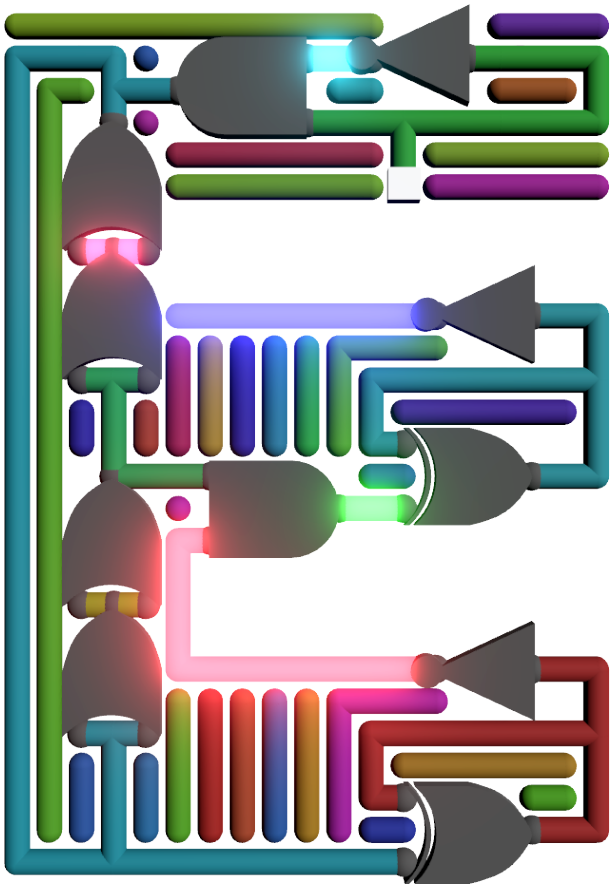
Creation of an educational video game

The Making of Electrica



Matura Paper	**Kantonsschule Sargans**
Author	Jonas Maier, bNP15
Supervisor	Bernhard Frei
Submission Date	January 7, 2019

Table of Contents

1 Introduction

1.1 Motivation

Since the first time playing a video game, they have been an active interest in my life. During the time at high school, my interest in video games has increased even further. Moreover, since the beginning of high school, I have been interested in programming. Combining these two interests, creating a game seemed to be a great idea which I could benefit from by deepening my skills in programming and learning what is needed in order to create a working game.

Another interest of mine has always been electricity and electric devices. For a long time, a processor seemed like an incredibly complex machine to me. After I recently learned how logic gates are used in a computer, I began to get a better understanding of the whole. Having learned how logic gates are used in a computer recently, I wondered if it was possible to create a working processor by myself. Because a lot of components are needed in order to create a basic processor, it seemed to be a good alternative to create one digitally.

Combining the two aforementioned ideas, I thought it would be a great idea to create a game that teaches the inner workings of a processor by making the player build one by themselves.

1.2 Objectives

The goal of this project is to create a working prototype of the game, which should be able to teach any interested person with no prior knowledge how logic gates, simple circuits and eventually how a Central Processing Unit[1] works. In the beginning, logic gates are introduced. The player is being told about their function and uses. Based on the aforementioned basic logic gates, the game ought to introduce increasingly complex circuits, such as registers, decoders, and finally circuits that operate with stored binary numbers. Using registers and circuits that can manipulate binary numbers, the user should be able to single-handedly build an Arithmetic Logic Unit[2] and later a whole CPU. The architecture of said CPU ought to be kept as simple as possible in order not to overburden the player with useless information.

1.3 Inspirations

At least some inspiration for this project came from Ben Eater, who created a working 8-Bit computer using nothing but cables, breadboards and basic chips, most of which have the functionality of logic gates. He documented and filmed the creation of said computer and uploaded the videos as a guide on how to build a computer to YouTube. (Eater, 2018)
Going in the same direction, another element that inspired this project was surely the concept of

[1] CPU is the abbreviation which will be used from now on.
[2] The abbreviation ALU will be used from now on.

a Minecraft redstone computer. A Minecraft world consists of equally sized cubes. Redstone is a material that can be compared to an electric conductor. Using different arrangements of redstone blocks, it is possible to replicate logic gates. Because Minecraft is a sandbox survival game, it is not very suitable for simulating logic circuits. Creating a game optimized for logic circuits should solve some issues that exist with redstone computers.

1.4 Procedure and Method

A game is usually started by building a basic prototype for the core mechanics of the game, which is also how this project is going to be started. The core mechanics include, but are not limited to the electric simulation and building of cables and logic gates. After completing the core mechanics, the rest of the program needs to be created, such as the User Interface and a guide for the player. During the entire procedure, the game also needs to be tested for errors.

1.5 Structure of the Paper

This paper consists of three main parts.
Chapter 2 explains the basics of creating a video game, as well as which tools were used in the process of creating the game.
Chapter 3 and 4 tell the idea and the process of creating the game respectively.
In chapter 5 the core gameplay is shown based on the creation of an SR-Latch.
To conclude the paper, a summary is given.

2 Basics of creating a game

In this part of the paper, it is explained what programs and programming languages were used in the development of the game.

2.1 Programs

2.1.1 Unity

Unity was a central part of the creation process of the game, namely being the game engine. The game engine is responsible for a big part in every game, as it provides a solid foundation. Included in the game engine are a lot of aspects that are not game-specific. To elaborate further, non-game-specific features are needed in almost every game and are not what makes a game unique. Such features are 3D- and 2D-rendering, audio, game physics, user interface, artificial intelligence, animation and a lot more. One game does not necessarily use all of the aforementioned features, but they are used in enough games that they are worth to be included in the game engine so that the

individual game studios do not have to program it themselves every time they create a game. Unlike many other engines, Unity makes it effortless to create a cross-platform game, which means that with only small changes the game can be run on any operating system. The great cross-platform support, as well as the easy to use interface and C# scripting were the main reason for choosing Unity over other game engines.

