EÖTVÖS LORÁND TUDOMÁNYEGYETEM

Informatikai Kar

Komputeralgebra Tanszék

Latency reduction in real time applications

Ligeti Péter

egyetemi docens

Végvári Zalán

Programtervező informatikus

Budapest, 2021

Table of contents

Introduction

The goal of this project was to bring together two formerly known and used techniques in the area of game development. The results are not restricted to such applications, but this is the most relevant area where they fit very well.
This goal was fully achieved. The code provided as an attachment to this document contains a Vulkan API based rendering engine that implements a frame graph based approach for the organization of work; it contains an abstract implementation of a graph based algorithms required to process profiling information, which is necessary for reducing latency; and the code also integrates these two areas by specializing the abstract latency interface into the used frame graph model.
In this chapter I would like to introduce the reader to some of the basic concepts in the area of latency and frame graphs.

So what exactly is this latency?

In the context of this project latency refers to the time between pressing a button and seeing the result on the screen. In the example application (introduced later) there is a so-called “latency flash indicator” which is a white square that appears for 0.3 seconds after pressing the left mouse button. This will be useful for physical measurements, but it’s also a good visual example that helps understand what’s exactly measured.
The input signal has to pass through many things in the computer, which my software has no influence on:

1) The mouse/input device itself, it takes time to notify the computer
2) The internal hardware of the computer
3) Operating system
4) The software
5) The resulting image might need to go through the operating system again
6) The monitor (it takes time light up pixels and it’s only refreshed at fixed rate, not immediately on most devices)

The latency can be reduced at many of these stages, which are orthogonal to the topic of this project. Using a wired mouse will have lower latency than a

bluetooth one; It's possible to use a so-called raw input API interface to bypass the operating system's input queue; softwares under Windows platform are able to obtain direct access to the display (this is known as the "fullscreen mode"); it's possible to buy a monitor with G-sync support to get rid of the fixed refresh rate and present the image when it arrives; some monitors light up pixels in significantly lower time than others. All of these latency optimizations are independent of the software latency reduction.

Onwards from this point latency will only consider the software latency component, except for the measurements with the LDAT device, where it will be explicitly pointed out. This eliminates a lot of noise produced by the rest of the components, because they are not necessarily a constant value.

Software latency

This sounds very straightforward and simple at first glance. If it takes the application 16ms to produce an image (a frame), then the latency should be 16ms, right? Well, no. The computer has several independent computational units: several CPU cores and the GPU. Games usually try to use up as much of the available computational resources as possible, but a lot of work depends on previous work. In simplified terms, GPU or *device* work depends on CPU or *host* work. In order to maximize CPU and GPU occupancy, several frames are processed at the same time. While the GPU processes the *n*th frame, the CPU could already be processing the *n+1*th frame.

Ok, so is the latency CPU work time + GPU work time? Still no. Such resources usually act as a pipeline. CPU generates GPU work at its own rate and the GPU processes them at a different rate. The GPU will get left behind if it's slower than the CPU, thus accumulating undone work. There is always a limit to this, but it can be several frames of work. The accumulated work directly increases latency. This is still not all, there are inter-frame sources of latency that will be described later. This is the area where frame graphs are really useful. Without a frame graph, it's really difficult to assess the inter-frame latency and even more difficult to eliminate it.

Frame graphs

The concept of a frame graph is a model that improves parallel executions and transient resource usage in real time applications. The main idea is that workload is grouped into packages, which are executed serially on their own and asynchronously to each other. The only dependency between separate work packages is what the programmer has defined explicitly when designing the application. Two work packages can be executed simultaneously and in any order if they don't have a dependency. In case of a dependency, the dependent work must be fully completed before starting the dependee.

Based on the previous description we can already see how this forms a graph. The work packages are the nodes in the graph and the dependencies are the edges. It should be clear that this is a directed graph and assuming a valid program design, it's also a flow graph. The very first work package(s) cannot depend on anything. There always has to exist an ordering of these packages that allows serial execution while satisfying all dependencies. So a topological order of these nodes is granted. Since a game produces many images, almost exactly the same way, we can expect that this graph has a period to it, where each segment is either identical or similar to the other ones. The example application presented in this document uses identical segments of the framegraph.

When using a frame graph, it's possible to accurately measure many performance metrics, which will be required for fine-grained latency reduction. It also allows smart scheduling of the workload in the available threads or device queues.

Does latency really matter?

Previously the most important performance metrics used to be the frame rate of an application, referred to as *FPS* (which stands for frames per second). It's also quite clear why this is the case. A game with low FPS will be frustrating to watch and difficult to play. Lower frame rate will make the game look less dynamic and less continuous.

Latency on the other hand is less obvious. Its effects are more hidden, but that doesn't mean that they are negligible. It affects two main things: user

experience and performance(Spjut et al.). If a player encounters a game with high latency, it's very easy to mistake it for low FPS. It will feel like a game with low fps. Just like low fps, high latency causes frustration, because the player won't see a quick response to his actions. A game development company can invest a lot of money to achieve a high frame rate, but if the latency is not optimized properly, the overall user experience could stay just as bad as it would have been with low frame rate.

The effect on the player's performance is even more subtle(Banatt). **Most** users can't even tell the difference if the latency increases or decreases, but the statistics can(Liu et al.). When it comes to competitive shooter games, even as little as 5ms of latency can result in a measurable difference in performance. With that said, latency is not equally important in all situations.

Latency in an application that runs at a comfortable 60 fps (standard refresh frequency of many monitors) can vary between 60-150ms when considering the full end to end latency. Software latency can be between 20-110ms at least on hardware, which I have access to.

My contributions

My primary contribution in the area of latency reduction is a framegraph based interface and a graph based algorithm behind it. The interface allows framegraph based applications to provide the required information to the algorithm for easy implementation. The algorithm itself is a result of research and experimentation. For the most part I have followed the methodology presented by Activision and the work of NVIDIA, which defined the end goals of the algorithm, but I have also experimented with different approaches. Both of these sources proved correct about the approach that they have implemented.

The novelty of this project is the possibility of reducing latency inside a frame by reordering some of the nodes or delaying them relative to each other if possible. This aspect is tested and the results show success, although this is not the focus in any of the tests. After reading my approaches in detail, it should be obvious that the latency is reduced by performing actions in different order. The tests focus on the overall correctness of the algorithm, because the exact algorithm is also something that I have implemented by myself, the resources

that I used only provided ideas and concepts. I haven’t found any copy-pastable code that would be backed by any other papers.

In addition to the above, I have implemented a Vulkan API based rendering engine that follows a framegraph based design. This work is not relevant in latency reduction directly, but it creates a configurable framework, where the latency reduction can be tested with convenience.

Defining research goals

In this chapter I would like to give an insight into how latency looks in a graphical application and what we can expect, how much reduction is possible, what is the cost of reduction. In addition to these, I would also like to explain what's important when developing a video game, regarding performance metrics at least.

A few words about the value of specific performance metrics

I have already described that frame rate and latency are important. I would like to augment this with the fact that frame rate directly affects software latency. Just considering that a game produces images at a specific rate, already creates latency. If a player pushes a button, it will have to wait for **at least** the completion of the next image. Of course in practice it will take more time than that. Considering the above, it should be evident that it's a really bad idea to design a latency reduction system that decreases the practical frame rate of an application, although it's a different question if the game is able to run at a higher frame rate, than the monitor's refresh rate. More on this later.

One aspect that hasn't been mentioned before, which this project puts a lot of emphasis on, is the number of skipped frames. What if the game is running in vsync mode at 59 FPS on a monitor that has a refresh rate of 60 hz? In other words, images are only presented to the screen at the start of every ~16.7ms interval. Depending on the circumstances and options, it's possible that the game will skip every 2nd frame and will effectively run at 30 FPS. This is the case with VR headsets for example. In this device low latency is crucial, because high latency can induce feelings of sickness. VR based applications must maintain a very high frame rate with very low latency. Depending on the specific device, they might run at 120,90,72 FPS at normal rate, but this rate will be locked to the half if the game doesn't deliver frames at the correct rate.

If latency reduction is done incorrectly, it's possible that it pushes a frame's completion over the 'deadline', which could result in the exact same effect as having insufficient frame rate: the frame rate will get locked to the half of the original value defined by the device. In order to deal with this issue, I have

implemented a separate mode of operation of the latency reduction system that allows the game developers to make a conscious decision about taking the risk of skipped frames to further reduce latency. Unfortunately this is a trade-off. **Minimizing latency will result in a higher chance of skipping frames**.

Let's see what an unoptimized frame looks like

I would like to show what a performance capture looks like and explain how to interpret it. The captures that I will present in this document are made with a custom tool.

The performance capture stores the timing information of each work node from the frame graph which belongs to a specific frame (*the target frame*) or has the option to parallelly run with the target frame. The capture stores when the nodes *start*-, when they *finish* execution and also when they are *available*. In addition to these, the capture also shows a logical and a physical grouping of nodes. All nodes belong to one logical group, which is defined by the programmer and they are executed on one specific resource in the computer.

The logical grouping in my case:

- Simulation stage (CPU)
- Before draw stage (CPU)
- Record stage (CPU)
- Execution stage (CPU)
- Device work (GPU)
- Readback stage (CPU)

The physical groups are all the threads used by the application and the device queues. The thread used by the execution stage is designated as a separate resource in the capture. For simplicity it's assumed that all threads run on separate cores. More on this in the future work section at the end of the document.

The simulation stage processes all simulation work done by the application, which is required for proper execution, but not required for rendering. There is no access to the GPU at this stage. The before draw stage still has no access to the GPU, but it's specific to actions required for rendering. A server side application would not need this, just the simulation. The record stage uses information created by the before draw stage and prepares it for the GPU. The

reason for this separation is that the record stage could require more strict synchronization due to access to the GPU. A before draw stage can process work that doesn't require as much synchronization which allows it to be better parallelized. The execution stage is what actually dispatches the prepared GPU. The separation is required here, because the record stage can easily become the bottleneck and dispatching work could be computationally expensive depending on the application. Immediate dispatching of GPU work is not necessary so this separation can increase frame rate. The device work stage should be clear, it's where the GPU does the dispatched work. The readback stage allows the CPU to obtain information from the GPU. Since the GPU lags behind the CPU, so does this stage. It's intended to be called, when all device work is done, so no extra synchronization is required to access device resources.

Figure 1: A capture from the example application without latency reduction. The two 'delay' labels show gaps of time, where none of the work required for that target

frame is being done. Instead all resources work on queued up work. The latency indicates the time that passes between the input sampling step and the presentation. This is the full software latency. Nodes that belong to the target frame are highlighted and have '(0)' written after their names. The time information on the bottom of each node displays: *availability*, *start*, *finish* times in milliseconds relative to the earliest appearance of the target frame.

The capture in *figure 1* shows a frame from a scene with GPU *bottleneck*. This simply means that the GPU's performance is the weakest compared to the present workload and as a result it defines the framerate of the application. These cases are the most interesting for this document, because device work will accumulate in these cases.

Basic idea behind latency reduction

The goal is to bring the *input sampling* step and the frame's full completion closer together (readback stage doesn't matter in this case, just the device work). The input sampling step is where the game reads the operating system's input queue and processes all input that has been created since the previous input sampling step.

Looking at the capture in *figure 1*, we could ask why the execution of the target frame even begins as soon as it does. The game doesn't win anything, the frame rate won't increase and the latency is high. So all we need to do is delay the start of the frame, or more precisely, the input sampling step(Activision) (NVIDIA). The action that we need to take is really that simple, although figuring out how much to wait is not so much.

