CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Hardware Supported Frame Correction in Touch Screen Systems

For a Guaranteed Low Processing Latency

Master's thesis in Computer science and engineering

GUSTAV NILSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

Hardware Supported Frame Correction in Touch Screen Systems

For a Guaranteed Low Processing Latency

GUSTAV NILSSON

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Hardware Supported Frame Correction in Touch Screen Systems
For a Guaranteed Low Processing Latency
GUSTAV NILSSON

Supervisor: Erik Sintorn, CSE
Examiner: Pedro Petersen Moura Trancoso, CSE

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Description of the picture on the cover page (if applicable)

Typeset in LaTeX
Gothenburg, Sweden 2021

Hardware Supported Frame Correction in Touch Screen Systems
For a Guaranteed Low Processing Latency
GUSTAV NILSSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Devices utilizing touch input have become prevalent in various areas. Touch interfaces are intuitive and easy to use as they attempt to mimic real-life actions such as grabbing and moving objects around. However, the illusion of manipulating real-life objects is broken due to high end-to-end latencies, where objects are visibly lagging behind the finger. A 1 ms latency is required for good touch interactions, optimally calling for 1000 frames per second (fps) content. This thesis investigates creating an acceleration system that adjusts finished rasterized graphics based on the latest touch-input, hiding the processing latency and producing high frame rates. First, a list of the required functionality for the acceleration system is defined. Next, a hardware architecture is designed, which is implemented in VHDL. Lastly, the architecture is tested on an FPGA card connected to a touch screen and a single-board computer running two test applications. The resulting architecture achieves a guaranteed 83μs processing latency in 1024x768 at 120fps while supporting the most common single-touch user interfaces. Predictably, the system can support 1000fps in 4K if implemented as an ASIC with sufficient off-chip memory bandwidth. Despite 1000fps being the long-term goal, the system can potentially provide significant improvements to devices with displays updating at the standard 60-120Hz. Although promising, details on how to best implement the acceleration in a real-life system are yet to be investigated.

Keywords: touch input, latency, high frame rate, computer architecture, computer graphics

Acknowledgements

First, I would like to thank my supervisor Erik Sintorn for helping me throughout the project, always patiently listening and giving feedback, ensuring everything moved in the right direction. I would also like to thank Pedro Petersen Moura Trancoso for listening to my ideas and helping me get this project off the ground.
I want to thank my family for supporting me through all the long days in front of the computer, where I was not very fun to be around. Lastly, I would like to thank Chalmers and all the people running the Department of Computer Science and Engineering for providing me with a fantastic education that helped me understand what I have dreamt of understanding for many years.

Gustav Nilsson, Gothenburg, June 2021

Contents

List of Figures

List of Tables

1

Introduction

1.1 Context

Devices utilizing touchscreens for input have become more and more common in recent years. Their adoption can be seen the strongest in handheld devices such as smartphones and tablets, with increased usage in other areas like public displays and cars. This results from their flexibility and ease of use, which has made them the preferred choice for many types of applications.

Touch is one of the most intuitive user interfaces available as objects are interacted with directly on screen, with no abstraction in between. Therefore, using the device often comes naturally as the interactions mimic the physical world, where objects are touched to be manipulated. These manipulations include selecting what you want, which is similar to how you reach out to grab something in real life, as well as moving objects by touching and dragging.

Although the touch user interface aims to mimic physical objects in many ways, this illusion is broken due to limitations in today's technology. When a physical object is touched or moved, it reacts instantly to the force of the hand by the law of physics, which is not the case on a touch screen. There is a delay between the action being performed and the result being visible to the viewer, caused by the end-to-end latency present in the system: the time it takes to receive the input, use the input to generate a result, and display that result on the screen.

End-to-end latency is more noticeable in touch systems than traditional inputs, such as mouse pointers, due to the presence of a clear reference point [1]. When dragging objects, the eye tracks the finger moving over the screen. This tracking makes it is clear when an element is lagging behind as the distance to the finger visibly changes over time. Even when not noticed directly, it subconsciously tells the brain that the finger is not actually dragging the item. As put by Paul Dietz in the video presentation on their research paper on low-latency touch systems; experiencing low latency touch interactions triggers a "perceptual cliff" where it starts to feel like manipulating a physical object, making the lower latency feel better to the user [2][3].

1.2 Problem

Although modern devices have seen a significant reduction in touch latency compared to older generations, the requirements for an optimal touch experience are vastly beyond what is currently offered. In work performed by Ng et al., it was concluded that a latency even below 1ms in touch applications might still provide perceptible improvements to users, even though they could not try it as that was the limit of their test setup[3].

Achieving a 1 ms latency puts three strict requirements on the system. First is the processing path. It must never take more than 1 ms to use the input to calculate a finished image, which sets very limiting timing restrictions on the OS, the software running, and the GPU. A way to avoid this could be to predict future events, where the processing is started on a touch event before it has happened. However, as detailed in Section 2.2.1, this approach has not been shown to work reliably, in which the 1 ms processing requirement remains.

Second is displaying the finished image. Since modern displays constantly display an image (called sample-and-hold), the less often the image is updated, the more error there will be in the image currently being shown. As the finger moves across the screen, the object will freeze in position until the next frame arrives, in which it returns to the finger again. Therefore, to achieve a 1 ms latency, each image can only be displayed for 1 ms, as after that, the image will be too out of date.

Only showing an image for a short period can be achieved in two ways. The first is to turn off the pixels after the desired time, essentially displaying a black image in-between each frame. Decreasing the time pixels are active has been tested as a successful way of improving the performance of touch systems [1]. However, it leads to a darker image. It can partly be compensated by boosting the brightness of the pixels the moments they are displayed. However, this has several drawbacks. The amount of boosting available is limited while also increasing the power consumption for the same perceived brightness with faster wearing down of the display [4].

The other approach for showing each image a shorter duration is to increase the frame rate, allowing an image to be displayed at all times for maximum brightness. Therefore, for optimal display performance, future touch displays must be fed with 1000 frames per second (fps) content, as concluded by Ng et al. [1]. This 1000 fps need in turn gives the third strict system requirement: the touch input has to be sampled at an equally high rate of 1000Hz to provide a new touch value for each rendered frame.

As refresh rates and restrictions on latency increase, it is not feasible to run the system as usual with all operations done at a higher frequency. Even if possible performance-wise, alternative ways to handle the demands are needed for improved efficiency. When the frame rate increases, the difference between each generated frame decreases. This means running the entire application logic and frame generation from scratch becomes more and more wasteful.

Several approaches to alleviate this have been explored (detailed further in Chapter 2), with strong promise being shown in adjusting the rendered image with the latest input before display. That way, a frame only has to be rendered from scratch a fraction of the times a frame is displayed. This concept has been explored in cloud streaming, AR, and VR, both using software and dedicated hardware, and has been proven to increase frame rates and reduce perceived latency without the proportional performance hit.

As the high frame rate and low-latency demands of those types of systems are similar to those in touch input devices, it could be beneficial to explore similar frame correction ideas in the context of touch input. Due to the often portable nature of touch devices, it is extra important to keep energy consumption low. Running any proposed frame correction on dedicated hardware instead of general-purpose chips could prove to give significant energy savings, especially since touch input is the primary interface and is used frequently when utilizing the devices.

1.3 Objective

This thesis aims to design a system for correcting rendered images with the latest available touch input before display. The actual correction is to be performed in a dedicated hardware unit. The work consist of three main parts:

First, touch interfaces and their commonly used interactions are investigated as a basis for designing a set of requirements for what types of corrections the system should be able to perform. As touch input is completely flexible in terms of what can be done, the goal is to find a feature set supporting most touch interactions in use while minimizing the complexity. Reducing the complexity is essential to provide a meaningful speedup/energy reduction and make any integration with the system simpler.

Next, this minimized feature set is used as the basis for the design of a custom hardware architecture. The hardware will support image corrections based on the latest touch input at high frame rates, working as an independent unit (similar to the previous work detailed in Section 2.2.3.3). The original system, dubbed the host system, is then only responsible for asynchronously providing the acceleration system with the appropriate image and control data to accelerate the user interfaces displayed.

Lastly, the architecture is implemented and evaluated in a test system. The test system consists of an FPGA card accelerating the user interfaces of two custom-made software applications running on a separate host computer.

1.4 Research Questions

The main research question of the thesis is:

> How should a system be designed to provide low-latency touch interactions that are indistinguishable from running the entire system at a very high frame rate with strict timing restrictions?

This question can in turn be divided into multiple sub-questions to be investigated, split into two separate parts. First, the questions regarding the design of the system:

- What type of touch interactions should be supported by an input correction system?
- What features are required to allow for the correction of said interactions?
- How should the image data be structured to allow for the needed image manipulations?
- How should the hardware architecture be structured to support these requirements?

Second, the questions regarding the achieved performance of the proposed system:

- How low latency is achieved?
- Can the suggested architecture support 1000 fps in high resolutions?
- How does the experience compare to traditional touch systems?

1.5 Deliminations

This thesis ignores the latency produced by input and output devices, such as the latency in physically switching the colors of pixels in a display. Those are limited by hardware technology and physics and are out of scope. Instead, the focus lies on achieving low latency in the processing path, using the input to calculate the output.

This processing focus also means that the ability to drive a physical display at those refresh rates is ignored. Instead, the correct pixel values are generated and present in memory for use by any potential display technology. Similarly, the challenges of providing touch input samples at such a high rate are ignored, including any filtering or touchpoint recognition. Instead, finished touch coordinates are assumed to be available for use.

The research on required functionality will discuss and propose solutions for multi-touch displays for multiple fingers. However, the designed hardware architecture and test implementation will only support a single-touch input for simplicity.

1.6 Thesis Outline

In Chapter 2, several related topics are elaborated, including discussing previous work and other methods of improving touch experiences.

In Chapter 3, an overview is given of the touch latency reduction approach taken in this thesis.

In Chapter 4, different types of touch inputs and use-cases are investigated, ending with a proposal of the required features in a touch acceleration system and suggestions for how they can be supported.

In Chapter 5, the generation of final images for display is investigated regarding data requirements and needed algorithms for input processing and rendering.

In Chapter 6, both the work on input design and graphical operations are combined into a proposal for a complete hardware architecture.

In Chapter 7, the properties of the resulting hardware architecture are presented, including the evaluation of a test system implementing the proposed architecture.

In Chapter 8, the results are discussed, and conclusions are drawn.

2

Background

This chapter details previous work in the area of touch systems and relevant work done in other fields.

2.1 Maintaining High Frame Rate

When rendering is demanding, it might not be possible to generate enough images to saturate the display's capabilities. This performance limitation becomes especially true in very high frame rates such as 1000fps, where very little time is left for each frame. As increasing the frame rate of applications and systems has been an objective for a long time, several different approaches have been investigated.

2.1.1 Reduce Demands

One solution is to reduce the computational demands of the rendered image to reach the target frame rate. The entire field of real-time computer graphics is in a sense dedicated to this. Several techniques are utilized, such as level-of-detail assets, where objects far away are replaced with simpler versions. That way, the number of polygons per pixel on screen can be kept relatively constant. However, despite this, a varying amount of elements are present on-screen at each instance, making the frame rate fluctuate. In order to keep the frame rate consistently above a certain threshold, the displayed content must therefore be made simple enough that it can be rendered on time, even in the worst case.

A common technique for achieving a minimum frame rate, utilized in many modern video games, is dynamic resolution, where the resolution of the frame is temporarily decreased in order to make sure it is delivered on time [5]. However, this results in a periodically less sharp image and problems predicting when frames will miss their deadline as the resolution must be decided before rendering. Picking a too high resolution might make the frame miss, while picking a too low resolution makes the image unnecessarily blurry. This technique is also unsuitable for increasing the maximum frame rate, such as in high frame rate scenarios, as it would be equivalent to rendering everything at a lower resolution.

Another approach to reducing the demands is to decrease the resolution in ways

that do not affect the perceived quality. One of these approaches is to render only every other frame in low resolution. The main idea is that the higher frequencies are enhanced in the full-resolution image to compensate for their loss in the low-resolution image. This approach was shown by Denes et al. to give a similar experience to regular rendering, with the rendering demands nearly cut in half [6]. However, this has limitations in the speedup achievable, as every other frame still needs full-resolution rendering. A similar idea is that of foveated rendering[7]. Here, the idea is to only render a small section of the image in full resolution, where the user is actively looking. This approach is especially well-suited for VR, where a large field of view is rendered, with only a small part of the image being in focus at once.

An apparent limitation with decreasing the rendering demands regardless of approach is that the application might go from being GPU bound to CPU bound. As each frame is still generated from scratch, it is possible for the processing between the rendering to become too long for the frame deadlines to be met. This potential issue would be especially true in high-frame-rate scenarios, where the application logic would be evaluated at increasingly higher frequencies.

2.1.2 Interpolation

Another approach for increasing the frame rate is to generate intermediate frames by interpolation, allowing new frames to be generated based on the current and the previous frame instead of rendering from scratch. It can be performed on the GPU, as demonstrated by L. Yang et al. where they use data from two consecutively rendered frames [8]. It is also a widespread technique in TVs to increase the smoothness of the displayed motion. However, this technique suffers from increased latency, regardless of execution, since it is one frame behind, as the next frame is needed before the interpolation can start. It is therefore not suitable for latency-sensitive applications.

2.1.3 Adjust Previous Frame

Another way is to generate new frames based solely on previously rendered frames. One approach for this in the context of VR was suggested by Regan et al., where a complete 360 view was rendered and stored [9]. It allowed for displaying different sections from the pre-rendered image when the viewer looked around by recalculating the memory addresses from which the display is driven. Unfortunately, it only supports head rotations, not translation in space, in which a new image had to be rendered from a different location. This limitation makes it expensive when users walk without rotating the head, as much larger images than used are continuously generated.

Mark et al. instead investigated an approach relying solely on the current view, where the previous image was warped based on the latest camera position [10]. By utilizing the Z-buffer, accurate per-pixel translations could be performed. However, it did not handle the exposure of previously hidden areas where no information is

available. Continued work alleviated these issues by using a secondary reference image, whose information was used to fill in the gaps [11]. However, this approach still has the problem of choosing an appropriate secondary view beforehand. Therefore, it suffers from similar issues as predictive methods, as it can not produce satisfying results when the wrong choice was made.

Apart from using the warping approach during the entire run time, it can also be utilized as a last resort during heavy load. This approach is taken by the Oculus VR headset, which uses frame warping in order to guarantee consistent frame rates [12]. When it senses that a new frame can not be generated on time, it warps the previous frame to approximate how a newly rendered image would look. This approach allows them to maintain fluidity in the motion with no missed frame deadlines. It is worth noting that Oculus fills in the newly exposed areas by simply stretching the image, potentially creating trailing artifacts. To minimize the visibility of these artifacts, the warping is used for at most every other frame and not as a way to boost from low to very high frame rates [13].

2.2 Latency Reduction

The classical way to reduce latency is to optimize every step of the chain to take as little time as possible, which minimizes the whole. However, this has its limits. As each stage does have to perform work, reducing latencies further is not trivial. Therefore, additional techniques have been investigated, which can be used to alleviate the latency issues.

2.2.1 Prediction

The first approach is to try to predict future input. That way, images are generated with the known latency in mind, trying to create an image that will be correct at the time of display. N. Henze et al. investigated this approach by training a machine-learning algorithm to predict where the finger would be in the future, based on previous motion [14]. It improves the interaction considerably, but as with all predictions, it is not perfect, and the object will therefore not behave correctly in all situations.

A clear example of a situation where this approach fails is when the finger is first put on screen. At that time, there is no motion in progress, with the application waiting for an interaction to occur. Only once it happens can it start to act, making it display the usual latency during the start of the interaction.

2.2.2 Frameless Rendering

Another approach to decrease latency is the notion of frameless rendering. The main idea is that instead of updating the entire image at once, a randomized subset of all pixels is updated each screen refresh [15]. After a certain number of updates, the entire frame is replaced without the processing being visible. That way, pixels can

get updated in the middle of a frame update without visible screen tearing. This continuous updating allows for using the latest input at tighter intervals and not just at the switch between frames. However, the downside is the presence of blurring as the object is drawn in multiple locations simultaneously. It is also best suited for rendering approaches where the pixels can be effectively rendered out of order, such as raycasting/raytracing.

2.2.3 Correction

A different approach to latency-reduction is to try to correct the image after it has been generated. By using an input sample from after the image has already been completed, the image can be adjusted so it looks like it was rendered based on the newer input. This approach was investigated by Mark et al. in the context of remote displays, where the user of the system is located far from the server in which the image is generated [10]. By warping the resulting image locally, the round-trip time of sending the input to the server and receiving the image back was compensated.

It is important to note that this approach does not reduce the actual latency of the system, only the *perceived* latency. This distinction means that it allows interactions to feel responsive, while the actual triggering of events in the program code would be equally delayed. For example, moving a file on the screen with the mouse might feel responsive as the image is corrected to the latest mouse position, making the object follow the mouse well. However, placing the file on the thrash can icon would give an equally delayed deletion, as that event has to be detected and processed by the software before it can be displayed.

This image correction can be done in two different ways: on-system warping and a separate low-latency system, as described in the following subsections.

2.2.3.1 On-System Warping

The first approach is to do the warping on the same system in which the data is generated. Typically the process would start with the application rendering the frame on GPU as usual but to memory instead of display. After the rendering has finished, the application sends another job to the GPU for warping the image with newer input as the final step before displaying it on screen. However, as the original frame must finish rendering before the warping can commence, the frame deadline might be missed if the original frame takes too long to render.

A way to run the warping reliably and asynchronously to the regular rendering is preferable to ensure a new image is always generated on time. Utilizing a GPU for this is not trivial as they do not support more than one rendering job and preemptions typically have significant delays. Attempts have been made to run warping on the CPU on a separate core to allow for genuine asynchronous updates [16]. However, this was ineffective, especially from a power and thermal standpoint, due to the CPUs inefficiency at texture filtering.

Modern GPUs and drivers have improved the delays related to preemption, allowing applications to relatively quickly pause currently running work for time-sensitive jobs [17]. This improvement has allowed modern VR headsets to run warping asynchronously on GPU:s. An example of a system successfully doing this is the Oculus VR headset. It uses frame warping to adjust the generated image to the latest head orientation to minimize the perceived latency when looking around [18].

2.2.3.2 On-System Warping

The first approach is to do the warping on the same system in which the data is generated. Typically the process would start with the application rendering the frame on GPU as usual but to memory instead of display. Then, after the rendering has finished, the application sends another job to the GPU for warping the image with newer input as the final step before displaying it on screen. However, as the original frame must finish rendering before the warping can commence, the frame deadline might be missed if the original frame takes too long to render.

A way to run the warping reliably and asynchronously to the regular rendering is preferable to ensure a new image is always generated on time. Unfortunately, utilizing a GPU for this is not trivial as they do not support more than one rendering job and preemptions typically have significant delays. Attempts have been made to run warping on the CPU on a separate core to allow for genuine asynchronous updates [16]. However, this was ineffective, especially from a power and thermal standpoint, due to the CPU's inefficiency at texture filtering.

Modern GPUs and drivers have improved the delays related to preemption, allowing applications to relatively quickly pause currently running work for time-sensitive jobs [17]. This improvement has allowed modern VR headsets to run warping asynchronously on GPU:s. An example of a system successfully doing this is the Oculus VR headset. It uses frame warping to adjust the generated image to the latest head orientation to minimize the perceived latency when looking around [18].

However, the OS, application logic, and GPU are still involved in the generation of each frame. At high refresh rates, this approach puts higher and higher demands on the preemption and control of the system to be certain that frames are delivered on time. Also, the time to switch jobs on the GPU is still not insignificant, taking around 100μs [17]. This delay means that at 1000fps, 10% of the time would be spent to pause the currently running job before the warping can start.

To reduce some of these issues, the image warping could be done on a secondary GPU. This approach was investigated by Smit et al., which utilized two GPUs to generate consistently high frame rates for VR [19]. By dedicating a second GPU for warping and displaying the finished images on display, the first GPU could focus solely on rendering the base images without any preemption or enforced deadlines.

2.2.3.3 Separate Low-Latency System

Another approach is to have a low-latency system updating the screen contents as a complement to the standard high-latency system. This approach allows for a complete bypass of the demands and restrictions in a traditional system, including any additional latencies being introduced. The use of a separate low-latency system has been successfully used in several test systems. Lincoln et al. created an AR system with only 80μs of latency between input and displayed output [20]. They rendered images on a standard CPU+GPU system at 60Hz and then corrected them with the latest input at 16,000Hz on an FPGA card. This dedicated supplemental system allowed them to achieve extremely low latency and high frame rates without directly modifying the original system.

The same approach has also been used for frameless rendering in VR [21]. By using an FPGA performing ray casting to calculate the values of each pixel in a random order, a low latency of around 1 ms was achieved for part of the image.

A separate low-latency system has also been investigated for latency reduction in the context of touch interfaces without image warping. Ng et al. designed a low-latency touch system for demonstrating the perceived effects of low-latency touch using an FPGA [3]. In order to provide similar latency improvements to real touch systems, they propose an approach where a low-latency system draws a simplified UI on top of the original rasterized graphics displaying the latest input. By providing separate low-latency graphics, the users feel the elements' responsiveness without directly altering how images are rendered on the main system. However, drawing a secondary UI on top of the application means that the real application graphics lag behind with equal latency as previously. This creates a discrepancy where the user moves a UI element that the object is hunting instead of moving the object directly.