2.1.2 Blender

Blender is a 3D rendering program with a lot of different features. It is certain that not all aspects of Blender are used in this project, but only a handful. Blender can be used for modeling, rigging, animating, simulating, rendering, compositing and motion tracking, and even video tracking. (*Blender Documentation*, 2018)

With Blender, an experienced person could create an entire animated movie without once having to use different software. However, for this project only the modeling part of Blender is going to be used. Modeling describes the process of creating three-dimensional objects consisting of polygons. These objects can then be exported to Unity and used in the game as assets.

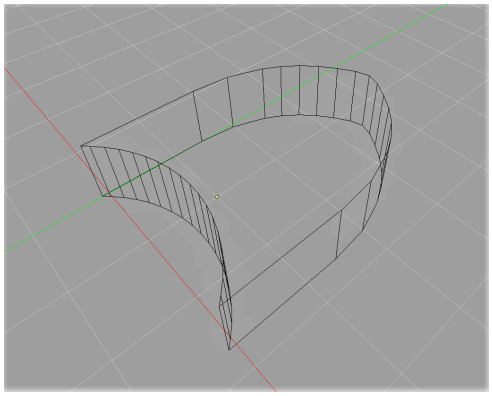


Fig. 1: 3D-object in Blender

2.2 Programming Languages

2.2.1 C# and Scripting in Unity

C#, pronounced C-Sharp, is the programming language that Unity utilizes for its front end. ”[It] is a general-purpose, multi-paradigm programming language encompassing strong typing, imperative, declarative, functional, generic, object-oriented (class-based), and component-oriented programming disciplines.” (*C Sharp (programming language)*, 2018)

In Unity, a C# script is treated like any other component that can be attached to a GameObject (*GameObject*, 2018).
Such scripts need to derive directly or indirectly from the class $MonoBehaviour$ so that they can be properly used by the game engine. (*MonoBehaviour*, 2018)

The most important parts of such a script are the $Start()$ and $Update()$ methods. $Start()$ gets called once when the game is started, which is why this method is used to initialize the object. $Update()$ gets called in every frame and is suitable for actions that happen over time, such as a uniform movement of an object. The behavior of a script is not limited by anything except one's ability to program it. For instance, a more complex script might manage the keyboard input for all the other objects in the game.

An important part of scripting in C# is the Application Programming Interface[3] provided by Unity itself. As its name suggests, it is the interface where game specific scripts can interact with the game engine. As an example, the API provides the class $Vector3$, which represents a three-dimensional vector. The class also has some useful functions, like $Vector3.Dot(vector_0, vector_1)$, which can be used to calculate the dot product of two vectors.

2.2.2 HLSL

HLSL stands for High Level Shading language and is, as its name suggests, used to program shaders in Unity. It is fundamentally different from C# concerning the syntax and function. Although the name suggests that it is a high-level language, it is much closer to the specific hardware than C# and only supports a small range of programming paradigms.

A shader can be separated into two parts: A vertex shader and a fragment shader.

The vertex shader can move the vertices of an object each frame before it is drawn. A vertex is one of many points that make up the polygons which each three-dimensional object consists of. By moving the individual vertices, the vertex shader can alter the shape of an object. A practical application for this would be water movement for a lake or grass swaying in the wind.

A fragment shader is executed once for every pixel of a triangle. It determines the color of each pixel using image textures, the direction of the light source and other parameters. This shader can also take into account bumps in the surface or reflectiveness of the surface.

(*Microsoft Docs - Reference for HLSL*, 2018)

3 The Concept of the Game

3.1 Game Design

3.1.1 Mechanics

A good principle to follow in video game design is to only use a handful of game mechanics while providing enough possibilities with these mechanics so that the player does not become bored. On the other hand, if there are too many mechanics, the player can feel overwhelmed or may have to

[3] API for short

relearn the game upon adding new mechanics. This prevents a natural learning curve by keeping the player from improving their previous skills. A rule of thumb is to have a few mechanics that can be used in many different ways to provide a rich experience while keeping the game familiar.

Following this rule, there are only two mechanics in the game: building logic gates or electric chips and connecting them with cables. Using these, the player can progress through the entire game because there are almost endless possibilities to combine the same logic gates to achieve different behaviors. To start the game, a task is to build a working XOR-gate consisting of only AND, OR and NOT gates. At a later stage in the game, there is a task to build a circuit that adds two binary numbers.

3.1.2 Art style and Aesthetics

The style of the game is kept as simplistic as possible for several reasons. When building bigger circuits, it remains easy to recognize different parts, where high-quality graphic effects might be distracting. The second and more obvious reason is to have the least possible amount of work while creating the game. The finished program should be seen as a proof-of-concept and not as a finished game. By reducing the time spent on creating high-quality graphics, it is possible to use the saved time to create more functionality in the game.

3.2 Theoretical Aspect of the Content

3.2.1 Logic Gates

Logic gates are the building blocks that determine the behavior of a logic circuit. The most basic logic gates have up to two inputs and one output. The input and output values can have exactly two different states: On or off[4]. If given the same input, the output will always be the same for the same logic gate.

A logic gate can be described as a quite primitive function with up to two arguments, which have either a value of zero or one. The output of the function also is either zero or one.

A function $y = f(i_0, i_1)$ can be found for every primitive logic gate, where the function's domain and range are $\{0, 1\}$. The correct mathematical representation of AND, OR and NOT gates are given by equations 1, 2 and 3 respectively.

$$a(i_0, i_1) = i_0 \wedge i_1 = i_0 * i_1 \tag{1}$$

$$o(i_0, i_1) = i_0 \vee i_1 = i_0 + i_1 - i_0 * i_1 \tag{2}$$

$$n(i) = \neg i = 1 - i \tag{3}$$

The specific solution to all possible inputs for these logic functions are given in tables 1, 2, and 3. Interestingly enough, AND and OR gates can be constructed using only the other two basic gates.