Let's see what the previous frame should have been like:

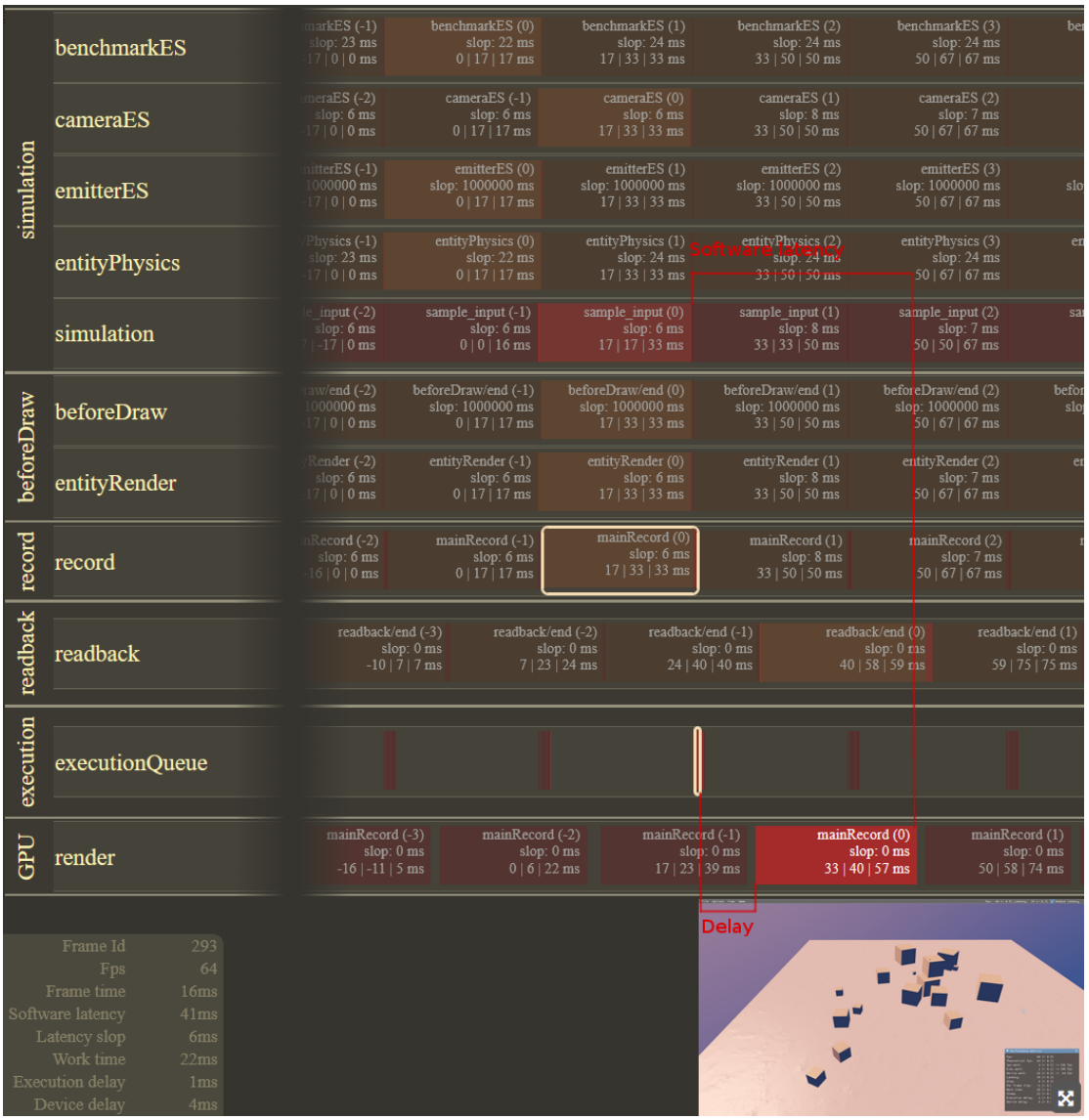


Figure 2: A capture from an optimized scene. The latency reduction is done inside of the *sample_input* node, so the software latency should be counted from somewhere at the end of the node.

This capture in *figure 2* looks significantly better, and it also feels better when using the software. But there is still some delay present. Can we really expect to get rid of it entirely? Well, the short answer is no, but let's see this in a bit more detail.

What can we expect?

As it turns out, the presence of some delay is necessary, depending on how consistently the game runs. In fact, it plays two roles in the presented solution, which makes delay a requirement for proper operation:

The first role is to contain the variance in frame rate. As mentioned before, latency reduction should not decrease frame rate and it should not cause skipped frames. If the latency reduction doesn't delay the completion of any frames, then it satisfies both conditions. If a frame starts at the latest possible time to not delay the frame (calculated using extrapolated data) and the frame turns out longer than expected, then the completion would be delayed. If the time required to complete a frame can be modelled using a normal distribution with a known expected value and standard deviation, we can take an educated risk when deciding how much delay we want to leave as headroom. This will be described in more detail later.

The second role is to recover or adapt to drastic changes in frame dynamics. To understand these, let's consider a situation, where we had to delay every frame by 8ms, because the GPU took 8ms longer to complete than the CPU. Due to some change in the game (a complicated object disappeared from the screen), the GPU is no longer slower. The latency reduction system had no way of anticipating this sudden change so it is expected to apply the same delay as before, which will now delay the frame's completion. The problem is that now the latency reduction defines the completion time. In this case there is exactly 0 delay (which could be measured to be around ±1ms). In this case it will look like latency reduction with just the perfect timing. Compared to previous statistics, the frame won't take longer. It will take longer than it could have, but that's not obvious. If the CPU takes 8ms, and GPU load drops from 16ms to 8ms, then the frame rate should increase from 62.5 to 125 without using latency reduction. Using the latency reduction, that targets exactly 0 delay, it's possible that the frame rate will remain 62.5. In worse cases, if the measurements are not accurate (they **are not** accurate), it's also possible that the algorithm always registers more possible latency reduction and it keeps increasing the delay and thus reducing the framerate, until some specified limit is reached and it's not allowed to wait longer. This is the most dangerous scenario when trying to reduce latency. Positive feedback loops should be avoided. A very important part of the algorithm is that **it has to use latency reduction invariant values**. This will be described in more detail later.

Considering these limitations, the possible latency reduction is limited by the application's consistency. The good thing is that it's possible to manipulate

accumulated delay quite well using the solution presented in this document. The goal is not to get rid of all delay, this software aims to give the programmers an option to make a conscious decision on how much risk to take and how much delay to keep. In case of consistent applications, it's possible to leave low delay and in inconsistent games with dynamic content, could opt for higher *latency pools*. It's also possible to decide dynamically based on performance statistics.

Inter-frame latency and frame graphs

The previous section only considered the latency originating from delay between stages in the rendering pipeline. I have designed the example application in a way that this aspect is easy to take advantage of. This is where frame graphs are very important, because this would be difficult without them.

The input information is not required for all calculations that take place in a frame. The most interesting calculation group is physics. This is what I have used in the demo application to illustrate how this works. I have picked it, because it's very common in many real time applications including video games and real time simulations. Physics calculations can be quite demanding, so they have a significant contribution to latency. Physics calculations do use input information directly or indirectly in most cases. For example, in a car simulation the care has to physically move, when the user tries to accelerate. On the other hand, it's not required to use the latest input in physics. Fresh input is usually the most important in the user interface (camera rotation, cursor movement) and game logic systems (shooting in combat games). It's not necessarily a problem if the physics lags behind the input by one frame. In cases like this, it's possible to only sample the user input after the physics calculations are completed. This way all other systems will receive more up-to-date input, which will decrease latency.

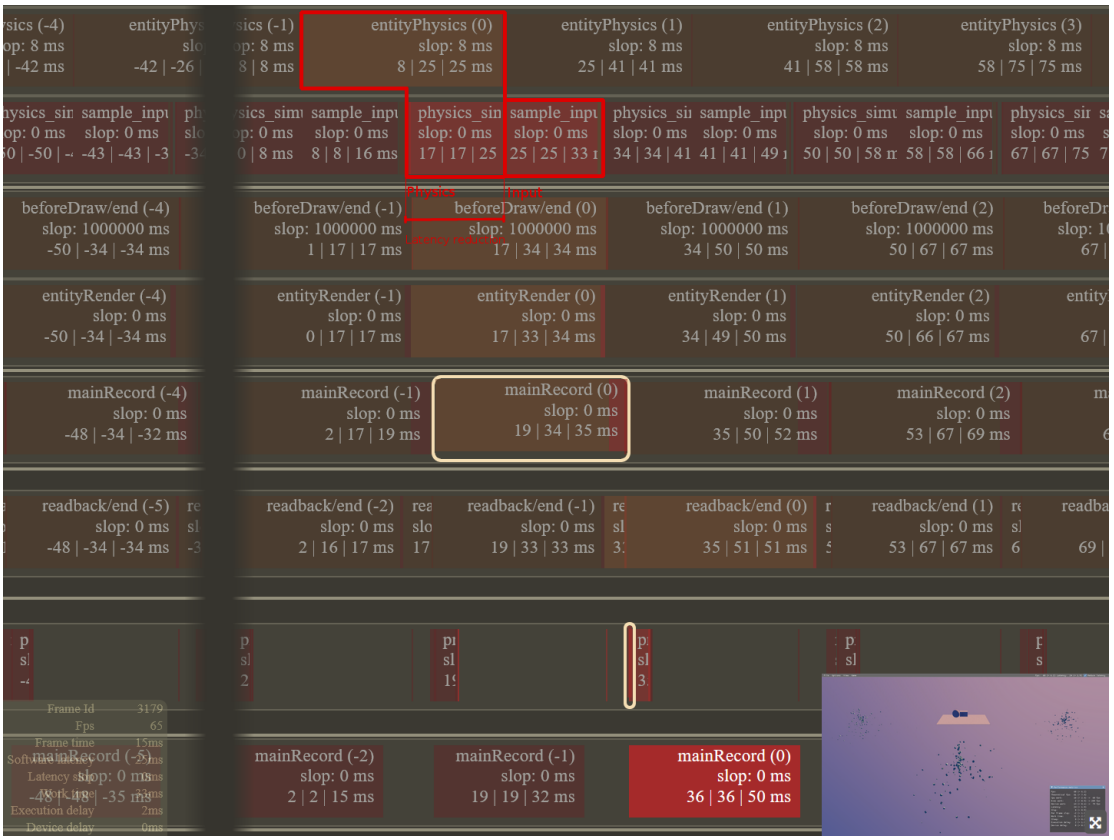


Figure 3: Physics simulation is completed before input sampling in a physics-heavy scene. The latency is reduced by the duration of the physics simulation compared to the case, where these operations are done in different order.

Figure 3 is a very good example of inter-frame latency reduction. A frame graph usually has a resource management system, which keeps track of which resources are provided and consumed by what nodes. In this case, the input data is a resource that needs to be managed by the frame graph's resource system. Only consumers of this data need to be dependent nodes and it's enough to complete the input sampling by the time the first dependent node would start. Since physics is not a dependent node, it's possible to perform before sampling the input.

The solution presented in this document aims to take advantage of such inter-frame latency reduction opportunities. There is no additional danger in reducing inter-frame latency, if we already have a positive delay or *latency pool*. The latency pool is sufficient to absorb all measurement errors and deal with inconsistencies on its own (assuming it's large enough). In the case of inter-frame latency, the scheduling of work can be an issue. The latency reduction system doesn't have an influence on which threads do the work. If physics is done on the same thread as the input sampling, then this is no longer

possible. In such cases, it’s possible to identify such cases using the presented algorithm. Although this was not implemented in this project, it would be a possible and reasonable expansion to warn the user or make a suggestion about cases, where inter-frame latency could be reduced, but it’s not possible due to the way the workload is distributed among threads. The algorithm is only expected to take advantage of inter-frame latency reduction opportunities that are possible in the existing circumstances. One specific issue that it needs to consider is if there is a significant amount of work on the same thread as the input sampling (after the input sampling). In this case, the input sampling would be required to be performed possibly sooner than the results are actually needed, because the used thread is required for other workloads.

A brief overview of the literature and background of the problem area

In this chapter I would like to show some general ideas for latency reduction and some specific approaches for both latency reduction and frame graphs.

Latency reduction so far

Most games and other applications in the past used to rely on, and most of them still do, a synchronization between the CPU and GPU stages with a specific offset. To explain this better, I'll show two example implementations:

Serial execution: The idea is very simple here, execute the CPU, then the GPU. There is no parallelism, the CPU will wait for the previous GPU work to complete before continuing. With this approach the delay is 0, but the frame rate is limited to *1/(CPU work + GPU work)*, where the work time is expressed in *seconds*, and the result is interpreted in *frames per second*. There is no equivalent configuration in the latency reduction approach presented in this document. This solution is inferior to all other approaches that I have encountered so far, but it is very simple to implement.

Double buffering: In this approach GPU and CPU are allowed to work parallelly, but under some restrictions. They don't behave as stages in a pipeline, unlike the example application presented in this document. This approach keeps most of the simplicity of the previous solution, without the frame rate limit. In this case, the frame rate limit is *1/(max(CPU work, GPU work))*. Note that this model treats CPU work as a single serial unit of work, but that's not necessarily the case. It is possible to achieve higher frame rate in more complex models, where the CPU work can further be distributed into more parallel stages. Double buffering will provide a low latency, but it could be lower. In GPU limited cases, the delay will be *GPU work - CPU work*. In CPU limited cases, there is no delay.

It should be clear why we need something better than the serial execution, but it's less obvious why we would want to improve on the simple double buffering technique. In well balanced applications the latency is very low for double buffering. The latency reduction technique presented here can provide very little improvement over this. One problem with this approach is related to the

presentation engine. It is present when **vsync** is used by the application. The purpose of the presentation engine is to manage presentable images, which can be displayed on the screen. It is a black box. The application can ask for an image to render to, and it can give the complete image back, which will then be presented to the screen. The presentation engine deals with things like the display's refresh rate. In case the application's potential frame rate is higher than the display's resolution, then the application will have to wait when trying to access a new image. This is fine so far, the problem is that the application will wait after sampling the OS input queue, which can add a significant amount of latency. This waiting happens when the application tries to acquire a new image. If we try to do that before input sampling, then we might limit frame rate in some other cases. Double buffering with vsync will add latency to the end of the frame as well, because the execution is synchronized with the display's refresh rate, but instead of matching the frame completion to the when the displayed is refreshed (more accurately the vblank), it matches a specific point of the application's CPU stage to the time an image is released, which is simple but not really useful for anything.

In some cases it is necessary to add more stages to the application's pipeline for independent reasons (for example the execution stage is a common practise), which can increase latency. In addition the double buffering technique doesn't deal with inter-frame latency at all. Although this technique can be a really good choice in many cases.

I would like to explain the previous approaches a bit more generally. Let's assume that the application follows a pipeline architecture (like the demo application), which means that there is a series of stages (eg. simulation, record, GPU), where the next stage is dependent on the previous stage's results. The stages are able to run parallelly, and work packages can be accumulated between them. The issue is when the first stage is faster to produce work, than one of the later stages to consume it. Such cases lead to accumulation of work. The general naive approach to deal with this is to synchronize the first stages with the last stage and optionally synchronize any stage with the next one. The synchronization has 3 attributes:

1) A point in the source stage's execution (within a frame)
2) A point in the destination stage's execution (within a frame)

3) A frame offset

The destination stage will stop at the specified point and wait until the source stage reaches the corresponding specified point in the execution of the earlier frame, specified by the offset.

