Tapio Lehtimäki

Realization of 3D Environment and Control Console

Metropolia University of Applied Sciences
Master of Engineering
Information Tecnology
Master's Thesis

22 July 2024

PREFACE

In addition to the creation of the 3D environment and game console mentioned in the title, this thesis describes an exploration into the universe called Unity, which had been foreign land to the author until the year 2023. So this paper is not just a conceited account of successes — although it is that too — but a story about the learning process, which progressed largely through trial and error.

As much and as easily as information is available on almost any subject today, the actual development still seems to take place the hard way. Or as Bob Dylan (1) put it: "How many times must a man look up before he can see the sky?" My answer: "So many times to understand what is seen".

As the sole shareholder and employee of Tietomato Oy, even difficult cases have become familiar to me during more than thirty years of entrepreneurship. But still. While doing this job, I had to look up at the sky more times than I care to think.

Espoo, 22 July 2024
Tapio Lehtimäki

Abstract

Author:	Tapio Lehtimäki
Title:	Realization of 3D Environment and Control Console
Number of Pages:	83 pages + 7 appendices
Degree:	Master of Engineering
Degree Programme:	Information Tecnology
Professional Major:	Networking and Services
Date:	22 July 2024
Supervisor:	Ville Jääskeläinen, Principal Lecturer

This project was commissioned by Tietomato Oy to address two main issues: the skills gap in modern game development tools and the lack of experience in building custom controllers.

The company's current tools were deemed cost-ineffective, and their distribution channels were economically unprofitable. Expanding into the hardware sector was seen as an opportunity for a new business venture.

A 3D environment was created using the Unity Engine, with objects primarily modeled using Blender software. Additionally, a control console was developed to operate within this 3D environment. The project involved implementing a virtual environment and mastering the tools necessary for its creation, while also enhancing existing expertise. The environment was not intended to be finalized as a commercial product. Similarly, the control console aimed to expand expertise from mechanical devices towards embedded systems.

The thesis results demonstrate that both Unity and Blender successfully fulfilled their roles, and that Arduino-based hardware development is suitable for controlling applications created with Unity.

Keywords: Arduino, Blender, C#, C++, Programming, Unity

Contents

List of Abbreviations

1 Introduction 8

1.1 As an Apple Picker 8
1.2 Peeks Beyond the Fence 9
1.3 Dream of New Approaches 10
1.4 Strange Machines 11
1.5 Challenge 12
1.6 Objective 12
1.7 Structure of Thesis 13

2 World to Build 15

2.1 Blender 15
2.2 Photoshop 15
2.3 Unity 15
2.4 Arduino IDE 16
2.5 Cura 16
2.6 Develop Environment Hardware 16
2.7 Subject of the Scene 17

3 Game Object Modeling Workflow 20

3.1 Blender Modeling 20
3.2 Blender and Unity Compatibility 21
3.3 Skinning and Dressing 22
3.4 Connecting Parts 22

4 Peculiar Application Development 23

4.1 Component-based Programming 23
4.2 Fourth Dimension 24
4.2.1 Managing Flood of Messages 25
4.2.2 Cyclic Events and Particle Generator 27
4.2.3 CycleSwiz Class 28
4.2.4 Sounds 28

4.3 Summary 29

5 Rails and Bogies 30

5.1 Laws of Nature 30
5.2 Staying on Track 32
5.3 Direction and Position Corrections 33
5.4 Railroad Switches 35
5.5 Tracing Holes in Colliders 35
5.6 Programmed Gravity 36
5.7 Surprising Dimension 37

6 Different Angles 38

6.1 Euler Angles 39
6.2 Quaternions 40
6.3 Duel of Titans 41

7 Avatar’s Anatomy 43

7.1 Character Mobility 44
7.2 Nested Heads 45
7.3 Self-made Homunculus 47

8 Full Pixel Jacket 48

8.1 Pre-fabrication 49
8.2 Awkward trouble – Teleportation 50
8.3 Bullets are not Eternal 51

9 Environmental Effects 52

9.1 Immutability of Meshes 52
9.2 Impact Patterns 53
9.3 Through-Holes in Meshes 54
9.4 Disassembly of the Object 56
9.4.1 Convex Hulls 56
9.4.2 NP-completeness 58
9.4.3 Pruning of Options 59
9.4.4 In the Footsteps of Robinson Crusoe 60
9.4.5 Demolition Accomplished 61
9.5 Follow the Money 62

10 Control Console 63

10.1 Reactive Planning 63
10.2 Drafting of the Mechanics 64
10.3 Circuitry 67
10.4 Enclosure 70
10.5 Protocol 72
10.6 Operating Logic 73
10.7 Settings 74
10.8 Serialization and Deserialization 75
10.9 User Experience 77

11 Discussions and Conclusions 78

11.1 Cost-effectiveness 78
11.2 Achieving the Objectives 79

Reference List 80

Appendices
Appendix 1: Program Code of KeySwiz class
Appendix 2: Program Code of CycleSwiz class
Appendix 3: Rail Elements
Appendix 4: Code Snippet of the Bullet
Appendix 5: Placement Algorithm for Bullet Impact Marks
Appendix 6: Electronic Components of the Console
Appendix 7: Control Console’s Logic

List of Abbreviations

2D:	Two-dimensional.
3D:	Three-dimensional.
C:	C is a high-level programming language that provides low-level memory access and control, often used for system and application software development due to its efficiency and performance.
C++:	A general-purpose high-level C-derived programming language.
C#:	A general-purpose high-level C-derived programming language.
FFF:	Fused Filament Fabrication.
FPS:	First Person Shooter (game).
IDE:	Integrated Development Environment.
IoT:	Internet of Things.
MIT:	Massachusetts Institute of Technology.
NP:	Nondeterministic polynomial-time
Slerp:	Spherical Linear interpolation
UWTB:	Universal Will To Become
VAT:	Value Added Tax.

1 Introduction

This section describes the context, challenge, objective, and key tools, including the IT hardware, which are related to the goal of creating a three-dimensional environment resembling a game world and an electromechanical device called a control console used to manage events within that environment.

1.1 As an Apple Picker

Tietomato Oy first entered the game market in 2013 with its 3D game called CubeGo. The game was programmed with Apple's XCode IDE using the Objective-C language and C++ language with OpenGL commands embedded in it. The game, in all its modesty, was an Othello game board moved onto the surface of a cube that can be rotated on the iPhone screen (Figure 1).

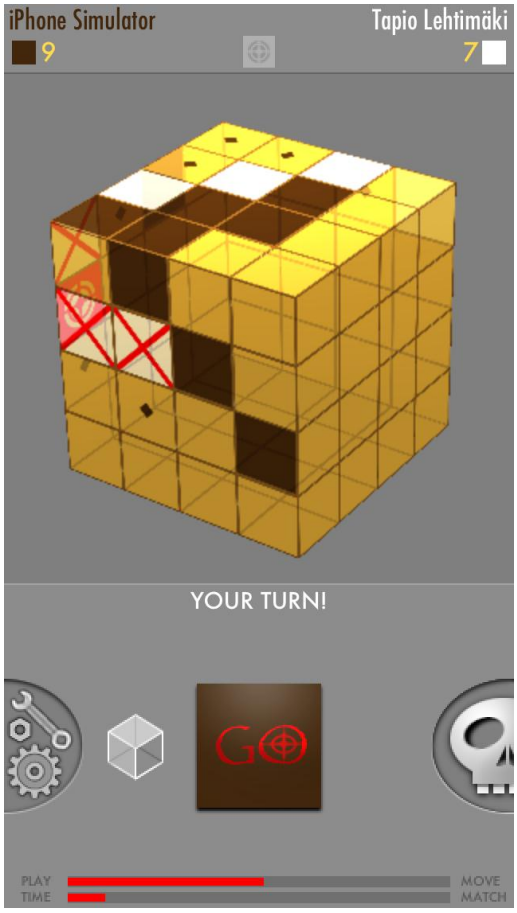


Figure 1.
Tietomato Oy's first commercial 3D game: CubeGo.

Programming even such a simple application with native tools was tedious. In practice, building a model with the tools of the time started with defining a point in an empty space. About half a man-year was spent writing CubeGo's source code. If it is compared to the customized office applications for the Windows platform produced by Tietomato Oy at the same time and the invoicing generated from them, the value of the work embedded in the game would have been roughly 50,000 to 100,000 euros.

Apps made with Apple tools run only on Apple devices and were distributed

only by Apple. Review rounds for program updates easily took weeks, and programs offered for sale were often rejected on arbitrary grounds. CubeGo was sold on the AppStore from August 2013 to September 2020 and generated a total of $23.35 after VAT. After eight years of persistence, the company parted ways with the fruitless partnership with Apple. The fact that Apple grabbed a 30% share of the sales revenue also had an effect on giving up the cooperation.

Even though Tietomato Oy's initial foray into the gaming industry was rocky, the business sector in question never stopped being interesting. The company felt that the biggest challenge was finding a programming tool that was more cost-effective than corporate fittings, and learning the skills required to implement one, if and when a suitable tool could be found somewhere. It would also be desirable to be able to produce distribution versions for different platforms from the same source code with little effort and enable the sale of products past monopolies that only monitor their own interests.

1.2 Peeks Beyond the Fence

In 2020, Tietomato Oy came across the free Godot game engine by MIT. There was also interest in another tool, Unity, which is free until the products made with it exceeds $200,000 turnover (2).

Figure 2.
2D game demo from 2020 made with Godot.

The first instrument was chosen in alphabetical order as a familiarization object. Godot's learning threshold turned out to be low and in a month it was possible to produce a small tank demo (Figure 2). The bar was not set

too high, and the demo was made in 2D mode, but still Godot's performance was not convincing and Tietomato Oy continued to Unity.

However, praised as easy, Unity was hard to approach. It was impossible to catch with own hands, and the help videos made by enthusiasts did not encourage to continue learning about the subject. The final conclusion was that Godot is weak and Unity is difficult.

1.3 Dream of New Approaches

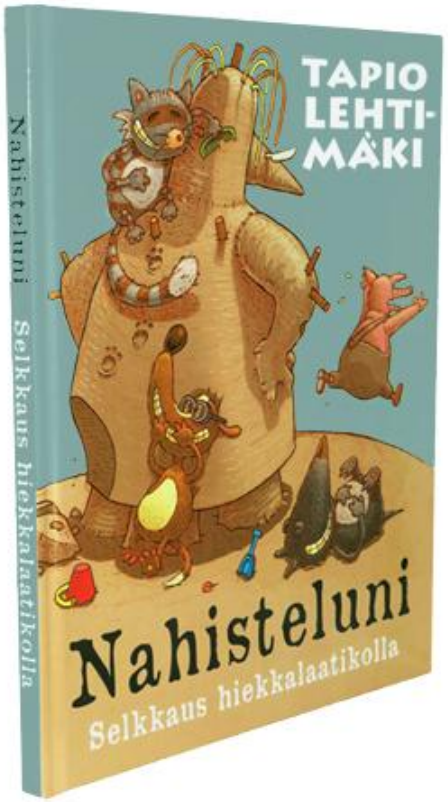


Figure 3.
The first book published by Tietomato Oy

In 2022, Tietomato Oy expanded its operations to the publishing industry (Figure 3). Aware of the plight of print media and dissatisfied with the prevailing digital costs, the company began to investigate the possibility of publishing and distributing written and graphic material in a way different from the common digital formats of books.

The Metropolia Viope course on Unity game programming completed by the author of the thesis in the summer of 2023 was an eye-opening experience. So it was not love at first sight, but at the second glare. With it, Tietomato Oy brought back the once-buried dream of game programming into the daily routine. The traditional game business is not the only goal of the company's Unity production. The ability to cost-effectively produce game-like applications that are not actually games is an equally important goal. Books could perhaps be digitized with Unity too.

1.4 Strange Machines

If the Unity course was eye-opening, another similar one was the IoT course of Metropolia's IT Master's program, which involved building an embedded system. With the capabilities obtained from the course, it is possible to make connections to a control device that would make it more intuitive to steer vehicles in a virtual world.

So, the company manages well with electronic circuits, but manufacturing mechanical machines is a completely different challenge. At this point, it is necessary to highlight Tietomato Oy's long-standing traditions in the field of specialized devices. It must be acknowledged that the contraptions produced by the company somewhat resemble the crazy devices of Bonk Business Inc, but unlike those beautiful yet invariably non-functional contraptions (3), Tietomato's machines usually serve some purpose, which they fulfill with varying degrees of success (Figure 4).

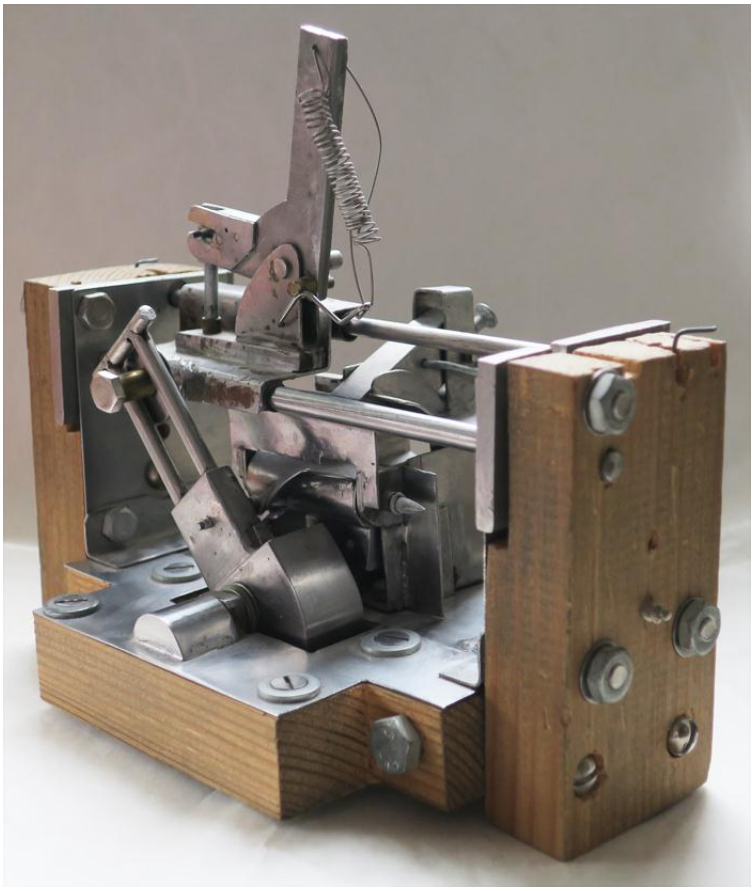


Figure 4.
Feed mechanism for a nylon plug-firing machine gun from the 1990s, made of wood and steel.

The company's level of expertise in the field of mechanical engineering seems to be roughly equivalent to that of the boggies in the Tolkien parody Bored of the Rings: “They don't like machines more complicated than a garrote, a

blackjack, or a luger” (4 p6). What can be achieved from these starting points will be seen.

1.5 Challenge

Tietomato Oy feels that the development tools it has used until now are cost-inefficient in producing computer games and narrative material. Likewise, the distribution of the material in question has been difficult and the sales income modest in ecosystems controlled by large companies.

It has therefore not been possible to manufacture the products quickly enough with the company's current equipment, nor to distribute them effectively to the market.

Better development tools have been available for some time, but the company did not know how to use them. It was also difficult for a long time to decide which tool should be invested in studying.

Controlling characters in the virtual world with a keyboard and mouse is not necessarily very user-friendly. Different steering wheels and joysticks, on the other hand, seem to overshoot this need and, aimed at the mass market, are inevitably compromises between general usability and a solution designed for a specific purpose. In addition to all that, the main construction material of commercial wheels, sticks and game platforms is plastic, and they break down with intense use.

1.6 Objective

Unity and Blender, which have been tested in preliminary trials and through supplementary online courses, will be incorporated into Tietomato Oy's competence repertoire.

The suitability of the aforementioned tools for Tietomato Oy's development needs will be ensured by creating a small-scale virtualized 3D environment where is possible to move around in some kind of vehicle and act as an avatar character. In other words, a game-like world is put together with Unity, the models of which are primarily implemented with Blender and whose "forces of nature" are mainly handled by Unity's physics engine. In addition to the laws of physics built into Unity, characters in the environment are controlled and phenomena are manipulated with scripts written in C#.

Application development in an environment like Unity is called the component-based game programming model and it differs in certain aspects from the models of traditional office application programming. The differences between the models and also their convergence points are explained in their essential parts. The key points of the development stages will be documented in terms of Unity, Blender and programming to clarify the workflow and problem solving methods.

A record is also kept of the difficulty of manufacturing software components. Since it is a pilot project where new work tools are introduced, significantly more working hours will be wasted than in a more familiar environment. Therefore, instead of staring at an hourglass, it is examined how many lines of source code are needed to implement certain functions.

In addition to the programmatic part described above, a small but solid control console is built to facilitate the use of the application.

1.7 Structure of Thesis

This thesis is divided into eleven sections. The first section introduces the problem and context, while the second section describes the 3D environment that was modeled and the tools used for the modeling process.

The third section outlines the character modeling workflow, followed by the fourth section, which details the software implementation. The fifth section explains the functionality of the game track.

In the sixth section, the orientation of objects and the challenges of alternative angular notifications are discussed.

The seventh section examines the structure of the avatar, the eighth addresses the management of fast-moving objects, and the ninth focuses on damage modeling.

The tenth section discusses the implementation of the control console, and the eleventh section presents discussions and conclusions.

2 World to Build

Before delving into the details, it's essential to introduce the tools utilized to address the business challenge. The 3D world primarily relies on three key tools: complex models have been created in Blender, adorned with textures designed in Photoshop, and ultimately brought to life in Unity using Unity's physics engine and the C# programming language.

The console device is an embedded system and the control code it requires was written in C++ in the Arduino IDE.

2.1 Blender

Blender is free and open source 3D creation suite. It supports the entirety of the 3D pipeline — modeling, rigging, animation, simulation, rendering, compositing and motion tracking, even video editing and game creation. There are official releases of Blender for Windows, MacOS and Linux. (5)

Blender version 2.80 was harnessed to create the characters for this project.

2.2 Photoshop

The most well-known of the tools used in the final work is probably Adobe Photoshop. It is a raster graphics editor developed and published by Adobe Inc. for Windows and macOS (6).

Photoshop version CS5 was used for this project.

2.3 Unity

Unity is a cross-platform game engine developed by Unity Technologies. The engine has since been gradually extended to support a variety of desktop, mobile, console and virtual reality platforms. It is particularly popular for iOS and

Android mobile game development, is considered easy to use for beginner developers, and is popular for indie game development. (7)

The engine can be used to create 2D and 3D games, as well as interactive simulations and other experiences (8). The engine has been adopted by industries outside video gaming, such as film, automotive, architecture, engineering, construction, and the United States Armed Forces (9).

Unity version used in creating the 3D world of thesis was “2018.4.16f1 Personal”.

2.4 Arduino IDE

The version of the Arduino IDE used to program the microcontroller was 2.0.3. The programming language used with Arduino is based on C++.

2.5 Cura

Ultimaker’s Cura, an end-to-end solution for fused filament 3D printing was used as a slicer when Blender meshes were printed for the game console. The version of Cura was 3.6.0.

2.6 Develop Environment Hardware

In addition to the applications mentioned above, the project naturally also needed hardware. The computer was an ancient Mac mini of the year 2012. It has a 2.6 GHz Intel Core i7 processor, memory of 16 GB 1600 MHz DDR3 and an SSD hard disk. The operating system is MacOS Sierra, which is also left by time.

In the development phase of the console, a full size Arduino Uno R3 board was used, but a tiny ATMega328P deployment board was implanted in the final device.

The 3D prints needed for the console were made by company's old faithful Ultimaker 2+. Instead of the modern resin filament machines it is a FFF device, which does not reach as accurate resolutions as resin printers, but is still very reliable and materials of which are quite low-priced.

A computer that is more than ten years old is understandably weaker in terms of performance than modern devices. One bright side of using old device is that the software has to be carefully optimized or else it freezes the system – and on newer platforms, the program is guaranteed to run smoothly. Another thing worth noting is that both Unity and Blender are not resource-hungry like many commercial products, but instead work perfectly on different platforms of different ages.

And as an old Finnish folk wisdom says "Poverty is not joy, even if laughter sometimes occurs". Regardless, these tools put a smile on the face of a poor indie game developer this time.

2.7 Subject of the Scene

So the construction tools were clear, but what kind of world would be built with them? Or better put, Why did it end up that a belligerent roller coaster was built?

The fields of computer games are invariably finite. The borders of these worlds are usually invisible walls, bottomless chasms or impassable barricades. The train track-like solution takes care of demarcating the field by itself. The user's wandering in the terrain does not need to be restricted by artificial obstacles.

Although the company's staff had zero knowledge of Unity when the Master's degree program started, it was no longer the case after taking the Unity game programming online course. In the course in question, were operated, among other things, with a flying device and as a final exercise a small car game was made (Figure 5).

Figure 5.
Modeling the car's kinetics in Unity was surprisingly more difficult than getting a device with the aerodynamics of a brick to stay in the air.

Due to the strange legalities of the Unity Engine, it was easier to make a functioning aircraft than a ground vehicle. In a way, the roller coaster combines the aforementioned forms of movement: flying and traveling on land. After all, the cart travels both vertically and horizontally, but its route is nevertheless tied to the tracks. And unlike in the real world, that bond is more difficult to achieve in Unity than unrestricted movement in space or speed driving in the open.

A piece of rail (Figure 6) lying in the company’s basement also served as a source of inspiration. Something in its rusty intransigence probably appeals to everyone's artistic side.

Figure 6.
A piece of rail whose cross-section dimensions were used to make a Blender model.

“We choose to go to the moon in this decade and do the other things, not because they are easy, but because they are hard”, said President John F. Kennedy (10). In the same spirit, it was decided to create a world woven of rails, populated by gravity-defying carriages rattling on steel wheels.

Not because it seemed easy, but because the difficulty was underestimated.

3 Game Object Modeling Workflow

When building an object for a game, the division of labor between previously mentioned tools usually starts with Blender, takes off from Photoshop and ends with Unity. The latter also provides its own set of three-dimensional primitives, such as spheres, cones, cubes, planes, and others. These primitives can be edited in Unity primarily by scaling, stretching, and flattening their meshes. Blender, on the other hand, was originally made specifically as a modeling tool. There, only imagination limits the possibility of modifying meshes.

That's why, in practice, all the models of this work were made in Blender, from where they were transferred to Unity.

3.1 Blender Modeling

The subject to be modeled can be approached in Blender in many ways. A large number of the models were made more or less creatively by improvising shapes while making the figure. However, it is possible – as strange as it sounds – to “draw through” characters from two-dimensional photographs. This is exactly what was done with the Lewis rapid rifle of the First World War (11 p113) during this thesis. The technique is especially suitable for objects that imitate reality.

The Lewis model contains about 11,500 vertices, and it is the largest single object in the final work. It has also some active parts, controlled by Unity's C# script. Not only does it look authentic, but it also replicates its prototype relatively faithfully with a moving trigger, bolt, charging handle and rotating magazine.

First, pictures of the topic from the side and above were imported into Blender. Based on them, the dimensions were measured correctly and the different parts were set in their correct places. The Vintage Aviator website's photos (12) of the

Lewis Gun Mk2, which was the version of that weapon for aircraft use, served as reference images (Figure 7).