3

Latency Reduction Approach

This chapter gives a brief overview of the general approach taken to reduce the touch latency produced by system processing, as proposed by this thesis.

3.1 System Layout

Traditionally, a touch controller utilizes and interprets the signals from a touch panel to generate discrete touch samples. These are in turn utilized by the system for input, driving an application. The application calculates a finished image, which is in turn displayed on screen. This process makes the displayed image increasingly more out of date the longer time is spent processing. The approach investigated in this thesis is instead to let the touch controller have the final say over the produced image before it is output to display. The resulting system layout can be seen in Figure 3.1, which shows the proposed system next to the traditional approach. The proposed design makes sure the latest touch input is utilized to generate the final image, bypassing the latency of the host system/application.

This approach shows several similarities with the approach suggested by Ng et al., detailed in Section 2.2.3.3, which utilizes a low-latency system to improve the touch interactions of a high-latency system [3]. However, they suggest providing low-latency feedback in the form of graphical overlays on top of the original graphics to keep the implementation complexity low. This thesis instead focuses on exploring the more implementation complex goal of supporting the acceleration of the rasterized graphics to produce images indistinguishable from ordinary rendering. This goal is to be achieved by letting the low-latency system be the sole driver of the display, generating all final images by altering provided rasterized graphics to create a seamless and unified graphical experience.

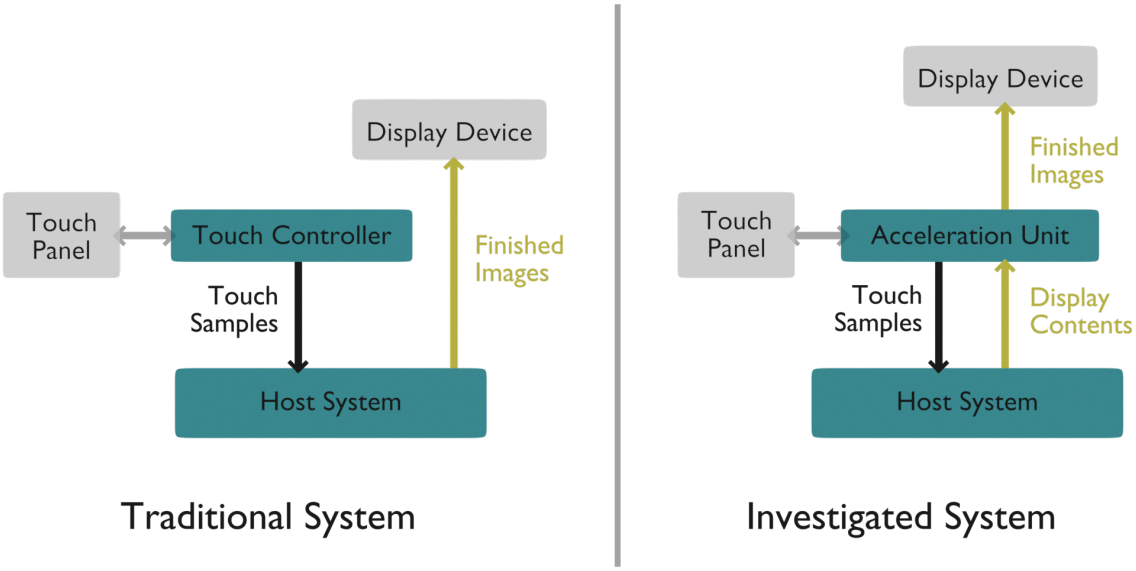

Figure 3.1: An illustration of the difference between the traditional touch system and the proposed approach of focus in this thesis.

3.2 Minimizing Host Involvement

In order to support consistent low-latency interactions, the new system must be self-sufficient and not rely on the host system for making decisions. This requirement means that whatever touch input it receives, it must already have the necessary information on board to provide the correct visual feedback for the action. Any host communication re-introduces the system latency, as the OS and concerning software get involved again, making the interaction potentially as delayed as previously.

Regardless of the type of acceleration provided, the system must still be able to display all types of content. This requirement is supported by letting the acceleration system accept a sequence of images like a traditional display. However, unlike traditionally, the host is no longer required to deliver a frame on every screen refresh. This job is instead relegated to the acceleration system. This approach creates a priority-driven system, where new data is generated when needed, such as the work by Regan et al. on address recalculation [9]. When the currently displayed frame is no longer sufficient according to the host, a new one should be generated and sent for use. If there are no changes in the displayed content, the host application does not need to communicate further with the acceleration system.

In order to reduce energy consumption, the features detailed in Chapter 4 are designed to minimize the number of times the host has to generate new frames. For any interaction not supported by the acceleration system, the normal process of generating and displaying frames at the highest possible frame rate would have to be used. As this essentially bypasses the acceleration system, the usual system latency and power consumption are naturally re-introduced.

4

Interaction Support Requirements

This chapter discusses what is required to improve the touch experience for the user interfaces on modern devices and provides concrete suggestions for how to support those requirements in an acceleration system. This discussion includes the control data needed by any application/unit utilizing the acceleration system.

4.1 Interactions of Focus

All types of actions available on computers can be performed through touch as it is a general input. When programming applications, touch actions can trigger any code execution or events. For example, a button tap on a touch screen can start a long calculation. As these types of events are dependent on unknown program logic and functionality, supporting an acceleration of them would require supporting everything possible in a regular system. This complete feature-support means we would be back at square one again in terms of complexity and therefore latency, meaning they are not suitable for an acceleration system of this kind.

Instead, the focus is on providing the user with instantaneous visual feedback of the action performed, not performing those actions faster. In the example of tapping a button, the tapping would be visualized with no latency, while the calculation's speed is unaffected. In order to provide this, a minimum set of functions is designed to provide instantaneous visual feedback for the most common touch interactions in modern systems. In order to build a general system, any functionality added should preferably support a wide range of general uses instead of specific use-cases.

Note that the inability to run code and generate images limits several common types of visual feedback from being accelerated. An example highlighted by Ng et al. is a UI where the value of a slider is also visible as a number on screen[3]. Even if the slider's position is accelerated to follow the finger, the changing of the number will not be updated with the same responsiveness, as that is a result of processing the input and not the input itself.

However, the inability to accelerate those types of visual feedback should not be of strong concern. The type of latency that is most noticeable for a user is where the user has a physical reference point [1]. When dragging an element using touch,

the eyes track the finger as it moves across the screen. This tracking makes it easy to notice when an element is lagging behind through the distance to the finger, compared to when no physical reference point exists. These issues are easiest to see with 1:1 movements as the distance between the finger and the object should be constant at all times.

The act of touching the elements on screen also suffers from latency sensitivity due to a physical reference point. When performing a high-latency tap, the brain receives an image where the finger touches the screen, yet the object is not reacting. Conversely, after performing the tap, the finger has left the screen, yet the object still reacts or has yet to begin.

Therefore, the proposed system focuses on accelerating these two aspects:

- Touch Feedback - Providing visual feedback when the finger is registered on screen or not.
- Dragging Movements - Trying to keep the correct part of the object under the finger at all times.

4.2 Touch Feedback

Showing visual feedback on the presence of a touch action can be done on two different levels. The first one is to notify the registration of the touch itself. This is, for example, done by Windows 10, which draws a circle under the fingertip when it registers a tap. However, as this is on an OS level and does not involve the application, no information is given on which object is selected or interacted with.

The other level involves the application and therefore gives feedback on which object the touch triggers. This is done by most touch interfaces in one form or another, both when doing quick taps and holding down on an object. The type of visual feedback differs, where common effects include scaling up the object, drawing a border, or fading to a different color. The proposed system should ideally provide visual feedback in a similar manner, where the exact visualization is up for debate.

4.2.1 Effect Type

Considering the proposed system's inability to run program logic and limited knowledge of the displayed content, creating custom shapes or other advanced effects would not be feasible. Instead, a single, general effect applicable for all types of applications should be used. As speed is of the essence, the simpler the operation, the better. Also, the effect should be non-intrusive in the sense that it should fit with the visual style of any application running.

The effect suggested by Ng et al. in their overlay acceleration system is to draw a square border around the interacted object as it is selected [3]. However, as it essen-

tially displays a bounding box, it could prove to be distracting when manipulating non-square objects.

This thesis instead suggests a tinting of the color. By adding a static color to the object being displayed, various effects could be created to suit the software running while retaining the object's shape perfectly. An example could be to add RGBA(0.0, 0.0, 0.2, 0.0) to give the touched object a blue highlight, or to add RGBA(0.0,0.0,0.0,-0.5) to make it semi-transparent. It also allows the application to reveal previously hidden information or elements by adding to the alpha channel, bringing previously alpha zero colors into the visible range.

By tinting on an object level, the amount of data needed is kept low, as it only requires a single extra color value per object. The feature is also easy to enable or disable without any extra data required. Leaving the tinting color at 0 effectively disables it, as the color remains constant, and setting it to a non-zero color makes it visible.

4.2.2 Latency Consideration

In order to achieve the lowest possible latency, the touch indication must be binary, just like the action of touching on most touch panels; the display is either touched or not. By introducing animation, there will be a latency compared to when the touch was first registered, which is why those types of effects are omitted.

A potential exception to this would be pressure-sensitive displays, such as Force Touch on Apple devices (which has been phased out). There the amount of highlighting could be based on the pressure. In that case, there would be a natural fading effect as the pressure increases/decreases and not as an effect of latency-adding interpolation. Another interesting addition would be so-called 3D touch panels, which sense the finger before it touches the surface. In that case, the highlighting could increase as the finger approaches for instant feedback on which object the user is about to press.

4.3 Dragging Interactions

Dragging is when the user touches an item, drags the finger over the display, and releases. The most common form of dragging is moving items on the screen. This interaction shows up in several user interface elements, such as panning, dragging and dropping, and scrolling in lists. Although used in different forms, the core action is the same for them all; items are moved from their original position to a new position on screen. Therefore, supporting them through the same core functionality is suggested to make the system as general as possible.

As most touch interfaces generally perform manipulations of objects in a 2D plane, supporting the dragging of objects in perspective correct 3D space is omitted to reduce complexity and increase performance. This simplification gives a required

functionality in the general case of displacing the selected item along the X- and Y-axis on screen by the same distance as the finger is dragged.

4.3.1 Limiting Movement

The commonly discussed interaction types such as scrolling and dragging elements have another consideration. Usually, there is a limit to how far something can be moved. With a scrolling list, the content can only be moved along a single axis and only for a certain distance (until the end of the list contents). Conversely, when re-positioning individual items on screen, it might be within an area the item is not allowed to leave. Without these limitations, the application logic becomes visibly broken during interaction. Therefore, supporting them is imperative to be able to represent user interfaces faithfully.

4.3.1.1 Supported Limits

In which way these limits should be set and controlled by the application needs to be considered. The more control the application has, the more unique interfaces can be faithfully accelerated but at the cost of increasing size and complexity.

The proposed limit in this thesis is to set a square limitation per object, which states how far it can be displaced in each of the four directions, +/-X and +/-Y. For example, a scrolling list would have a value of 0 in both +X and -X, preventing it from moving sideways. Similarly, +Y and -Y would be set based on the length of the list, preventing it from scrolling further than its size. Static items such as buttons can be created by leaving all limits at a default zero, preventing them from moving.

The square allows wide ranges of uses while keeping the added data to a constant four values. Supporting more elaborate shapes would increase the amount of data needed, but more importantly, potentially make the calculations more complex. If the limits are no longer strictly horizontal or vertical, the limit in X will start depending on the position in Y and vice versa. With squares, the X and Y axis can be evaluated fast and independently. As most user interfaces and object limitations are based on squares, supporting anything more complex is hard to warrant.

4.3.1.2 Collision Detection

There is no object collision proposed in the accelerations system. Doing so would require looking at all the edges of the object to determine the final position, introducing a new level of complexity to the system as individual edges start determining the position of each other. Instead, the proposal lets the host application set the appropriate square limitation to make the object stop in the right location, allowing for fast and easy evaluation in the acceleration system where time is as most critical.

As this approach only supports square limitations, free form collisions become limited, such as dragging round objects against each other. It can also not provide

proper collision against multiple objects simultaneously, regardless of shapes, as the entire allowed area can not be simplified as a single square. These shortcomings are illustrated in Figure 4.1.

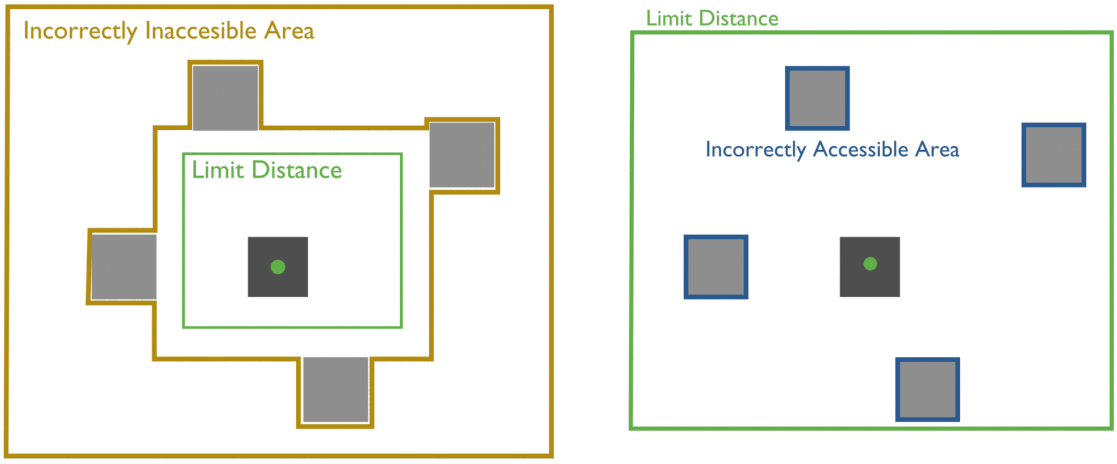

Figure 4.1: An illustration of how the placement of the limiting distance produces varying types of inaccuracies for complex collisions.

Three different approaches are suggested to allow for collision detection in applications using only the square limitation:

The first one is the most reliable. Here collisions are ignored by the acceleration system. Instead, the user is informed of illegal intersections by some visual feedback generated by the application, such as changing the object's color. That way, the object always moves freely while dragging regardless of the shapes present and is never incorrectly stopped.

The second approach also allows free dragging and intersections of objects on screen like previously. However, when the host application notices a collision, instead of notifying the user, it takes over the placement by disabling the dragging and moves the object back out to the closest allowed position. This approach gives only a short initial intersection with increased latencies when the object has been locked outside (as the host application temporarily controls it). However, as the object is no longer following the finger directly, it should be less noticeable.

The third approach tries to emulate real-time collision by quickly updating the distances based on the current object location. As it approaches a new object, the square limitation is adjusted to encapsulate the closest and most likely limitations better. This approach is error-prone with fast movements as the object can temporarily be locked in a position it should not and jump to the finger when the limits have been recalculated. The positive is that visible intersections can never occur.

4.3.1.3 Visual Feedback

When moving an item that gets locked in position due to a limitation, it is important to make the user aware that the position is deliberately limited. Otherwise, a locked position can be mistaken for an unresponsive system, as the object is not affected by further movement.

The exact implementation of visual feedback for these types of events differs. One approach used by different systems, notably certain Android variations, can be seen as a soft limit. They allow the object to, with increasing resistance, temporarily go beyond the limits. For example, it is possible to scroll beyond the contents of a scrolling list, and when letting go, the list returns to the actual limit position. Other systems that use what could be called hard limits lock the object when the limit is reached. They usually use some other visual indicator to notify the user, with methods such as drawing an additional shape at the limiting edge.

One could argue that this feedback could be received with normal latency, letting the host application generate the visual effect when the limit has been reached. However, support for it in the acceleration system is still suggested as it can be implemented with little overhead. By utilizing the already existing tinting color proposed in Section 4.2, no extra data is required from the host application, with little extra work by the acceleration system. The proposed use increases the tinting proportionally by the amount of distance the object is being limited. This way, the user is presented with continuously updated feedback, ensuring that the system is receiving the request. The visual feedback is illustrated in Figure 4.2.

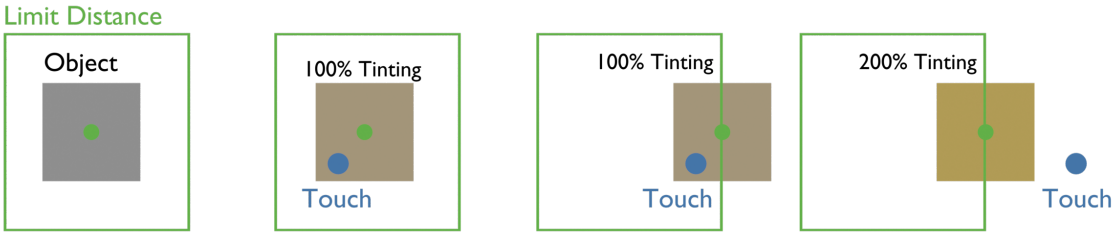

Figure 4.2: An illustration of how the tinting of an object is increased when being locked by the square limits.

4.3.2 Related Items

In several cases, objects are not placed independently but are part of a dependency chain. For example, a window can be moved, where all the content is moved along with it. At the same time, the content within the window can be moved independently. Therefore, items must have the ability to build their position based on others, which is usually done by specifying child/parent relationships.

When evaluating the position of an object, the position of all of its parent must be considered, as each parent's position affects the final position. The deeper the dependency chain, the slower the evaluation becomes as more parents have to be

evaluated. This variability can be problematic in a latency-sensitive system. Therefore, support for a limited four depth dependency is suggested, giving a consistent performance. This depth supports an item within a window, within a window, within a window, which should allow for the acceleration of the most common use cases.

4.3.3 Windowed Content

Computers have long supported the moving and placement of separate windows on the same screen. Similarly, most applications are divided into multiple areas, each displaying independent content. An example would be a scrolling list only partially covering the screen. The currently visible content in the list is allowed to be dragged outside of the area, but by doing so, it should become invisible.

The suggested approach for letting objects be visible only in a particular section of the screen is to store a square per object, denoting a window in which it is visible. This functionality will not allow any type of complex shapes, such as rounded corners, but it should be sufficient for most UIs.

Areas and windows resizing is a common user interaction, commonly on computers and increasingly on smartphones, such as when utilizing a split-screen mode. Supporting this directly on the acceleration system is not suggested, as one resize could affect any number of viewable areas through a shared edge. Instead, an approach is suggested that resembles how many user interfaces currently visualize resizes: by only visually moving the edge until it is released. This way, the existing windows and their content does not need any modification mid-resize. It allows the user to see where the new area will reside, and as modern interfaces typically rearrange the elements when resized, providing live updates of the viewable area on the acceleration system provides little benefit.

Lastly, to properly support windowed content, it must be possible to lock the window area to dragged objects, keeping the same section of the object visible. For example, moving a window around should not affect which part of the scrolling list inside is visible. Supporting this is suggested by setting a count for how many parents it should follow, from none to all three. As the parents can be assigned in any order, the count allows the window area to follow any combination of parents.

4.4 Other Considerations

This section details and discusses other considerations needed when replicating common touch user interfaces.

4.4.1 Touch Areas

There is a need to support touch areas that do not align with the content being displayed. This is commonly being used for increasing the touch area for easier

selection of small elements. It can also be used for the opposite, with only a small part of the visible element acting as a touch area. An example of this is a window, where only dragging on the top bar moves it, while nothing happens if touched anywhere else.

The suggested approach to this is to separate viewable and interactable areas. Then the host application can state which areas can be seen and which takes interactions. These areas should be allowed to overlap, so the drawn items do not need to be split up into separate parts based on where the touch areas are. Instead, two or more unique areas can be defined separately while still belonging to and affecting the same object.

4.5 Implicitly Supported Interactions

This section shortly describes some common interaction types supported by the already established functionality in the previous section.

4.5.1 Hold and Drag

Hold and drag is a type of input used to initiate a dragging of a currently static element. It is done by holding the element for a short while until it signals that it can be moved. This signal is usually similar to the ones used to visualize the touching of an object, such as an increase in size. As the holding per design is static, with no movement, it is not latency-sensitive. Therefore, the acceleration system has no support for this kind of interaction. Instead, it is up to the host to detect the hold, and after the appropriate time, send a new frame with a visual cue and dragging activated for the object. The only requirement from the acceleration system then is to allow for the dragging limits to be updated mid-touch, making the locked object free to move.

4.5.2 Initiated Directions

Another common user interface is locking the direction in which something can be moved based on the initial motion. For example, starting the dragging by moving to the right will lock the object to only move along the X-axis. This type of interaction can be seen in applications such as Excel.

Although not critical to be supported, it is worth noting that such functionality can be replicated reasonably well. The object starts with the dragging limits set to the maximum, essentially disabling the limiting. As the host application notices the direction in which the dragging is happening, it can smoothly shrink the limits along the appropriate axis to zero through continued frame updates, bringing the object back to the line in which it should move.

4.6 Omitted Interactions

This section shortly describes common interaction types which are not suggested to be supported directly by the acceleration system. They are omitted due to implementation complexity in combination with the availability of a sufficient alternative.

4.6.1 Temporary Static Positions

A common use case for touch is the ability to drag something around and have it jump into pre-designated positions, assisting the user in the final positioning of the object. As the finger approaches a valid position, the object jumps to it and locks in position. When the finger moves away, the object returns to follow the finger. The sequence is demonstrated in Figure 4.3.