In case of the *serial execution*, we would specify the beginning of the CPU stage, with the end of the GPU stage with a frame offset of 1. In case of the *double buffering* approach, we would synchronize the image acquisition within the CPU stage with the end of the GPU stage using a frame offset of 2. The problem with this kind of synchronization is the rigidity. The synchronization cannot be too tight, because it could decrease the frame rate in some cases. For example double buffering with frame offset of 1 would only allow some of the work to be done parallelly. In some cases, it might be the optimal, but in others, it would limit the frame rate. There is no way for the application to dynamically adopt. On the other hand, if the synchronization is too loose, it will add a high base latency in cases, where the later stages take longer to execute (the application is GPU limited).

I think it's worth mentioning a specific technology that helps with such cases. Depending on the application it can be adjusted to dynamic load. I'm talking about the *latency waitable objects* in DX12. This allows the application to allow a high frame offset for the synchronization, but it can be dynamically adjusted to lower values. It only allows adjusting the frame offset by entire frames, so it's a bit coarse grained. The destination stage (CPU) synchronization point can be selected, but the source stage (GPU) synchronization point is fixed at the presentation of the image, which is usually the last thing that happens on the GPU in a frame. The latency waitable objects don't provide any interface for deciding what frame offset to use. The application needs to handle that manually. Vsync cases still have the same issue as described for double buffering.

Latency reduction in Call of Duty

Call of Duty is a competitive first person shooter game, which means that latency is especially important here. The developers of this game did a great and very important work related to latency reduction(Activision). They describe the concept of '*slop*', which is similar to what I have described so far as 'delay'

previously. I will be using the concept of ‘slop’ with a detailed explanation in later chapters. They have also dealt with the display’s refresh rate and their solution is able to synchronize the frame’s completion with the vertical blanks (that’s when a new image is presented to the display when using vsync).

Their paper describes the general concept very well. The most important part of their paper (for this document at least) is the formula that decides how much the input sampling step has to be delayed:

waitUntil = intendedFlip - workForecast - targetSlop[(Activision, page 125)]

The *intendedFlip* refers to the vblank, when the image is planned to be presented to the display. The *workForecast* is the expected time that is required to complete the frame starting from the input sampling. The *targetSlop* is what I have been referring to as *latency pool* so far. It’s basically extra latency that we decide to add to each frame in order to avoid skipping vblanks or delaying frames in general in case the *workForecast* was lower than the actual work.

This formula is very simple, but the application needs to acquire these values at runtime, which is not as trivial. In their paper, workForecast is obtained by measuring previous work and applying an exponential average with an adjustable weight. I have applied the same approach in my solution as well. The targetSlop is a parameter, so we need both the intendedFlip time and the previous work time to calculate workForecast. The process of obtaining these values is not described in their paper.

My contribution is a frame graph based framework that allows us to obtain these values. The intendedFlip can be defined in three different way, which will be the different modes of operation:

- Unlimited mode
- Limited frame rate mode
- Vsync mode

The value of intendedFlip has to be calculated differently in each of these cases, with different goals in mind. The work time is also not trivial to calculate in a highly parallel application. It’s crucial to make all measurements independent of the latency reduction, which are used to calculate the *waitUntil* value.

NVIDIA Reflex API

NVIDIA has done a very similar work to this project[(NVIDIA)]. Reflex is an API created by NVIDIA, that is intended to be integrated into video games in order to provide latency reduction. Their approach is a bit different from mine, because one of their goals was easy integration. The Reflex API provides a higher level solution than the solution presented in this document. They rely on special driver support for their feature which provides their API with detailed information about GPU work, which means games don't have to deal with manual measurements on GPU, but in return, it only runs with NVIDIA video cards. The CPU side measurements are done with a few simplified markers, which makes it easy to integrate and it's also a very generic approach in terms of what application architectures it supports, but it can't take advantage of inter-frame latency reduction opportunities that are possible when using a frame graph.

Frame graph in Frostbite

Frostbite is a game engine created by Electronic Arts. They have implemented a frame graph based architecture in their game engine. Their paper[(Frostbite / Electronic Arts)] in this field is the most important resource that I have found. The frame graph that I have implemented in this project doesn't exactly follow the design of Frostbite, after all it's not designed for the same scale as Frostbite. In most cases, I have just followed the path of least resistance when designing my implementation. The latency reduction system should work in other frame graph based applications and just this specific implementation, so it wasn't important to have the same design. The concepts used when designing my implementation of the frame graph were based on Frostbite's design.

Detailing the elaboration of the topic

Sample application

I have mentioned the sample applications many times before, but I haven't introduced it yet. This application is designed to be ideal for performance measurements. It's a 3D simulation with physics support, where simple objects can be placed on the scene. The camera moves freely, without any character attached to it, to avoid interaction between the camera and the game logic. This way, measurements are repeatable with different camera movements. That's important, because some features have higher performance costs depending on what the camera sees.

I have created simplified and therefore more controllable features that represent different workloads in a real game. This is true both on the simulation and the rendering side.

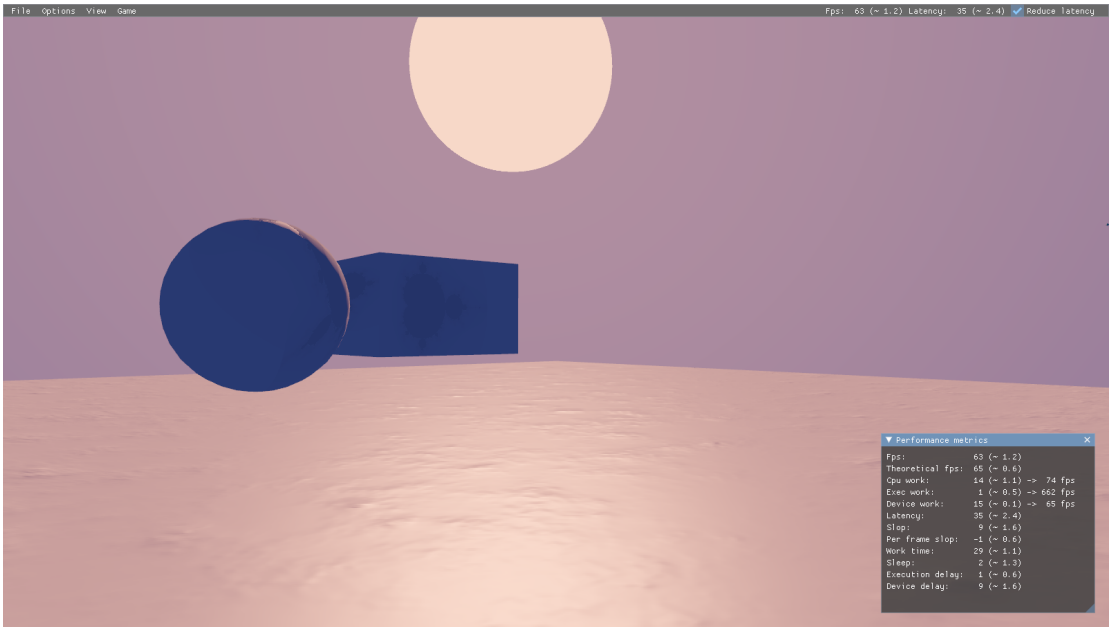


Figure 4: Static objects that have very low simulation cost, but they are expensive to render on the GPU side. This cost depends on how many pixels they occupy. In a game, this could be a particle heavy fancy effect, in this case there is a mandelbrot on these objects, that's configured to decrease frame rate to 60 fps, when looking from ~2 meters.

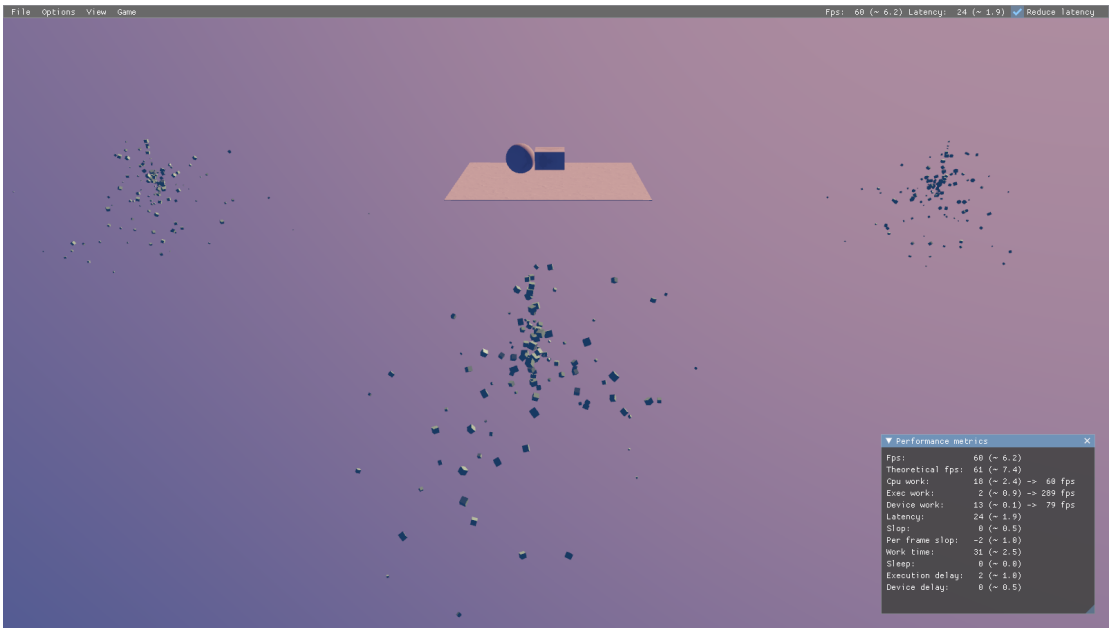


Figure 5: There are three cube emitters, which represent a very high simulation cost with moderate CPU-side render cost. The physics simulation cost and the amount of dynamic objects is configured to limit frame rate to about 70 with a realistic amount of variance. This cost is independent of the camera. The CPU-side render cost adds a dynamic cost to the CPU, which doesn't limit frame rate, but affects latency just enough. This work is done under the *mainRecord* markers in the captures. The amount of this work depends on how many objects are visible on the screen. Distance doesn't matter.

CPU load: Workload can be added to the simulation by increasing physics simulation quality; by adding more dynamic objects to the scene; or by causing more collisions between the existing objects. There is also a configurable workload simulation tool, that performs a so-called *busy sleep* in one of a few places in the applications pipeline. The busy sleep means an empty loop that runs for a specific time duration (instead of number iterations). This way the waiting thread cannot be used to do other work. This was important, because this tool is used to adjust variance to the desired value. The physics simulations can cause a lot of variance, more than what's necessary, and most importantly more than realistic. A real video game performs a lot of operations that have a lot more stable and predictable cost, than physics. I have picked the physics workload, because it's a very common feature in video games or simulation alike and it provides a great opportunity to reduce inter-frame latency. More on this in the next section. Rendering objects adds an overhead to the record stage of the application. The application has to generate commands for the GPU that will cause it to draw these objects with the correct parameters and these

commands need to be dispatched to the GPU. The dynamic objects are *culled* based on view. A bounding sphere is tested against the camera's frustum and draw commands are only issued if the bounding sphere intersects with the frustum. This kind of workload is very common in features of video games or any 3D rendering application for that matter, and it can be a big challenge if we want to make predictions on workload. The camera can turn around relatively quickly, which could mean a sudden increase or a sudden decrease in workload. I have created a benchmark just to test the latency reduction systems capabilities of adjusting to dynamic changes of increasing magnitude.

GPU load: The application features two sources of GPU workload: a fully configurable manual workload, which is similar to the CPU version and the objects with a mandelbrot. The manual workload is also a mandelbrot, that's drawn behind the skybox. It represents const cost in the GPU pipeline. In a real game, this would be equivalent to having a deferred resolve shader or a postfx or UI draw, etc. These are all relatively constant and the number of visible objects or their type doesn't matter. Rendering the skybox or normal objects without mandelbrot takes a very little time. As described in *figure 4*, 'heavy' objects have a view-dependent cost. This is important for the same reason as the CPU side view-dependent effects, described in the previous paragraph.

Physics for inter-frame latency

As mentioned before, physics simulation is a perfect candidate for showing how inter-frame latency can be reduced. Not all features of a game are equally sensitive to latency. Let's assume that the physics simulation takes 4ms. This would mean that if the input is sampled right before the physics simulation, that would add 4ms to the total latency compared to sampling it right after the physics simulation. Although, sampling right after the physics simulation would add an entire frame of latency to the physics simulation. When running at the refresh rate of a normal display, the game would run at 60 fps, which means ~16.7ms of frame time. So moving the input sampling after the physics would result in the following:

- -4ms latency to the user input, cursor, camera movement, etc.
- +12.7 latency to the physics simulation

In many applications this is very-well worth it, especially in competitive first person shooter games. The cursor's latency highly impacts how quickly and accurately a player can aim(Spjut et al.). The physics latency only means how quickly the user input affects the physical simulation. This could mean the character's movement, but that is a lot less important than the cursor.

Measurement of latency

I would like to introduce the techniques used to measure latency in the test application and an extra technique that was the earlier physical input-to-display measurement technique.

