SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

The Z3 in VR - An Interactive Learning Environment for One of the First Computers

Lukas Moersler

SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY

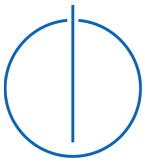
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

The Z3 in VR - An Interactive Learning Environment for One of the First Computers

Die Z3 in VR - Eine interaktive Lernumgebung für einen der ersten Computer

Author:	Lukas Moersler
Supervisor:	Prof. Gudrun Klinker, Ph.D.
Advisor:	Dr. rer. nat. David A. Plecher, M.A.
Submission Date:	15.09.2023



I confirm that this bachelor's thesis in informatics: games engineering is my own work and I have documented all sources and material used.

Munich, 15.09.2023 Lukas Moersler

Acknowledgments

This project would not have been possible without the plentiful resources provided by the *Konrad Zuse Internet Archive* [8], and especially professor Raúl Rojas, whose works on Konrad Zuse's calculating machines are the de facto primary resource for anyone intending to learn about them.

Abstract

This paper presents an interactive simulation of Konrad Zuse's Z3 computer in a Virtual Reality environment to give users the full experience of working with this machine. It details the creation of the project in various aspects, documents findings from developing and working with it, and evaluates how effective it is as an educational tool in terms of teaching users about the Z3 itself and some concepts of low-level programming.

Contents

1. Introduction

Teaching people about historically significant places and objects through facts and figures alone may be an effective approach in some cases, but inherently interactive things such as the first computers demand more than that. Virtual Reality lends itself to creating immersive experiences of such devices, letting users relive what it was like to work with them, and giving a far better impression of these machines than a textbook ever could. This project (hereafter referred to as Z3VR) aims to implement such a learning environment for the Z3, arguably the first (built) programmable computer. Working with the Z3 computer is, as with most early computers, a very hands-on experience. Any attempt at simulating it through a 2D interface will inherently lose this aspect, and without actually seeing the machine in front of you and hearing it operate, that interface will only ever be an obtuse canvas of buttons and displays with an obscure programming model behind it. The historical significance of this machine should be evident, but the reasons for recreating the Z3 specifically reach further than just that.

For one, the machine is hardly known outside of limited circles due to various factors, and the author believes more efforts should be taken to rectify this.

Secondly, while there are currently some working recreations of the machine on display in museums around the world, none of them are open to the public to work with, and the one built under the supervision of Konrad Zuse himself is completely inaccessible at the time of writing. Some digital simulations of it and how they're lacking in terms of providing insight into programming the Z3 will be detailed in Chapter 2.

Another factor is that the programming model of the Z3 is remarkably similar to modern assembly languages, thus offering an opportunity to teach users not just about this machine and its historical significance, but also about hardware-level programming, in a way that the learned skills can easily be applied to modern machine languages. Interaction with the machine is also extremely simple compared to other early computers like the ENIAC or Mark I, as such the barrier to entry is low enough that users do not first need to read through pages upon pages worth of instructions to get started with it.

Finally, the relatively compact yet open layout of the machine's different components (Fig. 1.1), as well as their clear separation in terms of function, makes it a lot easier to show users what each part does and how they interact with each other.

Figure 1.1.: The 3D model created for Z3VR

The main target demographic of the project is people in their late teens and up, with at least marginal interest in computer science. Z3VR places users in an interactive scene where they can get the full experience of working with the Z3. Various interfaces are provided to make programming the machine as accessible as possible while trying to maintain immersion and historical accuracy. This paper will attempt to answer whether this is an effective method for teaching users about this machine and low-level programming concepts.

Chapter 2 will go into prior works concerning simulations of the Z3 and other historical computers, plus some other VR teaching environments. The following Chapter (3) covers Z3VR in more detail, explaining the different interfaces implemented within it and the methods used to recreate the machine. Chapter 4 explores observations made while working with the Z3 that are hardly mentioned in other literature, and ends with an example program exhibiting conclusions drawn from those observations. Chapter 5 presents and evaluates the data gathered from playtests, with emphasis on the two iterations of Z3VR's introduction system and their respective reception. Chapter 6 outlines aspects of the project that were planned but ultimately cut due to time constraints, as well as lessons learned and how these things could be of interest for future works. The last Chapter (7) gives a summary of the paper and project, and an outlook for the future of Z3VR.

2. Related Work

2.1. Previous Z3 simulations

Aside from the mentioned physical reconstructions of the Z3, there have been various digital simulations and emulations of the machine, with varying degrees of accuracy. Details about how the Z3 operates will be covered in Chapter 3, for this section it's only necessary to know that it operated on two binary floating point registers, and it both received and displayed numbers in a decimal floating point format.

Riley

A simulation by Mike Riley, published on their website [7], presents the user with a standard application window, with interactive buttons and display fields mimicking the layout of the machine's console (Fig. 2.1). Tabs along the top of the window allow the user to switch to an "Assembler" view where programs can be written or loaded, a "Memory" view where 64 memory cells are each represented by a hexadecimal number field, and a "Debug" view where extra information is shown during operation.

As Riley mentions in an included text document, numerous assumptions were made concerning the console interface and operation specifics, some of which are unfortunately erroneous. For instance, the function of the $\nearrow$, $\searrow$, and A buttons was misinterpreted as "load from memory", "write to memory", and "display result" respectively, with the third and fourth digit of the decimal mantissa input providing the address for memory operations. In reality, these buttons serve the functions of "input number", "display result", and "start program" ($A \rightarrow$ "Abtasten"). What Riley's implementation does execute faithfully though, is the algorithms used to compute the different operations. The registers are abstracted as unsigned integers, and all the algorithmic steps as described by Rojas et al. [11] are taken to compute the results. The code for this was a valuable resource for checking the correctness of Z3VR's implementation of the arithmetic unit.

PipZuseZ3

A different project named *pipZuseZ3* [6] falls short on the algorithmic accuracy aspect by abstracting the two operands as modern-day float values, and implementing the

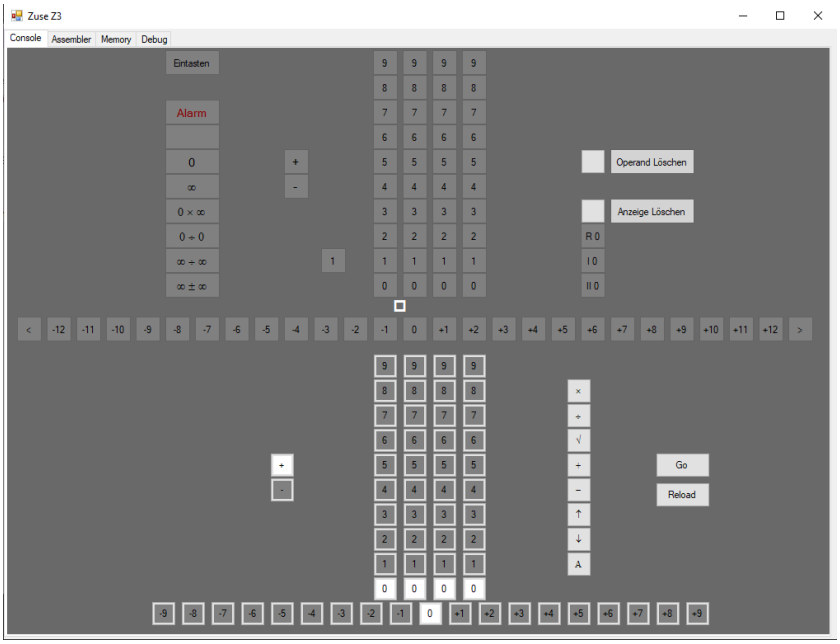


Figure 2.1.: Riley's windowed Z3 simulation

instructions by utilizing the provided float operations. The program runs in a terminal and expects the name of a text file containing a Z3 program as an argument on startup, and there is no option to simulate manual operations through the console. As the program terminates after the instruction sequence has ended, the simulated machine's memory is not persistent, making programs that operate on the results of prior programs difficult. Furthermore, there is no option to create looping programs (possible on the real machine by sticking both ends of the program together), and the simulation does not implement the Z3's exception handling.

VRML Simulation

The prior work closest to Z3VR (as far as Z3 simulations are concerned) would be the 3D simulation of the machine available on the *Konrad Zuse Internet Archive* [8, 9] (Fig. 2.2). It was constructed using the *Virtual Reality Modeling Language* [3], an archaic file format used for creating interactive 3D environments and sharing them through websites. Despite its name, it cannot be used for VR applications. Today it needs specialised software to run in a web browser, and getting this specific simulation to run requires downloading the web page and scouring through error logs to determine the file paths for the required textures to manually download those separately. Once it

does run, the user is presented with a simple 3D scene featuring the two relay cabinets and console, which can be navigated using the mouse via tank controls, or by switching between preset perspectives. The console can be interacted with by clicking on the buttons, and it displays values and exceptions by illuminating the appropriate fields. Outside of the 3D view, the web page also features a field where users can write a program to be run, though seemingly this also does not give users the option of looping programs.

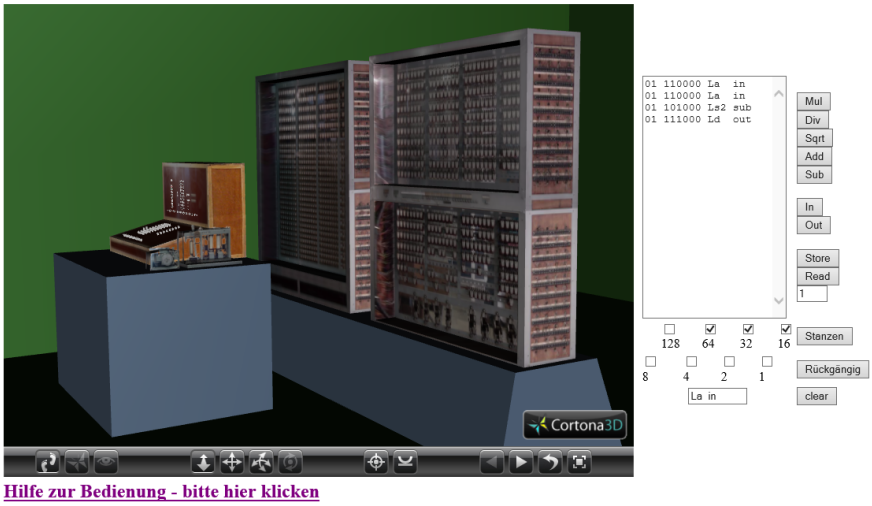


Figure 2.2.: 3D VRML scene

Unfortunately, while the console interaction and programming both work as expected, the machine itself behaves in strange ways. The button intended for displaying the output instead acts as the input button, whereas pressing the proper input button throws an exception in the simulation's output log and leaves the virtual machine running indefinitely. Pressing *A* should start the program written on the right, but instead, the machine terminates after a short delay without requesting a number input, and supplying two inputs anyway has no effect. These issues are likely caused by incompatibilities between the specific file format version used by the simulation and what the VRML software expects, and attempts at unpacking the VRML file to try and fix these were unsuccessful.

These three projects are currently the only publicly available, mostly functional simulations of the Z3. As all of them are either broken due to using archaic formats or do not represent its interface accurately, there is no way for someone to work with the Z3 as you would with the real machine, a gap that Z3VR aims to fill.

2.2. Simulations of other historic computers

Of course, the Z3 is not the only early computer to have been simulated, and it's worth taking a look at the approaches taken to simulate other pioneering machines.

ENIAC Java applet

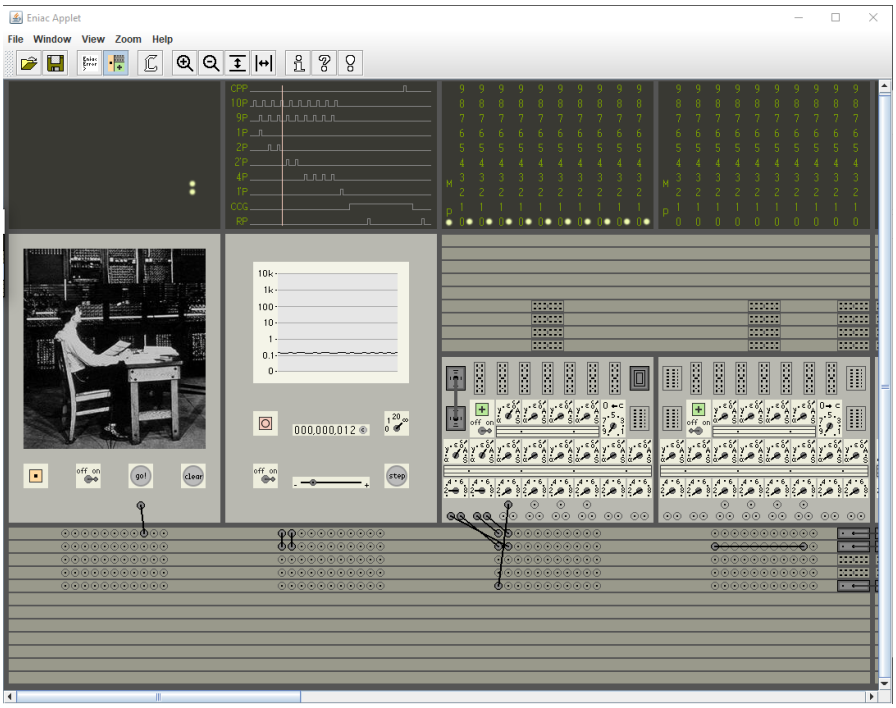


Figure 2.3.: ENIAC simulation Java applet

The *Konrad Zuse Internet Archive* also features a portable Java application which emulates the ENIAC computer [16], shown in Figure 2.3. This applet presents the user with the option to initiate the machine with example programs like Fibonacci on startup, after which a window showing a 2D representation of the ENIAC's interface is created. Unlike the Z3, programming the ENIAC required a lot of manual labour, connecting different sections of the machine with cables and turning dials to the correct settings. Due to the very horizontal nature of the ENIAC, a scroll view is used to navigate to different parts of the machine. Tooltips give the names of all visible components, and the various dials and plug sockets can be turned and connected by clicking and dragging across them. The simulation is quite function-oriented, it does not feature an introduction and the FAQ section recommends users to try and understand and modify

example programs to get started on programming. Alternatively, the accompanying paper [16] provides an in-depth guide on the programming model.

