Assignment of bachelor's thesis

Title:	Virtual mirror effect for museums
Student:	Richard Boldiš
Supervisor:	Ing. Jan Buriánek
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Computer Graphics
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

Incorporating digital exhibits into modern museums' exhibitions has become a widespread practice for many good reasons. Instead of passively absorbing information, visitors become part of the exhibition, which provides them with a whole new experience. This work aims to create a digital exhibit that captures the observer's area with a camera and uses that as a source for the moving texture on a 3D model in real-time. This digital exhibit shall be a generalized virtual mirror that serves as the base for similar techniques such as virtual water surfaces, anamorphic imaging, etc.

1. Explore platforms appropriate for implementing a virtual mirror, especially on mini-computers suitable for deployment in exposures.
2. Design the architecture for the virtual mirror for the selected platform.
3. Implement a prototype virtual mirror application and test it in actual museum operation.
4. This thesis shall contain a detailed description of the implementation and annotated source code.

Electronically approved by Ing. Radek Richtr, Ph.D. on 20 April 2022 in Prague.

Bachelor's thesis

Virtual mirror effect for museums

Richard Boldiš

Department of Software Engineering
Supervisor: Ing. Jan Buriánek

May 11, 2022

Czech Technical University in Prague
Faculty of Information Technology
© 2022 Richard Boldiš. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Boldiš, Richard. *Virtual mirror effect for museums.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on May 11, 2022 . . . .

Abstrakt

Digitální zařízení již nejsou neobvyklou součástí standardních expozic moderních muzeí. Vedle rozvoje grafických aplikací si digitální zařízení získávají stále větší oblibu díky bezproblémovému začlenění do nejrůznějších scén a prostředí, nemluvě o jejich nekonečném potenciálu pro interaktivitu s návštěvníky. Širšímu využití těchto zařízení však někdy brání jejich relativně vysoká cena, a proto se často hledá jejich cenově výhodná náhrada. Jedním z řešení je nalezení vhodného hardwaru, který splňuje požadavky na výkonnost jednotlivých projektů. Většinou toho lze dosáhnout pouze pomocí chytrého a na míru šitého softwarového balíčku, který v ideálním případě stojí jen zlomek jeho alternativ. V této práci bylo vytvořeno virtuální zrcadlové zařízení s levným minipočítačem, připomínajícím obraz odrazu vody. Tvoří kulisu moderní muzejní expozice tím, že snímá své bezprostřední okolí a na výstupu vytváří upravený živý obraz, čímž pro návštěvníky vytváří iluzi, jako by pozorovali svůj vlastní odraz na vodní hladině. Tato práce popisuje návrh softwarového řešení a techniku živého zpracování zachycených obrazů, které umožňují vybranému minipočítači zamýšlené provedení.

Klíčová slova digitální exponáty, virtuální zrcadlo, vestavěné systémy, Linux, NVIDIA Jetson Nano, EGL, OpenGL ES, sdílená paměť, zpracování živého vstupu z kamery

Abstract

Digital devices are no longer an unusual part of modern museums' standard exhibitions. Alongside the development of graphical applications, the seamless incorporation of digital devices into all kinds of stages and settings, not to mention its endless potential for interactiveness with the visitors, has gained them increasing popularity. However, the relatively high cost of such devices sometimes hinders their wider usage; thus, a cost-effective substitute is often sought after. One solution is to find appropriate hardware that fulfills performance requirements for individual projects. Most of the time, it can only be achieved with a clever and tailored software stack, ideally costing only a fraction of its alternatives. This work created a virtual mirror device with a low-cost mini-computer, resembling water reflection images. It forms the setting of a modern museum exhibition by capturing its immediate surroundings and outputting a modified live image, generating an illusion for the visitors as if they are observing their own reflection on a water surface. This thesis describes the software stack's design and the live processing technique of captured images that enable the chosen mini-computer for intended performance.

Keywords digital exhibits, virtual mirror, embedded systems, Linux, NVIDIA Jetson Nano, EGL, OpenGL ES, shared memory, live camera feed processing

Contents

Introduction **1**
Goals . . . 2

1 Analysis **3**
1.1 Virtual mirror . . . 3
1.2 Requirements . . . 4
1.2.1 Performance . . . 4
1.2.2 Cooling . . . 4
1.2.3 Durability . . . 4
1.2.4 Recovery . . . 5
1.2.5 Dimensions . . . 5
1.3 Computer device . . . 5
1.3.1 System on Chip . . . 5
1.3.2 System on Module . . . 6
1.3.3 Candidates . . . 6
1.3.3.1 Raspberry Pi 4 B . . . 7
1.3.3.2 NVIDIA Jetson Nano . . . 7
1.3.4 Conclusion . . . 8
1.4 Camera . . . 9
1.4.1 Sony IMX219 . . . 10

2 Design **11**
2.1 Digital exhibit . . . 11
2.2 Software . . . 12
2.2.1 Operating system . . . 12
2.2.1.1 Jetson Linux . . . 13
2.2.1.2 Vendor-provided SD card image . . . 14
2.2.2 Windowing System . . . 14
2.2.2.1 X Window System . . . 14

2.2.2.2 Embedded-System Graphics Library 14
2.2.3 Graphics API 14
2.2.3.1 OpenGL ES 15
2.2.4 Libraries 15
2.2.4.1 OpenGL Mathematics 15
2.2.4.2 OpenGL Image 15
2.2.4.3 Open Asset Import Library 16
2.2.4.4 POCO C++ Libraries 16
2.2.5 Platform-specific libraries 16
2.2.5.1 NVIDIA Tegra Multimedia API 16
2.2.6 Container formats 17
2.2.6.1 DirectDraw Surface 17
2.2.6.2 Filmbox 17

3 Proof of concept **19**
3.1 OS optimization 19
3.1.1 The approach 19
3.1.1.1 Stripping 19
3.1.1.2 Bootstrapping 20
3.1.2 Minimal system installation 20
3.1.3 NVIDIA packages 21
3.1.4 Emulating the rootfs 23
3.1.5 Cleaning up 23
3.1.6 Packaging the rootfs 24
3.1.7 Partition table 24
3.1.8 Populating the partitions 26
3.1.9 Automation 27
3.1.10 Results 27
3.2 Development environment 28
3.2.1 Physical exhibit simulation 28
3.2.2 Remote development 30
3.3 Implementation 30
3.3.1 Project structure 30
3.3.1.1 Addons 30
3.3.1.2 Application 31
3.3.1.3 Graphics 31
3.3.1.4 Math 31
3.3.1.5 Utils 31
3.3.2 Error handling 31
3.3.2.1 Assertions 32
3.3.2.2 Exceptions 32
3.3.2.3 Logging 32
3.3.3 Initialization 33
3.3.3.1 Application 33

3.3.3.2 Window . . . 33
3.3.3.3 Graphics context . . . 33
3.3.3.4 Physical camera . . . 34
3.3.3.5 Scene . . . 35
3.3.4 Main loop . . . 35
3.3.4.1 Retrieving camera frames . . . 36
3.3.4.2 Rendering . . . 38
3.3.4.3 Synchronization . . . 38
3.3.4.4 Flushing . . . 38
3.3.5 Rendering system . . . 38
3.3.5.1 Mesh . . . 39
3.3.5.2 Shaders . . . 39
3.3.5.3 Renderers . . . 40
3.3.6 Scene . . . 40
3.3.6.1 Walls . . . 40
3.3.6.2 Water . . . 40
3.3.6.3 Camera frame . . . 41
3.4 Deployment . . . 41
3.4.1 Overheating . . . 43
3.4.2 Glossy display surface . . . 43
3.4.3 Calibration . . . 44

Conclusion **45**
Future improvements . . . 46
Camera latency . . . 46
Wayland . . . 46

Bibliography **47**

A Acronyms **51**

B Contents of enclosed CD **53**

C NVIDIA L4T packages **55**

D SD card image build automation structure **57**

E Rendering pipeline **59**

F 3D model **61**

G Development model view **63**

H Museum exhibit dimensions **65**

I Device dimensions **67**

List of Figures

1.1 SoM with its carrier board 6
1.2 Raspberry Pi 4 B 7
1.3 Raspberry Pi 4 B specifications 8
1.4 NVIDIA Jetson Nano Developer Kit 8
1.5 Jetson Nano development kit specifications 9
1.6 Sony IMX219 sensor capture modes 10

2.1 Natural mirrors 12
2.2 Exhibit design 13
2.3 Graphics APIs supported by Linux for Tegra 15
2.4 Shared memory 17

3.1 Removing the desktop environment 20
3.2 Bootstrapping the foreign architecture rootfs 21
3.3 Directory structure after bootstrapping the rootfs 22
3.4 NVIDIA repositiory configuration 22
3.5 NVIDIA packages 23
3.6 Emulation of binaries in a foreign rootfs 23
3.7 Cleaning up APT and logs 24
3.8 Usage of `flash.sh` 25
3.9 Jetson Nano SD card partitions 25
3.10 Allocating the SD card image 26
3.11 Adding partition to GPT table 26
3.12 Populating the SD card partition 27
3.13 Comparison of SD card image size 28
3.14 Comparison of resources utilization 29
3.15 Exhibit development version 29
3.16 Application layers 31
3.17 Logging macro usage 33
3.18 Graphics context requirements 34

3.19 Initializing scene objects 35
3.20 Camera frame acquire thread 37
3.21 Creating OpenGL ES texture from shared frame buffer 37
3.22 Walls of the well 41
3.23 Distortion using DuDv map 41
3.24 Museum exhibit 42
3.25 Jetson Nano placement 42
3.26 Application deployed into exhibit 43
3.27 Visible reflection on display plastic protection 44
3.28 Project structure 44
3.29 A secondary camera capturing the camera-to-screen latency 46

Introduction

Incorporating digital exhibits into modern museums' exhibitions has become a widespread practice for many good reasons. Instead of passively absorbing information, visitors become part of the exhibition, which provides them with a whole new experience. Digital technologies appear in museums on all kinds of topics, from prehistoric, anthropology, and natural museums to modern, cutting-edge computer museums.

Digital exhibition using a virtual mirror is one of the many instances of interactive applications that alter the observer's area or blend it with a virtual 3D scene. The exhibition's computer system must instantaneously pick up the live camera input from its ever-changing surroundings and present graphical output in a reasonable time frame. This level of responsiveness is demanding for the supporting hardware. Thus a prior evaluation of its criteria is necessary.

Industrial computer systems with hardware-accelerated graphics can be overkill for simple interactive applications that do not demand high-end features, resulting in increased expenses for hardware and power consumption. Some of the recent hobbyist mini-computers offer impressive graphics performance and are capable of hardware-accelerated video processing, 3D graphics, and even artificial intelligence. Most of them have various IO options which allow them to connect to other devices such as industrial cameras and sensors. Such powerful and extendable characteristics give hobbyist mini-computers unlimited potential for a wide range of applications.

In the current project, the author attempts to create a virtual mirror that resembles the water surface of a well. The well is situated in a museum exhibition that tries to reconstruct the central European village life in the Middle Ages as a part of its setting. The virtual mirror, i.e., a rectangle monitor that displays the water surface, is physically installed at the bottom of a deep and narrow cavity that simulates a well. When visitors approach the well's opening, the camera installed at the bottom of the cavity captures the visitors' images and renders them on the monitor together with a 3D water effect to create an illusion for the visitors as if they see their reflections on the

water surface. The author considered a few options and ended up choosing NVIDIA Jetson Nano because this mini-computer's spectrum can meet the required criteria for this project while being portable and cost-efficient at the same time.

Goals

This work aims to

- create an efficient foundation for implementing digital exhibition applications that operate for many hours daily;
- implement a virtual mirror application that graphically modifies camera input and presents it as a part of the exhibition;
- incorporate the finished product as a part of a museum exhibition.

Chapter 1

Analysis

1.1 Virtual mirror

Digital devices used in modern exhibition is manifold, and a virtual or artificial mirror is one such device. As its name indicates, it resembles a mirror's function by displaying the appropriate image on a screen. In reality, how it achieves this effect cannot be more different than an ordinary mirror.

A mirror is a wave reflector that reflects light waves, which we perceive as an image. Light waves are reflected from the mirror surface at the same angle yet opposite from which they strike the reflecting surface. Therefore, when we look at a mirror, we see a mirrored image of ourselves and the objects that stand in front of the mirror. Depending on the incident angle, we can even perceive objects that are out of our field of view (FOV), such as objects around a corner. Mirrors are one of the essential tools we use in our everyday life.

A virtual mirror creates a mirror-like effect by displaying an image most commonly captured by a digital camera. Unlike an ordinary mirror, it does not have to provide an authentic reflection of the physical world, which opens the door of great potential for its applications. Some versions of the virtual mirror feature an augmented reality that mixes the real-world environment with computer-generated objects. This combination is already successfully deployed in the virtual fitting rooms application that helps customers decide whether to purchase specific clothing or accessories and reduce the time spent in the shop [1]. Virtual mirrors can also completely remove the real component of the physical world and use an entirely virtual graphical avatar representing the user observing the mirror. The movement of the virtual avatar is usually decided with the help of motion capture hardware, such as Microsoft Kinect. Such scenarios are used in virtual trainers for motor learning applications that teach users to perform specific movements, such as dancing, fitness, or rehabilitation [2].

1.2 Requirements

The following subsections discuss the requirements for a device that should operate in a potential museum exhibit simulating a virtual mirror.

1.2.1 Performance

The virtual mirror application must instantaneously pick up the live camera input from its surroundings and present it modified as a graphical output in a reasonable time frame. Therefore, the graphics accelerator and the capturing device must be powerful enough to handle this processing with acceptable latency and frame rate. Opinions in the area of perceived delay vary and depend on the individual user. It is noteworthy to mention that the type of user involvement also plays a significant role. According to a study, motor-visual activities involving the user's input are more likely undetectable if their delay is limited to around 100ms [3]. Concerning that the virtual mirror application does not require the user's active input as a response, it is reasonable to assume that the delay will elude the user's attention even longer.

1.2.2 Cooling

Exhibit computers operate in environments that are vastly different from those of personal computers. Many operate in environments that might demand a more comprehensive temperature tolerance range. There might be situations where the computer system must be built into the exhibit, and the airflow could be critically reduced. The exhibit itself may be placed indoors or outdoors, which means the computer inside would be required to function during the cold months of winter or hot months of summer. The upper-temperature limit introduces challenges in most cases, as most computer components are not designed for extreme heat. Using active cooling might be an attractive solution. However, this introduces another point of failure. In the event of a cooling failure or dust build-up on the heat sink, the computer may shut down. Additionally, the increased noise due to active cooling may be unacceptable in some situations. [4] Therefore, it is generally preferred to use passive cooling.

1.2.3 Durability

The device must be able to operate daily during the opening hours of the exhibition. In order to prevent interruption of exhibit service, the device must be able to quickly recover on its own in the event of software failure, either by restarting the whole system or the application controlling the exhibit. The system boot-up speed should be in the range of seconds.

1.2.4 Recovery

The digital exhibit can be switched off simply by unplugging from the power supply without sustaining any data loss or hardware damage. Because the museum staff powers down the exhibits at the end of a business day, not via sending instructions in the console and then disconnecting the power but simply by flipping the power switch. Hence the system must be prepared to lose power at any time and be able to recover after the power is resupplied. This feature also becomes convenient in case of an unintended power outage.

1.2.5 Dimensions

Computer systems are made in all shapes and sizes. Recent mini computers can be as small as an external 3.5" hard drive or even a thumb drive. The small size gives them apparent advantages, such as being more portable, producing less heat, consuming less power, and being easier to be incorporated into complex museum exhibits.