Markers in the program: This technique involves adding a marker to when the input is sampled and another when the frame is completed. It's very straightforward, but also error prone and ignores several factors. The good thing is that it only ignores irrelevant sources of latency with a single exception. Considering that the presentation engine that presents the image on the screen is a black box, there is no general way of knowing when the image is actually presented. This latency source is important for the vsync mode only. Other ignored latency sources are the latency of hardwares, the operating system or any other independent middle software, like drivers. These sources should be ignored when trying to optimize for software latency, because they can produce noise in the measurements. The software latency in the application is primarily measured using markers. It's only compared to LDAT in one specific test, that is meant to show the correlation between the different techniques of measurement. The most important reason for choosing this technique is that it can measure latency per-frame. No commonly used technique besides this would be capable of measuring latency with the same granularity, which is very important if we want to test how the latency reduction system reacts to sudden changes in workload.

LDAT: This is a tool specifically designed for latency measurement. It consists of a special mouse, a brightness sensor and an application. The mouse and the sensor are connected; the application is able to send a click signal to the mouse and receive data from the sensor. When the mouse clicks, it sends a signal directly to the sensor, which then starts a timer. The timer is stopped when a sufficient magnitude of increase in brightness is detected. It requires a small

modification to the game that we need to measure. It needs to support a *latency flash indicator*, which is a white rectangle that's placed on the screen for short duration, when the user clicks. The sensor is placed on the screen in the exact place, where the latency flash indicator would appear. This way the sensor will get the time of the click and it will also know when the reaction from the game reaches the screen. One large issue with this is that the measurements are noisy and it requires several samples to be taken, which take a long time. In the configuration I used 1 click per second, for 20 seconds. This means it can only be used to measure static scenes. The advantage is that it doesn't miss any sources of latency, so it's very reliable besides the noise in the measurements. It's used in some of the benchmarks and it's compared to the software latency.

High speed cameras:

This was the physical way of measuring latency before LDAT. It has similar advantages and disadvantages as the LDAT, but instead of requiring a relatively cheap gear, and a simple-to-use software, it requires a very expensive high speed camera and a complicated set-up to use. As far as I know the latency was calculated by comparing when a LED lights up on the mouse upon clicking and when the result appears on the screen. This technique is not used in any of the benchmarks.

Modes of operation

I have promised to explain the different modes of operations in more detail, so here it is.

Control: This is when the latency reduction system is not active. The game uses the naive approach to limited delay in the pipeline as described earlier. Up to 5 frames are allowed to run parallely in the game, which means that the simulation stage will wait for the completion of the frame from 5 frames ago. This allows the application to take advantage of all the stages under any kind of workload to achieve its full potential in terms of frame rate. This way the application increases slop in the general case(Activision page 55) to allow high frame rate.

Unlimited: In this case the goal is to minimize the latency while preserving the frame rate. There is one known case, where the frame rate is expected to be lower than in the control case, which is when the workload suddenly decreases

by a significant amount. In such cases, the latency pool defines how quickly the latency reduction system can react. If the workload drops faster than the algorithm can react, then the frame rate will lag behind, meaning that it will start out from the original low frame rate and it will reach the frame rate of the control application after a short delay. The frame rate won't get lower than it was before the drop in workload. In this case, the latency is also expected to increase, but it may never exceed the latency in the control case. In any other case, the frame rate has to be equal and the latency cannot be higher than that of the control case. In this mode of operation the *intendedFlip* is calculated by approximating when the current frame can be completed based on previous performance statistics. Since several frames are allowed to run simultaneously, it has to predict quite far. In order to achieve this, the application maintains a queue of expected *frame offsets* for the unfinished frames. The frame offset means the time difference between starting two frames. It's not the total work time, because frames do run parallelly. When approximating the *intendedFlip*, the current frame offset is approximated based on statistics from fully completed frames, then all the approximated frame offsets are summed up and are added to the completion time of the last complete frame.

Limited: In this mode, the frame rate is limited by the user to the *target frame rate*. If the game is not able to achieve the target frame rate, then this mode is expected to behave exactly as the unlimited mode. In case that the game does achieve the target frame rate, then the frame offset is set to *1s/target frame rate.* This will cause the frame rate to drop. After explaining how important frame rate is, it might sound weird to want to do this, but higher frame rate means higher energy consumption by the video card and more resources are used on the computer, which could be used for other things. If the user doesn't need a higher frame rate, than a specific value, then this mode is a good option. There is one interesting case, which is when the game barely achieves the target frame rate. In such cases, the game wouldn't maintain a high enough latency pool that ensures a stable frame rate, so the latency reduction system increases framerate over the target value to make up for the lost latency pool. This happens when the difference between the achievable frame offset and the target offset is less than the latency pool. This effect can be seen in some of the plots presented in the results section.

VSync: Based on seeing the performance statistics, it is easily mistaken for the limited case, because a very similar result can be observed, but the underlying goal is very different. This mode aims to align each frame's completion with the next vertical blank. These events happen periodically at equal time intervals. The demo application uses a simulated vertical blank signal, to make it configurable and hardware independent, and also because the vblank signal is not generally visible to the application. Specific drivers can expose this information through their API, but this is not necessary for them to do. In the case of VSync mode, having an image for all vertical blanks is the high priority task instead of achieving a high frame rate. This will usually limit the application to the same frame rate as the vsync refresh rate, but if the game is capable of achieving a multiple of that, depending on a few settings, it might run at a multiple of the target frame rate. The opposite effect is also possible, if the game can't achieve the target frame rate, it could get locked to a half frame rate, skipping every second vertical blank. This is not always the case, it's just a tendency of the latency reduction system. The software latency in the measurements doesn't include the time between the completion of the frame and the next vblank, because it's defined to be the difference of the frame completion and the input sampling. In this mode of operation, what we expect is that all frames will complete right before one of the vertical blanks (the difference should be the specified latency pool); and the frames should be done with minimal slop (in this case, the latency pool is the time after the frame completion, not the slop, or delay). This mode is not necessarily a good idea if the application cannot achieve the target frame rate. In such cases, it could be better to turn off vsync. The vsync is used to avoid screen tearing. If this is very important to someone, vsync mode does work at lower frame rate than the target frame rate. In the case of 60 fps for the target frame rate, a game that would achieve 44 fps, will probably get locked to 30 fps, which is a significant difference.

Description of the methods used in the processing of the topic

Runtime measurements

The latency reduction system needs to be able to react to dynamic changes in the application's workload. This is only possible if the appropriate measurements are being done while the application is running. In this section I would like to describe what and how is measured in the solution.

Let's start with how the data is organized: The frame graph provides a very useful framework for doing the required measurements and it allows them to be collected per-node, but only in the CPU stages. A CPU stage is able to dispatch several work packages to execution, where each work package can optionally dispatch additional GPU work. The frame graph manages these collectively per-node, but the measurements need more fine grained information in this case. So the structure of the measured data the following:

- CPU nodes exactly as present in the frame graph
- Execution packages
- GPU packages

CPU packages have all the dependencies from the frame graph, which involve another CPU stage. This means the stages: simulation, before draw, record. The readback stage is ignored here, because it's intended to be called after everything else, so it doesn't impact the latency of the application, it runs in parallel with a low workload, so it doesn't impact frame rate either.

Execution packages in the measurement graph depend on the CPU node that submits them.

GPU packages in the measurement graph depend on the execution package that submits them.

These packages are collected every frame from a specified amount of frames that have been completed before the current frame. More frames provide better quality information, but that also means that these are less up-to-date and the application won't handle dynamic changes so well. In the final solution the quality of information proved to be sufficiently good with just a single frame of

data, and the freshness of the information proved very important. So a single frame of data is used, but the software is configurable, because in some very stable applications it might be worth using more frames of data.

To sum it up, we have a second graph, which is very similar to the frame graph, but it only contains a small snapshot of the application's execution, the readback stages are completely ignored and the execution along with the GPU stages are split into individual submissions. There is just one last thing to add, that makes this graph valid: The computational resources create implicit dependencies in the graph. Since this graph is created after the frame has been completed, we know what computational resources were used in the process. There is a *queue id* attached to every package or node in the measurement graph, which defines a general computational resource id. In the case of the demo application it means thread ids on the CPU and device queue ids on the GPU. An implicit dependency is automatically created between any two packages that are executed on the same resource without any other node in between (on the some thread). This information could be used to identify optimization opportunities, because packages with implicit dependency could have been done in different order. Latency can be calculated in each ordering and if the other order works better, the programmer could be warned about wrong scheduling. This is not done in this application, but the framework would support this feature.

The measured information:

- When a packages is available
- Start of the execution
- End of the execution
- Manual slop
- Latency sleep

The availability, start and finish timings are trivial when using a frame graph, but that only applies to CPU packages. Execution packages are similarly easy, but GPU packages are more difficult to measure. They require the use of a GPU-internal clock, that measures time in it's own time space. This has to be calibrated with the CPU clock and it's not enough to do so once, because the two clocks drift apart over time. Calibration has to happen regularly and there is about ~1ms calibration error based on measurements. The latency pool has to

be able absorb such measurement errors. The manual slop is a duration that the application spent on waiting for a previous work, but manually. The frame graph deals with most of the synchronization, but it can't always be applied. For example, when the application tries to acquire a new image from the swap chain, it might have to wait for an image that's still in use. This would qualify as slop, because this duration could have been eliminated just by starting the whole frame later (this is the case when the application waits for work that has been submitted in previous frames). The latency sleep is the delay added by the latency reduction system. The slop graph handles several nodes with latency sleep, but in practise only one exists: the input sample node. That's the only place that can benefit from latency sleep. This value is required, because all measurements need to be independent of the latency reduction. The latency sleep appears in measurements as longer work time for a specific node. This has to be negated.

Extrapolated values

Before showing the graph based algorithm, I would like to first explain what is expected from the algorithm and why.

The formula for the latency sleep is the following:

waitUntil = intendedFlip - workForecast - targetSlop

workForecast: As mentioned before, this value is an extrapolation of work times from previous frames. The graph based algorithm has to be able to calculate this value. This would be very easy in a single threaded application that only uses CPU. Simply the duration of all work packages would be summed up. The demo application uses many threads, the GPU, where it could have used several device queues. This parallelism is quite common in video games. In theory, the value that we need here is the duration of a single frame taken out of context. In other words, if we just started a single frame without any other frames in progress and without starting any other frames before this one finishes, then we would need the duration that it took to complete. This theoretical frame is a *compact frame*. An extra complication is that the measured time would still contain the latency sleep, which has to be negated by the application.

intendedFlip: This is even more tricky to properly estimate, than the compact frame duration. As mentioned before, we need the *frame offset* to calculate this value. The frame offset is the time that the next frame has to wait before starting after the current frame starts. The idea behind finding this value is to construct two of a compact frame (without latency sleep) and to shift the second one until there is no overlap on any of the independent computational resources (CPU threads / GPU queues). The frame offset will be equal to the shift offset. The main challenge is that the compact frame can be constructed in many different ways, which yield the exact same frame duration, but they can yield very different frame offsets. It's required to construct a compact frame with minimal frame offset.

Why don't we just extrapolate the slop itself? The algorithm does calculate the slop, which is exactly the time that could have been used for latency sleep. Of course, latency sleep affects this value, but it's easy to restore the real slop:

real slop = latency sleep + measured slop

If latency sleep is higher than real slop, then the measured slop will be 0, so this condition breaks, but this is still not an issue, because it is possible to recover from this situation thanks to the latency pool. The slop is also a lot easier to calculate than the frame offset for instance. The issue with slop is that it can change very rapidly and it's very sensitive to latency reduction. The slop would be the ideal latency sleep duration, but this only means the slop of the current frame. **Extrapolating slop does not work in practise**.

The graph based algorithm

Now we know what we have and where we want to arrive. Let's fill in the gap with the algorithm, that is the heart of this project.

The algorithm consists of three main parts and a final small part that reads out a few values from the results. So let's see these in more detail:

Topological ordering

Since the edges in the measurement graph represent dependencies between work packages, we can assume that the game specified them in a valid way without deadlocks, therefore the measurement graph is a flow graph. This

makes it possible to perform a topological ordering. Each node will only depend on nodes that are earlier in the ordered array.
The topological ordering is performed in two steps:

Prepass

The number of dependencies for each node is counted. Just to clarify, nodes with 0 dependency don't depend on other nodes. It's guaranteed to have at least one such node, because the graph is a finite flow graph.

Ordering

A loop goes over all nodes, and inserts the index of nodes with 0 dependency count into an indexing array. When the index of a node is inserted into this array, the dependency count is decremented for all dependent nodes. If the dependency count of the dependent nodes reaches 0 with this decrement and they have been visited by the loop, they will be inserted to the array as well recursively. The implementation uses an equivalent loop instead of the recursion. If the inserted node hasn't been visited by the loop before, then the loop would place it into the ordering array for a second time.

Dynamic programming

This section implements a recursive function using dynamic programming technique to calculate several values. The function is applied to nodes, and the result only depends on the current node and nodes that depend on it. This means that, the result for the *i*th node in the topological ordering, the function only depends on nodes in the range of *i..N*, assuming that there are *N* nodes in total.

The calculated values

$sloppedMin_i$:= min({$startTime_j$ + $slop_j$ +$latencySleep_j$ + $totalSlop_j$ - $depOffset_j$ | for j in dependents of the *i*th node})

$maxWorkTime_i$:= max({$workTime_j$ | for j in defenders of the *i*th node})

$maxSpecializedWorkTime_i$:= max({$specializedWorkTime_i$ | for j in defenders of the *i*th node which are executed on the same stage (CPU/GPU/Execution)})

$totalSlop_i$:= $sloppedMin_i$ - $endTime_i$

$workTime_i$:= $maxWorkTime_i$ + ($endTime_i$ - $startTime_i$ - $latencySleep_i$)

$specializedWorkTime_i$:= $maxSpecializedWorkTime_i$ + ($endTime_i$ - $startTime_i$ - $latencySleep_i$)

The meaning of these values

$sloppedMin_i$: This is the latest time, when the *i*th node could have finished without delaying the frame's end. It assumes all dependent work to be delayed as much as possible without delaying the frame as well.