EDSAC Simulator

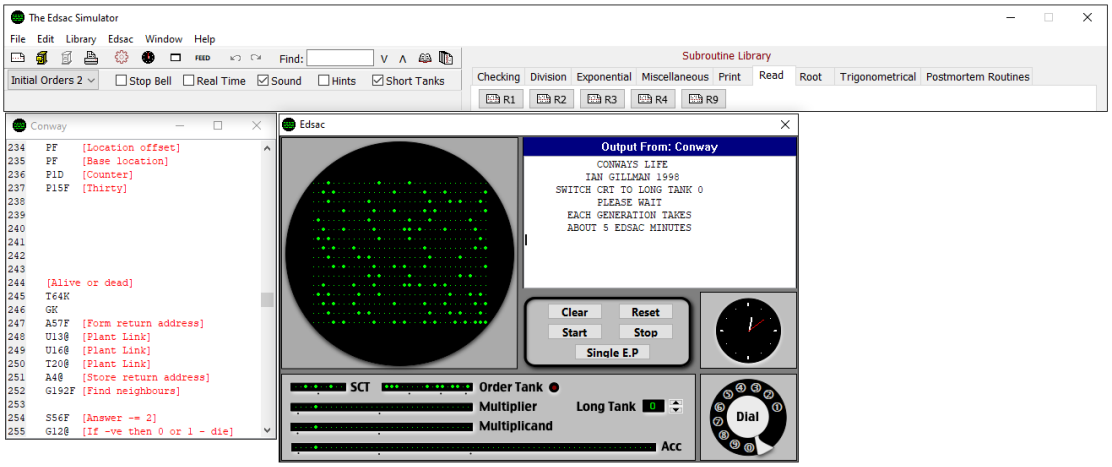


Figure 2.4.: EDSAC simulation

Martin Campbell-Kelly presented a simulation of the EDSAC computer in 1996, which is still available on the website of the University of Warwick [1, 2]. Much like the ENIAC Java applet, this program opens a window with a 2D representation of the machine's main interface. Additionally, a secondary window (at the top of Fig. 2.4) allows users to load EDSAC programs, of which several examples are prepackaged with the software, including Conway's Game of Life and OXO, one of the first video games. Also included in the files are several documents detailing the EDSAC itself and how this simulator is operated, though no explicit tutorial is provided within the program.

ENIAC-VR

A remarkably similar project, only found after Z3VR's completion, is ENIAC-VR [15], presented at the MuC 2020 (Fig. 2.5). It had the same goal of providing an immersive virtual environment for users to experience programming one of the first computers. This was extended further by including a guided tour which details the historical context and significance of the machine, as well as a *Maintenance mode*, in which users are instructed on how to spot and fix issues with the machine, which were a frequent occurrence on the real one.

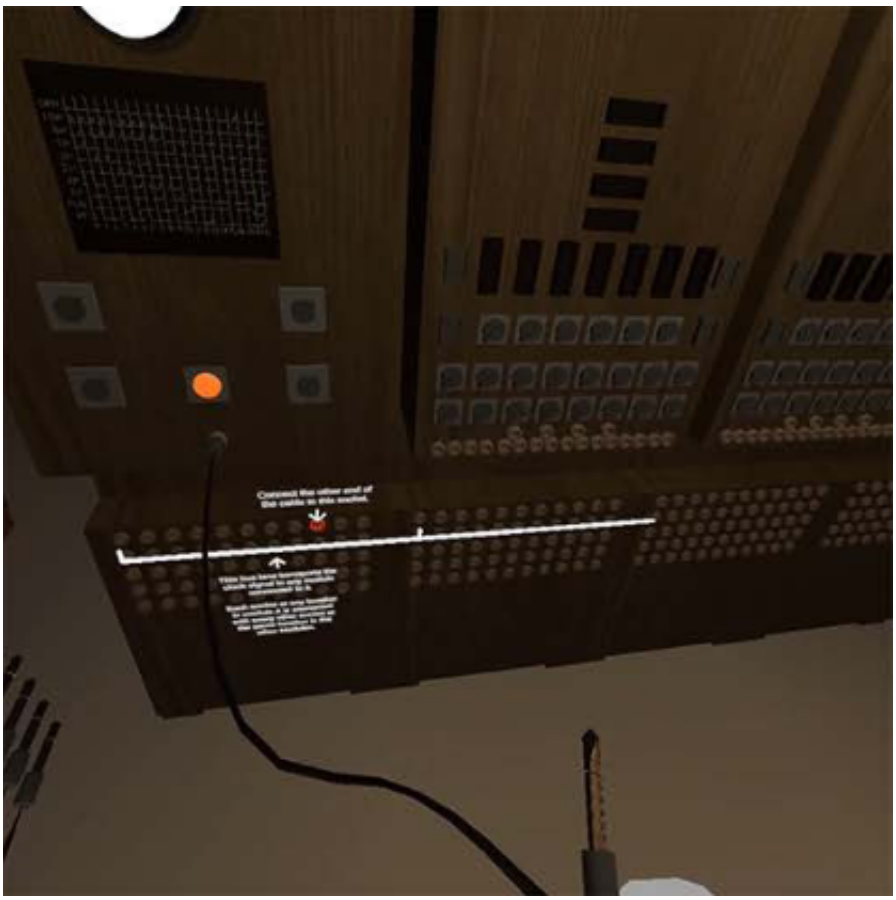


Figure 2.5.: Screenshot of the introduction to ENIAC-VR

It features an introduction quite similar to Z3VR, which guides users by explicitly telling them what actions to perform to arrive at their first program. Given the vastly higher number of interactable components, and from a modern standpoint unintuitive programming model, ENIAC-VR makes heavier use of text-based instructions (just about visible in Fig. 2.5), though it's difficult to imagine a purely visual or gesture-based tutorial that can convey this level of information density effectively. Just like Z3VR, ENIAC-VR was realised through the Unity3D game engine[1], though it runs on the Oculus Quest headset[2] natively rather than in a PC-hosted form like SteamVR[3] or older Oculus headsets with a cable [15].

[1] `https://unity.com/`, accessed 30.8.2023
[2] `https://www.oculus.com/`, accessed 30.8.2023
[3] `https://www.steamvr.com/`, accessed 30.8.2023

Unfortunately, no contact with the main developer could be established and the project is not publicly available, so a deeper comparison than this is not possible.

2.3. Other VR learning environments

Besides ENIAC-VR, there are also other VR applications with the intent to teach users about programming or other technical subjects, two of which will be highlighted here.

Zenva Sky

Figure 2.6.: A Program within Zenva Sky

Zenva Sky is a game available for Oculus VR devices intended to teach young users basic programming concepts [5]. The game revolves around the player being placed within a vehicle that can be moved around a tile-based virtual space using program instructions, similar to previous teaching applications like *Karel the Robot / Robot Karol*[4]. The user has buttons in front of them corresponding to each possible instruction, and pressing these adds them to an instruction list, not unlike the one featured in Z3VR (Fig. 2.6[5]). This starts out with simple instructions for moving forward and turning 90

[4] `https://xkarel.sourceforge.net/eng/`, accessed 5.9.2023
[5] Screenshots taken from `https://www.youtube.com/watch?v=hMqiroApt2I`, accessed 5.9.2023

degrees, and is later expanded upon by loops. The game consists of a series of levels with increasing difficulty, where the vehicle is used to reach an exit, later complicated by introducing locked doors and boolean inputs that the vehicle can activate, combined with logic gates to open said doors.

Figure 2.7.: Part of Zenva Sky's introduction

The project is a lot more gamified than Z3VR, it begins with a short scene where users are introduced to the game world and given an end goal of freeing an NPC, justifying why they must solve these puzzles. This intro scene and the later explanations of the interface and programming concepts are all done through text bubbles (Fig. 2.7), something which the development of Z3VR has shown to be far from optimal for effectively teaching users. The commentary of a playtester confirms this:

"I'm not sure if this has anything to do with learning programming. The things they're showing me in between they could have just given me in a PDF."[6]

During the first several levels, Zenva Sky places a lot more focus on introducing logic gates than making the user familiar with programming the movement system and tries to make the connection between the puzzle and what it would look like in code by just putting a screenshot of equivalent Python[7] code in front of the user after the

[6]Direct translation from German, `https://youtu.be/hMqiroApt2I?t=460`, accessed 5.9.2023
[7]`https://www.python.org/`, accessed 6.9.2023

level. A better approach to this might be to have this code somewhere within the game world already and dynamically show changes in its evaluation as the user activates the boolean inputs.

Network engineering in VR

Frezzo et al. presented an immersive VR application for Oculus Quest devices in which users could learn about various aspects of network engineering [4]. Different *widgets* within the application put the user in unique scenes, each tailored to communicate different concepts.

These widgets are:

- **Hardware** → The user is put in front of a server rack and has to connect different slots using ethernet cables. (Seen in Fig. 2.8)
- **Be-the-device** → The user is in a scene where they receive ethernet frames (visualized as cubes with different bits of information on each side) and has to perform the role of a switch, consulting a switching table and forwarding the message to the correct port.
- **Logical topology** → The user has to solve topological problems involving device icons and connections.

The back-end for this project is provided by Cisco Packet Tracer [8], an educational tool featuring similar *widgets*, though only as 2D representations. Aside from a graded single-user format, the paper also presents concepts for involving an instructor and other students through asymmetric VR as a form of remote teaching, as well as the possibility for a competitive mode where students in and outside of VR are split into teams [4].

[8] `https://www.netacad.com/courses/packet-tracer`, accessed 6.9.2023

Figure 2.8.: The "Hardware" widget

The interaction model of this project seems comparable to both Z3VR and ENIAC-VR, being quite physical with the user grabbing and manipulating objects as the main method of interaction. The plugging and unplugging of cables is especially similar to that seen in ENIAC-VR.

3. Recreating the Z3 in VR

One thing to note is that Z3VR recreates the Z3 that the *Zuse KG* rebuilt in 1960 for display in the Deutsches Museum, and not the *original* Z3, as it was destroyed during WWII, and no photographs have survived that the 3D model could be based on. Functionally the two machines are mostly identical, the only differences being that the original had twice the number of memory cells and a few more instructions for operations like doubling or halving a number or multiplying it by 10 or 0.1 [18], which can be emulated by the rebuilt machine with some extra steps (bar the larger memory). The following will cover how Z3VR was developed, starting with a brief overview of the object of interest.

3.1. Z3

The Z3 was a programmable calculating machine operating on binary floating point numbers, completed by the engineer Konrad Zuse in 1941 [11]. His prior machine Z1 worked entirely mechanically (the Z2 being the Z1 but with a relay-based processor), and was conceived as he was frustrated with statics equations which he didn't want to work out himself [11]. Functionally the Z1 and Z3 are mostly identical, as Zuse had developed an abstract design language that could be realised through both mechanical logic gates and electromechanical relays [17], with the latter being the case for the Z3.

Architecturally, these machines were quite similar to what is now called a *Von-Neumann architecture*, being split into a control unit, arithmetic unit, input/output unit, and memory [11]. Furthermore, the machine operates on floating point numbers which are remarkably similar to the modern float standard, and as such the arithmetic unit works in much the same way as modern ones, handling the exponents and mantissa separately, with binary adders for both and a shifting unit for the latter [11]. The Z3 also has encodings for the cases of 0 and ∞ and features automatic exception handling. If an operation like $0/0$ or $\infty \pm \infty$ is detected, the machine halts and shows an appropriate error on the console. The only things separating the Z3 from later computers are that the programs (including memory addresses) were static and stored on film stock with holes punched into it, and that conditional jumps were not supported. Raúl Rojas has previously shown that these features can be simulated using a *very* long program, though this method has no practical applications [10]. Loops can be implemented quite

Instruction	Console button	Meaning
Lu	↗	Input number
Ld	↘	Display content of R_1
Ls_1	+	Add
Ls_2	−	Subtract
Lm	×	Multiply
Li	:	Divide
Lw	√	Square root
Pr x		Read from address *x*
Ps x		Save to address *x*
	A	Start program

Table 3.1.: Instruction set of the Z3

simply by sticking both ends of the program film together once in the reader, though this means that loops always run over the entirety of the program. The implications of this will be explored in Chapter 4.