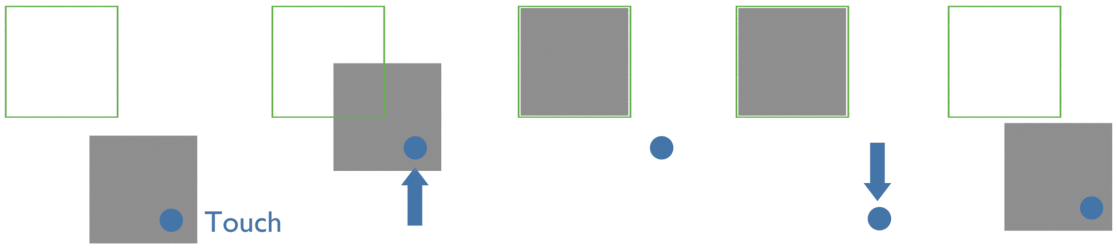

Figure 4.3: An illustration of how object are temporarily moved to the closest accepted position.

Although visually clear and pleasing, it can be replaced with a different visualization that serves the same purpose. The area to which the object would move can be indicated by drawing an additional object, creating a kind of highlight. As this is not latency-sensitive, it can be performed by the host.

4.6.2 Touch to Focus

A common interaction is touching an object to bring it into focus, which could be done by changing its look or bringing it to the front of the screen if other objects previously overlapped it. This is commonly used when multiple windows are displayed, in which the selected window is brought to the front. These actions are not especially latency-sensitive as they are not a direct result of what is done on screen. The object position in X and Y can still correctly follow the finger on the screen, and the selected object can still be highlighted through tinting, even if the focus aspect is delayed. As these actions are not trivial to support, having to reorder objects and defining a secondary look, it was decided not to support it in the acceleration system. Instead, it is up to the host application to rearrange and draw the items appropriately before sending a new frame, letting the focus change appear with a slight delay.

4.7 Future Interaction Support

This section describes common interactions not supported by the system designed in this thesis but should be in a real touch acceleration system. Included are short suggestions for how support for these interactions could be added in a future implementation.

4.7.1 Inverse Parenting

A common case in user interfaces is inverse parenting, where dragging on an item moves the entire area. For example, in a scrolling list with options, each option can be selected independently. However, scrolling on the same option should scroll the entire list, including all other elements. Therefore, it must be possible to scroll on a child in order to affect its parent. However, letting a parent be affected by all children is potentially costly from an implementation perspective, as one object can potentially be affected by an unbounded number of objects.

This behavior is instead suggested to be achieved by continuing the search for an interacting object whenever the user drags on top of an object that lacks dragging support (dragging limits set to 0). That way, when the user selects an object by touching it, the object will be highlighted. When the user starts dragging the finger, the search for a new object is started, one that allows for dragging. In the scrolling list example, it would be the list object behind the selected option, switching to dragging the entire list. Note that this behavior is not always wanted and should therefore be settable with a flag.

4.7.2 Continuous Object Selection

The system also needs to support continuous object selection. For example, when playing an on-screen piano, the user should be able to drag the finger over all the keys, triggering them in sequence. This can be achieved through the suggested functionality in the previous subsection, searching for an object accepting a dragging interaction. As long as the object currently being dragged does not support dragging, the search for a new selection will continue, making sure key after key will be selected and highlighted. However, for proper support, it must also be possible to prevent objects that are behind from being selected. For example, there could be other elements present behind the keyboard which should not react. This indicates the need for a second flag, limiting the search to only visible objects residing on the top layer.

4.7.3 Static Selections

When selecting an object on the screen, the selection should occasionally be kept even after the finger is lifted. An example is when marking multiple photos for sharing. Each selected photo should be highlighted with no delay and remain highlighted until it is deselected again. Support for this can be achieved through a flag stating

that the tinting color should be kept after the object is no longer touched. Without support built into the acceleration system, the highlighting would disappear for a short time between when the finger is lifted, and a new frame showing the highlight arrives from the host, creating a flicker.

The proposed flag can be used in other contexts as well, such as when using toggles for the wi-fi or similar settings. It could also reveal additional controls with a tap, such as play controls when viewing videos, as the kept tinting allows the revealed controls to remain on screen indefinitely.

4.7.4 Momentum

Another consideration that must be made is the presence of momentum. When an object is moved and let go while the finger is in motion, the object should usually continue in its path instead of halting directly. This is commonly used in different user interfaces, for example, when scrolling in lists, where a fast flicking motion makes the list scroll a long distance. This continued motion can not be handled by an animation on the host as the object will freeze in place until the animation arrives. Instead, the animation must be taken over by the acceleration system directly on release. Similar to the discussion in Section 5.6.5, this requires the acceleration system to keep track of a temporary transformation being applied to the object. Furthermore, this transformation must now continue to be updated every frame for continued smooth movement.

For simplicity, a simple value that states how long it should take for the object to halt is suggested. A longer time creates the appearance of stronger momentum. The current behavior of no momentum is supported by setting the time to zero. Although different fall-off curves might be in use in different applications, this should be able to be replaced with a single one for simplicity, with little user experience being lost.

4.7.5 Non-1:1 Movements

Some touch interfaces include dragging movements where the item is not following the finger 1:1. This is used to shorten the distance the user has to drag the finger, or the opposite, to require large movements to trigger actions, such as bringing in a small menu from outside the screen. These movements are less latency sensitive than 1:1 movements due to the non-constant distance between the finger and the object and are not of immediate priority. However, as these interactions are relatively common, especially in gaming where the camera can be panned around by small finger movements, support for them is recommended.

The suggested implementation is to store a finger influence factor per object, stating how many pixels the object should be displaced per pixel moved by the finger. The value would be set to 1 by default for regular touch interaction.

Although not directly crucial for the touch latency experience, it is worth noting that the finger influence value also allows simulated 3D effects in the form of parallax

movements to be performed on the acceleration architecture. As objects follow the finger more or less strongly, they will move in relation to each other, giving an illusion of depth to the user interface. Supporting this alleviates the host application from having to animate such effects frame by frame.

4.7.6 Non-translational Dragging

There are two common special cases of dragging elements where the entire object is not moved uniformly in the direction of the finger: scaling and rotation. Combined with the translation discussed previously, they cover all three standard displacement operations in two dimensions.

Scaling elements are used for various interactions, with common examples including zooming in/out of a map or a photo. A scaling operation is dependent on an origin point, which determines how the different parts of the object are displaced on the screen. A touchdown closer to the origin point gives a stronger scaling effect for the same distance moved. In figure 4.4, the effect of different pivot points has on the same touch interaction is illustrated. In order to support all types of scaling, the ability for the host to set the origin point on each object is required.

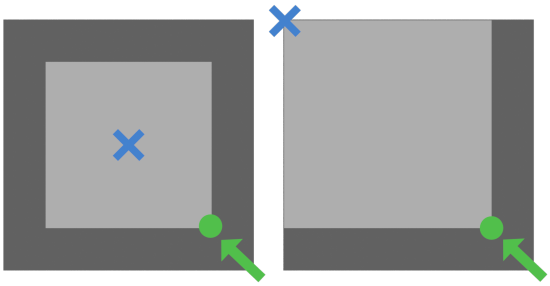

Figure 4.4: An illustration of how the pivot point affects the transformation when scaling a square. The pivot is shown in blue, and the dragging motion in green.

Note that it is important for the object to have a limit in the allowed scale. This limit could, for example, be the largest and smallest zoom on a map. As the scaling happens at different speeds depending on the touchdown point, the dragged distance can not be the limit. Instead, it must be on the actual size of the object. Therefore, the ability to set the scaling limitation as a range of allowed sizes is suggested.

Similarly, rotation also depends on the position of a pivot point. Rotation is the most complex motion, as the location of the points is no longer linearly driven by the touch. Instead, it becomes a function of trigonometric functions, which would require more advanced calculations in determining the final object position. Rotations must also be able to be limited by the host to replicate different user interfaces. The suggested limitation is setting a range of angles from the current rotation, allowing for up to +/-360 degrees.

Lastly, dragging on different areas should support starting different interactions,

such as translation or scaling. That way, a single object can be modified in different ways, even with single-touch, where different areas or buttons can be used to select the action. Therefore, this thesis suggests setting flags on an area basis stating which action it supports, dragging, scaling, or rotation. Note that scaling and rotation should be able to be performed simultaneously in a single-touch context since those interactions can be done in a single motion with a set pivot point.

4.7.7 Multi-touch

Multi-touch is a common part of most touch panels today and must be supported by an acceleration system. Luckily, this can be done with only minor modifications to the already specified functionality.

4.7.7.1 Interaction Types

Multi-touch interactions can essentially be divided into two types. First is multiple parallel single-touch interactions, for example, pressing on multiple keys on a piano on the screen at once. It is the same action as the single-touch, only being performed by multiple fingers simultaneously. The ability to support this is straightforward. The system needs to replicate the functionality so multiple events can be performed in parallel. This means that when objects are checked for interactions, more than one object can get selected, one per finger, and as such, the visual feedback for more than one object is performed.

The other type of multi-touch interaction needs more consideration; the multiple-fingers, single action. An example of this is using two fingers to rotate an image. Here the presence of more than one finger alters the action it performs, combining them into a single unique action. Support for this is suggested in the form of setting an allowed finger count per touch area. That way, different overlapping touch areas can be defined, providing different actions when different finger counts are present.

4.7.7.2 Finger Count Support

A touch application might use any number of fingers. However, there are essentially only two types of finger counts: the single touchpoint and the dual touchpoint. As the interface is in 2D, more fingers than two do not provide more information for manipulation. Instead, actions with three or more fingers are a way to provide multiple actions on the same area, such as the four-finger swipe up to go home on the iPad.It is essentially a single touch movement triggered by a certain finger count.

With a multi-touch display, scaling, rotation, and translation can all be performed using two fingers. Supporting this requires some additional support, as another touchpoint now controls the resulting rotation and scale at the same time. Support for this is suggested by letting the first finger act like during a single touch input while the second finger sets the pivot point. Then the only difference between multi-touch or not during manipulation is whether the pivot point is static or locked to a finger position.

The ability for the second finger to modify the pivot point allows for translation, rotation, and scaling to happen simultaneously. As multiple actions are now possible on the same touch area, it must be possible to allow or block these individually in cases where only one or two of them should be possible at once.

4.8 Summary

Concluding the chapter is a summary of the operations the system should support, with the proposed solutions on how to achieve them, available in Table 4.1. The interactions listed in green are supported in the test system made in this thesis, while interactions in orange are suggestions for a future real touch acceleration system.

Table 4.1: A summary of the requirements for input correction with the related proposed features for supporting them.

System Requirements	Proposed Solution
Interacting with multiple elements independently	Keeping separate objects
Tapping/Holding feedback Movement Limit Feedback	Color tinting support
Dragging Objects	Activate on an area basis
Limit placement of objects	Assign each object a square distance limit
Variable touch areas	Enable visibility and touch on an area basis
Placement based on other objects	Storing up to three parents per object
Inverse Parenting Continuous Object Selection	Search for a dragging-accepting object
Static Selections	Keep color tinting
Momentum	Halting time Multiple in-progress transforms
Non-1:1 Movement	Finger influence
Scaling Objects Rotating Objects	Per touch area pivot point Activate on an area basis
Multi-touch	Multiple single-touch actions in parallell Accepted finger count per touch area Lock pivot point+position to second finger

5

Generating Images For Display

This section details the specific algorithms and processes used to generate the final images for display. Consequently, this also determines large parts of what data will be required from the host application to support the operations.

5.1 Touch Data

When the finger first touches the display, the X and Y coordinates of the finger are stored as the initial touch location, called $X_{touchdown}$ and $Y_{touchdown}$. The current finger position is also required, called X_{pos} and Y_{pos}. As the finger is dragged, the current finger position is continuously updated while the touchdown values remain static as long as the finger remains on the screen.

The distance the finger has been dragged on screen is calculated as the difference between the touchdown and the current position:

$$X_{dragdistance} = X_{touchdown} - X_{pos}$$

$$Y_{dragdistance} = Y_{touchdown} - Y_{pos}$$

All of these values are in screen space, independent of any object. The same touch location on the screen results in the same coordinates regardless of the screen contents, making the values applicable throughout the system.

Note that the touch data is stored in the same resolution as the displayed content. Therefore, all touchpoints and distances are in a number of full pixels.

5.2 Required Frame Data

Traditionally when frames are sent for display, typically from the GPU, the frame consists of a single bitmap (a grid of pixel colors). No extra data is provided regarding the frame contents, limiting the types of operations that can be done post-rendering. In displays such as TVs, where several operations are supported, all are

based solely on analyzing the pixel values of the arriving frame. This approach can lead to issues in the displayed images, such as artifacts during motion interpolation.

5.2.1 Individual Objects

As detailed in Chapter 4, the touch acceleration system must support direct interaction with multiple on-screen objects. This requirement means that the system must know which parts of the frame belong to which independent object. Even if the objects themselves can be separated visually by analyzing the bitmap, extracting correct interaction rules, such as if the object can be dragged, would not be possible. As the program logic can differ even for identically produced images, achieving such a feat would be akin to reading the programmer's mind.

It is clear that sending additional data is needed. In order to allow different parts of the image to behave differently, the interaction rules must be connected to the specific areas of the image in which the objects reside. The suggested approach for specifying these areas is through triangles, which are simple and can approximate any shape efficiently.

Note that the positioning of the triangle vertices on-screen is in a pixel resolution, without sub-pixel precision or floating-point values. This resolution is adequate for two reasons. First, the touch data is provided in a pixel resolution, meaning no displacement of the triangle can place it in a non-pixel aligned position, as detailed in Section 5.6.1. Secondly, the triangles are only utilized for denoting which pixel on screen belongs to which object, which is by its nature on a pixel level.

5.2.2 Revealing Previously Hidden Content

For touch feedback, the information of the location of each object on the screen is sufficient, as the color of the appropriate area can be tinted. However, the operation of dragging items provides additional difficulties. As an object is moved, it should reveal previously hidden content. Handling the newly exposed areas can be done in two ways. The first is to guess the area behind, through a simple extension of the edge colors or more complex operations. However, regardless of the method, it will not be perfect. Artifacts will be apparent, especially when a new frame is received from the host, as the approximation is visibly replaced with the correct values.

Even more challenging is the case of semi-transparent objects. As the content behind is visible through the object, simply moving the object's pixels would incorrectly move the content behind as well. Anti-aliased edges, which also depend on transparency, would similarly be an issue since the background color is already multiplied into the object's pixels. Therefore, support for multiple alpha-enabled bitmaps is suggested instead of the traditional single bitmap. By making each object an independent image, all objects can be transformed while having the necessary data for correctly revealing the content behind it, including semi-transparent areas.

5.2.3 Bandwidth Implications

The number of images required depends on the number of individual objects on the screen that can be dragged. As they should be able to move independently, each is required to be stored as a separate image. However, the actual number of images has little effect. Instead, the total area of the images determines the needed bandwidth for transferring a new frame from the host. In the best case of no draggable objects present, a single screen-covering bitmap is required, equaling the non-accelerated system. In the worst-case, a complex frame containing multiple large, overlapping items, the data size is increased multiple times. However, the bandwidth increase can be mitigated by only sending new images when the look of an object has changed since the previous frame. Due to the often static nature of user interfaces, this allows for considerable bandwidth savings.

Another possible bandwidth-saving approach is waiting and seeing which object is selected for interaction and then sending the required object as a separate image from the host. This approach would limit the number of images to a background, the modified object, and a foreground. However, this leads to the start of the interaction not being low-latency as the host needs involvement to initiate the action. Therefore, it is suggested to keep all draggable objects separate on the acceleration system at all times.

5.2.4 Ordering

Traditionally when rendering images, all objects have a different depth, making it possible to accurately determine the visibility of objects on the screen in a perspective correct manner. However, as the acceleration system only focuses on 2D, the actual depth is of no interest, making the traditional use of a floating-point number to denote depth unwarranted. Instead, the only importance is their internal ordering: which object is in front of which.

Internal ordering can be achieved by placing objects among a set of discrete layers. When rendering the final image, objects are drawn such that objects on higher layers are in front of those on lower layers. However, this approach makes it difficult to add and remove objects. If there are no layers left between two layers in which an object should reside, multiple objects have to switch layers to retain the correct internal ordering.

Instead, an implicit depth ordering based on the order in which the objects are stored in memory is suggested. When objects arrive from the host, the acceleration system treats the first object to arrive as closest to the front, with the next object falling behind it, and so on. That way, no depth has to be explicitly stated, neither during transfer nor storage.

Note that requiring the host to sort all triangles before sending increases its workload, which could not be motivated solely through a slightly reduced data size alone compared to storing a depth explicitly as in the 3D case. However, all triangles

already arriving in depth order also reduces the complexity and increases the speed of the acceleration system, both when handling interactions, discussed in the next section, as well as when rendering, discussed in Section 5.6.4.

5.2.5 Identification

As an object can consist of multiple triangles, it must be possible to refer to the rest of the object through a single triangle. That way, a touchpoint can be detected in one triangle, while the actual interaction is performed on all triangles in the object, acting as a unity. The principle is shown in Figure 5.1.

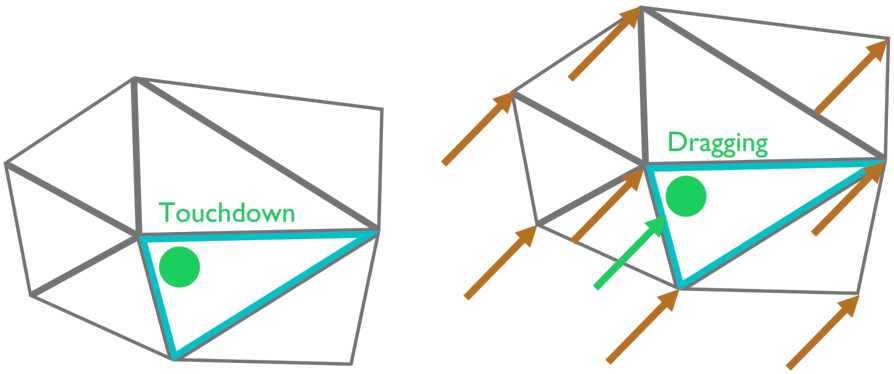

Figure 5.1: An illustration of how touching a triangle affects the entire object.

Triangles can be grouped to form objects in two different ways. The first approach would be to let the host signal the start of a new object as the triangles are transferred. For example, three triangles are sent, followed by a "new object"-flag, ending with five more triangles. This transfer results in two objects, consisting of three and five triangles, respectively.

When interacting with a single frame, this would work well. However, when generating the next frame, the object order of the previous frame would still have to be retained. Otherwise, the interaction currently in progress might start to refer to a different object, switching which object is affected on the screen. This limitation requires the host application to make sure the object stays in the same position order-wise, hindering its ability to add and remove screen contents freely.

The suggested approach is instead to use explicit Object IDs to identify and refer to objects, slightly increasing the data size that needs to be transferred. More importantly, it increases the administrative overhead as the host has to assign and keep track of unique IDs. However, it does allow for reliable referencing between frames for any number of objects. That way, objects can be added, removed, or reordered at any time without interfering with any interactions currently in progress on the acceleration system.

5.2.6 Frame Data Example

An example of how a common touch application, Google Maps, could be represented on the accelerating system is available in Figure 5.2. The triangles are colored in depth order, going from red in the front to blue in the back. Note that all rounded corners are represented through the transparency of the object image. More triangles could be used to represent the area better for a closer touch area. However, as the touch area is typically enlarged, extending outside the rounded corners should not be an issue.

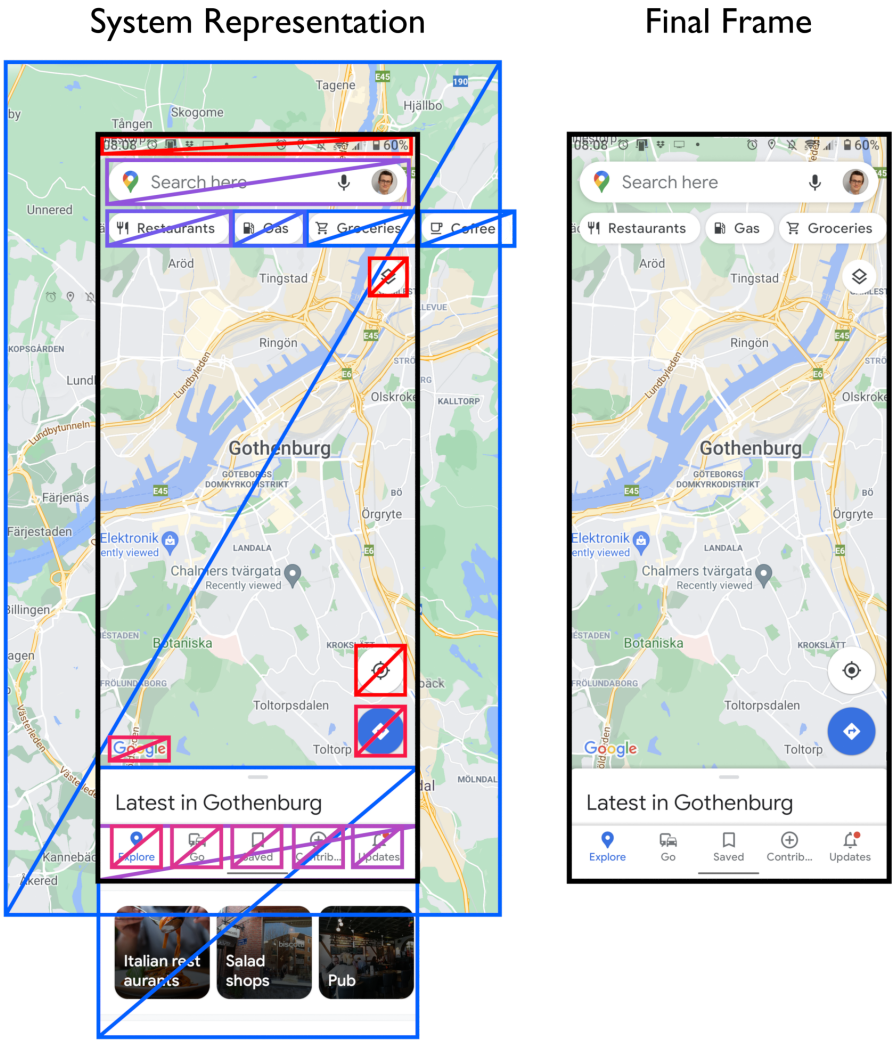

Figure 5.2: An example of how the touch application Google Maps could utilize the suggested acceleration system. The data as sent and stored on the acceleration system is shown to the left, with the resulting final image for display from the acceleration system shown to the right.