$maxWorkTime_i$: The time it would take to finish a **compact** frame starting after the *i*th node.

$maxSpecializedWorkTime_i$: The time it would take to finish the current stage of a **compact** frame starting after the *i*th node.

$startTime_i$: The measured starting time.

$endTime_i$: The measured finish time.

$slop_i$: The manually recorded slop time.

$latencySleep_i$: The time this specific node was delayed in order to decrease latency. In the demo app (and in the usual case) it will only contain any non-zero value for the input sampling node.

$totalSlop_i$: The time this node could have been delayed without delaying the frame's completion.

$depOffset_i$: This value is specified if a dependent node is able to start execution earlier than it's dependency is actually completed. This offset is the time difference between the end of the dependency and the actual execution of this node. If the *i*th node started later than it's dependency, then this value is 0. It's only applied to GPU work packages, in which case there is exactly one dependency.

$workTime_i$: The time it would take to finish a **compact** frame starting before the *i*th node.

$specializedWorkTime_i$: The time it would take to finish the current stage of a **compact** frame starting before the *i*th node.

There are a few other values calculated, but those are not important for latency reduction. Instead they provide better debugging information.

Compact frame calculation

The compact frame timings would be calculated in a separate function, but it is compatible with the loop used in the dynamic programming implementation of the previous function, so these are also calculated here:

$highestWorkTime = \max(\{workTime_i \mid i \text{ for all nodes}\})$

$highestCpuWorkTime = \max(\{specializedWorkTime_i \mid i \text{ for all CPU nodes}\})$

$highestExecWorkTime = \max(\{specializedWorkTime_i \mid i \text{ for all execution nodes}\})$

$highestDeviceWorkTime = \max(\{specializedWorkTime_i \mid i \text{ for all GPU nodes}\})$

The above values are used to calculate latency reduction independent frame limit overall, on CPU, on execution queue and on GPU respectively.

In addition to these the first and last node in each computational resource (queue id) are collected out, but only from a single frame. If the measurement graph contains information from several frames, just one is considered here. This will be used to construct a compact frame.

Extraction of compact frame timings

We can already construct **a** compact frame:

$compactStartTime_i := workTime_i$

Although this will not provide minimal frame offset. Several nodes have a certain freedom of movement, which affects frame offset.

This section of the algorithm, the start and end nodes are considered on all computational resources independently. It's enough to calculate the minimum time that needs to pass between the start node and the end node on each computational resource in order to calculate the frame offset. These values can be calculated again with dynamic programming. There is a *duration* value assigned to each node. The dynamic programming implements the following recursion between the indices (*start*, *end*):

$localDuration_i := endTime_i - startTime_i - latencySleep_i$

$duration_i := \max(\{localDuration_i\} \cup \{duration_j + localDuration_j \mid \text{where } node_i \text{ depends on } node_j \text{ and start} <= j < i\})$

This calculation includes all nodes that need to happen between the start and end nodes, regardless of the queue id (computational resource). This is important, because some threads could be heavily synchronized with another.

In such cases, the frame offset would be defined by the other thread. This method of calculation considers all synchronizations.

Reading out results

The work time will be $\max(\{workTime_i \mid 1 <= i <= N\})$.

Let's see how we can obtain the frame offset:

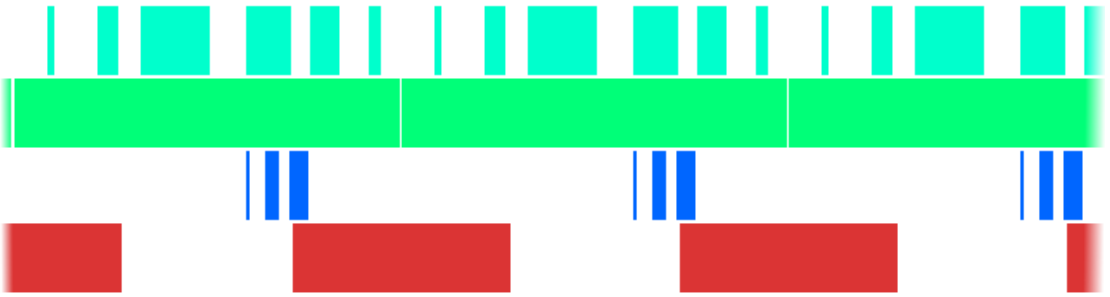


Figure 6: The different lanes are different computational resources. The first two correspond to CPU threads, the blue is the execution queue (which is also a CPU threads, but it has a designated role) and the red is the GPU. The frame offset will be the duration of the green queue.

Figure 6 illustrates why we need to use the *duration* values calculated in the previous part of the algorithm. The *duration* for the first lane is almost as high as for the green lane. It has a lot less work, but it is heavily synchronized with the green lane. The GPU finishes later than the green lane, but also starts later, so it doesn't limit frame rate any more. The frame offset is defined by the highest duration, because picking a smaller duration queue would cause other queues to overlap between frames.

The duration of each queue is the $duration_{end}$, where end is the index of the end node in that queue. The exact timing of every node is possible to calculate in the compact frame with minimal frame offset, but it's not necessary. The latency reduction algorithm only needs the frame offset itself.

Integrating the software into other systems

The latency reduction algorithm assumes a frame graph, but it's not strictly necessary to have an explicit frame graph implementation in the algorithm. For example in a simple single threaded application, which has a single CPU stage, that processes all frame information at once, and a GPU stage, then these

would be the two nodes of the implicit frame graph. In this case, the algorithm wouldn't be able to reduce inter-frame latency, but it can still handle delays between the two stages.

The latency reduction algorithm requires the implementation of an interface that provides the algorithm with the required information. This includes the following information:

- Number of nodes (this can be different in every frame)
- Number of child nodes for every node (nodes that depend on the specified node)
- The id of said child nodes
- Node information:
    - start time
    - end time
    - manually created slop
    - latency sleep duration inside the specified node
    - whether the node may be delayed (only the node, that provides the image to the screen is not delayable in normal cases)
    - type of node (CPU, execution, GPU)
    - frame id (which frame the node does work for)
    - thread id (the computational resource id)

The interface will also provide feedback for every node:

- latency (time from the node until presentation/target node)
- directs lop (latency generated directly after this node)
- implicit dependency (a node, that's only depending, because it shares the same queue)
- extra slop without implicit dependency (this can be used to identify cases, where the implicit dependency limits latency)
- sleep time (same as input latency sleep)
- work time (time it takes to finish the compact from starting from this node)
- specialized work time (same, but only considering the current stage)

There is also some global information returned by the algorithm:

- The feedback info for the input node (this is redundant, but makes the interface more usable)

- work time: required for latency reduction
- frame offset
- CPU work time
- CPU frame offset
- execution work
- execution offset
- device work
- device offset

The stage specific values are not required for latency reduction, but they are useful for debugging otherwise. These values are all independent of latency reduction. This way the user can see which computational resource is limiting the frame rate of the application. The latency reduction would not be considered when using general techniques for calculating these values.

Weak points

Threads share computational resources

So far the algorithm assumed that a different thread uses a different CPU core. This is not guaranteed. The application would not function properly in such cases.

Chaotic workload

The latency reduction works with extrapolated values. A chaotic workload makes this problematic. Increasing the latency pool does solve this case, but it also increases latency.

Sleep time

The latency is reduced by performing a latency sleep for a specific duration. Unfortunately, normal sleep doesn’t guarantee precision. In some cases, I have measured the sleep duration to be twice as much as it should have been. To counteract this, I have used a *busy sleep*, which is basically an empty loop that only stops after a certain time has passed. This is not a nice solution, especially when it has to wait for long.

Measurements

There is a problem with measurement errors. On the hardware configuration that I used, I measured the GPU-CPU timeline calibration to introduce about ~1ms measurement error to all timings. This continuously drifts even further apart, so recalibration is regularly required, which means that GPU measurements before and after calibration wouldn’t agree. Such measurement errors need to be handled by the latency pool. The latency pool has to be at least as large as the measurement error.

Summary evaluation of the results and description of the conclusions to be drawn

Static measurements

In this category I have run the same benchmark scene with several different settings. This scene is static, which means that the base workload is as close to constant as possible. The workload in different configurations is managed by manual workload, which means that both the amount and the variance of the workload is controlled. Some of the tests are intentionally done with varying workload. I used normal distribution evaluated at every frame for generating a specific workload using a specified average and standard deviation.

This benchmark scene allows the usage of the LDAT device. The LDAT device measures **input-to-display latency**, while the internal measurements correspond to software latency. Conversion between these values is not possible, because the input-to-display latency includes noise values outside of the software latency, but the correlation is still visible.

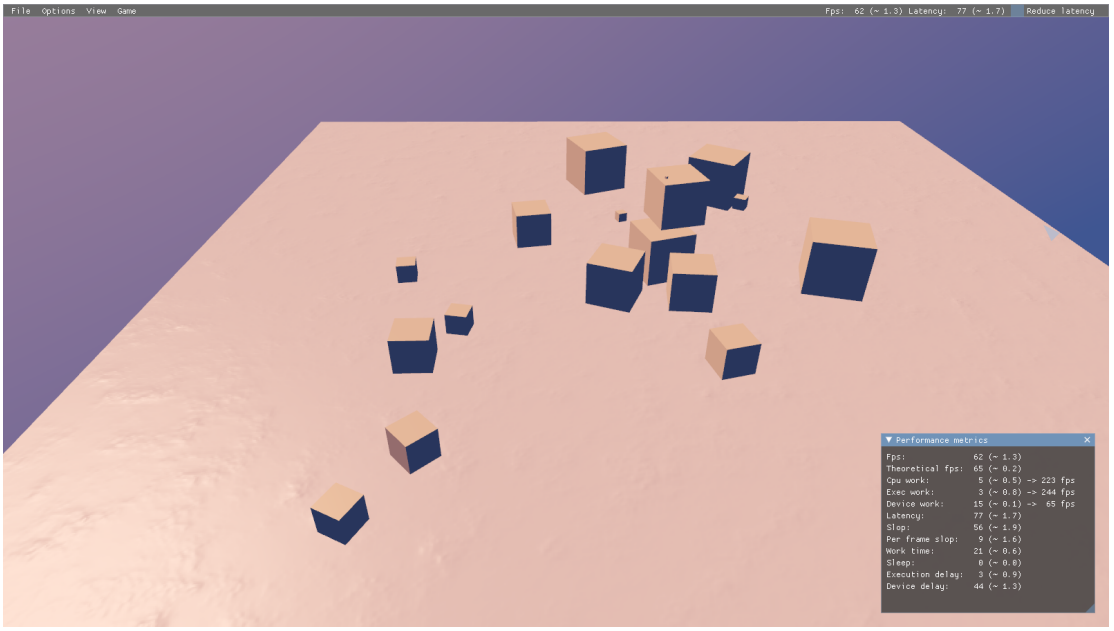


Figure 7: The scene used for static tests. The camera is stationary and there is no physics. Every object is cheap to render.

Mode	Noisy	Pre-input work	Latency pool	Target fps	Bottleneck	FPS	Latency	Missed frames	LDAT
none	yes	no	3		cpu	61.4	22.1		84.3
unlimited	yes	no	3		cpu	61.2	22.3		73.5
unlimited	yes	no	10		cpu	61.4	22.2		72.6
vsync	yes	no	3	45	cpu	45.4	28.5	9.0%	74.8
vsync	yes	no	10	45	cpu	45.7	29.8	0.3%	72.4
none	yes	yes	3		cpu	61.1	22.1		67.3
unlimited	yes	yes	3		cpu	61.0	22.2		64.7
unlimited	yes	yes	10		cpu	61.0	22.3		61.6
vsync	yes	yes	3	45	cpu	45.2	29.6	0.4%	70
vsync	yes	yes	10	45	cpu	59.4	22.8	0.0%	59.3
none	yes	no	3		gpu	60.9	79.5		134.3
unlimited	yes	no	3		gpu	62.2	36.9		87.7
unlimited	yes	no	10		gpu	62.2	44.0		90.4
vsync	yes	no	3	45	gpu	45.3	38.5	2.6%	73.1
vsync	yes	no	10	45	gpu	45.5	38.6	0.9%	70.3
none	yes	no	3		exec+gpu	61.0	80.3		144.2
unlimited	yes	no	3		exec+gpu	62.1	45.4		90
unlimited	yes	no	10		exec+gpu	62.3	52.5		83.2
vsync	yes	no	3	45	exec+gpu	45.7	46.6	7.1%	78.2
vsync	yes	no	10	45	exec+gpu	45.7	46.7	0.7%	73.6
none	yes	no	3		exec	58.0	86.7		142.2
unlimited	yes	no	3		exec	57.5	42.9		74.8
unlimited	yes	no	10		exec	58.6	49.5		81.4
vsync	yes	no	3	45	exec	45.7	44.6	10.7%	78.8
vsync	yes	no	10	45	exec	45.8	45.5	0.0%	75.7
none	no	no	3		cpu	59.9	23.1		86.4
unlimited	no	no	3		cpu	60.0	23.0		84.5
vsync	no	no	3	45	cpu	45.0	29.7	0.0%	74.4
none	no	yes	3		cpu	60.0	22.5		77.6
unlimited	no	yes	3		cpu	60.1	22.6		79.6
vsync	no	yes	3	45	cpu	44.7	29.9	0.9%	65.8

none	no	no	3		gpu	61.2	78.4		148.1
unlimited	no	no	3		gpu	61.7	36.9		96.6
vsync	no	no	3	45	gpu	45.1	38.4	0.7%	69.9
none	no	no	3		exec+gpu	60.0	79.7		148.2
unlimited	no	no	3		exec+gpu	61.7	45.0		98.7
vsync	no	no	3	45	exec+gpu	45.1	46.3	0.9%	77.6
none	no	no	3		exec	57.4	86.3		140.2
unlimited	no	no	3		exec	58.0	41.7		73.7
vsync	no	no	3	45	exec	45.0	44.6	0.2%	79.1
Control LDAT app									68.3

Mode: The mode of operation

Noisy: The manual work is noise (regarding all added workload)

Pre-input work: The added CPU workload is partially placed before the input sampling.