The operational model of the Z3 is effectively *Reverse Polish Notation*[1]: a mathematical operation requires first giving the operands and *then* the operator. The machine features two main registers which all operations execute on, and which can be thought of as a stack. These registers will be referred to as R_1 and R_2 throughout this paper. Operations that 'add' to the stack are *Input* (to prompt the user to input a number) and *Read address* (to read a number stored at the given memory address). Two-operand mathematic operators like *Add* or *Multiply* require both registers to be filled and reduce the stack to one filled register. *Output* (to show a number on the display) and *Save address* (to save a number at the given memory address) both expect exactly one register to be filled and clear the stack upon completion. *Square root* is a unique operation in that it requires one filled register but also leaves that register filled with the result.

Interaction with the machine happens almost exclusively through the console unit (Fig. 3.1). This box features a lamp matrix for displaying results and exceptions, and a keyboard which allows users to input decimal floating point numbers and manually execute mathematical operations without the need for a program. Table 3.1 shows all operations supported by the Z3 with their names given by Zuse and their respective console button and meaning [11]. *Input number* is technically separate from ↗ as when the former is executed in a program, the machine halts and waits until the user *presses* ↗ to confirm their input and translate it to binary.

Table 3.2 shows some example programs exhibiting most of the Z3's instructions.

[1] `https://en.wikipedia.org/wiki/Reverse_Polish_notation`, accessed 1.9.2023

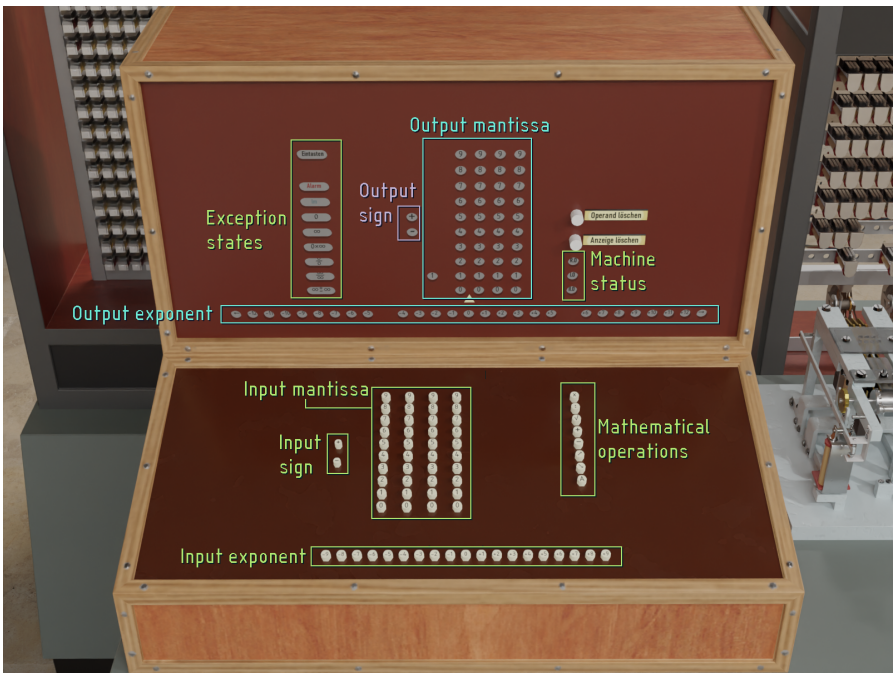


Figure 3.1.: Z3 Console

3.2. Implementation

The project was implemented within the Unity3D game engine[2], which provides an OpenXR[3] API allowing the application to be used with virtually any VR hardware. Behaviours of objects within the engine are primarily governed by C# scripts. In Z3VR, each of the machine's separate units and peripherals has one such script attached to emulate its function, with references to others allowing for communication where necessary. Some more fine-grained separations were made to decrease code coupling and improve maintainability. For instance, each digit of the mantissa input (vertical columns of buttons in Fig. 3.1) has its own script to manage the 10 buttons it's composed of (each of which also has a universal push-button script attached), to ensure that only one of them is pressed at a time.

The arithmetic unit is simulated on an algorithmic level. This means that the sets of relays making up the mantissa, exponent, and sign of registers are abstracted as integers and a boolean respectively, however, all the algorithmic steps that the real machine takes to work out results are replicated. Some temporary registers were

[2] `https://unity.com/`, accessed 30.8.2023
[3] `https://www.khronos.org/openxr/`, accessed 30.8.2023

Program 1	**Notes**	**Program 2**	**Notes**
Lu A	Input A	*Lu A*	Input A
Lu B	Input B	*Ps* 1	Save A at address 1
Ls_1	$A + B$	*Pr* 1	Read A
Ld	Display result	*Lu B*	Input B
		Ls_1	$A + B$
		Ps 2	Save $A + B$ at address 2
		Pr 1	Read A
		Lu C	Input C
		Ls_2	$A - C$
		Pr 2	Read $A + B$
		Lm	$(A - C) * (A + B)$
		Lw	$\sqrt{(A - C) * (A + B)}$
		Ld	Display result

Table 3.2.: Example programs for $A + B$ and $\sqrt{(A + B) * (A - C)}$

omitted (namely C and D, which only serve as intermediaries to make the addition of R_1 and R_2 possible [11]). The algorithms are executed using C# coroutines, which allows for simulating the speed (or lack thereof) of the machine by timing out the coroutine for the duration of a real cycle after each simulated one.

Some technicalities are unaccounted for however, mostly due to time constraints and because users who are not extremely familiar with the intricacies of how the Z3 is built and operates will never notice them. One of these is the fact that different registers have a different number of bits for their mantissa, making some more accurate than others [11]. In Z3VR all mantissas are assumed to have 14 Bits of accuracy and are represented by 32 Bit integers, where only the lowest 15 Bits ($14 + 1$ for the implicit 2^0 Bit) are actually used.

Another detail is that all operations are treated as equal in terms of when they start execution. The Z3's cycles consist of various steps, one of which is specifically reserved for writing a result to memory. For instance, a "write" operation following an "add" operation should already be executed within the end of the "add" operation's cycle [11]. As such, the simulated Z3 is technically slower than the real machine in these specific instances, though by a virtually unnoticeable amount.

A further aspect is the behaviour of the machine in various edge cases, which is largely undocumented and would require an extensive deep dive into its electrical plans. Especially questions regarding the handling of invalid instruction sequences, like "What happens if the machine encounters a third input instruction, when both

registers are already filled?" or "Is a write or output operation valid if both registers or neither are filled?" arose during development. Z3VR implements a *watchdog*, which monitors the current state of both registers and the upcoming instruction, and throws a generic exception if an invalid sequence is detected. This system has shown to be an invaluable aid for new users, as it prevents them from accidentally putting the machine in an invalid state. Such a system only requires a check of how many registers are currently filled and how many would need to be filled for the next operation, and as such is feasible to have been implemented on the real machine.

Another safeguard in Z3VR that likely wasn't part of the real machine, however, is that the machine puts itself in a valid state when a new program is inserted, by clearing both operands. If an exception occurs or the user actively interrupts a program by turning off the reader or yanking out the program, there will likely be leftover values in one or both registers. If these remain when a new program starts it's almost guaranteed to trigger the watchdog and throw another exception. A user who has not yet learned to watch the register state indicators on the console and clear the registers by hand when necessary would be thrown off by this, and a lengthy error message explaining exactly what has happened would be counterproductive. Simply clearing the registers when inserting a new program is the simplest way to ensure proper operation, even if it's not quite accurate to the real machine.

3.3. Interface

A simulated Z3 is all well and good unless you can't operate it. The VR interface places the user in the virtual space and allows them to move freely and interact with the machine intuitively and physically. These mechanics work to give users more intuition and knowledge of this machine than a text description ever could. Furthermore, users are actively involved in handling the machine and running programs. They don't just click a button and watch the application *do stuff*, they engage with the process and do things themselves to understand how they work.

The Z3 was mainly programmed on paper. "Calculating plans" (Rechenpläne) were written by hand, with tables to keep track of which variables and constants are in which memory cell, and annotations describing what number inputs were expected at which time. These plans were then made physical by punching holes corresponding to the bitcode of each instruction in 35mm film stock, though notes about the program had to be included to tell the operator how to use it properly. Such notes were especially important if a calculation required the sequenced execution of several programs in a specific order.

Handing a person pen and paper in VR is a recipe for disaster, and letting them type

everything on a keyboard would turn anyone away within seconds. As such Z3VR provides a number of interfaces and interactable elements to make the programming process as convenient as feasible.

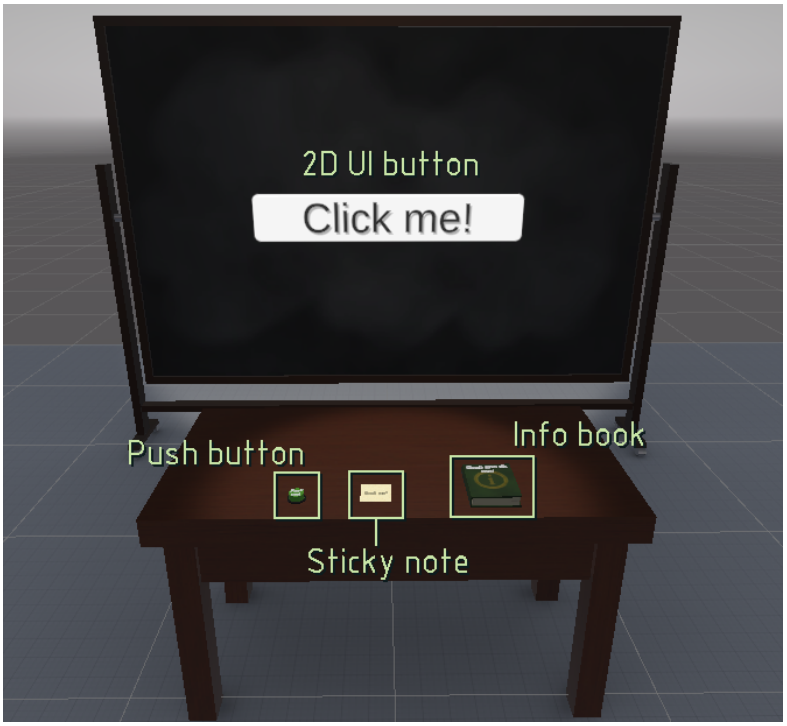


Figure 3.2.: Introduction of interaction types

These systems are introduced step by step, so as not to overwhelm users with everything at once. Upon startup, users are placed in a bare scene with nothing but an empty table and a text bubble in front of them, instructing them how to move and rotate in the virtual world using the controllers, and how to open the options menu. After that, four objects appear one after the other in the scene once the user interacts with the previous one, introducing each of Z3VR's modes of interaction. These objects are, in order: a push button, a sticky note, a book, and a blackboard bearing a 2D UI button (Visible in fig. 3.2).

Interactable objects in Z3VR can be categorized as such:

- **Push buttons** → Physically based buttons that incrementally depress as you move your virtual finger into them. Once a threshold is reached, they snap to a fully depressed position and trigger whatever they're linked to. Examples include the console keyboard and the lever buttons of the film puncher.

- **Grabbables** → Objects that can be picked up and carried around. They can be let go in mid-air without falling or put into specialised *deposits*, which can have various functions. Examples include sticky notes and program films.
- **Openables** → Grabbables that can be opened by tilting them such that their spine points downwards. These include the info book and program folders.
- **2D UI buttons** → Standard Unity UI buttons on world space canvases. A raycast visualised by a laser pointer attached to each hand acts as the user's cursor. The laser is only visible when aiming at such a canvas.

Interactions (other than ones by raycast) produce a small haptic feedback in the respective controller, to further enhance the immersion and responsiveness of the interface.

Some notes on the interabable objects:

The most common items that users will be interacting with are the program films. They're grabbables that open a display of the program held within when you hold your hand near them, shown in fig. 3.3. This display also features two buttons with which you can toggle to a plain text view (eye icon) and whether the program should loop or not (∞). The plain text view translates the instruction code names given by Zuse to a literal name describing what it does, for instance, *Ls1* becomes *Add*.

The book, referred to as the *Info Book*, is present in the main scene as well and provides extra information and trivia about various things, including individual sections of the machine and background info about how and why film strips were used for programming. Once opened, the user can point at a section of the machine marked with a square, or hold it near an object emitting particles with the same *i* logo as on the book, and the text within will change to said information. If the text exceeds the two pages, arrows next to the page numbers below will indicate that there are more pages, and the user can simply grab the corresponding edge to switch to that page.

Sticky notes were added as a simple and intuitive way to both name programs and metaprograms for saving, and to define the expected inputs of a program. A typewriter-style keyboard is provided to allow users to write anything on them. The program display that opens when you hold a punched film has an outlined box at the top, and placing a sticky note there will let you save the program under that name. The display also bears a list of these boxes on the right-hand side, titled "Inputs" (Fig. 3.3). This list has the same number of boxes as there are input instructions in the program. If the machine runs a program that has at least one expected input defined, it will show an outline of the info book on the console, and placing it there will show the list of expected input values within it, including an arrow indicating which input is currently requested.