5.3 Object Selection

When an on-screen interaction is started, the first step is to determine which object the user is interacting with. This information allows the acceleration system to

apply the interaction to the correct part of the image. As the areas of the objects are defined through triangles, object detection will be done through point-in-triangle tests with the touchdown coordinates $X_{touchdown}$ and $Y_{touchdown}$. Such tests are used in two places in the acceleration system and are explained in detail in Section 5.4.

When checking for a touch intersection, two extra considerations must be made. First, it should only check triangles that are part of a touch area. Secondly, each triangle has a visibility window defined, allowing it only to be drawn within a particular area, as described in Section 4.3.3. The part of the triangle not currently visible should not be open for interactions as the user must see what is selected. This is handled by checking the right, left, top, and bottom of the visibility window and see if the touchpoint lies inside all four limits.

The algorithm for finding the touched object is as follows:

Algorithm 1: Finding the Interacted Object

```
for all triangles do
    if (the triangle is a touch area) and (the point is inside the triangle) and
      (the point is inside the visible area) then
        return triangle objectID;
    end
end
```

As the triangles are stored in depth order, front-to-back, the first triangle with a valid intersection is the one closest to the viewer. Therefore, the algorithm can exit on the first hit, as no later triangles could have been touched. In the worst case, all triangles have to be checked for intersection, giving an O(n) complexity.

Multiple triangles could be checked in parallel to improve the performance, as long as it retains the property of the touched triangle first in order always returning before any later triangle. However, due to the relatively low number of triangles present, the sequential algorithm is used, checking all triangles one by one.

5.4 Point-In-Triangle Test

The chosen algorithm for checking if a point is within a triangle is based on the line equations of the triangle edges [22]. An approximation of the distance between the point and the edge is calculated, disregarding the magnitude. Instead, the sign is of interest, being plus or minus depending on which side of the line the point resides. A result of 0 means that the point lies directly on the line. If the touchpoint distance has the same sign for all three triangle edges, the point is inside the triangle. Note that this requires the vertices to be defined in a specific order, with clock-wise ordering utilized in this system.

A line is defined as two points, allowing the vertex positions to be used directly. The two edge points are denoted x_1, y_1 and x_2, y_2 while the touch down point is

denoted x and y. The resulting formula for checking the point against the edge is:

$$(x - x1) * (y2 - y1) - (y - y1) * (x2 - x1)$$

This computation is performed once per edge. When the results of all three computations are positive, the point resides inside the triangle. The algorithm is computationally inexpensive, requiring fifteen subtractions and six multiplications in total per test, without any divisions, making it suitable for hardware acceleration.

5.5 Rendering Approach

In order to generate final images for display, a framebuffer will be used. It allows the final pixel values to be calculated step-by-step, as the pixel value can be continuously updated. This approach simplifies the design as all triangles contributing to a specific framebuffer pixel do not have to be searched for and considered simultaneously. Instead, each triangle can be handled independently in sequential order, continuously updating the framebuffer until all triangles have been processed.

The general approach follows that of a traditional GPU. First, the triangles are processed to determine their final on-screen position. Next, they are rasterized into final pixels in the framebuffer, which is then displayed on the screen.

5.6 Triangle Processing

The first step of rendering finished images on the acceleration system is to process all stored triangles belonging to the latest received frame from the host.

5.6.1 Displacement

In a typical GPU, the vertices are displaced before rendering through vertex shaders, which execute code that calculates the final position. In this system, the complexity of arbitrary code is avoided by instead specifically supporting the functionality described in Chapter 4 through dedicated hardware. At its core, this consists of displacing the triangles based on touch input, moving them into a new position on the screen.

First, each triangle must be evaluated if it should be affected by the touch input by checking if the triangle belongs to the touched object directly or if it is one of its parents. To premier evaluation speed over space, each triangle holds the ID to all of its parents directly. That way, no searching or jumping between objects is needed during evaluation to determine if an object is being affected.

In total, four IDs are checked: the ID to which the triangle belongs, as well as the ID of its three parents. If either match, the triangle should be affected by the

touch input. This approach works well since it does not matter which object in the dependency chain is dragged; if either of them are, the object should move.

An unaffected triangle is left untouched, using its stored vertex positions directly:

$$X_{vertex} = X_{vertex\_base}$$
$$Y_{vertex} = Y_{vertex\_base}$$

An affected triangle is displaced by the same distance as the finger, available as $X_{dragdistance}$ and $Y_{dragdistance}$. However, as the user is not allowed to move all objects freely, the dragging distance is first limited against the dragging limits, as detailed in Section 4.3.1.1. The dragging limit of the touched object is used, which makes sure all objects are limited by the same amount, regardless if the triangle is part of the object directly or through parenting. The final dragging distance is given as follows:

$$X_{dragdistance\_final} = \min(\max(X_{limit\_min}, X_{dragdistance}), X_{limit\_max})$$
$$Y_{dragdistance\_final} = \min(\max(Y_{limit\_min}, Y_{dragdistance}), Y_{limit\_max})$$

It is then added to the base position of all vertices for the final position:

$$X_{vertex} = X_{vertex\_base} + X_{dragdistance\_final}$$
$$Y_{vertex} = Y_{vertex\_base} + Y_{dragdistance\_final}$$

Note that since the final dragging distance is identical for every object, all affected triangles will be moved as a unit without distortion.

5.6.2 Using the Latest Touch Input

Whenever the touch input is updated in the middle of rendering a frame, a choice must be made on which input to use. The first option is always to utilize the latest touch input, making sure the triangle position is as up-to-date as possible for minimal latency. However, this can generate screen tearing as the triangle might have been different when the previous part of the frame was rendered. A way to avoid this is frameless rendering, as discussed in Section 2.2.2.

The other option is to lock the touch data at the start of a frame, making sure the locations of all objects are consistent across the entire frame, mimicking what is commonly called V-sync. However, this gives the bottom of the frame a higher latency than the top as it is drawn later. Nevertheless, this approach is used by the proposed system, as the goal is to provide enough images per second to display that mid-frame updates are not needed to achieve the latency goal.

5.6.3 Culling

Triangles should be discarded as soon as it is known that they will not contribute to the finished image to reduce unnecessary work. A triangle is guaranteed not to contribute to the image if it is outside of the area that is being rendered since no reflections or similar effects are utilized. If a triangle is within the rendered area or not can only be determined after the displacement has been finalized, in which they can be safely culled if they are outside. The triangle is culled if all three vertices are either above, below, to the left, or to the right of the rendering area, guaranteeing it can not be visible.

Traditionally, back-face culling is performed in 3D graphics as an optimization not to render triangles known not to face the camera. This culling is done by looking at the ordering of the vertices within the triangle. Although a vertex ordering is present for efficient intersection testing, detailed in Section 5.4, it is not utilized for culling. As all graphics and operations are two-dimensional, there is no case where a triangle could rotate on the acceleration system, going from being front-facing to back-facing. It is instead up to the host to only send triangles that should be displayed.

5.6.4 Ordering

As the system should support transparent objects, the order in which the triangles are rendered becomes essential. The traditional way to render with correct transparency is to render the objects in depth order, back-to-front [23]. This approach is called the painter's algorithm, as it resembles how a painter adds layers to a painting for a final result. Traditionally, the painter's algorithm is only utilized on transparent triangles, as it requires them to be sorted. For all opaque content, a Z-buffer is used to determine which parts of the triangles should be visible or not in the final image.

In this acceleration system, all triangles arrive in depth order from the host, as detailed in Section 5.2.4. This ordering makes transparency correct rendering through the painter's algorithm free, performance-wise, on the acceleration system itself. There is also no need for a Z-buffer, as no triangles will be drawn out of order. This approach also makes sure every triangle on screen has a unique depth consistent between frames, avoiding race conditions on which triangle is drawn in front of the other.

One issue with the painter's algorithm is that large portions of what is rendered end up being completely overwritten when opaque triangles overlap. Therefore, the system will require the triangles to be ordered front-to-back to minimize unnecessary work, as a later triangle can not cover already drawn triangles. Supporting this ordering requires each framebuffer pixel to hold an additional coverage value, stating how see-through it is. That way, it can be determined if a pixel is fully opaque and already completed, in which all further writing to that framebuffer pixel can be discarded. The use of coverage requires slightly more space compared to back-to-

front, with the significant benefit of fewer framebuffer pixels having to be calculated and overwritten.

5.6.5 Keeping Positions

As the triangles have been moved to a new position, they should remain there after the finger has left the screen not to create any visible jumps. This requirement means that the acceleration system must retain the new position after the action has been completed. Achieving this is trivial without further interaction as the current state can be kept. The problem comes when a new action is initiated, as two non-default transformations must be kept in memory to retain the on-screen view. More objects in sequence result in having to keep track of more objects.

These positions can be kept in two ways. The first is to alter the stored positions of the triangles in memory, essentially creating a new base position when the action completes. However, as new frame data arrives from the host, it could potentially overwrite the values stored by the acceleration system. The generated information becomes especially difficult to retain when the content changes between frames.

The second approach is only to support a set number of interactions in a row. That way, the current transformations can be kept separate for fast access, allowing new frame data to arrive freely from the host. As only a set number of temporary transformations can be active simultaneously, the host becomes responsible for baking the interactions into the new frame data. When a new frame arrives with the dragged position included, the temporary transformations must be cleared through a flag to avoid applying the same transformation twice.

This approach is used by the proposed system, supporting a single dragging interaction in a row. After the finger is released, it is up to the host to update the frame data, making sure the acceleration system is ready for the next interaction. Since the human hand can only move so fast, there should be more than enough time for the host to generate a new frame between the time the finger is lifted and returned. The acceleration system should not accept any new touch input until the handled flag is received from the host to ensure no incorrect behavior is displayed.

Note that for more advanced features, such as proposed momentum and multi-touch, there is a need for the acceleration system to correctly handle more than one interaction in a row without the assistance of the host. The number of interactions can also be increased to improve the resilience of a temporarily non-responding host system.

5.7 Rasterization

With completed triangles available, the next step is to convert them into pixels on the screen. This conversion is done through a rasterizer, which takes a vector shape and converts it into a pixel grid.

5.7.1 Affected Pixels

The process of converting a triangle into on-screen pixels relies on finding which pixels the triangle covers. This evaluation has traditionally been done by traversing the triangle edges, filling in the affected pixels row by row. However, that suffers from being hard to parallelize efficiently.

Instead, edge equations are utilized, which allows all pixels to be evaluated independently, marking them as either inside or outside the triangle, finding all pixels the triangle overlaps [22]. The technique for checking if a point is inside a triangle is detailed in Section 5.4.

5.7.2 Fill Rules

For a correct result, it is not possible to render each triangle in its entirety independently. As two triangles share an edge, all pixels residing on the edge would be rasterized twice. Apart from the potential performance loss, it also means that any semi-transparent pixel will become wrong, creating a visible line. Therefore, it is crucial to have rules for when a pixel is included in the triangle to avoid any overlap.

A common rule for handling this is the top-left rule utilized by Direct3D [24]. It states that a pixel should only be included when the pixel center lies within the triangle or directly on a top/left edge. An edge is defined as a top edge if it is completely horizontal above the two other edges and defined as a left edge if it is sloping upwards when using a clock-wise ordering. It is only possible for a triangle to have one top edge, while it can have two left edges.

Note that the host computer that generates the triangles and pixel data must agree with the accelerator architecture on the rasterization rules. That way, both agree on when a pixel should be included or not. If either of the two includes or omits a pixel which the other does not, the two will not line up, and the object's edges will not be correctly represented.

5.7.3 Minimizing work

As the rasterization evaluates pixels independently for inclusion in the triangle, there must be a decision of which pixels to evaluate. The trivial approach is to check every pixel on-screen for inclusion in every triangle. However, as each triangle generally only covers a small section of the screen, doing so results in a vast amount of tests that do not contribute to the final image. The slightly better alternative is to evaluate all pixels within the triangle bounding box, which is easy to find by taking the minimum and the maximum of all three vertices.

Another common technique is tiling, where the screen is divided into small sections, evaluating each section independently [25]. This approach allows for quickly accepting and rejecting entire blocks of pixels when processing instead of doing it on a pixel level. That way, the amount of per-pixel tests can be minimized to the tiles the

triangle intersects. Therefore, the use of tiles has been adopted for the acceleration system.

5.7.3.1 Tile Rejection

If a triangle is entirely outside of a tile, it can be rejected directly. This evaluation is done by checking each edge of the triangle against one corner of the tile. The algorithm for checking a point against a line is described in Section 5.4.

The corner to check against is selected based on the slope of the edge. As the vertex order is known, the slope reveals which side of the triangle the edge resides, allowing the outmost tile corner to be selected. An illustration of this is shown in Figure 5.3, where the orange corner being outside of the orange line guarantees that the triangle can not intersect the tile. It is enough for either of the three checked edges to evaluate the selected corner as not inside for the tile to be rejected.

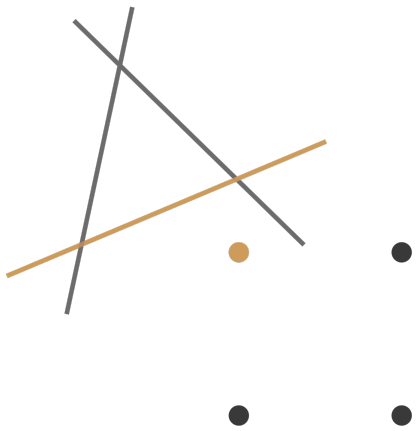

Figure 5.3: An example of a successful tile rejection, where the closest tile corner is outside the closest edge.

5.7.3.2 Tile Acceptance

An entire tile can also be accepted if all corners of the tile lie within the triangle, reducing the number of per-pixel tests required. However, this is currently not utilized in the acceleration system as all pixels pass one by one through the intersection testing. In its current design, marking an entire tile as accepted does not improve the speed as one clock cycle is spent per pixel regardless.

5.7.3.3 Binning

Traditionally, tile-based renderers use binning before rasterization. It is a technique that groups triangles per tile. That way, not all triangles have to be read from memory for every tile, throwing away most of them as the currently rendering area is typically small.

As the acceleration system is optimized for few large triangles, this stage is omitted. Dispatching and displacing all triangles in each section of the frame is a relatively

small overhead since all triangles fit in on-chip memory and can be processed well within the available time. This approach avoids the step of grouping triangles, writing them to memory, and then retrieving them again. Instead, the focus is on making each framebuffer section complete as quickly as possible through a simple and predictable pipeline for minimum latency.

5.7.4 Shading

On a traditional GPU, the pixel color is decided through fully programmable shaders, calculating the final pixel value based on mathematical expressions and potentially fetched data, such as textures. In this system, all pixel colors are instead pre-rendered on the host and delivered as finished textures. The shading of a pixel is then essentially reduced to a texture lookup, fetching a pre-calculated texel value for display. Utilizing finished textures is similar to baking, where expensive computations are done once and stored in a texture for reuse. By letting the host generate all texel values, the acceleration system does not have to care for anti-aliasing or filtering since it is already included in the textures.

5.7.4.1 Texturing

Textures must be stored in memory for easy access. Although it could be possible to store all the texels sequentially in memory, with no wasted space, calculating the correct memory address for the current pixel is made difficult. Instead, for fast and efficient texel lookups, the textures are stored in rectangles, where each row is stored sequentially in memory. This layout allows for moving between rows by jumping a constant distance in memory.

Storing a square texture per triangle means half of the texture is wasted. Therefore, there is a need to share textures between triangles, allowing the texture to be potentially fully used. The sharing is realized by introducing texture IDs, allowing different triangles to be connected to a specific texture.

All textures provided should be of equal resolution as their size on the screen, matching texel to pixel 1:1 for a pixel-perfect result. This constant size also simplifies the UV coordinates as only the position of a single vertex within the texture must be stored. The first vertex in the triangle was arbitrarily chosen. The final UV coordinate for each texel can then be found by displacing the base UV coordinate with the difference in position between the rendered pixel and the first vertex:

$$U_{pixel} = U_{first\_vertex} + X_{pixel} - X_{first\_vertex}$$
$$V_{pixel} = V_{first\_vertex} + V_{pixel} - Y_{first\_vertex}$$

Note that as all triangles are displaced by a number of full pixels, the texture and the pixels on the screen will always line up perfectly, with no need for taking multiple texture samples with texture filtering.

5.7.4.2 Color Tinting

The color value of the fetched texel makes up the base color of the pixel. After, there is only one more step before the final color is calculated. It needs to be tinted, as required by the visual feedback described in Section 4.2.1.

To perform the tinting, the texel color and the tinting color are added for each channel individually. The tinting color is increased proportionally by the maximum distance in which the object has been limited against either of the four dragging limits, as described in Section 4.3.1.3. Note that the limiting amount is not added to the alpha channel not to affect what is being visible. A $scale\_factor$ determines how long distance in pixels is required for reaching a 200% tinting.

$$\begin{aligned} limited\_distance = max(0, X_{dragdistance} - X_{limit\_max}, -X_{dragdistance} - X_{limit\_min}, \\ Y_{dragdistance} - Y_{limit\_max}, -Y_{dragdistance} - Y_{limit\_min}) \end{aligned}$$

$$\begin{aligned} R_{new} &= R_{base} + R_{tint} * (1 + limited\_distance/scale\_factor) \\ G_{new} &= G_{base} + G_{tint} * (1 + limited\_distance/scale\_factor) \\ B_{new} &= B_{base} + B_{tint} * (1 + limited\_distance/scale\_factor) \\ A_{new} &= A_{base} + A_{tint} \end{aligned}$$

Note that the sum of each channel is limited to stay within the minimum and maximum values, preventing overflows leading to incorrect colors.

5.7.4.3 Color Mixing

When the color is completed, it is added to the framebuffer. The merging of the current framebuffer pixel and the new pixel is done using the Porter and Duff Over-operator [26]. How much the new color contributes to the final framebuffer pixel depends on two values. As the alpha of the color is increased, the contribution to the final output increases proportionally. Adjusting the pixel coverage in the framebuffer, denoted C, has the opposite effect. If the coverage is 100%, the pixel is already completed and should not be altered. If it is 0%, it should be replaced entirely. Similarly, adding a 100% opaque color should bring the coverage to one, completing the pixel.

These are the resulting formulas:

$$\begin{aligned} R_{final} &= R_{old} + R_{new} * A_{new} * (1 - C_{old}) \\ G_{final} &= G_{old} + G_{new} * A_{new} * (1 - C_{old}) \\ B_{final} &= B_{old} + B_{new} * A_{new} * (1 - C_{old}) \\ C_{final} &= C_{old} + A_{new} * (1 - C_{old} \end{aligned}$$

5.8 Data Sizes

This section details the number of bits required for all the different data types to meet an appropriate functionality. This includes both the sizes as received from the host and the sizes used internally for processing.

5.8.1 Positions

With 4K resolutions becoming the norm in many devices, this is the assumed largest supported screen resolution for this design. As various aspect ratios are used, screen sizes up to 4096 pixels in both axes are supported, requiring a 12-bit value per axis.

All objects must have the ability to both leave and enter the display area as they are dragged. Therefore, supporting placement only within the viewing area is not enough. In the worst case, a single object must be able to be dragged from the leftmost edge to the opposite side. At the same time, it must be ready to be dragged from the rightmost edge in the opposite direction. This requirement implies that the object must have one screen width available to both the left and right of the display. The argument for the height of the object is the same. Therefore, the worst-case object to be received from the host is three times the screen width, visualized in figure 5.4, requiring two extra bits for a total of 14 bits precision per axis.

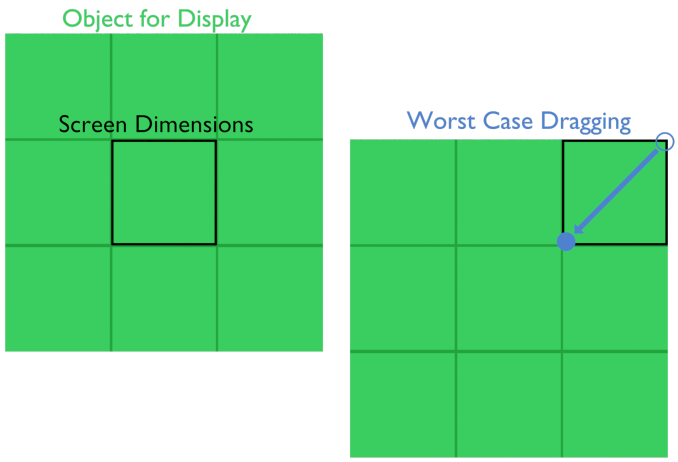

Figure 5.4: An illustration of the required available area of an object for supporting free translation.