1.3 Computer device

There are many compact-sized computers designed for headless or kiosk operation. However, not all of them are suitable for 3D graphics rendering. Industrial computers designed for graphics applications usually contain a dedicated graphics card that introduces high costs and power consumption. Some of the recent low-cost hobbyist mini-computers and compute modules are equipped with components capable of impressive graphics performance while being power efficient. The minimalist design of these power-efficient devices is possible due to the increasingly evolving system-on-chip concept.

The development of such devices is excellent news for electronic enthusiasts. The relevant low cost makes the devices easily affordable and accessible, and the powerful computing capacity allows great development and deployment possibilities. These simple single-board mini-computers have, no wonder, captured the interest and attention of the enthusiasts. In return, the devices gain further potential by receiving constant support and improvement from the community. Against this background, mini-computers have become ever more reliable and popular; they are now an essential part of many companies' IoT deployments.

1.3.1 System on Chip

System on Chip (SoC) is an integrated circuit that encloses most or all computer system components into a single chip. The circuit usually contains a central processing unit (CPU), graphics processing unit (GPU), memory interfaces, peripheral controllers, radio modems, and more. In contrast to the

standard personal computer (PC) that has its component installed separately on a motherboard, the SoC integrates all components into a single small integrated circuit. Although the integrated components cannot be replaced or upgraded, experience shows that a more tightly integrated system design improves performance and reduce power consumption and mean time between failure [5]. Due to these advantages, SoCs are rising in popularity where only microcontrollers were used previously.

1.3.2 System on Module

System on Module (SoM) or Computer on Module (CoM) is a printed circuit board (PCB) that integrates essential computer system components. The module usually includes SoC and additional components such as memory modules. Unlike single-board computers, the SoM needs a carrier board that supplies the module with power and attaches various peripherals. Figure 1.1 contains a picture of the NVIDIA Jetson Nano module (on the left) with its development carrier board (on the right).

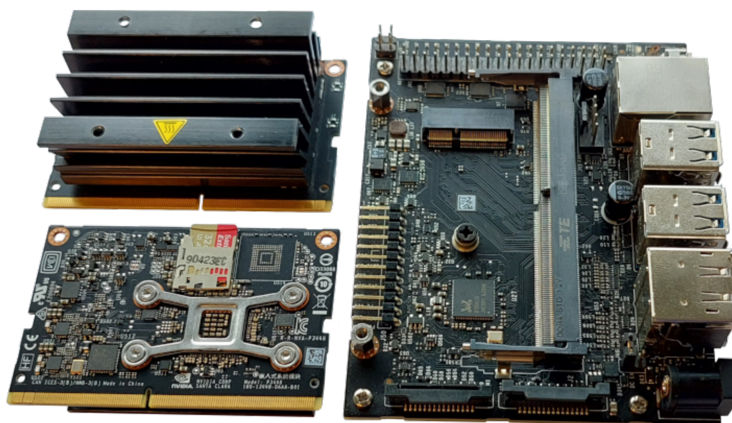


Figure 1.1: SoM with its carrier board

1.3.3 Candidates

Based on the requirements of section 1.2, we have selected the following candidates among the many single-board computers with graphics hardware acceleration. The prices shown in the following comparisons are as of May 2022.

1.3.3.1 Raspberry Pi 4 B

The leading and most famous mini-computers are certainly the Raspberry Pi series single-board computers developed by Raspberry Pi Foundation. Initially invented to promote teaching introductory computer science in schools and developing countries, the phenomenon quickly became widespread outside its targeting market. Due to its low cost, modularity, and relatively good performance, the Raspberry Pi became famous for electronic hobbyists' projects and computer science experiments. The most recent version 4 is equipped with Broadcom BCM2711 SoC integrating quad-core ARM Cortex-A72 64-bit CPU accompanied by a low-power mobile multimedia processor VideoCore VI. The LPDDR4 main memory size comes in various options, so one can choose the best suitable memory size for the individual project while being cost-efficient at the same time. [6]

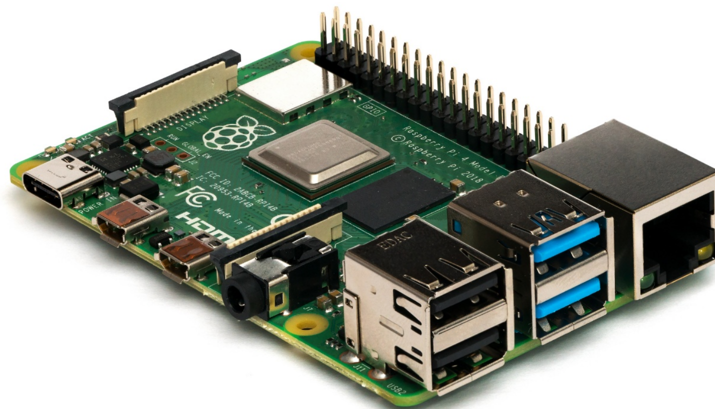


Photo by Michael H. („Laserlicht") [7], licensed CC BY-SA 4.0 [8]

Figure 1.2: Raspberry Pi 4 B

1.3.3.2 NVIDIA Jetson Nano

Jetson Nano from the NVIDIA Jetson product family [9] is an energy-efficient SoM containing NVIDIA Tegra SoC integrates NVIDIA Maxwell GPU with 128 CUDA cores, quad-core ARM Cortex-A57 64-bit CPU. The module contains 4 GB of LPDDR4 shared memory, suitable for most of the applications of its scale. [10]. There is also a 2 GB memory version of this SoM with the exact specifications that cost half the price of the 4GB version. Both modules are available as a standalone SoM or development kit that includes a reference board with various IO, including HDMI and Ethernet. Figure 1.4 shows an image of such a development kit. More detailed specifications are listed in Figure 1.5 .Appendix I additionally shows the dimension of the connected device enclosed in a protective case.

GPU	Broadcom VideoCore VI, 64 GFLOPS (FP16)
CPU	Quad-core ARMv8-A Cortex-A72 64-bit @ 1.5 GHz
Memory	1/2/4/8 GB LPDDR4
Storage	microSD
Networking	RJ45 10/100/1000 BASE-T Ethernet, 2.4 GHz and 5 GHz 802.11b/g/n/ac, Bluetooth 5.0
Video encode	H.264 1080p @ 30
Video decode	H.265 4K @ 60, H.264 1080p @ 60
Display	2x HDMI 2.0, MIPI DSI
Camera	MIPI CSI-2
Other peripherals	2x USB 3.0, 2x USB 2.0, 4x I2C, 4x SPI, GPIO, UART
Dimensions	85.6 mm x 56.5 mm x 17.0 mm
Power	USB-C 5V⎓3A / PoE
Operating temp.	0-50ºC
Price	80€ (2 GB) / 100€ (4 GB)

Figure 1.3: Raspberry Pi 4 B specifications

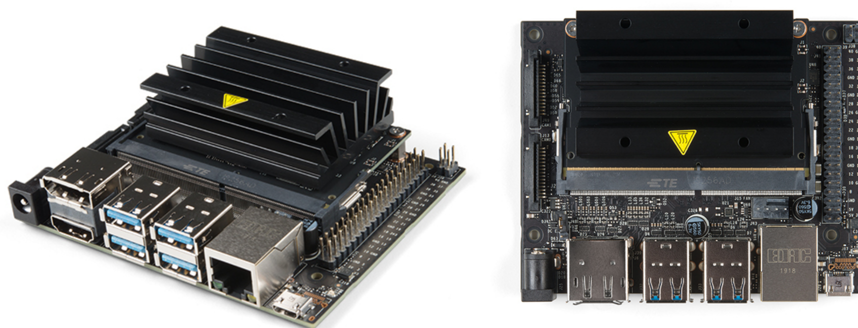


Photo by Spark Fun Electronics [11][12], licensed CC BY 2.0 [13]

Figure 1.4: NVIDIA Jetson Nano Developer Kit

1.3.4 Conclusion

The dimensions of both devices certainly meet the requirements of compactness. At the size of a credit card, they are great candidates for integration into practically any museum exhibit. Although Raspberry Pi 4 B encloses a newer ARM processor of type Cortex-A72 that theoretically offers 90% greater performance than Cortex-A57 [14], NVIDIA Jetson Nano outweighs some other

GPU	128 NVIDIA Maxwell cores, 472 GFLOPS (FP16)
CPU	Quad-core ARMv8-A Cortex-A57 64-bit @ 1.43 GHz
Memory	2/4 GB LPDDR4
Storage	eMMC 5.1, microSD
Networking	RJ45 10/100/1000 BASE-T Ethernet
Video encode (H.264/H.265)	4K @ 30, 4x 1080p @ 30, 8x 720p @ 30
Video decode (H.264/H.265)	4K @ 60, 2x 4K @ 30, 8x 1080p @ 30, 16x 720p @ 30
Display	DisplayPort 1.2, HDMI 2.0
Camera	2x MIPI CSI-2
Other peripherals	PCIe 2.0 x4, USB 3.0, 3x USB 2.0, 3x I2C, 2x SPI, I2S, GPIO, UART, PWM fan header
Dimensions	100.0 mm x 79.0 mm x 28.3 mm
Power	Micro-USB 5V⎓2.5A / DC barrel jack 5V⎓4A
Operating temp.	0-60ºC
Price	74€ (2 GB) / 132€ (4 GB)

Figure 1.5: Jetson Nano development kit specifications

properties that are more important in this context. It is worth mentioning that some speed tests do not come out significantly better compared to the older processor [15]. The Raspberry Pi might be a better value for money for regular day-to-day activities, such as browsing the internet and playing multimedia content. The network connection possibilities are also much more extensive out of the box. However, the graphical capabilities of the Jetson Nano are much better suitable for 3D graphics applications. The specification sheet shows that the 128 NVIDIA Maxwell cores are theoretically more than seven times more powerful than Broadcom VideoCore VI. The Maxwell architecture and NVIDIA CUDA technology expand the opportunities to areas of artificial intelligence, especially computer vision.

1.4 Camera

Jetson Nano has multiple interfaces for connecting a camera, such as USB, Ethernet, and MIPI CSI-2. The industry standard MIPI CSI-2 is the world's most widely implemented embedded camera and imaging high-speed interface [16] that provides transmission between cameras and host devices. The development carrier board has two J13 connectors for connecting up to two cameras. With a more comprehensive carrier board from third-party ven-

dors, the SoM can link up to four connected cameras at once. Jetson Nano supports a diverse ecosystem of cameras, from low-cost hobbyist cameras to industrial high-performance cameras designed for AI. Most of the low-cost cameras available for Raspberry Pi are also supported on Jetson Nano. Jetson partner supported cameras page [17] lists all officially supported cameras.

1.4.1 Sony IMX219

The camera for the virtual mirror does not need high FPS or depth sensing. Therefore we chose a low-cost and low power consumption camera with a Sony IMX219 CMOS sensor capable of resolution up to 3264 x 2464 (8.04M pixels) at 21 frames per second (FPS). There are a few other modes that allow framerate up to 60 FPS at lower resolution. Figure 1.6 shows sensor capture modes revealed by the `v4l2-ctl` tool.

Resolution	# pixels	FPS
3264 x 2464	8.04M	21
3264 x 1848	6.03M	28
1920 x 1080	2.07M	30
1280 x 720	0.92M	60

Figure 1.6: Sony IMX219 sensor capture modes

Chapter 2

Design

The following chapter describes the theoretical design and technique of the virtual mirror application. Furthermore, it describes some of the available technologies exclusive to the Jetson Nano and their potential to render the camera frame more efficient.

2.1 Digital exhibit

We set our course to create a water surface in an attempt to construct a virtual mirror. The water surface is all around us and acts as a native mirror. Whether muddy or not, there is always some level of reflection that reflects the surroundings of the water. Water surfaces can have different shapes depending on if water is still or in motion. In the natural environment, season, time, weather, the constitution of the water, and many other factors all contribute to the unique conditions of the water as well as its surface, thus making its simulation both challenging and exciting.

This work aims not to create a realistic-looking water surface with all the water's physical properties. Instead, it should demonstrate a real-time rendering of a moving texture with a camera frame image projected onto it. A water-like illusion can be further achieved by adding an efficient shader technique that makes the moving texture resembles the water surface's intrinsic properties.

Apart from a piece of shielded glass which is most commonly seen in everyday life, the concept of a mirror has more comprehensive applications. For example, a wet or water surface creating specular reflection is a mirror, as seen in Figure 2.1.

In order to simulate reflection similar to the water surface, the camera must be pointed in the direction of reflection from the potential visitor's view. Aligning the camera can be difficult on a stage accessible to visitors from different angles. The exhibit cannot track the observers looking angle and then adjust its capturing angle. If more than one visitor stood next to the exhibit,

Photo on left by Max.kit [18], licensed under CC BY-SA 4.0 [8]

Photo on right by Daniel Lowth [19], licensed under Pixabay License [20]

Figure 2.1: Natural mirrors

everyone's view angle would be different, creating a situation impossible to simulate with one camera and one display device. For this reason, we decided to significantly limit the visitor's angle of view to a near-vertical view by simulating the water level inside the well. By positioning the water surface deep enough, the angle between the water surface and the observer's view angle is small enough to become neglectable. When the observer looks down the 2 meters deep well to the water surface while being distanced from its center by 35 centimeters, the angle between the water surface and the observer's view angle is less than 10 degrees. Figure 2.2 shows a sketch of the potential digital exhibit, in which I represents the view vector, N the water surface normal vector, and R the reflection vector.

2.2 Software

The following subsections walk through the software stack required to run graphical applications and introduce technologies used for implementing the exhibit application.

2.2.1 Operating system

An operating system (OS) for traditional embedded computer systems is typically a specialized software system designed for a specific purpose to increase reliability for achieving a specific task. [21] Depending on the task requirements, these systems must meet specific time constraints, and therefore this type of OS is frequently considered a real-time OS (RTOS).

Modern embedded systems and single-board computers are equipped with multi-core microprocessors can run the usual time-sharing OSes compatible with the processor architecture, making it possible to adapt existing modular

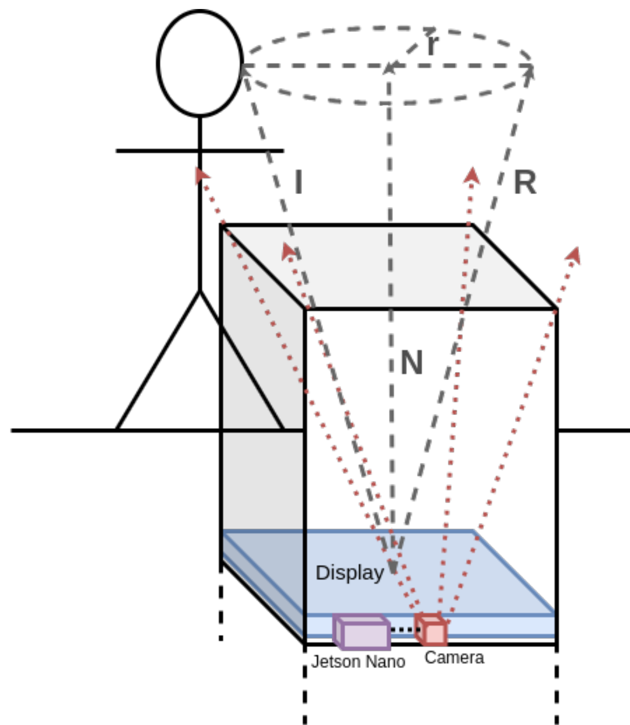


Figure 2.2: Exhibit design

OSes to perform a more comprehensive set of tasks. These include but are not limited to smart TVs and phones, tablets, entertainment systems, navigation systems, and networking equipment.