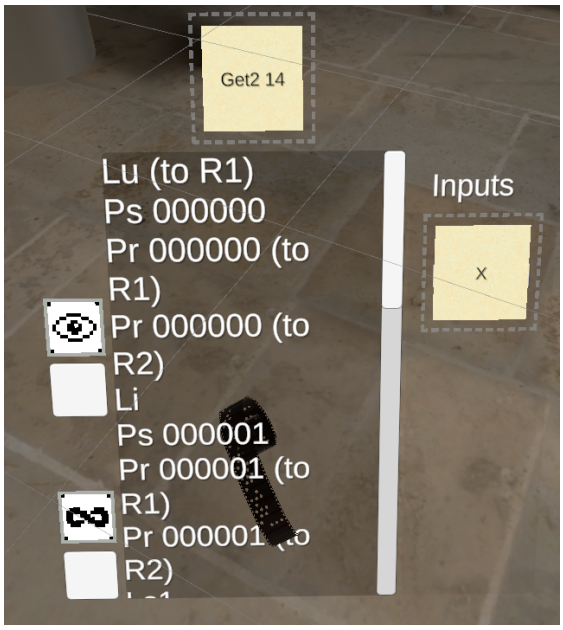


Figure 3.3.: Instruction list and sticky notes for name and input

The Z3 does not have an instruction for immediate values, as such the only way to obtain constants is through user input, which of course can be *anything*. Furthermore, the machine has no way of communicating which variables should be input in what order, which might be arbitrary depending on the program. The sticky notes were implemented to emulate the hand-written notes on a real program while being as simple to grasp and use as possible. However, this system alone is not sufficient to communicate sequences of programs to the operator, which is where program folders come in.

Program folders are used to store program films within them in a defined sequence. They open like the info book and offer a clock-like arrangement of boxes where programs can be put, shown in figure 3.4. The number and positioning of these slots adjust automatically depending on how many programs are contained. The folders themselves can be named and saved as well, to make for more compact entries in the saved program list. The name *Metaprogram* is introduced to describe such sequences of programs. This system works well for most applications, though there are edge cases where this may not suffice. One can imagine a case where a value is approximated by a process that executes one loop until a stop condition is reached, and then another, alternating between the two repeatedly. Though given the skill level necessary to design such metaprograms, no work was put into designing a system that allows for adding further operation notes to folders.

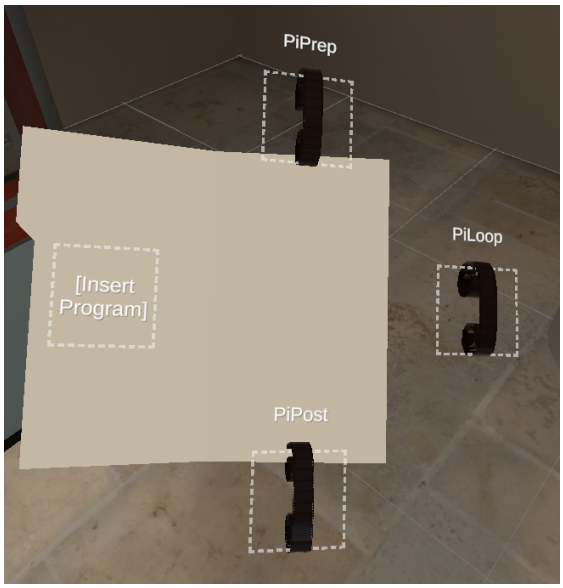


Figure 3.4.: Program folder containing three programs

Once the user has completed the interaction introduction, they're transported to the main scene, where the Z3 is situated. The room is vaguely inspired by the room in the museum where the real machine used to be, though this one is far smaller, ensuring the user can't stray too far from the machine. At this point, the console does not feature any of the operation buttons, and the only option users have is to start selecting a number using the mantissa and exponent buttons. Once a number has been selected, a small tutorial will start, guiding the user through their first calculation of $A + B$. There is no text telling them what to do, instead translucent hands indicate the action to be performed, in this case mainly pressing the operation buttons (Fig. 3.5). The next step doesn't start until the previous one has been performed, leaving users to comply at their own pace. The only text added to the console is large numbers which reflect the currently selected number input and output number in the format $+1234 * 10^5$. This was added since some people have difficulty making out the button labels, not helped by the lacking resolution of the headset used during playtests. These help texts can be disabled through the options. An extra aid in this regard is the option to double the size of the keyboard, which users are reminded of by further text on the console which disappears after their first number input.

This introduction system, internally titled *Step Tutorials*, has been implemented as a simple state machine, where each state defines which scene objects should be enabled and disabled once it is reached, and which other states can be reached with which

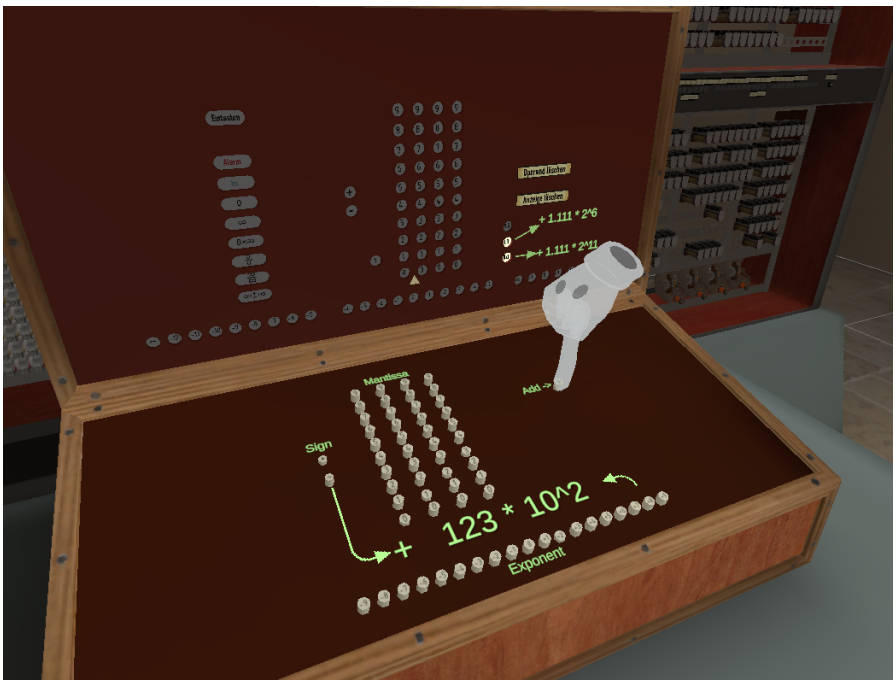


Figure 3.5.: Console introduction. A hand is shown telling the user to press the *Add* button.

inputs. Inputs can come through push buttons or any class implementing a *Trigger* interface, and each input object passes a unique identifier to allow distinguishing between them.

Once the simple console introduction has been completed, all the operation buttons reappear, and the user is shown towards the next scene element. This is telegraphed through a green arrow under the player, pointing towards the next tutorial which only appears once the user has reached it. To facilitate the writing of programs, various interactable "stations" have been placed in the scene, each with a specific function. The following will outline these elements in the order they're introduced to the user.

Programming board

The programming board is the main utility for writing programs. A scroll view in the middle shows the current list of instructions, and the left-hand side features buttons for each of the available operations. Clicking on one of these operations will add it to the instruction list where the cursor is (indicated by <, can be moved by clicking on list entries). On the right are the same buttons as on punched film displays for toggling the

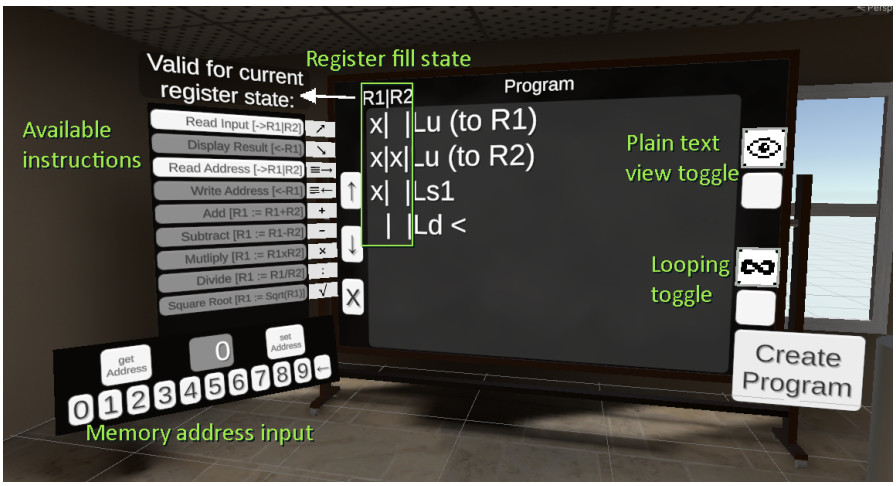


Figure 3.6.: Programming board

plain text view and program looping, as well as one for instantiating a punched film object with the current program on it.

As you can see in figure 3.6, some operations are greyed out. The *watchdog* used for ensuring that the machine can't enter an invalid state during manual operation is also used here to determine which operations are valid to put at the cursor and only lets users insert those. The columns for $R1$ and $R2$ show the user which registers are filled after each instruction. Should an invalid sequence still be created, whether by inserting or deleting an instruction in the middle of the program, the invalid instruction will be marked red.

Program eraser

This is a shredder which can be used to dispose of wrong or unneeded programs, to keep things tidy.

Film puncher

The film puncher is an accurate rendition of the device used for creating programs for the real machine and can be used to punch films by hand in Z3VR as well, to get the full experience of programming the Z3. Users press the lever buttons corresponding to the 1-bits of the current instruction code, then click the wheel on the right-hand side to advance to the next instruction (Fig. 3.7). A list to the left of the puncher shows what instructions have been punched, so users can see if they've made a mistake right away. Once done they can pull out the film from the far end. There is no apparent source

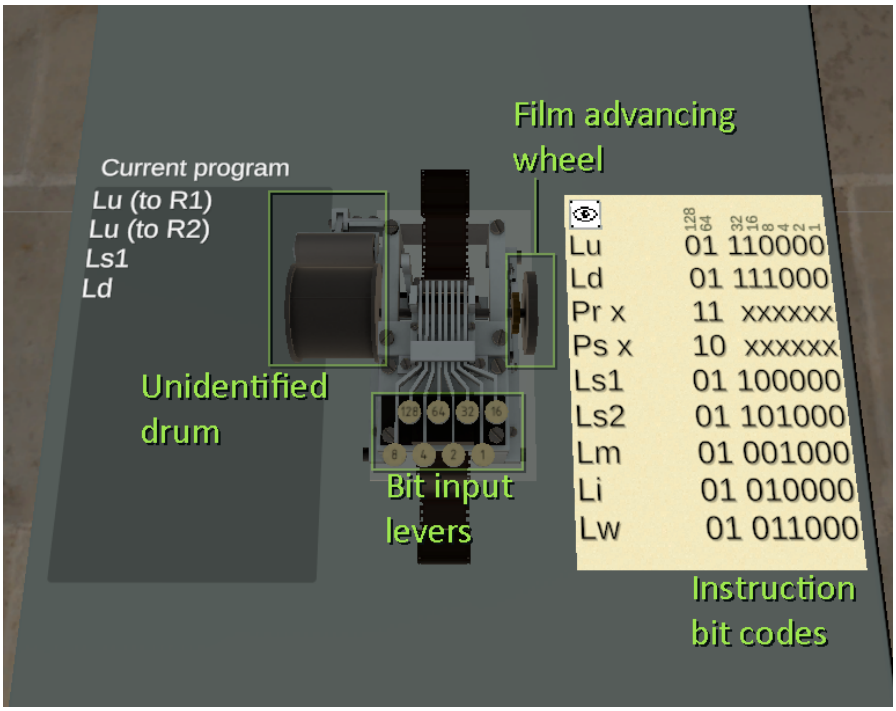


Figure 3.7.: Film puncher

on what the large drum on the left-hand side does, the current working theory is that it was used to flatten sticky tape that was used to cover up erroneous holes, though no such system has been implemented in Z3VR. The reconstruction of the Z1 in the late 1980s (also by Zuse himself) features a puncher of the exact same design, however without this specific part, further throwing its function into question[4].

Program saving

This table features an unlimited supply of sticky notes and the keyboard used to label them. Labelling a sticky note requires it to be placed on a deposit just above the keyboard, shown in figure 3.8. An unlimited stack of folders is also introduced later. Labelled programs and metaprograms can be saved by placing them in the nearby filing cabinet. Attempting to save an unnamed program makes a short text appear above, informing the user that this is not possible.

[4] `https://youtu.be/_K7Z0afc4wk?t=113`, accessed 21.8.2023

Figure 3.8.: Table featuring folders, sticky notes, a keyboard, and a filing cabinet

Program loading

The loading interface consists of a blackboard bearing a list of buttons, each labelled with the name of a program that has been saved on disk. Clicking an entry instantiates a punched film or folder containing the selected program or metaprogram on a plinth next to the blackboard.

3.4. Visuals

Despite the reconstructed Z3 (hereafter referred to as just "Z3" for simplicity) having been on display in the Deutsches Museum for well over 50 years, the number of publicly available photos of it on the internet is shockingly limited. The author's attempt at contacting the Museum directly for measurements and better pictures was ignored, as such the 3D model is based entirely on these public images.