Latency pool: The application tries to maintain this amount of latency slop. Specified in milliseconds.

Target fps: In vsync cases, this is the simulated display refresh rate.

Bottleneck: This is the stage with the highest workload. exec+gpu means that execution is more expensive than CPU and GPU is more expensive than execution, so work accumulates before both execution and GPU stages.

FPS: Frame rate in frames per second.

Latency: Software latency measured by the application in milliseconds.

Missed frames: Percentage of the simulated refresh periods, where no frame has been presented until the end of the period.

LDAT: The measured input-to-display latency in milliseconds.

Control LDAT app: This application has almost no work to be done in each frame, so the input-display latency only contains the latency sources independent of software latency and the latency that comes from the applications refresh rate, which is expected to add half of the refresh time as extra latency on average.

Details about how the measurements were conducted

The LDAT measurements were done separately from the others, because those require manual measurements. Other tests were automated. In case of LDAT measurements the equipment was set up to automatically click 20 times with 1 seconds between each click. The measured values are averaged from these 20 samples for each configuration.

The rest of the measurements were done in an automated manner. Each configuration has been running for about 10 seconds with an extra 5 seconds at the start which allowed the application to settle everything. The measurements were averaged over the 10 seconds interval. All configurations were executed right after each other without delay, so the computer stays warmed up and cooling doesn't influence the results. There was also extra configuration at the beginning that wasn't recorded in the results. This was added to warm up the computer for the proper tests.

Controlled dynamics

This benchmark is created to test one specific aspect of the latency reduction algorithm: how well it responds to dynamically changing content.

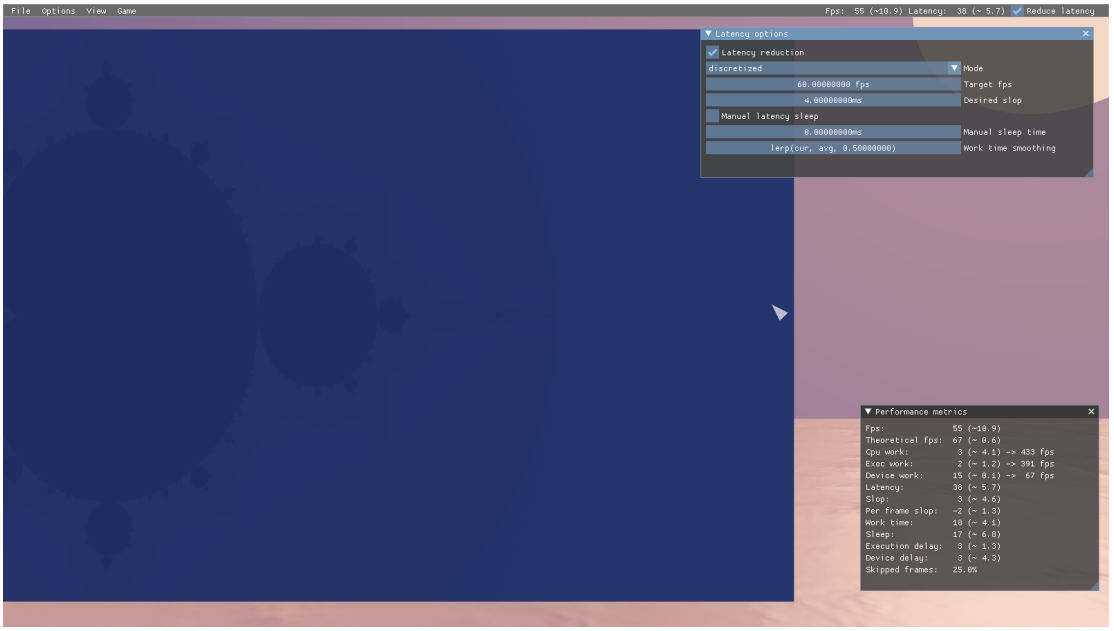


Figure 8: Controlled dynamics benchmark scene. The camera is placed right in front of a large object, which puts a large amount of workload on the GPU depending on how many pixels it occupies on the screen. The camera is rotated to face the object,

then face away, then back. The speed of rotation increases throughout the benchmark.

The magnitude of the change in the workload is controlled. In the beginning of the benchmark the change is very gradual and it speeds up as the benchmark progresses. As the camera is rotated, the workload increases and decreases alternatingly, so both kinds of dynamic change are tested.

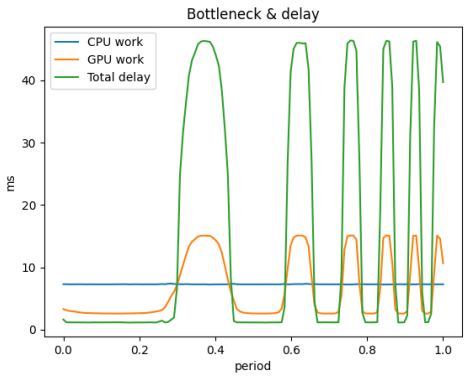
Details about how the measurements were conducted

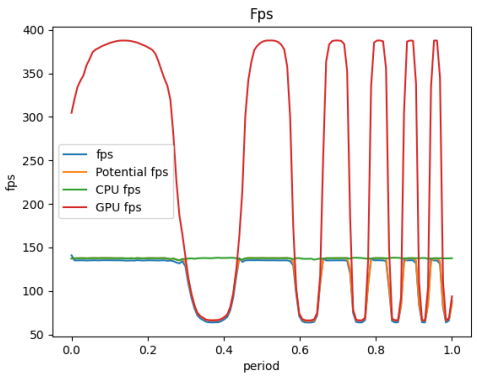
The computer is warmed up as before to avoid cooling related differences in the results and the different configurations are executed one after the other without delay.

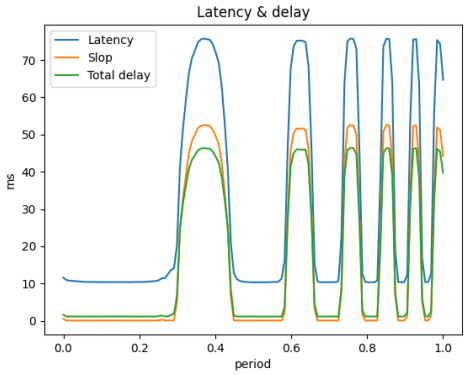
Each configuration has a 5 seconds preparation time to settle everything. Then the application completes 30 periods of 60 seconds measurement. The measured values are filtered and resampled (downsampled) from each period, then they are averaged. The resampling is required, because the sample points are the frame-end times, which are not consistent. Even the frequency changes. The filtration is required, because the displayed plots can't handle the amount of raw data and it's highly prone to aliasing. Especially the missed frames value.