Many vendors base their systems on the popular open-source Linux kernel, mainly because of its versatility, portability, and vast availability of device drivers. Another excellent example of such an OS would be Android, a mobile OS based on a modified version of the Linux kernel, whose development is sponsored by Google. [22]

2.2.1.1 Jetson Linux

Jetson Linux or NVIDIA Jetson Linux Driver Package (L4T) is a software package for Jetson series devices. The package contains Linux Kernel 4.9, bootloader, NVIDIA device drivers, and utilities for customizing and flashing the filesystem image. Next to the L4T software package, NVIDIA provides an SD card image, drivers source code, documentation, and other related tools that might help during the development process on Jetson Nano. [23] It is important to check for the latest Jetson Linux release on the archive [24] page. The default download page does not offer downloads for the Nano boards anymore.

2.2.1.2 Vendor-provided SD card image

The NVIDIA supplied SD card image's rootfs is based on the popular Ubuntu Linux distribution. It includes a full-sized Ubuntu Desktop environment, application demos, device drivers, and documentation. We discuss the minimization of the OS better suited for production use in the section section 3.1.

2.2.2 Windowing System

A windowing system is essential for managing different parts of the graphical display, enabling applications to create windows and draw user interface components such as buttons, menus, and icons.

2.2.2.1 X Window System

The X Window System or X11 is an architecture-independent client/server windowing system for displays used on Unix-like operating systems. [25] X11 provides an abstract interface for applications to draw windows and handle input devices interactions. The clients are applications, while the server displays the windows and handles input devices such as keyboards, mice, and touchscreens

2.2.2.2 Embedded-System Graphics Library

Embedded-System Graphics Library (EGL) is an interface between graphics APIs and the underlying platform window system such as X11. It specifies mechanisms for managing graphics API context, creating the surface the graphics API renders to, and synchronizing graphics API operations. EGL spares a programmer from touching the platform-specific API, such as GLX on X11-based platforms.

2.2.3 Graphics API

Graphics API allows the programmer to access the graphics hardware capabilities without writing hardware-specific code. These capabilities can efficiently calculate and draw graphics in a frame buffer intended for output to a display device or perform general-purpose parallel computing for the scientific domain.

GPU vendors implement the API specifications in the OS drivers. Therefore, we have to look for a graphics API supported by GPU vendors and their compatible OS. Linux for Tegra features several well-known modern graphics APIs. Figure 2.3 lists graphics APIs available in Linux for Tegra.

Specification	Version	Release year
Vulkan	1.0.2	2016
OpenGL	4.5	2014
OpenGL ES	3.2	2015

Figure 2.3: Graphics APIs supported by Linux for Tegra

2.2.3.1 OpenGL ES

OpenGL ES or GLES (Open Graphics Library for Embedded Systems) is a royalty-free, cross-platform graphics hardware API specification for rendering 2D and 3D graphics on embedded and mobile systems - including gaming consoles, phones, and multimedia devices. The API is a subset of OpenGL, a more complex API in terms of the number of functions, making it easier for vendors to implement it on devices with simpler and low-powered hardware.

2.2.4 Libraries

Implementing well-tested libraries is generally considered good practice in software development for it saves time and minimizes the risk of error. Mature libraries offer various reliable functions that can be easily incorporated into projects without writing all logic from scratch. Such practice saves much valuable time for development and ensures less chance of making errors. The following subsections introduce libraries used to implement certain required functionality of a graphics application, such as mathematical functions or interaction with container formats described in subsection 2.2.6.

2.2.4.1 OpenGL Mathematics

OpenGL Mathematics (GLM) is a header-only, platform-independent C++ mathematics library for graphics designed and implemented with the same naming conventions and functionality of OpenGL Shading Language (GLSL). Along with the GLSL related features, the library provides an extension system that offers helper functionality for matrix transformations, quaternions, and more. [26] GLM takes advantage of the C++11 and utilizes SIMD instruction set extensions to increase performance [27] when the same mathematical operations are performed on multiple data objects.

2.2.4.2 OpenGL Image

OpenGL Image (GLI) is a header-only, platform-independent C++ texture image loading library supporting popular formats such as Khronos Texture (KTX) and Microsoft DirectDraw Surface (DDS). GLI is developed by the same author (G-Truc Creation) as the GLM library and shares many similarities when it comes to following the GLSL specification. Additionally to

texture image loading, the library features OpenGL and Vulkan texture creation helpers and automatic mipmaps generation. [28]

2.2.4.3 Open Asset Import Library

Open Asset Import Library (Assimp) is a widespread, open-source library for importing and processing geometric scenes. The library unites more than 50 file formats under a common API, including popular Wavefront OBJ, COLLADA, and FBX. A programmer can specify post-processing steps such as mesh optimization and accommodating geometry calculation. The imported meshes, materials, bone animations, and other associated model data are accessible in a hierarchical data structure.

2.2.4.4 POCO C++ Libraries

POCO C++ Libraries, the abbreviation for portable components, is a highly portable collection of client and server libraries helping programmers solve frequently-encountered practical problems. The library complements the standard C++ library (STL) with many practical extensions, wrappers, and helpers for data structures, multithreading, parsers, application logging, etc.

2.2.5 Platform-specific libraries

In order to access all of the hardware and features on an embedded device, low-level libraries are often developed directly by the hardware vendors. These libraries are available in NVIDIA's apt repositories configured via the Jetson Linux package. We explain how are the repositories installed during the filesystem bootstrapping in section 3.1.

2.2.5.1 NVIDIA Tegra Multimedia API

NVIDIA Tegra Multimedia API or NVIDIA Multimedia Utilities is a collection of lower-level APIs, libraries, and documentation for developing embedded applications for the Jetson platform. The package also contains sample application source code that demonstrates the usage of lower-level APIs for the underlying hardware blocks and high-level patterns and solutions for understanding multimedia development. The following two libraries were used for acquiring frames from the physical camera:

- **NVIDIA Buffer Utilities** is a closed source [29] set of utilities for hardware buffer allocation and management. This library allows us to administer the camera frame buffers shared between CPU and integrated GPU to gain zero-copy benefits. Figure 3.3.4.1 describes the benefits of zero-copy in more depth.

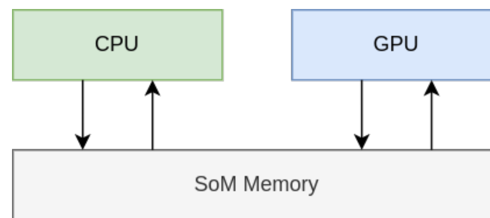


Figure 2.4: Shared memory

- **NVIDIA Argus** library allows us to acquire pictures and associated metadata from Jetson-supported cameras. The library handles the transmission of the camera sensor data and processes it into the output of various encodings suited for further graphics processing. Available capture configuration allows achieving a variety of use cases, such as traditional or computational photography, computer vision, and other fields. [30]

2.2.6 Container formats

Container formats specify how are various types of digital information stored on a computer system. The following subsection describes two industry-standard container formats used to store graphics data loaded by our implementation.

2.2.6.1 DirectDraw Surface

The DirectDraw Surface is (DDS) a container file format developed by Microsoft for storing uncompressed and compressed texture image data. [31] Compressed variations use a previously proprietary S3 Texture Compression (S3TC) that enables the image data to be transferred and stored in compressed format, thus improving rendering speed, lowering the loading times, and allowing the application to utilize higher resolution textures with the same memory footprint. [32]

2.2.6.2 Filmbox

Filmbox (FBX) is an industry-standard geometry exchange format owned by Autodesk that is compatible with almost all 3D graphics and animation software. The format describes detailed geometry data, lights, cameras, materials, and animations. While Autodesk's implementation is proprietary, its format description is exposed in the FBX Extensions SDK, which provides header files for FBX readers and writers. For this reason, many open-source projects implement the FBX specifications, which further enhances its popularity.

CHAPTER 3

Proof of concept

3.1 OS optimization

A fully equipped distribution like the one provided in the SD card image available from Jetson Download Center [33] might be helpful for home experimentation and demonstrations of enclosed application demos. However, the standard desktop environment consumes significantly more resources than the minimal environment running only the necessary processes. At the moment, there is no official NVIDIA-produced SD card image without the desktop environment [34]. In the case of minimalist systems like the Jetson Nano, it makes sense to reduce the operating system footprint so our application can fully utilize the underlying hardware capacity.

The main memory is physically shared between the central and graphics processing units. Therefore, decreasing the base memory footprint allows the GPU to utilize more memory, giving the application programmer more opportunities for performance optimizations, and space for graphical content.

3.1.1 The approach

Stripping and bootstrapping are two of the approaches for minimizing the initial resource requirements through OS optimization. Both options have their advantages and disadvantages and require different tools.

3.1.1.1 Stripping

Stripping the existing root filesystem (rootfs) would require fewer steps than creating the filesystem from scratch. We could flash the official image onto a physical SD card, boot the Jetson Nano, and remove the unwanted packages and files on a running system using `apt`, the package manager and filesystem utilities, such as `rm`. It will allow us to test immediately if the recently removed packages and files are causing the system or application instability. However, we might miss removing the packages or especially the files that

are not required. The sample SD card image contains some files that are not part of any packages, so removing the packages would still keep those files on the filesystem. Removing a large number of existing files might also possibly introduce fragmentation [35]. Figure 3.1 list commands used for removing the desktop environment - one of the most significant packages that a single graphical application environment does not need.

```
# Stop and disable gnome desktop manager
systemctl disable --now gdm3

# Change systemd boot target
systemctl set-default multi-user.target

# Remove ubuntu desktop and desktop manager packages
apt remove ubuntu-desktop gdm3

# Remove no longer required dependencies
apt autoremove
```

Figure 3.1: Removing the desktop environment

3.1.1.2 Bootstrapping

Bootstrapping the filesystem allows a more precise selection of packages, files, and services that are included in the system. Bootstrapping is done by installing the base OS files into a subdirectory of an existing Linux system that does not necessarily have to be Jetson Nano. Using a utility that allows us to change the current root directory, we can execute the binaries for installing desired packages and configuration files. With the technologies that we describe in subsection 3.1.4, the host does not even need to match the processor architecture of the Jetson Nano, therefore allowing us to produce the filesystem on an entirely different system, such as the Linux desktop workstation with the x86-64 processor architecture.

After preparing the rootfs, an image of the SD card and its GPT table is created, which must conform to the structure defined in the NVIDIA documentation. Finally, the packaged rootfs partition, bootloader, and other system partitions are written to the SD card image. This process, along with the more detailed steps of the filesystem preparation, is described in subsection 3.1.2.

3.1.2 Minimal system installation

Bootstrapping allows us to customize the system to a greater extent and, at the same time, helps us to deeper understand the graphics software components and their dependencies. Therefore we decided to choose the bootstrap process.

In the following subsections, we explain what tools we used to bootstrap the structure of the rootfs, how we installed the required packages and files, and finally, how we, with the help of NVIDIA tools, constructed an SD card image structure.

`debootstrap` is a tool for installing the Linux directory structure and essential system packages into a subdirectory of another already installed system. The tool downloads the packages from the `apt`-compatible repository and installs them the same way as the apt tool would in a non-simulated environment.

`qemu-debootstrap` is a wrapper of `debootstrap` that allows us to bootstrap the system of different processor architectures. It installs and uses `qemu`, an emulation software that handles the execution of the foreign architecture binaries during the bootstrapping process.

Command in Figure 3.2 consisting of the following arguments bootstraps Ubuntu 18.04.6 LTS (Bionic Beaver) of an ARM64 architecture into `/mnt` directory. Unless otherwise stated throughout the sections below, `/mnt` refers to the directory that stores the prepared rootfs.

- `--arch` Desired processor architecture
- `--variant` The name of the bootstrap script to use that decides what packages to install. In this case, the `minbase` variant is used that installs only the `apt` package manager and required packages for the base system.
- `--include` Additional packages to install during the bootstrap process. The `systemd-sysv` package installs the system and service manager for Linux, that is not included in the `minbase` script.
- `--cache-dir` Path to optional package cache to prevent repeated downloading of already fetched packages.

```
qemu-debootstrap --arch=arm64 \
                 --variant=minbase \
                 --include=systemd-sysv \
                 --cache-dir="$(pwd)/packages-cache" \
                 bionic \
                 /mnt
```

Figure 3.2: Bootstrapping the foreign architecture rootfs

3.1.3 NVIDIA packages

Jetson Nano requires several packages present on the root partition to boot and utilize its capabilities. NVIDIA provides these packages via their public

```
root@hostname:/mnt# ls -a /mnt
.    ..    bin   boot  dev   etc  home  lib  media  mnt
opt  proc  root  run   sbin  srv  sys   tmp  usr    var
```

Figure 3.3: Directory structure after bootstrapping the rootfs

package repository that needs to be configured on the system before issuing the `apt install` command. Figure 3.4 demonstrates the steps of registering the NVIDIA-issued public key for package integrity validation and installation of two repositories.

```
# Add repository key
curl -L -o /mnt/etc/apt/trusted.gpg.d/nvidia_jetson.asc \
'https://repo.download.nvidia.com/jetson/jetson-ota-public.asc' \

# Add NVIDIA common repository
echo \
'deb https://repo.download.nvidia.com/jetson/common r32.7 main' \
> /mnt/etc/apt/sources.list.d/nvidia_jetson.list

# Add NVIDIA t210 (Jetson Nano) repository
echo \
'deb https://repo.download.nvidia.com/jetson/t210 r32.7 main' \
>> /mnt/etc/apt/sources.list.d/nvidia_jetson.list
```

Figure 3.4: NVIDIA repositiory configuration

The commands mentioned so far do not require any emulation of binaries located inside the prepared rootfs. We can alter the configuration files outside the root directory with host tools not necessarily present on the guest system. However, some tools, such as the package manager or other administrative tools, have to reside inside the managed rootfs. Moreover, some tools might not exist for the host processor architecture, especially third-party tools from NVIDIA or other providers that only target Jetson's processor architecture, ARM 64. Therefore emulation of these binaries inside the rootfs is necessary.

After installing the repositories and updating the packages cache with emulated `apt update` command, we can start installing the essential packages, such as bootloader, kernel, firmware, and graphics API libraries. Figure 3.5 lists the packages that we require inside our custom SD card image. The installed size row represents the package sizes without their required dependencies. The extended list with the dependencies of other NVIDIA packages can be viewed on a graph in Appendix C.

[1]See subsection 3.1.4

Package(s)	Description	Installed size
nvidia-l4t-core	Core shared libraries	7.7 MB
nvidia-l4t-bootloader	Bootloader	11.3 MB
nvidia-l4t-initrd	Initial ramdisk	7.1 MB
nvidia-l4t-kernel nvidia-l4t-kernel-dtbs	Kernel	120.0 MB
nvidia-l4t-firmware nvidia-l4t-xusb-firmware	Firmware	2.3 MB
nvidia-l4t-jetson-multimedia-api	Multimedia API	95.7 MB
	Total:	244.1 MB

Figure 3.5: NVIDIA packages

3.1.4 Emulating the rootfs

To make the rootfs work with the actual device, we need to execute additional modification steps before transforming the rootfs into an SD card image. These steps include the installation of packages critical for device booting. Therefore, we need to use a method for executing these steps in an emulated environment on a computer system that builds the SD card image.