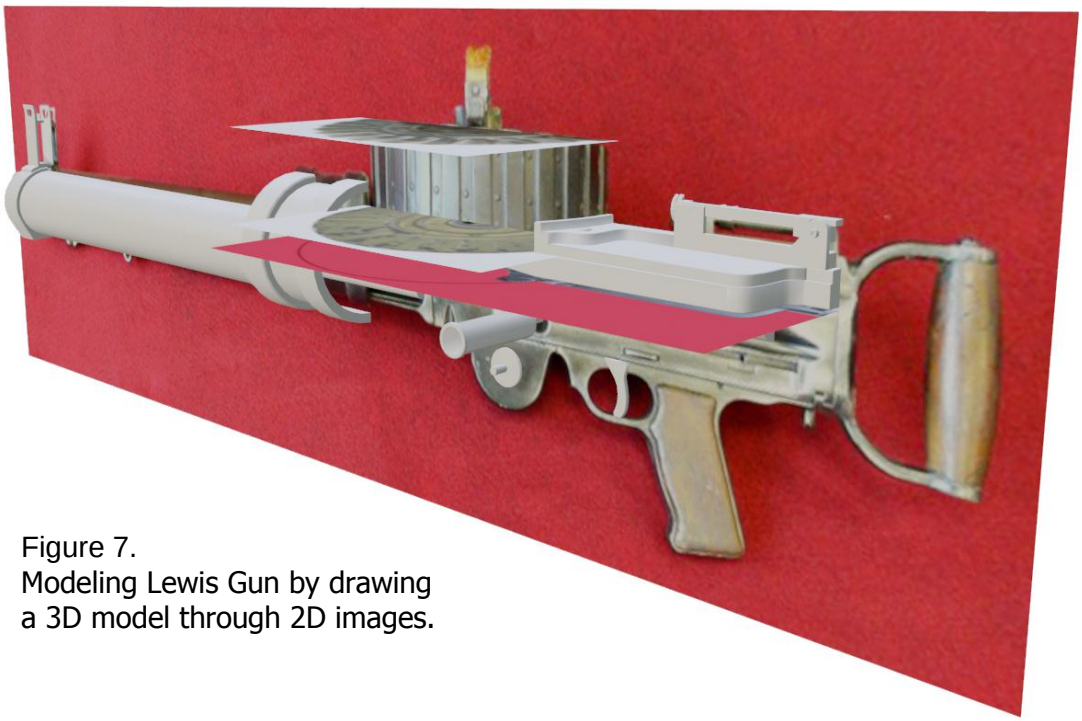


Figure 7.
Modeling Lewis Gun by drawing a 3D model through 2D images.

3.2 Blender and Unity Compatibility

Once the figure is finished, it needs an upholstery. Blender and Unity have their own ways of defining materials, and not all their properties may remain unchanged when moving from one tool to another. Materials that are simply defined in Blender with RGB colors (Figure 8) transferred without problems in the experiments, and so did UV textures.

Photoshop's default format (.psd) is fine for Blender as it is, and Unity understands .blend-files made with Blender in addition to .psd-images. It is therefore not necessary to

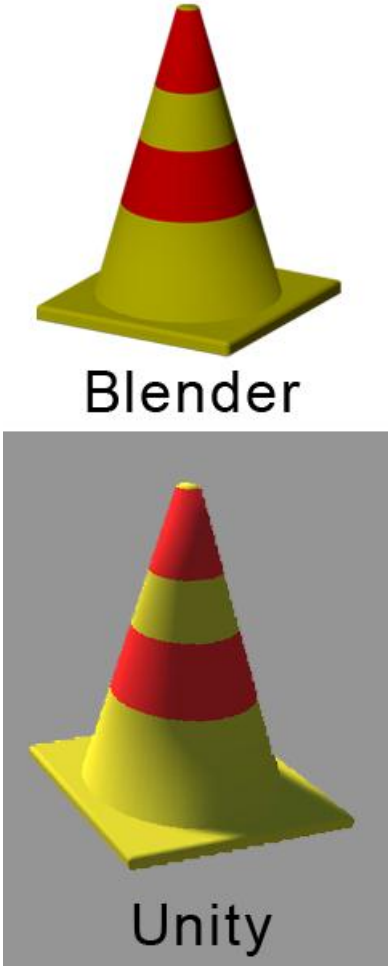


Figure 8.
The differences between the objects coated with simple color materials in Blender and Unity are mainly due to the lighting.

export images and meshes to more universal file formats.

3.3 Skinning and Dressing

A significantly more challenging method than color coating is upholstery based on UV maps, where the Blender model is first skinned into "sewing patterns", which, when adapted to a two-dimensional image texture, produce a coat over the object. The UV maps of the roller coaster world were made with Photoshop by drawing by hand and editing photos (Figure 9).

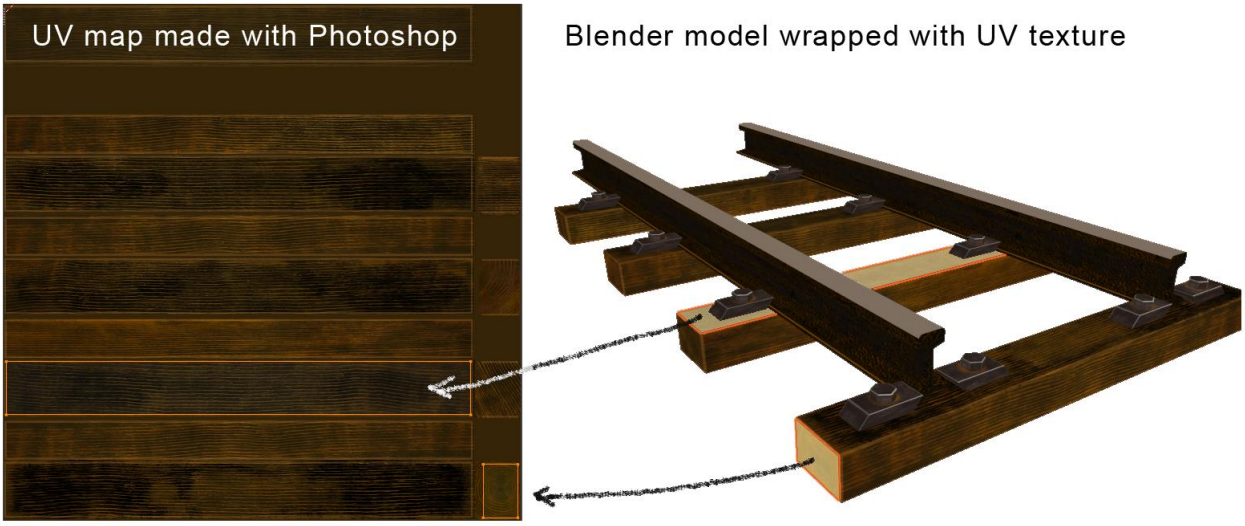


Figure 9.
The picture illustrates how the parts of the two-dimensional UV map cover the three-dimensional model.

3.4 Connecting Parts

In Unity, a playing field is assembled from pieces created with Blender and covered with Photoshop. There, the whole is made greater than the sum of its parts. The game is brought to life with programming code that, in Unity, follows the component-based game programming paradigm; more on that in the next section.

4 Peculiar Application Development

Programming in general may be considered a peculiar activity, but the experience is highly subjective. If one is used to working with office applications, game programming will certainly surprise with some of its unique features.

4.1 Component-based Programming

Component-based software development is not a new invention. For example, Tietomato Oy's most productive programming tool, the Delphi Developer, already launched in the 1990s, is strongly based on components that could be used to build user interfaces, database handlers, connections to the network and so on (13).

In Delphi, the components and their event handlers were by default combined into a main program body presented in a single source code unit, which the programmer was free to modify. In Unity, such a visible source code frame is not shown to the developer, but the C# language scripts written in the components are separate entities.

As modern a tool as Unity is, it has a faint hint of the procedural programming model of the past. The main loop of yesteryear, also known as the game loop, is almost visible (Figure 10). A direct quote on the Game Programming Patterns website (14) describes that secret entity beautifully:

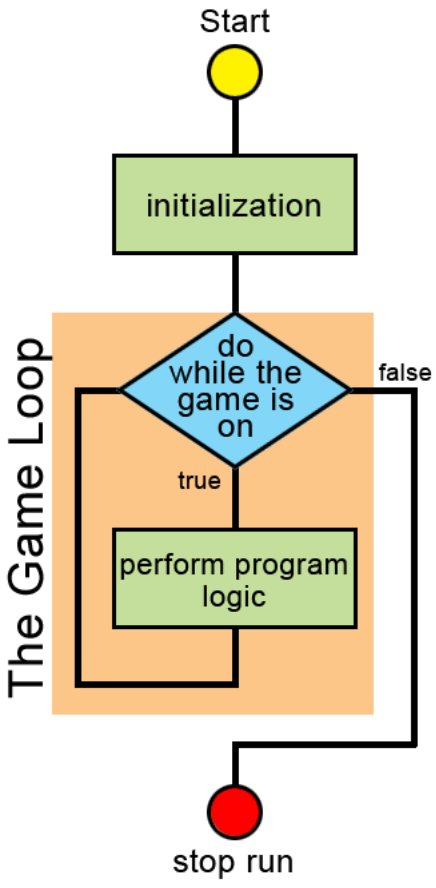


Figure 10.
The mysterious game loop of a program in all its simplicity.

“Game loops are the quintessential example of ‘game programming pattern’. Almost every game has one, no two are exactly alike, and relatively few programs outside of games use them”. (14, game-loop.html)

Each of so called “MonoBehaviour” class type representatives can be connected to the hidden game loop of the main program for example with the FixedUpdate method, which is called by the main thread of the running program 50 times per second. These scripts are also able to communicate with each other.

The idea of such a game loop is foreign to traditional office application development, even though it is commonplace in game programming.

It is also special that a script attached to an instance of the GameObject class sort of takes over its host and seemingly retypes it. Of course, the owner of the script still remains a GameObject, but it appears to scripts attached to other components as an instance of the class defined by its own script component.

Unity manual says, that “Unity’s Game Object class represents everything which can exist in a Scene” (15 class-GameObject.html). A game object can be composed of several components, and each game object also acts as an assembling host class for its subordinate child game objects, if necessary. Such child objects can be added directly in the hierarchy view of Unity's user interface, but nothing prevents the creation of corresponding member objects at runtime from a script. The latter method is familiar to conventional software culture too; more on that in section 8 and prefabs.

4.2 Fourth Dimension

The aforementioned game loop brings an extra dimension to 3D programming: the time. In office applications, the developer does not need to consider the flow of time. In game programming, it is essential. In the attached picture, the Lewis gun made with Blender has been brought to life. It spews shells, flames, and

puffs of smoke (Figure 11). Next, what it required from the program code will be examined.

Figure 11.
Blender made Lewis gun in action. It is made possible by the particle generators and the author's KeySwiz and CycleSwiz classes.

4.2.1 Managing Flood of Messages

When implementing something as simple as turning the lights on and off by pressing a button on the keyboard in traditional programming environments, the operation in question triggers a “key down” event, and the event handler then turns the lights on or off depending on their current state.

In Unity game programming, the approach is different and the state of the key is caught in the "FixedUpdate" method attached to the game loop. Even if the user just touches the key lightly, the loop that reads the key's status 50 times per second has time to generate several press-down messages from the event. Similarly, not pressing will generate an infinite number of messages to the contrary.

So the lights would flicker wildly and it would be up to pure chance whether they would eventually stay on or whether they would turn off as a result of pressing the switch.

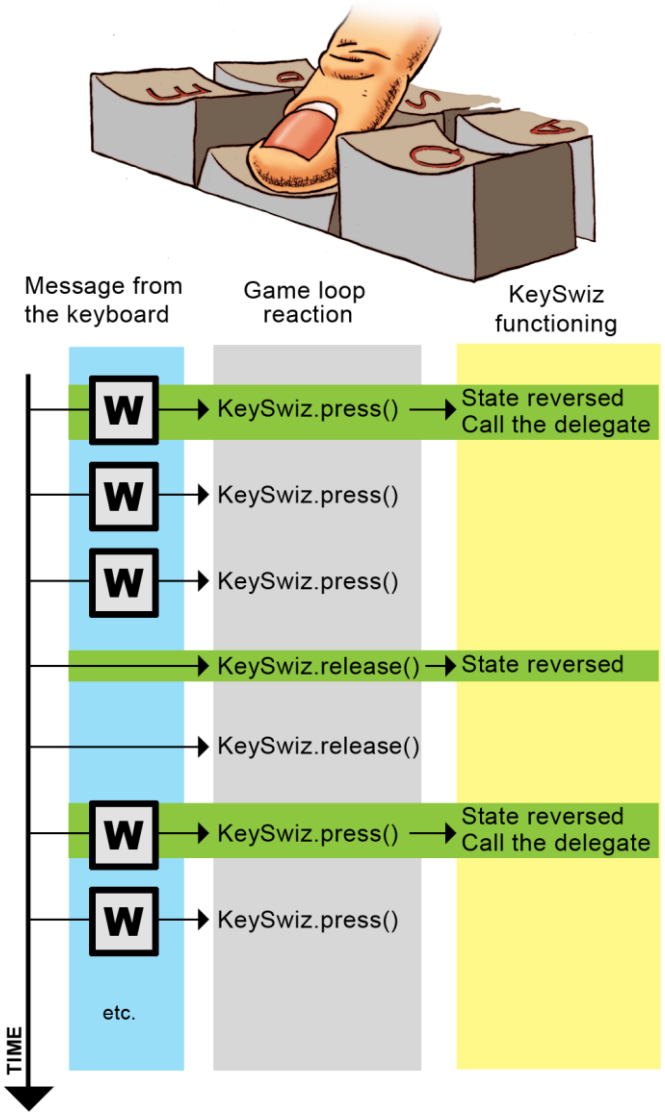


Figure 12.
KeySwiz object processing the input stream from the keyboard. Those marked in green pass the sieve.

To manage these situations, the KeySwiz class was built. An instance of it can be bound to the chosen key. When the key is down, the method "pressed" of the KeySwiz instance is called, and when it is up, the method "release" is called (Figure 12). The self-made switch object filters noise messages by reacting only to changes in keyboard button's state. That is, it calls the assigned delegate only at the first time of the series of presses. Next, it changes its state only after receiving the first message from the user's finger leaving the button.

For example, the control of the carriage's headlight and the selection of the active camera were implemented with instances of the KeySwiz class. The program code of the class can be found in appendix 1.

4.2.2 Cyclic Events and Particle Generator

To manage repetitive visual events in Unity, there is an ingenious component called “Particle System”. It can be used to create fog, splashes, a campfire with playful flames, and so on. The particle generator uses the so-called “Sprite Atlas”, i.e. a set of two-dimensional images, from which it chooses one image at a time to be presented. Unity's particle generator has numerous parameters (Figure 13) that can be used to adjust each image's life cycle, direction of travel, frequency of occurrence, translucency change, growth or shrinking rate, etc.

In practice work, the shells and smoke flying from the gun have been implemented with the Particle System.

The basic idea of particle generators is to produce the illusion of a three-dimensional phenomenon with two-dimensional images. These little information technical rascals, the sprites, are always turned face-to-face towards the camera. So they never reveal their subtle essence to the viewer. The technique also lightens the calculation load, as the shape of a sprite can be defined with four vertices, while even a simple 3D object usually has at least eight vertices, or even dozens.

Figure 13.
Unity’s Particle System component has dozens of adjustable parameters.

Figure 14 shows a sprite composite of Lewis rifle shells. In the sample work Unity's particle system was also asked to cast shadows from the particles, which increase the feeling of three-dimensionality.

When needed these particle systems can be activated or deactivated from a C# script with just one command.

Figure 14.
Shell sprites made for one of the particle systems used in Lewis gun model.

4.2.3 CycleSwiz Class

The particle system is an excellent tool for visual enjoyment, but it does not bend to the cyclicality of logical operations. For the steadily pulsating turn signals of a vehicle or the self-loading mechanisms of machine guns, the Cycle Swiz class was developed. It is somewhat similar to the previously described KeySwiz class.

The main difference between them is that CycleSwiz always completes the set of cycles defined for it. It calls its delegate as many times as its cycle variable determines. This prevented the Lewis gun from leaving the bolt and charging handle halfway and the muzzle flame burning when the trigger is released. The program logic of the class is presented in appendix 2.

4.2.4 Sounds

Shooting is notoriously noisy, so a component that enables sound playback was added to the machine gun class. An audio source member object with the Lewis_shot.wav file, which represents a single shot, was added to the Lewis class, transforming the silent machine gun into a loud chatterer.

When the user presses the trigger (i.e., the left mouse button), the shooting sound is played in a loop until the user releases the trigger.

4.3 Summary

In the hourglass of office applications, the sand is in second-sized particles, in game programming the grain size is in the order of milliseconds. In the latter, it is as straightforward to connect cyclic actors to the flow of time as it is to plant a pinwheel in a stream. In the first mentioned, the flow of time is hidden and such regularly repeated operations require special measures – in them the impeller needs a timer object as its motor.

Designing the user interface of so-called utility software is like painting a canvas with oil paints, while in the over-emphasized visuals of game programming, it is like directing a movie. However, the principles of programming are the same in both, and the step from desktop applications to the world of game programming is not as daunting leap as one might think.

5 Rails and Bogies

In the virtual railway world, the tracks, as might be expected, play quite a role. Even a small test track requires a surprisingly large amount of rail. In the real life the iron parts of the rail are practically identical to each other to the point of monotony, and the sleepers are not exactly different from each other either. It was therefore worthwhile to transform their virtual counterparts into modular units both to simplify the construction of the track and to save computing capacity. It is more cost-effective to assemble a railway track from a single preloaded template than try to make each inch a unique railroad.

A modular track consists of elements with lengths of 1—8 sleepers, from which the desired type of roller coaster can be freely assembled. List of different rail pieces made for the track is in appendix 3.

5.1 Laws of Nature

At first, the cart was tried to make run on the rails with flanged wheels and Unity physics. It happened, however, that in a curve the wheels did not turn the axle in the right direction, but instead got stuck or wobbled when trying to do so.

There are laws of nature in Unity, but they are not exactly the same as in the real world. Although they work well with an object like a bowling ball placed on an inclined surface, more complicated technical constructions do not automatically start working with those rules. Unity in all its wonder sometimes resembles witchcraft, and a nursery rhyme reminds how magic works with mechanics:

You see,
the power of a witch
does not work with machines
nor with engines
nor with switches

nor with screws
nor with gears
nor with pedals
all in all:
a machine has the rhythm of a machine (16).

Understanding this fact proved to be the beginning of wisdom in Unity. Therefore, the carriage was changed to kinematic, which means that the control of the object is transferred from the Unity Engine to the shoulders of the programmer. At this point, the location measurements of the axles were decided to do in the Unity way — with the help of rays.

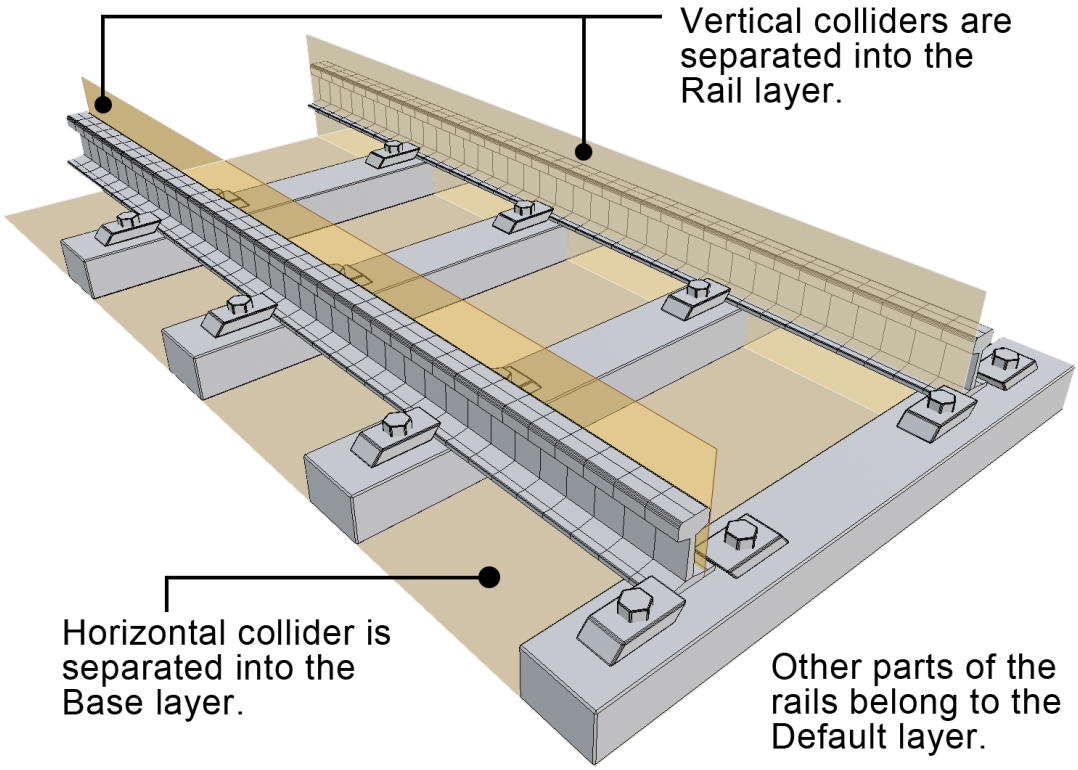


Figure 15.
In the structure of the rail pair, the need for colliders that keep the cart on the tracks has already been taken into account in Blender modeling.

This caused changes in the structure of the rails too. Colliders were added to them in Blender, from which virtual sensors measure axle direction errors to be corrected programmatically (Figure 15). In the final game, those colliders will

not be seen, because they are marked as unrendered. They still serve while remaining invisible to everything except the measurement beams sent 50 times per second by the axle sensors suspended in the game loop.z

5.2 Staying on Track

The rotating cornerstone of the game is a bogie. A cart has two bogies and five sensors are connected to each of them, which emit ten measurement beams in each iteration of the game's update loop (Figure 16). The bogie axle turns independently according to the shape of the track. The front bogie of the cart is dominant in such a way that a shift command is sent to the rear bogie if the rear axle lags behind or shortens the permitted distance between the axles.

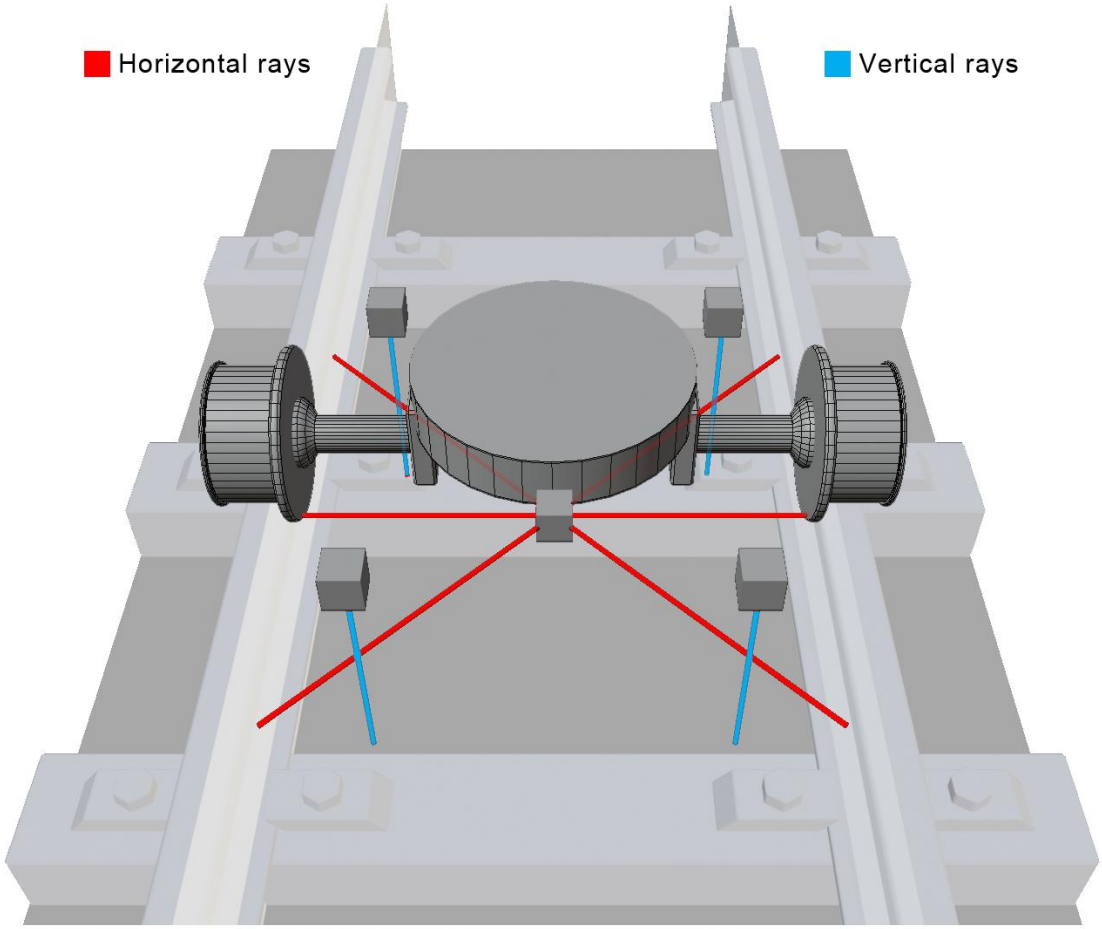


Figure 16.
Each bogie is an independent system that measures its position and orientation with multiple rays sent by sensors in each iteration of the game loop.