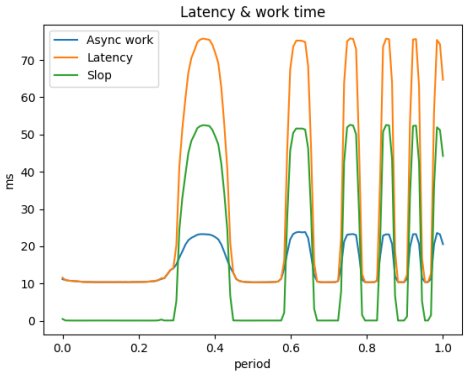
Results

Control

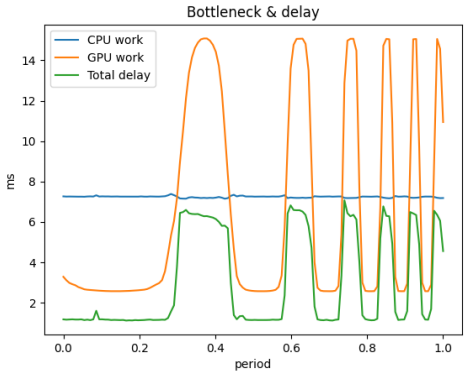


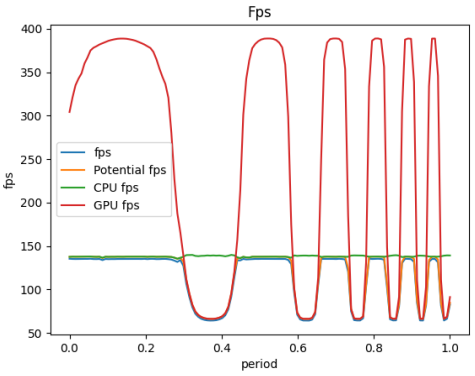


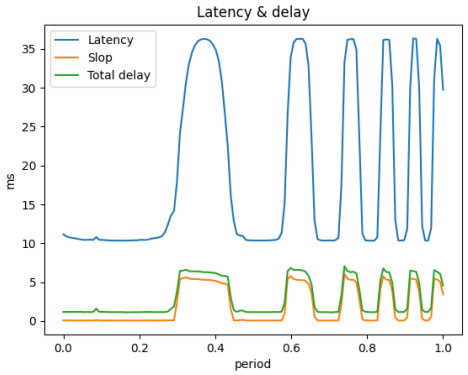


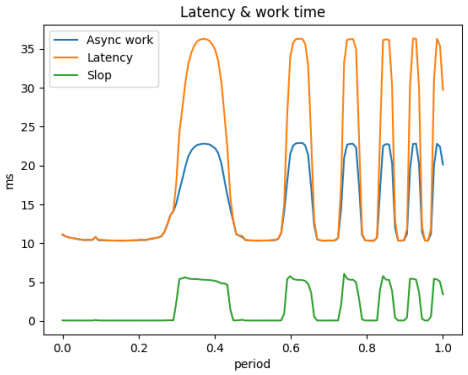


Unlimited

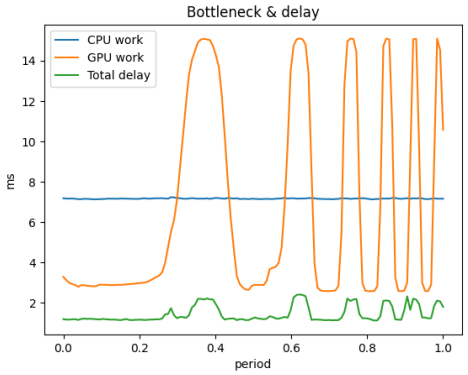


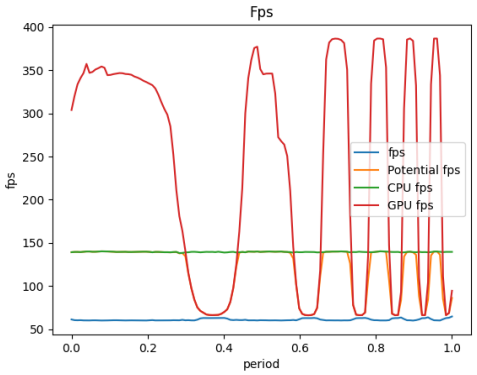


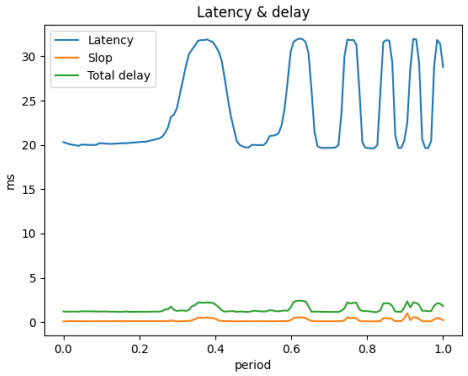


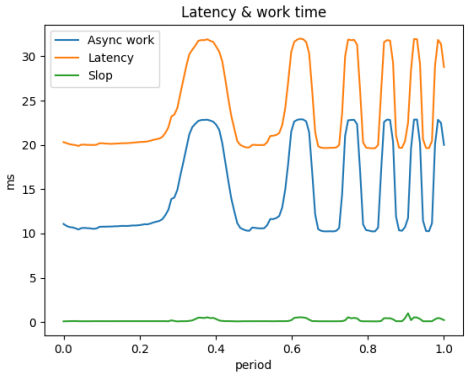


Limited

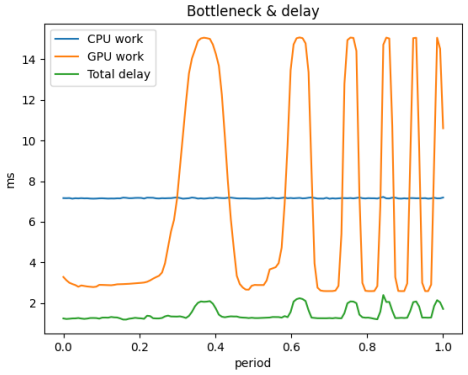

Bottleneck & delay
CPU work
GPU work
Total delay
ms
period

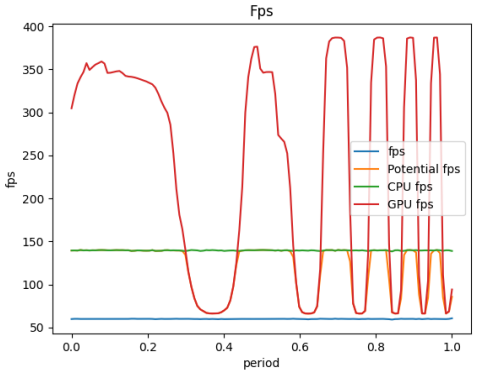

Fps
fps
Potential fps
CPU fps
GPU fps
fps
period

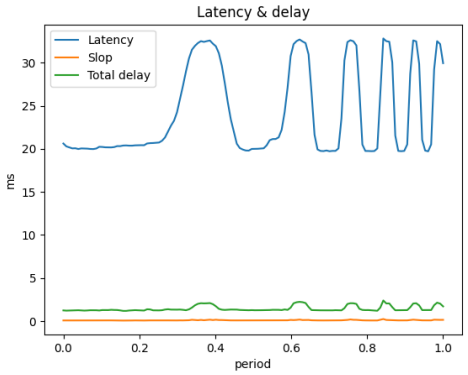

Latency & delay
Latency
Slop
Total delay
ms
period

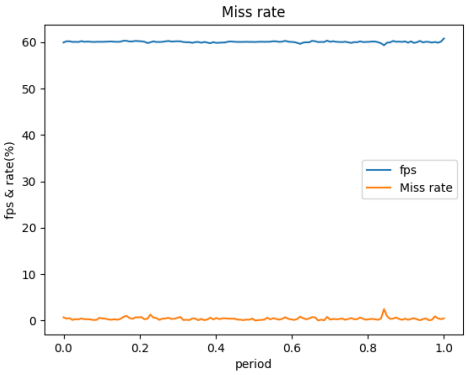

Latency & work time
Async work
Latency
Slop
ms
period

Vsync

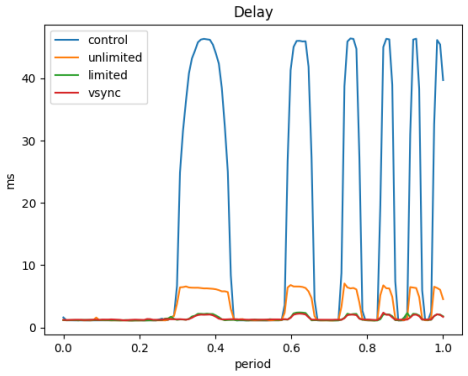

Bottleneck & delay
CPU work
GPU work
Total delay
ms
period

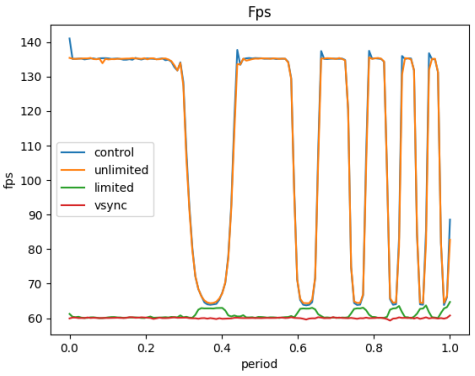

Fps
fps
Potential fps
CPU fps
GPU fps
fps
period

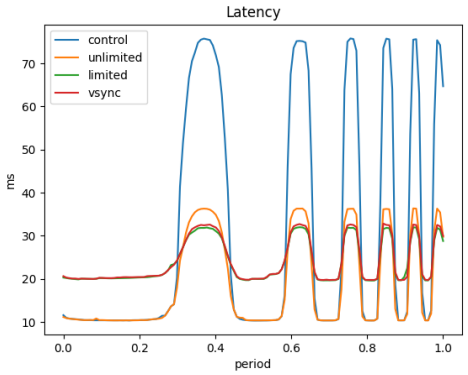


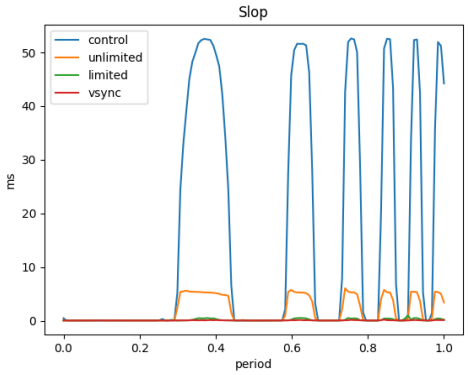


Combined









Interpreting the results

I would like to say a few world about some of the important aspects:

As clearly visible on the combined FPS plot, the unlimited case with latency reduction **preserves frame rate** and almost exactly matches the original

values. The limited case keeps a consistent frame rate and slightly increases it, when the potential frame rate drops to close to the target frame rate. This prevents delaying any frames in case of an unexpectedly long frame.
The vsync case maintains the target frame rate and **barely misses any frames** even with dynamic workload. This was achieved by just 4ms of latency pool.
The **latency is reduced** in all cases as expected.
The *bottleneck & delay* and the *latency & delay* graphs show how the GPU bottleneck leads to accumulation of delay and thus latency.

Realistic scene

This scene is designed to model a realistic workload. There is dynamic workload generated by box emitters (*figure 9*), which add high frequency noise to the CPU workload; the two large objects in the middle are expensive to render on GPU and the time it takes depends on how many pixels they occupy on the screen. There is a mandelbrot rendered behind the skybox, which adds a const amount of workload. There is also manual CPU workload to moderate the amount of noise.
This scene is set up to follow the following measurements:

- Looking at the two large objects from close up limits FPS to 60
- Looking at the sky would increase the GPU limit to 80 FPS
- The CPU always limits the frame rate to about 70

The most important thing about this measurement is the camera movement. I have entered free camera mode and recorded a camera movement that I would deem as realistic for a first person shooter game. The benchmarks replays this camera movement.

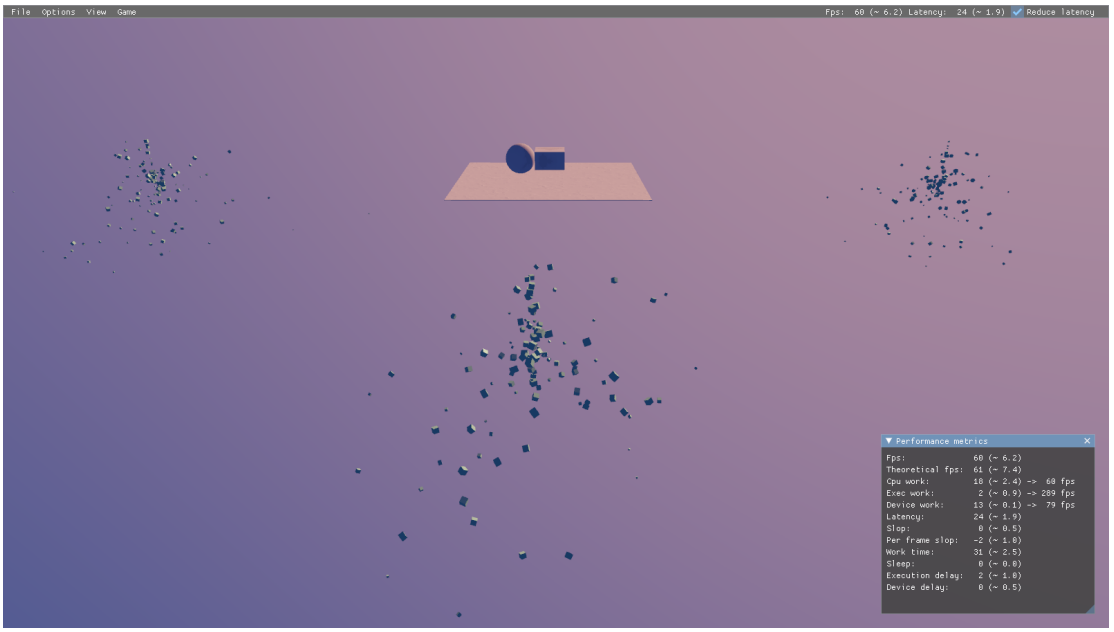


Figure 9: Realistic benchmark scene. The two large objects in the middle are expensive to render on the GPU. The rendering time depends on the amount of occupied pixels. The box emitters generate noise workload on the CPU in the form of expensive physics calculations. These objects also create workload on the recording stage due to their high count. The camera follows a manually recorded path.

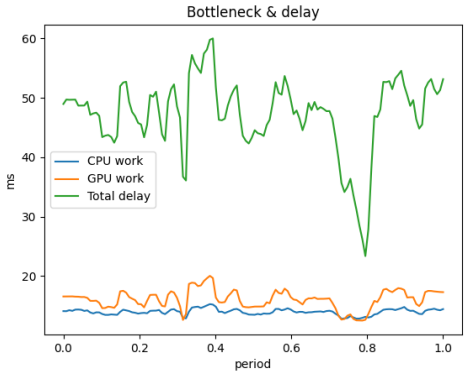
Details about how the measurements were conducted

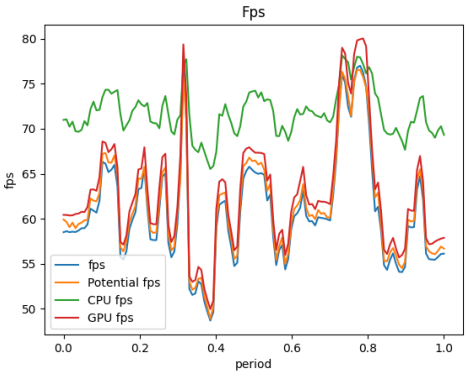
The measurements were done in the same manner as the previous benchmarks. The benchmarks start after the computer has warmed up; they run in succession without delay; there is a 5 seconds startup time for each configuration, where no measurement is done; then the camera replay is repeated 60 times and the results are processed the same way as before.

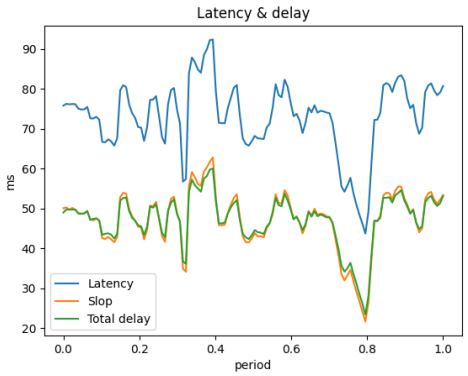
The camera movement is replayed by performing a linear interpolation in the recorded values. The camera translation and orientation are interpolated independently. Quaternions are used for the orientation component to always interpolate through the shortest path. The position and orientation are sampled by time and not frame id, because different configurations will have different frame distribution and count throughout the benchmark.