`chroot` is a utility that changes the apparent root directory for a running process and its children. In the combination of chroot and emulation software, we can execute foreign architecture binaries in our virtual rootfs, as shown in Figure 3.6.

```
# Change root password
chroot /mnt qemu-aarch64-static /usr/bin/passwd root

# Install packages
chroot /mnt qemu-aarch64-static /usr/bin/apt install \
                                --no-install-recommends -y \
                                openssh-server rsync
```

Figure 3.6: Emulation of binaries in a foreign rootfs

3.1.5 Cleaning up

When performing changes with the emulated binaries, some tools, mainly package manager, accumulates cache and log files inside the virtual rootfs. This cache grows into significant sizes, unnecessarily increasing the overall size of the future SD card image. It is wise to remove the cache and other unwanted files before packaging the rootfs. Commands in Figure 3.7 erase the package manager cache and empty log files.

```
# Clean apt cache
chroot /mnt qemu-aarch64-static /usr/bin/apt clean

# Remove logs
rm /mnt/var/log/apt/*.log.xz
find /mnt/var/log -type f -exec truncate -s 0 {} \;
```

Figure 3.7: Cleaning up APT and logs

3.1.6 Packaging the rootfs

Before packaging the rootfs into a raw image, the final step is creating a boot menu entry by copying the reference config from the NVIDIA L4T driver package. After that, based on the board and storage type, the NVIDIA-made script called `flash.sh` additionally replaces bootloader configuration placeholders and converts the rootfs directory into a raw partition image that can be written into the first SD card partition. Figure 3.8 demonstrates the usage of the script supplied with the parameters suitable for Jetson Nano boards that use the SD card storage medium. The script requires the following few environment variables arguments.

Environment variables:

- `BOARDID` Board identification
- `FAB` Board revision
- `BUILD_SD_IMAGE` SD card image mode

Arguments:

- `--no-root-check` Do not validate root user (compatibility issues)
- `--no-flash` Skip flashing to to physical device
- `--sign` Sign the partition images
- `-S` Root filesystem size with reserve

3.1.7 Partition table

Embedded systems do not have a BIOS or UEFI [36]. Instead, the firmware stretches across the SD card partitions [37][38]. These partitions are vital in the device booting process. The existence, order, sizes, and contents of these partitions must comply with the specification described in the NVIDIA documentation [38]. The partitions, other than rootfs, are binary data instead of the standard filesystems that can be mounted or used as regular partitions.

```
# Estimate root partition size (real size + 20%)
ROOTFS_DIR_SIZE="$(du -bms /mnt | awk '{print $1}')"
ROOTFS_DIR_SIZE="$(("${ROOTFS_DIR_SIZE}" * 12 / 10))"

BOARDID=3448 \
FAB=200 \
BUILD_SD_IMAGE=1 \
bash /opt/l4t/flash.sh \
     --no-root-check
     --no-flash \
     --sign \
     -S "${ROOTFS_DIR_SIZE}MiB" \
     jetson-nano-qspi-sd \
     mmcblk0p1
```

Figure 3.8: Usage of `flash.sh`

Number	Name	Description	Size in bytes	Required
1	APP	Root filesystem	-	✓
2	TBC	TegraBoot CPU-side binary	131 072	✓
3	RP1	Bootloader DTB binary	458 752	✓
4	EBT	CPU bootloader binary	589 824	✓
5	WB0	Warm boot binary	65 536	✓
6	BPF	SC7 entry firmware	196 608	✓
7	BPF-DTB	BPMP DTB binary	393 216	✗
8	FX	Fuse bypass	65 536	✗
9	TOS	TOS binary	458 752	✓
10	DTB	Kernel DTB binary	458 752	✓
11	LNX	U-Boot	786 432	✓
12	EKS	Encrypted keys	65 536	✗
13	BMP	Boot splash screen image	81 920	✗
14	RP4	USB module's firmware	131 072	✓

Figure 3.9: Jetson Nano SD card partitions

See Figure 3.9 for the complete list of supported partitions by the Jetson Nano device.

We start by allocating the SD card image file. The size of this file is estimated based on the sum of the GPT header and partitions size. The enclosed source code reveals a technique that calculates the exact size of the image file. Please refer to the file `system-image/scripts/201-allocate-image.sh`. Figure 3.10 shows the steps for allocating a 2 GB image file and writing the empty GPT table to the beginning of the image that stores the information

about the partitions. The `dd` command fills the allocated space with `NULL` bytes to further improve the compression potential of the free space provided by the 20% space reserve applied in the Figure 3.8.

```
# Allocate SD card image file
dd if=/dev/zero \
   of=jetson-sd-card.img \
   bs=1048576
   count=2000

# Write GPT table
sgdisk -og jetson-sd-card.img
```

Figure 3.10: Allocating the SD card image

Figure 3.11 shows the creation of a single partition with a given number, number of sectors, and name. The number of the sectors is calculated by dividing the partition size by the sector size, in this case, 512.

```
sgdisk -n "${NUMBER}:0:+$((${SIZE} / 512))" \ # Number of sectors
       -c "${NUMBER}:${NAME}" \               # Partition name
       -t "${NUMBER}:8300" \                  # Partition type
       jetson-sd-card.img
```

Figure 3.11: Adding partition to GPT table

3.1.8 Populating the partitions

In this work, we aim to customize only the first partition containing the root filesystem. Other partitions from the Figure 3.9 are to be populated with the partition images supplied in the NVIDIA L4T driver package. After creating the GPT table and adding the partitions, we are ready to mount the image virtually, using the tool for loop device management. The host kernel then detects the image, and all its partitions are recognized and accessible via the `/dev` directory. The pattern for each partition is `/dev/loopXpY`, where the `X` is the first free loop device number and the `Y` is the partition number from the Figure 3.9.

Figure 3.12 shows the process of mounting the SD card image virtually using the `losetup` utility and then writing the partition data into the corresponding partition on the SD card image.

```
# Mount the SD card image
LOOP_DEVICE="$(losetup --show -f -P jetson-sd-card.img)"

# Write partition contents
dd if="${PARTITION_IMAGE_PATH}" of="${LOOP_DEVICE}p${NUMBER}"

# Unmount the SD card image
losetup -d "${LOOP_DEVICE}"
```

Figure 3.12: Populating the SD card partition

3.1.9 Automation

The developer often needs to install additional libraries and files to the device because of the updated application requirements. Therefore, it is practical to automate the process of SD card image creation so the developer can apply the changes and generate an updated image efficiently. Furthermore, adding a versioning system to this automation makes it possible to build multiple slightly different images that share the same foundation. The scripting approach also allows the developer to track down the unused packages and files and measure their actual file size after the unpacking.

The enclosed solution to the process automation uses Docker for OS-level virtualization, making the build process portable and secure by isolating the build processes and data operations into a virtual container. The building process consists of smaller scripts executed in an order that apply various independent changes, making it possible to choose which changes to apply and which to skip. Refer to the Appendix D for a brief description of the scripts.

3.1.10 Results

Rootfs customization measures significantly decreased the overall SD card image size, improving the SD card flashing speed by more than ten times[1]. Figure 3.13 compares the size of the custom production, custom development, and vendor-provided SD card image in three different categories.

- **Real** category represents the sum of the sizes of all files contained on the rootfs.
- **Uncompressed image** category represents size of the SD card image in raw disk image format (.img).

[1]Compared to uncompressed vendor SD card image

- **Compressed image** category represents size of the SD card image in raw disk image format (.img) compressed with `zstd --adapt -T0` (adaptive compression level, all CPU cores).

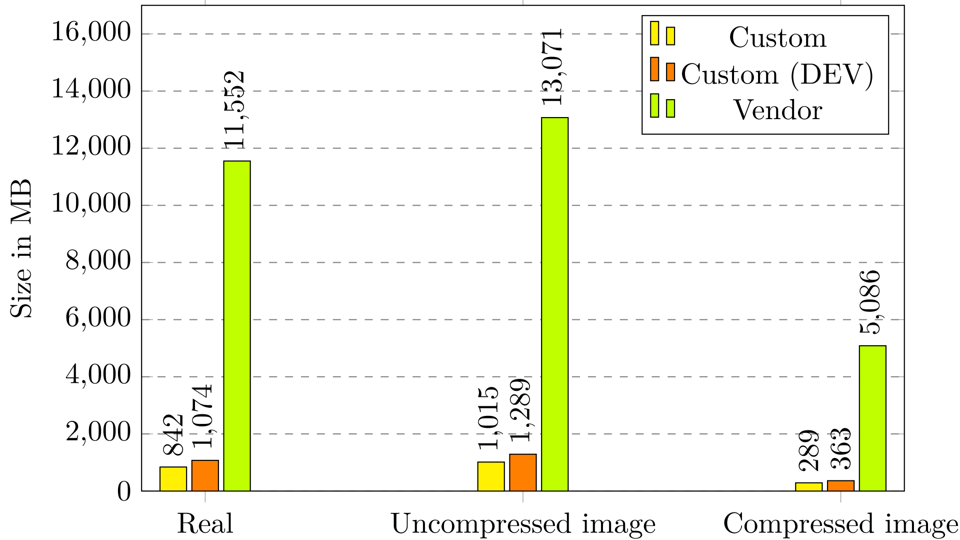


Figure 3.13: Comparison of SD card image size

Furthermore, the memory usage with the exhibition running decreased roughly from 2 GB to 0.5 GB, and the load average (5 min) from 1.42 to 0.85, resulting in more available memory for application and lower device temperatures measured during the operation. Figure 3.14 shows the graphical comparison of both mentioned resources.

3.2 Development environment

A development environment is a collection of tools for developing and debugging an application. The following short section briefly describes the parts of the development process worth mentioning.

3.2.1 Physical exhibit simulation

In addition to the software stack, the environment consists of a dedicated monitor for displaying the exhibit's digital output. The monitor is suspended in the air with a mechanical arm and has a box mounted on its front to simulate the walls surrounding the well's water. The box helps to calibrate the scene rendering. Figure 3.15 shows the development version of the exhibit.

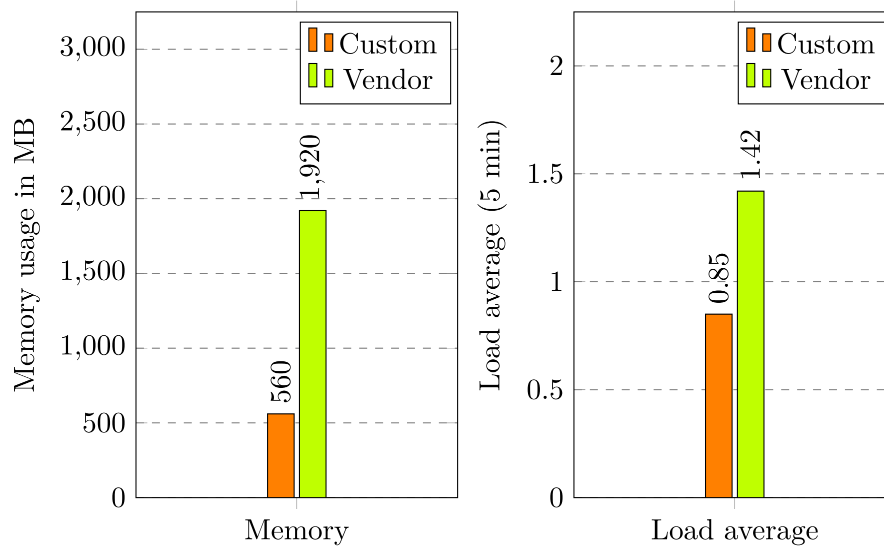


Figure 3.14: Comparison of resources utilization

The picture on the left displays the dedicated screen, Jetso Nano sitting on top, and a modified PC power supply powering the device. The picture on the right reveals the view into the exhibit with the camera mounted on the upper part of the monitor.

Appendix G additionally reveals a more detailed view of the early development version of the exhibit with calibration tapes in place to align the water surface.

Figure 3.15: Exhibit development version

3.2.2 Remote development

The device we are using has unusual hardware not available on a regular desktop computer a developer would use. When developing an application for concrete hardware with a foreign architecture, the development process must be implemented on the targeting device unless a virtualized environment simulating all device components is available. Remote development was made possible by CLion IDE's remote development feature that automatically synchronizes the project files between the development device and the programmer's system. The IDE also provides a remote debugging interface that collects debugging data from the remote GDB server running on the device, making the development experience as if the whole process took place on the programmer's computer.

3.3 Implementation

The following section describes the architecture behind the graphical application and the implementation specifics for the virtual mirror deployed in the museum's exhibit. Words that use the `monospace` font style through this section refer to source code file names, functions, and class names.

3.3.1 Project structure

The application comprises several packages representing various layers and blocks of the application. This separation allows the programmer to navigate the application implementation parts quickly during the development process. The implementation is distributed into classes, each representing an object, regardless of the object's multi-instance potential. The only component that does not follow this pattern is the Utils component.

3.3.1.1 Addons

Addons package consists of classes that wrap the functionality of third-party libraries. The classes simplify the usage of the underlying library APIs by combining multiple API calls into a more high-level interface used by the application. The framework implements the following three addons:

- `ArgusCamera` for acquiring frames from the physical cameras using powerful NVIDIA Argus API;
- `SharedFrameBuffer` for management of the shared hardware buffers using closed-source NVIDIA Buffer Utils;
- `X11Display` for creating window and receiving the input events using X11 protocol.

3.3.1.2 Application

The application package provides classes for bootstrapping different layers of the application. Classes from the upper layers inherit the logic from the lower layers and gradually define a more complex application. Figure 3.16 describes the functionality on different layers up to the entry point class, `WellApplication`.

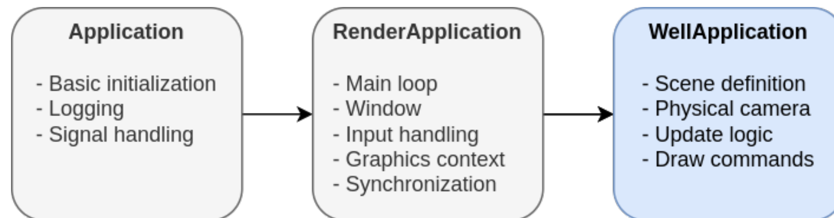


Figure 3.16: Application layers

3.3.1.3 Graphics

The most extensive package among all others, containing over 75% of the code base, is the Graphics package. It contains mainly classes representing OpenGL ES objects whose functionality is wrapped into more high-level and easy-to-use methods. This interface is compatible with other framework components, is more manageable, implements automatic deallocation, and performs real-time validation of the OpenGL ES state machine.

3.3.1.4 Math

The Math package defines Matrix and Vector types generated from the templates provided by the OpenGL Mathematics library. The generated types follow the OpenGL ES specification naming and application's code style.

3.3.1.5 Utils

The Utils package covers functionality shared among multiple classes, including pseudo-random number generator, mathematical functions, string helpers, and timing functionality.

3.3.2 Error handling

Graphical applications are long-living dynamic processes with various components that can sometimes operate incorrectly due to a software bug or hardware failure. It is essential to implement measures to reduce the chance of errors created by the programmer and to terminate the program safely in time if they occur. Failure to release some device resources before application

termination may result in the system not recovering even after restarting the application.

Several concepts were implemented during the development process to detect errors created by the programmer, such as assertions and run-time checks.

3.3.2.1 Assertions

Assertions are programming language macros that test certain program logical assumptions. Violation of these assumptions results in immediate program termination. Our implementation uses two types of assertions:

- **Compile-time** or static assertions are evaluated during compilation and result in a compilation error with an accompanying message if an asserted expression is negative. Static validations are helpful when compile-time restrictions, such as template evaluations or constant variables, need to match specific properties.

