rendering with the no instanced renderer between fifty thousand objects and five hundred thousand objects, when changing the width of the scene – increasing the triangle account massively each time. The same is true for both instanced renderers at five hundred thousand objects, there being no scaling present with a massively variable triangle count. However, there is some scaling present at fifty thousand objects when using both renderers. The scaling trends towards zero going from five thousand objects to fifty thousand objects with the instanced renderers. Based on this, increasing the number of vertices that must be processed by the vertex shader is not a cause for the lower performance of the renderers at high object counts, instead pointing to unknown factors taking up much of the render time.

As well as this, a random distribution of objects draw order could be used to determine what the average effect of object placement has on pixel overdraw, as the results can vary greatly based on how much of the screen is rendered to early on and is also close to the camera.

If the artefact were to be built again from scratch, research into CPU multithreading and general-purpose GPU (GPGPU) computing to accelerate processes within. The biggest speed up from this would be the CPU single threaded sort of the objects. Using either CPU multithreading or GPGPU parallelisation to calculate the correct order for these would potentially provide a substantial acceleration. As well as this, further thought into the architecture of the artefact would provide its own benefits, for example, memory usage is relatively high for a project this simple – whereby using the object count variable the memory usage of the project is an order of magnitude higher than is ever copied to the graphics card.

Future Work

As mentioned in the conclusion, there are several avenues for further research based on this paper. Much of the artefact's features do not take full advantage of modern graphic card features, such as general-purpose computing or multithreading CPU.

There are also more graphics specific optimisations that can be explored, such as culling objects that are not visible before they are even sent to the graphics card, reducing the number of invocations of the vertex shader, as the pixels will not get rendered anyway – since they are occluded or off screen. Culling on its own also has many avenues to explore, such as frustum culling, which culls anything that is outside of the view of the camera, as well as this, binary space partition-based visibility culling could also be explored, which would cull objects that are known to be completely occluded or not visible from an area.

Beyond improvements to the artefact itself, more tests could be done or an improved testing methodology used. How the data was recorded for each width and object count combination is bulky and difficult to draw conclusions from. Data must be looked at from across several different graphs to draw conclusions about the performance at large – such as analysing the raw performance difference between different object counts, a width count must be picked and then each test conducted with that width must be consulted. The same applies to looking at the performance impact of different widths, an object count must be picked and data collated from each graph of the same object count which increases in width.

Other work could be focus on different environments, emulating more 'realistic' scenes depicted in games instead of a field of trees. Such as a large and open environment with many types of trees, grass and other flora or dense cities packed full of junk or other miscellaneous objects.

Bibliography

Amiri, P. (2024) 'Review of Rendering Evolution of Games Engines in the 3D Era' *LUTPub*, 2024. Available at: https://urn.fi/URN:NBN:fi-fe2024070159958 (Accessed: 17/12/2024).

Wynters, E. (2011) 'Parallel Processing on Nvidia Graphics Processing Units using CUDA' *The Journal of Computing Sciences in Colleges*, 26(3), pp. 58-66. Available at: https://www.researchgate.net/profile/Debra-Major/publication/234803601_A_CS0_course_using_Scratch/links/55ee340d08ae0af8ee19f652/A-CS0-course-using-Scratch.pdf (Accessed: 06/01/2025).

Wloka, M. (2003) '"Batch, Batch, Batch:" What Does It Really Mean?' *Games Developers Conference*, 2003. Available at: https://www.nvidia.com/docs/IO/8228/BatchBatchBatch.pdf (Accessed: 07/01/2025).

Peddie, J. (2023) 'AMD's Unified Shader GPU History' *IEEE Computer Society – Chasing Pixels*. Available at: https://www.computer.org/publications/tech-news/chasing-pixels/amd-unified-shader-gpu-history (Accessed: 08/01/2025).

Szijártó, G. and Koloszár, J. (2003) 'Hardware accelerated rendering of foliage for real-time applications' *Proceedings of the 19th Spring Conference on Computer Graphics*, pp. 141-148. doi: https://doi.org/10.1145/984952.984976.

Khronos Group. (No date) 'Vertex Rendering – Instancing' *OpenGL Wiki*. Available at: https://www.khronos.org/opengl/wiki/Vertex_Rendering#Instancing (Accessed: 08/01/2025).

Cebenoyan, C. (2004) 'Chapter 28. Graphics Pipeline Performance' *GPU Gems*. Available at: https://developer.nvidia.com/gpugems/gpugems/part-v-performance-and-practicalities/chapter-28-graphics-pipeline-performance (Accessed: 14/01/2025).

Peddie, J. (2023) 'The History of the GPU – Steps to Invention' *Springer Nature*.