5.3 Direction and Position Corrections

There are five types of position errors on the bogie. The first of these is directional error, which is usually caused by horizontal bends in the track (Figure 17).

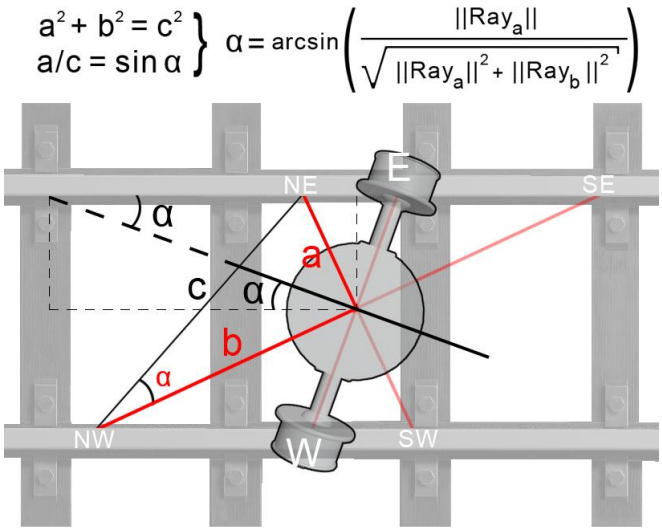


Figure 17.
From the length differences of the rays sent diagonally in the direction of travel, the correction angle for the direction of the axle can be obtained using the Pythagorean theorem and the arcsine.

The second and third deviations are lateral and vertical displacements. The axle must stay in the middle of the pair of rails and it must not rise into the air or sink below the rail surface (Figure 18).

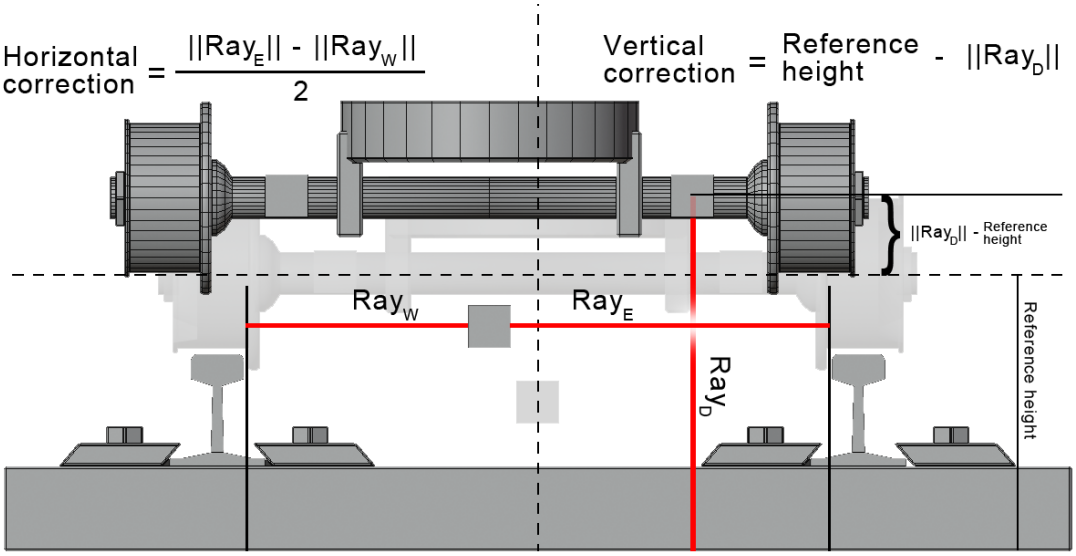


Figure 18.
The horizontal shift is corrected by the average length difference of the rays sent horizontally in opposite directions. The height error is corrected by the difference between the length of the vertical beam and the reference height.

Fourth, the lateral inclination of the axis is corrected (Figure 19).

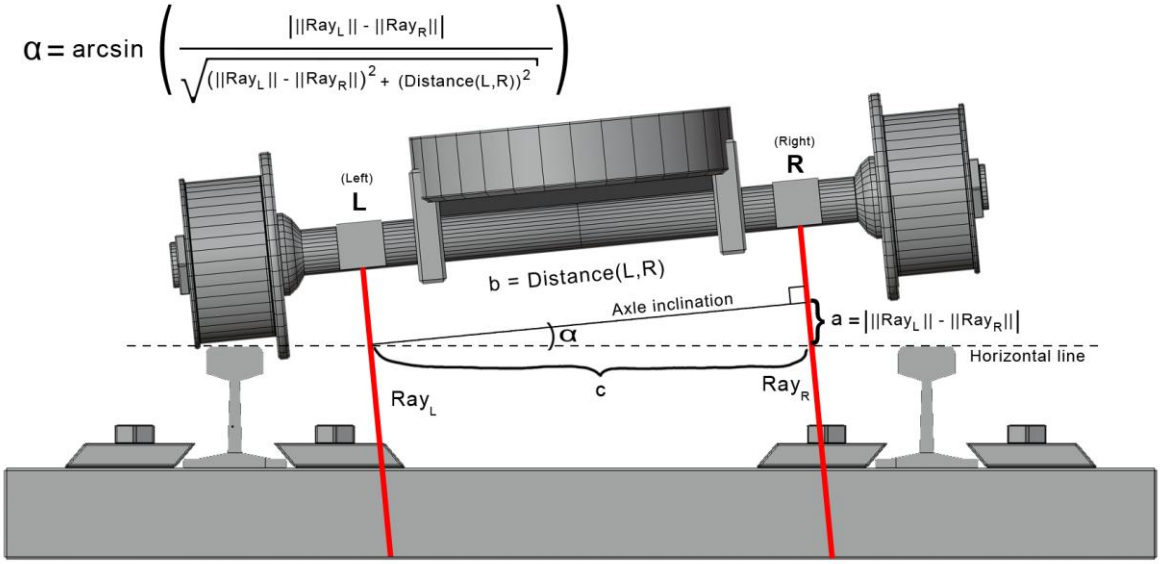


Figure 19.
The lateral inclination correction angle is obtained from the measurement difference of the opposite vertical rays using the Pythagorean theorem and arcsine.

The fifth type of error is tilting forward or backward, which occurs on uphill and downhill slopes (Figure 20).

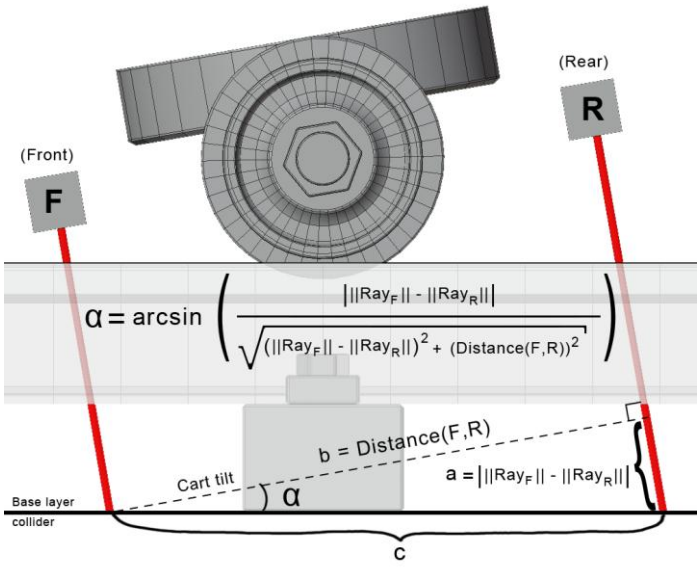


Figure 20.
The so-called somersault correction is calculated from the difference between the measurement radii of the vertical front and rear sensors using the Pythagorean theorem and arcsine.

5.4 Railroad Switches

The switches are implemented with branching track modules, which have both a straight and a bend pair of colliders (Figure 21). Which one is activated depends on which route the user chooses.

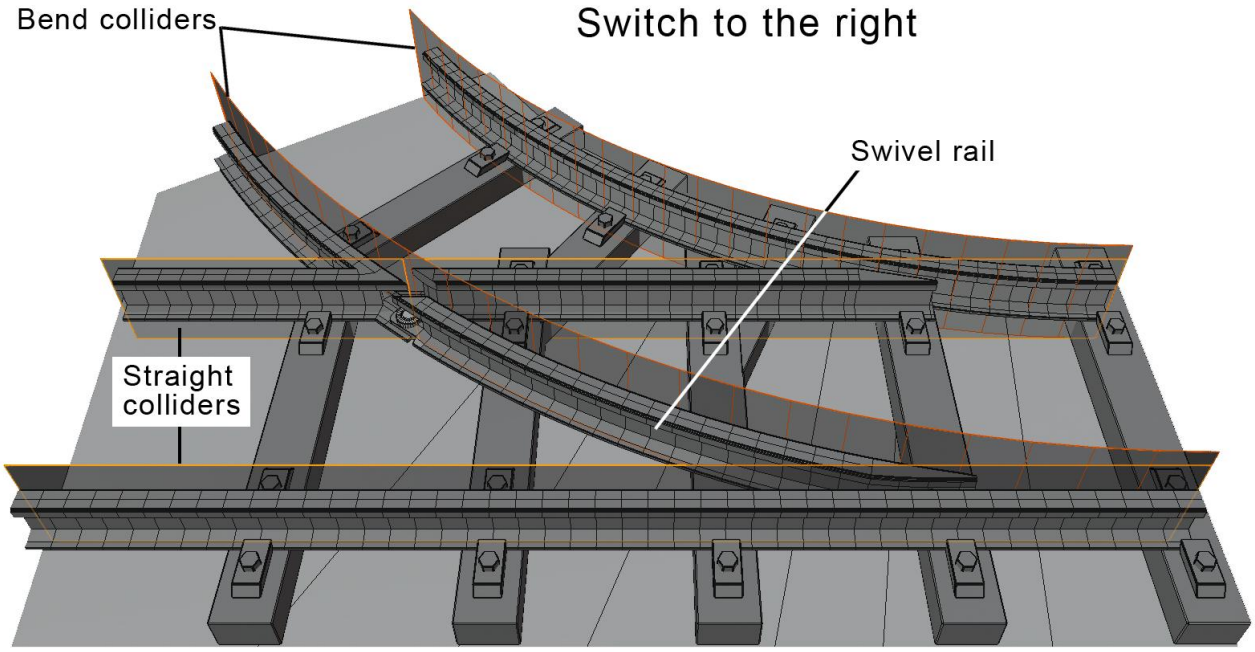


Figure 21.
The gear module has an extra vertical colliders and a functional swivel rail, the meaning of which is decorative. The cart is controlled only with colliders.

5.5 Tracing Holes in Colliders

Colliders, especially two-dimensional plane type flat meshes, are not 100% reliable. For example, flat rail colliders gave over 99% response at reasonable speeds in a test run; 103,657 measurements produced 9 incorrect results when the beam passed through the collider. At high speeds, the cart flew sometimes off the rails. Convex colliders produced only two incorrect results out of 105,709 measurements and the cart stayed better on track despite the mistakes. All these results can be considered good.

Derailments are most likely caused by missing colliders or gaps between colliders on track sections. To find them, a pair of axles flagged in debug mode was used to drive around the test track. When a measuring beam found an error, the sled stopped and marked the spot with a yellow beam (Figure 22).

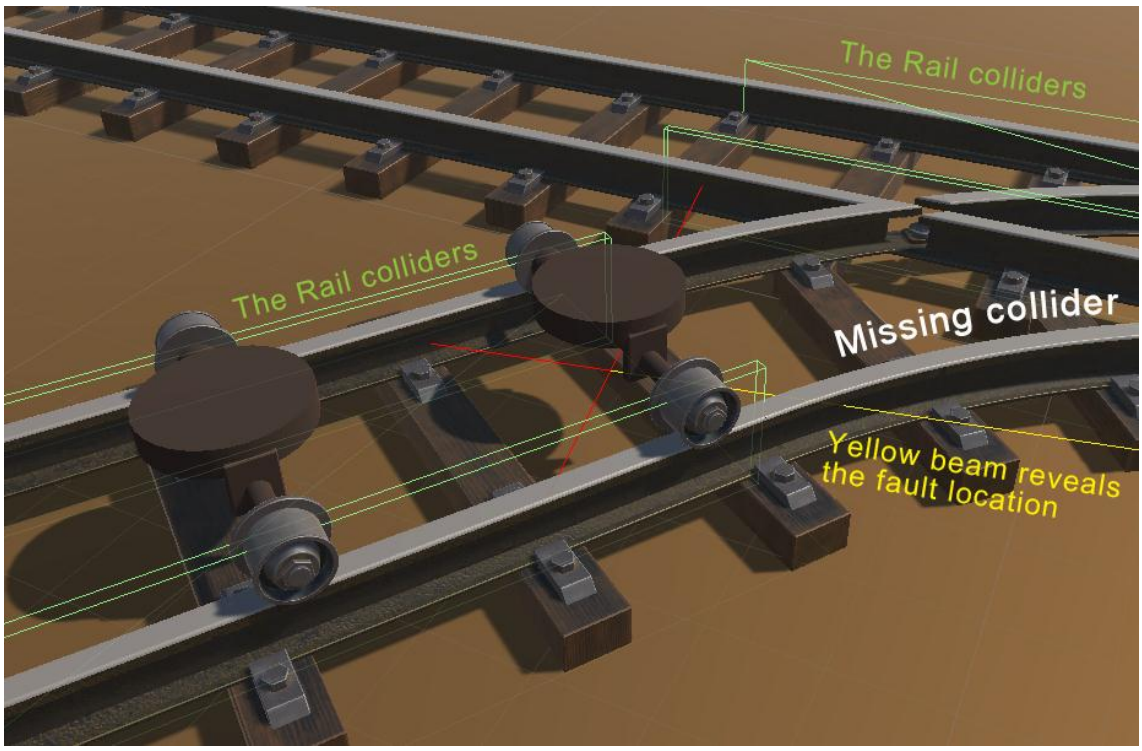


Figure 22.
Error localization with the Unity model. The virtual truck serves as a tracking device.

5.6 Programmed Gravity

Since the cart and the bogies were released from the gravitational force maintained by Unity Engine, a custom artificial gravity had to be created for them so that the carriage would naturally roll down slopes when neither braking nor engine propulsion affects its movement. The illusion of gravity is created by multiplying the sine of the tilt of the bogie in the direction of the tracks by the acceleration coefficient. The same force affects the speed of the carriage by slowing it down on inclines and increasing it on declines.

If and when the carriage derails, the bogies and the cart are returned to the gravitational pull of the earth — or rather, the Unity Engine — and they are free to fall against the earth's surface.

5.7 Surprising Dimension

The implementation of the rails shows how it is possible to divide the environment into visible and hidden objects in Unity. Invisible colliders serve as a kind of metaphysical figures whose influence is only indirectly felt by the game user. In this case, the truth behind the visible world manifests as a cart that stays on the rails even in the wildest twists and turns.

Philosophers from Plato (17) to Descartes (18) would probably have been happy to witness such phenomena. After all, it's something much more palpable than the shadows on the cave walls (17 p301).

Unity, clearly, has more than just three dimensions. If it has been calculated correctly, it's already the fifth.

6 Different Angles

In the previous section, it was explained how to get the bogies to stay on the track. However, that is only half the story, because it is not said at all that a cart placed on top of a pair of bogies would automatically stay facing the right way (Figure 23). The biggest risk of the body flipping upside down or the stern turning into a bow is in the angular notation characteristic of 3D environments.

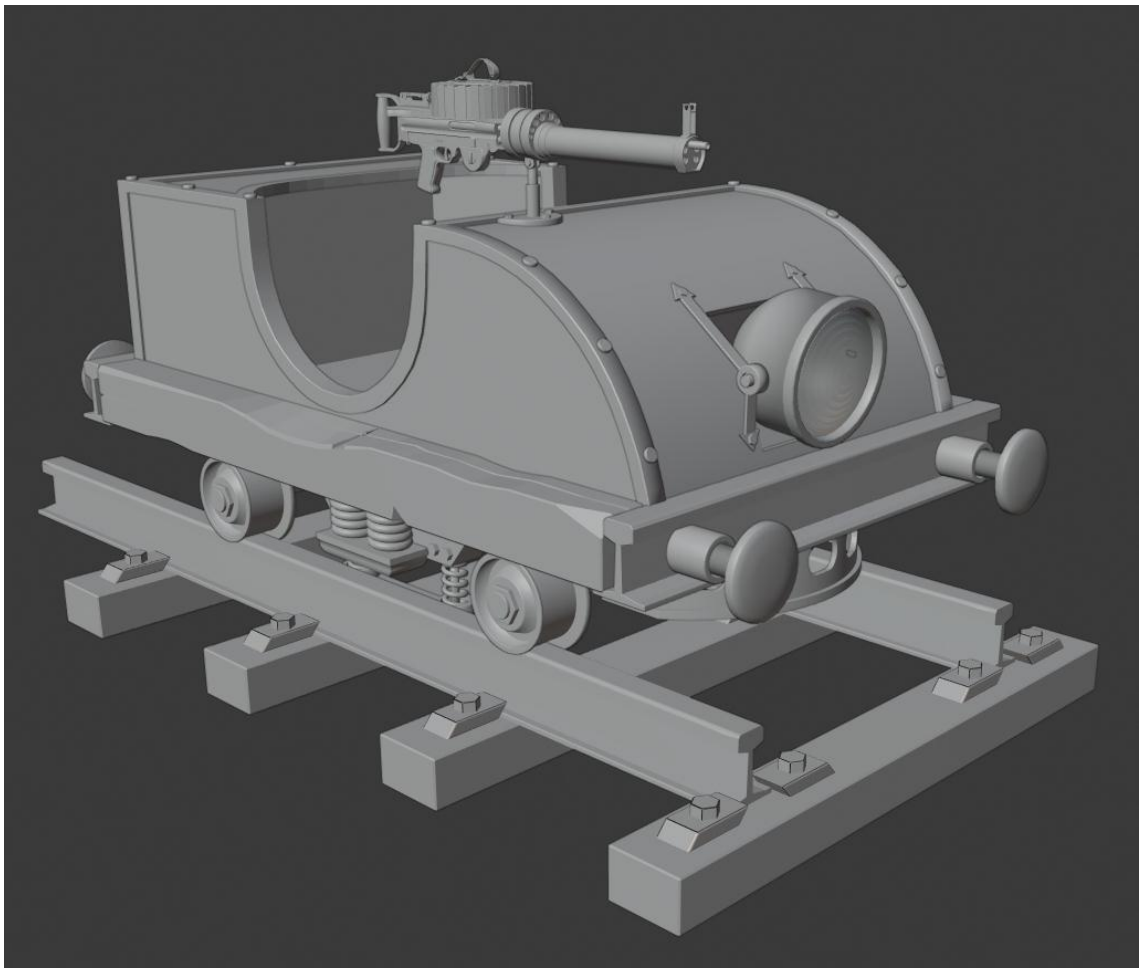


Figure 23.
The cart rests on two bogies that turn independently both horizontally and vertically. The positions of the bogies follow the shapes of the rails running under them.

In 3D games, objects are frequently rotated in relation to themselves, to each other, and to the surrounding virtual world. To manage these rotations, Unity offers information about an object's orientation in terms of the x, y, and z axes in various spaces as Euler angles (radians or degrees can be selected), as well as using the slightly more complex quaternions.

6.1 Euler Angles

Euler angles are named after their developer Leonhard Euler. The basic idea behind them is to define an angular displacement as a sequence of three rotations about three mutually perpendicular axes. The most common convention is so-called “heading-pitch-bank” convention for Euler angles (19 p229).

Euler angles have some advantages:

- they are easy for humans to use
- they are the most economical way to describe an orientation
- any set of three numbers is a valid set of Euler angles.

But they have also disadvantages like that, the representation of given orientation is not unique (Figure 24). (19 p236-237).

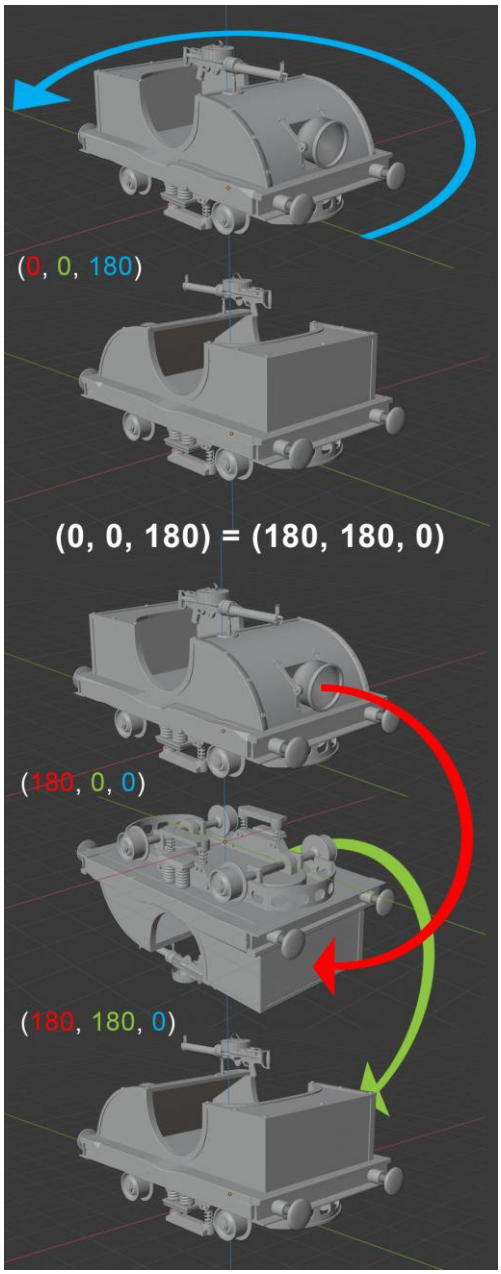


Figure 24.
An example of the redundant nature of Euler angles notation. A 180-degree rotation around the z-axis produces the same result as a 180-degree rotation around the x-axis followed by a 180-degree rotation about the y-axis.

Orientation can therefore be marked with Euler angle sets equally well (0, 0, 180) or (180, 180, 0).

The eulers became problematic when calculating the orientation of the cart body. It should always be the average of the front and rear bogie orientations.

In cases where the bogies' own direction angles were on both sides of zero: one bogie's rotation with respect to a certain coordinate axis was given a value close to zero, and the other's value was a number near 360 degrees (or zero and 2π, for those who prefer radians). Their average was 180 degrees (or π), i.e. the cart settled in exactly the opposite position as it should.

Normalizing the angles between -180 and +180 degrees does not help the matter due to the ambiguity of the notation method described above. Interpolation between the two directions is notoriously problematic with Euler angles (19 p237).