[4] In this context, on and off is analog to 1 and 0

i_0	i_1	a
0	0	0
0	1	0
1	0	0
1	1	1

Tab. 1: AND Logic

i_0	i_1	o
0	0	0
0	1	1
1	0	1
1	1	1

Tab. 2: OR Logic

i	n
0	1
1	0

Tab. 3: NOT Logic

i_0	i_1	$a(i_0, i_1)$	$o(i_0, i_1)$
x	0	0	x
x	1	x	1
x	$n(x)$	0	1

Tab. 4: AND and OR logic with one variable

When negating the inputs and output of the OR function, we get the AND function and vice versa. The mathematical proofs of this lie in equations 4 and 5.

$$\begin{aligned} a(i_0, i_1) &= n(o(n(i_0), n(i_1))) \\ &= 1 - ((1 - i_0) + (1 - i_1) - (1 - i_0) * (1 - i_1)) \\ &= 1 - (2 - i_0 - i_1 - (1 - i_0 - i_1 + i_0 * i_1)) \\ &= 1 - (1 - i_0 * i_1) \\ &= i_0 * i_1 \end{aligned} \tag{4}$$

$$\begin{aligned} o(i_0, i_1) &= n(a(n(i_0), n(i_1))) \\ &= 1 - ((1 - i_0) * (1 - i_1)) \\ &= 1 - (1 - i_0 - i_1 + i_0 * i_1) \\ &= i_0 + i_1 - i_0 * i_1 \end{aligned} \tag{5}$$

3.2.2 Storing Data

There are different ways to store data only using logic gates. One of the simplest ways is using an sr latch, which stands for SET-RESET-Latch. Made up of only two NOR-gates[5], it can store an alterable value. It is possible to set the value by enabling the set-input of the circuit, and to reset the value by enabling the reset input of the circuit [6]. (Kuphaldt, 2007, Chapter 10.2)

The sr latch is depicted in figure 2. S, R, V, and W are the set and reset-inputs, the stored value and the complement of the stored value. The value of V and W are given by the equations 6 and 7

[5] equal to OR-gate with NOT-gate on the output

[6] Setting and resetting the value describe the actions of storing a value of 1 or a 0 respectively.

respectively.

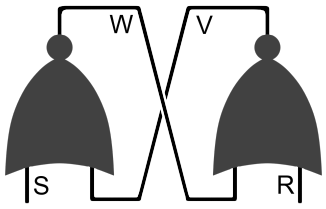


Fig. 2: Labeled SR Latch

$$V = n(o(R, W)) \tag{6}$$

$$W = n(o(S, V)) \tag{7}$$

State 1a: $S = S, R = 0$

$$W = n(o(S, V)) \tag{8}$$

$$\begin{aligned} V &= n(o(R, W)) \\ &= n(o(0, W)) \\ &= n(W) \\ &= o(S, V) \end{aligned} \tag{9}$$

State 1b: $S = 0, R = R$

$$V = n(o(R, W)) \tag{10}$$

$$\begin{aligned} W &= n(o(S, V)) \\ &= n(o(0, V)) \\ &= n(V) \\ &= o(R, W) \end{aligned} \tag{11}$$

State 2: $S = 0, R = 0$

$$\begin{aligned} W &= n(o(S, V)) \\ &= n(o(0, V)) \\ &= n(V) \end{aligned} \tag{12}$$

$$\begin{aligned} V &= n(o(R, W)) \\ &= n(o(0, W)) \\ &= n(W) \\ &= n(n(V)) \\ V &= V \end{aligned} \tag{13}$$

State 3.1: $S = 1, R = 1$

$$\begin{aligned} W &= n(o(S, V)) \\ &= n(o(1, V)) \\ &= n(1) \\ &= 0 \end{aligned} \tag{14}$$

$$\begin{aligned} V &= n(o(R, W)) \\ &= n(o(1, W)) \\ &= n(1) \\ &= 0 \end{aligned} \tag{15}$$

State 3.2: $R = 0, S = 0$

$$\begin{aligned} W &= 1 \\ W &= n(V) \qquad (16) \\ W &= 0 \perp \\ V &= 1 \\ V &= n(W) \qquad (17) \\ V &= 0 \perp \end{aligned}$$

Provided that the R-input is 0, the value stored in the latch is 1 when either the S-input is 1 or the value is already 1. This is shown with state 1a. State 1b shows the same thing for the R-input and the inverted value. When no input is 1, the value stays the same, which is shown with the equations of state 2. State 3 is an illegal state, because it causes undefined behavior of the circuit. When both inputs are 1 and at the exact time set to 0, V and W cannot be determined. This is shown in state 3.1 and 3.2, meaning that first the inputs are set to the state of 3.1 and immediately afterwards to the state 3.2. The state is called illegal because it is possible to reach that state but it is not desirable. After reaching the illegal state, the value of the sr latch is in a mathematical or simulated context undefined until one input has the value 1. In a real world application, this would just mean that the sr latch obtains a random value.

Using multiple latches, it is possible to store a binary number where the value of the different digits are given by the states of the individual latches.

3.2.3 Binary System

The binary system is the most basic number system which all computers employ. A binary digit[7] can only have two different values: zero or one. Using the latches explained in chapter 3.2.2, numbers can be stored quite easily. The value of a latch or the latch itself represent one binary digit. Due to this, the possible size of a number is limited by the number of latches that are used for a number. Eight latches correspond to eight bits which limits the stored numbers to a maximum of $2^8 - 1 = 255$.