- **Run-time** assertions, in contrast to static assertions, test software state while the program is running. Assumptions about how the code will behave while running is often wrong. Checking the program's state during execution can prevent execution over unexpected data that might lead to severe inconsistent or unrecoverable states. Compared to exceptions, run-time assertions terminate the program immediately without an attempt to restore a healthy state or perform the cleanup. When sufficiently implemented, run-time assertions can quickly detect and prevent bugs from reaching the production system. Our implementation contains over a hundred run-time checks, mainly checking the initialization state and passed method arguments for unexpected values.

3.3.2.2 Exceptions

Exceptions are unexpected software events that occur during the program execution. The software may still recover and continue normal execution or log the error, safely release resources, and shut down the application.

3.3.2.3 Logging

Software logging is a crucial component in tracking the application's behavior and can help track mistakes quickly, even without debugging. POCO C++ provides logging macros that prepend the log message with the source component name, file, and line. With this format, a developer can save time by narrowing the area of browsed or debugged code.

Across the implementation, the logging system tracks checkpoints of application initialization, dynamic configuration values, and resource allocations. Figure 3.17 demonstrates the logging of camera initialization using negotiated sensor mode.

```
poco_information_f4(Poco::Logger::get("Camera"),
    "Configured camera %u with sensor mode %ux%u @ %.3hf",
    static_cast<uint32_t>(this->_deviceIndex),
    this->_width, this->_height,
    1.0f / (static_cast<float>(this->_frameTime.count()) / 1e9f));

// Outputs: Configured camera 0 with sensor mode 1640x1232 @ 30.000
```

Figure 3.17: Logging macro usage

3.3.3 Initialization

Application entry point lies in the source file `WellMain.cpp`. It uses a macro from the Poco library that expands to a classical C++ entry-point, the `main` function. The macro accepts the name of a class that must implement the `Poco::Util::Application` interface. The interface defines the basic application structure that forces the programmer to define the initialization, deinitialization, and main logic methods.

3.3.3.1 Application

The application's functionality is separated into different layers. Each layer is responsible for handling its feature. The lowest layer, contained in `Application` class, handles basic application logic, such as initialization of the logging system and exception catching. `RenderApplication` layer adds initialization and management of the window, graphics context, and graphics synchronization. The highest layer initializes the shading system, connects to the physical camera, and invokes the loading of the models and textures.

3.3.3.2 Window

The initialization of the X11 client begins with retrieving the `DISPLAY` environment variable, which instructs the client which X11 server it is to connect to. `Poco::Environment` helper class attempts to retrieve the value of this variable and sets it to the default value (typically `:0`) if it is missing. The logic for creating the display is stored inside `X11Display` addon, a wrapper around `Xlib` API calls to create and manage a simple full-screen window and receive keyboard input.

3.3.3.3 Graphics context

Graphics context represents a state machine that stores all the application's rendering data. Since the display and graphics context are not part of the same specification, connecting them is platform-dependent. EGL handles the

connection of graphics context with the underlying platform windowing system and other features such as rendering synchronization and object binding.

EGL provides an interface for obtaining `EGLDisplay`, an abstract representation of the underlying platform. From obtaining this object, the further interaction with the display from the graphics API is using this object. Another related object is `EGLSurface` that represents the native window and associated buffers, such as color buffer, depth buffer, stencil buffer, and alpha mask buffer [39].

Before obtaining the graphics context, the application needs to create a surface that the graphics API uses to draw the frame. The drawing surface has few attributes that the application can require prior to creating. Figure 3.18 demonstrates how such requirements look like and what are their typical values.

```
const EGLint configAttribs[] = {
  EGL_RED_SIZE,         8,                  // Red channel size
  EGL_GREEN_SIZE,       8,                  // Green channel size
  EGL_BLUE_SIZE,        8,                  // Blue channel size
  EGL_ALPHA_SIZE,       8,                  // Alpha channel size
  EGL_DEPTH_SIZE,       24,                 // Depth buffer size
  EGL_STENCIL_SIZE,     8,                  // Stencil buffer size
  EGL_SURFACE_TYPE,     EGL_WINDOW_BIT,     // Surface type
  EGL_RENDERABLE_TYPE, EGL_OPENGL_ES3_BIT, // Graphics API
  EGL_NONE                                  // End of requirements
};
```

Figure 3.18: Graphics context requirements

With a drawing surface created and display abstraction acquired, drawing context is created and activated using `eglMakeCurrent`.

The graphics API separately provides error handling for context-related errors to improve the ability to locate the GLES errors. It is possible to check for debugging support prior to enabling by inspecting the `GL_CONTEXT_FLAGS`. Calling `glEnable` with the `GL_DEBUG_OUTPUT` activates the debugging and should be accompanied by the call with `GL_DEBUG_OUTPUT_SYNCHRONOUS`. Synchronous mode guarantees the callback to be called from the same thread as the context, assuring that the programmer can find the origin of the error in the call stack.

Similar to the graphics API debugging, the EGL registers a debug callback using `eglDebugMessageControlKHR`.

3.3.3.4 Physical camera

Sony IMX219 camera is operated by the `ArgusCamera` addon, which is a wrapper around the NVIDIA Argus API. The addon adds modularity and auto-

matically handles the compatible buffer allocation for captured frames.

The initialization starts by selecting the camera device by its physical connector index. Prior to selecting the sensor mode, the graphics context must exist so that the camera can allocate the internal DMA buffers used to store the captured frames. The external camera frame buffer managed by the rendering thread is allocated similarly to internal DMA buffers via `CreateCompatibleFrameBuffer` method of `ArgusCamera` wrapper.

As the last step, the application spawns the camera frame acquire thread. subsubsection 3.3.4.1 describe the process of retrieving camera frames in more detail.

3.3.3.5 Scene

In this particular application, the scene is relatively simple. It contains the 3D model of the walls, a quad displaying the water surface, and a quad for projecting the camera frame. Loading of all the objects is handled by their `Create` method and unloading by their `Destroy` method. The loading of the objects is thus simple, as shown in Figure 3.19, and it is invoked inside the `WellApplication::Create` method.

```
Model wellModel;
wellModel.Create("well");

CameraFrameQuad cameraFrameQuad;
cameraFrameQuad.Create(1640.0f / 1232.0f);
cameraFrameQuad.SetPosition({-0.1f, -0.025f, -90.0f});
cameraFrameQuad.SetScale(2.225f);
```

Figure 3.19: Initializing scene objects

3.3.4 Main loop

After initialization, the application enters the so-called main loop, where it performs a series of actions repeatedly until the exit request is received. The actions usually process the inputs, update the program state, and generate the outputs. One iteration of the loop is called a frame. [40] Most real-time computer graphics applications perform 30, 60, or more FPS and process the inputs at least from the keyboard or mouse.

In the case of unattended computers or embedded devices, the user input is processed only in particular situations, like initial setup or calibration. Refer to the subsection 3.4.3 for the example from this project. Instead, the non-user applications usually process the input from other places, such as GPS, sensors, and cameras. This input can be further processed and represented graphically or, in the case of cameras, passed directly to the shader unit.

The application state is updated according to the inputs or additional predefined update logic, such as updating the position of a moving texture or animated mesh. The updated state is then transformed into graphical output, and the frame is presented on display.

The implementation of the main loop is defined in the following methods:

- `X11Display::ProcessEvents` registers keyboard events that occured since last invocation;
- `WellApplication::Update` updates the scene state according to the pressed keyboard keys and predefined update logic;
- `WellApplication::Draw` renders the meshes using their renderers and shaders;
- `RenderApplication::Sync` waits for the graphics card to finish the execution of drawing operations;
- `RenderApplication::Finish` sleeps the application until the frame deadline, calculates a new frame delta, and swaps the display frame buffers.

3.3.4.1 Retrieving camera frames

Retrieving frames from the camera is blocking operation. Therefore, the frame retrieval runs on a separate thread to avoid blocking processing on the rendering thread. In the initialization step, the application spawns execution of a static method `WellApplication::FrameAcquireThread`. The method enters a loop that repeatedly acquires the camera frame, copies it into a side buffer used by the renderer on another thread, and releases it back to the camera processing logic.

We encountered a few unexpected data races in the Argus library that caused visual glitches and tearing when the acquired frame was used directly by the renderer. Therefore the application must clone the acquired buffer and release it as soon as possible. ”Copy frame” block from Figure 3.20 represents the logic that clones the acquired frame buffer into another frame buffer accessed by the renderer.

The camera sends captured frames to the main memory via an attached interface, such as USB, CSI, or PCIe. Further processing of these captured frames inside the shader requires the data to be transferred into video memory, where the pixel data is accessible by integrated GPU. When processing live footage from a camera that produces tens to hundreds of frames, there can be significant delays that will not allow all of the frames produced to be processed. NVIDIA Buffer Utilities library allows us to take advantage of the fact that the memory of Jetson Nano is shared between CPU and integrated GPU.

`NvBufferCreateEx` function creates a hardware frame buffer that can be accessed without any additional copy operation by the camera sensor and integrated GPU shader unit. `SharedFrameBuffer` addon wraps the functionality

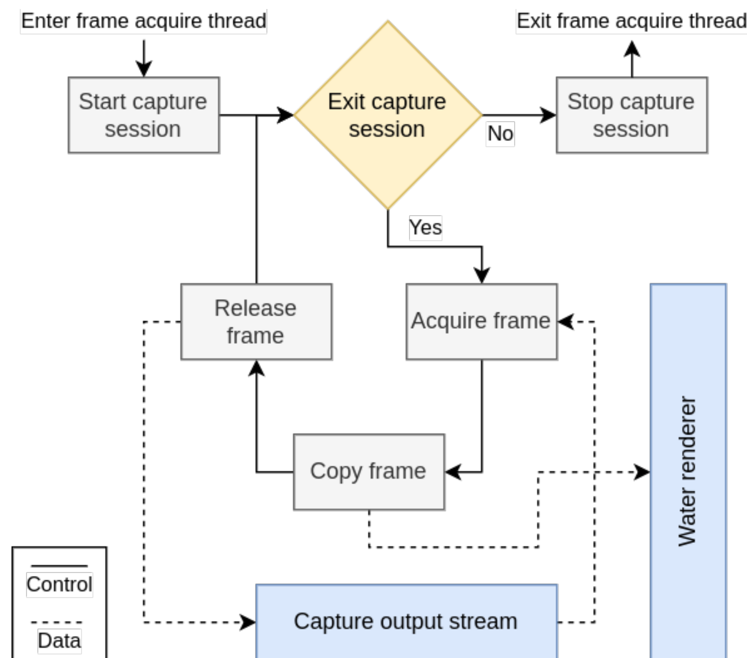


Figure 3.20: Camera frame acquire thread

of the NVIDIA Buffer library. It provides additional methods for more convenient manipulation and conversion into `EGLImageKHR` object that acts as a representative on the EGL side.

The actual camera frame is represented by the `ExternalTexture`, a wrapper around a special type GLES texture that is provided by the GLSL external image extension. The actual pixel data is then accessible inside the shader via `samplerExternalOES` that has similar properties as standard 2D texture `sampler2D` [41]. Figure 3.21 demonstrates how to assign a shared frame buffer to `ExternalTexture`.

```
SharedFrameBuffer cameraFrameBuffer;
cameraFrameBuffer.Create(graphicsContext,          // Context
                         1920, 1080,                // Resolution
                         NvBufferColorFormat_NV12); // Format

ExternalTexture cameraFrameTexture;
cameraFrameTexture->Create();
cameraFrameTexture->Bind();
cameraFrameTexture->Data(graphicsContext, cameraFrameBuffer);
cameraFrameTexture->Unbind();
```

Figure 3.21: Creating OpenGL ES texture from shared frame buffer

3.3.4.2 Rendering

The rendering of the water surface and the surrounding walls takes three steps. The first two steps render the scene from two different perspectives while omitting some parts of the scene. Both results are stored into textures for later use instead of storing the result onto the display frame buffer. These two textures are then used to render the complete scene in the third step. The three steps are:

- **Refraction pass** renders the scene under the water surface while culling the geometry above the water surface;
- **Reflection pass** renders the scene from the water's view towards the well's entrance while culling the geometry below the water surface;
- **Scene pass** then takes reflection and refraction textures, combines them to create an illusion of a water surface with refraction and reflection properties, and adds distortion to mimic a wavy surface.

More details about the rendering system can be found in subsection 3.3.5.

3.3.4.3 Synchronization

OpenGL ES rendering commands are asynchronous. Whenever the application invokes a function to initiate rendering, it is not guaranteed by the specification that the rendering has finished when the call returns. The synchronization process in the graphics context ensures that the graphics API rendering pipeline executed all the commands queued for execution. Modern graphics APIs, such as OpenGL ES 3.0, feature multi-threading support and allow synchronization via synchronization objects. The virtual mirror framework implements the synchronization logic in `RenderApplication::Sync` method. This method blocks the execution on the current thread until all queued rendering operations are executed.

3.3.4.4 Flushing

At the end of the frame rendering, the application waits for the frame deadline to meet the required frame rate. Then it calculates the frame delta used in scene update logic to maintain independent animation speed from rendering speed. Finally, the application commands the graphics context to swap the display frame buffers to present the updated frame to the physical display.

3.3.5 Rendering system

The rendering system consists of multiple sub-rendering systems that can draw different types of meshes into the display frame buffer. A model is a collection of different types of meshes. The scene of this exhibit consists of a single model

that contains all types of mentioned meshes. Mode details about the exhibit's scene are described in subsection 3.3.6. There are three types of meshes in this implementation:

- `BasicMesh` represents basic 3D textured mesh that can use diffuse, normal, and specular textures;
- `CameraFrameQuad` represents a 2D quad of variable size that projects the physical camera frames onto its surface. The shader that outputs the fragments representing the quad uses a special external texture that contains camera frame pixel data;
- `WaterMesh` represents a water surface that refracts the scene under the water and reflects the scene above the water. The shader that outputs the fragments representing this surface mixes refraction and reflection and applies slight distortion that updates in time to simulate water movement.

3.3.5.1 Mesh

A mesh is a collection of interconnected vertices that form a structure of a model. Vertices and their arrangement of interconnections are represented by vertex array object, a list of vertex buffer objects that are automatically bound when the vertex array object is bound. The following two types of vertex buffer objects are on the list of the vertex array object:

- **Vertex data buffer** contains data associated with vertices. Each vertex consists of its position, normal vector, texture coordinates, and tangent vector. `VertexAttributesConfig` helper class is used to generate GLES commands that configure the layout of the vertex data buffer;
- **Indices buffer** contains indices to vertices in vertex data buffer. GLES draw command uses these indices to render primitives from the vertex data buffer.

3.3.5.2 Shaders

Rendering a geometry into a frame buffer takes a sequence of steps, also referred to as a rendering pipeline. The rendering pipeline is initiated whenever the application invokes a rendering operation. Some parts of the rendering pipeline are programmable by the application programmer, while others are implemented by a GPU vendor and cannot be changed. A simplified version of the OpenGL ES pipeline without additional optional shader stages is shown in Appendix E. The shader is a small user-defined program executed by shader units of GPU in different programmable stages of the rendering pipeline. Shaders are written in GLSL, a C-style programming language. The implementation stores all shader program source files in `Resources/Shaders`

directory, where each shader program source files has the `<name>.<type>` naming pattern.

3.3.5.3 Renderers

The renderer takes care of configuring the graphics API to render a specific type of mesh using the appropriate shader system. It handles the activation of the appropriate shader program that can draw the geometry of the given mesh, followed by uploading the necessary uniform variables to the shader program. In order to render the complete model, the model have to pass through all renderers associated with its mesh types.