A higher bit count is needed when calculating the placement of objects based on interactions internally. As can be seen in Figure 5.4, the three screens wide object can be moved a screen width in both left/right and up/down. This interaction indicates a need to make vertices placeable within a five screens wide area, resulting in three extra bits compared to the screen resolution, for a total of 15 bits precision per axis.

5.8.2 Limits

Objects need to have the ability to be limited to any possible position, even outside the screen. Therefore, this thesis suggests a limit resolution identical to the placement of vertices at 15 bits.

5.8.3 Color

To match most systems today, the color precision used is eight bits per channel for all stored textures and the framebuffers. The color tinting has a smaller precision of five bits per channel, with one bit reserved to denote the sign, allowing it to both add and subtract color. With a limited color range, the host application will have to pick the closest color among those represented. However, this should not pose any problems due to the generally static nature of the tinting color, without the need for full precision to replicate smooth gradients and similar effects.

5.8.4 Object Count

The number of individually interactable items on display that should be supported is hard to estimate. The simple calendar application included with Windows 10 has around 150 touch areas, as can be seen in Figure 5.5, with specific functionality on each as they light up on touch. With multiple applications and larger zoom-outs displaying more information at once, this number would increase. The maximum number of objects for this system was set to 256, which allows for reasonably complex UI:s to be touch accelerated, requiring eight additional bits per triangle to store which object it belongs to. Additionally, 24 bits are needed to store its three parents.

If more objects can be dragged on screen than supported, only some can be accelerated for direct visual feedback. In that case, the host application must prioritize, accelerating the items most likely to be picked. If the objects are only used for tapping, they could be grouped into larger touch areas, providing less fine-grained visual feedback, still informing the user of a registered touch.

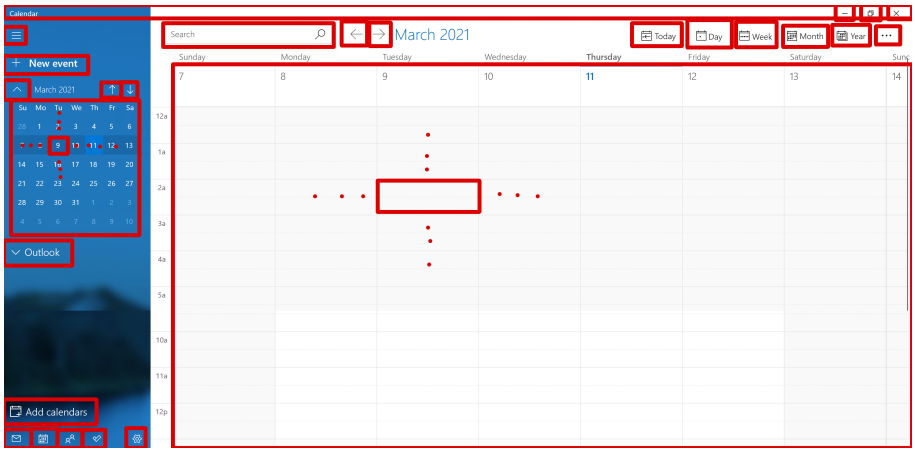

Figure 5.5: A screenshot (reduced in height) of the built-in calendar in Windows 10, visualizing the large amounts of independent touch areas ready for interaction.

5.9 Summary

In short, the graphics system suggested consists of a simple tile-based renderer. No vertex shaders are supported, with the vertex displacement being hardcoded to only support dragging interactions. Triangles are culled if outside the currently rendering area. Binning is omitted due to low triangle counts. Fragment shaders are not supported, with the shading instead consisting of a single hardcoded texture-lookup with optional tinting.

6

Hardware Architecture

This chapter describes the different stages and hardware units involved in the resulting hardware architecture. The hardware implements the operations detailed in Chapter 5 through multiple interconnected hardware units. A complete high-level overview of the architecture can be seen in figure 6.1. This figure is good to refer back to throughout this chapter to track how the current subject relates to the proposed architecture.

In the first subsection, different design considerations are described. In the following subsection, the architecture is detailed, including a description of each hardware unit.

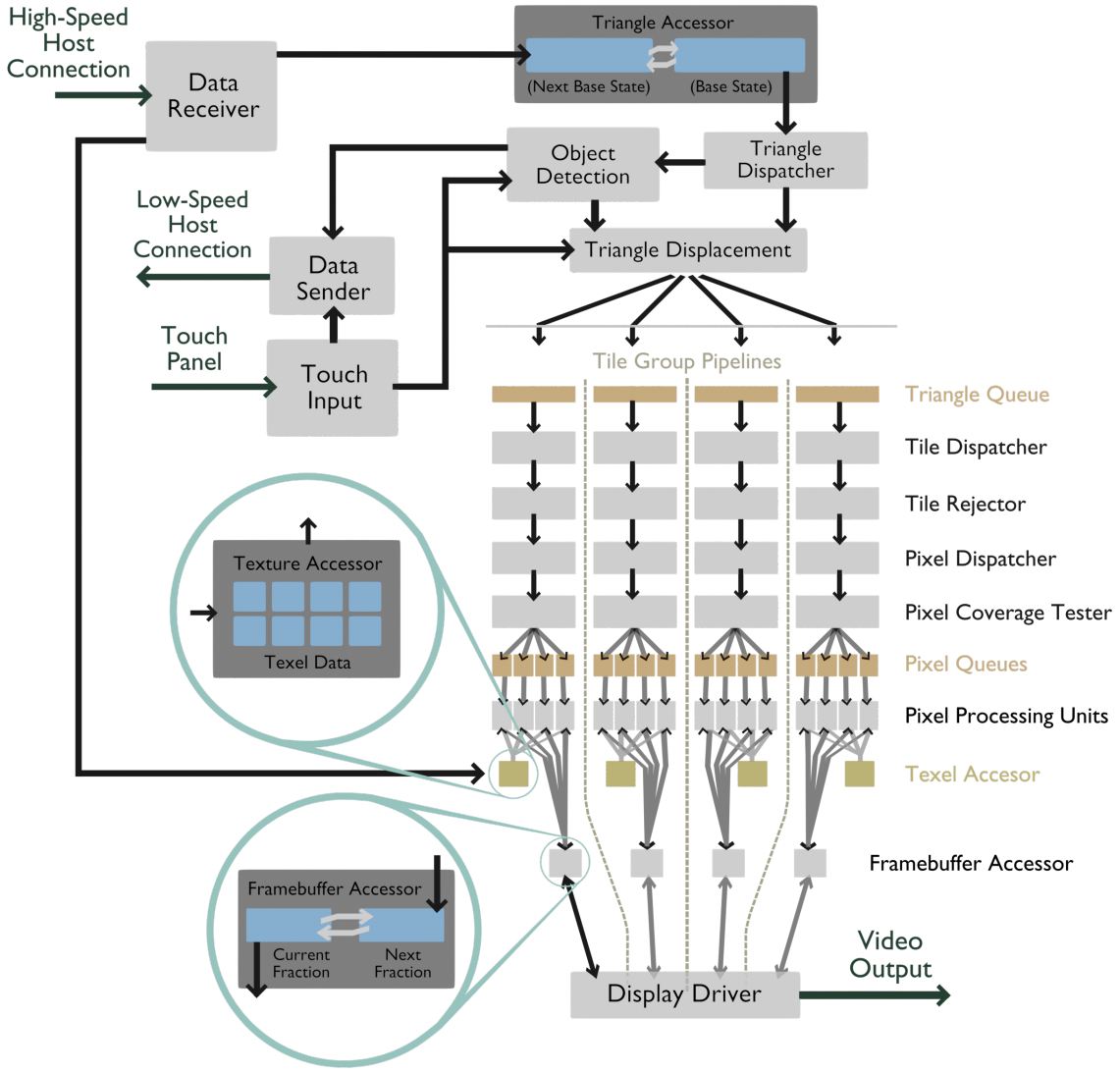

Figure 6.1: An overview of the different units in the resulting architecture and their interconnections.

6.1 Considerations

This section discusses several special considerations that were taken into account when designing the hardware architecture.

6.1.1 Double buffering

Two separate parts of the system need to utilize double-buffering, keeping two copies of the same type of data in memory at once. First is the framebuffer for display. In order to generate the next frame, while the current one is being displayed, two frame buffers must be kept in memory. That way, new color data can be written to the framebuffer until it is complete, while the previous frame is currently being displayed. After a frame has been completed, the next framebuffer is swapped in, allowing the old framebuffer to be rewritten with new data.

Similarly, there is a need for double copies of the triangle data received from the host. It must be possible to receive the triangle data for the next frame while the previous frame data is utilized for display, allowing the transmission of frame data to happen asynchronously over as long a time as required. When the transfer of the new frame has been completed, it is swapped in for use, while the old memory is rewritten during the next transfer.

As the double buffering stores data in two separate parts of the memory, switching between them is highly efficient. All needed is a change in which address is being read or written, making it instantaneous, with no data being transferred.

The texture data received from the host is handled slightly differently. Like the framebuffer and triangle data, the texture must not be modified while being drawn to the display. However, this is not achieved by keeping two copies of all texture data. Instead, it is up to the host not to modify a texture currently in use. Instead, whenever a texture needs to be updated, the host sends the texture under a new ID. That way, the old texture can be used for continued drawing until the transfer of the next frame, including texture, has been completed. The new ID is then used in subsequent frames, making the new version of the texture visible on the screen.

The ability for textures to live over multiple frames allows for a significantly reduced data transfer, as the textures only need to be updated by the host when a change has happened. When a frame transfer has been completed, it is safe for the host to overwrite any textures that were not part of the frame sent, as they are guaranteed not to be visible anymore.

6.1.2 Memory Allocation

When designing a system, there is an obvious limitation in the amount of data available on fast on-chip memory. This limitation is true on FPGAs with fast on-chip Block RAM and when designing an ASIC. Therefore, a decision has to be made what type of data should be kept on-chip for fast access and what data should be streamed in on request.

When the framebuffer is generated, each pixel will be updated multiple times before a final value is reached. This continuous updating is due to the layered approach needed for proper transparency, where colors are blended in one by one. As the framebuffer needs continuous access during processing, it must reside on-chip. It is not feasible to bring it in and out of memory each time the pixel value is updated.

The largest type of data received from the host is the textures, potentially covering the entire display multiple times. Therefore, they must be kept on off-chip RAM since there not enough room. In order to receive the texture data on time, prefetching is supported. The triangle data could be off-chip as well, as they are streamed in for usage in a predictable sequence. However, as the system is designed to be optimized for few triangles, resulting in a small amount of data, it should be considered to place them on-chip for fast and energy-efficient direct access, as well

as simplifying the design.

6.1.3 Test Architecture Simplification

To reduce the implementation complexity for the test architecture, all texture data is stored on-chip for direct access. Instead, to verify that the system would work if the texture data resided on a high-latency off-chip memory, the architecture is set up so the texture data is not used until after a set number of cycles from when it could first be requested, essentially supporting prefetching. This is done through extended Pixel Queues, delaying the time between the Pixel Coverage Tester, where the prefetch would be initialized, and the Pixel Processing Unit, where the texel value is used. The supported delay was arbitrarily chosen as 50 cycles.

Due to insufficient space for storing full-resolution textures, the test architecture utilizes simplified textures of 1x1 texels in size compared to the suggested 4096x4096 textures that should be supported in a real acceleration system. In order to cover large on-screen objects while still mapping texels to pixels 1:1, texture repetition is utilized, essentially creating single-colored objects.

The effects the texture memory has on the resulting system, including both latency and bandwidth, are discussed further in Section 8.1.5.

6.1.4 Framebuffer Storage

Due to the limited size of the Block RAM, the entire framebuffer can not be kept on-chip at once. Instead, it contains a small section at a time. Thanks to tiled rendering, this works well, as only the section of the framebuffer currently rendering needs to be on-chip. After one section of the framebuffer has been completed, it continues with processing the next one.

Working on a small section at a time also reduces the time it takes to complete the pixels, improving the per-pixel processing latency, which is of great interest in this work. Therefore, the framebuffer section is kept smaller than required by the memory available, giving ample room for keeping both copies required by the double buffering on-chip. This placement allows all pixels to be read from the Block RAM directly to the screen without involving any off-chip memory.

6.1.5 Parallelism

The key to achieving good performance in the hardware architecture is to work on multiple data in parallel. Similar to how a regular GPU works, the architecture will evaluate multiple pixels in parallel to generate finished images in a short amount of time. This section discusses how rendering a single frame is divided over multiple hardware units to achieve fast rendering.

Table 6.1: The number of tiles needed to create common display resolutions

Frame Width	**640**	**1024**	**1280**	**1920**	**2560**	**3840**		Tile Group Width
	5	8	10	15	20	30		**128**
Frame Height	**480**	**768**	**720**	**1080**	**1440**	**2160**		Tile Group Height
	60	96	90	135	180	270		**8**
Total Tile Groups/frame	300	768	900	2025	3600	8100		

6.1.5.1 Work Division

Since this architecture does not support any filters or other advanced effects, each pixel in the finished image can be rendered in isolation, with no influence on neighboring pixels. However, rendering every pixel in parallel is not feasible as it requires too many processing units. Instead, multiple pixels must be assigned to the same processing unit, sharing resources.

In grouping pixels together for work, the memory layout of the FPGA card used for implementing the architecture was considered. The framebuffer is stored on on-chip Block RAM, consisting of several separate memory modules. As they are separate memories, they can be read and written independently of each other. Therefore, all pixels on one memory module are seen as a separate group, which can be accessed and handled independently.

Due to the usage of a tiled renderer, one logical grouping of pixels already exists in the form of tiles. As some rendering optimizations will be performed on the tile as a unit, the pixels within a tile should not be split up. Consequently, assigning pixels to a Block RAM means that entire tiles should be assigned to it. This assignment leads to each memory module hold a set of tiles, referred to as a tile group.

A Block RAM module is 36Kbits in size and can be configured in different widths; 18, 36, and 72. Each pixel consists of 24-bit color (RGB) and 8-bit coverage for computing accurate blending, for a total of 32 bits. The memory width of 36 is deemed the best as it allows one complete pixel to be stored per address with little waste, for a total of 1024 pixels per Block RAM.

A tile group size of 128x8 pixels is suggested for three reasons. First, it maximizes the single BRAM module, equating to the 1024 pixels available. Secondly, these dimensions are evenly divisible with most commonly used resolutions, as seen in Table 6.1. Lastly, as displays are typically driven line by line, complete framebuffer lines must be present in memory. A small height allows large resolutions to be supported on a small memory as it allows the full width of the display to fit the memory. A resolution of 3840x2160 can be supported through a single row of 30 groups (requiring the frame to be rendered in 2160/8=270 sections).

Internally, the tile group consists of tiles of size 4x8. This tile size was selected for two reasons. First, it simplifies the design as the tile group consists of a single row of tiles. Secondly, it gives an equal amount of tiles per group as there are pixels per tile, 32, which has an advantageous property described in Section 7.1.2.

6.1.5.2 Per-tile group Parallelism

The architecture consists of multiple Tile Group Pipelines, each handling the rendering of one tile group. Together, they render the entire current section of the framebuffer. As the BRAM is limited to one write per cycle, one new pixel must be ready to be written every cycle for maximum performance. Therefore, the pipeline should be designed to generate one completed framebuffer pixel per cycle.

The act of generating a framebuffer pixel takes several cycles. First, it must fetch the current pixel value from the framebuffer. If the pixel has already been completed, the new pixel can be dropped immediately without further work since it will not be visible in the finished image. If the framebuffer pixel is not complete yet, the unit fetches the correct texel value from memory, blends it with the old pixel value, and writes it back to the framebuffer.

As the Block RAM should be provided with a new pixel write every cycle for maximum usage, it is clear that more than a single pixel operation must be in progress at the same time. The architecture utilizes multiple pixel processing units working independently, each processing a single pixel, allowing them to wait for data without affecting each other. By having multiple units active at once, the likelihood of at least one unit having a complete pixel value is high, allowing the BRAM to perform a meaningful write most cycles.

There are two different issues as multiple units are writing to the same Block RAM. The first is that the writing order must be upheld. For correct depth ordering of all triangles, a pixel can not be merged to the framebuffer before those that came earlier, essentially skipping in line. All pixels belonging to the same framebuffer pixel must be written in order for correct alpha blending.

The second issue is that of mutual exclusion. As a unit reads the value of a framebuffer pixel, modifies it, and writes back, another unit could write to the frame buffer pixel first, whose result then gets overwritten. When a unit has started processing a framebuffer pixel, no other unit may use that value for further processing as the value will be replaced soon.

The proposed solution to both of these issues is to let each pixel in the framebuffer belong to a unique pixel processing unit. That way, only one pixel processing unit is modifying the pixel, satisfying the mutual exclusion. It is also guaranteed that a single unit processes the pixels in the order they arrive, satisfying the ordering.

Having a unique pixel processing unit for every framebuffer pixel is not feasible. Therefore, multiple framebuffer pixels must be assigned to the same unit. This assignment has the potential for load balancing issues. If all pixels arriving for processing are assigned to the same unit, only one unit will be used while the others sit idle. To reduce the risk of this imbalance, neighboring pixels are assigned to different units, as can be seen in figure 6.2. Any triangle bigger than a pixel is guaranteed to hit more than one pixel processing unit with this assignment, allowing

for parallelism. The units are laid out with maximum variation along the x-axis as the pixels are dispatched in row order, increasing the likelihood that each unit will receive a pixel.

The worst case for this approach is a narrow triangle spanning over a single pixel column, activating one-quarter of the pixel processing units. However, as the tiles are only eight pixels high, it would quickly proceed to the next tile to the right, which hopefully has more variation in the pixel processing units it hits. Note that pixel queues are added before the pixel processing units to mitigate irregularities in the activated units, allowing more pixels to be dispatched at once, increasing the likelihood of each unit receiving a pixel.

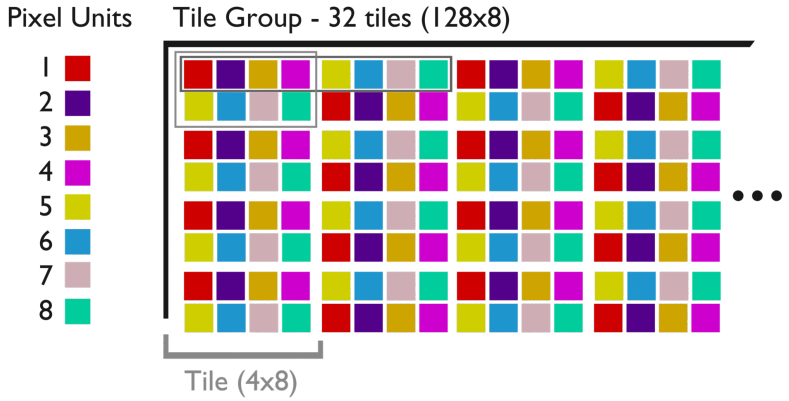

Figure 6.2: The assignment of pixels within a tile to different pixel processing units.

6.1.5.3 Tile Positioning

As tile groups and tiles are used for rendering, they should be placed within the screen area. As each pixel only needs rendering once, no tiles should overlap. Therefore, the screen area is divided into a grid based on the size of the tile groups, 128x8 pixels. This division gives a total of 512 positions along Y and 32 positions along X when supporting up to a 4096x4096 display. Therefore, locating a tile group on screen requires 9 and 5 bits, respectively. The tiles within the tile group are located using no extra bits along Y since it is only one tile high and 5 bits along X. This organization makes the tile group and tile position directly readable from a pixel address by looking at the correct subset of bits, as demonstrated in Figure 6.3.

6.1.6 Pipeline Control

As different stages take a different amount of time to complete depending on the input, hardware units must have a way to notify each other of their state. This communication allows the entire pipeline to halt if a unit can not accept any more data, making sure nothing is lost.