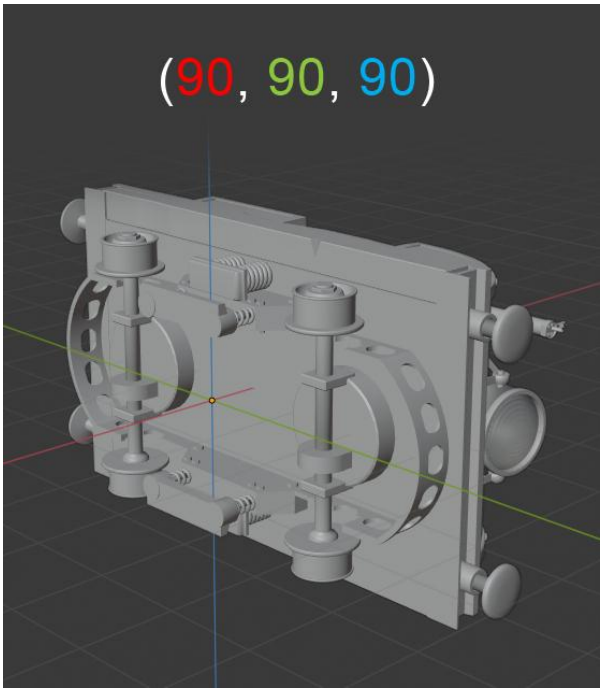


Figure 25.
To keep the cart right side up, division should not be trusted with Euler angles.

The problem can be approached through an example. The rotations (0, 0, 180) and (180, 180, 0) of Figure 23 gave an identical result. Thinking with common sense, the average of these two rotations: (0+180, 0+180, 180+0) / 2 = (90, 90, 90) must also end up in a similar orientation. Instead, the cart ended up on its cheek (Figure 25). Common sense does not always have much of a role in the 3D world.

6.2 Quaternions

The father of quaternions is the Irish mathematician William Hamilton, who in 1843 scribbled his insight into a three-dimensional system on a stone of Brougham Bridge during a walk. He named his new system of numbers “quaternions” because each number quad-ruple had four components (20).

Hamilton's method is not troubled by ambiguity; there is a unique representation for each orientation. Hamilton’s quaternions sort of implemented competing

Euler's rotation theorem, where any orientation **a** can be rotated into the orientation **b** via a single rotation about a carefully chosen axis (19 p255).

Unity uses the Quaternion class to store three-dimensional of GameObjects, as well as using them to describe a relative rotation from one orientation to another (15 class-Quaternion.html).

Unfortunately, quaternions are not very intuitive to humans.

6.3 Duel of Titans

After the above introduction, it should be easier for the reader to understand Euler and Hamilton's (Figure 26) battle for the soul of the author of the thesis. This time, that competition can be considered to have ended in a draw.

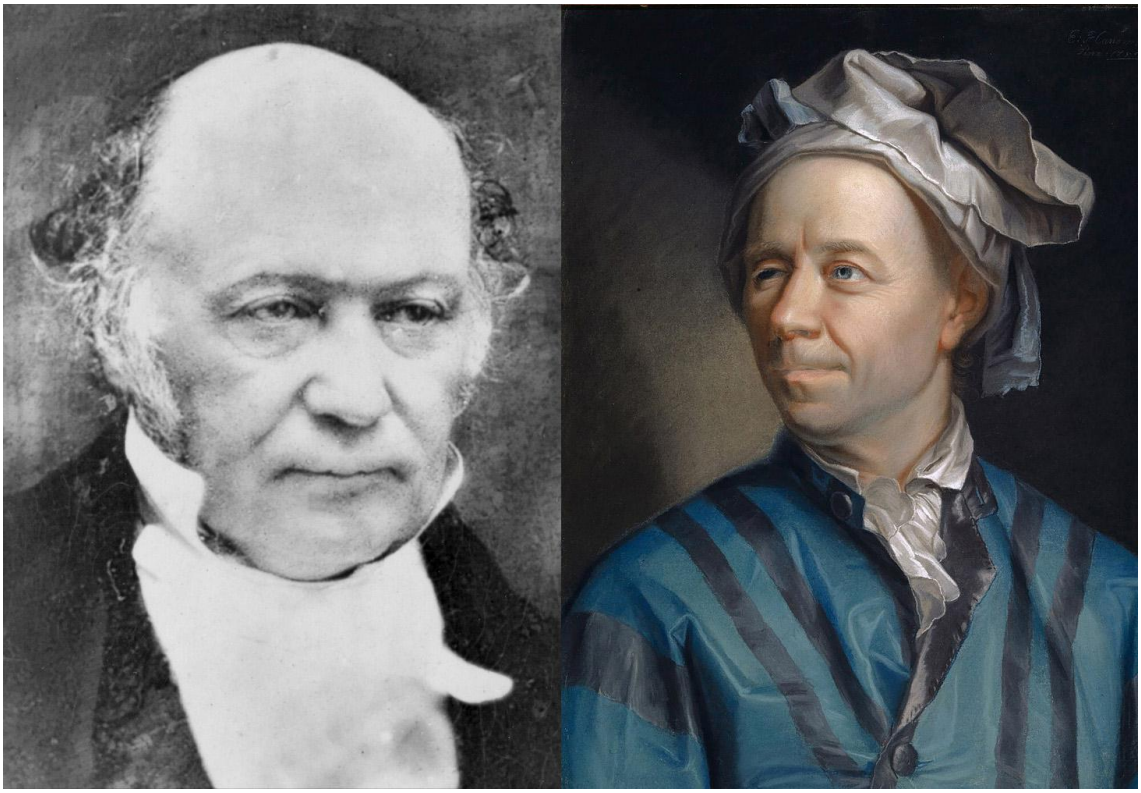


Figure 26.
Sir William Hamilton (1805 — 1865) and Leonhard Euler (1707 — 1783), the inventors of angle systems used in Unity (image source Wikipedia).

The spatial rotations of the bogies are calculated with Euler angles, because they are a logical and easily understandable representation of the object's orientation in space. Instead, the position of the cart relative to the bogies is calculated using quaternions.

”3D Math Primer for Graphics and Game Development” says that quaternions can’t be divided (19 p254). Even the oracle of our time, GhatGPT, stated that: “Calculating the average of two quaternions is not trivial, because quaternions are complex numbers with four components, representing rotations in three-dimensional space. Attempting to compute the average of two quaternions can lead to counterintuitive or even impractical results since there is no linear order between quaternions” (21).

Still, in the spirit of experimentation cart’s angle was heretically calculated, regardless of the warnings, as an average of two quaternions with a formula: $\mathbf{Q_{cart} = (w_{bf} + w_{br}) / 2, ((x_{bf} + x_{br}) / 2, (y_{bf} + y_{br}) / 2, (z_{bf} + z_{br}) / 2)}$, where **bf** stands for front bogie and **br** for rear bogie. With this method the cart was successfully kept in the correct orientation in all situations — probably because the reference values were very close to each other. However, it resembles an old rule from the 3D jungle: “There is no winning without faking” (22).

A more correct way to find the right orientation in the middle ground of two quaternions is of course the spherical linear interpolation, also known as slerp (19 p259). Such can be found from Unity as a static method of the Quaternion class (15 class-Quaternion.html), and it was also used as a final solution when calculating cart’s orientation. The experimental and misguided (but for some reason surprisingly well-working) average algorithm was therefore replaced with a method call: $\mathbf{Q_{cart} = Quaternion.Slerp(Q_{bf}, Q_{br}, 0.5f)}$, which is also a very nice and concise solution – and which is not booby-trapped by any potential malfunction.

7 Avatar’s Anatomy

In the Objective of the thesis, an avatar that replicates the user's actions is defined as one of the goals. In regards to placing human figure in the virtual world, it was necessary to consider which technique would be the best for its implementation. One option would have been to build a controlling frame (also called a rig) for the character already in the modeling environment, i.e. in Blender (23 p129-237), and use it to create a few frame-long animation sequences from different functions (23 p238-361). However, it seemed like a very laborious task and poorly suited to the need, where the character has to hang on to the machine gun and turn it according to the movements of the mouse.

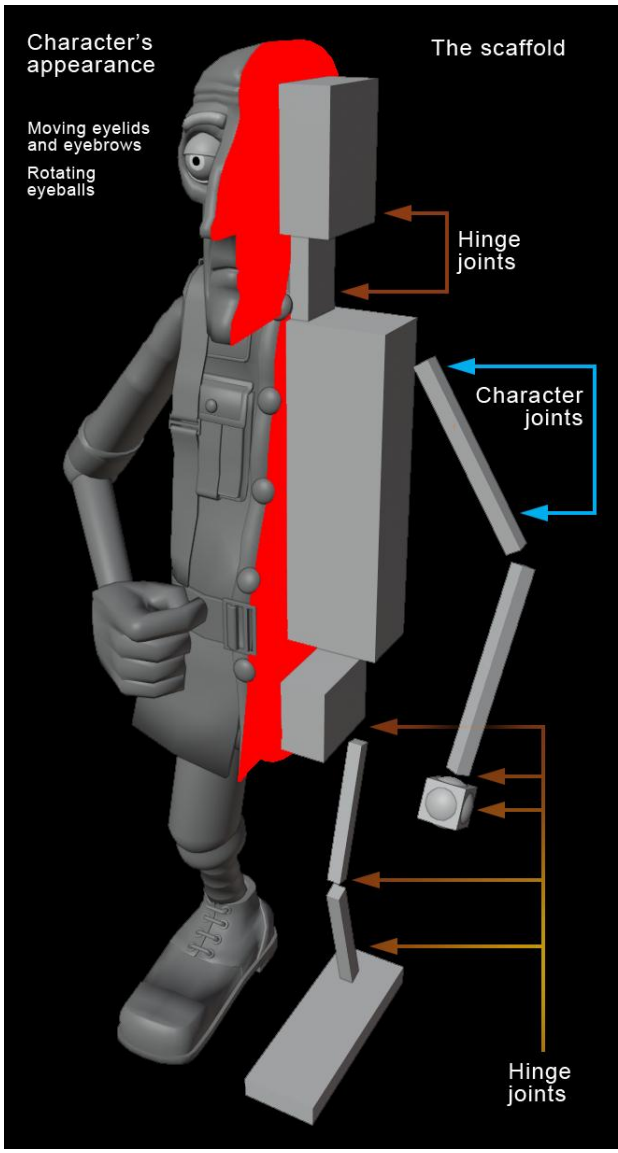


Figure 27.
Otopsy of an avatar. The model has both gravity reacting joints and programmatically operated parts.

A marionette-like figure fits that need better than an animated creature. Unity would have provided a Ragdoll (15 ragdoll-physics-section.html) component suitable for the purpose, with a whole set of joints and body parts prepared for a human-like character. However, it was not considered necessary to resort to that, instead it was decided to make an avatar gentleman from scratch, who would borrow his forename and facial features from the late actor, Sir Alec Guinness. The support skeleton of the character was assembled from Unity's primitives and finally the habitus made with Blender was put on the scaffold–like skeleton (Figure 27).

Someone may ask why the character is dressed as a British soldier. Well, naturally, this thesis is related to an English-language degree. It's somewhat elementary.

7.1 Character Mobility

Does the dog wag the tail, or does the tail wag the dog? Alec the Avatar's palms are attached to the handles of the Lewis gun (Figure 28) – and the gun is manipulated by the game player's mouse hand. Programmatically managed movement of the gun integrates with the weight and momentum of the avatar maintained by Unity's game engine. The hips are also raised and lowered programmatically to create the illusion that the shooter is using his legs to shift position. In reality, the legs are articulated to the hips – so in this case, the answer to the initial question posed is: the tail wags the dog.

Figure 28.
The combination of programmatically controlled and subject to the laws of nature made the character remarkably agile and convincingly adept at handling weapon and cart's control lever.

So Alec hangs limply in a gun that is controlled by a script. Except for his hips and eyes, he's like a puppet hanging from two strings. Such a hybrid distinguishes itself favorably from an animated model in that it interacts more

realistically with its environment compared to a character following pre-modeled motion paths.

For example, in the tactical shooting game Ghost Recon at the turn of the millennium, there were numerous death animations during which characters transitioned from time to eternity almost as dramatically as in B-movie westerns (24). In the next generation of the game series, characters hit by gunfire no longer went through pre-prepared choreography; instead, they were left as rag dolls at the mercy of game engine's natural forces in their final moments (25). The heroes who adopt a slacker posture upon falling have clearly made their mark in the gaming market from the early stages.

7.2 Nested Heads

The avatar crafted for the virtual world is not exactly an emotions interpreter, but it's capable of portraying the basic soldier's mental states, ranging from boredom to aggression, without forgetting the focused gaze when aiming a weapon (Figure 29). The character also blinks every fourth second, except when shooting, after which it blinks rapidly twice.

Figure 29.
The avatar does not yet master the representation of as wide a range of emotions as its prototype, Sir Alec, but it performs its self-assigned role exceptionally well. The emotional states from left to right: bored, attentive, and angry.

Blinks and the aiming expression are created by programmatically moving the eyebrows and rotating the eyelids. Producing a fierce grimace, on the other hand, requires a skull swap. The eyebrows and eye constructs remain unchanged, but within the head representing the basic expression lies its own open-mouthed mesh complete with teeth. When the character grimaces after shooting continuously for a certain period, the tamer skull is hidden, and the lurking grimacing skull within it is rendered.

It's somewhat misleading to speak of dressing Blender meshes over the skeleton, as in the Unity hierarchy, they are actually placed inside skeleton's primitives – even though the meshes may be visually larger than the primitives they are contained within. In Unity, a one-liter container can easily hold many liters of content. In other words, this situation reflects the ability of the GameObject mentioned in section 4 to serve as a directory-like host for its sub-objects. It's also noteworthy that leaving such a host unrendered does not affect the visibility of the objects it contains.

Objects nested within a GameObject inherit their host's orientation, so they are automatically in the right positions when needed.

In the final game, the avatar is rarely visible. Even now, it is lured in front of the eyepiece with a selfie camera, which can be considered one of the necessities of modern times. In a First Person Shooter game, instead of selfies, something akin to Figure 30 is shown to the player.

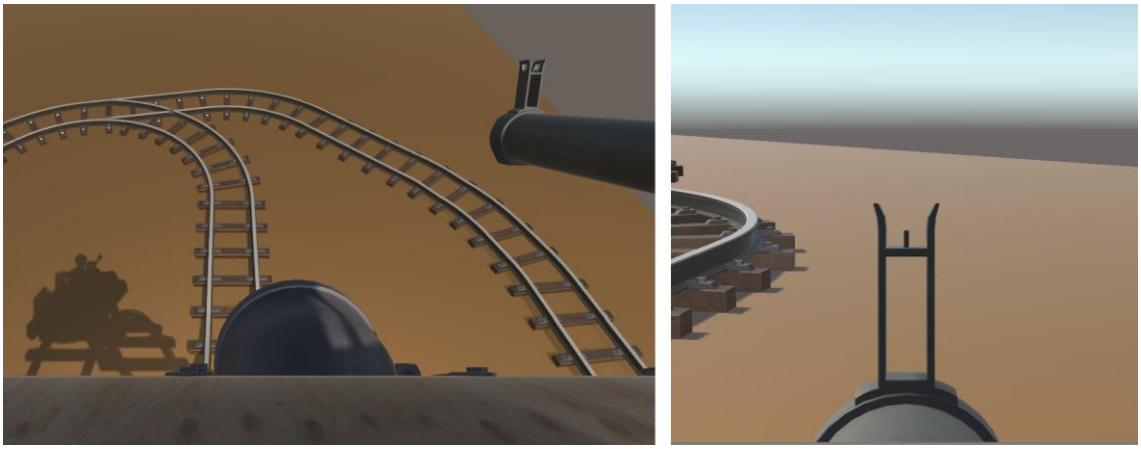
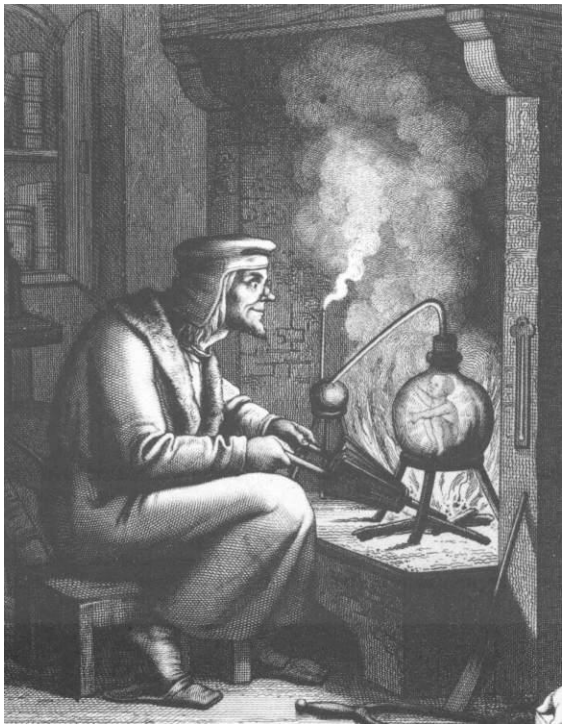


Figure 30.
FPS-style perspectives: the driver view on the left and the shooter view on the right.

7.3 Self-made Homunculus

It's hard to put into words what it felt like to see a self-made virtual human blinking and rolling its bloodshot eyes for the first time. How touching it was to see it move, express, shoot and steer the cart with firm hands.

Although reason told that the creature was only a reflection of its creator, eyes and heart said that it was much more — almost alive. Now it is easy to understand how Geppetto felt when his carved puppet started talking (26), or what the alchemist thought as the Homunculus came to life in the glass bottle (Figure 31). Let the floor be granted to a more eloquent wordsmith:



“The glass resounds with lovely power,
It clouds, it clears up: thus it must come to be! I see in the dainty figure;
A nice little man making gestures.

Come, press me right tenderly to your heart! But not too hard, so that the glass doesn’t break! That is the attribute of things: Natural, all the world is barely enough; What’s artificial demands an enclosed space” (27 p253).

Figure 31.
Wagner and Homunculus from Goethe’s Faust.

Creating the character was also staggering in that it succeeded unusually without the usual fumbling. Perhaps that made Unity appear as if it had a good dose of Kurt Vonnegut's imaginary UWTB, what makes universes out of nothingness — that makes nothingness insist on becoming somethingness (28 p138).

8 Full Pixel Jacket

In this thesis, the primus motor of programmatically controlled damaging interaction between objects is the bullet. When it hits a suitable target, it either leaves a mark on the surface or alters the state of the target in a way that creates the illusion of changes in the target's topology.

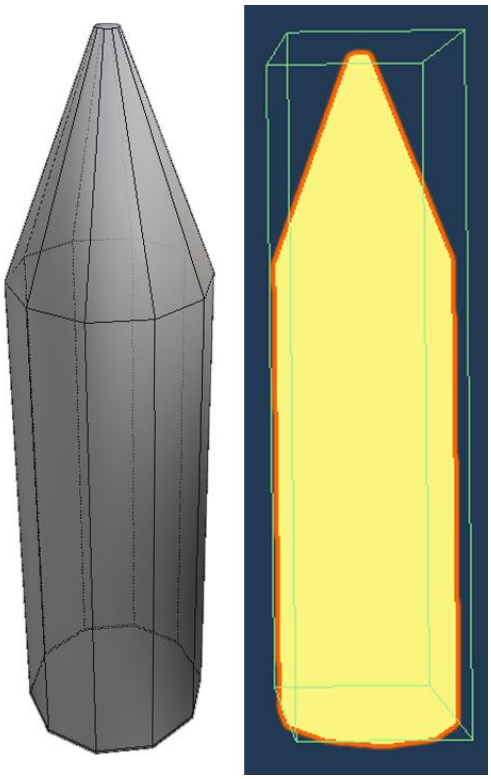


Figure 32.
On the left, the bullet in Blender and on the right in Unity.

The bullet was created in Blender and consists of 40 vertices (Figure 32). The eight corner points of the box collider covering the object in Unity would have been sufficient to define the essence of the slug, but during the development stage, working in Unity was made easier by being able to determine the projectile's orientation at first glance from its pencil-like shape. In the final game, shots are not rendered except in debugging mode, so unnecessary vertices do not affect performance.

The bullet is coated with a material that is independent of lighting and stands out like a spark when rendered visible for some reason or another.

A quick trip down Memory Lane, to Huovinrinne in the 1980s, where the writer participated as a conscript in nocturnal live-fire exercises with tracer rounds, clarifies the matter. They are a type of ammunition whose light composition, consisting of phosphorus and magnesium, burns for a long time even in deep snowdrifts.

Both the ammunition streaking across the sky at Huovinrinne and the self-illuminating bullets crafted in the Unity world have the same purpose: they show

where the bullet spray is heading (29 p207). In the gaming surround, emphasizing the visibility of small, fast-moving objects helps in dealing with them. In addition to the brightness of the shot, thin light tails are drawn for the projectiles, akin to the light trail left by tracer bullets lingering on the viewer's retina. Unlike the bullets, those tails are drawn also in the final game.

8.1 Pre-fabrication

Following its prototype, the virtual Lewis gun fires almost 550 bullets per minute, which is about 9 bullets per second (12). In the game, bullets are created from Unity-specific prefabs, meaning they are copied from a model bullet created in Project Assets during runtime (Figure 33). Instantiation can also be done in Unity for objects found in the project hierarchy, but the prefabs offer the advantage of being available for use in all game scenes.

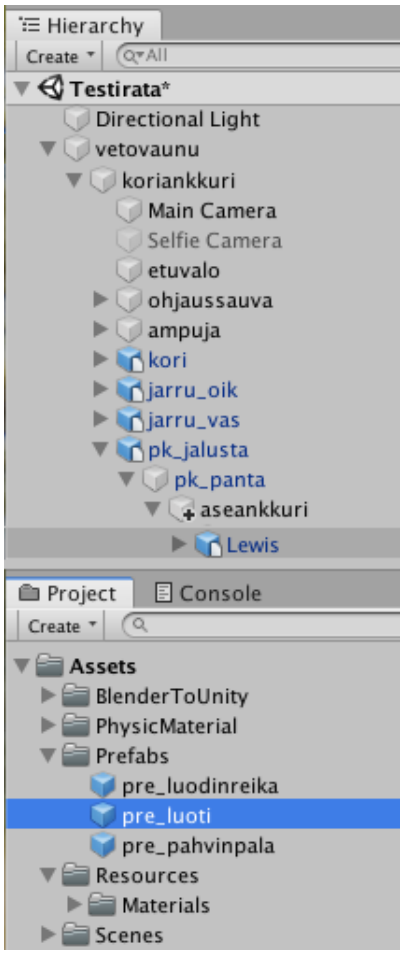


Figure 33.
Prefabs are located in Unity in project assets

The prefab can be created in the same hierarchy view as regular GameObject-based characters – and once the component is ready, it's simply dragged from the hierarchy tree to the project assets. Even the most intricate prefab creation requires only one command; below is an example of bullet creation where the projectile is placed in the muzzle position tip facing forward:

```
GameObject bullet = Instantiate(bullet_prefab,
flash.transform.position, Quaternion.identity);
```

Noteworthy about prefabs brought to life is that their initial actions should be done in the Awake method rather than the Start method, unlike conventional components raised from the hierarchy. Awake is called immediately after the Instantiate command, while Start is only called in the next iteration of the game

loop (30 MonoBehaviour.Awake.html). It's just a matter of milliseconds, but even in this project, other methods of the prefab were executed before the Start method's initial actions – and that understandably caused problems until the initiation functions were moved to the Awake-method.

8.2 Awkward trouble – Teleportation

An airborne projectile would be expected to be one of the simplest components (Figure 34). Indeed, it is, but it has one characteristic that makes it a challenging model in the gaming world. It is fast. So fast that a game loop updating 50 times per second is far too slow to carry a bullet traveling through the air across all possible intermediate points from point A to point B.