[7] commonly known as bit

3.2.4 Arithmetic Operations

Storing numbers in a circuit is not of much use if there is no possibility to do anything with them. That is what arithmetic is for. By using advanced circuits that consist of logic gates, it is possible to perform basic mathematics on stored values. A basic example of an arithmetic operation is addition. A circuit that adds two binary numbers operates surprisingly similar to the way addition is taught to children in primary school. However, because the numbers are binary, the maximum value that has to be carried between two addition steps is one. Binary addition starts with the least significant digit. The two digits are added, where the sum can have a value between 0_b and 11_b. When the sum of these two digits is bigger or equal to 10_b, a 1_b is carried to the next two binary digits. The carried value then adds to the next sum, which in turn can cause a value of 1_b to be carried to the following two digits. (Kuphaldt, 2007, Chapter 9.2 - 9.3)

3.2.5 Arithmetic Logic Unit

The arithmetic logic unit is a collection of all arithmetic and logic operations of a processor. Arithmetic operations include addition, subtraction, multiplication and division. Logic operations include bitwise AND, OR, NOT or XOR. It is useful to keep all the operations in one part of the CPU, as it makes it easier to execute different operations in a similar manner.

An ALU has two inputs which are called *operands*, and an opcode which indicates which operation to perform on these operands.[8] The ALU also has an output, which is the result of the operation.

A processor has a fixed set of instructions, and the majority of these instructions are arithmetic operations. The other instructions control the execution flow of the loaded program[9] These types of instructions are not arithmetic operations and therefore do not need an ALU.

3.2.6 Bus

The term *bus* refers to the universal electric data line that is connected to all substantial parts of a processor, for instance, the registers, the ALU or the Control Unit. Its purpose is to transport data from one location in the processor to another one. Each part that is connected to the bus can essentially either write to the bus or read the bus's value. In case of the execution of an arithmetic operation like the addition of two numbers, these numbers are sent from the RAM to the ALU in order to be stored there. Following that, the desired opcode is sent from the Control Unit to the ALU, where the ALU executes the operation with the given opcode. Finally, the ALU writes the result to the bus and a different register copies the content of the bus and stores it.

[8] e.g. Addition could have the opcode 0, subtraction the opcode 1, etc

[9] e.g. loops, conditional jumps

3.2.7 Control Unit

The control unit of the processor is the part of a processor that pulls the strings behind the scenes. It consists of a few parts: A program counter, which indicates which instruction to execute next, a few registers to temporarily store the data, and controlling connections to every other component in the processor. In each clock cycle, the control unit takes the current instruction that is given by the program counter and temporarily stores it. If for example the instruction has a leading one, it stands for a control flow instruction, like a jump instruction. Depending on that, the control unit either executes a specific control flow execution, like jump or conditional jump, or an arithmetic operation by manually opening the bus at specific registers and letting the ALU do the work.

3.2.8 Processor

A processor consists of all parts discussed in the three aforementioned sections. Each one is integral to the functionality of the CPU and it is not possible to have a CPU without every single one. Although a normal computer consists of more than just a CPU, a CPU can function on its own[10]. A typical processor nowadays also has connections to the buses on the motherboard and thus a connection to parts like memory, video cards, hard drives and each external part that is plugged in.

3.3 Game Content

This section describes the features that are included in the game. More specifically, the most important parts of the game that are included in the current version are described here.

3.3.1 Simulation of Electricity

First and foremost is the electricity. The electricity that is simulated in the game is not comparable to real electricity, but rather a depiction of logic gates interacting with each other. It is the core element of the game, which the rest is built around.

3.3.2 Saving and Loading

Although it is already a big achievement to simulate logic circuits, it is not possible to complete big projects in the game if there is no possibility to save the made progress. This is why the saving and loading of electric circuits was implemented soon after finishing the simulation of electricity.

3.3.3 Creating Custom Circuits

After having tried to create bigger circuits, it was obvious that there are a lot of repeating patterns in certain logic circuits. Because of this, the feature to create custom circuits with a certain number of

[10] Assuming that this theoretical processor is not in need of electric energy

in- and outputs was made. As seen in figure 3, the final product looks and behaves like a computer chip. The arrows indicate on which side the in- and outputs are.

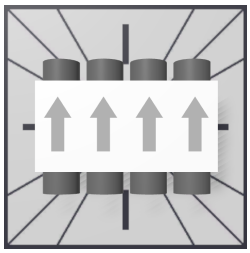


Fig. 3: Custom Circuit

3.3.4 Campaign

Due to the slow nature of logic circuits that the players can build themselves and because there still needed to be a way for the player to progress through the game, a campaign mode was created. It consists of multiple tasks that the player can finish one after another.

A task consists of a description what kind of circuit the player ought to build and a set number of in- and outputs that the player can connect to their circuit in the electric simulation. The player then has to build a circuit that fits the task's description. When the player finishes building the circuit, they can test it by pressing a button in the UI. If the circuit is built correctly the task is completed, otherwise, the player can edit the circuit until it works in a correct fashion. When a task is completed, the player unlocks an electric chip with the functionality of what they previously built, with the simple difference that it completes a computational step in only one game-tick, rather than the number of ticks that the player's circuit actually needed. Some tasks require another task to be completed first which prevents the player from trying to build a circuit that is too complex to understand at the moment, so that there is a natural progression through the game. This campaign acts as a replacement to the custom circuits because it is faster in the game and unlike the custom circuits, it guides the player through the game.

4 Game Development

4.1 Modeling Logic Gates

Creating these props for the game was not a big task, as the only gates needed were the AND, OR, NOT, XOR, and NAND gates. Having used *Blender* before, it was merely a task of a few minutes to complete each gate. Figure 4 displays the aforementioned gates (left to right, top to bottom). Surprisingly, not any other 3D model was used in the game, as all the cables and electric chips consist of default shapes provided by *Unity* such as cylinders, spheres or cubes.

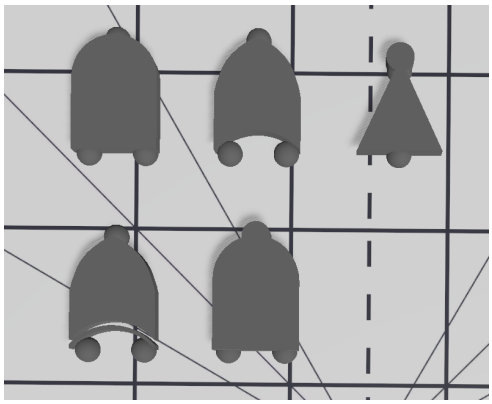


Fig. 4: Basic Logic Gates

4.2 Generic Game Programming

Although creating game-specific scripts[11] was not a small task, there is also much that has to be programmed generally for a lot of games, with no regards to the type of the game. Such generic scripts include a window manager, a loading screen script, game settings, and a persistent data storage script, among other things. It is very different from programming core elements of the game but is needed just as much.