Each unit in the pipeline has four control signals, two inputs, and two outputs, illustrated in Figure 6.4. The `output_available`-signal notifies the next unit in

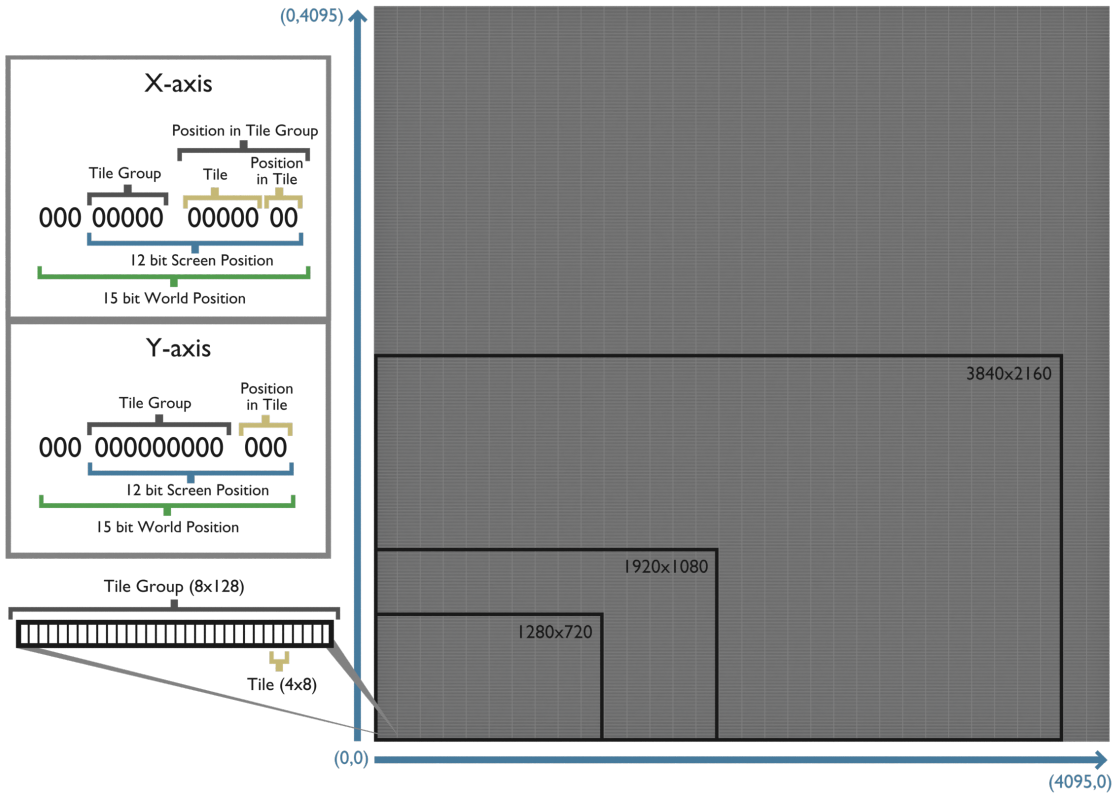

Figure 6.3: The screen and world space relation along with the possible placement of tile groups over the viewable screen area. Note that three leading zeroes in the pixel position denotes a pixel being visible within the display area.

line if the provided output is valid. A unit reads the `input_available`-signal to see if the currently provided input is valid. These two signals are therefore connected, providing a signal between two units on data validity.

The `ready_for_input`-signal declares if a unit is open to accepting new data. The `output_read`-signal indicates that the output has been read, allowing the sending unit to output the next unit of data. Therefore, the `ready_for_input`-signal is connected to `output_read`, making the sending unit not switch the output data unless the signal is set. If both `ready_for_input` and `input_available` are set, data is successfully transferred.

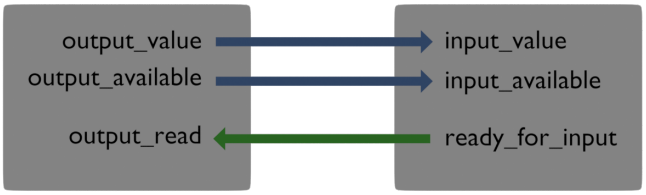

Figure 6.4: An illustration how two units in the pipeline communicate during the passing of data.

It is often beneficial to know when a pipeline is empty in hardware architecture,

knowing that all work has been finished. However, any such functionality is omitted from this architecture. No benefit is provided from knowing the current state of the processing units within the pipeline. Framebuffer sections are rendered at set intervals, and any application running should be designed such that the work finishes within the available time frame. If the pipelines are not empty on time, the host behaves incorrectly, as detailed in Section 7.1.2. Therefore, it is a behavior that the acceleration system should not handle.

6.2 Hardware Description

This section details all the different hardware units in the complete architecture and their relationships, as can be seen in Figure 6.1.

6.2.1 Master Controller

The master controller is a finite state machine residing at the top level of the architecture, orchestrating all units. Its primary function is to trigger different work sequences and handle the double-buffered memories for correct data access.

When a touch is first detected on the screen, the Touch Input-unit signals the master controller, which initiates the object detection sequence. The purpose of the sequence is to find which object the user selected, which is done by dispatching all triangles to the Object Detection unit. When a hit is registered in one of the triangles, the Object Detection unit signals the master controller, which stops the triangle dispatching. This sequence is performed once on the first touch and not continuously while dragging.

Once every frame section, after any potential object detection, the master controller initiates the rendering. During this sequence, the master controller dispatches all triangles directed to the rendering pipelines. Note that the triangles are dispatched multiple times per frame as only a small section of the display is rendered at once. The repeated dispatching simplifies the design as no geometry has to wait in the pipeline to become relevant later; if it is not relevant at this point in time, it is discarded.

It is also up to the master controller to assign which pixels on the screen are being processed by which Tile Group Pipeline. This assignment is performed by providing each Tile Group Pipeline with a tile group x and y coordinate, specifying its location on the screen. When the current section has finished displaying, the framebuffer memories are swapped, starting the display of a new section while rendering is started on the next one.

6.2.2 Data Receiver

The Data Receiver is responsible for receiving and processing data sent from the host. Each message received has a size of 16 bits. The types of messages and

Table 6.2: The format of the data received from host.

Command \ Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Start Frame	0	0	Clear*													
End Frame	0	1														

*Current Transformation

Triangle Start	1	0	WindowParent	12-bit Texture X-coordinate												
	8-bit Texture ID			12-bit Texture Y-coordinate												
		15-bit Triangle Coordinates - Vertex 1, X														
		15-bit Triangle Coordinates - Vertex 1, Y														
		15-bit Triangle Coordinates - Vertex 2, X														
		15-bit Triangle Coordinates - Vertex 2, Y														
	Visible	15-bit Triangle Coordinates - Vertex 3, X														
	Touch	15-bit Triangle Coordinates - Vertex 3, Y														
	8-bit Object ID								8-bit Parent ID 1							
	8-bit Parent ID 2								8-bit Parent ID 3							
	5-bit Tint Color - Red				5-bit Tint Color - Green				5-bit Tint Color - Blue				5-bit Tint Color - Alpha			
		15-bit Window Limits - Right														
		15-bit Window Limits - Left														
		15-bit Window Limits - Top														
		15-bit Window Limits - Bottom														
	12-bit Drag Limits - Right												12-bit Drag Limits - Left			
									12-bit Drag Limits - Top A							
					12-bit Drag Limits - Bottom											

Texture Start	1	1	6-bit Texture ID										4-bit Texture Width			
	8-bit Texel Red								8-bit Texel Green							
	8-bit Texel Blue								8-bit Texel Alpha							

embedded data can be seen in Table 6.2. When a completed frame has arrived, the unit signals the master controller. When the currently displayed image has finished rendering, the double-buffered memories are switched to utilize the newly received data for the next displayed image.

6.2.3 Touch Input

The Touch Input-unit drives and reads the touch panel to generate touch input information to be utilized by the rest of the system. As generating the touch input is not of focus in the proposed architecture, this unit can be treated as a black box. The provided data consists of a signal stating if a touch is present, the touch down coordinate, and the distance the finger has been dragged from the touchdown point.

6.2.4 Data Sender

As the acceleration system acts as the touch controller, touch data must be passed along to the host system for use in program logic. For simplicity, the touch data is sent continuously, with each message consisting of ten bits. The start of a new touch sample is denoted by sending nine zeroes in a row, which can only occur at the start of a touch sample as each following byte of data sent is divided by two ones. This allows the host to start listening at any point in time, waiting for the sequence of nine zeroes. The format of the sent data can be seen in Table 6.3. Note that a Touched Object ID of zero denotes no object currently being touched.

Table 6.3: The format of the data sent to host.

Command \ Bit	9	8	7	6	5	4	3	2	1	0
Transfer Start	1	0	0	0	0	0	0	0	0	0
	1	1	8-bit Touched Object ID							
	1	1	12-bit Touch Down X							
	1	1					12-bit Touch Down Y			
	1	1								
	1	1	12-bit Drag Distance X							
	1	1					12-bit Drag Distance Y			
	1	1								

6.2.5 Triangle Dispatcher

The Triangle Dispatcher is a simple unit that reads the triangle data from memory sequentially and dispatches them one by one. Each dispatched triangle contains all relevant data, such as the Object ID and the tinting color. As detailed in Section 6.2.1, the dispatched triangles are routed for use in both object detection and rendering by the master controller. It is controlled through a single start signal, which initiates the dispatching when it goes high. Setting it to high while the dispatching is already in progress restarts the sequence allowing for rendering to commence directly after the touched object has been found.

6.2.6 Object Detection

The Object Detection unit receives a sequence of triangles along with a touch-down coordinate. The coordinate is tested against the triangles, as detailed in Section 5.4. The work is divided into multiple pipeline stages, testing one triangle per cycle once the pipeline has been filled. The first stage calculates all subtractions. It also checks so the touch point is within the visibility window, as an area that can not be seen should not be selected. The second stage performs all multiplications. The last stage checks so the differences between the multiplications are all positive, at which point there is a valid intersection.

The master controller is signaled when an intersecting triangle has been found. The unit provides the ID of the intersected object along with the dragging limits of the intersected triangle, as the dragging limits of the touched object should be used for all interactions.

6.2.7 Triangle Displacement

The Triangle Displacement unit receives dispatched triangles and adjusts them based on the action being performed by the user. These adjustments require information on the touched object, the dragged distance, and the dragging limits, all of which it receives from the Touch Input and Object Detection unit.

The triangle data is adjusted in three ways. First, it displaces the vertex positions by the dragged distance if the triangle is part of the interacted object through parenting or directly. Secondly, it moves the visibility window if it has been connected to the

object currently being dragged.

Lastly, it is responsible for adjusting the tinting color. If the object is not touched, the tinting color is replaced with all zeroes, making sure the color is not affected later during rendering. If the object is touched, the tinting color is kept as it. If the object is being dragged and limited, the tint is proportionally increased by the amount the object is being limited, as detailed in Section 4.3.1.3.

During rendering, each displaced triangle is sent to all Tile Group Pipelines in the system. The triangle is considered read if all pipelines are free to accept it, in which the unit moves on to processing the next triangle.

6.2.8 Tile Group Pipeline

The Tile Group Pipeline is the main rendering pipeline of the system, responsible for rendering one tile group. The master controller assigns the specific tile group each pipeline handles. The pipeline accepts triangles as input and writes finished pixels to its connected framebuffer memory.

6.2.8.1 Triangle Queue

The first step of the Tile Group Pipeline is the triangle queue. It serves two purposes. First, it allows the pipeline to accept a set of triangles from the Tile Displacement unit without halting. A queue size of six is used in this implementation. Secondly, it discards all triangles that are guaranteed to not be within the area of the tile group that is being rendered, as detailed in Section 5.6.3.

The queue is implemented in a simple pipeline fashion, with a chain of stages, each holding a triangle. When the stage in front is ready to receive, it passes on the triangle. The passing of triangles means that the selected queue size adds six cycles to the total latency, regardless of whether the queue is full or not, as the triangles have to pass through each stage.

Note that all triangles are inserted into the queue regardless if they will be visible on-screen or not. Instead, triangles are dropped at stage two and three if they are outside the tile group. Stage two of the pipeline checks if all vertices are above or below the tile group, and stage three checks if all vertices are to the left or right of the tile group. This checking allows multiple triangles to enter the queue and be discarded, regardless of whether the pipeline is halted up ahead.

6.2.8.2 Tile Dispatcher

The Tile Dispatcher accepts triangles from the Triangle Queue and dispatches tiles for processing sequentially one by one. As each tile group consists of 32 tiles, it takes 32 cycles to handle one triangle, assuming the pipe ahead is free. Note that the triangle data is passed along with the tile as it is needed for processing in later stages. When all tiles in the tile group have been dispatched, the process is repeated for the next triangle.

6.2.8.3 Tile Rejector

The Tile Rejector receives triangle data along with a tile and drops the data if the triangle is not overlapping the tile. This rejection is done through several pipeline stages, checking one tile per cycle once the pipeline has been filled. The algorithm used is detailed in Section 5.7.3.1, split into separate stages. The first pipeline stage looks at the edge slopes to determine which vertices to check against the tile corners. Stage two performs all subtractions, while stage three multiplies these differences. The fourth stage checks if either of the differences between the multiplications is negative. If so, the data is discarded in stage five.

6.2.8.4 Pixel Dispatcher

The Pixel Dispatcher receives all tiles which have not been discarded. It dispatches all pixels within the tile one by one, going in row order. As a tile consists of 32 pixels, it takes 32 cycles to complete a tile. This unit is also responsible for making sure content is not drawn outside the visibility window. This visibility limitation is done by dropping all pixels which lie outside the limits.

Lastly, the unit is also responsible for generating the final UV coordinate for the texture lookup. Previously, the UV coordinate referred to the position of the first vertex in the triangle. After this stage, it refers to the position of the specific pixel for direct use.

6.2.8.5 Pixel Coverage Tester

The Pixel Coverage Tester receives a pixel and a triangle. The pixel is tested against the triangle, and if it lies within, the pixel is passed on. For accurate coverage testing, the unit implements the fill rules detailed in Section 5.7.2. As the calculations performed mimic the intersection testing done by the Object Selection unit, the work is divided into similar pipeline stages.

This unit is also responsible for initiating any prefetching of texel data by relaying the UV coordinate and texture ID to the Texture Accessor.

6.2.8.6 Pixel Queue

The verified pixels are sent into different pixel queues depending on their on-screen position. This division allows each Pixel Processing Unit to fetch new pixels for processing from its own designated queue. The queue is implemented similarly to the Triangle Queue, consisting of multiple pipeline stages, passing data to the next stage each cycle.

The longer the queue, the longer time the pixel dispatching can continue without halting, increasing the chance of each Pixel Processing Unit having something in their queue, preventing them from being idle. A queue depth of two was selected for this purpose, only halting once a fourth pixel is designated to the same Pixel

Processing Unit (where one pixel is currently being processed, two are in the queue, and one is waiting to enter the queue).

The queue depth is also expanded to match the 50 cycle texture memory latency the test architecture is designed for, as described in Section 6.1.3, which in this case requires a depth of eight. Once eight spots are taken, any new value entering is guaranteed not to reach the Pixel Processing Unit for 8*6=48 cycles, as six cycles are the optimal processing time of one pixel. This delay, plus two cycles needed within the Pixel Processing Unit, guarantee that the requested texel value is used earliest 50 cycles after it could have been requested from memory, preventing any halting. In total, this gives a resulting queue depth of 2+8 = 10.

6.2.8.7 Framebuffer Accessor

The Framebuffer Accessor is responsible for providing read and write access to the framebuffer section of the Tile Group Pipeline. There are three types of access available. First is reading from the completed framebuffer, utilized by the master controller, which relays the data to the Display Driver. As only one unit reads the data, the master controller is given constant access, providing a pixel address and receiving a pixel color for display every cycle.

When a pixel is read from the completed framebuffer section, the stored pixel value is overwritten with zeroes. This deletion makes sure that the memory is clean when it is swapped back for rendering again, not letting old data affect the resulting image. Discarding the values directly after reading works well since the result of the framebuffer section is designed only to be used once. After the section has been displayed, the framebuffer memory will contain a new section of the frame, having no use of the old data.

The two other types of access are reading from and writing to the in-progress framebuffer, which the Pixel Processing Units utilize. As the BRAM can only read and write once per cycle, read and write access is given to one unit each for every cycle. All Pixel Processing Units request access through two bits, one for reading and one for writing.

For fairness, a round-robin accessing pattern is utilized among those requesting access. The ID of the last unit accessing is stored, selecting the next in line in the next cycle. For example, if units 1,3, and 5 are signaling that they want to access, and the last unit to access was 2, unit 3 will be picked. This order guarantees that each unit will get access every Nth cycle in the worst case, where N is the number of Pixel Processing Units in the Tile Group Pipeline.

6.2.8.8 Texel Accessor

The Texel Accessor provides two types of accesses to the stored textures. First is writing access, which is exclusively used by the Data Receiver, allowing it to have a direct connection. Secondly is reading access, which must be shared between all Pixel

Processing Units. A round-robin scheme is utilized, similarly to the Framebuffer Accessor, giving a worst-case waiting time proportional to the number of Pixel Processing Units.

In the current system, all texture data is replicated once per Tile Group Pipeline for simplicity. In a real system, each Texel Accessor would hold the (potentially prefetched) texel data explicitly requested by the pixels in that pipeline.

6.2.8.9 Pixel Processing Unit

The Pixel Processing unit is the final step of the pipeline, where the final pixel values are generated and written to the framebuffer. The unit is designed as a finite-state machine, processing the pixel through multiple stages until completion, in which it moves on to the next one.

The first stage requests the corresponding pixel value from the framebuffer. When received, the unit enters stage two, where the framebuffer pixel is analyzed. If it is already completed, the pixel is dropped, and the unit moves on to the next one. If it is not completed, the corresponding texel value is requested. When received, it enters stage three, where the final color is calculated by adding the tinting color, as described in Section 5.7.4.2. In stage four, the resulting coverage is calculated, used in stage five to calculate the final framebuffer pixel. In stage six, a write to the framebuffer is requested.

The unit moves on to the next pixel before the write has been granted to reduce latency. This is done by setting the final pixel aside, waiting to be written, independent of the state machine. It is guaranteed that write access is given before the next pixel value is completed, as framebuffer access is given in a round-robin fashion with an equal amount of participants as the optimal processing time. Note that in stage one, the pixel currently waiting to be written must be checked first before any pixel is fetched from the framebuffer not to have a read-write conflict.

6.2.9 Display Driver

The Display Driver drives the connected display with the generated image. As the driving of displays is not part of the proposed architecture, this unit can be considered a black box, requesting a pixel address from the master controller, which returns the pixel color to be displayed. Therefore, it is up to the Master Controller to direct the memory request to the correct Block RAM module based on how they are organized on the screen.

Regardless of the implementation, the critical part is that all pixel data is available in memory for readout by any potential display technology. However, the Display Driver must request the pixels in an agreed-on pattern for the master controller to be able to provide the pixels on time. Therefore, the Display Driver has two additional signals for notifying its future intentions, signaling which fraction will be required next, as well as when the frame has been finished.

7

Results

In order to evaluate the proposed hardware architecture, it was implemented in the form of VHDL code, loaded onto an Arty A7-100T FPGA development board. This allowed for reasoning regarding the achieved processing latency and supported complexity, detailed in Section 7.1.

The FPGA board was then fitted into a larger test system, connecting it to a touch panel and a display. In turn, the FPGA board was driven by a separate microcontroller, acting as a host system running two test applications. This allowed for verifying the correctness of the architecture, as well as evaluating the achieved touch experience, as detailed in Section 7.2.

7.1 Hardware Architecture

This section focuses on the implementation of the proposed hardware architecture and its resulting characteristics.

7.1.1 Processing latency

The achieved processing latency of the designed architecture can be seen through simulation and reasoning. By counting the clock cycles needed from start to finish, the total time spent can be calculated as the clock frequency is known.

In the current configuration of the architecture, eight rows of pixels are generated at once. Once completed, the processing of the next eight rows starts. This dividing of the image gives a hard deadline for the processing time inherent to the architecture, where the worst-case processing latency is the time it takes to display eight rows of pixels.

The current configuration drives a display in a 1024x768 resolution at 120fps. Including synchronization times, pauses between each row and frame, the equivalent of 1344x806 pixels are sent to display. One pixel is output per cycle, giving a clock frequency of 1344x806x120=130MHz. This output frequency means that eight rows of pixels have a time of 8*1344=10752 cycles to complete. At 130MHz, this corresponds to $10752/130^{*}\times10^{9} = 83\mu$s of processing time. Therefore, the system's

processing latency is 83μs, as it is the time between taking the input to completed pixels in memory.

7.1.2 Supported Complexity

With a constant time available for processing, there is a limitation in how complex content can be displayed. When this limit is exceeded, the graphics displayed will no longer appear correct. An example of this is visible in Figure 7.1, where the triangles which do not finish on time incorrectly continue their rendering onto the next framebuffer section. This behavior contrasts with traditional rendering, where the process can take as much time as needed, affecting the achieved frame rate and latency instead of image correctness.

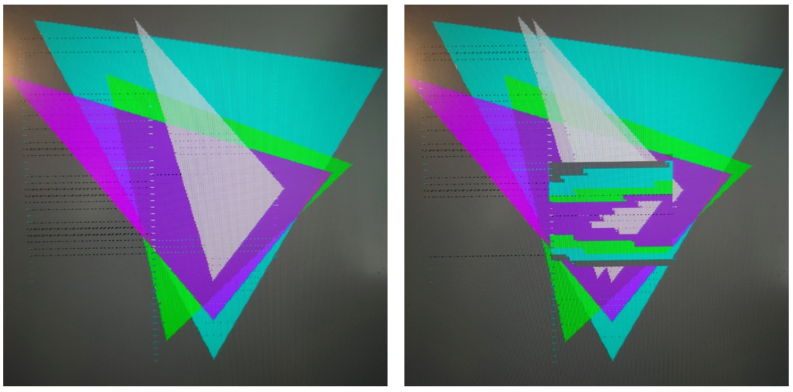

Figure 7.1: An example of exceeding the supported complexity on the acceleration system. The figure to the left renders all requested triangles on time[1]. Adding another triangle makes several framebuffer sections not finish on time, as visible in the image to the right.

Note that complexity in this context is the representation of content in the view of the acceleration system. Multiple triangles and objects increase the rendering complexity while displaying textures with increasingly advanced pre-rendered graphics does not.

The main limitations of the system can be reasoned about as the general process, and total clock cycles available are known. Note that the following calculations are simplifications that should be seen as an indication of the supported complexity. In practice, the limits are lower due to additional costs, such as checking for the selected object, filling the pipelines, and halting costs.