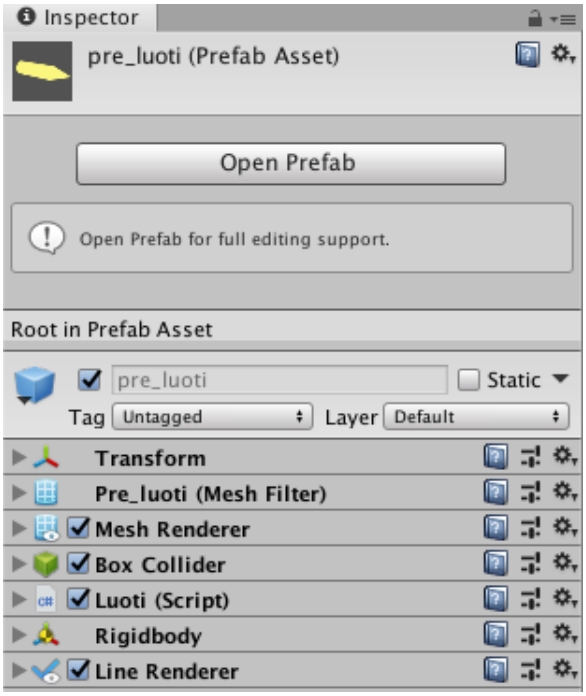


Figure 34.
Despite its simplicity, the bullet prefab contains a whole array of interesting components, such as a box collider, script, and a line renderer that is used to draw the bullet's fiery tail.

A virtual bullet simulating speeds exceeding twice the speed of sound (746.76 m/s) should cover nearly 15 meters (or equivalent amount of game units) in one update iteration of the game loop (12). In practice, the bullet teleports from one place to another without being present at many points during its flight. What makes this problematic is that, due to its quirky movement, the shot cannot hit nearly any obstacle along its path, unless it is helped a little.

Therefore, the bullet scans the terrain ahead with a beam (like the bogies in section 5) for the length of the next leap. If the beam encounters an obstacle, the length of the teleportation is shortened exactly to the length measured by the beam, causing the bullet's collider to interact with the obstacle's collider after the jump (Figure 35). If the obstacle's collider is not of the trigger type, the

collision prompts Unity's physics engine to rotate the bullet's orientation, and the bullet rebounds, just like in real life. Huovinrinne with its tracer rounds is here again!

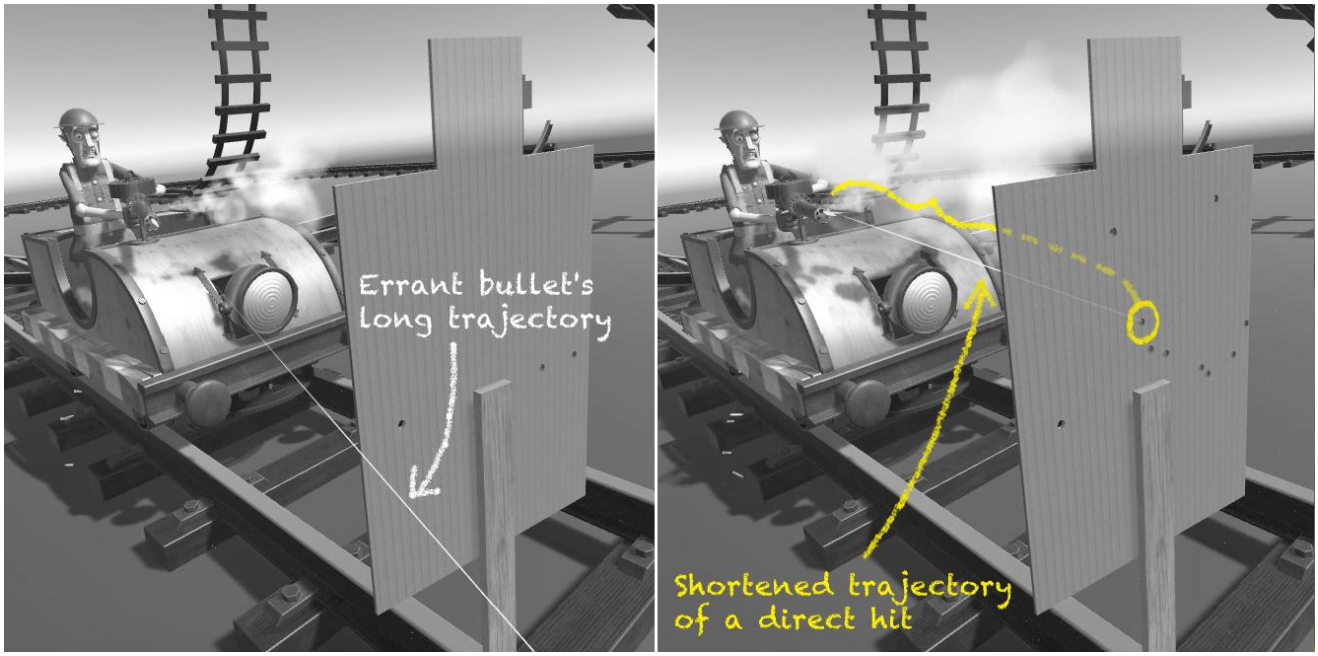


Figure 35.
The bullet, whose trajectory is unobstructed, travels the distance required by its speed within one iteration of the game loop. If there are obstacles in the bullet's path, the trajectory is shortened to match the distance to the nearest obstacle.

8.3 Bullets are not Eternal

The Lewis replica fires bullets at a commendable pace. Although the fruits of the submachine gun are relatively simple and not even rendered, they still consume computer resources purely due to their sheer quantity. Additionally, they burden memory with their changing locations and continuous probing of their trajectories. That's why the bullets were made short-lived. Their lifespan is only 2.5 seconds, after which they self-destruct (appendix 4).

9 Environmental Effects

Although one could infer from the title that the section deals with societal themes such as green transition and biodiversity loss, readers may be disappointed in that regard. Environmental impacts in this context only concern the interactions between entities constructed from bits – and that, by the way, is more challenging than one might imagine.

The thesis writer's most active gaming hobby was in the first decade of the millennium. Since then, the fields of action games have expanded, and the landscapes of game worlds can now be realized in almost photorealistic detail. However, one thing has remained rather unchanged: the player's ability to influence the landscape is somewhat minimal.

While the doors of virtual houses may open, windows shatter, and various objects can be blown to smithereens, often these events simply follow a pre-established and repetitious pattern, leaving a lackluster sense of real interaction while observing animated damage models.

So this time, it was decided to try a small-scale, but more unpredictably responsive damage modeling than what is customary.

9.1 Immutability of Meshes

As mentioned in section 3, Unity game objects are modified during the game development phase, and primarily, this work is done using software other than Unity. Therefore, characters appearing in the game can hardly be altered in real-time – they can certainly be stretched in various ways, but the topology of the shapes with their vertex structures remains unchanged. If the human character Alec is recalled in section 7 – even his facial expressions were realized with nested heads, each with its own expression.

From these premises, an attempt is thus made to manipulate the shape of virtual world objects, or at least create the delusion of it, during program execution. What can be done about it will be examined. However, an easier task will be taken up first as a warm-up.

9.2 Impact Patterns

The first encounter for the author of the thesis with environment-aware game design was Blood from 1997 (31). In it, bullets left comic-like impact patterns on surfaces. It was a stunning improvement over gaming classics like Doom and Castle Wolfenstein, where only monsters took damage.

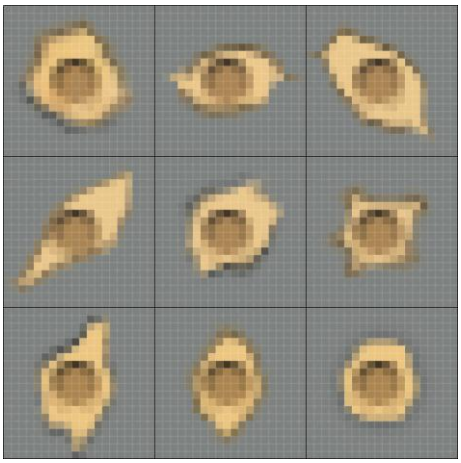


Figure 36.
Impact patterns for box-shaped objects.

In the 3D prototype developed for the thesis, encounters between bullets and box-like objects result in impact patterns reminiscent of those seen in the PC game Blood. The patterns, created using Photoshop, have been assembled into a two-dimensional sprite-type image file, containing a total of 9 different impacts, each measuring 20 x 20 pixels (Figure 36). A special prefab is used to assist the creation of fake holes.

A general class was written for impact patterns, which, when a bullet hits an object with an instance of the class, determines which of the six facets the impact pattern is drawn on, how it is oriented, and whether the pattern needs to be clipped at the edges (Figure 37). It also

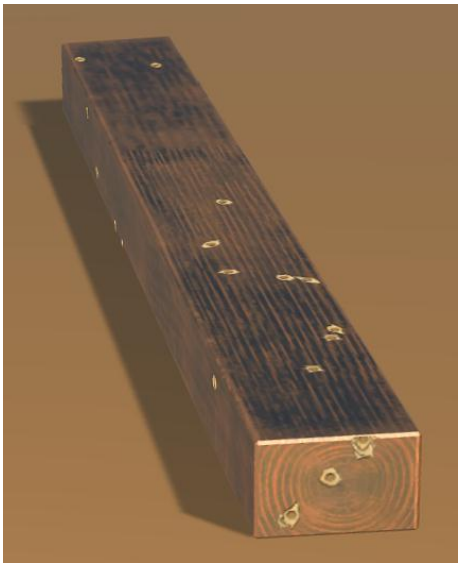


Figure 37.
Holes produced by the boxholes component and a hole prefab in the railroad sleeper.

randomly selects one of the nine alternative impact patterns to be glued onto the surface of the target object like a sticker (appendix 5).

9.3 Through-Holes in Meshes

After the warm-up, it's time to inflict more serious damage on virtual world objects. Instead of applying superficial marks, it's about piercing the cardboard for holes where the daylight shines through.

3D crafting and sleight of hand both rely on deception (22). For the viewer to experience the trick as believable, the magic requires careful preparation. So at first two identically sized cardboard cells were created in Blender; one of them intact, while the other features a bullet hole. The intact cell is visible by default, and the one with the hole is hidden.

In Unity, a script stacks the cells to form a target. Since the dynamically created target is not visible in Unity's design-phase scene view, a simple placeholder was made from a single Blender object. Once the program starts, the temporary placeholder is hidden, and prefabs are assembled into the final target, consisting of 1300 cells, resembling cardboard.

The 1300-piece puzzle-like target also closely resembled the placeholder, so the latter was labeled "Place holder" for clarity (Figure 38).

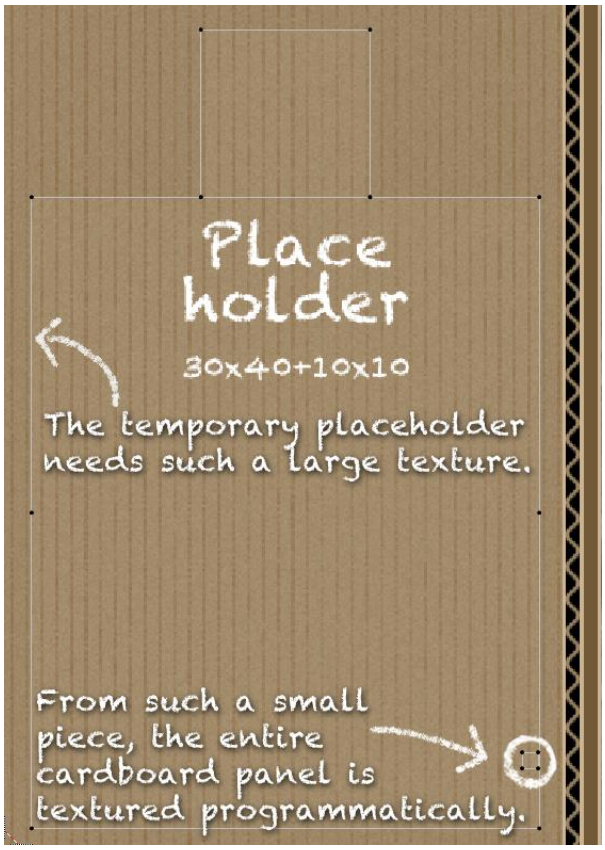


Figure 38.
Different spatial requirements. The texture surface areas used by a single-piece cardboard panel versus a panel assembled from cells.

The tiny cardboard pieces in Unity have been boosted into quite the small giants by adding to each of them their own particle system, a collision-triggering box collider, and, of course, logic written in script.

When a bullet hits an intact cell, it hides its intact mesh and makes the perforated one visible (Figure 39), emitting cardboard debris with its particle system, and communicates the impact it received to the host object assembling the target.

When a cell receives another hit, it hides even its perforated mesh, and its state changes to "wiped off". In the cardboard target, such a cell appears as a square-shaped gap; at the edge of the target, it gives the impression of a bullet-bitten edge wound.

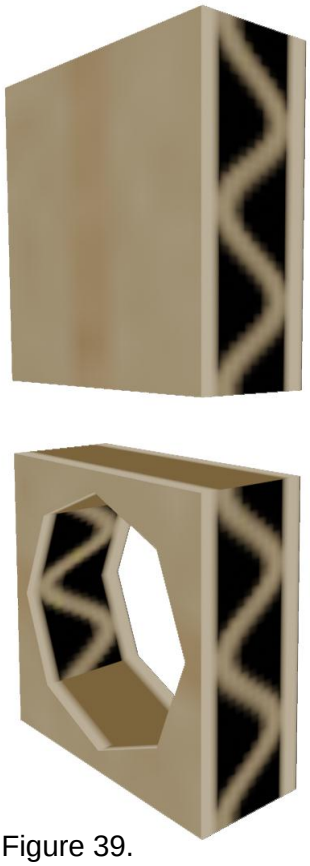


Figure 39.
Intact and perforated cell.

Due to the high-speed nature of the bullets, the velocity of the projectiles was reduced, and their lifespan was increased during the debugging phase to visually assess the viability of the concept (Figure 40).

It's easy to see where such a development path leads. Of course, to the point where parts can be sawn off from the cardboard target with a bullet spray. From there, is delved into the next level illusion: the apparent dismantling of the mesh in real-time.

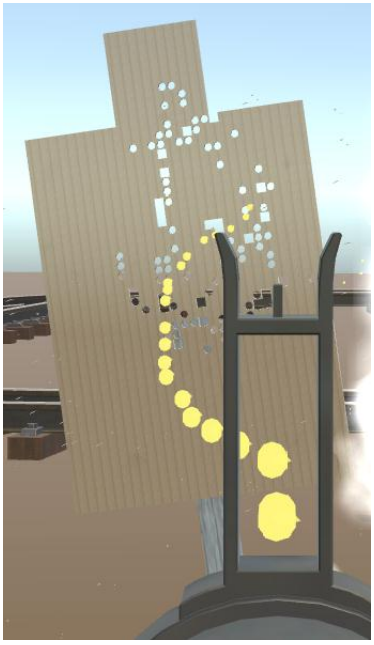


Figure 40.
The slowed down tracer rounds helped to visualize the functionality of the collision detection and perforation algorithm of the intelligent cell-based cardboard panel.

9.4 Disassembly of the Object

Disassembling the target into cardboard planks of undetermined size and shape required some additional preparation of the cell-level components. Therefore, each cell was made aware of its neighbors, in other words, the eight closest cells surrounding it (Figure 41). Cells positioned at the edges and corners of the board have fewer adjacent cells, and for them, missing neighbors are replaced with a null value in the cell-specific neighbor array.

Cell networking enables the exploration of cutting lines and islands through recursive subroutines, which seamlessly navigate from one cell to another, utilizing the pointer list in each cell that leads to its neighboring cells.

Figure 41.
The building blocks of the cardboard target are networked. They are aware of the presence, as well as the absence, of their neighbors.

9.4.1 Convex Hulls

The mathematical term for the natural boundary of a point set is the convex hull (32 p359). It is more common to search for boundaries for a known set of points than to define the points enclosed by a specific existing boundary. However, the latter is done concerning the parts detaching from the target. The rapid rifle punctures the target with lines, which eventually form a continuous path cutting through a piece of the cardboard.

Different cutting patterns can be broadly divided into five types:

1) splitting cases, where the target is cut either vertically or horizontally

2) same-sided fractures, where the cutting line starts and ends on the same side of the target.

3) o-loops, or complete loops within the target

4) p-loops, which have a tail leading to the edge of the target.

5) false loops, where the cutting line crosses the so-called support line to which the target is attached to its support structure.

Each of these types can be found using the same search algorithm, but the post-processing of the found paths depends on the path type (Figure 42).

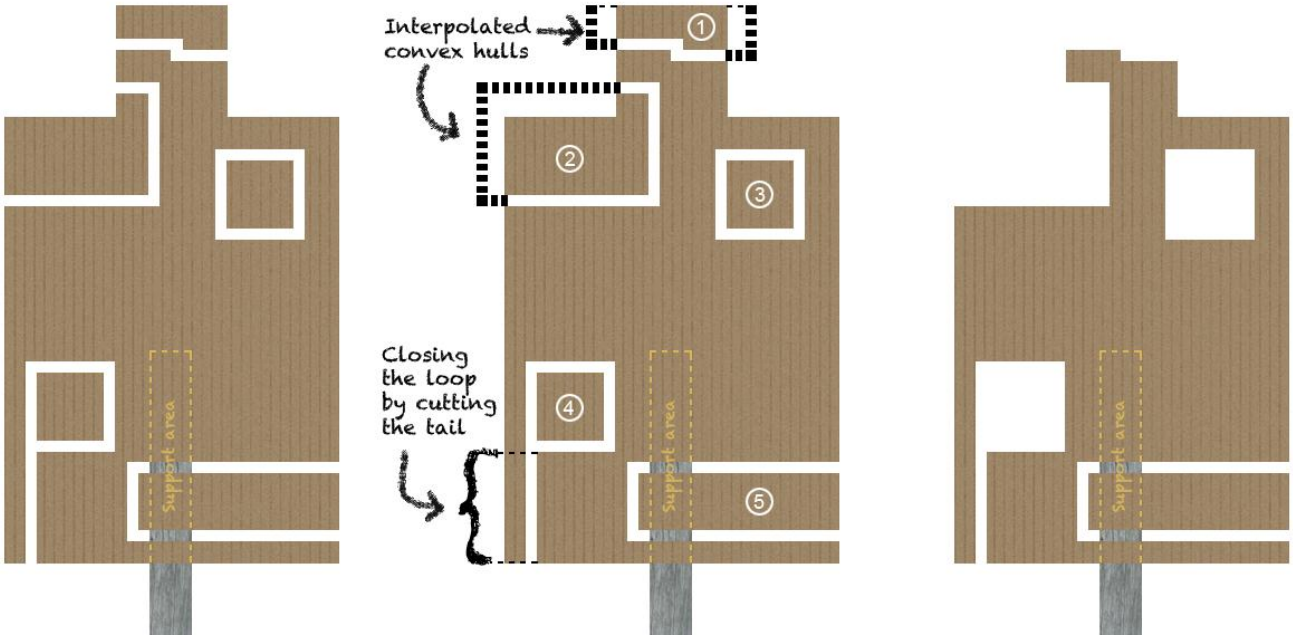


Figure 42.
Five different patterns: Convex hulls of 1 and 2 are finalized by interpolation, 3 is accepted as is, 4 has its tail cut, and 5 does not have a cutting pattern at all because it cross the support area.

9.4.2 NP-completeness

Although cardboard fragments may fall off due to shooting, diminishing its visual appearance, the programmed structure of the board remains unchanged; only the states of matrix cells change. Already used cutting lines can still be found, even if the cells around them have long fallen to the ground like leaves from an autumn tree.

Often, art imitates reality, but this time reality imitates art, as the virtual cardboard could be straight out of a Matrix movie (33). It is something entirely different from what it seems. Behind the messy visible reality lies a layer of a different truth: the invisible but permanent logical structure. However, the matrix faithfully preserving its form has a flip side. Instead of lessening the route options for potential cutting lines as the visual board shrinks, they actually increase exponentially.

To identify paths formed by erased cells, the searching algorithm progresses from wiped off cell to another in a way that it never retraces its path but chooses one neighbor cell at a time from a maximum of three horizontal and vertical neighbors. The algorithm does not consider diagonal directions because the pieces that detach from the board shall not to be connected to each other, not even at the corners of the cells (Figure 43).

Figure 43.
The route from cell 616 to the left cell 615 can branch in three directions. There are a total of 1146 cells in the board that branch in three directions (brown cells) and 148 cells that branch in two directions (green cells). From the yellow corner cells, the journey continues in one direction only.

With these rules there are about $2^{148} * 3^{1146}$ possible routes on an empty board, which is $2^{148} * 2^{1.58496 * 1146} \approx \mathbf{2^{1964}}$ routes (or 10^{593} for those who value a decimal expression more). In other words, a search algorithm based on cutting paths is like skating on thin ice, beneath which lurk the uncharted depths of NP-complete problems and Hilbert spaces bordering on infinity.

9.4.3 Pruning of Options

To prevent the wandering of paths marked by bullet holes into the jungle of countless options, the algorithm ruthlessly prunes routes it identifies. Firstly, each path is scored based on the number of untouched cells along it. Only the branch with the highest score advances. Unfortunately, at this stage, it is not yet known whether the cells bringing points are on the side to be cut off or on its opposite verge.

The branch of recursive path finding terminates itself unsuccessfully if any neighbor of an encountered cell is already found in the list of cells on the path, which the algorithm drags along like a cryptocurrency drags its blockchain. Moreover, recursion does not proceed further upon encountering a fallen-flagged cell. These measures have succeeded in bypassing memory traces haunting the matrix of previously successful but subsequently unproductive cutting paths in earlier searches.

Cutting paths are not sought aimlessly; instead, cells wiped off the board are recorded in a to-do list, and the operations handling it are carefully spoon-fed to the game loop. This prevents the occurrence of performance-sapping spikes. While in C#, it would be possible to distribute the workload to threaded parallel processes according to traditional programming models, such an approach was not taken this time. Anticipating the completion of tasks hung in the main thread's game loop by scheduling, is easier than dealing with more randomly behaving threaded operations.

Despite the delaying tactics, the detachment of pieces from the target shot like a sieve eventually becomes hopelessly laborious. From that predicament, the slicing logic of the cardboard was saved by Daniel Defoe (34), who lived between 1660 and 1731.

9.4.4 In the Footsteps of Robinson Crusoe

Why seek ponds when islands can be found? The aphorism above is not Zen Buddhist wisdom but a guideline for identifying the scattered cardboard puzzle pieces. The idea struck the mind of the thesis writer as an association from Robinson Crusoe (34), following the footprints left by Friday on the sandy shores of the deserted island inhabited by the castaway.

When a bullet hits the cardboard and the lower limit of intact cells defined for the target is breached, what is called an island lottery is conducted. Up to ten cells are randomly selected from the set of intact cells for island testing. In other words, one strolls along the neighboring network to the right until encountering a shoreline formed by wiped-off, fallen, or null-valued cells. The programmatic Robinson Crusoe marks that spot as its starting point.

Once the shoreline is found, an instance of the BeachWalker class circumnavigates the island formed by intact cells while keeping removed cells on its right, akin to the sea. When the beach walker arrives at the tip of a peninsula, it executes a full turn and retraces its steps, still keeping the "sea" on its right in the new orientation. This way, the walker navigates around the island until it has crossed its starting point twice — twice because a single-cell-wide starting point must be crossed once halfway to find both ends of the island. The second crossing occurs once the journey is complete.

In many cases, Robinson thus circumnavigates the island twice, but the additional route points he records are removed from the convex hull of the island during the sorting and normalization of route points.

Following Robinson's path along the shorelines, there is an equal probability of creating a map of either a previously removed piece, such as a pond, or an intact island. However, they are easy to distinguish from each other. The number of cells along the shoreline of ponds is the same as the number of intact cells they enclose, while in cardboard islands, there are either more cells than their shoreline or the total number of cells enclosed by the border is equal to the number of cells along the shoreline (for one-cell-wide islands).