In Unity, the game is split into different scenes. A scene may not necessarily be a level, but can also function as the main menu screen. If not explicitly specified, GameObjects are unloaded upon loading of an other scene. Thereby, all data stored in the objects is lost. This means that it is difficult to transfer data from one scene to another, for example from the main menu to the level. A possible use case for such data transfers is the name of the save, so that the game scene knows which save to load from. For this purpose, the scripts PersistentData.cs and DataStorage.cs were created, which can be found in Appendix A.

The script can be added to any scene and ensures that there is only one 'main' object which holds all the data. Other parts of the game can take any PersistentData object and store variables with the use of a key. The data is preserved on scene changes and can thus be used by different objects in an other scene.

4.3 Cables

Creating games from code did not seem like a big task at first, but there are a lot of details to consider. Cables can connect to connectors of logic gates or to other cables. A cable cannot be built diagonally in order to prevent chaos. If a cable is built, it has to be checked if the cable crosses an other cable anywhere and has to be split into two parts at the intersection if that is the case. To make it easier for the player, the current that flows through a bunch of connected cables is the same everywhere. This means that the value[12] of several indirectly connected cables needs to be synchronized.

[11] Such scripts include the electric simulation among other things
[12] Value of 0 or 1 of the simulation

On a similar note, the cables appearance also needs to be programmed. For this, a simple shader was programmed with Unity's surface shader function. The shader can be given a color variable, which controls the color of the cable and a glow variable, which controls how intense the cable glows when current passes through. The file is called CableShader.shader and can be found in Appendix A. Programming shaders is a lot different than programming simple scripts, which can also be seen when comparing the different code snippets in the appendix with the shader.

4.4 Logic gates

Logic gates were fairly easy to implement. They have a fixed amount of input- and output-connectors, which only can connect to cables. A function in the class of the gate determines which value the connectors have the next game-tick.

4.5 Self-created Circuits

A player can create his own custom circuits. These circuit then can be used in the game like any other logic gate. It resembles a microcontroller and behaves similarly. All the inputs are on one side and the outputs on the other side. It is used to simplify the user's task. An sr latch, as an example, can be made compact by building it as a custom circuit. Once made, it can be used multiple times in the same game. It assists the player by releasing him from the task of repetitively building the same circuit over and over again.

4.6 Tasks or Missions

As mentioned in chapter 3.3.4, a campaign mode was added to the game to give a rough guide for the player. As it turned out later, it is suitable for the core game of teaching how a processor works. It is possible to teach the player step by step what a complex circuit consists of. From the programming side of view, it was not a very complex task. All that was needed was a script that compares the outputs of the circuit built by the player and a reference circuit, given the same inputs. The script automatically generates different inputs to test. The program then tests twenty to fifty times with different inputs if the outputs of the two circuits are the same. If the player's circuit passes all these tests, he unlocks a new item and can build that instead of building the same circuit each time. As mentioned in chapter 3.3.4, this replaces the self-created circuits, which is the reason why that feature is not included in the latest build of the game.

5 Gameplay

In this section, screenshots of the game are shown as well as an example of a circuit that can be built.

5.1 Basic Gates

Some basic components are needed for most circuits that the player can build. Cables can be used to connect these components. A glowing cable indicates that it is powered. To control if a single cable is powered an electric switch can be used, as demonstrated in figure 5. A switch is not a logic gate, but still one of the most important basic components. The three other basic gates shown are AND, OR and NOT gates. The output of a NOT gate is powered if the input is not powered and vice versa, as demonstrated in figure 6. The output of an OR gate is powered if at least one input is powered, whereas an AND gate's output is only powered if both inputs are powered, which is shown in figure 8 and figure 7 respectively. Figures 7, 8, and 6 are analog to tables 1, 2, and 3 in chapter 3.2.1.