There are two main limitations in play. First, there is a limitation in accessing the framebuffer memory. For each pixel designated for the framebuffer, the current pixel value must be read from the framebuffer first. As only one read can be performed per cycle, and with 10752 cycles available, this limits all triangles within the tile group to cover at most 10752 pixels.

[1]The incorrect pixels in the images are due to unresolved hardware implementation issues.

Second, whenever a triangle intersects a tile, each pixel in the tile must be tested for inclusion in the triangle. As one pixel is tested per cycle, it takes 32 cycles to evaluate the entire tile, allowing for a total of 10752/32=336 tiles to be evaluated per tile group. On average, this gives roughly 336/32=10 triangles per tile.

Note that the extreme case of 336 triangles hitting a single tile is supported. With an equal amount of tiles per tile group as pixels per tile, the tile rejection completes while processing the pixels of a single tile, which is the smallest possible intersection. Writing 336 times to the same framebuffer pixel is not a problem either. Generating each write takes eight cycles, giving a total count of 8*336=2688 cycles, well within the available time. However, what must be considered is the total number of pixels the triangles cover, so it remains below the stated limit.

As each tile group pipeline works as an independent unit, both presented limitations are on a tile group level. However, there is a third limitation affecting the system as a whole. As the triangles are dispatched in order, multiple triangles in a row going to a single tile group pipeline can potentially starve the other pipelines from work due to the full pipeline halting further triangles from being dispatched. Triangle queues are inserted to mitigate this imbalance, but whenever they are filled, the dispatching must stop. Therefore, in the extreme case, the limitations of a single tile group can be seen as a global limitation for all pipelines as there is only one Tile Group Pipeline active simultaneously.

7.2 Test System

This section focuses on the test system and its achieved characteristics, including the usage of different test applications.

7.2.1 Hardware Overview

The Arty A7 FPGA development board is loaded with the designed hardware architecture. It sends resulting images to display by utilizing a Digilent VGA expansion module. The module is connected through a VGA cable to an AOC 24 inch 120Hz monitor. The VGA port is simple to drive from a technical standpoint, as it accepts a stream of data controlled by vsync and hsync signals. This connection allows the test system to ignore the handshaking and complex protocols of modern interfaces.

The touch input is received through a 12.1 inch 8-pin resistive touch panel directly connected to the FPGA board's analog inputs. The pins are shorted in order to utilize the panel as a 4-pin touch panel for simplicity. Resistive touch panels are simple to read, as the touchpoint is found by reading the voltage of two different pins, for X and Y, respectively. Utilizing a raw touch panel without a controller also allows for complete control over when the panel is sampled, giving more predictable latency. The touch panel is mounted on top of the display to allow for simultaneous touch input and viewing of the displayed images, creating a touch screen.

The FPGA is connected to an Arduino Uno microcontroller development board acting as a host. The transfer of data from the host utilizes five GPIO-pins on both the FPGA and the Arduino. One pin is dedicated as a clock to detail when new data is available, and the remaining four pins are used for data transfer of a half byte per tick. As the Arduino outputs 5V while the Arty A7 uses 3.3V, a voltage divider was used. The lower speed data transfer to the host is realized through two pins, one for the clock and one for the data. As the signal goes from a 3.3V to a 5V system, a level shifter was used. The complete system overview can be seen in Figure 7.2.

7.2.2 End-to-end Latency

Measurements of the end-to-end latency in the test system were performed, comparing it to several high-end touch devices. The devices were filmed in 240fps utilizing a OnePlus 7 Pro smartphone, allowing for a temporal resolution of roughly 4 ms. The setup can be seen in Figure 7.3. The highest refresh rate among the tested devices is 120Hz, allowing for two recorded frames per frame displayed. When swiping on screen, it was compared how many frames in the recorded video it took until the object appeared in the correct location. In a zero-latency system, the object would always be in the correct location. In contrast, a higher latency system will have the item appear in the correct location several recorded frames later.

This approach gives a somewhat coarse precision limited to the video's frame rate, in this case roughly +/-4 ms. The precision is also worsened by potential human error in the measuring within the video. Still, it indicates the major differences between the devices. The item tested was the app list on the device, if available, to get as close to the pure OS experience as possible. On the test system, a custom application replicating a similar scroll list was tested instead.

On Windows, an overcompensation seems to be performed when dragging, as the app list moves beyond the finger position and then returns. Although an interesting technique to improve the user experience, it makes it difficult to measure the latency accurately. Instead, a window was dragged on the screen as this interaction does not seem to utilize the overshooting technique. Frames from the captured videos demonstrating the measurements can be seen in Figure 7.4.

As each image is displayed on screen for a specific duration, the latency of each device is divided into two values: when the correct image first appears and when it is replaced. These two indicate a minimum latency, when the image is as new as possible, and a maximum, when it is as old as possible. Note that the difference between the minimum and maximum is strictly a function of the frame rate, as it depends on how long each frame is shown. Therefore, the value to focus on is the minimum latency, which indicates when the image was ready for use.

The full results can be seen in Table 7.1. To summarize, the modern touch devices showed a minimum latency of roughly 50-80ms, while the test system had a minimum latency of roughly 4ms.

Table 7.1: The end-to-end latencies measured while dragging objects on different touch devices.

Device	Year	OS	Display FPS	Frames of Latency in 240fps Video	Minimum Latency (ms) *(+/- 4)*	Maximum Latency (ms) *(+/- 4)*
Microsoft Surface Book 3	2020	Windows	60	20	**83**	100
Sony Xperia 5ii (60fps mode)	2020	Android	60	16	**67**	83
Apple iPhone 12 Pro	2020	iOS	60	14	**58**	75
Sony Xperia 5ii (120fps mode)	2020	Android	120	12	**50**	58
Apple iPad Pro 3rd Generation	2018	iOS	120	12	**50**	58
Test System	-	-	120	1	**4**	12

7.2.3 Test Applications

Two different test applications were created, listed in the following subsections, each utilizing and demonstrating different functionalities supported by the acceleration system. The test applications are written in C++ specifically for Arduino Uno, as the GPIO is utilized directly by reading and writing specific memory addresses.

Each of the test applications has been shown to work. The applications allow for low touch latency while retaining the possibility to remain in control of what is displayed on the screen through program code. In addition, the application being shown is switchable from one frame to the next, demonstrating the system's general flexibility in displayed content.

Note that a plastic piece was placed on the finger to produce a more reliable touch input during the demonstrations of the test applications, shown in Figure 7.5. However, as the work of this thesis is not about creating reliable touch input, this does not affect the functionality demonstrated.

7.2.3.1 Window

The Window application replicates a window as often seen in a multitasking OS, as shown in Figure 7.6. The entire window, including contents, can be moved around on the screen by dragging the top bar. Dragging on the window background does nothing. The left part of the window is a scroll area that can be scrolled up and down, only displaying content within its designated area. The right part of the window contains two objects that can only be dragged within their containing area, demonstrating multilevel parenting. The application utilizes the following functionality:

- Dragging limits
- Visibility Window
- Parenting chains
- Keep object position *(Performed by Host)*

As the content is static, the application sends one frame to the acceleration system at the start, proceeding to only listen to incoming touch samples. When the user has finished an interaction, the host updates the positions of the objects and sends a new frame. A clear transformation flag is included in the new frame to remove the current transformation from the acceleration system as it is now included in the frame data. Note that since the window's content is not changing, textures are only sent once on the first frame. All subsequent frames only contain triangle data.

7.2.3.2 Game

The Game application replicates a classical ping pong game, with a ball and two paddles, as can be seen in Figure 7.7. The left paddle can be moved up and down by dragging anywhere on the left side of the screen. The application sends a new frame roughly 60 times per second, animating both the ball and the automatically playing right paddle. Touch data from the acceleration system is read with an equal frequency, used for executing the game logic. Note that the touch control of the left paddle is still updated with the maximum 120fps on the acceleration system. This demonstrates the host's ability to provide new frames independently of the speed at which the touch correction and display refreshes are performed. The application utilizes the following functionality:

- Increased touch area
- Interactivity *(Performed by Host)*
- Lower Frame-rate Animation *(Performed by Host)*

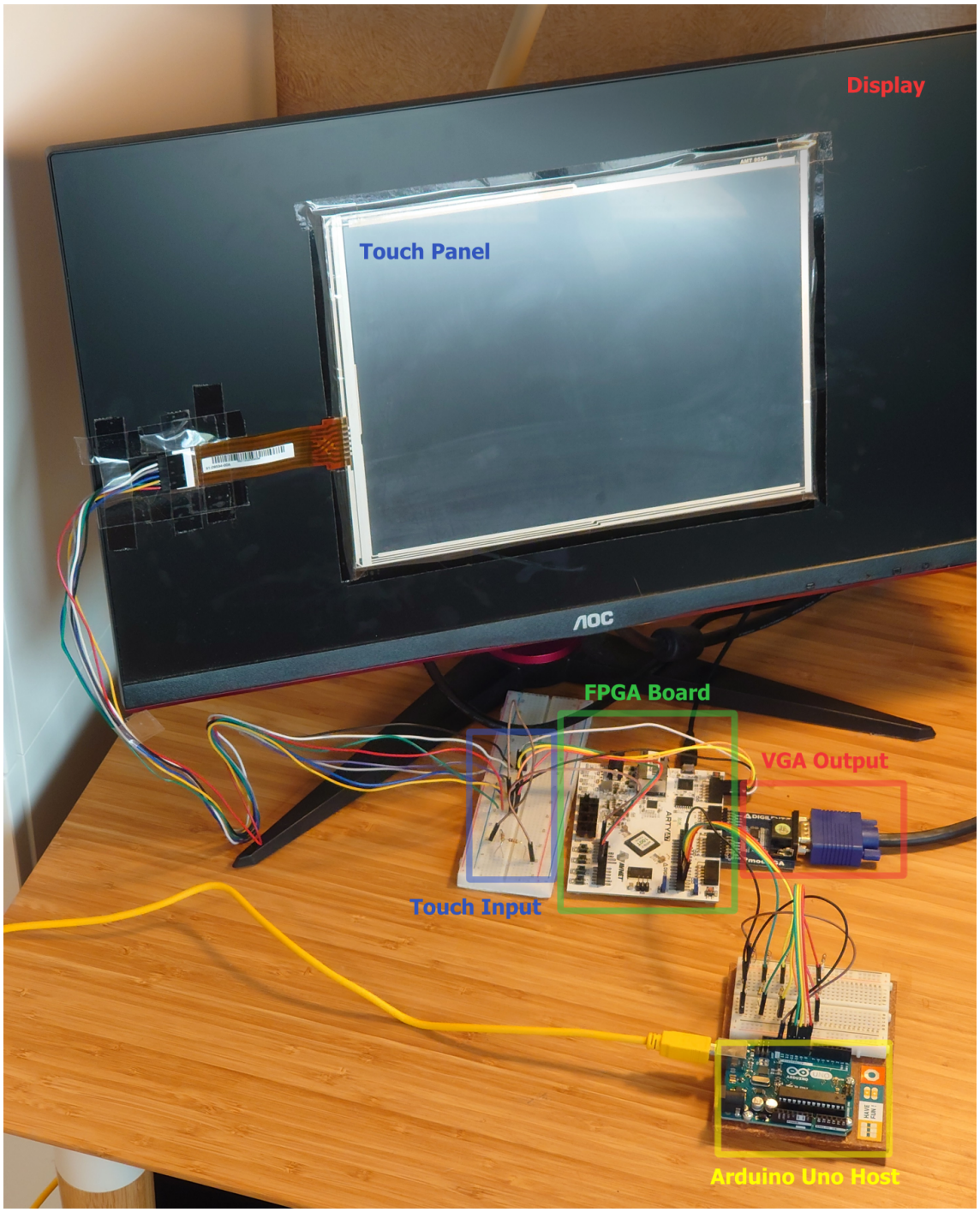

Figure 7.2: An overview of the test system with its separate parts highlighted.

Figure 7.3: The end-to-end latency test setup, filming different devices using a tripod mounted smartphone capable of 240fps video.

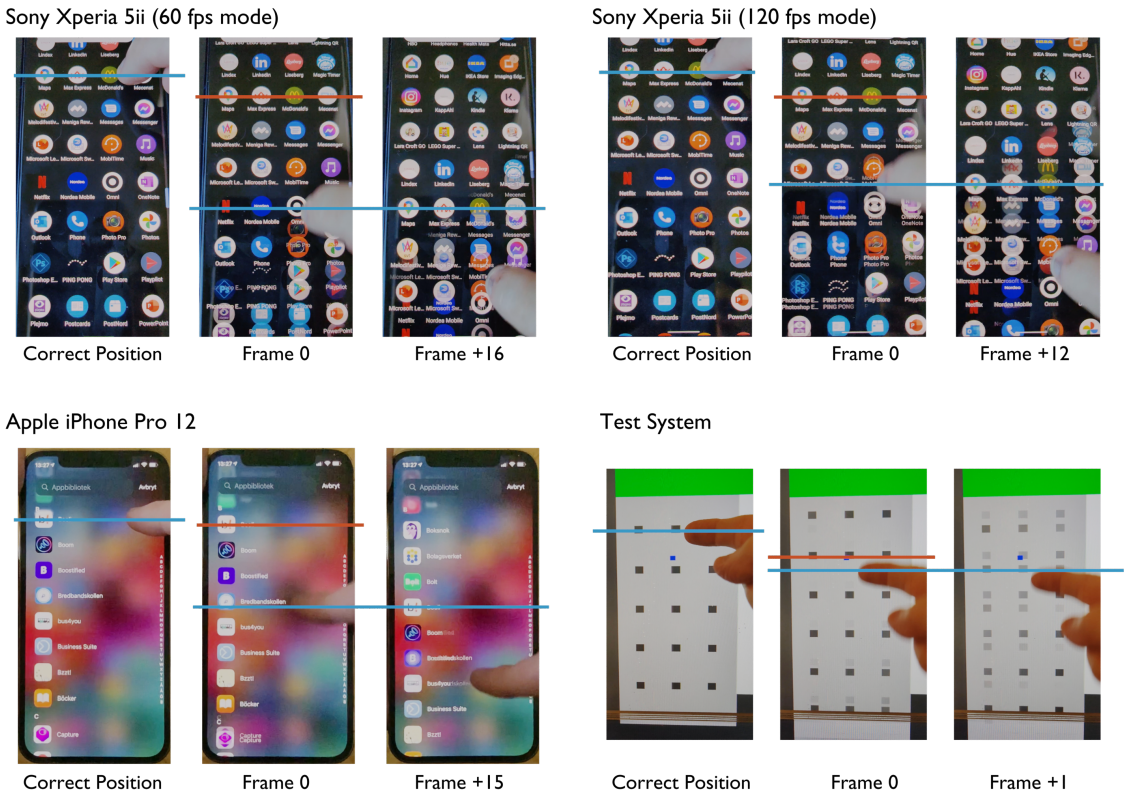

Figure 7.4: A measurement of the end-to-end latency on different devices. The left image shows the desired finger and content relation. The center image shows their position mid-swiping, dubbed frame 0. The right image shows the same swipe n video frames later, where the content appears in the location it should have had in frame 0.

Figure 7.5: The plastic piece placed on the finger during the demonstrations of the test applications for achieving more reliable touch input.

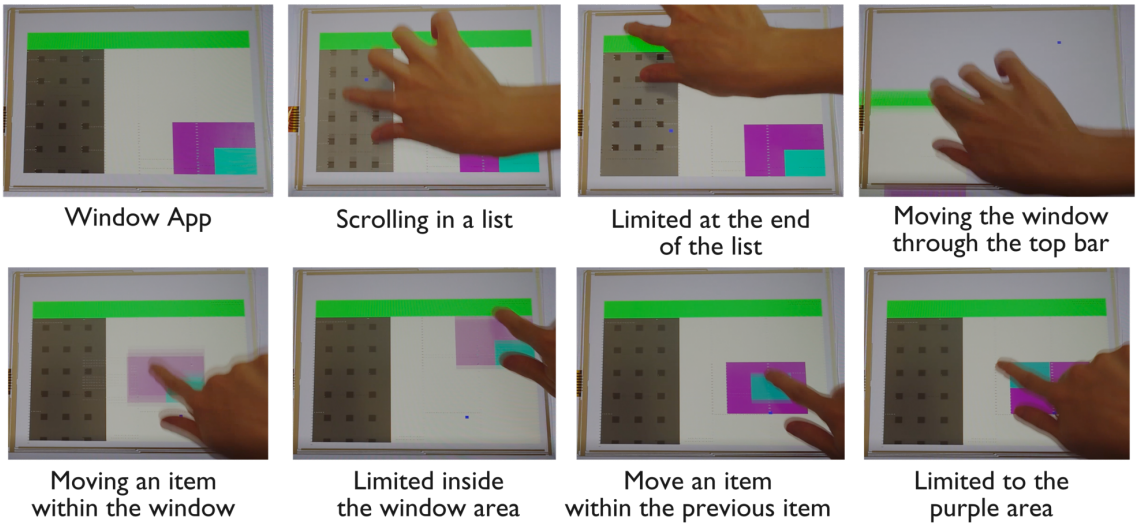

Figure 7.6: The top left corner shows the Window test application as on startup, while the rest of the images shows possible interactions.

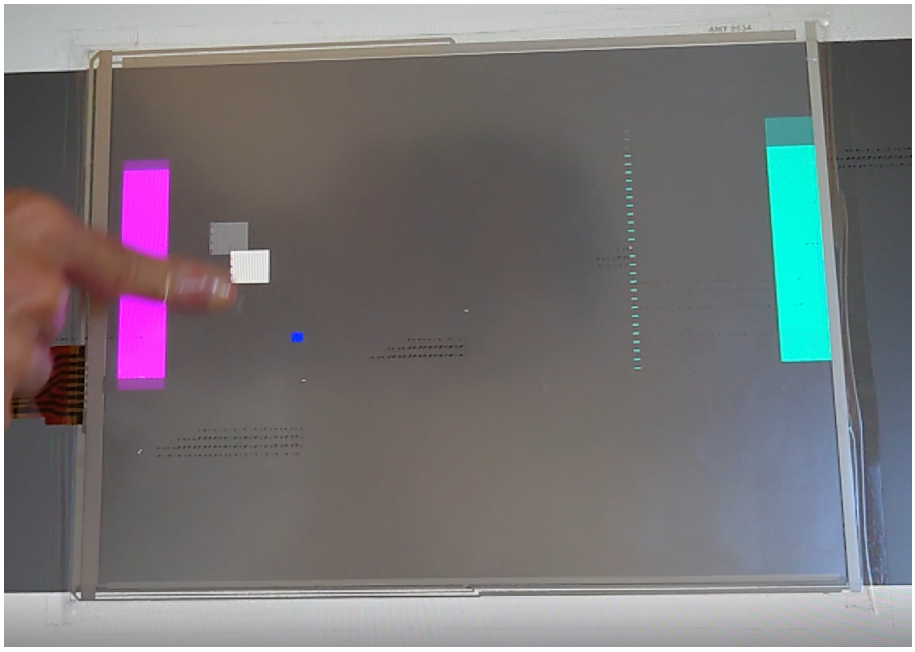

Figure 7.7: The Game demo application during playing.

8

Conclusion

This chapter discusses the results achieved and states the answers to the research questions, ending with a summarizing conclusion and suggestions for future work.

8.1 Discussion

In this section, different aspects of the results presented in Chapter 7 are discussed, including what conclusions can be drawn.

8.1.1 Supported Frame Rate and Resolution

The current test system supports 1024x768 at 120fps on an FPGA. Considerable performance gains are expected if this design is turned into an ASIC for use as an actual product. Going from an FPGA to an ASIC provides two significant differences in the types of design possible.

First, the amount of logic available for the design increases vastly. This increase can effortlessly be utilized for improved performance as the architecture is designed around several parallel pipelines. The number of pipelines can be increased by replication, processing more pixels at once, allowing for higher resolutions without affecting the achieved latency. For example, by multiplying the pipeline count by 11, the architecture could go from supporting the current 1024x768 to a 4K resolution.

Second, the clock frequency can be increased considerably in an ASIC. A frequency of roughly 1GHz in a final chip gives eight times as many cycles per second, allowing for roughly 1000fps compared to the current 120. This example shows that the initial design goal of 1000fps at high-resolutions is indeed possible from a processing standpoint. Note that this statement ignores any added complexities in fetching data from memory at higher rates, which are discussed in Section 8.1.5.

8.1.2 Achieved Latency

The achieved processing latency in the FPGA of $83\mu s$ is around 8.3% of the total time allowed for a 1ms latency. Although this might sound low, it is not insignificant. For a 1 ms latency, the generated image can only be shown for 1000-83=917μs

before it is out of date. This, in turn, increases the frame rate requirement to 1000/917=1090fps for a sample-and-hold display device (even without considering the latency of the input/output devices).

The introduction of an ASIC and higher clock frequencies could potentially improve these numbers. Increasing the frequency by eight for 1000fps support would bring the processing latency down to 1/8th, at 10.4μs, taking roughly 1% of the total time for achieving 1ms latency. This latency then increases the needed frame rate to just 1000/(1000-10.4)=1011fps.

It would be interesting to compare the achieved processing latency to a traditional system. Unfortunately, the processing latencies in real touch system are not available. It is also not possible to measure by merely utilizing the device. When looking at the visible end-to-end latency, as done in Section 7.2.2, it does not reveal where this latency resides, how much is due to processing, and how much is due to the input/output interfaces.