As finding islands was as straightforward as can be inferred from the above, in testing the beach walker, resorting to color coding was deemed best: the shorelines identified by the algorithm were colored blue and the islands red. In the final program, pieces are not colored but are allowed to spread out like sheets on the ground (Figure 44).

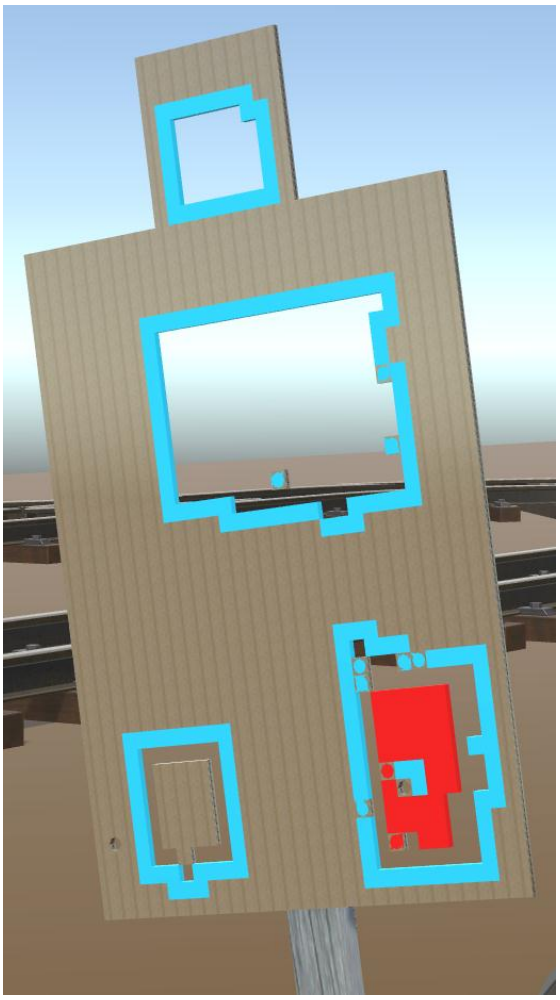


Figure 44.
Illustrating the recognition of so-called ponds and islands during the testing phase by color-coding the shorelines and islands.

9.4.5 Demolition Accomplished

Disassembling the cardboard was successful when the topology of the panel, comprised of pieces, was examined from two perspectives instead of one, in which the heavy lifting is done by a recursive path finder measuring the depths of options, and the work is completed by a simple beach walker based on a while-loop. What made the task difficult was that the state of an individual cell depended not only on neighboring cells but also on those that were not in its immediate vicinity.

For instance, a virtual brick wall to be dismantled could be constructed from blocks stacked on top of each other, susceptible to gravity. When bricks from the lower layers were shot, breaking them into rubble, Unity's physics engine would take care of the fate of the unsupported upper bricks, eliminating the need for algorithms like path finding and beach walking.

There may be more elegant methods worldwide for slicing the cardboard, but isn't an engineer recognized for their ability to develop solutions using available tools and relying on their own ingenuity?

9.5 Follow the Money

The scarcity of destructible environments in games, which was marvelled at the beginning of this section, may have just been explained by the lines above – and above all, by the sheer number of those lines. Creating the illusion of a destructible object in a non-deterministic manner requires effort, much like demolishing the Eiffel Tower demands the skill of a real-life magician. This effort costs money, and the decay algorithms consume resources from the device running the game. It is easier and safer to settle for applying textures to objects' surfaces and presenting pre-made destruction animations.

However, it would be desirable that the pursuit of taking the easy route is not the primary goal for those building 3D worlds.

10 Control Console

When the 3D environment was finally completed, construction of a control console, a device better suited for controlling the carriage than a keyboard, began. In the world of engineering, device design typically starts with drafting drawings. Having graduated as an architectural draftsman ages ago, creating such drawings would have been quite familiar to the company's design engineer. However, this time, things proceeded differently, on an ad hoc basis. Only a rough sketch outlining the desired appearance of the device was prepared in advance with just a few lines (Figure 45).

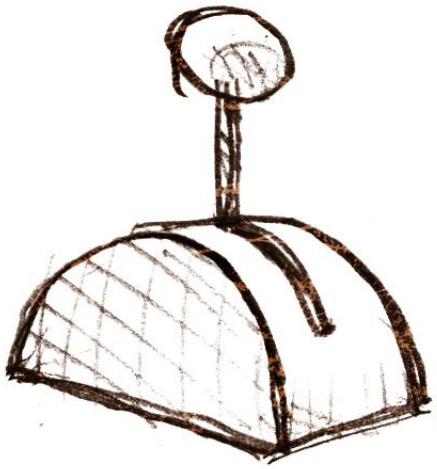


Figure 45.
The rough initial draft of the console did not address the functional structure of the device, but only its appearance.

10.1 Reactive Planning

As the construction progressed, quick sketches were made regarding various aspects of the process. Primarily, these drawings served as visual notes, such as a scribble of the dimensions of a ball bearing (Figure 46) or a graphical exploration of the lever mechanism (Figure 47). They were not intended for any level of precision.

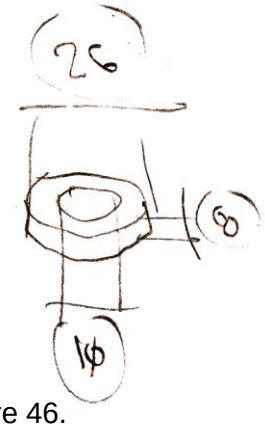


Figure 46.
Visual note of the ball bearing.

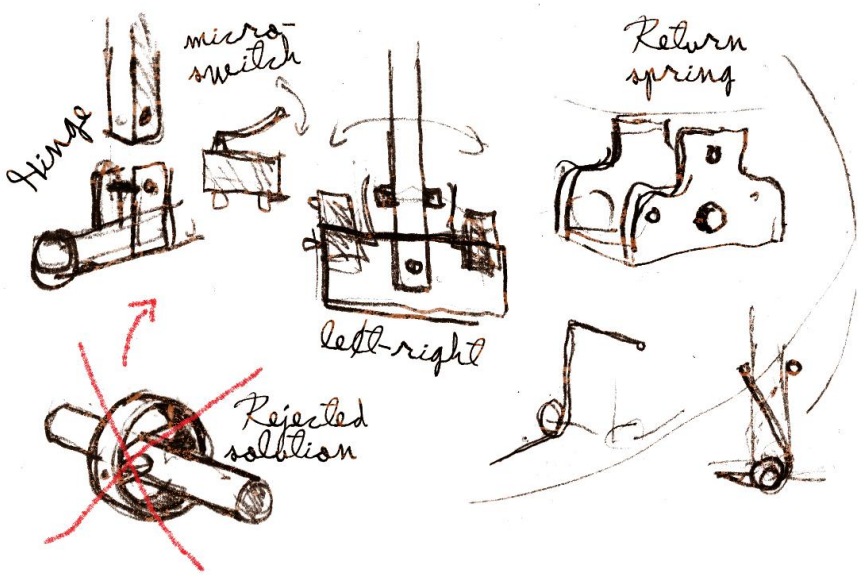


Figure 47.
Sketch of a lateral-moving lever to be attached to a rotating axis. Note the ring-like solution with a red cross drawn on top. It was rejected due to difficulties in installing the return spring.

10.2 Drafting of the Mechanics

Based on previous experience constructing machine-like objects, Tietomato Oy has found that it's easier to model them using 3D components rather than on a 2D drawing board. Sketching could have been considered using objects in a virtual environment like Blender, but it felt unnecessary. Instead, tangible materials were immediately chosen: wood and steel.

Playfully, one could say that sometimes it's more productive to think with pliers and a jigsaw than with one's brain. Thus, the construction of the console began by sawing a 10-millimeter-diameter steel tube into a ten-centimeter-long axle. The choice of a ten-millimeter tube was due to the fact that Tietomato Oy's workshop had suitable ball bearings left over from previous projects that fit the shaft with internal dimensions. The construction process was strongly guided by both the available materials and the shapes and dimensions of the sensors to be placed in the device (Figure 48). In other words, the machine's design was adapted to the prevailing conditions.

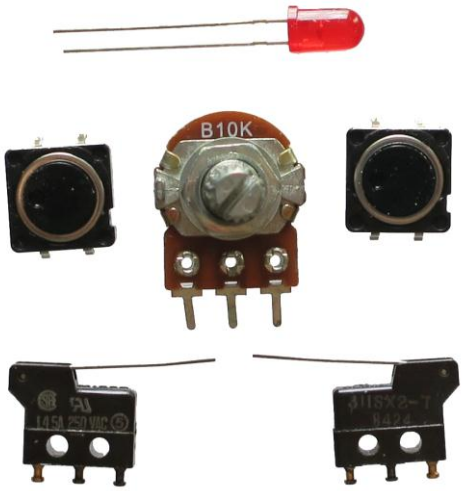


Figure 48.
The device reads user commands through two membrane buttons, two microswitches, and a potentiometer. With a red LED, the device communicates to the user that the control stick is in the brake-reserved area of its motion path.

Naturally, these components to be embedded in the device influenced both the mechanical design of the machine and its external appearance.

A length of 2 x 20 millimeter galvanized steel strip was wrapped around the tube, and a control lever was hinged onto it. Positioned within the armpits of the lever, two microswitches register the lever's lateral movements – in this case, sufficient precision was the knowledge of whether the lever was centered or tilted left or right. A cradle made of 4-millimeter-thick birch plywood was constructed for each microswitch, allowing them to be securely attached to the axle and positioned exactly where desired (Figure 49).

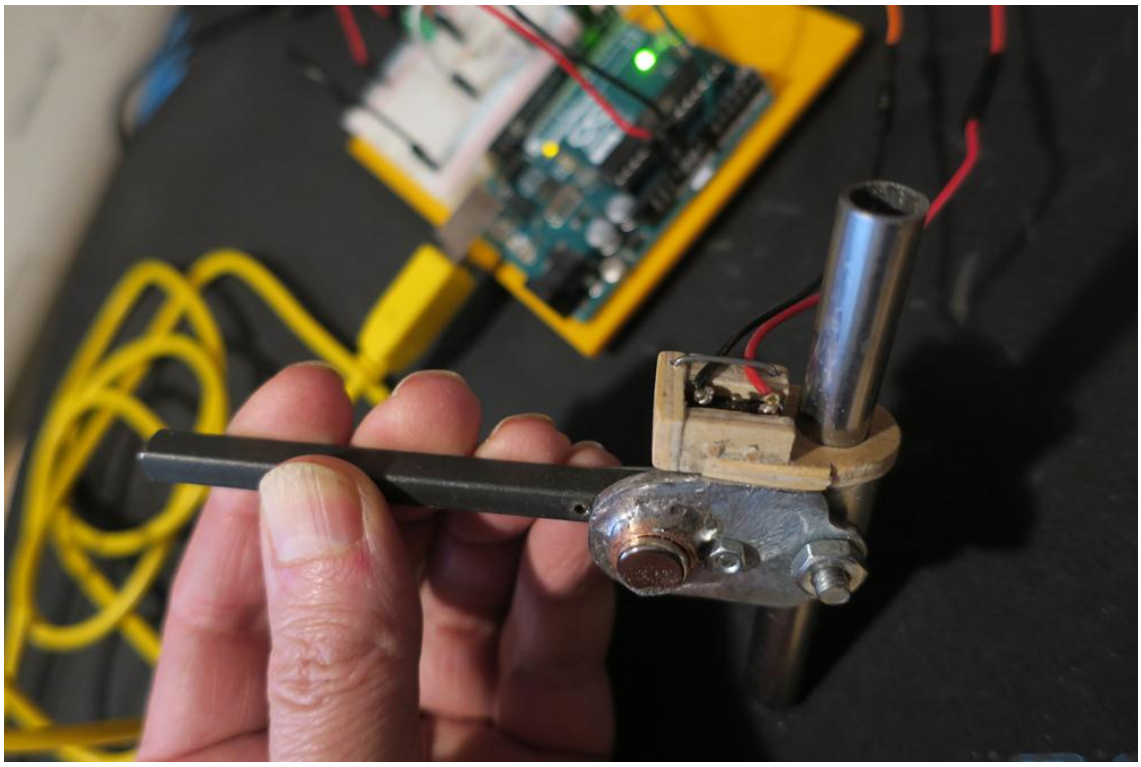


Figure 49.
The first test run of the integrated control lever and microswitch.

The transmission ratio of the axle, which rotates about 180 degrees, was adjusted with gears. The back-and-forth motion of the lever is registered by a potentiometer connected to the axle with gears, allowing for the full 360-degree rotation movement to be utilized thanks to the elevation gear mechanism implemented with gears (Figure 50).

Figure 50.
The gears needed for turning the potentiometer were modeled in Blender and printed using ABS plastic on a 3D printer.

When the lever connected to the axle was found to work with the microswitches, a suspension made of plywood was built onto the axle. The 18-millimeter-thick plywood was well-suited for embedding the potentiometer and drilling bearing seats for the axle's ball bearings. A gear was attached to one end of the axle, and a brake wheel was placed on the opposite end to prevent the axle from

rotating due to gravity. At the end of the lever, a round knob was shaped from a 2” x 2” piece of wood (Figure 51).

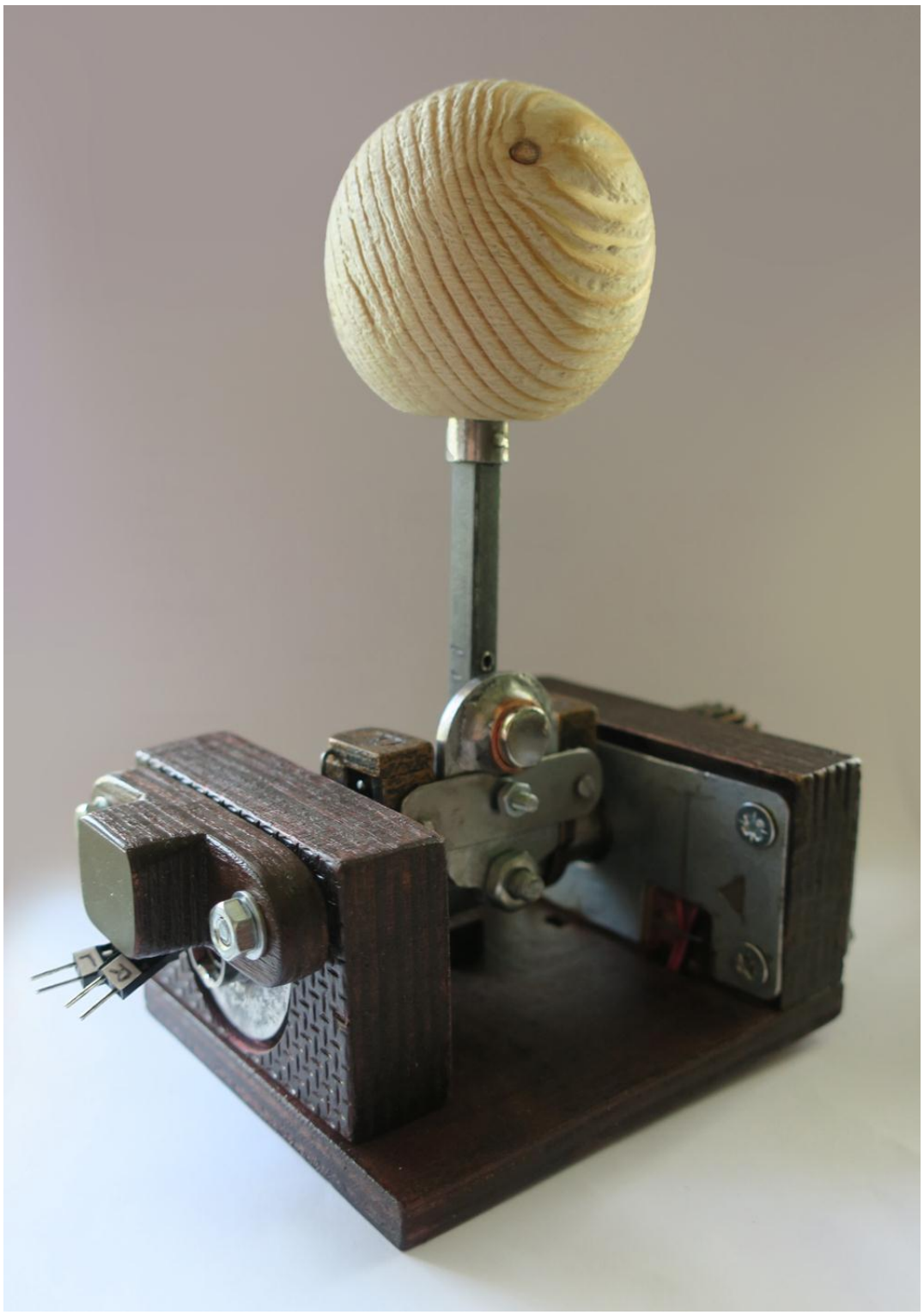


Figure 51.
The mechanical core of the device before enclosure and connection to the microcontroller.

10.3 Circuitry

When the mechanical core of the device, along with its embedded sensors, was successfully assembled, a wiring diagram was drafted. It indicates which pin each electronic component would be connected to (Figure 52). The diagram also reveals the grounding of switch-like buttons with pull-down resistors (35 p35), so that when the switches are open, the pin reads a LOW value, as the current in the conductor dissipates through the resistor serving as a drainage ditch. A decoupling capacitor for the potentiometer (35 p65) can also be observed, along with a resistor designed to prevent the LED from receiving too much voltage (35 p25).

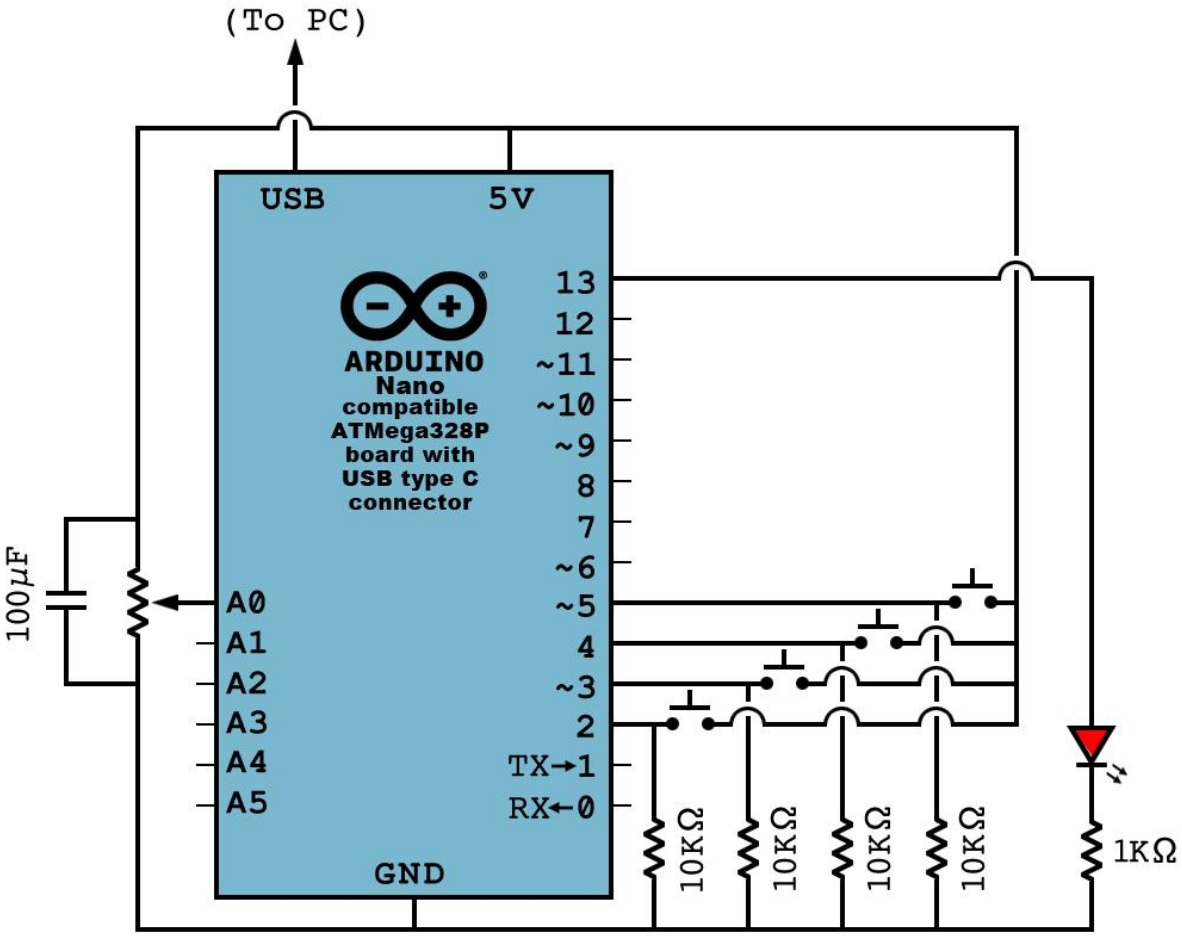


Figure 52.
Schematic view of the control console electronics.

During the testing phase of the connections, an Arduino Uno R3 board was utilized. Despite having only a few components, they still created quite a mess on the breadboard (Figure 53).

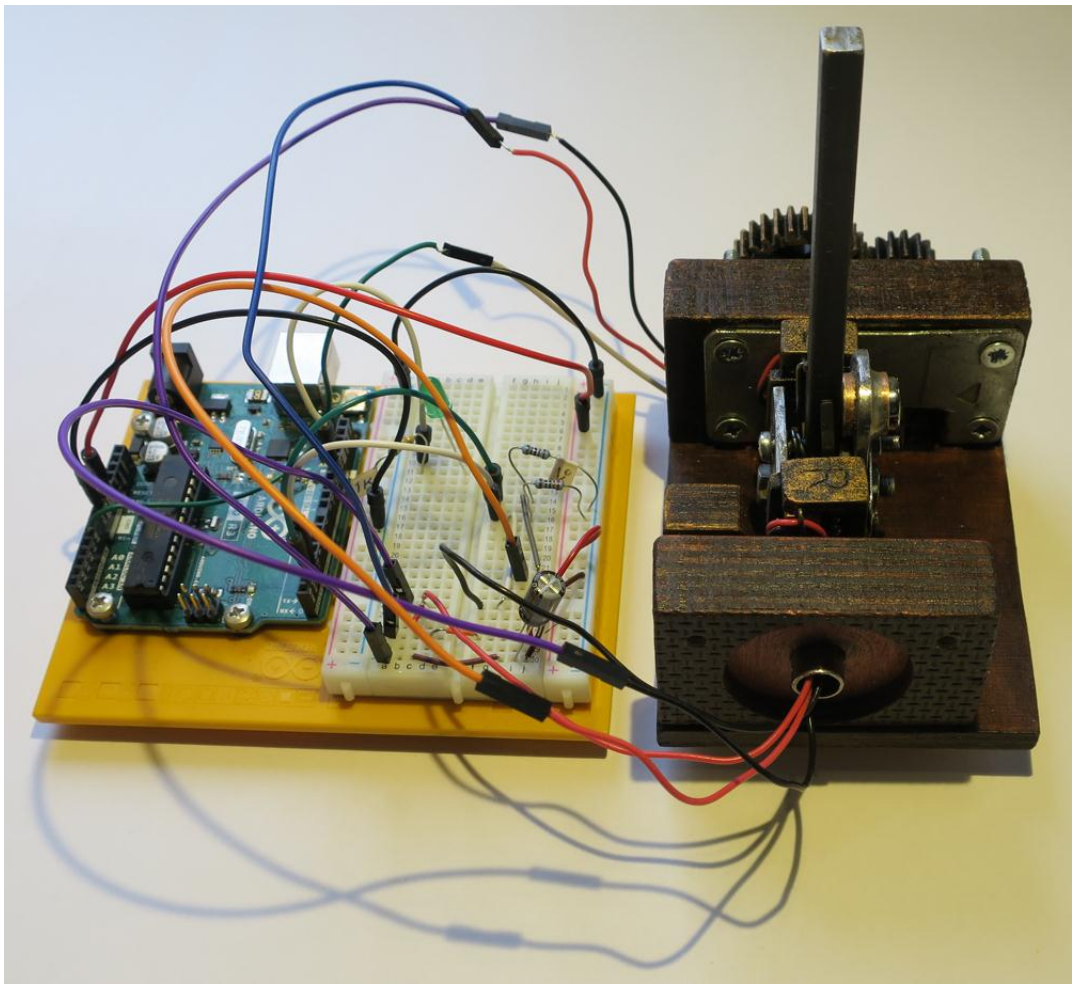


Figure 53.
The pre-test of the console was a tangle of wires and chaos management, even though every switch was not even attempted to fit onto the board simultaneously.