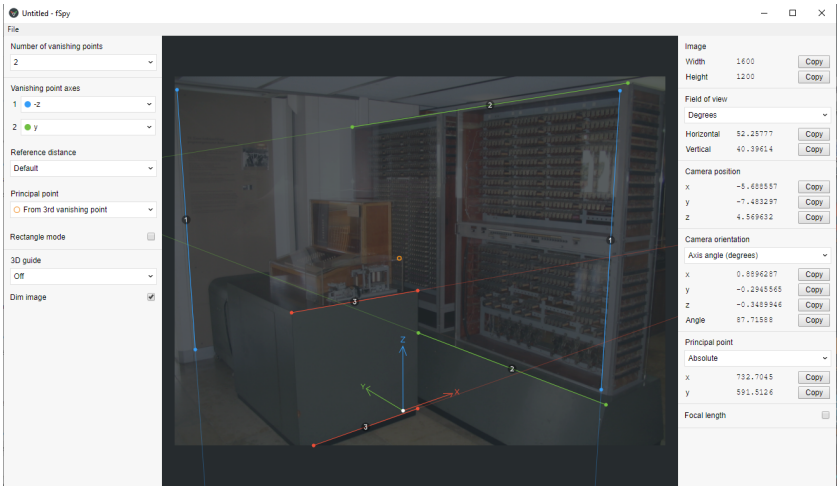
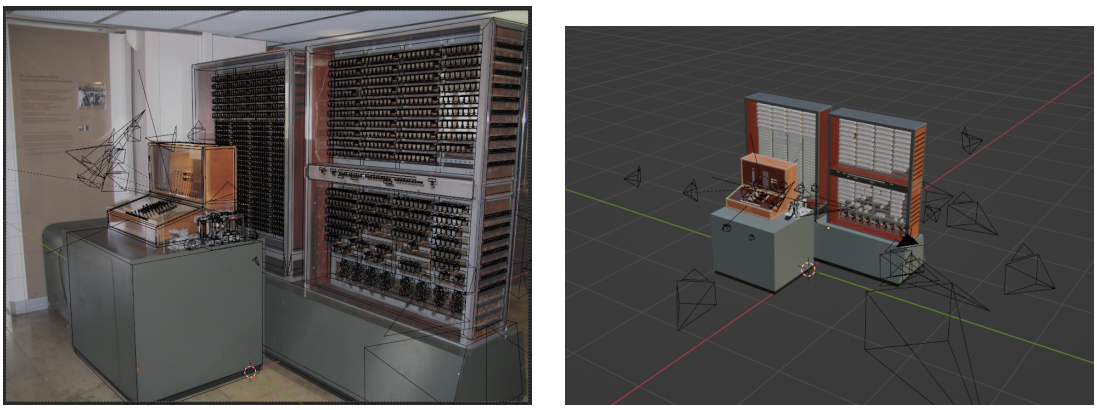


Figure 3.9.: The fSpy interface



(a) View from one camera, 3D model visible as wireframe overlay

(b) Scene view with all reference cameras

Figure 3.10.: 3D recreation process

The main method used to make the 3D model was using a program called *fSpy*[5], pictured in figure 3.9, which allows users to import a photo and, by marking a few orthogonal lines within it, extrapolate the position and orientation of the camera that took it. This data can then be imported into 3D modelling programs like *Blender*[6], where a virtual camera is placed in the scene, and the photo is set as a transparent background for that camera's view (Fig. 3.10a). Once a few of these views have been imported (Fig. 3.10b), it's possible to approximate the basic shapes of the machine and to iterate on these over time to approach an accurate model. Something that *fSpy* cannot extrapolate, however, is the scale. Fortunately, the assumption that the film strips used for programming the machine were the standard 35mm (later confirmed by plans of the reader) and a plan that revealed the height of the arithmetic unit and memory racks to be 2 Meters [18], were enough to make an educated guess at the dimensions of the machine.

There are several shortcomings in terms of the accuracy of the model. As mentioned the number of reference photos was relatively limited, and the number of those that were usable for the stated method was even lower, as *fSpy* can't be applied to images with even slight distortion, which can occur when pictures get squashed or stretched for display on a website, or if the initial photo was taken using a fisheye lens. With the amount of references being so small there are also parts of the machine where there are simply not enough pictures available to be able to reconstruct them accurately. Examples of this would be the debug bar at the front of the arithmetic unit (Fig. 3.11),

[5] `https://fspy.io/`, accessed 21.8.2023

[6] `https://www.blender.org/`, accessed 21.8.2023

and the pulse drum around the back of the machine's base (Fig. 3.12). The latter features a lot of visual complexity, which is nearly impossible to recreate without a sufficient number of different angles available, and while the debug bar can be seen in most photos, none of them is close enough or has a resolution high enough to make out finer details, like the content of the four labels around it, or the shape of the buttons on either side. Combing through the electrical plans of the machine allowed for deducing its function, and making an educated guess for the labels and buttons possible, though confirmation of these will remain impossible until the real machine is accessible again.

Figure 3.11.: A low-resolution view of the debug bar[7]

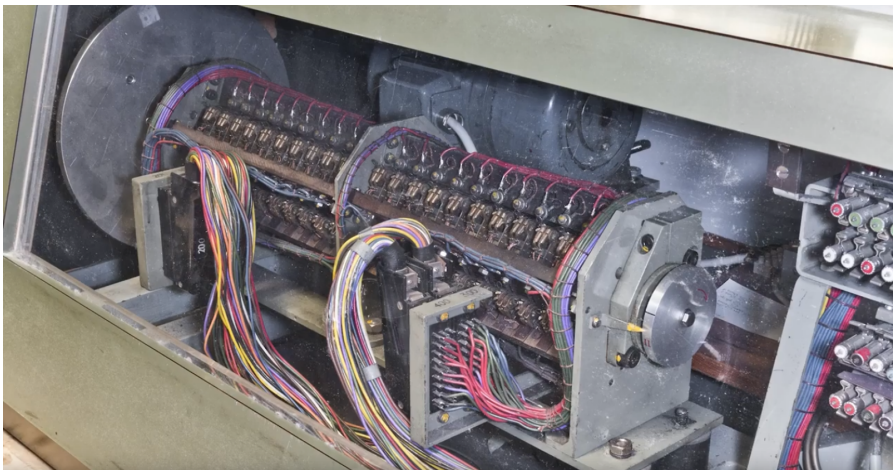


Figure 3.12.: The only clear picture of the pulse drum[8]

[7] `https://www.youtube.com/watch?v=5TNUebJ-BmU`, accessed 29.8.2023
[8] `https://www.youtube.com/watch?v=TbW-qNxD11E`, accessed 29.8.2023

A notable detail of Z3VR is that the punched films that store programs have procedurally generated textures to reflect the program within them. A base texture of ordinary film stock is combined with 8 masks for the alpha channel (one for each bit) which are either applied or not depending on the bits of the given instruction's code. The resulting texture for a lengthy program is shown in figure 3.13. When a program is run through the reader, an offset is applied to the material every time the machine reads an instruction, moving the whole film forward by the length of one instruction, and further emphasizing the fact that the holes encode the program to the user.

Figure 3.13.: A punched film, encoding a program to turn a non-zero input into 2^{14}

Some shortcuts were taken in this system for readability and timely implementation. The hole pattern on the real machine is interlaced, with bits 1, 2, 4, and 8 of one instruction coming after bits 16, 32, 64, and 128 of the next one [18]. This makes it very difficult to tell different instructions apart and would hinder the users' association between the bits they punched and the holes they see in the final film. Instead, the holes of an instruction code in Z3VR are packed together, effectively leaving a one-step gap between instructions. Running such a film through the real machine would work just fine, as an all-zero instruction equates to "no-op" and is simply skipped[9]. Another shortcut taken was using hand-made meshes for the handheld and placed film strips. This limits the length of the visual program, while the number of stored instructions is virtually unlimited, and further introduces the issue of the looping film mesh on the reader not reflecting the actual length of the program. For non-looping films, this was mostly hidden using a coiling start and end. Fixing this would require implementing a dynamic mesh generation system, which would have blown the workload for this way out of proportion with the effect of the result.

[9] `https://youtu.be/Ct2WHmu6OJ4?t=439`, accessed 21.8.2023

3.5. Audio

A critical part of creating a believable VR space is sound. Aside from greatly amplifying the immersion [14], it also provides feedback to the user that *things are happening*. This ranges from the immediate response of a clicking noise when a button on the console is pressed, to the symphony of relays clacking away while the machine is crunching numbers. Two videos exist on the YouTube channel of the Deutsches Museum[10] which show the Z3 working, though only one of them could be used for sampling audio as it has little to no background music[11]. During operation, a short loop of the relays working is played until the machine either finishes running a program or until it halts for requesting a number input or displaying a value. To break up this loop and give at least some indication of which instruction is being run, an additional single-shot relay sound is played during memory access operations, implying the activation of the address decoder. The two audio sources for the relay loop and memory access sound are placed within their respective relay cabinets, and spatial audio helps users tell where each of them is coming from. When running a program, an additional loop of the pulse drum spinning is played while the machine waits, and is stopped when a program ends. Technically, this loop should be playing permanently as there is no way of turning off the machine entirely in Z3VR, but this would likely be annoying to users after a while and lead to confusion about whether the machine is doing something or not, so it was used to telegraph that a program is running instead.

Further sounds were added to the console buttons and puncher levers to indicate when they've triggered, and custom foley was recorded to give life to the info book.

[10] `https://www.youtube.com/@DeutschesMuseum`, accessed 21.8.2023
[11] `https://www.youtube.com/watch?v=aUXnhVrT4CI`, accessed 21.8.2023

4. Findings from working with the Z3

Some aspects of working with the Z3 are hardly explored in literature and only become evident once one has written a few more advanced programs and familiarised themselves with things to look out for and consider. These shall be outlined in the following.

4.1. Metaprograms

The most glaring one of these is the reason for the program folder system. As programming loops is only possible by taping both ends of the punched films together, looping programs almost always necessitate a pre- and sometimes post-processing program, to input variables and constants, and accumulate and output results respectively. Such metaprograms can consist of multiple loops as well, as will be shown in an upcoming example, and their potential complexity can be vast. They're also quite useful if there are specific programs which either act as libraries to be used in other metaprograms or if they produce a specific constant and only need to run once, placing their result in a predetermined memory cell for other programs to access.

4.2. Flooring

There is no built-in instruction for reducing a number to its integer component. However, due to the limited accuracy of floating point numbers, an arbitrary amount $n \in [0, 14]$ of fractional digits can be eliminated by adding and subtracting 2^n, with $n = 14$ flooring the number to an integer. This can then also be used to implement *modulo*, another unsupported operation. Getting the number 2^{14} in the first place without relying on the operator to input a sufficiently large number requires starting from 1 by dividing an input number by itself (though this still breaks if the user inputs 0), and growing it through additions and multiplications. Recalculating this from scratch in every program would be quite inefficient, instead running a single program once to calculate it and place it in memory would be advised. The shortest number of instructions (starting with 1) for this found during development is 20 (Table 4.1), though this introduces the question of optimization. Different operations take

a different number of cycles to execute. Addition and multiplication take 3 and 16 respectively [11], which is a substantial amount of time at the machine's 5 cycles per second. Adding a number onto itself instead of multiplying it by 2 is always the quicker option, though when multiplying with higher powers of 2 this becomes more complicated and would warrant further investigation which is out of scope for this paper.

Instruction	**Notes**
Lu X	Any non-zero number
Ps 0	
Pr 0	
Pr 0	
Li	$\rightarrow 1$
Ps 1	
Pr 1	(Starting with 1)
Pr 1	
Ls_1	$\rightarrow 2$
Ps 2	
Pr 2	
Pr 2	
Ls_1	$\rightarrow 2^2$
Ps 4	
Pr 4	
Pr 4	
Lm	$\rightarrow 2^4$
Pr 4	
Lm	$\rightarrow 2^6$
Pr 2	
Lm	$\rightarrow 2^7$
Ps 14	
Pr 14	
Pr 14	
Lm	$\rightarrow 2^{14}$
Ps 14	

Table 4.1.: Program for obtaining 2^{14}. See table 4.2 for instruction code meanings

4.3. Stopping loops

The reading unit of the Z3 has a small switch that can be used to stop a program at any moment, but unless the operator has quick reflexes and is watching the reading levers intently, they will not know what the machine is currently doing and when would be the right time to abort a loop. There are two strategies to get around this. For one, the program could include a display operation which stops the machine and shows the current state of a variable. Examples of such a variable would be a simple countdown, where the operator would stop the machine once it reached 0, or, if the loop is used to approximate a value, the difference between the previous and current result, where one stops the iteration once a sufficiently small difference is achieved. The drawback of this is that every run of the loop requires the operator to clear the display by hand and to make sure to stop it at the right moment.

A more sophisticated and automated solution to this is the use of exceptions. Among other cases, dividing zero by zero is detected by a special circuit which halts the machine and displays an appropriate exception on the console. Stopping the loop once a countdown reaches zero is trivial with this method, achieved by dividing the counter variable by zero and discarding the result afterwards. Stopping when a difference is small enough can be implemented similarly, but by adding and subtracting a large number 2^n before the division to eliminate some digits and floor the number to zero once it's smaller than 2^{-14+n}.