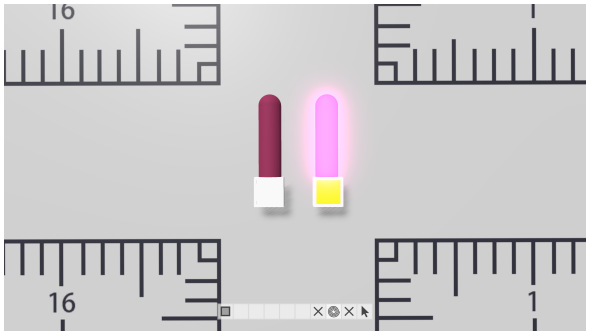


Fig. 5: Electric Switch

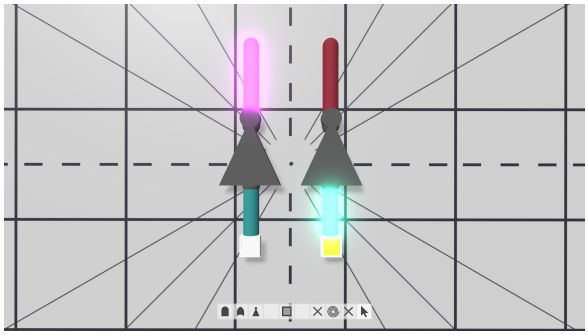


Fig. 6: Not Gates

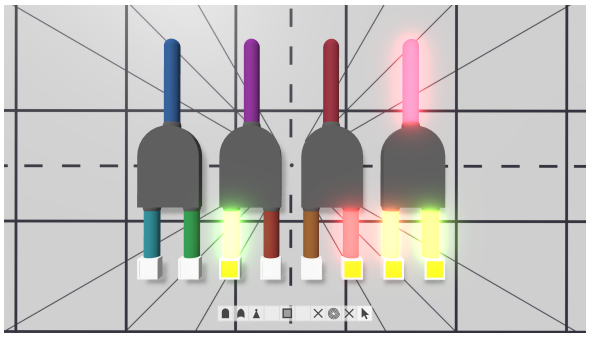


Fig. 7: And Gates

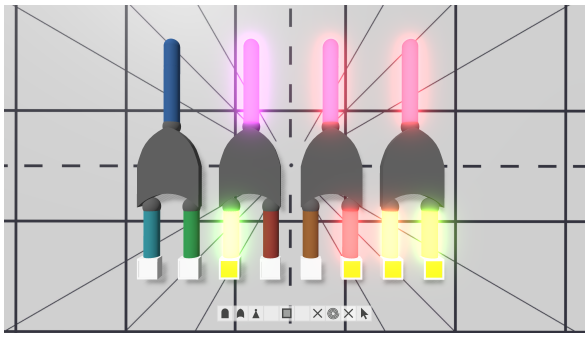


Fig. 8: Or Gates

5.2 Advanced Circuits

In this section, the creation of two electric circuits is described.

As mentioned in chapter 3.2.2, an sr latch can be used to store data digitally. It does so by having two inputs to the circuit, which is where the switches are connected to. Those two inputs are known as *S* and *R*, or *Set* and *Reset*, which gives the sr latch its name. From a logic standpoint, the circuit is symmetrical, which means that enabling the *S* switch has the same effect as the *R* switch, just flipped horizontally. The two inputs should never be powered at the same time, as this would cause undefined behavior. Turning on the switch on the right side, as shown in figure 10, caused the output of following OR gate to be powered and the output of the subsequent NOT gate to be not powered. The output of said NOT gate is connected to the OR gate on the left side, and since both

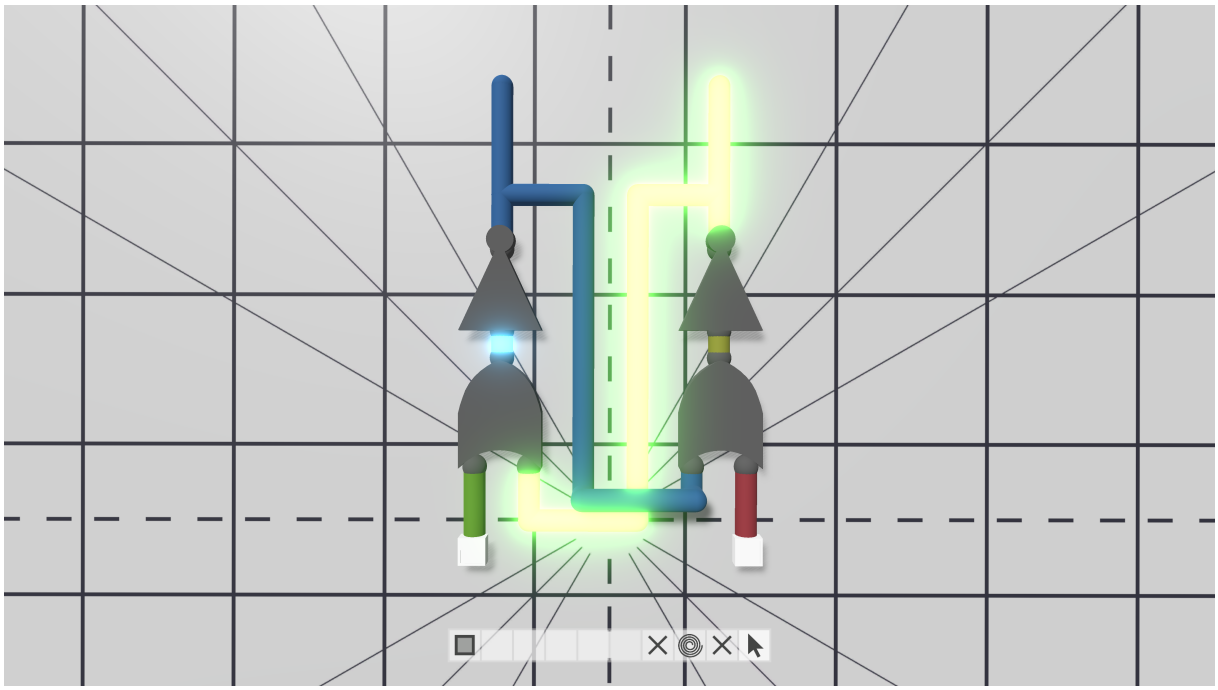


Fig. 9: SR Latch

circuit inputs should not be powered at the same time, it is to assume that the switch on the left side is not powered. Therefore the output of the left OR gate is not powered, whereas the output of the left NOT gate is. One input of the right OR gate is now powered and upon disabling the right switch, the outputs of the OR and NOT gates are the same, which causes the cables to keep the values, as seen in figure 11.

It is possible to do the aforementioned process on the left side, which causes the values of the cables to be inverted due to the circuits symmetry, which can be seen in figure 12 and 13. With this circuit, it is possible to store one bit, without external influences like the player. Additionally, other circuits can edit the value that is stored in the SR-Latch.