For the sake of clarity, an instructional diagram was first created for a Veroboard circuit, implemented with jumper wires and DuPont connectors, onto which capacitors, resistors, and pin headers would be soldered. Then, a piece of Veroboard was cut to size and the components were soldered onto it according to the pictorial assembly instructions (see Figure 54).

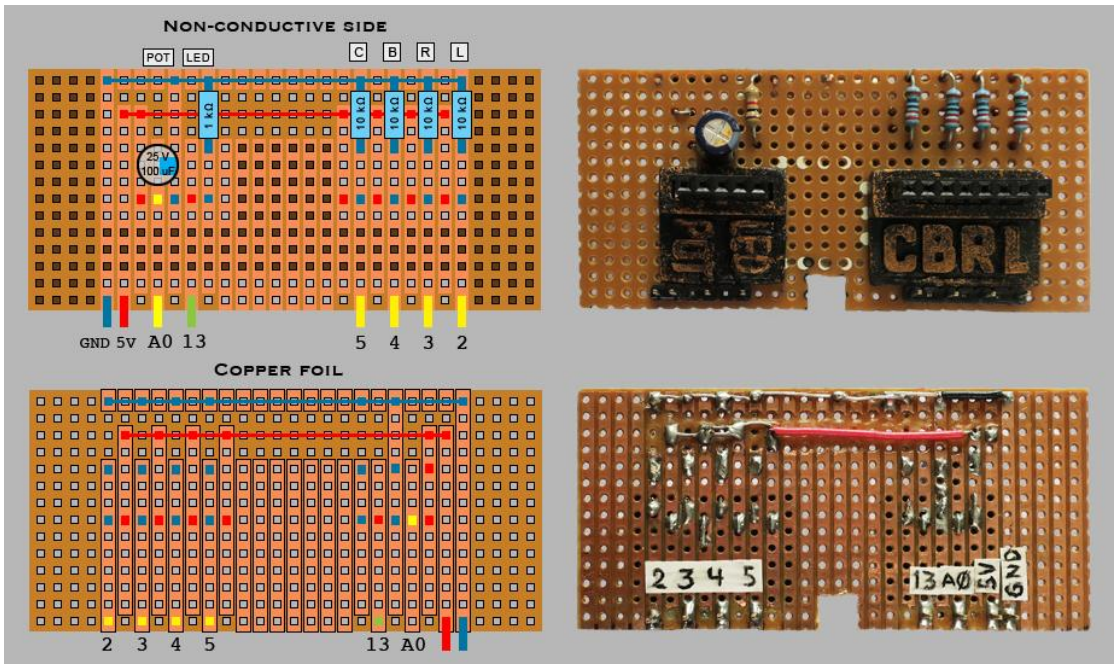


Figure 54.
The circuit board design and its implementation. To support the connectors, pieces were 3D-printed from ABS plastic, with each connection type marked in raised letters: **Pot**entiometer, **Led**, **C**-button, **B**-button, **R**ight-switch, **L**eft-switch. On the other side of the board, various pins on the Arduino are also labeled (2, 3, 4, 5, 13, A0, 5V power supply, ground).

The completed circuit board was coated and attached to the console connector, and the functionality of the connections was tested once again before the final assembly of the device (see Figure 55).

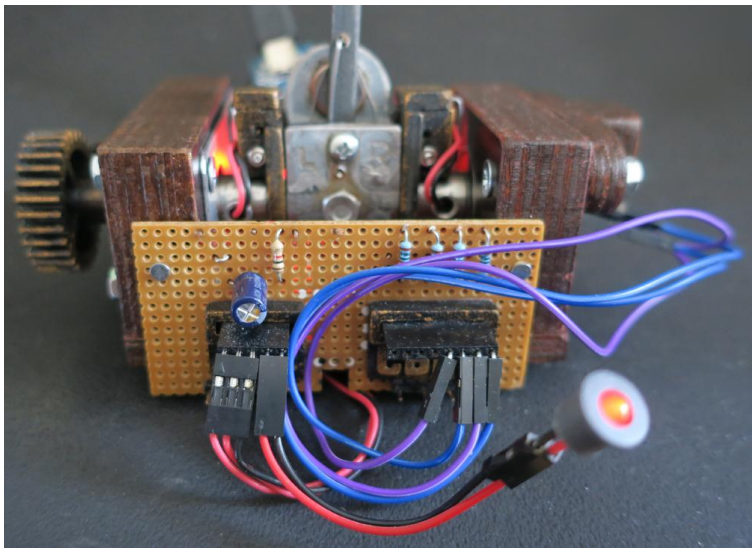


Figure 55.
A circuit board built on a stripboard connected to the mechanical core of the console. The LED that serves as a brake light lights up bright. Only the C and B buttons integrated in the case are not connected at this stage. They are only connected during the assembly phase of the device.

10.4 Enclosure

The casing was primarily made from 20 mm thick boards, with a base plate of 4 mm plywood. The cheek pieces were constructed in layers from two boards glued together. Using separate pieces made it easier to shape them to mimic the device's core, as the inner layers of wood could be simply cut away with a jigsaw to match the core profile. If the sides had been made from a single piece, a relatively complex concave shape would have had to be carved with a chisel and router.

The parts of the casing are joined together with wooden dowels, and the plywood base is attached to the casing with four wood screws. This keeps the casing securely assembled, yet it can still be easily disassembled if needed. The integrated buttons and hemispherical cable dome were modeled in Blender and 3D-printed from ABS plastic (Figure 56).

Figure 56.
The steel brushed wooden parts of the case were stained with tinted wood wax, and the final surface gloss was achieved with teak oil. The plastic parts were primed black, then dabbed with brass color, and finally finished with a semi-gloss lacquer.

The wires connecting to the Veroboard were soldered at one end to the microcontroller board which was attached to the base plate with small bolts and nuts. Through the integrated USB port at the other end of the circuit board, the device receives its operating power and communicates with a computer connected to the port (Figure 57).

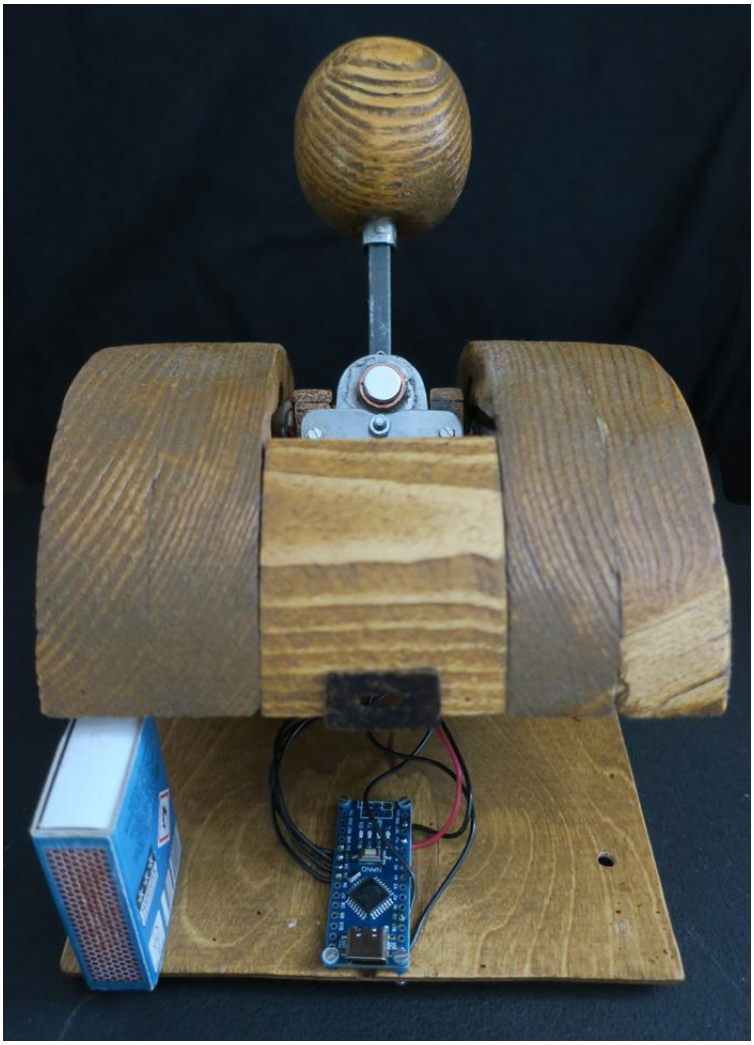


Figure 57.
The mounting board is only a fraction of the size of the development board. The picture clearly shows the smallness of the stamp-sized heart of the device.

Although it was mentioned at the beginning of the section that no actual preliminary design drawings were made for the device, creating them afterwards was much easier using the now-familiar Blender. The placement of the electronic components in relation to the device's body and casing is shown in appendix 6.

10.5 Protocol

To enable the track world created in Unity to communicate with the controller connected to the computer, a simple character-based communication protocol was developed between them (Figure 58).

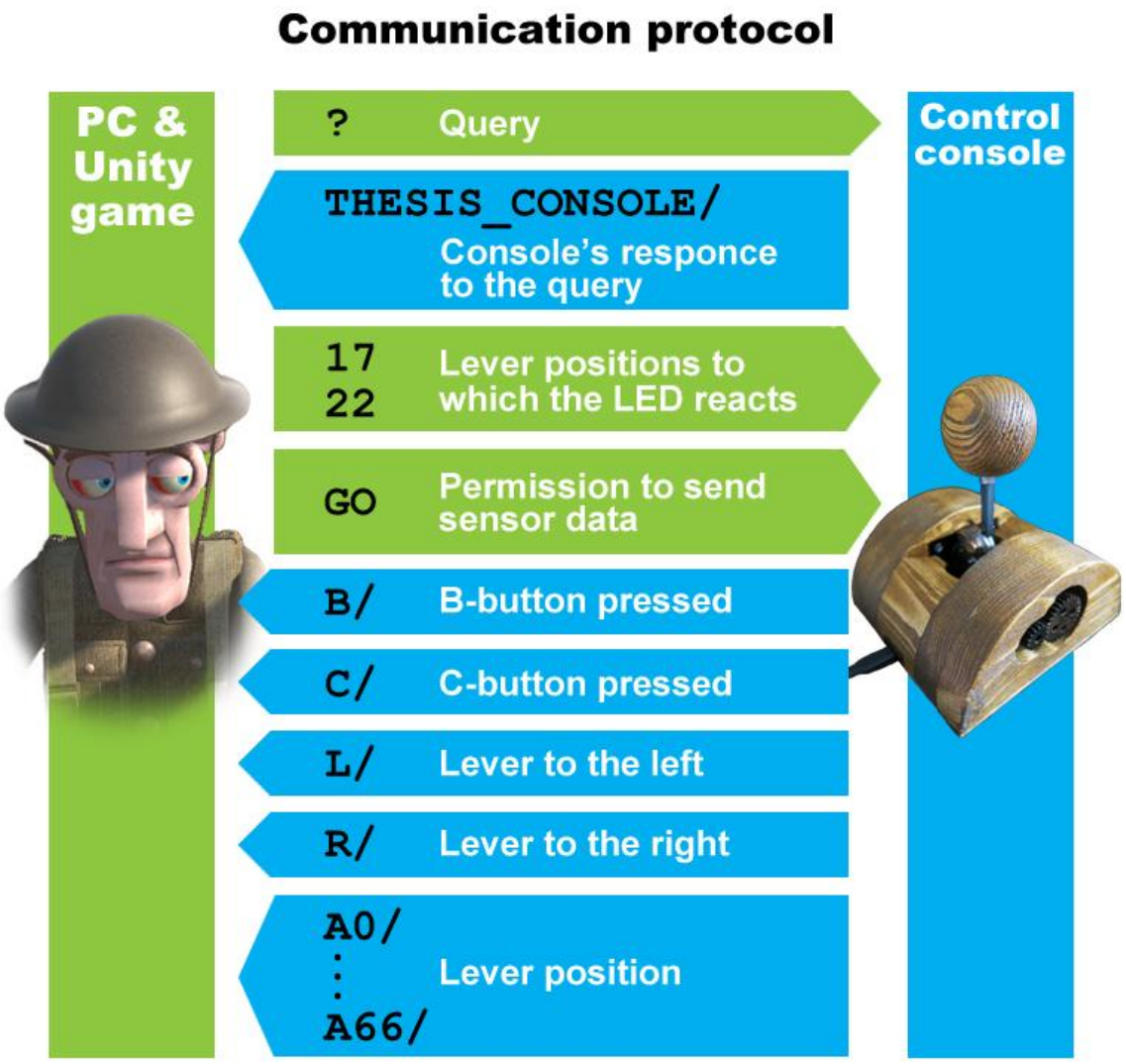


Figure 58.
First, the Unity application ensures it is communicating with the correct device by sending a question mark to the serial port. A strange device probably answers nothing, but the console responds by identifying its type. If the type name ('THESIS_CONSOLE' used in this case) matches the expectations, the application sends the threshold values for lighting up the LED and the 'GO' command to the console. After that, the console starts sending character-encoded information about the user's actions to the application. The messages sent by the console are separated by a slash.

10.6 Operating Logic

The console's program code was written in C++ using the Arduino IDE (appendix 7). The C++ language, derived from C, is a so-called hybrid language that supports both object-oriented programming and more traditional procedural logic (36 p16). The control console logic follows a procedural style.

Creating program logic for environments like Arduino is reminiscent of the 1980s in the sense that both the processor's performance and memory storage capacity are very limited. Therefore, the code should be kept simple and the logic compact; indeed, the program's logic is very straightforward. First, the device reads incoming messages from the serial port and responds to them according to the protocol. Second, the device writes events registered by its sensors to the serial port.

In terms of the performance of the PC and console setup, the aforementioned serial port predictably became the bottleneck. Data transfer between the devices was kept interference-free at a speed of 19200 baud, so the console was given time to process the data packets sent to it. Continuously bombarding the console listening on the serial port, caused interruptions in the command chain, so messages sent to the console were synchronized by adding a small delay between different commands.

The console was also programmed to use the serial port sparingly. Regarding the buttons, information is sent to the computer only when there is an actual change in the state of each button. The console code has the same kind of game loop as the Unity application, so each press would endlessly trigger information about the sensor’s current state unless such noise was somehow filtered out.

The potentiometer used, on the other hand, is high-precision and capable of producing just over a thousand different voltage readings. The resolution was unnecessarily fine-grained, so it was scaled in the C++ code by dividing the

value by 15. When the lever is turned, its scaled value is compared to the previous value sent to the serial port, and information about the lever's position change is sent to the computer only if it has changed from the previous value.

When the lever is turned to the so-called braking zone, the console code turns on the red LED, and when exiting the zone, it turns off the LED. The code responsible for braking the roller coaster car on the Unity side does not send braking information to the console. This was attempted, but the LED lit and extinguished too sluggishly. The current solution is fast and straightforward: the console does not react to events in the Unity game; it simply sends readings of its sensor states, and the game interprets them according to its own rules: the B button turns on/off the headlights, the C button changes the camera view, the L and R switches set the next track switch to left or right, and the potentiometer values increase speed forward/backward or trigger the handbrake when reaching the aforementioned braking zone.

10.7 Settings

A user interface for console settings and calibration was added to the 3D application implemented in Unity. The calibration view can be opened by pressing the letter “k” on the keyboard (Figure 59).

In the console search, a filter condition can be entered to speed up the device search. Once the console is identified, the extreme positions of the control lever's movement range can be determined by turning the lever back and forth as far as it will go.

The braking point can be set in the same way by turning the lever to the desired

Figure 59.
Console calibration view.

position. The setting algorithm calculates a margin of two units in both directions for the position. The resulting low and high values are sent to the console, which then turns the brake light on and off when the lever enters or exits the area between these two values.

Lastly, there are the timeouts for connection establishment and messages in tenths of a second. These values are used for synchronizing the messages sent to the console.

10.8 Serialization and Deserialization

As easy as calibrating the console is, it would certainly be convenient if the settings of a calibrated device could be saved. With this in mind, a class was created that can save the console settings to the hard drive and read the settings from the drive to be loaded onto the console. In practice, it is just a record description implemented as a class:

```
[Serializable]
public class KonsoliRec
{
  public string name {get; set;}
  public int min {get; set;}
  public int max {get; set;}
  public int low {get; set;}
  public int high {get; set;}
  public int port_ds {get; set;}
  public int general_ds {get; set;}
  public KonsoliRec() { }
 }
```

A list object was also created for the console records, allowing records for multiple devices to be stored:

```
private List<KonsoliRec> konsolit = new List<KonsoliRec>();
```

When the records compiled in the list need to be written to the drive, an instance of an XML serializer is created, and the task of data storage is delegated to it:

```
XmlSerializer serializer = new XmlSerializer(typeof(List<KonsoliRec>));
StreamWriter sw = new StreamWriter("konsolit.xml");
serializer.Serialize(sw, konsolit);
sw.Close();
```

Reading the stored data into a list containing instances of the record class is very similar to storing the data:

```
XmlSerializer serializer = new XmlSerializer(typeof(List<KonsoliRec>));
StreamReader sr = new StreamReader("konsolit.xml");
konsolit = (List<KonsoliRec>)serializer.Deserialize(sr);
sr.Close();
```

As a result of serialization, an XML file is created in this case with only one record, because there is only one device so far:

```
<?xml version="1.0" encoding="utf-8"?>
<ArrayOfKonsoliRec xmlns:xsd=http://www.w3.org/2001/XMLSchema
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <KonsoliRec>
    <name>/dev/tty.wchusbserial1440</name>
    <min>0</min>
    <max>65</max>
    <low>22</low>
    <high>26</high>
    <port_ds>30</port_ds>
    <general_ds>20</general_ds>
  </KonsoliRec>
</ArrayOfKonsoliRec>
```

Transforming code objects into persistent objects, based on what has been learned above, is a fairly easy operation in C#. Just as the console configuration was serialized into an XML file, all the characters in the 3D world, such as the rocking cart and whistling bullets, could be saved to the drive on the fly. The object classes to be saved would just need to be made serializable. This is definitely something worth trying from the perspective of further developing 3D games.

10.9 User Experience

Having played various games with all sorts of controllers, it felt exciting to try out a self-made controller for the first time. Would it be as wobbly as many steering wheels, slipping off the edge of the table at the worst possible moment in a hairpin turn of a racing game, or would it simply be unintuitive and unsuitable for the program it was built to control?

The worry about wobbliness disappeared immediately, as the weight of the control console and the rubberized feet embedded in the base plate kept the controller firmly in place. There was also no fear of unintuitiveness. Steering with the console was in a completely different league compared to using a keyboard. Steering forwards and backwards, as well as selecting direction at crossing tracks, could be done without lifting a hand from the controller lever. When fumbling with the keyboard, mispresses were more the rule than the exception. Similarly, braking the carriage worked well – installing a red brake light on the controller was a good idea in its clarity.

In Unity, the selfie camera placed in the carriage showed how the avatar steering the carriage faithfully mimicked the hand movements of the human user (see Figure 60). In short, it was a dizzying experience and encourages the development of a more versatile keyboard substitute for action games as well.

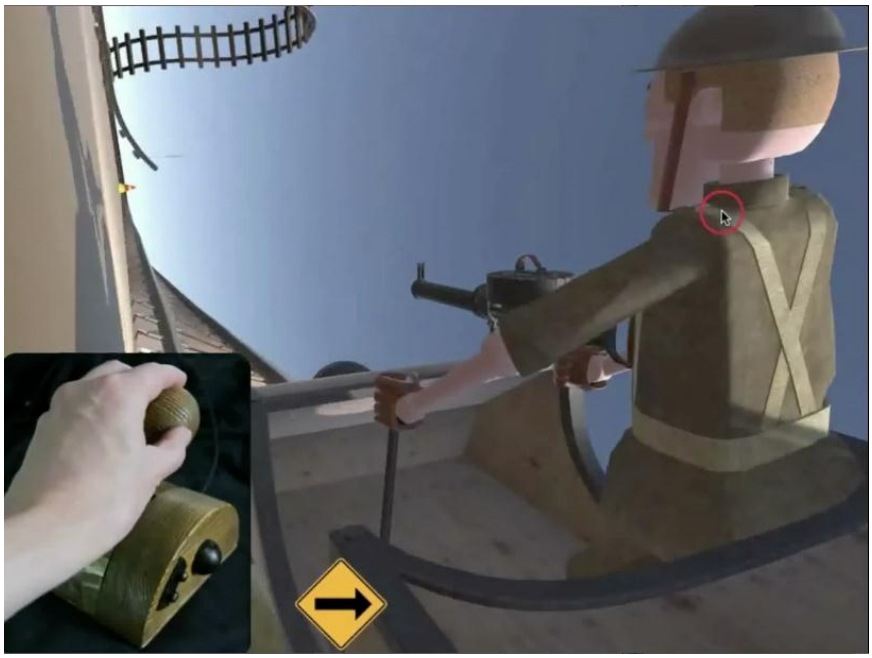


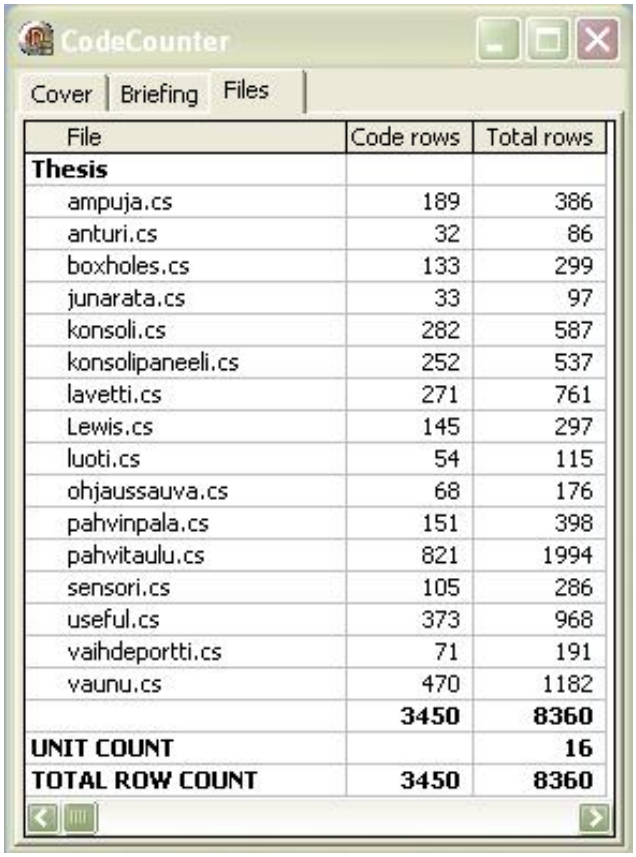
Figure 60. The avatar faithfully mimics the user's steering movements.

11 Discussions and Conclusions

One of the research objectives of this thesis was to evaluate the efficiency of new tools. Since it involved a programming environment that was new to the author, the work took more time than the number of lines of code written would suggest. This was somewhat anticipated beforehand. The lines of source code were counted using the Code Counter program developed by Tietomato Oy for this purpose at the beginning of the millennium, now optimized for measuring C# code.