4.4. Example program

The program shown in table 4.2 is used to approximate Pi using the Leibniz formula[1]. It's split into 4 separate programs, namely an input handler, preprocessing loop, main loop, and accumulation and output handler. The program starts with a single input, which will determine the desired accuracy of the approximation. The first loop then iterates over the given accuracy variable to produce the value $2^{accuracy}$, which is then used in the second loop to check whether the difference of the last two results is smaller than $2^{-14+accuracy}$. The final step multiplies the current result by 4 and displays it.

[1] `https://en.wikipedia.org/wiki/Leibniz_formula_for_%CF%80`, accessed 8.9.2023

Prep	**PrepLoop**	**MainLoop**	**Post**	**Addresses of variables**
Lu Accuracy	*Pr* 0	*Pr* 4	*Pr* 2	0 → Input, counter
Ps 0	*Pr* 7	*Ps* 6	*Pr* 2	1 → 1
Pr 0	*Li*	*Pr* 1	Ls_1	2 → 2
Pr 0	*Ps* 31	*Pr* 3	*Pr* 4	3 → Odd dividend
Li	*Pr* 8	*Li*	*Lm*	4 → Current result
Ps 1	*Pr* 8	*Ps* 5	*Ld*	5 → Temporary value
Pr 1	Ls_1	*Pr* 4		6 → Previous result
Pr 1	*Ps* 8	*Pr* 5		7 → 0
Ls_1	*Pr* 0	Ls_2		8 → Accuracy addend
Ps 2	*Pr* 1	*Ps* 4		31 → Dump
Pr 1	Ls_2	*Pr* 3		
Pr 2	*Ps* 0	*Pr* 2		**Instruction reference**
Ls_1		Ls_1		*Lu* [Value] → Read console input
Ps 3		*Ps* 3		*Ld* → Display content of R_1
Pr 1		*Pr* 1		Ls_1 → Add
Ps 4		*Pr* 3		Ls_2 → Subtract
Pr 2		*Li*		*Lm* → Multiply
Ps 8		*Ps* 5		*Li* → Divide
		Pr 4		*Pr x* → Read from address *x*
		Pr 5		*Ps x* → Save to address *x*
		Ls_1		
		Ps 4		
		Pr 3		
		Pr 2		
		Ls_1		
		Ps 3		
		Pr 6		
		Pr 4		
		Ls_2		
		Pr 8		
		Ls_1		
		Pr 8		
		Ls_2		
		Pr 7		
		Li		
		Ps 31		

Table 4.2.: Example program for approximating Pi

5. Evaluation

User tests played a critical role in the development of the VR interface and especially the introduction system. Participants were given a short info sheet (Appendix A.1) to explain that the gathered data would be handled anonymously and that the test could be aborted at any time. After each test, they were handed a questionnaire (Appendix A.2) to fill out, consisting of the *Standard Usability Scale*[1] questions as well as some supplementary entries that were more project-specific. The project-specific questions include how immersed the participants felt, how clear the introduction was, whether they would revisit the application in their spare time, their prior experience with low-level programming, and some relating to what sort of programming challenge they underwent and how they felt about it. The first three of these questions were intended to evaluate the project in various aspects and analyze how the sentiment of users evolved throughout development, however, they were not effective in this regard, as will be discussed later.

Before the first tests, the plan was to limit the time each participant had to 5 minutes, hence the related bullet point in the questionnaire, but it quickly became evident that such a time limit was not realistic and it was dropped during testing.

Tests were performed by letting the introduction system guide testees at first. Sometimes explicit instructions by the test conductor were necessary to help them along, but the amount of required intervention sunk drastically with the second introduction version and throughout its further development. Once testees had completed the introduction, they were asked if they would like to take a programming challenge, and if so at what difficulty level. Tests were concluded either if they opted out of the challenge or if they attempted and completed it (or gave up on it in rare cases). Testees were given the option to continue if they wished, though hardly any took advantage of this. Tests generally lasted between 20 and 30 minutes, with some outliers lasting 40 minutes or even up to almost 2 hours.

Programming challenges were split into three difficulty levels:

- Easy $\rightarrow$ Could be solved by only working within the two registers directly. Examples include $(A + B) * C$ and the average of two numbers (with 2 being provided as a third number input)

[1] https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html

- Medium → Involved memory accesses. Examples include $(A + B) * (A + C)$, $A + 1$, and $A * 2$, with only A being allowed as an input for the latter two.
- Hard → Also required a loop and pre-/post-processing programs. Examples include computing Fibonacci numbers and the average of an arbitrary amount of numbers.

In terms of the demographic, testees consisted mostly of informatics students between the ages of 20 and 25, as they were the easiest to contact and the GamesLab within the TUM Mathematics and Informatics building already provides VR equipment which could be used for testing.

Standard Usability Scale

The *Standard Usability Scale* (SUS hereafter) used during testing is a standardized set of 10 questions developed by the *Digital Equipment Corporation* in 1986 for evaluating the usability of system interfaces. Since then it has become an industry standard, being used virtually anywhere where applicable, making for an adequate metric to compare different systems. Evaluation of the responses produces scores in the range of 0-100, and a comparison of hundreds of studies has shown the average score to be 68.[2]

5.1. Introduction version 1

The first introduction system was quickly thrown together just before the first round of playtests. It consisted of a series of text bubbles that showed users around the different elements in the room one at a time, which at this point in development were all visible from the start. Needless to say, the combination of being overwhelmed by the number of elements introduced at once, as well as the walls of text users were presented with to explain how they worked, meant that the vast majority of testees were quite lost and had to be given explicit instructions to proceed.

Gathered data

The first round of tests only accumulated a sample size of 7, so the discrete numbers should be taken with a large grain of salt. Nevertheless, they reflect the perceived performance of the first introduction system, with an average SUS score of 52.86, way below the general SUS average of 68. The scores are also extremely varied, ranging

[2] `https://measuringu.com/sus/`

Question	Average answer (0-4)
Immersion	3.43
Clear introduction	2.43
Would revisit Z3VR	3.43
Prior experience[3]	3.14

Table 5.1.: Project-specific answers (Version 1, $n = 7$)

between 25 and 72.5 and with a standard deviation of 18.1, with higher scores possibly being a result of politeness rather than honesty.

The data obtained through the project-specific questions at the bottom of the questionnaire is virtually useless at 7 participants, which is even more evident once the respective values for version 2 are considered, but shall be listed here for completeness' sake (Table 5.1). Only one person actually attempted a challenge, so plotting this datum would yield no further insight. They took an "easy" challenge and didn't find it to be harder or easier than advertised.

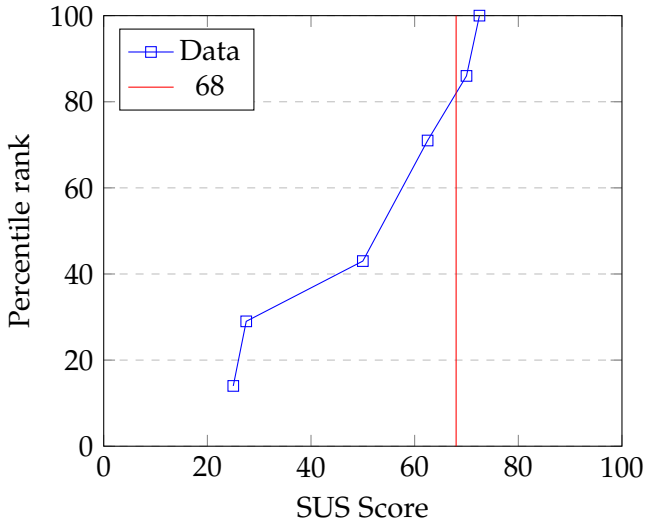


Figure 5.1.: Percentile distribution of SUS scores (Version 1, $n = 7$)

[3]Prior experience with low-level programming

Question	Average answer (0-4)	Standard deviation
Immersion	3.27	0.57
Clear introduction	2.87	0.88
Would revisit Z3VR	3.20	0.75
Prior experience[4]	2.93	1.34

Table 5.2.: Project-specific answers (Version 2, $n = 15$)

5.2. Introduction version 2

After the disastrous failure of the first introduction system, work began on designing a new approach based entirely on visual indicators and the iterative introduction of elements, with as little actual text as possible. This system is described in more detail in Chapter 3. The amount of external guidance required during these tests was drastically lower and kept decreasing as development continued and the introduction was further refined. There are a few parts where the visual guidance was insufficient and testees consistently required a quick explanation, so text had to be added to these areas. Specifically, these are the text on the console to remind users that they have the option to make the buttons bigger, a note to explain that the lit "Eintasten" field on the console display means that a number input is requested, and some text to explain that the numbers on the film puncher are buttons corresponding to instruction code bits. Though even still, while testees got along fine with most scene elements, very few people were enthusiastic about using the film puncher.

Gathered data

Tests of the new iteration of the introduction spanned over several weeks, during which some aspects were further refined. All in all, this produced a sample size of 15, and the average SUS score of 70.83 clearly shows the major step forward that this system achieved. The data is also less spread out than in the first test, with outliers of 52.5 and 90, and a standard deviation of 11.89. The overall feedback from participants was also generally positive. Few actively complained about any part of the interface, which also dropped to zero as development went on.

As with version 1, the data from the project-specific questions is listed in table 5.2.

In total, 80% of testees gave an answer of 3 or 4 on the question of whether they would revisit Z3VR in their spare time (if they had a VR setup), further indicating that

[4]Prior experience with low-level programming

the experience was at least somewhat enjoyable and that there might be significant interest in a public release of the project.

While on average most participants had some prior experience with low-level programming languages, this question had the largest variance, with five testees answering 2 or less. Nevertheless, there appears to be no correlation between low prior experience and lower SUS scores or perceived difficulty of the chosen challenge, with the latter further indicating that the introduction was successful in conveying at least the bare minimum of low-level programming knowledge required for the challenge.

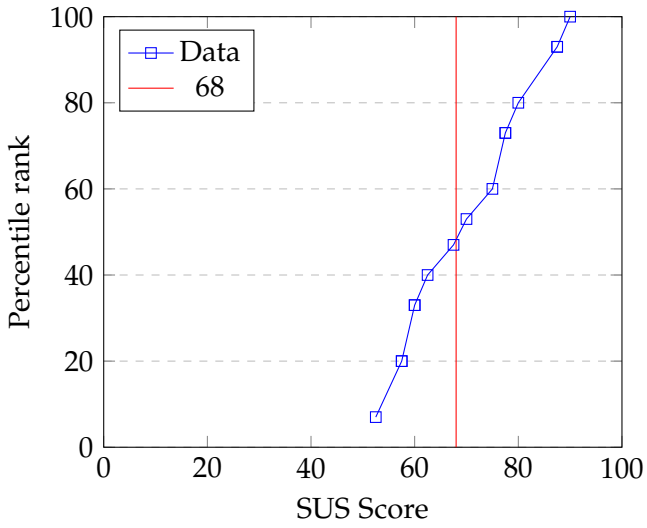


Figure 5.2.: Percentile distribution of SUS scores (Version 2, $n = 15$)

80% of participants chose to accept a challenge, with only two of those perceiving it as harder than advertised, those being the two who chose a medium challenge. This may be down to poor communication as to the difficulty of the challenge levels, but a larger sample size would be necessary to determine this for sure. For the most part, only people who answered 4 regarding their prior experience with low-level programming chose a medium or hard challenge, the only exception to this is a single testee who claimed to have "0" experience and yet chose a hard one. Of the people who did choose to take a challenge, nearly all of them completed it successfully as well, including users who claimed to have little to no experience with programming on a per-register level. This would imply that the introduction was successful in imparting at least some low-level programming concepts, though subsequent challenges would have been necessary to confirm this.

While the new SUS score of 70.83 might suggest an above-average system, the

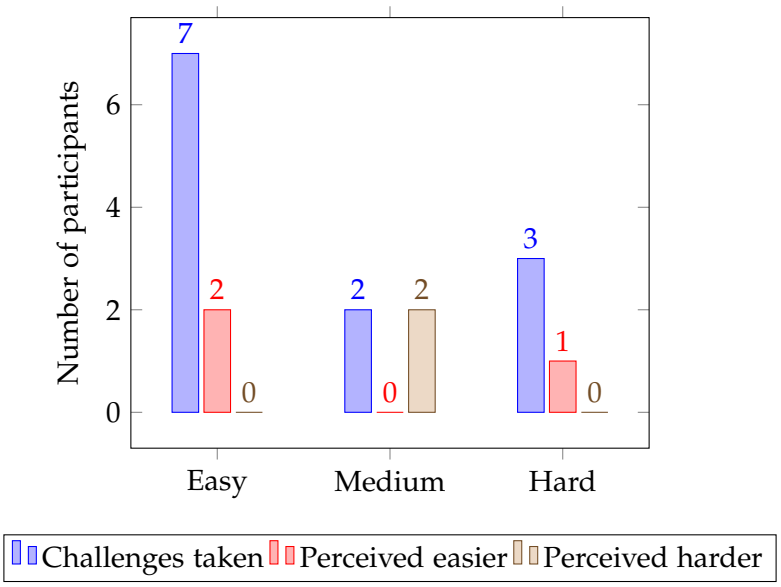


Figure 5.3.: Challenges taken and their perceived difficulty (Version 2, $n = 15$)

standard deviation of 11.89 raises doubts, and a T-test against the SUS average of 68 all but confirms these. A one-sample T-test of the gathered SUS scores against the general average SUS score of 68 (with the null hypothesis being that the implemented interface performs about average) produces a t-value of 0.8905. The sample size of 15 and two-tailed $\alpha = 0.20$ give a critical t-value of 1.345. Our t-value is well below that, so it's safe to say that the results of the interface version 2 are not significantly above or below the expected average.