Glatter. (no date) 'Rise of 3dfx' *Vintage3D*. Available at: https://vintage3d.org/3dfx1.php (Accessed: 15/01/2025).

Owens, J. *et al*. (2008). GPU Computing. *Proceedings of the IEEE*, 96(5), pp. 879-899. doi: https://doi.org/10.1109/JPROC.2008.917757.

Pelzer, K (2004) 'Chapter 7. Rendering Countless Blades of Waving Grass' *GPU Gems*. Available at: https://developer.nvidia.com/gpugems/gpugems/part-i-natural-effects/chapter-7-rendering-countless-blades-waving-grass (Accessed: 22/01/2025).

Gerling, K. *et al*. (2013) The effects of graphical fidelity on player experience. Available at: https://hdl.handle.net/10779/lincoln.25166207.v2 (Accessed: 27/01/2025).

Karmaker, J. (2016) Vegetation Creation for Video Games. Available at: https://80.lv/articles/vegetation-creation-for-video-games/ (Accessed: 27/01/2025).

Nvidia. (2025) GeForce RTX 5090. Available at: https://www.nvidia.com/en-us/geforce/graphics-cards/50-series/rtx-5090/ (Accessed: 27/01/2025).

Glatter. (no date) 'NV1 (Edge 3D)' *Vintage3D*. Available at: https://vintage3d.org/nv1.php (Accessed: 19/02/2025).

Shimpi, A. (1999) ‘One Underrated Weakness: The OpenGL ICD’ *AnandTech*. Available at: https://www.anandtech.com/show/272/10 (Accessed: 19/02/2025).

FalconFly. (2004) ‘3dfx Reference Drivers’ *FalconFly Central*. Available at: http://falconfly.3dfx.pl/voodoo1.htm (Accessed: 19/02/2025).

Nvidia. (1999) ‘Microsoft® DirectX® 7: What’s New for Graphics’ *Technical Brief*. Available at: https://developer.download.nvidia.com/assets/gamedev/docs/Microsoft_DirectX_7.PDF (Accessed: 19/02/2025).

Engel, W. (no date) ‘Direct3D 7 IM Framework Programming’ *Gamedev Archive*. Available at: https://archive.gamedev.net/archive/reference/programming/features/d3d7im2/page2.html (Accessed: 19/02/2025).

Dietrich, S. (2001) ‘Dx8 Pixel Shaders’ *Games Developers Conference 2001*. Available at: https://developer.download.nvidia.com/assets/gamedev/docs/GDC2K1_DX8_Pixel_Shaders.pdf (Accessed: 20/02/2025).

Nvidia. (No date) ‘Microsoft® DirectX® 8: Raising the Ante for Realism in Graphics’ *Technical Brief*. Available at: https://www.evga.com/articles/images/11directx8.pdf (Accessed: 20/02/2025).

Ghorpade, J. *et al*. (2012) GPGPU Processing in CUDA Architecture. *Advanced Computing: an International Journal*, 2012. doi: https://doi.org/10.48550/arXiv.1202.4347.

Rejhon, M. (no date) *VGA Quake versus 3Dfx OpenGL Quake*. Available at: https://www.marky.ca/3d/quake/compare/ (Accessed: 20/02/2025).

White, S. *et al*. (2020) Using Shaders in Direct3D 9. Available at: https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-using-shaders-9 (Accessed: 20/02/2025).

Segal, M. and Akeley, K. (2004) The OpenGL® Graphics System: A Specification (Version 2.0 - October 22, 2004). Available at: https://registry.khronos.org/OpenGL/specs/gl/glspec20.pdf (Accessed: 20/02/2025).

Kirk, D. (no date) *GeForce3 Architecture Overview*. Available at: https://developer.download.nvidia.com/assets/gamedev/docs/GF3ArchitectureOverview.pdf (Accessed: 20/02/2025).

TechPowerUp. (no date) *Nvidia GeForce3*. Available at: https://www.techpowerup.com/gpu-specs/geforce3.c738 (Accessed: 20/02/2025).

TechPowerUp. (no date) *NVIDIA GeForce4 Ti 4200*. Available at: https://www.techpowerup.com/gpu-specs/geforce4-ti-4200.c2133 (Accessed: 20/02/2025).

TechPowerUp. (no date) *NVIDIA GeForce FX 5950 Ultra*. Available at: https://www.techpowerup.com/gpu-specs/geforce-fx-5950-ultra.c79 (Accessed: 20/02/2025).

Nvidia. (2006) ‘NVIDIA GeForce 8800 GPU Architecture Overview’ *Technical Brief*. Available at: https://www.nvidia.co.uk/content/PDF/Geforce_8800/GeForce_8800_GPU_Architecture_Technical_Brief.pdf (Accessed: 20/02/2025).