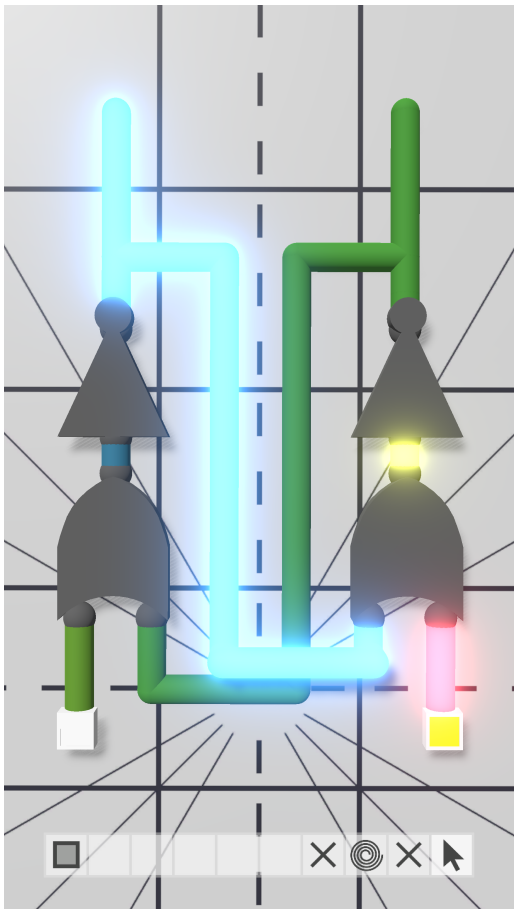


Fig. 10: SR Latch, right input powered

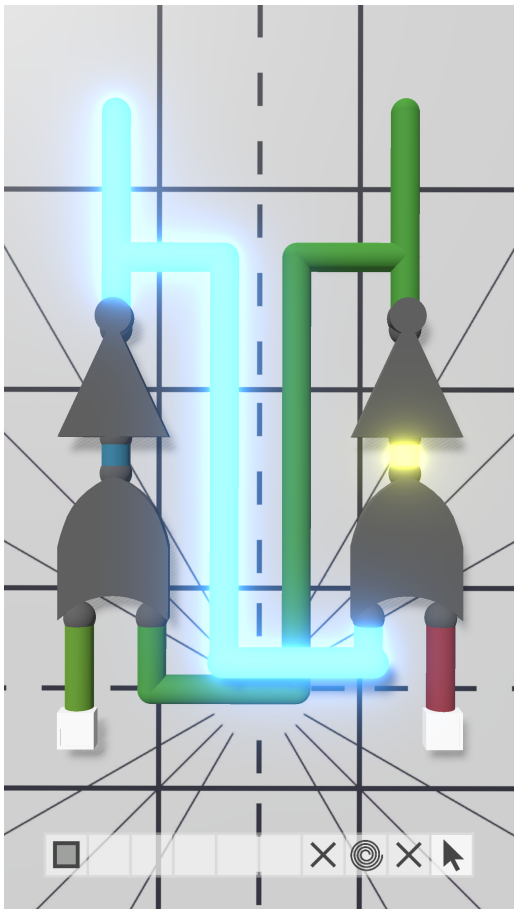


Fig. 11: SR Latch, no input, value 1

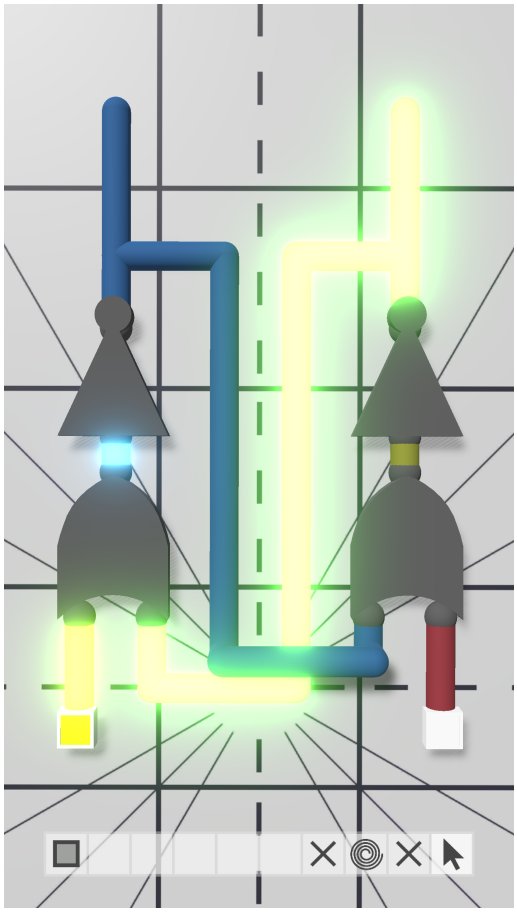


Fig. 12: SR Latch, left input powered

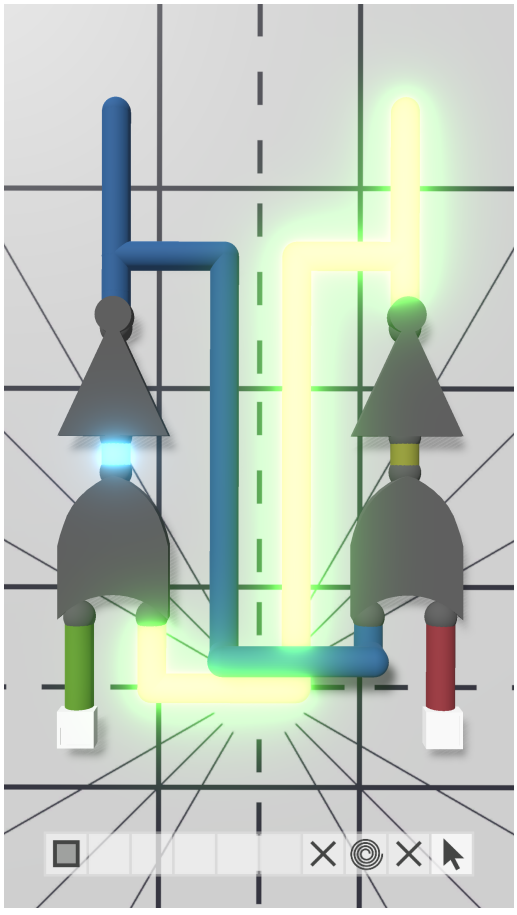


Fig. 13: SR Latch, no input, value 0

6 Summary

6.1 Expectation vs Reality

6.2 Problems

A lot of problems that were faced during the development of the game were related to Unity itself. Although this could be caused by using the newer 2018 version of the engine, instead of using the reliable version of 2017, it is still worth mentioning. One example for this would be problems with the hit-boxes. When creating an object from a script, the hit-box of said object often did not work properly and it would seem as though there was no hit-box at all. This was fixed by disabling and re-enabling hit-boxes when editing them.
Another problem with Unity were scripts that directly modified *Scenes*. Upon restarting the editor or starting the game, all changes that were made by the script were reset and caused a mild infuriation on the author.