Real mobile devices utilize far more advanced and reliable touch input through complex touch controllers. As the focus of this thesis is not the touch controller itself, the utilization of the touch panel is fast and simple, with common issues of incorrectly detected touches. As it is unknown how large a portion of the end-to-end latency relates to creating reliable input, the amount of improvement provided by the touch acceleration system is unclear. This limitation means that no direct conclusions can be drawn from the end-to-end latency measurements performed in Table 7.1, comparing the test system to current high-end mobile devices. Instead, these tests should be seen as a demo of the feasibility behind the idea, indicating the kind of latencies that would be possible when combined with low-latency input/output interfaces.

Realistically, 83μs/10μs is a lot lower than the processing latency of current systems, as many applications commonly spend quite some time calculating each frame before it is rendered, even without considering the OS and other software-induced latencies. However, no conclusions can be drawn on the actual differences. What can be said is that the suggested approach detaches the perceived processing latency from the complexity of the application and state of the host, meaning that equally good touch interactions are guaranteed to be achieved at all times. This point is illustrated in Figure 8.1, where the time spent by the OS and the application does no longer directly relate to the perceived latency.

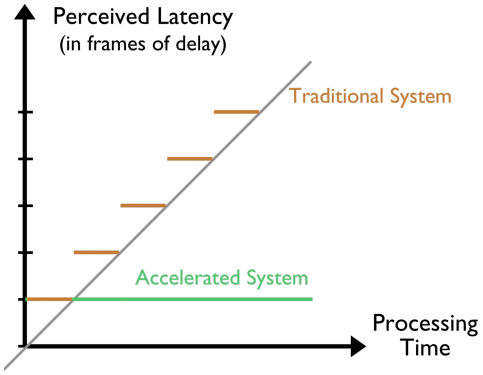

Figure 8.1: An illustration on the effect of longer time spent processing, either in OS or the application, has on the perceived latency of the touch input. As soon as the processing is not complete when the frame is rendered, the result is not visible until the next frame, giving the stair-case appearance of the traditional system.

8.1.3 Supported Complexity

The calculations on the supported complexity should be seen as rough estimates. The exact complexity limitations are hard to reason about as multiple units are involved, behaving differently depending on the exact content being displayed. Therefore, a well-calculated safety margin should be applied in a real-life system to minimize the risk of incorrectly displayed data ever appearing.

One could argue that it would be better to design the system to allow for increased latency when the processing does not finish on time, making sure a user never is presented with broken graphics. Support for this could be realized through storing the complete framebuffer and displaying the previous frame until the next one is completed. However, doing so allows the host applications to ignore the limitations as slightly longer latencies are not directly apparent to the user in the same way as broken graphics. By making it a strict deadline, hosts of the acceleration system are forced to use it correctly, guaranteeing low latencies at all times.

Regardless of the complexity supported in absolute numbers, it should be noted that the architecture provides a direct relationship between the supported complexity and the latency achieved. This allows any specific implementation to make adjustments if the currently supported complexity is deemed insufficient, at the cost of a longer processing latency along with more chip area. Consequently, with the introduction of an ASIC, it would be possible to spend some of the increased clock frequency on increasing the supported complexity instead of only decreasing the latency.

8.1.4 Energy Consumption

The energy consumption of an acceleration system is of the highest concern due to the prevalence of touch input in mobile devices. An essential factor in a system's energy consumption is the amount of data it interchanges with memory. It is especially relevant here as all data is only used once per frame, requiring constant

streaming. At high resolutions and frame rates, this results in large amounts of data being transferred during each frame. The effects of memory usage and ways to limit it to save energy are discussed in Section 8.1.5.

Regardless, it is unavoidable that higher frame rates do increase energy consumption since more data is handled. However, it is worth considering that reasonably high power consumption during touch interaction (where a maximized frame rate would be utilized) might be a fair trade-off for the improved experience. Users do not have the finger on screen at all times but instead spend a lot of time reading/watching the content on screen. Also, with a priority-driven acceleration system present, static content on screen could potentially be rendered more efficiently than currently, where each image is often rendered from scratch at a set frame rate. This minimizing of re-rendering could potentially make the device's energy consumption the same or lower, despite high frame rates during interaction.

8.1.5 Memory Utilization

The test architecture is designed for a texture memory latency of 50 cycles, as detailed in Section 6.1.3. However, as this support is achieved by having queues of pixels in progress, larger latencies can be supported as long as there is enough chip area for increased queue sizes. A GPU memory, which provides the massive bandwidth required for this type of application, can have a latency of 1000 cycles or more [27]. However, even latencies of that magnitude have a relatively small effect on the total processing latency, as the pipeline depth only matters on start-up. Adding a 1000 cycle wait-time for data to arrive is only 1000/10752=9.3% of the total rendering time available, requiring the supported complexity to be decreased by a corresponding amount.

Although pre-fetching gives a consistent and predictable pipeline execution, and therefore processing latency as required, it has a potentially dire impact on energy consumption. As texel values are fetched before they are known to contribute to the final image, large amounts of fetched data will not be used. This effect would worsen the more covered-up areas there are in the finished image.

Ideally, texels should only be fetched when they are guaranteed to contribute to the final image. Support for this is already in place in the test architecture by not utilizing prefetching, as the texel value is not read if the framebuffer pixel is already completed. However, this approach only supports low latencies of a handful of cycles, which are hidden through the existence of multiple Pixel Processing Units. Therefore, it is not a realistic approach without advanced low-latency memory present.

Instead, ways to reduce the unnecessary data while still utilizing prefetching should be investigated. One way is to read the framebuffer pixel right before prefetching and check if it is already completed. If so, it can be dropped directly without affecting the finished image. However, as other pixels are in progress in the queue ahead, the framebuffer pixel could be finished before the current pixel reaches the

Pixel Processing Unit, in which the requested texel will be discarded without being used. Still, this should mitigate most unnecessary texel fetches.

Note that if the framebuffer pixel value is read and stored early, long before it is used, it could be modified by another pixel ahead in the queue, creating a read-after-write problem. This problem could potentially be fixed by letting the pixels in the queue snoop framebuffer writes and grab a newer value if available.

Another way to reduce unnecessary texel fetches would be to look at what area is being modified compared to the previous frame. As the triangles are displaced early in the pipeline, this information would be available during the prefetching to decide which texels should be fetched. By storing a complete framebuffer, the unchanged areas could prefetch finished pixel values directly for usage instead of the texels it consists of, allowing for a single, guaranteed used, fetch per pixel output. This would be especially helpful with transparent areas where each framebuffer pixel requires the values of multiple texels.

The available bandwidth is also of concern in a finished system. As the frame rate and resolution increase, more pressure is put on the memory system to provide the Pixel Processing Units with all necessary texel data. Unfortunately, caches are inefficient since the texel data is only used once per frame. Instead, all data must be continuously streamed in.

A modern high-end GPU has a memory bandwidth of over 900GB/s [28], equating to a texel count that could fill the entire display with 25 layers at 4K 1000fps. Thus, from a strict possibility standpoint, it is clear that the raw memory bandwidth would not be a limiting factor in a real implementation of the system in a high-end stationary use case.

It seems possible to achieve similar frame rates on high-end mobile devices from a memory standpoint as well, although with less certainty. At an available bandwidth of over 40GB/s, it allows for more than a full layer in 4K, with commonly used lower resolutions increasing the supported complexity to around four layers[29]. However, the thermal budget must be shared with the regular GPU and other components, potentially making the bandwidth available to the touch acceleration unit considerably lower.

8.1.6 Test System

The original intent in the sizing of the touch panel was to match the resolution it covered on the display. At 12.1 inches 4:3 aspect ratio, it would have fit a 1:1 display of 1024x768 on a 24 inch 16:9 display reasonably well. However, unlike expected, the low-latency mode of the chosen display stretches the image to fill the display partially instead of outputting the received image unscaled at 1:1. This behavior resulted in the displayed image stretching far beyond the touch panel, not allowing touch input on the entire displayed image. The test applications were adjusted to mitigate this, so all necessary touch input was present within the touch area. This

adjustment does not affect any conclusions drawn. The effect is merely a smaller resolution of the total rendered area being available for direct interaction, having no effect on the complexity in what is being rendered.

Although discussed and motivated in more detail in Section 8.1.2, it is worth noting once again that no direct conclusions can be drawn regarding the comparison of end-to-end latency in Table 7.1. Instead, these measurements should be seen as a demonstration of the feasibility of the proposed approach, indicating the latencies that would be possible at the refresh rates already available on devices today (when combined with low-latency input/output interfaces).

8.2 Research Questions Answers

In this section, the answers to the research questions detailed in Section 1.4 are provided.

What type of touch interactions should be supported by an input correction system?

What features are required to allow for the correction of said interactions?

> These are presented in the left and right column respectively in Figure 4.1 in Chapter 4.

How should the image data be structured to allow for the needed image manipulations?

> In order to support direct interaction with all options present on screen, the information of individual objects, their location on screen, and input settings must be provided by the host. In addition, each interactable object needs to be stored as a separate RGBA image, essentially a separate texture in memory, to allow for the generation of correct final images after manipulating the objects. The host should be responsible for creating the correct 1:1 textures for all objects, alleviating the acceleration system from concerns such as programmable shaders, texture filtering, and anti-aliasing.
>
> These considerations are explained in more detail in Chapter 5.

How should the hardware architecture be structured to support these requirements?

> For minimum latency, only a small part of the image should be processed at a time to allow for finished pixels shortly after the received input. Also, each triangle should be stored with all the relevant settings concerning the object to which it belongs. This duplication of data allows for fast and predictable triangle dispatching as all data needed is present from the start of the pipeline, allowing the processing to start without delay.

To alleviate the acceleration system, the host should deliver all triangles in order to prevent any sorting from having to be performed. Instead, all pipelines are designed to guarantee that pixel values are written to the framebuffer in the correct order. As merging multiple overlapping objects requires continuous writing to the same framebuffer pixels, the active section of the framebuffer is kept on-chip. Also, multiple pixels must be processed in parallel to allow for enough time per pixel.

Double buffering must be utilized to support both asynchronous receiving of frame data from the host and asynchronous rendering and sending of final images to display.

These considerations are explained in more detail in Chapter 6.

How low latency is achieved?

The processing latency of the proposed architecture on an FPGA is $83\mu s$. This latency is potentially decreased proportionally by the increase in clock frequency provided by utilizing an ASIC, depending on if the increased speed is utilized for lower latency or supporting rendering of more complex scenes.

The test system shows an end-to-end latency of 4ms (+/-4) compared to current high-end mobile devices at around 50ms (+/-4). However, no conclusions can be drawn from comparing these numbers as modern devices utilize more complex and reliable touch detection, increasing the input latency by unknown amounts.

Can the suggested architecture support 1000 fps in high resolutions?

Most likely, when converted to an ASIC. The FPGA supports 120fps at 130MHz, which equates to 1000fps on an ASIC at roughly 1GHz. However, some paths might have to be optimized to achieve that frequency. Consequently, 4K resolution can be supported by multiplying the pipelines and framebuffer section size by 11, which is also achievable on an ASIC (and more expensive FPGAs).

The effect this has on the memory system needs further investigation. However, the memory bandwidth of a modern graphics card indicates that a dedicated system should be able to feed the acceleration unit with enough texel data for processing. Latency-wise, it might be necessary to employ pre-fetching instead of the bandwidth/energy limiting technique of only fetching texel values known to contribute to the final image.

How does the experience compare to traditional touch systems?

Most common touch interactions are supported, and the utilization of

the test applications confirms that it is suitable for various uses. Subjectively speaking, there is nothing in the touch acceleration that deters from the experience when compared to traditional rendering, from what can be noticed in the simple test system. Furthermore, when the test system works well, the lower latency during object manipulating gives a significantly better experience than current mobile devices. This indicates a possibility for the suggested approach to improve the touch experience on devices utilizing the typical refresh rates of today, and not only on future displays with the long-term goal of 1000fps. However, the actual touch detection is unreliable due to a very simple touch controller implementation, making the test system's performance unsuitable for any real use.

8.3 Conclusion

This thesis investigated the creation of an acceleration system for reducing latency in touch input by correcting rendered images produced by the main system with the latest input. The work consisted of specifying the required input functionality and create a proposal hardware architecture supporting these requirements. The hardware architecture was realized on an FPGA card as part of a test system with two applications demonstrating common user interactions.

This work shows the architectural feasibility of supporting very low-latency processing in the context of touch input while presenting high fidelity graphics.

Accelerating touch input should see more and more interest as the touch sampling and frame rates on devices continue to increase, even though it could potentially provide significant latency and energy improvements to devices already in use today. The main question is how this approach could be incorporated seamlessly into the way touch systems are currently organized and utilized. Ease of use is critical for widespread adoption, especially from a software perspective, as that reaches beyond the device manufacturers and involves creators of touch applications worldwide.

8.4 Limitations

Comparison with current touch devices is difficult as they are delivered as complete packages, with little to no information on the system's individual stages. Therefore, the processing latency achieved can not easily be compared to others systems. Extra complexity comes from the variable nature of the processing latency in traditional systems, which on the other hand, favors the result of the constant latency presented in this thesis.

All discussions on lowering energy consumption are based on general design principles, such as minimizing data fetches from off-chip memory and reducing work. No actual measuring or comparison of the energy consumption as compared to tradi-

tional systems has been performed. Doing so needs proper integration into a real system as it would affect how images are rendered on host and, therefore, total energy consumption.

8.5 Future Work

This thesis has a large amount of potential future work. Following are several suggestions for aspects to investigate or improve:

- Adding support for full-resolution textures with unique data for every pixel on the screen, incorporating the use of off-chip memory in the acceleration system. This work might include optimizations such as compression of texels during transmission and storage, with an analysis of the maximum content resolutions supported.

- Incorporating all suggested functionality in Chapter 4, supporting multi-touch and operations such as scaling and rotation, with further expansions on how to realize them as a part of the hardware architecture.

- Investigating how the acceleration system would be best incorporated as an extension of a traditional system, minimizing the complexity for both implementation and usage. This includes how current GPUs and graphics APIs could potentially generate all the necessary host data before transfer to the acceleration system.

- Properly analyzing the suggested approach's energy effect compared to rendering all frames from scratch in a traditional system. For a fair comparison, work incorporating with traditional systems is required as it affects the energy consumption when generating the necessary data on host.

- Another potential optimization is to store the entire framebuffer for direct fetching instead of regenerating everything in each frame. It could give significant energy improvements with static content and overlapping areas that require multiple texel values to be fetched per final pixel. It could also allow for more efficient pre-fetching of texels, as it is known on an area basis which parts of the image will be re-rendered early in the pipeline.

- Lastly, supporting animation of object's position on screen should be examined for high frame rate motion while further decreasing the frequency in which the host has to send new frame data.

Bibliography

[1] A. Ng and P. H. Dietz, "The effects of latency and motion blur on touch screen user experience," *Journal of the Society for Information Display*, vol. 22, no. 9, pp. 449–456, 2014. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/jsid.243

[2] M. Research, "Applied sciences group: High performance touch," [Video file, time 01:37] Available at https://www.youtube.com/watch?v=vOvQCPLkPt4 (2021/03/13).

[3] A. Ng, J. Lepinski, D. Wigdor, S. Sanders, and P. Dietz, "Designing for low-latency direct-touch input," in *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 453–464. [Online]. Available: https://doi.org/10.1145/2380116.2380174

[4] F. Peng, H. Chen, F. Gou, Y.-H. Lee, M. Wand, M.-C. Li, S.-L. Lee, and S.-T. Wu, "Analytical equation for the motion picture response time of display devices," *Journal of Applied Physics*, vol. 121, no. 2, 2016. [Online]. Available: https://doi.org/10.1063/1.4974006

[5] D. Binks, "Dynamic resolution rendering," Available at https://software.intel.com/content/www/us/en/develop/articles/dynamic-resolution-rendering-article.html (2020/12/12).

[6] G. Denes, K. Maruszczyk, G. Ash, and R. K. Mantiuk, "Temporal resolution multiplexing: Exploiting the limitations of spatio-temporal vision for more efficient vr rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 25, no. 5, pp. 2072–2082, 2019.

[7] A. Patney, M. Salvi, J. Kim, A. Kaplanyan, C. Wyman, N. Benty, D. Luebke, and A. Lefohn, "Towards foveated rendering for gaze-tracked virtual reality," *ACM Trans. Graph.*, vol. 35, no. 6, Nov. 2016. [Online]. Available: https://doi.org/10.1145/2980179.2980246

[8] L. Yang, Y.-C. Tse, P. V. Sander, J. Lawrence, D. Nehab, H. Hoppe, and C. L. Wilkins, "Image-based bidirectional scene reprojection," *ACM*

Trans. Graph., vol. 30, no. 6, p. 1–10, Dec. 2011. [Online]. Available: https://doi.org/10.1145/2070781.2024184

[9] M. Regan and R. Pose, "Priority rendering with a virtual reality address recalculation pipeline," in *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '94. New York, NY, USA: Association for Computing Machinery, 1994, p. 155–162. [Online]. Available: https://doi.org/10.1145/192161.192192

[10] W. R. Mark, G. Bishop, and L. McMillan, "Post-rendering image warping for latency compensation," USA, Tech. Rep., 1996.

[11] W. R. Mark, L. McMillan, and G. Bishop, "Post-rendering 3d warping," in *Proceedings of the 1997 Symposium on Interactive 3D Graphics*, ser. I3D '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 7–ff. [Online]. Available: https://doi.org/10.1145/253284.253292

[12] P. P. Dean Beeler, Ed Hutchins, "Asynchronous spacewarp," Available at https://developer.oculus.com/blog/asynchronous-spacewarp/ (2020/12/12).

[13] M. Antonov, "Asynchronous spacewarp examined," Available at https://developer.oculus.com/blog/asynchronous-timewarp-examined/ (2021/02/23).

[14] N. Henze, M. Funk, and A. Shirazi, "Software-reduced touchscreen latency," 09 2016, pp. 434–441.

[15] G. Bishop, H. Fuchs, L. McMillan, and E. J. S. Zagier, "Frameless rendering: Double buffering considered harmful," in *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '94. New York, NY, USA: Association for Computing Machinery, 1994, p. 175–176. [Online]. Available: https://doi.org/10.1145/192161.192195

[16] J. M. P. van Waveren, "The asynchronous time warp for virtual reality on consumer hardware," in *Proceedings of the 22nd ACM Conference on Virtual Reality Software and Technology*, ser. VRST '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 37–46. [Online]. Available: https://doi.org/10.1145/2993369.2993375

[17] R. Smith, "The nvidia geforce gtx 1080 gtx 1070 founders editions review: Kicking off the finfet generation - preemption improved: Fine-grained preemption for time-critical tasks," Available at https://www.anandtech.com/show/10325/the-nvidia-geforce-gtx-1080-and-1070-founders-edition-review/10 (2021/03/23).

[18] A. G. Dean Beeler, "Asynchronous timewarp on oculus rift," Available at https://developer.oculus.com/blog/asynchronous-timewarp-on-oculus-rift/ (2020/12/12).

[19] F. A. Smit, R. van Liere, and B. Fröhlich, "The design and implementation of a vr-architecture for smooth motion," in *Proceedings of the 2007 ACM Symposium on Virtual Reality Software and Technology*, ser. VRST '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 153–156. [Online]. Available: https://doi.org/10.1145/1315184.1315212

[20] P. Lincoln, A. Blate, M. Singh, T. Whitted, A. State, A. Lastra, and H. Fuchs, "From motion to photons in 80 microseconds: Towards minimal latency for virtual and augmented reality," *IEEE Transactions on Visualization and Computer Graphics*, vol. 22, no. 4, pp. 1367–1376, 2016.

[21] S. Friston, A. Steed, S. Tilbury, and G. Gaydadjiev, "Construction and evaluation of an ultra low latency frameless renderer for vr," *IEEE Transactions on Visualization and Computer Graphics*, vol. 22, no. 4, pp. 1377–1386, 2016.

[22] J. Pineda, "A parallel algorithm for polygon rasterization," in *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '88. New York, NY, USA: Association for Computing Machinery, 1988, p. 17–20. [Online]. Available: https://doi.org/10.1145/54852.378457

[23] M. E. Newell, R. G. Newell, and T. L. Sancha, "A solution to the hidden surface problem," in *Proceedings of the ACM Annual Conference - Volume 1*, ser. ACM '72. New York, NY, USA: Association for Computing Machinery, 1972, p. 443–450. [Online]. Available: https://doi.org/10.1145/800193.569954

[24] "Rasterization rules," Available at https://docs.microsoft.com/en-us/windows/win32/direct3d11/d3d10-graphics-programming-guide-rasterizer-stage-rules (2021/4/8).

[25] "Gpu framebuffer memory: Understanding tiling," Available at https://developer.samsung.com/galaxy-gamedev/resources/articles/gpu-framebuffer.html (2021/3/24).

[26] T. Porter and T. Duff, "Compositing digital images," in *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '84. New York, NY, USA: Association for Computing Machinery, 1984, p. 253–259. [Online]. Available: https://doi.org/10.1145/800031.808606

[27] X. Mei and X. Chu, "Dissecting gpu memory hierarchy through microbenchmarking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 72–86, 2017.

[28] "Nvidia rtx 390 specs," Available at https://www.techpowerup.com/gpu-specs/geforce-rtx-3090.c3622 (2021/5/20).

[29] "Qualcomm adreno 650 specs," Available at https://gadgetversus.com/graphics-card/qualcomm-adreno-650-specs/ (2021/5/20).