3.3.6 Scene

In order to achieve the desired effect, the scene is split into three objects. Each of them is rendered by a different shader system. The following sections describe the methods used to render different object types.

3.3.6.1 Walls

The walls surround the water and have a clay texture with small rocks and cracks. Figure 3.22 shows rendered walls in Blender (on the left) and the implementation rendering system (on the right). They are rendered by the `BasicRenderer` that renders static textured meshes using `BasicShader`. The shader has the following properties:

- **Phong lighting model** with light located behind the camera;
- **Normal vector mapping** to make the cracks in the walls more realistic;
- **Attenuation** for decreasing the visibility after entering the water;
- **Corner darkening** to simulate superficial ambient occlusion.

3.3.6.2 Water

The water surface quad fakes the surface of the water in the well. The surface refracts the scene under the water and reflects the scene above the water. Before merging the refraction and reflection into the resulting texture, both components are slightly distorted using the DuDv map. The DuDv map is similar to a normal map in storing directional information, except it only stores two components of 2D direction. The directional information transforms the original uniform texture mapping to a wavy pattern. The offset of the DuDv map is additionally updated in time to produce the effect of the moving waves. Figure 3.23 shows the DuDv map used for producing the distortion effect and its effect when applied to the exhibit scene.

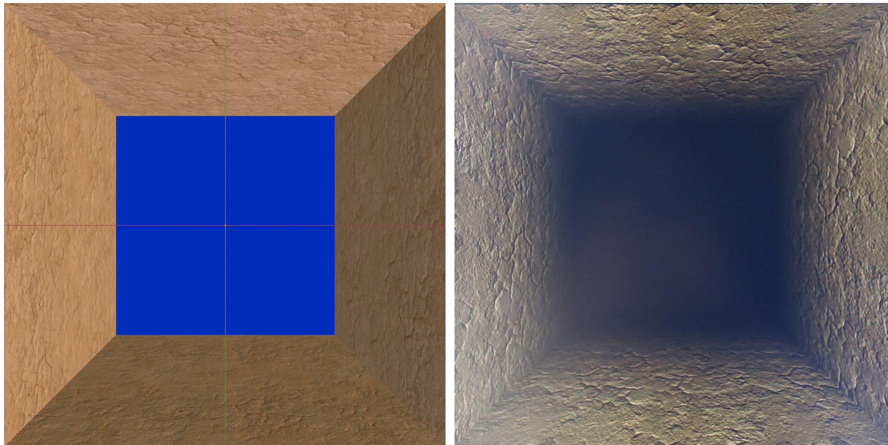


Figure 3.22: Walls of the well

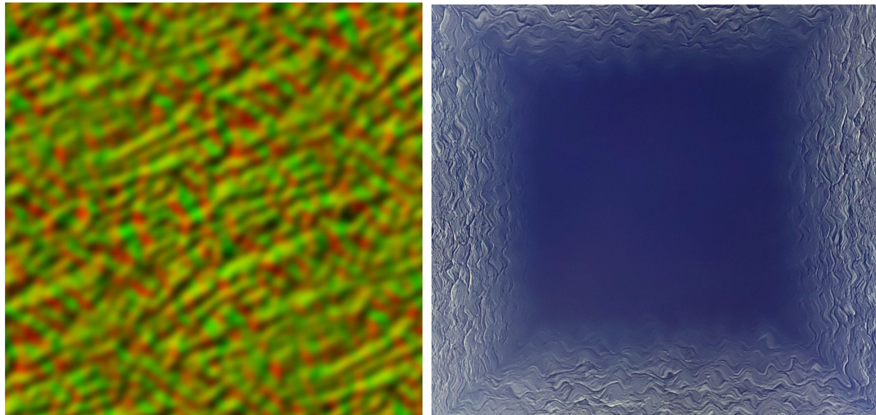


Figure 3.23: Distortion using DuDv map

3.3.6.3 Camera frame

The camera frame quad projects what is happening outside the well (in the real world) from the view of the water surface. It uses physical camera frames as its texture. Before mapping to the surface of the quad, the camera frame is blurred using the Gaussian blur function to decrease the image's high-frequency components. The quad's surface is then captured as a reflection from the water surface in the Reflection pass step from subsubsection 3.3.4.2 of the model rendering.

3.4 Deployment

The application was deployed in an existing exposure of an old wooden well from the Middle Ages. The well has a square shape and is approximately 1.8 meters deep and 70 centimeters wide. Visitors have a limited view into the

well, only from certain angles. The appearance of the exhibit can be seen in Figure 3.24.

Also of a square shape, a display device of size 60x60 centimeters and resolution of 1920x1920 pixels is located approximately 10 centimeters above the bottom of the well model to allow free passage of cables present for powering the display device and Jetson Nano. The screen surface is additionally protected against possible objects falling into the exposure by a plastic shield.

Figure 3.24: Museum exhibit

The short camera cable that forced us to install the device very close to the display device. Therefore, the Jetson Nano device is positioned right next to the display as shown in Figure 3.25.

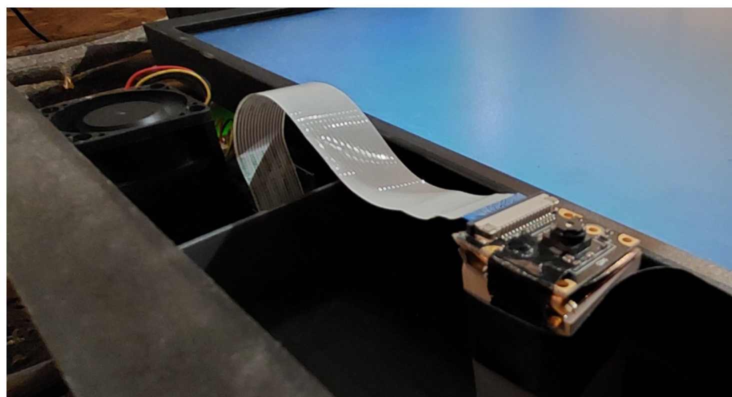


Figure 3.25: Jetson Nano placement

Figure 3.26: Application deployed into exhibit

3.4.1 Overheating

The higher resolution of the display device and closed operating space with no airflow contributed significantly to the operating temperature of the Jetson Nano deployed in the exposure. Compared to the measurements from the development phase, the temperature rose from 47 degrees to approximately 67 degrees, which began to cause occasional shutdowns. It takes a few minutes for Jetson Nano to cool down, and it is not possible to turn it on while it is cooling down. That caused downtimes, which are not acceptable in this situation.

The problem has been resolved by installing a small 40-millimeter cooling fan that blows air to the heat sink that dissipates the heat of the Jetson Nano SoC. The cooling fan is controlled by a script that periodically reads the CPU and GPU temperature and sets the appropriate PWM fan speed to stabilize the temperature below the critical value. The exhibit environment is dusty, which might later, in combination with the cooling fan, require maintenance for dusting off the hardware.

3.4.2 Glossy display surface

Plastic protection used to prevent damage in the event of a foreign object falling into the well is made of glossy material that, on top of the application's shader effects, unfavorably contributes to the shininess of the virtual water surface. In combination with the strong glaring of the ceiling lights, this unwanted effect makes the render of the water surface less realistic. A simple remedy to this problem is to either relocate the position of the ceiling lights or decrease the specular shader strength while finding a better replacement for the plastic protection is still desired.

Figure 3.27: Visible reflection on display plastic protection

3.4.3 Calibration

The construction and scale of the development setup differ significantly from the construction of the museum exhibit, primarily due to a lack of information. Photos provided at the beginning of the work only gave us a rough idea of the exhibit dimensions. Therefore, it was necessary to perform additional calibration of the quad that projects the camera frames. We implemented an interactive quad calibration using keyboard arrow keys that allows the technician to adjust the offset and zoom of the projected camera frame. Figure 3.28 describes the calibration control. The precision row represents the number of units per keystroke.

Keyboard key	Description	Precision
+ (plus)	Increase the scale	0.0250
− (minus)	Decrease the scale	0.0250
↑ (up arrow)	Increase the position along the Y axis	0.0125
↓ (down arrow)	Decrease the position along the Y axis	0.0125
→ (right arrow)	Increase the position along the X axis	0.0125
← (left arrow)	Decrease the position along the X axis	0.0125

Figure 3.28: Project structure

Conclusion

This work aimed to explore appropriate computer systems suitable for deploying graphics applications into museum exhibits. The author considers minicomputers as great candidates for this purpose as their recent development ensures their sufficient calculating capacity and optimal size for flexible incorporation into other hardware while at the same time being cost-efficient. While experimenting, implementing, and testing a prototype virtual mirror application in a museum operation, the author successfully designed the framework for the virtual mirror for the selected platform. The programmer is given complete control of the captured camera image in the GLSL shader, expanding the application possibilities even more than initially anticipated.

Furthermore, the automated system for creating a modified SD card image, which came about as a by-product of this work, allows developers to rapidly customize the OS and potentially reduce the requirements for the device specifications. In the case of the subject of this thesis, a 2GB version of Jetson Nano, having its OS undergoing customization, turns out to meet the overall resource requirements. It is worth noting that the version with less memory costs roughly 50% less than the initially chosen device.

Experience obtained from this project indicates that if one can minimize the overhead of an implemented framework by cleverly choosing the suitable technologies, the potential for complex and powerful graphical applications is endless. Due to its modularity design, the framework of the current project leaves room for further extension. Instead of limiting itself to applications that only utilize a single camera, the framework will become suitable for a much more comprehensive range of usage after a small expansion, such as taking on the multiple cameras system.

Future improvements

Camera latency

While the FPS was satisfactory, a slight latency is still perceptible in the resulting implementation. Measured delay exceeds the previously established limits in subsection 1.2.1 that are perceivable by humans. At a resolution of 3264 x 2464, Sony IMX219 sensor is capable of recording at 21 FPS, introducing $47.62ms$ latency. The rendering speed is set to 30 FPS, introducing an additional delay of $33.\overline{3}ms$. The resulting maximum theoretical delay should be thus $47.62ms + 33.\overline{3}ms \approx 81ms$. Later discovered behavior of image signal processor (ISP) when using Argus library multiplies the camera latency by a factor of 4 [42], resulting in a maximum theoretical delay of $4 * 47.62ms + 33.\overline{3}ms \approx 224ms$. Using a 60 FPS mode while sacrificing the higher resolution could decrease the delay to $4 * 16.\overline{6}ms + 33.\overline{3}ms \approx 100ms$. Figure 3.29 shows a comparison of a timer (on the left) and its captured frame via the virtual mirror application (on the right) to determine the camera-to-screen latency. A secondary camera captures the actual comparison. In this case, the measured difference is about $1.492s - 1.338s = 154ms$. The latency issue and its testing could be a topic for future work.

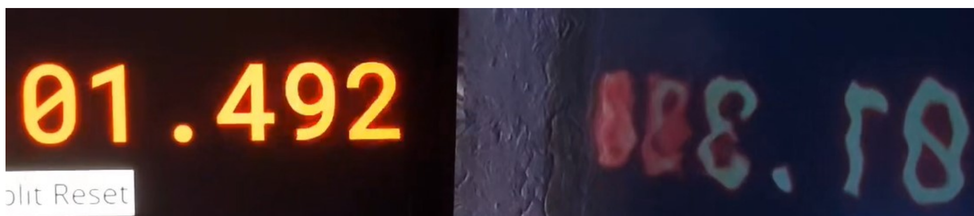


Figure 3.29: A secondary camera capturing the camera-to-screen latency

Wayland

The implementation of the museum exhibit uses the X windowing system for displaying the contents onto a physical display, which is a very old technology made since 1984. It has many flaws and potential security issues and has become incompatible with the design of modern hardware and applications [43]. A possibly better alternative could be Wayland, a modern windowing system also being supported by L4T. According to NVIDIA, Wayland performs better and is more suitable for many embedded and mobile use cases [44].

Bibliography

[1] Pachoulakis, I.; Kapetanakis, K. Augmented Reality Platforms for Virtual Fitting Rooms. *The International journal of Multimedia & Its Applications*, volume 4, August 2012: pp. 35, 41, doi:10.5121/ijma.2012.4404.

[2] Waltemate, T. Creating a Virtual Mirror for Motor Learning in Virtual Reality. 2018, doi:10.4119/unibi/2932705.

[3] Raaen, K.; Eg, R.; et al. Can gamers detect cloud delay? In *2014 13th Annual Workshop on Network and Systems Support for Games*, IEEE, 2014, p. 1.

[4] Should I take an actively cooled or passively cooled Mini-PC? [online; accessed 2022-05-04]. Available from: `https://www.spo-comm.de/en/blog/know-how/should-i-take-an-actively-cooled-or-passively-cooled-mini-pc`

[5] Staff, E. Is a single-chip SOC processor right for your embedded project? August 2013, [online; accessed 2022-05-07]. Available from: `https://www.embedded.com/is-a-single-chip-soc-processor-right-for-your-embedded-project/`

[6] Raspberry Pi 4 Tech Specs. [online; accessed 2022-03-18]. Available from: `https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/`

[7] („Laserlicht“), M. H. Raspberry Pi 4 Model B from the side. [online; accessed 2022-05-08]. Available from: `https://commons.wikimedia.org/wiki/File:Raspberry_Pi_4_Model_B_-_Side.jpg`

[8] Attribution-ShareAlike 4.0 International. Available from: `https://creativecommons.org/licenses/by-sa/4.0/legalcode`

[9] Jetson Modules. [online; accessed 2022-03-16]. Available from: https://developer.nvidia.com/embedded/jetson-modules

[10] Jetson Nano Developer Kit. [online; accessed 2022-03-18]. Available from: https://developer.nvidia.com/embedded/jetson-nano-developer-kit

[11] NVIDIA Jetson Nano Developer Kit. [online; accessed 2022-03-16]. Available from: https://www.sparkfun.com/products/16271

[12] Spark Fun Electronics. [online; accessed 2022-03-16]. Available from: https://www.sparkfun.com

[13] Attribution 2.0 Generic. Available from: https://creativecommons.org/licenses/by/2.0/legalcode

[14] Frumusanu, A. ARM Reveals Cortex-A72 Architecture Details. April 2015, [online; accessed 2022-03-19]. Available from: https://www.anandtech.com/show/9184/arm-reveals-cortex-a72-architecture-details

[15] Luigi_Morelli. Jetson Nano vs Raspberry PI 4 – CPU comparisons. August 2019, [online; accessed 2022-03-25]. Available from: https://www.moreware.org/wp/blog/2019/08/01/jetson-nano-vs-raspberry-pi-4-cpu-comparisons/

[16] MIPI Camera Serial Interface 2 (MIPI CSI-2). [online; accessed 2022-03-18]. Available from: https://www.mipi.org/specifications/csi-2

[17] Jetson Partner Supported Cameras. [online; accessed 2022-03-18]. Available from: https://developer.nvidia.com/embedded/jetson-partner-supported-cameras

[18] Max.kit. Eiffel Tower under cloudy sky. [online; accessed 2022-05-01]. Available from: https://commons.wikimedia.org/wiki/File:Eiffel_Tower_under_cloudy_sky.jpg

[19] Bird reflected in water. [online; accessed 2022-05-01]. Available from: https://pixabay.com/photos/bird-water-nature-pool-wildlife-3072175/

[20] Pixabay License. [online; accessed 2022-05-01]. Available from: https://pixabay.com/service/license/

[21] Jabeen, Q.; Khan, F.; et al. A survey: Embedded systems supporting by different operating systems. *arXiv preprint arXiv:1610.07899*, 2016.