TechPowerUp. (no date) *NVIDIA GeForce 7950 GT*. Available at: https://www.techpowerup.com/gpu-specs/geforce-7950-gt.c183 (Accessed: 20/02/2025).

Microsoft. (2024) *ID3D10Device::DrawIndexedInstanced method (d3d10.h)*. Available at: https://learn.microsoft.com/en-us/windows/win32/api/d3d10/nf-d3d10-id3d10device-drawindexedinstanced (Accessed: 20/02/2025).

Carucci, F. (2005) 'Chapter 3. Inside Geometry Instancing' *GPU Gems 2*. Available at: https://developer.nvidia.com/gpugems/gpugems2/part-i-geometric-complexity/chapter-3-inside-geometry-instancing (Accessed: 20/02/2025).

Gosselin, D., Sander, P. and Mitchell, J. (2005) '6.5 Drawing a Crowd' *3D Engine Design*, pp. 505-517. Available at: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=4fd92a5726fca22fb92dedbde232d93fe19e1c34 (Accessed: 21/02/2025).

Fan, Z. *et al*. (2015) 'Simulation and rendering for millions of grass blades' *iD3 '15*, pp. 55-60. doi: https://doi.org/10.1145/2699276.2699283

Corbalán-Navarro, D. *et al*. (2021) 'Omega-Test: A Predictive Early-Z Culling to Improve the Graphics Pipeline Energy-Efficiency', *IEEE Transactions on Visualization and Computer Graphics*, 28(12), 2022, pp. 4375-4388. doi: https://doi.org/10.1109/TVCG.2021.3087863.

Hovland, R. (2008) 'Latency and Bandwidth Impact on GPU-systems' *Norwegian University of Science and Technology*. Available at: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=8456cd707e76842aafa396add87ccf7cff5ac0fe (Accessed: 21/02/2025).

Caulfield, B (2009) 'What's the Difference Between the CPU and the GPU?' Available at: https://blogs.nvidia.com/blog/whats-the-difference-between-a-cpu-and-a-gpu/ (Accessed: 21/02/2025).

Sander, V., Nehab, D., Barczak, J. (2007) 'Fast Triangle Reordering for Vertex Locality and Reduced Overdraw', *SIGGRAPH '07: SIGGRAPH 2007 papers*, 2007, pp. 89-99. doi: https://doi.org/10.1145/1275808.1276489.

Hey, H and Purgathofer, W. (2001) 'Occlusion Culling Methods' *Start of The Art Report*, Eurographics 2001). Available at: https://diglib.eg.org/server/api/core/bitstreams/44ecd645-7e00-4333-ad67-035d11c004b5/content (Accessed: 22/02/2025).

Valve. (no date) 'Visibility optimization' Available at: https://developer.valvesoftware.com/wiki/VIS_optimization (Accessed: 22/02/2025).

Valve. (no date) 'Visleaf' Available at: https://developer.valvesoftware.com/wiki/Visleaf (Accessed: 22/02/2025).

Tuxedo Labs. (2022) 'Teardown' [Video Game]. Tuxedo Labs.

Dublish, S., Nagarajan, V. and Topham, N. (2017) 'Evaluating and mitigating bandwidth bottlenecks across the memory hierarchy in GPUs' *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2017, pp. 239-248. doi: https://doi.org/10.1109/ISPASS.2017.7975295.

PCISIG. (no date) 'PCI Express® 3.0 Frequently Asked Questions' [Archive] Available at: https://web.archive.org/web/20140201172536/http://www.pcisig.com/news_room/faqs/pcie3.0_faq/#EQ2 (Accessed: 23/02/2025).

Glawion, A. (2022) ‘Guide to PCIe Lanes: How many do you need for your workload?’ *CGDirector*. Available at: https://www.cgdirector.com/guide-to-pcie-lanes/ (Accessed: 23/02/2025).

Jones, B. (2025) ‘How Fast Is PCIe 5.0?’ *TechReviewer*. Available at: https://www.techreviewer.com/tech-answers/how-fast-is-pcie-5/ (Accessed: 23/02/2025).

TechPowerUp. (no date) *NVIDIA GeForce RTX 5060*. Available at: https://www.techpowerup.com/gpu-specs/geforce-rtx-5060.c4219 (Accessed: 23/02/2025).

Slyusarev, V. *et al*. (2022) *Optick: C++ Profiler For Games* [Software] Available at: https://github.com/bombomby/optick

TechPowerUp. (no date) *NVIDIA GeForce RTX 3080*. Available at: https://www.techpowerup.com/gpu-specs/geforce-rtx-3080.c3621 (Accessed: 24/02/2025).

Nvidia. (no date) *CUDA C++ Programming Guide – SIMT Architecture*. Available at: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#simt-architecture (Accessed: 25/02/2025).