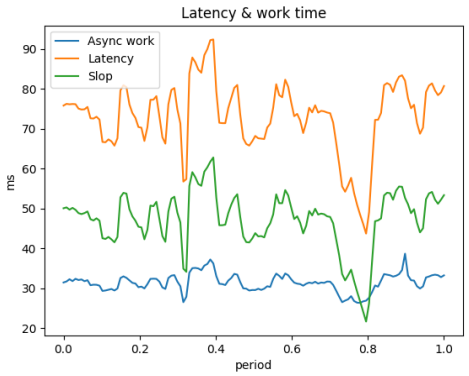
Results

Control

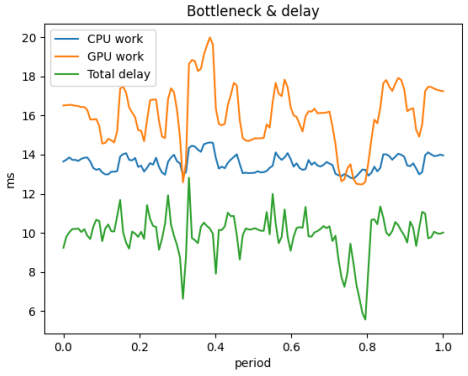


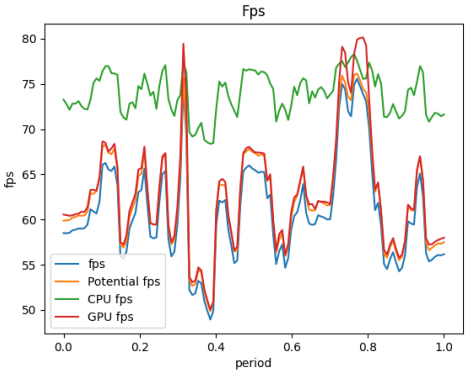


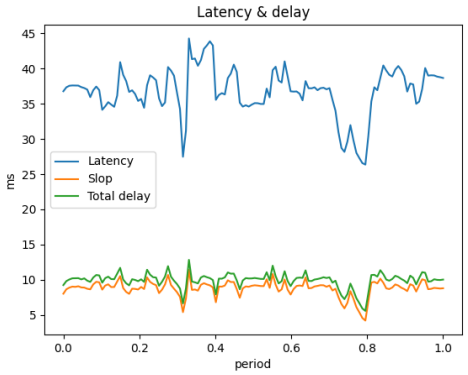


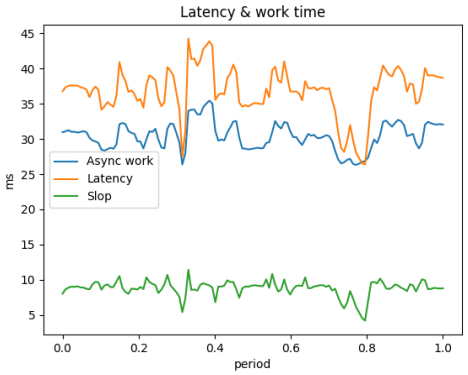


Unlimited

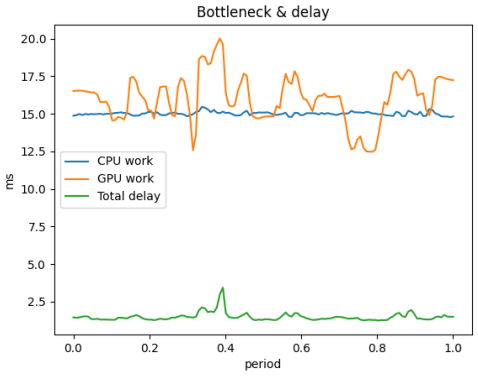


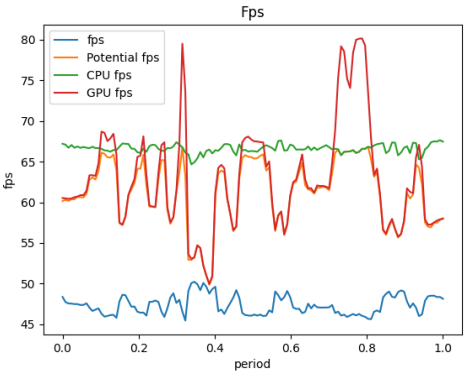


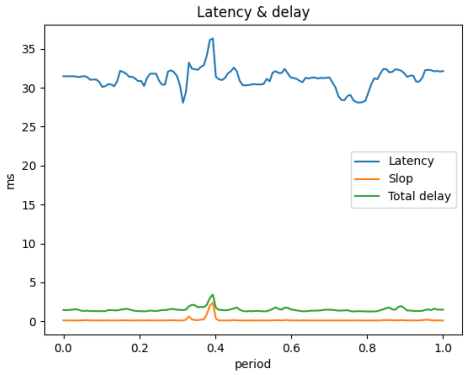

Latency & delay
ms
period
Latency
Slop
Total delay

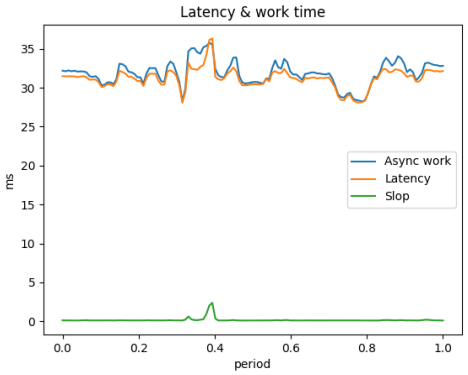

Latency & work time
ms
period
Async work
Latency
Slop

Limited

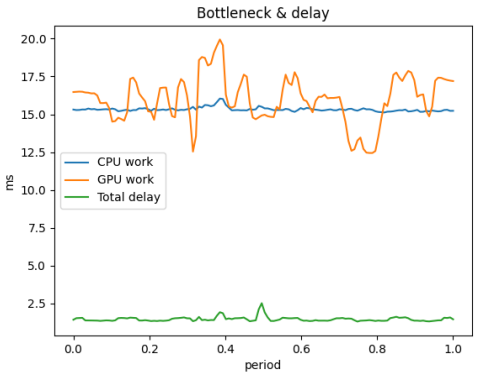

Bottleneck & delay
ms
period
CPU work
GPU work
Total delay

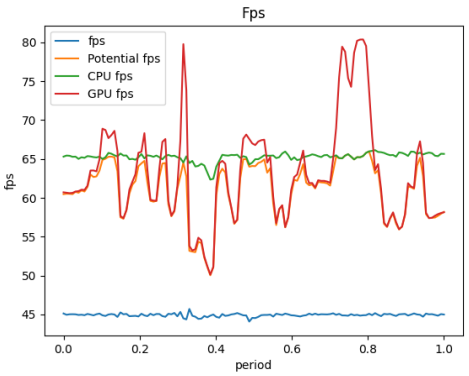

Fps
fps
period
fps
Potential fps
CPU fps
GPU fps

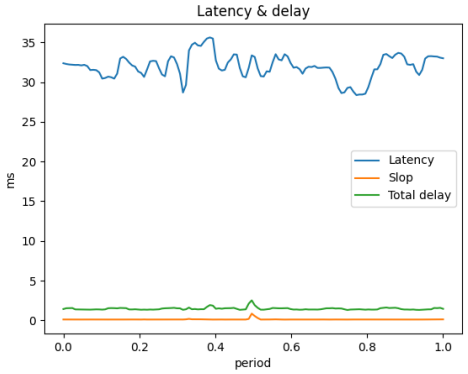

Latency & delay
ms
period
Latency
Slop
Total delay

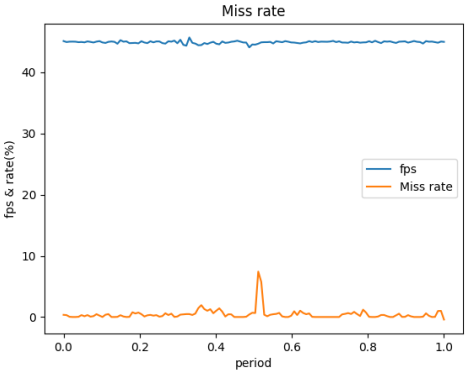

Latency & work time
ms
period
Async work
Latency
Slop

Vsync

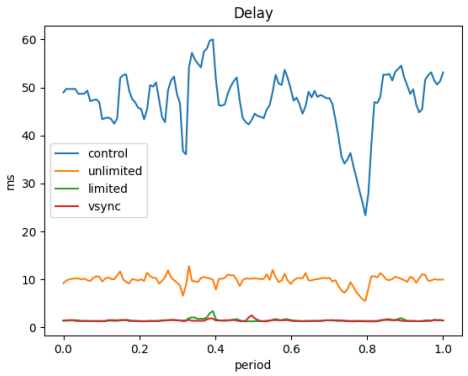


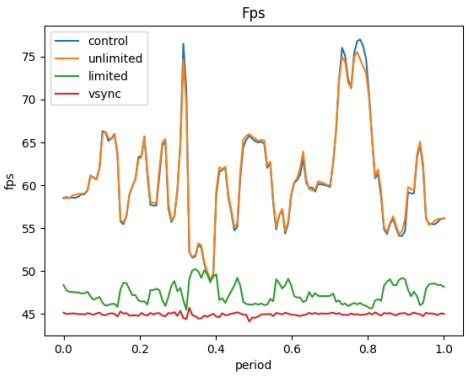


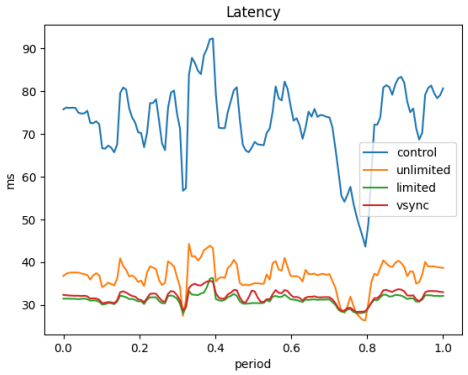


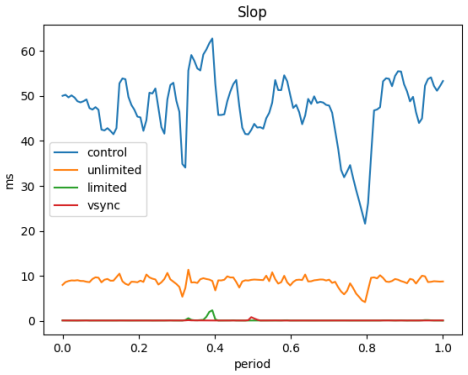


Combined









Interpreting the results

Fortunately the **frame rate is still preserved**, which can be seen on the combined FPS graph. Although there are **more missed frames** than in the previous benchmark, this is still acceptable. This was achieved by using 8ms for the latency pool. Using a higher latency pool would have resulted in a lower miss rate, but in my opinion that is no longer worth it. 4ms of latency pool resulted in too many missed frames.

Most importantly **latency** is again **reduced** in all cases.

Just to add an extra observation, the *bottleneck & delay* graph of the control case shows how strongly the delay reacts to change in bottleneck. The GPU load slightly dips under CPU workload (at ~0.75 period) and the delay starts dropping quite sharply. The entire duration of this period is about 30 seconds.

It's also interesting to observe how the limited case 'mirrors' the framerate of the unlimited cases. Due to the high latency pool, it's constantly in the gray zone, where it can't provide the target frame rate with the required latency pool. Therefore it increases the frame rate to make up for the latency pool.

Conclusions

The algorithm presented in this document is capable of reducing inter-frame latency and it also limits accumulated delay between different computational stages. All tests show the expected results. Measured software latency correlates with LDAT measurements; there is a high latency reduction in GPU limited cases; it is possible to schedule work before the input sampling if there is no logical dependency, which enables the reduction of inter-frame latency; the algorithm reacts well to dynamically changing environments.

The effectiveness of this algorithm is limited by the stability of the application that is being optimized. Too much variance in frame times requires higher baseline latency to avoid throttling the software. This limitation is not specific to this algorithm. The same result should be expected from any other approach that attempts to optimize latency by re-scheduling work.

This algorithm is not restricted to framegraph based applications, but in other cases the inter-frame latency won't be reduced automatically. It could still help to manually identify reordering possibilities if sufficient information is provided.

There is a potential limitation regarding supported platforms: the algorithm requires timings from the GPU timeline. If this is not supported on a specific platform for any reason, then latency can only be reduced by a limited amount. It can still be reduced by the amount of time spent on waiting for the swapchain when acquiring a new image. This time can be high in GPU limited applications. Inter-frame latency reduction is not affected in this case either. When GPU timings are not available, then the target node on the framegraph needs to be redefined. This would normally be the presentation event, when the image is presented on the display. Instead, it has to be defined to be something on the CPU timeline, where timings are available. For example the event, where the presentation command is dispatched to the GPU.

Future work

Threads vs cores

Currently the presented algorithm only considers threads, not cores. It assumes that each thread is a separate computational resource. Using a core id instead of thread id wouldn’t solve this, because running two operations on the same core doesn’t create a logical dependency unlike in case of threads.

Auto optimization suggestions

The algorithm returns information that can be used to identify cases where optimization is possible. It is possible to find implicit dependencies in the graph, where reversing the order of the two nodes would lead to lower latency. Implicit dependencies mean that the dependency only exists due to the common thread usage, but the two nodes could have been executed in any order or even parallelly. This can’t be fixed by this algorithm, it probably requires higher level design changes in the application. The goal would only be to warn the programmer about cases where this is possible.

Improve predictions

Currently the work time predictions use only statistical data about previous frames. In some cases the application might know in advance that a frame will take very long. This could be used to make better approximations, but this is difficult to take advantage of in practice.

Bibliography

Activision. *Controller to display latency in Call of Duty*. https://www.gdcvault.com/play/1026327/GDC. Accessed 1 11 2021.

Banatt, Eryk. *Input Latency Detection in Expert-Level Gamers*. 2017, https://cogsci.yale.edu/sites/default/files/files/Thesis2017Banatt.pdf. Accessed 1 11 2021.

Carmack, John. *Latency mitigation strategies*. *https://danluu.com/latency-mitigation/*. Accessed 20 11 2021.

Frostbite / Electronic Arts. *FrameGraph: Extensible Rendering Architecture in Frostbite*. *https://www.gdcvault.com/play/1024612/FrameGraph-Extensible-Rendering-Architecture-in*. Accessed 20 11 2021.

Ivkovic, Zenja, et al. *Quantifying and Mitigating the Negative Effects of Local Latencies on Aiming in 3D Shooter Games*. *https://web.archive.org/web/20170808203210id_/https://www.cs.usask.ca/faculty/gutwin/898-2015/readings/06-latency/ivkovic.pdf*. Accessed 20 11 2021.

Kim, Joohwan, et al. *Post-Render Warp with Late Input Sampling Improves Aiming Under High Latency Conditions*. *https://dl.acm.org/doi/10.1145/3406187*. Accessed 20 11 2021.

Liu, Shengmei, et al. *Lower is Better? The Effects of Local Latencies on Competitive First-Person Shooter Game Players*. 2021, https://dl.acm.org/doi/pdf/10.1145/3411764.3445245. Accessed 1 11 2021.

Mark, William R., et al. *Post-Rendering 3D Warping*. *https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.13.1789&rep=rep1&type=pdf*. Accessed 20 11 2021.

Microsoft. *Reduce latency with DXGI 1.3 swap chains*. *https://docs.microsoft.com/en-us/windows/uwp/gaming/reduce-latency-with-dxgi-1-3-swap-chains*. Accessed 20 11 2021.

NVIDIA. “NVIDIA Reflex.” *https://www.nvidia.com/en-eu/geforce/technologies/reflex/*, https://www.nvidia.com/en-eu/geforce/technologies/reflex/. Accessed 1 11 2021.

Spjut, Josef, et al. *Latency of 30 ms Benefits First Person Targeting Tasks More Than Refresh Rate Above 60 Hz*. 2019, https://research.nvidia.com/sites/default/files/pubs/2019-11_Latency-of-30/FPS_Tasks_sa2019_AuthorVersion.pdf. Accessed 1 11 2021.