[22] Linux range of use. [online; accessed 2022-03-22]. Available from: https://en.wikipedia.org/wiki/Linux_range_of_use#Embedded_devices

[23] Jetson Linux R32.7.1 Release Page. [online; accessed 2022-03-29]. Available from: https://developer.nvidia.com/embedded/linux-tegra-r3271

[24] L4T Archive. [online; accessed 2022-03-28]. Available from: https://developer.nvidia.com/embedded/jetson-linux-archive

[25] X.Org. [online; accessed 2022-04-01]. Available from: https://xorg.freedesktop.org/wiki/

[26] OpenGL Mathematics documentation. [online; accessed 2022-04-07]. Available from: http://glm.g-truc.net/0.9.8/api/index.html

[27] Lento, G. Optimizing Performance with Intel Advanced Vector Extensions. Technical report, Intel Corporation, September 2014, [online; accessed 2022-04-15]. Available from: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/performance-xeon-e5-v3-advanced-vector-extensions-paper.pdf

[28] OpenGL Image documentation. [online; accessed 2022-04-07]. Available from: http://gli.g-truc.net/0.8.2/api/index.html

[29] Richardds, D. NVIDIA Developer Forums - Source code of nvbuf_utils. [online; accessed 2022-04-06]. Available from: https://forums.developer.nvidia.com/t/source-code-of-nvbuf-utils/179079/3

[30] NVIDIA Corporation. *Libargus Camera API*. [online; accessed 2022-04-06]. Available from: https://docs.nvidia.com/jetson/l4t-multimedia/group__LibargusAPI.html

[31] Microsoft Corporation. *DDS*. [online; accessed 2022-04-07]. Available from: https://docs.microsoft.com/en-us/windows/win32/direct3ddds/dx-graphics-dds

[32] Dominé, S. Using Texture Compression in OpenGL. Technical report, NVIDIA Corporation, 2000, [online; accessed 2022-04-07]. Available from: https://web.archive.org/web/20041120095329/http://developer.nvidia.com/attach/6585

[33] Jetson Download Center. [online; accessed 2022-03-16]. Available from: https://developer.nvidia.com/embedded/downloads

[34] mau, D. NVIDIA Developer Forums - Jetson Nano Image without Desktop. [online; accessed 2022-03-28]. Available from: https://forums.developer.nvidia.com/t/jetson-nano-image-without-desktop/176571/3

[35] Ji, C.; Chang, L.-P.; et al. An Empirical Study of File-System Fragmentation in Mobile Storage Systems. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, Denver, CO: USENIX Association, June 2016, [online; accessed 2022-03-30]. Available from: https://www.usenix.org/conference/hotstorage16/workshop-program/presentation/ji

[36] bwana, d., Honey_Patouceul. NVIDIA Developer Forums - bios. [online; accessed 2022-04-07]. Available from: https://forums.developer.nvidia.com/t/bios/73787/3

[37] NVIDIA Corporation. *Jetson Nano Boot Flow*. [online; accessed 2022-04-09]. Available from: https://docs.nvidia.com/jetson/l4t/index.html#page/Tegra%20Linux%20Driver%20Package%20Development%20Guide/bootflow_jetson_nano.html

[38] NVIDIA Corporation. *Partition Configuration*. [online; accessed 2022-04-07]. Available from: https://docs.nvidia.com/jetson/l4t/index.html#page/Tegra%20Linux%20Driver%20Package%20Development%20Guide/part_config.html

[39] Khronos Group. *Introduction to managing client API rendering through the EGL API*. [online; accessed 2022-04-15]. Available from: https://www.khronos.org/registry/EGL/sdk/docs/man/html/eglIntro.xhtml

[40] Gregory, J. *Game Engine Architecture, Second Edition*. USA: A. K. Peters, Ltd., second edition, 2014, ISBN 1466560010.

[41] Khronos Group. *OES_EGL_image_external*. [online; accessed 2022-05-03]. Available from: https://www.khronos.org/registry/OpenGL/extensions/OES/OES_EGL_image_external.txt

[42] ShaneCCC. Low latency camera and software drivers? [online; accessed 2022-04-10. Available from: https://forums.developer.nvidia.com/t/low-latency-camera-and-software-drivers/109966/2

[43] Why Use Wayland versus X11? [online; accessed 2022-04-28]. Available from: https://www.cbtnuggets.com/blog/technology/networking/why-use-wayland-versus-x11

[44] Weston (Wayland) Windowing System. [online; accessed 2022-04-28]. Available from: https://docs.nvidia.com/jetson/l4t/index.html#page/Tegra%20Linux%20Driver%20Package%20Development%20Guide/window_system_wayland.html#wwpID0E0FJ0HA

Appendix A

Acronyms

API application programming interface

CoM computer on module

CMOS complementary metal–oxide–semiconductor

CPU central processing unit

CSI camera serial interface

DDS Microsoft DirectDraw Surface

EGL embedded-system graphics library

eMMC embedded MultiMedia Card

FOV field of view

FP16 half-precision floating-point format

FPS frames per second

GDB GNU project debugger

GFLOPS giga floating point operations per second

GLM open graphics library mathematics

GLSL open graphics library shading language

GPIO general-purpose input/output

GPU graphics processing unit

HDMI high-definition multimedia interface

I2C inter-integrated circuit

I2S inter-integrated circuit sound

ISP image signal processor

JPEG Joint Photographic Experts Group

KTX Khronos Texture

L4T Linux for Tegra

LPDDR low-power double data rate

OGL, GL open graphics library

OGLES, GLES open graphics library for embedded systems

PCB printed circuit board

PCIe peripheral component interconnect express

PWM pulse-width modulation

RJ registered jack

rootfs root filesystem

RTOS real-time operating system

S3TC S3 Texture Compression

SD secure digital

SDK software development kit

SoC system on chip

SoM system on module

SPI serial peripheral interface

STL standard library

UART universal asynchronous receiver-transmitter

USB universal serial bus

Appendix B

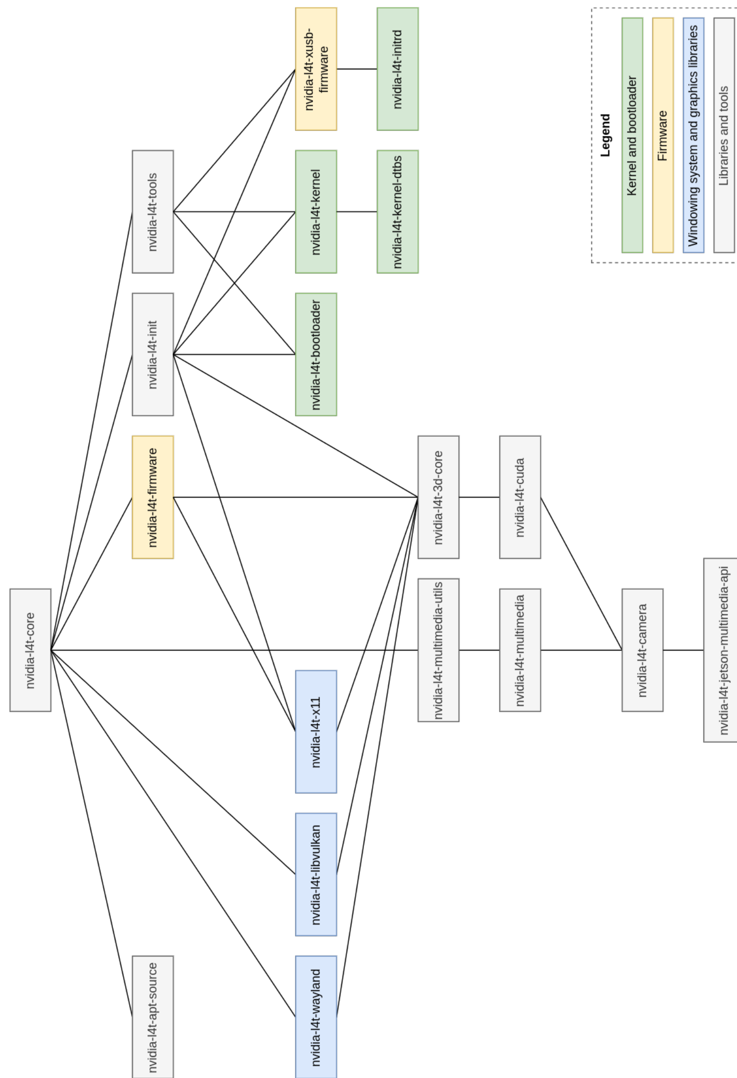
Contents of enclosed CD

```
dist .............................................. distributable files
├── jetson-sd-card ................................... SD card images
│   ├── id_jetson .......................... SD card image private key[1]
│   ├── id_jetson.pub ....................... SD card image public key
│   └── jetson.img.zst .................... production SD card image
└── well ...................................... implementation binaries
    └── JetsonWell ......... ARM64 executable compiled in release mode
src ................................................... source codes
├── system-image ......... source codes of the SD card image automation
├── thesis ............................ LaTeX source codes of the thesis
└── well ........................... source codes of the implementation
video ................................................... video files
├── 00-latency-test.mp4 ............... Camera processing latency test
├── 01-home-dev.mp4 ......................... Home development setup
├── 02-exhibit.mp4 ................. Museum exhibit before deployment
└── 03-digital-exhibit.mp4 ......... Museum exhibit after deployment
thesis.pdf ............................ the thesis text in PDF format
```

[1] Change the reference key pair soon as possible.

Appendix C

NVIDIA L4T packages



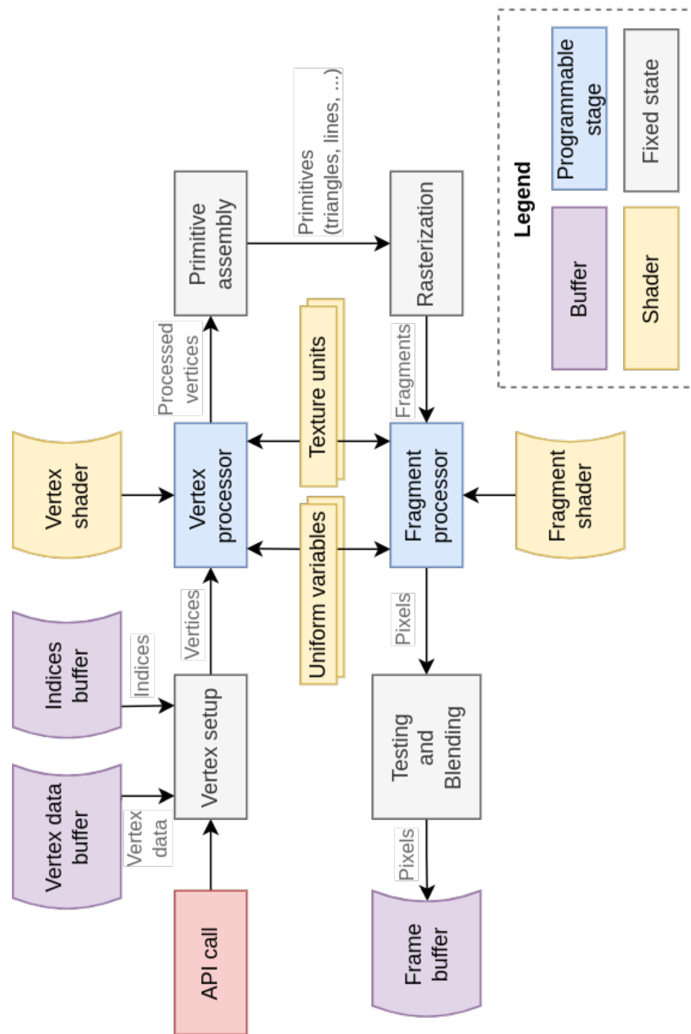
Appendix D

SD card image build automation structure

- `config.env` ... environment variables for emulated environment
- `docker-compose.yml` ... Docker compose configuration
- `Dockerfile` ... Docker image of SD card image build environment
- `files` ... configuration files for rootfs
- `packages` ... debootstrap packages cache
- `README.md` ... documentation
- `scripts` ... build scripts
  - `000-build.sh` ... starting point of SD card image build
  - `001-base-system.sh` ... base system using debootstrap
  - `002-system-config.sh` ... hostname and root password
  - `010-repositories.sh` ... additional repositories
  - `011-nvidia.sh` ... NVIDIA software
  - `012-network.sh` ... network renderer and firewall
  - `013-ssh.sh` ... SSH remote access
  - `014-display.sh` ... windowing system
  - `015-runtime.sh` ... application runtime libraries
  - `016-development.sh` ... development tools, libraries and header files
  - `017-app.sh` ... first-boot setup and application service
  - `100-cleanup.sh` ... clean up of cache and log files
  - `200-create-fs-images.sh` ... create various filesystem images
  - `201-allocate-image.sh` ... SD card image file allocation
  - `202-create-partitions.sh` ... create GPT table and partitions
  - `203-write-partitions.sh` ... populate partitions
  - `300-export.sh` ... compress SD card image with zstd compression