6.3 Future Plans and Possibilities

6.3.1 Expansion or Finishing of Game

At its current state, the game is not finished. Although creating the game until now has already benefited the author by learning how games are created and the game-engine Unity works, it is worth considering properly finishing the game or expanding in content. A good example for a possible expansion would be cable-bundles that can contain an arbitrary number of cables but only need the space of one cable, therefore saving the player a lot of time by not having to create as many cables.

6.3.2 Launch

Making a game that is ready to be launched was the maximum goal to reach with this project. If it is decided to finish and polish the game, it is surely not a huge step to actually launching the game, so it goes hand in hand with finishing the game.

6.4 Advice for people with similar ideas

Personally, I learned not only a lot about making games but also that it takes a lot of work and effort to create one. Having prior knowledge in the field of programming, I would say that it was a lot easier for me to create a game than it would have been for a newcomer. The idea did not seem huge to me in the first, but it turned out to be. Due to this, I advise not to underestimate the effort needed to create one's dream game and to keep the project small and simple.

7 Acknowledgment

I would like to thank Jonas Luther for testing the game and giving constructive criticism. It helped a lot and made it easier for me to see the weak points of the game and fix them. Also, I would like to thank Pirmin Neyer, Jasmin Maier and Jonas Holzdörfer for proofreading the document and giving helpful suggestions concerning sentence structure and phrasing.

Bibliography

Blender Documentation. (2018). Retrieved from https://docs.blender.org/manual/en/dev/

C sharp (programming language). (2018). Retrieved from https://en.wikipedia.org/w/index.php?title=C_Sharp_(programming_language)&oldid=873966208

Eater, B. (2018). *Building an 8-bit breadboard computer.* Retrieved from https://www.youtube.com/playlist?list=PLowKtXNTBypGqImE405J2565dvjafglHU

GameObject. (2018). Retrieved from https://docs.unity3d.com/ScriptReference/GameObject.html

Kuphaldt, T. R. (2007). *Lessons In Electric Circuits* (Vol. IV).

Microsoft Docs - Reference for HLSL. (2018). Retrieved from https://docs.microsoft.com/en-us/windows/desktop/direct3dhlsl/dx-graphics-hlsl-reference

MonoBehaviour. (2018). Retrieved from https://docs.unity3d.com/ScriptReference/MonoBehaviour.html

List of Figures

List of Tables

Appendix

Appendix A: Code Snippets

Appendix B: Game Build

Appendix C: Declaration of Authenticity

A Code Snippets

DataStorage.cs

```
using System.Collections.Generic;

namespace ScenePersistentData {
    public class DataStorage {

        private Dictionary<string, object> data;

        public DataStorage() {
            data = new Dictionary<string, object>();
        }

        public bool Set(string key, object obj) {
            if (data.ContainsKey(key)) {
                data[key] = obj;
                return true;
            }
            data.Add(key, obj);
            return false;
        }

        public T Get<T>(string key, T defaultValue = default(T)) {
            if (data.ContainsKey(key)) {
                var value = data[key];
                if (value is T) {
                    return (T)(value);
                }
            }
            return defaultValue;
        }

        public bool HasKey(string key) {
            return data.ContainsKey(key);
        }
    }
}
```

PersistentData.cs

```
using UnityEngine;

namespace ScenePersistentData {
    public class PersistentData : MonoBehaviour {
        private static PersistentData main;
        private DataStorage data;
        [SerializeField]
        private bool usePrivateData = false;

        private void Awake() {
            data = new DataStorage();
            if (main == null) {
                main = this;
                DontDestroyOnLoad(gameObject);
            }
        }

        private DataStorage GetDataStorage() {
            if (usePrivateData) return data;
            return main.data;
        }

        public T Get<T>(string key, T defaultValue = default(T)) {
            return GetDataStorage().Get(key, defaultValue);
        }

        public bool Set<T>(string key, T value) {
            return GetDataStorage().Set(key, value);
        }

        public bool HasKey(string key) {
            return GetDataStorage().HasKey(key);
        }
    }
}
```

CableShader.shader

```
Shader "Custom/Electricity/CableShader" {
    Properties {
        [PerRenderData] _Color ("Color", Color) = (1,1,1,1)
        [PerRenderData] _Glow ("Glow", Range(0, 10)) = 0.0
        _Glossiness ("Smoothness", Range(0,1)) = 0.5
        _Metallic ("Metallic", Range(0,1)) = 0.0
    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 200

        CGPROGRAM

        #pragma surface surf Standard fullforwardshadows
        #pragma target 3.0

        sampler2D _MainTex;

        struct Input {
            float2 uv_MainTex;
        };

        half _Glossiness;
        half _Metallic;
        fixed4 _Color;
        float _Glow;

        UNITY_INSTANCING_BUFFER_START(Props)
        UNITY_INSTANCING_BUFFER_END(Props)

        void surf (Input IN, inout SurfaceOutputStandard o) {
            fixed4 c = _Color;
            o.Albedo = c.rgb;
            o.Emission = c.rgb * _Glow;
            o.Metallic = _Metallic;
            o.Smoothness = _Glossiness;
            o.Alpha = c.a;
        }
        ENDCG
    }
    FallBack "Diffuse"
}
```

B Game Build

The builds of the game are publicly available on Github and can be played online or downloaded. The online version is WebGL based and can also be downloaded. The Windows download works best, because the WebGL version has saving problems and the Linux download has performance issues.

`https://jm4ier.github.io/electrica`.

C Declaration of Authenticity

I hereby declare that the work submitted is my own and that all passages and ideas that are not mine have been fully and properly acknowledged.

Flums, May 5, 2019