5.3. Summary and lessons learned

As the data shows quite clearly, version 2 of the introduction was a major improvement in terms of approachability for users. This was also apparent during the tests themselves, as most participants were quick to grasp the interface and core principles of working with the machine. The SUS score could still see some improvement, but raising the bar from *below average* to *average* is already a promising result, and even better performance seems within reach if more tests and iteration are undertaken.

Unfortunately, some of the data gathered through the custom entries of the questionnaire is difficult to evaluate in absolute terms, and, due to the way smaller sample size of the first test, a before-and-after comparison yields no valuable information. These questions should have had more thought put into them. A redundancy system like in the SUS format, where one metric is evaluated using multiple differently phrased ques-

tions, would have certainly helped. However, when adding more and more questions on top of the 10 SUS ones, participants are likely to start skimming over them and give untruthful answers, which must be taken into account when designing them.

Further studies should also include a questionnaire for the test supervisor to fill out, as some valuable data could have been gathered through observation alone, for instance, whether a person actually completed the challenge they chose successfully, how long they took to get through the introduction, how long exactly each test lasted altogether, or how often they got stuck and the supervisor had to intervene.

6. Future work

6.1. Further work on Z3VR

The initial implementation of the simulated Z3 went so quickly that plans were made to include two more calculating machines, but as the extent of work to be put into the VR interface became clear these were ultimately scrapped.

6.1.1. Original Z3

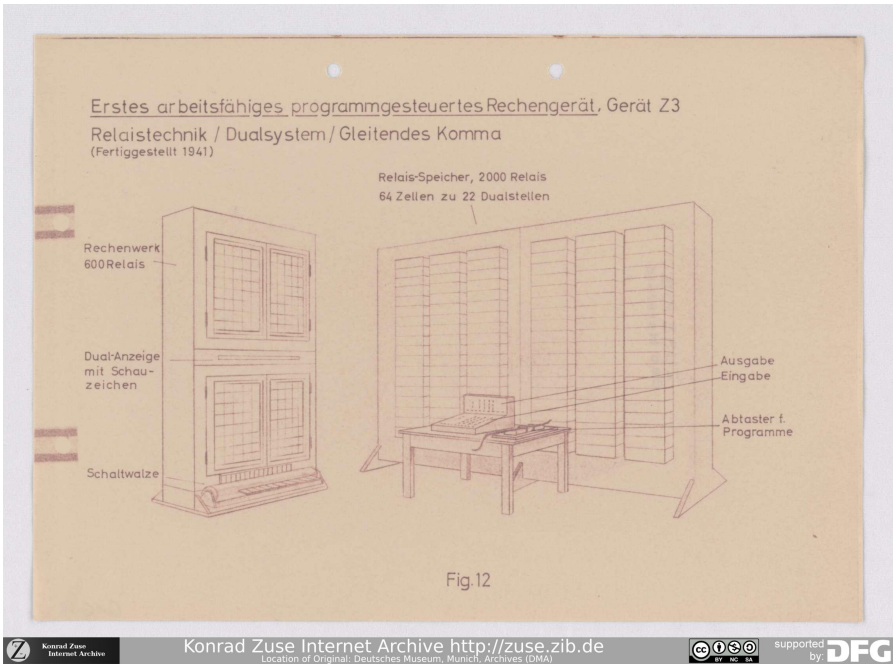


Figure 6.1.: A sketch of the original Z3 [18]

As mentioned previously, while the rebuilt Z3 and the original are functionally mostly the same, there are some differences that affect how programs can be written, and implementing this machine as an alternative option would allow for teaching more

about the history of it and its replicas. The original featured the extra instructions of $*2$, $:2$, $*10$, and $:10$. While $*2$ can be replicated easily by adding a number onto itself, all of the other operations require a constant to be stored in memory and take significantly longer to execute than the hard-coded implementation. The extended memory of 64 cells would make longer and more complex programs possible, though the number of people who could use this effectively is likely quite limited.

While there are no photos of this machine, there exists a sketch of it by Zuse which a 3D model could be based on (Fig. 6.1). Given that the design language of the Z4 (which has survived the war and was until recently on display in the Deutsches Museum) is nearly identical to the rebuilt Z3, it's safe to assume that the original Z3 also looked similar.

6.1.2. Z3+

During development, the question arose about how working with the machine would be affected if simple conditional jumps, subroutines, and instructions for immediate values were added. As such, plans were made for a hypothetical *Z3+* with such functionalities, with consideration that each of them could have been implemented on the real machine with only minor changes.

Way after the implementation of this variant was cut, a closer look into the specifications of Zuse's next computer, the Z4, revealed that it (eventually) did have all of these features, as such a far more interesting and historically significant idea would be to recreate this machine instead. The Z4 also bore other features that are very useful and extremely interesting and further warrant recreation of it in future works. For instance, it had a lamp matrix on the console where one cell would be lit, which could be moved around through program instructions. The point of this was to make putting in the values of a matrix easier, by placing a labelled piece of paper on top of the display and, while expecting an input, highlighting the cell of the matrix which was requested.

Nevertheless, the original plans for implementing these concepts in the alternative Z3 will be elaborated on and compared to how they actually were realised within the Z4.

Jumps

The jump system would have consisted of two new instructions; a *jump label* and a *jump/jumpIfNegative* operation. Both would use the lower 6 bits of the instruction code to define the address of a label, to declare where one is in the code or which one to jump to respectively. The decoding unit of the memory could then be reused to decode these labels, and a single extra relay (bit) per memory cell would allow the machine

to "remember" which labels it has already gone past by setting that new bit in the corresponding memory cell to 1. Once a jump instruction is read, a check of the marker bit would reveal whether that label has been visited already or not, in turn deciding whether the reader has to backtrack until it reads that label again (setting the marker bit of other labels to 0 along the way) or to skip ahead instead. The fill state of R_1 would determine whether the jump is conditional or unconditional: If R_1 is empty, always jump. If R_1 has a value, only jump if its sign is negative.

Assigning instruction codes to these could either be done by limiting the memory to 32 cells (as with the rebuilt Z3) and using bit 6 as an indicator whether the operation is a memory access or jump-related. Alternatively, a special instruction functioning as an escape character could mark an upcoming operation as a jump. This variant would also enable the other new instructions to be implemented more easily but would complicate finding labels during backtracking. Getting around this would involve adding a buffer to store a read address, or striking backtracking and the marker bits entirely and forcing all programs with jumps to loop such that skipping ahead will always reach the target label eventually.

Zuse solved this in a fascinating manner for the Z4. First, he introduced a separate set of instructions that operate on R_1 and produce a "boolean" value of either -1 or $+1$ (effectively clearing R_1 and using the sign bit). Examples of these are $x = 0$, $|x| \geq 1$, and $x =?$ (the Z4 also had a form of NaN's). The operation $x(-1)$ could be easily used to quickly invert a boolean value. Then, he added three conditional operations, that were only executed if the preceding boolean operation produced $+1$ [13]:

- **Fin'** $\rightarrow$ Conditional stop. If executed in the secondary reader (explained in the next section) this actually stops execution rather than returning to the main reader. A conditional switch to the main reader would instead be done by a conditional jump over a regular stop instruction.
- **Spr** $\rightarrow$ Conditional jump. All following instructions are skipped until a *start* instruction is reached.
- **Up'** $\rightarrow$ Conditional switch to secondary reader.

As you can see, jumps would only move forward and would always end at the next *start* instruction. This makes jumps to specific points in the program difficult, as a jump to a given target section would be interrupted if there was another target between the jump and the intended target, so programs had to be designed around this limitation.

Subroutines

Subroutines would be implemented by adding one or more secondary readers to the machine. A "switch to coroutine" instruction would be added which would contain an index, and a selector circuit would switch to the given reader, funnelling all further reading impulses and read codes between the control unit and the selected reader until the subroutine reached a stop instruction. Subroutines that need to be executed multiple times would of course have to be loops. To keep the number of structural changes to a minimum, no call stack was planned, instead any stop instruction would return execution to the main reader, but subroutines would still be able to call other subroutines.

Such a system, especially in combination with jumps, could eliminate the need for an operator for most metaprograms. In the Pi approximation example from Chapter 4, the pre- and post-processing programs could be combined into one program on the main reader, with two subroutine calls between the two halves. The loops could replace their exception-based stopping mechanism with a section in the program that contains a stop instruction, which gets jumped over until their stop condition is reached. Subroutines could also be used to implement a mentioned metaprogram case where two loops alternate repeatedly, with each calling the other subroutine in their stop section. Both would contain a second, "strong" stop section containing a final stop instruction.

The Z4 was fitted with a secondary reader, with capabilities to be extended by up to 4 more (never actually done). The *Up* (= "Unterplan") instruction was only valid in the main program and would move execution over to the secondary reader. Once a stop instruction was reached there, execution continued on the main reader [13]. In combination with conditional jumps, the single secondary reader could be used to hold any number of non-looping subroutines. By placing the index of the desired subroutine in a predetermined memory address in the main program, each subroutine in the secondary program could start with loading that value, subtracting its own index from it, and jumping to the next subroutine if the result differs from 0 [12].

Immediates

Instructions for immediate values would use the 6 lower bits to give the desired value in fixed point binary and act similarly to an input or memory read operation. Loading the number into a register would merely require a normalization step, as such it should take the same number of cycles as an *Add* operation or fewer.

In the Z4 this was implemented in a far more clever way, which allowed for constants that used its full 32-bit word length. The instruction At_0 denoted that the next 5

instructions would be the four bytes of a constant followed by a no-op, and the bits would be transferred straight to R_1 (or R_2) [13].

To better differentiate between the machines once the user was in their respective scene, the room and aesthetics would have changed as well to attenuate the different scenarios. The original Z3 would have been placed in the office of an aeroplane hangar, as it saw use in calculating wing surfaces during its short lifetime [11]. As the Z3+ is not based on actual history, it would have been put in a steampunk setting, to coincide with the "alternate history" aspect. If any future work does implement the Z4 in a similar way, an appropriate place to put it would be in the ETH Zurich, as the Z4 was rented out to it after the war [13].

6.1.3. Gamification

While some testers have already found the project to be somewhat enjoyable, an interesting opportunity would be a separate *gamified* mode, to reward users for their learning and leave them with a feeling of accomplishment. One suggestion that came up was a sort of "story mode", where the user was put in an exaggerated scenario where some electromagnetic impulse event has rendered all transistor-based technology inoperable and they must use the Z3 to calculate the trajectory of an incoming meteorite or similar.

The gamification can of course be much simpler and more reasonable, like an interface in the scene supplying users with tasks (not unlike the challenges testees received) and somehow rewarding users if they complete them, perhaps by unlocking further functionalities. Such a system could also be integrated into the introduction system, giving users very simple tasks at first and then incrementally increasing the difficulty level to gradually imbue them with the mindset necessary to write programs for the machine.

6.1.4. General improvements

Accuracy

Once the machine is back on display or if a collaboration with the Deutsches Museum can be arranged after all, acquiring higher resolution photos would allow for making the model far more accurate and modelling details that were hardly seen before, and extracting textures from photos directly may be possible. The machine hasn't been demonstrated publicly in decades due to fire hazard concerns and it's unknown if it will be brought up to modern standards once the exhibit reopens, so recording custom audio might not be possible, but if the raw footage of the Museum's later Z3 video

still exists, their recorded audio would be a far better alternative to the currently used sound effects, as those contain a lot of noise that was impossible to remove entirely.

More details

As it stands, all the relays of the 3D model are currently static. A short-lived experiment during development involved separating each memory cell into an independent mesh, where each of its 24 relays could be set to an ON or OFF position, to reflect the corresponding bit of the stored number. This turned out to take a large toll on performance, which is a crucial metric for any VR application. However, this trial took place before numerous performance improvements in other areas, and without consideration for *instanced rendering* (a large performance optimization which has to be enabled manually in Unity3D), so it may well be feasible to add this functionality after all. If successful, this approach could also be applied to the arithmetic unit. While a full simulation of every one of its 600 relays is out of scope for this project, at least the machine's registers could be simulated like this.

Interface

Another interesting addition would be a toggleable abstract representation of the arithmetic unit. It would reflect the contents of the registers at all times, and users would have the option to slow down time or stop it entirely and proceed one algorithmic step at a time, to be able to understand how the machine works out its results. As text objects can be manipulated like any other, further animations could be performed to show things like a mantissa being shifted during addition, or values being passed from one register to another. Since the Z3's arithmetic unit is virtually identical to that of any current-day processor [11], this would be a tremendous tool for teaching users about modern computers as well. As part of research into the Z3's architecture, Georg-Alexander Thurm created a Java simulation of it with just such a visual (minus the mentioned animations), as can be seen in figure 6.2 [11]. Unfortunately, this simulation is no longer available, and there appear to be no archived versions of it, hence it was omitted from Chapter 2.