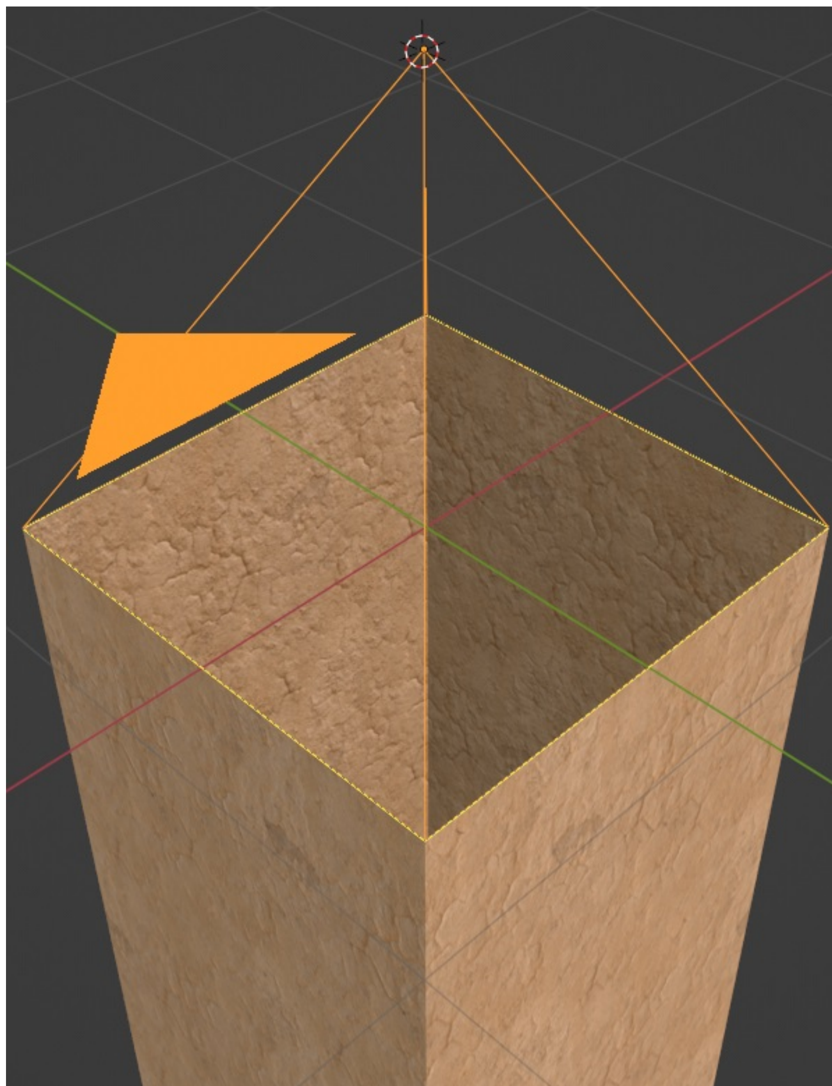
Appendix E

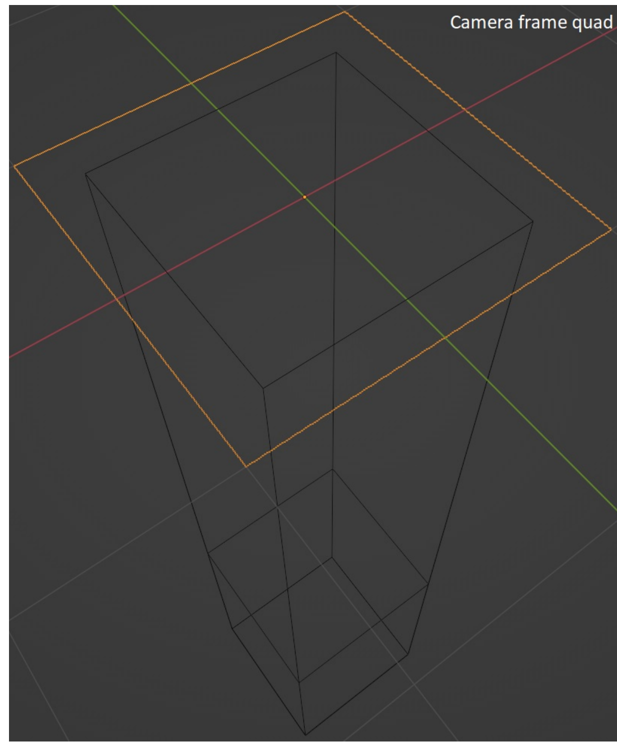
Rendering pipeline

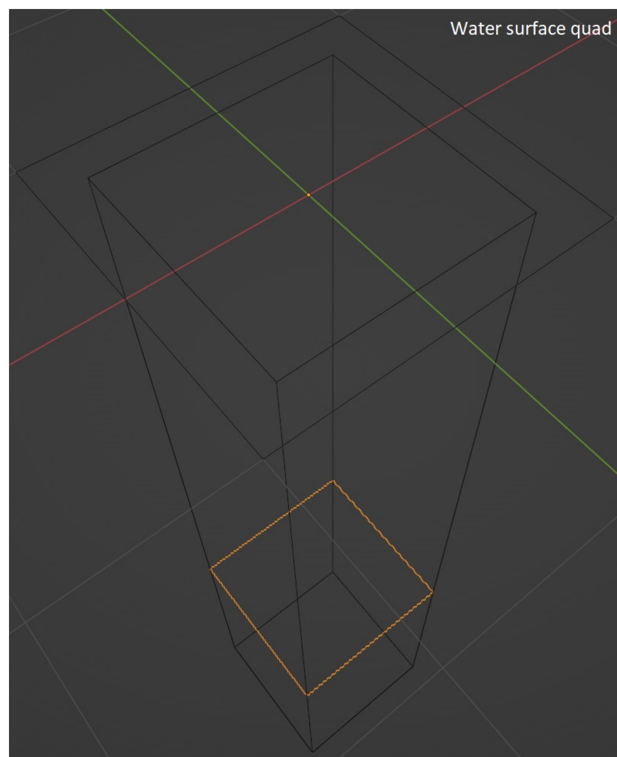


Appendix F

3D model

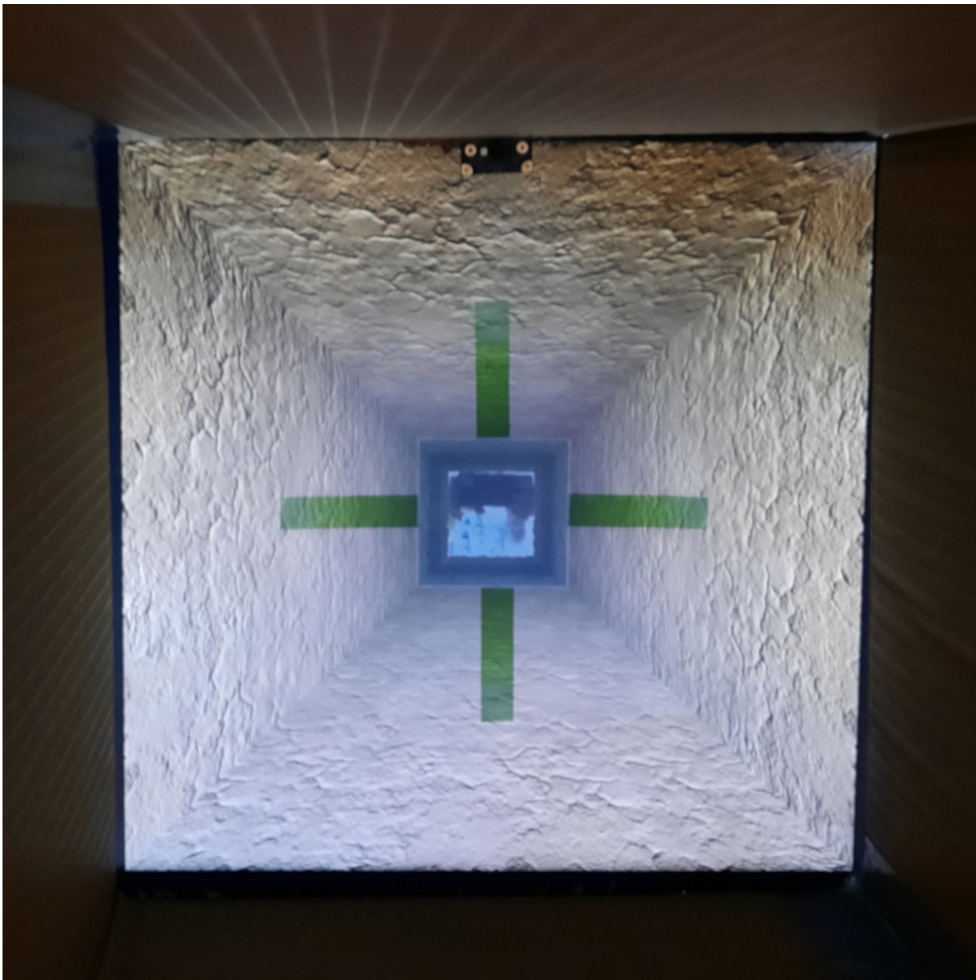



Camera frame quad


Water surface quad

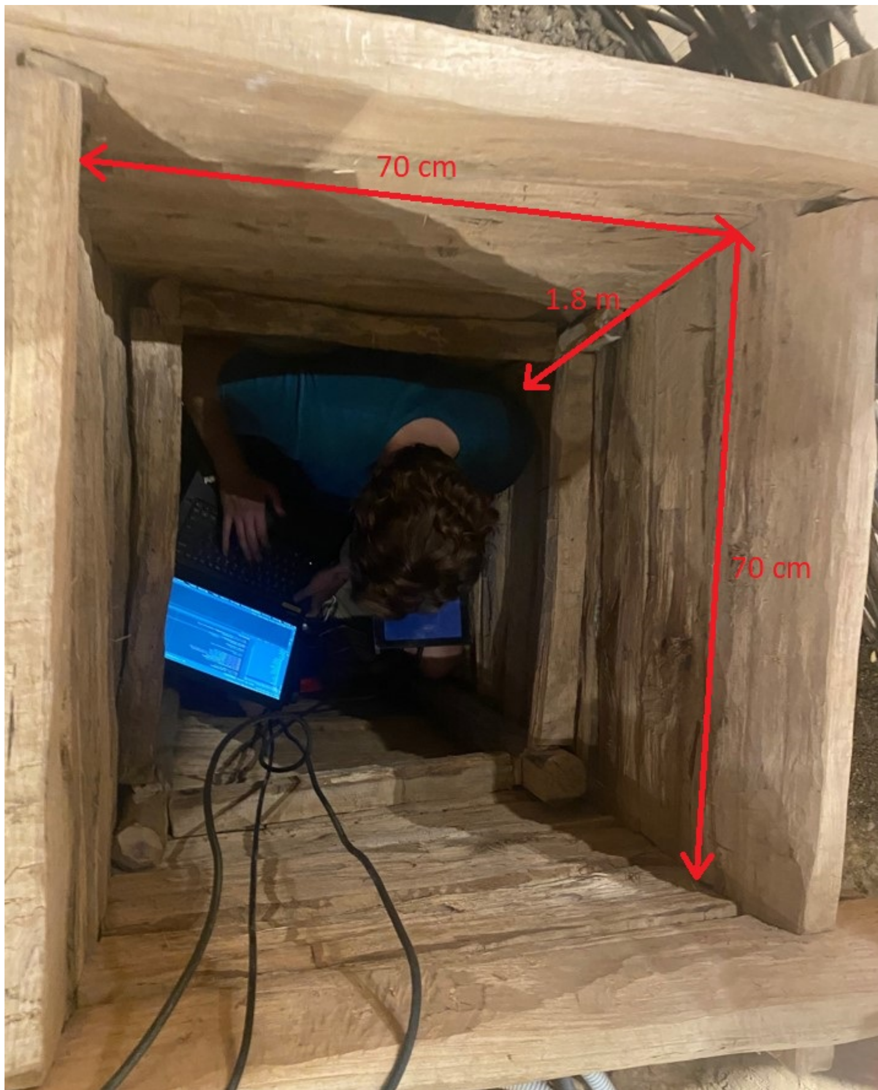
Appendix G

Development model view



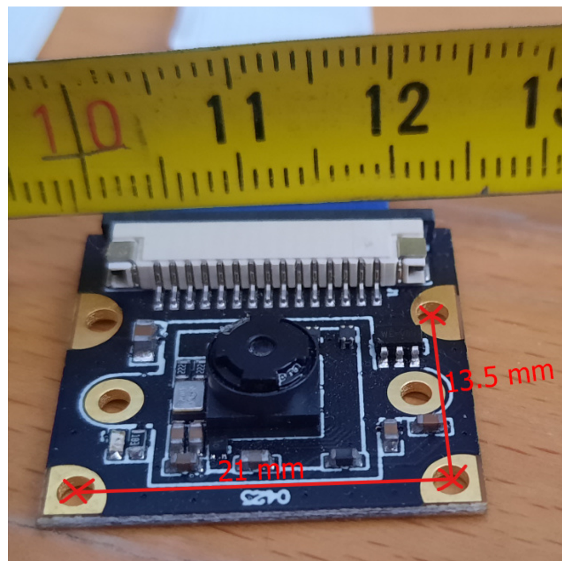
Appendix H

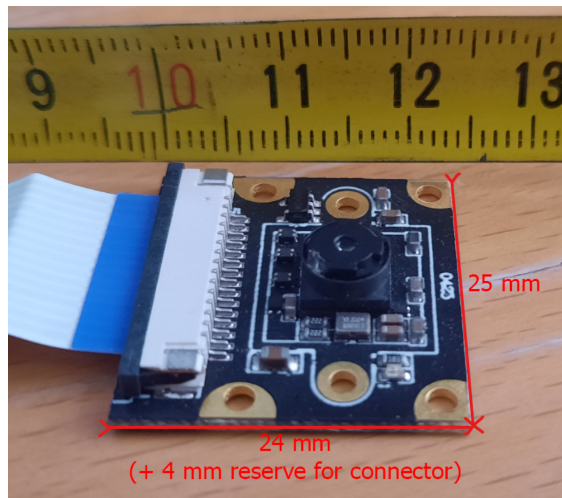
Museum exhibit dimensions

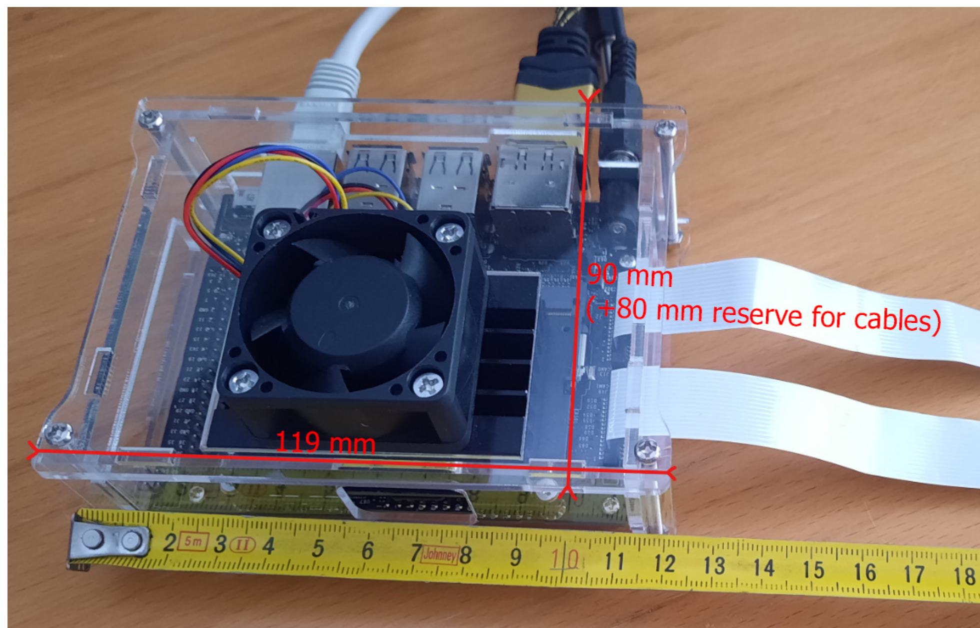


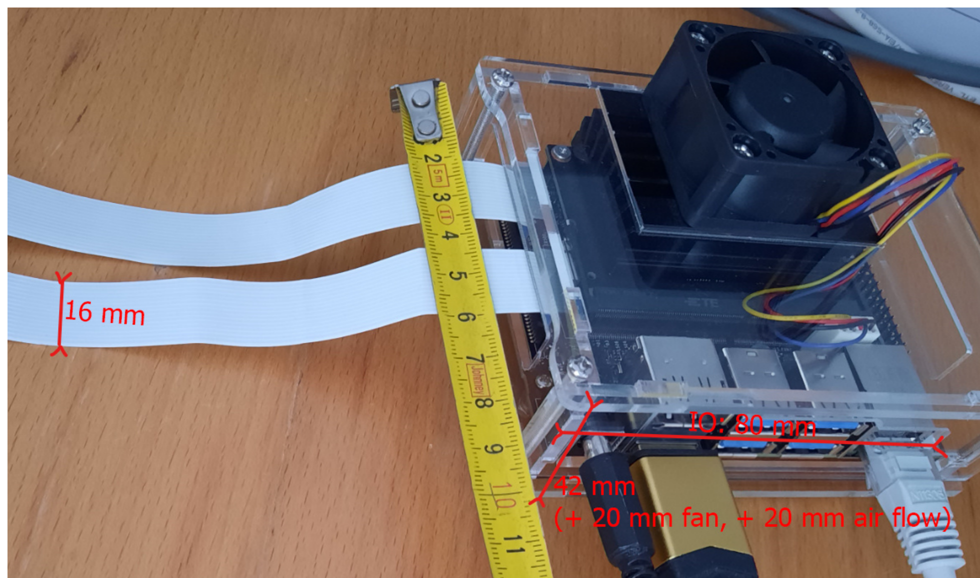
Appendix I

Device dimensions






90 mm
(+80 mm reserve for cables)
119 mm


16 mm
IO: 80 mm
42 mm
(+ 20 mm fan, + 20 mm air flow)