As briefly mentioned in Chapter 5, few people enjoyed working with the film puncher. Some testees struggled to make the logical connection of the instructions being encoded as bits which in turn were encoded through holes in the film, and of those who did make that leap, not many went ahead and actually punched the program on the board in front of them. Evidently, this interface requires some more work to make it more approachable and encourage its use. One way would be to add something like the console's watchdog, but more aggressive. Instead of only showing what the user has

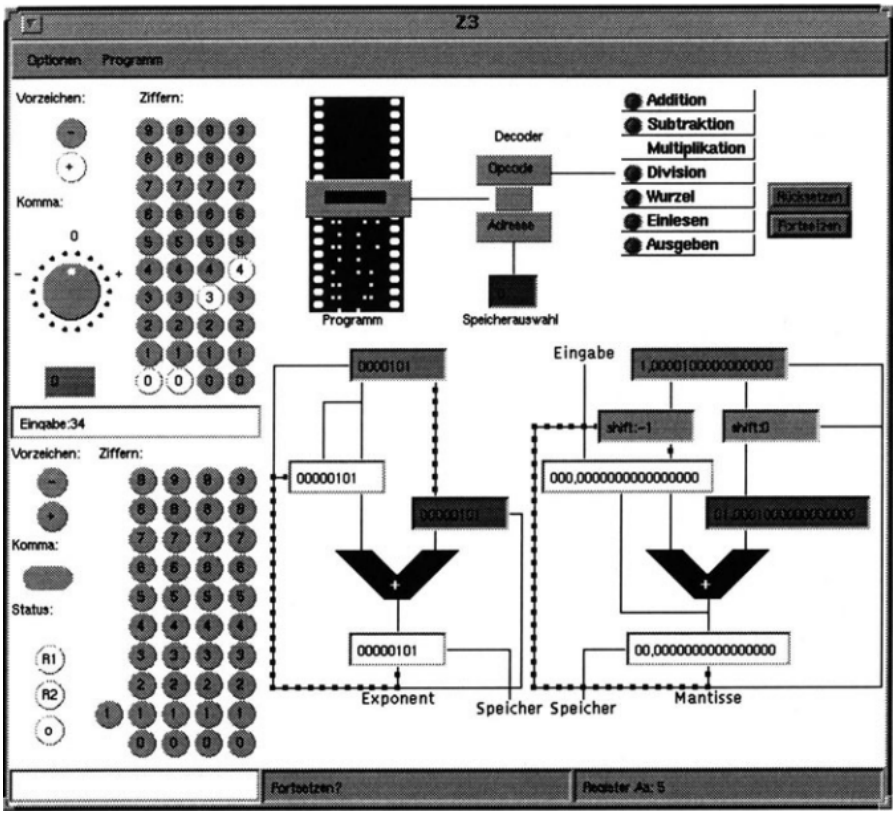


Figure 6.2.: Thurm's Java Z3 simulation

punched so far to the left of the puncher, a copy of the program on the programming board could be displayed instead, with yet-unpunched instructions being greyed out and only the bit-levers corresponding to the next instruction being able to be pushed down in the first place, perhaps highlighting them as well. This would more explicitly guide users to press the right levers and prevent them from accidentally punching an invalid instruction, which unfortunately happens somewhat frequently with virtual fingers that can easily slide across nonphysical levers. This latter issue specifically was planned to be solved by making the fingertip "stick" to the middle of a lever, but this could not be implemented in time.

Trivia

As briefly mentioned in Chapter 3, Z3VR currently features an info book that, among other things, holds trivia about the machine itself and things like the program films. What it does not yet cover, however, are more "meta" bits of trivia, such as who Konrad

Zuse was, under which circumstances he built the machine, that the machine users see is not the *original* Z3, etc. It was originally planned to have a second room in the scene, where a lectern would stand in the middle on which various trivia books could be placed to see larger text on a projector screen about the relevant subject. In fact, the walls of this second room are included in the current scene, but it is closed off with a non-interactable door.

Since these extra historical facts would need longer texts to fully cover, it may be more approachable to produce audio recordings or even small videos to explain them, and show *those* on the projector screen instead, though this would require a lot more work.

6.2. Lessons learned

6.2.1. Research goal

The research goal of evaluating how effective Z3VR is for teaching about the Z3 and low-level programming wasn't defined well enough when the first tests began and the questionnaire was created. As such the chosen project-specific questions are far from optimal for this evaluation, as mentioned in Chapter 5. If enough time and testers were available, a further study with a new set of questions would be desirable. A subset of questions to determine the participant's knowledge of the Z3 and low-level programming *before* and *after* the test could be interesting, but care must be taken not to make the questionnaire too lengthy to ensure testers would still answer properly.

6.2.2. Interface

The workload required to implement a full VR interface that handles well and intuitively was drastically underestimated. While the creation of 3D assets and implementation of the Z3's simulation only took a few weeks, the interface and introduction demanded about two months of work. This was further extended by the required total overhaul of the introduction and several interface systems that were too unintuitive for testers. The most prominent example of this is the info book and program folder interaction model. In their original form, opening and closing them required users to first hold them in one hand, then to grab them with their other hand as well. Turning pages in the book was done by moving a hand across them (without grabbing) as if you were swiping the page along. Telegraphing these behaviours to users was very difficult and left many confused.

These issues could have been identified and addressed far sooner if more testing had taken place during development, as such future endeavours within this project or on others will certainly involve way more testing way sooner.

7. Conclusion

7.1. Summary

To conclude, Z3VR has shown to be an effective tool for teaching people about the Z3 and how to operate it. It presents users with an approachable interface that allows them to experience this machine first-hand in an immersive and accurate fashion, mastering which takes most only 15 minutes. Whether the project is as effective for teaching concepts of low-level programming is difficult to determine through the gathered data alone, and further research would be necessary to come to a definitive conclusion, however, most test participants successfully applied what they learned in their given challenges. The project is also a prime example of the importance of early and frequent testing, which didn't start soon enough in this case, and the data-gathering methods left a lot to be desired. Nevertheless, the end product is a viable teaching tool with a lot of potential for future improvement and expansion.

7.2. Outlook

Use in musems

As for the future of the project itself, one can imagine it being used as part of exhibitions in museums. Cooperation with the Deutsches Museum would be especially beneficial, as the real machine can't be demonstrated anymore due to fire safety concerns and a full rewiring to fulfil requirements may not be possible[1]. As such, Z3VR could be an accompanying part of the exhibit to still let people see the machine in action without risking damage to the Z3 or its surroundings.

To better facilitate use in museums, a separate "guided tour" mode may be interesting. Since the general public (and especially children) are not versed in programming concepts at all and likely not too interested in learning how to program an 80-year-old computer, it would merely guide visitors around the different parts of the machine and run some predetermined programs. The occasional genuinely interested viewer would still have the option to switch to the programming mode.

[1] `https://www.youtube.com/watch?v=TbW-qNxD1lE`

Museum visitors would also inadvertently be playtesters, and while handing them questionnaires to fill out would likely be unproductive, merely observing their interaction with the project and potentially gathering some in-game metrics would be a tremendous help in finding further shortcomings and things to improve about the interface. Watching users fumble with some grabbable object or button was the major method of finding weak spots during development, often more so than the testee's own feedback.

General distribution

As stated in Chapter 2, there is currently no working, publicly available and accurate simulation of the Z3. With this historically significant machine being relatively unknown outside of select groups, there really ought to be something like Z3VR which anyone can access at home to learn about it. To this end, a non-VR version of the project would need to be made as well, as the adoption of VR is still fairly limited among consumers, even if the significant immersion factor would be mostly lost.

Distribution on a popular site for independent projects like itch.io[2] may then spark more interest in computer history and teach a new generation of internet dwellers about the works of Konrad Zuse.

[2] `https://itch.io/`

List of Figures

List of Tables

Bibliography

[1] M. Campbell-Kelly. *Edsac Simulator*. University of Warwick website (https://dcs.warwick.ac.uk/~edsac/, accessed 30.8.2023). 1996.

[2] M. Campbell-Kelly. "Past into present: the EDSAC simulator". In: (Jan. 2000), pp. 397–416.

[3] W. Consortium. *VRML97 Functional specification*. Format standard specification (https://www.web3d.org/content/vrml97-functional-specification-and-vrml97-external-authoring-interface-eai, accessed 14.8.2023). 1995.

[4] D. C. Frezzo and M. W. Waterman. "Immersive Virtual Reality for Teaching Problem Solving in Networking". In: *2020 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*. 2020, pp. 968–971. DOI: 10.1109/TALE48869.2020.9368450.

[5] P. F. Navarro. *Zenva Sky - VR App to Learn Computer Science*. Game published on Oculus store (https://devmesh.intel.com/projects/zenva-sky-the-world-s-first-vr-app-to-learn-computer-science). 4.7.2019.

[6] piper1. *pipZuseZ3*. SourceForge (https://sourceforge.net/projects/pipzusez3, accessed 13.8.2023). 2023.

[7] M. Riley. *ZuseZ3*. GitHub (https://github.com/rileym65/ZuseZ3, accessed 13.8.2023), personal website (http://www.historicsimulations.com/ZuseZ3.html, accessed 13.8.2023). 2019.

[8] J. Röder, R. Rojas, and H. Nguyen. *Konrad Zuse Internet Archive*. Website of the ZIB (http://zuse.zib.de, accessed 14.8.2023). 2013.

[9] J. Röder, R. Rojas, and H. Nguyen. "The Konrad Zuse Internet Archive Project". In: *Making the History of Computing Relevant*. Ed. by A. Tatnall, T. Blyth, and R. Johnson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 89–95. ISBN: 978-3-642-41650-7.

[10] R. Rojas. "How to make Zuse's Z3 a universal computer". In: *IEEE Annals of the History of Computing* 20.5997440 (1998).

[11] R. Rojas. *Die Rechenmaschinen von Konrad Zuse*. Ed. by R. Rojas. Springer Berlin Heidelberg, 1998. ISBN: 978-3-642-71945-5. DOI: 10.1007/978-3-642-71944-8.

[12] R. Rojas. "Konrad Zuse und der bedingte Sprung". In: *Informatik-Spektrum* 37 (1 Feb. 2014), pp. 50–53. ISSN: 0170-6012. DOI: 10.1007/s00287-013-0717-9.

[13] H. Rutishauser. *Gebrauchsanweisung Z 4*. Operation manual (https://www.e-manuscripta.ch/zut/doi/10.7891/e-manuscripta-98601, accessed 5.9.2023). 1952.

[14] R. Warp, M. Zhu, I. Kiprijanovska, J. Wiesler, S. Stafford, and I. Mavridou. "Validating the effects of immersion and spatial audio using novel continuous biometric sensor measures for Virtual Reality". In: *2022 IEEE International Symposium on Mixed and Augmented Reality Adjunct (ISMAR-Adjunct)*. 2022, pp. 262–265. DOI: 10.1109/ISMAR-Adjunct57072.2022.00058.

[15] E. Yigitbas, C. B. Tejedor, and G. Engels. "Experiencing and Programming the ENIAC in VR". In: *Proceedings of Mensch Und Computer 2020*. MuC '20. Magdeburg, Germany: Association for Computing Machinery, 2020, pp. 505–506. ISBN: 9781450375405. DOI: 10.1145/3404983.3410419.

[16] T. Zoppke. "Simulating the ENIAC as a Java Applet". PhD thesis. Diploma Thesis, Computer Science Department, Freie Universität Berlin, 2004.

[17] K. Zuse. *Der Computer - Mein Lebenswerk*. Springer-Verlag Berlin Heidelberg, 1984.

[18] K. Zuse. *Electrical plans of the rebuilt Z3*. Available on the Konrad Zuse Internet Archive (http://zuse.zib.de, accessed 14.8.2023). 1960.

A. appendix

Alle gesammelten Daten werden anonym behandelt. Bitte hinterlassen Sie keine identifizierenden Merkmale auf dem Fragebogen.
VR kann generell und besonders bei Bewegung *Cybersickness* hervorrufen. Falls Sie sich unwohl oder schlecht fühlen, kann das Experiment jederzeit abgebrochen werden.

Ich habe das gelesen und bin damit einverstanden: _____

All gathered data will be handled anonymously. Please do not leave any identifying marks on the questionnaire.
VR in general and especially during movement can cause *Cybersickness*. If you feel unwell or sick, the experiment can be aborted at any time.

I have read and agree to the above:
_____

Figure A.1.: Info sheet

System Usability Scale

	Strongly disagree				Strongly agree
1. I think that I would like to use this system frequently.					
2. I found the system unnecessarily complex.					
3. I thought the system was easy to use					
4. I think that I would need the support of a technical person to be able to use this system.					
5. I found the various functions in this system were well integrated.					
6. I thought there was too much inconsistency in this system.					
7. I would imagine that most people would learn to use this system very quickly.					
8. I found the system very cumbersome to use.					
9. I felt very confident using the system.					
10. I needed to learn a lot of things before I could get going with this system.					

	Strongly disagree				Strongly agree
• I felt immersed in the virtual space					
• The introduction was clear and concise					
• I would revisit Z3VR in my spare time (if I had a VR setup)					
• I've worked with low level programming before					
• The time I had for the challenge was sufficient					

	Easy	Medium	Hard
• The challenge I was given was rated as ___			

	way easier					way harder
• I felt the challenge was ___						

Figure A.2.: Questionnaire