11.1 Cost-effectiveness

The counting program excluded comments and lines with curly braces that denote do-end structures. Additionally, it condensed commands split across multiple lines into a single line. Calculated this way, there were 16 source code units with a total of 3450 lines of source code (Figure 61).



File	Code rows	Total rows
Thesis		
ampuja.cs	189	386
anturi.cs	32	86
boxholes.cs	133	299
junarata.cs	33	97
konsoli.cs	282	587
konsolipaneeli.cs	252	537
lavetti.cs	271	761
Lewis.cs	145	297
luoti.cs	54	115
ohjaussauva.cs	68	176
pahvinpala.cs	151	398
pahvitaulu.cs	821	1994
sensori.cs	105	286
useful.cs	373	968
vaihdeportti.cs	71	191
vaunu.cs	470	1182
	3450	**8360**
UNIT COUNT		**16**
TOTAL ROW COUNT	**3450**	**8360**

Figure 61.
The number of lines of source code used to implement the 3D world, as calculated by the CodeCounter program.

Programming the Arduino microcontroller, on the other hand, required 99 lines using the same condensed measurement method.

A little over 3500 lines in total is a small amount by any measure, considering what was

accomplished with that code. In other words, Unity proved to be an efficient and excellent addition to Tietomato Oy's toolkit.

Blender, with which there was already previous experience, also justified its cost. It was used to model both the characters of the 3D world and the gears, buttons, and other necessary parts printed on the control console, which were not practical to work from wood or steel. Finally, Blender was used to create a conceptual image of the console with its electronic components. The idea that Blender is a godsend for the poor Indie developer was only reinforced during this work.

11.2 Achieving the Objectives

The goal of constructing a small-scale 3D world with avatar characters was achieved successfully, even exceeding expectations in the sense that users are able to customize their environment to some extent in ways not predetermined. In commercial applications, it will still be beneficial to significantly reduce the mesh complexity of meticulously modeled objects (like the Lewis rapid rifle with over 11,500 vertices) to a fraction of the modeling precision used in this project, mainly for performance reasons.

Furthermore, the control console was built and seamlessly integrated into the 3D world. Its physical appearance blends harmoniously with the weathered, roller coaster-dominated environment. Moreover, the device functions effectively. While not as complex as a rocket ship, it stands as a testament to a certain level of skill and ingenuity.

While the work was challenging in many respects and required extensive research, alongside a fair amount of luck, it was never insurmountable. Much credit goes to the outstanding teachers and unbeatable courses at author’s alma mater.

Reference List

1 Dylan B. Blowin' in the Wind. New York: The Freewheelin' Bob Dylan. Columbia Recording Studios. 1962.

2 Open letter of the updates to the Unity Runtime Fee policy. Unity Technologies. 2023 Sep 22.

3 Lappi I. "Bonk-museo avaa uusia näköaloja Suomen historiaan" [Internet]. 2023. Available from: https://aamuset.fi/artikkeli/6010057. Kaupunkimedia Aamuset.

4 Beard H, Kenney D. Bored of the Rings. London: The Harvard Lampoon; 1969. 228p.

5 The Software [Internet]. 2023. Available from: https://www.blender.org/about/. Blender Foundation.

6 Introduction to Design Software [Internet]. 2023. Available from: https://guides.bpl.org/design/graphics. Boston Public Library.

7 Dealessandri M. "What is the best game engine: is Unity right for you?" [Internet]. 2020 [cited 2020 April 4]. Available from: https://www.gameindustry.biz. Gamer Network.

8 Axon S. "Unity at 10: For better – or worse – game development has never been easier" [Internet]. Ars Technica. 2016. Available from: https://arstechinica.com/gaming/2016/09/.

9 Government & Aerospace. Unity [Internet]. 2021. Available from: https://unity.com/solutions/governement-aerospace. Unity.com.

10 Kennedy JF. We Choose the Moon speech. Rice University. 1962 Sep 12.

11 Hogg IV. The Encyclopedia of Weaponry. London: Quarto Publishing plc; 1992. 224p

12 The Vintage Aviator Ltd. New Zealand [Internet] Available from: https://thevintageaviator.co.nz/projects/lewis-gun-mk2.

13 Developer's Guide Borland Delphi 5 for Windows 98, Windows 95 & Windows NT. Scotts Valley: Inprise Corporation; 1999. 760p.

14 Game Programming Patterns. Available from: https://gameprogrammingpatterns.com.

15 Unity User Manual. 2022.3 (LTS) [Internet]. Available from: https://docs.unity3d.com/Manual/.

16 Kunnas K. Tumpkin's Wonder Tree. WSOY. 1956. English translation by Happonen S. "The Witch on a Vespa" [Internet]. 2017. Available from: https://helda.helsinki.fi/.

17 Jowett B. The Republic by Plato. Andrews UK Limited; 2012. 385p.

18 Gaukroger S, Schuster J, Sutton J. Descartes' Natural Philosophy. London: Routledge; 2000. 780p.

19 Dunn F, Parberry I. 3D Math Primer for Graphics and Game Development. Second edition. CRC Press; 2011. 824p.

20 The Royal Irish Academy. [Internet]. Available from: https://www.ria.ie/hamilton-did-it/hamiltons-quaternions.

21 ChatGPT; 2024, [Internet]. Available from: https://chat.openai.com.

22 Laiho A. On a lecture of Metropolia's course of 3D-graphics. 2011. Espoo, Leppävaara's campus.

23 Mullen T. Introducing Character animation with Blender. Wiley Publishing Inc; 2007, 478p.

24 Red Storm Entertainment, Ubisoft. Tom Clancy's Ghost Recon, 2001. Video game.

25 Red Storm Entertainment, Ubisoft. Tom Clancy's Ghost Recon Advanced Warfighter 2 (GRAW), 2007. Video game.

26 Collodi C. Le avventure di Pinocchio. Firenze: Felice Paggi. 1883. 236p.

27 Von Goethe JW. Faust. New York: Algora Publishing; 2016. 424p.

28 Vonnegut K. The Sirens of Titan. New York: Dell Publishing; 1959. 319p.

29 Army Command. Small Arms Manual 2019. Tampere: PunaMusta Oy; 2019. 228p.

30 Unity Scripting API. 2022.3) [Internet]. Available from: https://docs.unity3d.com/ScriptReference/.

31 Monolith Productions. Blood. Europe: Eidos Interactive; 1997. Video game.

32 Sedgewick R. Algorithms, second edition. Addison-Wesley Publishing Company, Inc; 1989. 655p.

33 The Wachowskis. Matrix. Science fiction film. Warner Bros.1999. 136m.

34 Defoe D. Robinson Crusoe. Glasgow: The Grant Educational Co. Ltd.; 1719. 256p.

35 Fitzgerald S, Shiloh M, Igoe. The Arduino Projects Book. Arduino.CC; 2015. 170p.

36 Hietanen P. C++ ja olio-ohjelmointi; Porvoo: Docendo; 2001. 830p.

Program code of KeySwiz class

KeySwiz.cs

```
public class KeySwiz
{
    private System.Action<bool> Operation { get; set; }
    private bool mbPressed; // Button state (up/down)
    public  bool State;     // Switch state (on/off)

    public KeySwiz(bool State, System.Action<bool> operation)
    {
        this.State = State;
        this.Operation = operation;
        mbPressed = false;
    }
/* The switch functions like a ballpoint pen. Whenever the button is pressed
    down from the top position, it toggles the switch state on or off.
    Continuous pressing of the button does not change the state; instead, it
    needs to be released periodically, i.e., brought back to the top position. */
    public void press()
    {
        if (!mbPressed)
        {
            mbPressed = true;
            State = !State;
            this.Operation(State);
        }
    }
/* The button is released, causing the button's state to change, but not the
   switch's state. The switch state changes only with the next button press or
   by directly altering the state value. */
    public void release()
    {
        mbPressed = false;
    }
    public bool isPressed()
    {
        return mbPressed;
    }
} // EOF KeySwiz class
```

So little code, but big impact; a sieve that removes the echoes of commands from the time, that flows rapidly through the game loop.

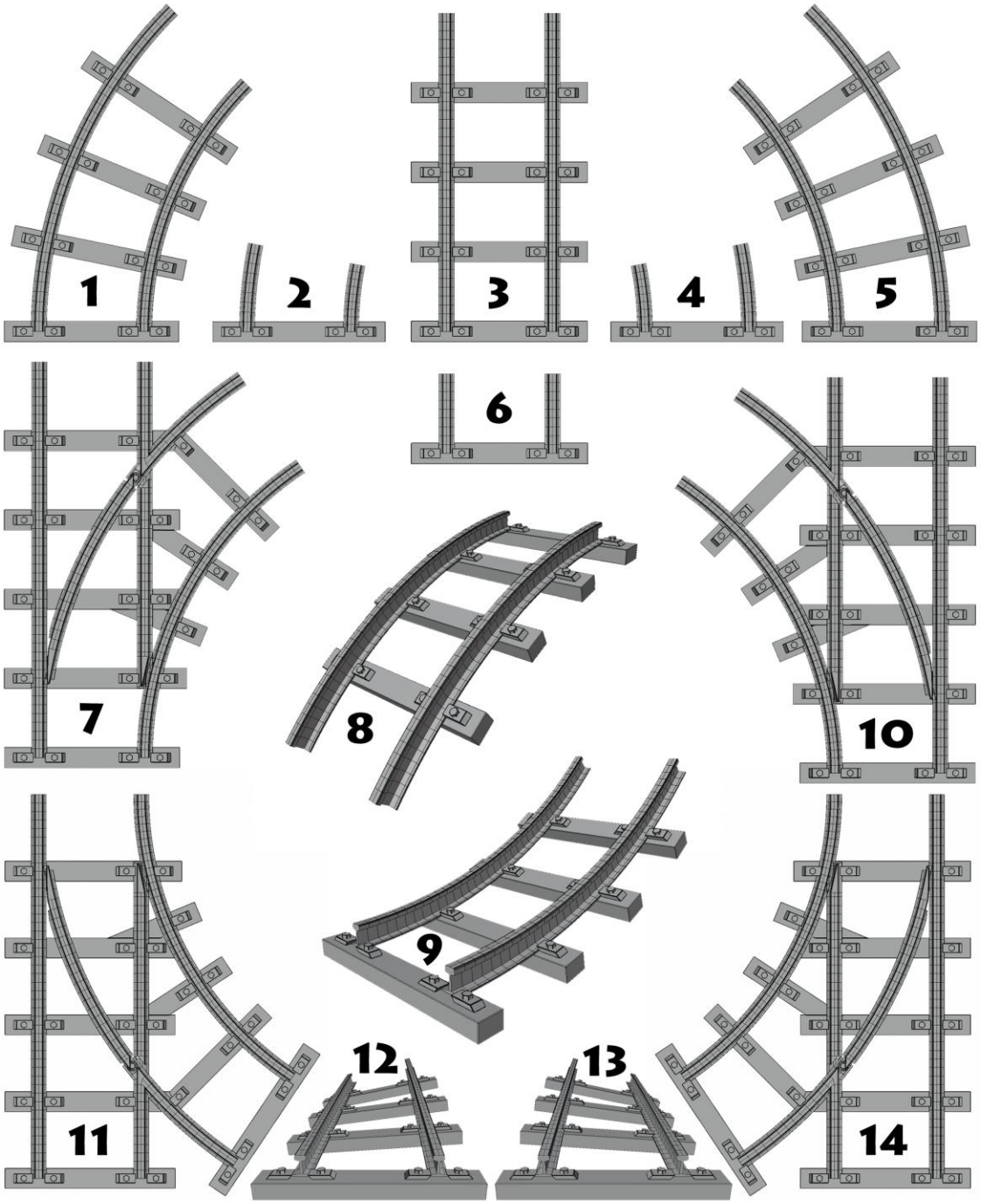
Program code of CycleSwiz class

CycleSwiz.cs

```
public class CycleSwiz
{
    private System.Action<int> Operation { get; set; }
    private bool mbPressed; // Button state (up/down)
    public  int Cycles;     // Cycle max count
    private int cycle_count = 0;
    public CycleSwiz(int cycles, System.Action<int> operation)
    {
        this.Operation = operation;
        this.Cycles = cycles;
        mbPressed = false;
    }
// When the switch is pressed, cycle messages start to flow.
    public void press()
    {
        mbPressed = true;
    }
    public void release()
    {
        mbPressed = false;
    }
/* A cycle message is sent one after the other to the listener until the
    button's status is no longer Pressed. Since the object is not hung directly
    in the game loop, this method must be called in the FixedUpdate of the
    object's owner! */
    public void fixu()
    {
        if (!mbPressed && cycle_count == 0)
        {    /* Do nothing */   }
        else
        {
            this.Operation(cycle_count);
            cycle_count++;
            if (cycle_count == Cycles)
            {
                cycle_count = 0;
            }
        }
    }
} // EOF CycleSwiz class
```

Channeling the time flow into an easier-to-manage form, this produces cycles of the desired size.

Rail Elements



1) Curve to the right
2) Short curve to the right
3) Straight rails
4) Short curve to the left
5) Curve to the left
6) Short straight rails
7) Track switch to the right
8) Downhill tracks
9) Uphill tracks
10) Track switch to the left
11) Return switch from the right
12) Longitudinal twist to the left
13) Longitudinal twist to the right
14) Return switch from the left.

Code Snippet of the Bullet

```
/*
-----------------------------------------------------------------------------
The Game Loop calls FixedUpdate 50 times per second
-----------------------------------------------------------------------------
*/
  void FixedUpdate()
  {
    if (Time.time - borntime >= lifetime || mnHitCount >= HIT_MAX)
    {
      Destroy(this.gameObject);
    }
    else
    {
      float distance = velocityPS * Time.deltaTime;
      Rigidbody rb = this.GetComponent<Rigidbody>();
      Vector3 v = this.transform.localPosition;
      Vector3 dir = new Vector3(v.x, v.y, v.z + distance);
      dir = (dir - v).normalized;
      dir = rb.transform.TransformDirection(dir);
      RaycastHit hit;
      if (Physics.Raycast(this.transform.position, dir, out hit, distance, LayerMask))
      {
        distance = hit.distance;
        mnHitCount++;
      }
      Vector3 pos = this.transform.position;
      this.transform.Translate(Vector3.forward * distance);
      if (fup > 0)
      {
        lineRenderer.enabled = true;
        lineRenderer.SetPosition(0, mvPrevPos);
        lineRenderer.SetPosition(1, this.transform.position);
      }
      mvPrevPos = new Vector3(this.transform.position.x + mfRadius,
                              this.transform.position.y + mfRadius, this.transform.position.z);
    }
    fup++;
  }
```

The to-do list of the game's most smoothly moving object, which it executes tens of times per second. It includes probing the flight path, adjusting the flight distance, collision detection, and ultimately initiating the self-destruct mechanism.

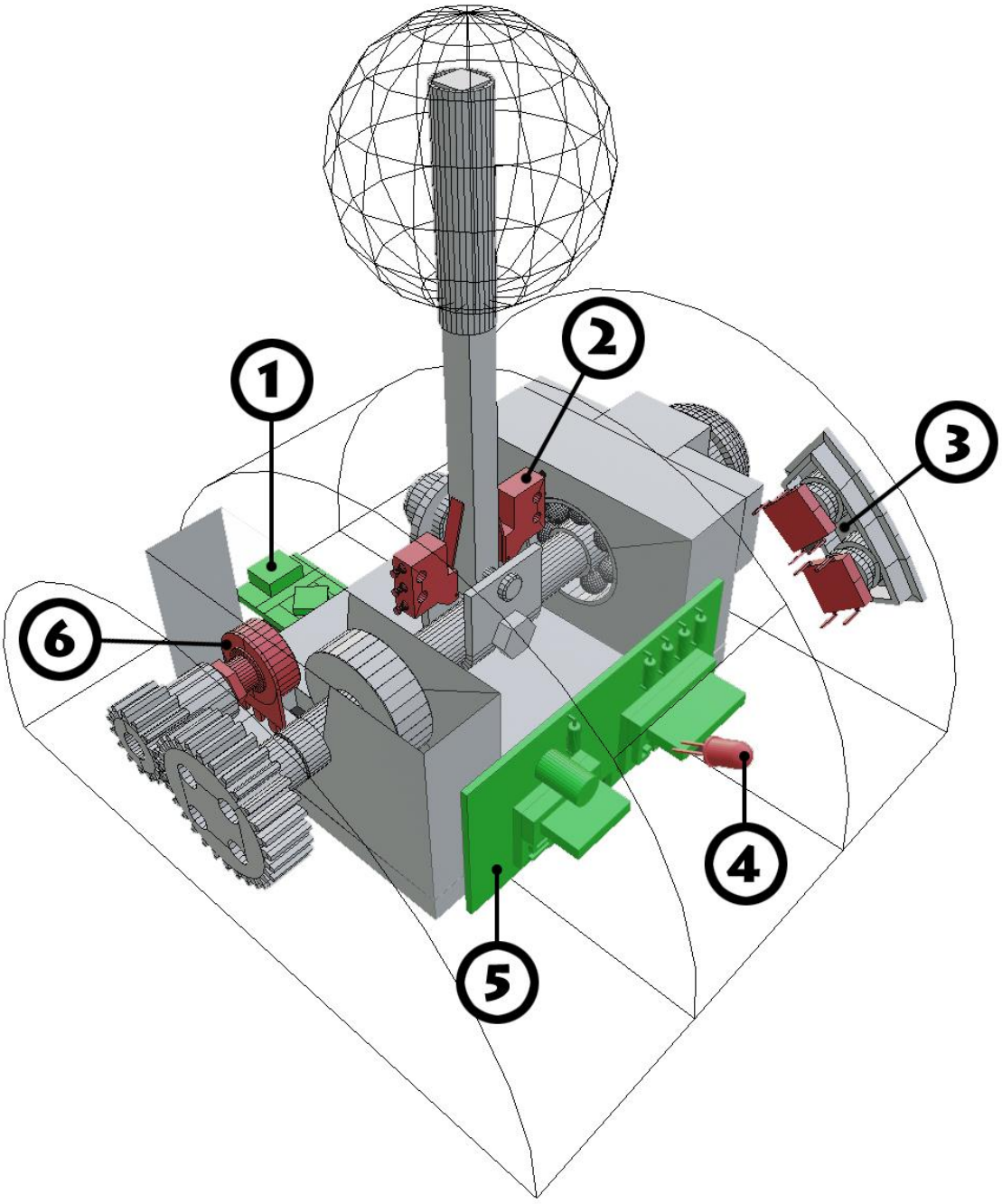
Placement Algorithm for Bullet Impact Marks

```
/*
-------------------------------------------------------------------------------
Calculate the fracture location and cutting requirement
-------------------------------------------------------------------------------
*/
  private HolePos holePos(float pos, float fSize)
  {
//  Let's examine whether the fracture extends beyond the box edges horizontally.
    HolePos res = new HolePos();
    int xy = 0; //
    int wh = HOLE_PIX_SIZE;
    float fPivot = 0.5f;
    float fCrossing = 0;
    float fLeftOver = 0;
    float f = Mathf.Abs(pos) + mfHoleSize/2;

//  The mark is trimmed from both sides if it is on a surface narrower than itself.
    if (mfHoleSize > fSize)
    {
      fPivot = -0.5f; // The negative value indicates to the caller a narrow wedge!
      xy = (int)Mathf.Round((mfHoleSize - fSize) * HOLE_PIX_SIZE / (2 * mfHoleSize));
      wh = (int)Mathf.Round(fSize * HOLE_PIX_SIZE / mfHoleSize);
    }
    else if (f > fSize / 2)
    {
      fCrossing = f - fSize / 2;          // The part crossing over the edge of the object
      fLeftOver = mfHoleSize - fCrossing; // The width of the remaining part of the hit mark
      f = fLeftOver / mfHoleSize; // The size of the leftover piece in relation to the whole mark
      wh = (int)Mathf.Round(f * HOLE_PIX_SIZE); // The size of the leftover piece in pixels
      if (pos > 0) // Right or top
      {
        xy = 0;
        fPivot = 1.0f;
      }
      else // Left or bottom
      {
        xy = HOLE_PIX_SIZE - wh;
        fPivot = 0;
      }
    }
    res.pivot   = fPivot;
    res.pixPos  = xy;
    res.pixSize = wh;
    return res;
  }
```

The holePos algorithm of the boxholes class returns the clipping requirement for the impact pattern when the hit occurs at the edge of a flat surface. The algorithm is called twice for each impact: once to calculate the horizontal correction and once for the vertical correction (or their relative equivalents).

Electronic Components of the Console



1) ATMega328P microcontroller board
2) Two microswitches
3) Two membrane switches
4) LED
5) Circuit board
6) A potentiometer.

Control Console's Logic

```
/*
-------------------------------------------------------------------------------
Thesis Console
Company: Tietomato Oy / Codger Corporation.
Author:  Tapio Lehtimäki
Years:   2023-2024
-------------------------------------------------------------------------------
*/
String inStr;
int pot_value = 0;
bool busy = false;
bool readyToGo = false;
String endStr = "/";

//Btn to left
int valueLeft = 0;
int prevLeft = -1;
unsigned long timeLeft = 0;

// Btn to right
int valueRight = 0;
int prevRight = -1;
unsigned long timeRight = 0;

// Btn B
int valueB = 0;
int prevB = -1;
unsigned long timeB = 0;

// Btn C
int valueC = 0;
int prevC = -1;
unsigned long timeC = 0;

int mnLedLow = 30;
int mnLedHigh = 30;
int miLed = 0;

const String S_NAME = "THESIS_CONSOLE";
const String S_QUERY = "?";
const String S_GO = "GO";
const int BTN_TOLERANCE_MS = 200;
const int BAUDRATE = 19200;
const int POT_TOLERANCE = 15; // Limit potentiometer output from 0-1007 to 0-66
const int POT_MAX = 1000;
const int PIN_LED = 13;
const int PIN_BTN_LEFT = 2;
const int PIN_BTN_RIGHT = 3;
const int PIN_BTN_B = 4;
const int PIN_BTN_C = 5;
const String END_CHAR = "/";
const int PIN_POT = A0;
const String POT_PREFIX = "A";

/*
-------------------------------------------------------------------------------
Initialization.
-------------------------------------------------------------------------------
*/
void setup()
{
  Serial.begin(BAUDRATE);
  pinMode(PIN_LED, OUTPUT);
  digitalWrite(PIN_LED, LOW);
  pinMode(PIN_BTN_LEFT, INPUT);
  pinMode(PIN_BTN_RIGHT, INPUT);
  pinMode(PIN_BTN_B, INPUT);
  pinMode(PIN_BTN_C, INPUT);
}

```

```
/*
--------------------------------------------------------------------------------
An infinite loop where:
- Data from sensors is read, processed, and sent through the serial port to the
  entity using the device (Unity game).
- The data stream also received from the serial port.
--------------------------------------------------------------------------------
*/
void loop()
{
  if (Serial)
  {
    pot_value = potHandler(PIN_POT, pot_value, POT_PREFIX); // Pot values -> Unity
  }

  // Messages from Unity
  if ((Serial.available() > 0) && (!busy))
  {
    busy = true;
    inStr = Serial.readString();
    inStr.trim();
    inStr.toUpperCase();
    if (inStr == S_QUERY)
    {
      Serial.print(S_NAME + END_CHAR);
    }
    else if (inStr == S_GO)
    {
      readyToGo = true;
    }
    else /*  Brake light settings */
    {
      int n = inStr.toInt();
      if (miLed == 0)
      {
        mnLedLow = n;
        mnLedHigh = n;
        miLed = 1;
      }
      else if (miLed == 1)
      {
        mnLedHigh = n;
        if (mnLedHigh < mnLedLow)
        {
          mnLedHigh = mnLedLow;
        }
        miLed = 0;
      }
      else // Unknown message is echoed
      {
        Serial.print(inStr + END_CHAR);
      }
    }
    busy = false;
  }

  btnHandler(PIN_BTN_LEFT, "L", &valueLeft, &prevLeft, &timeLeft);
  btnHandler(PIN_BTN_RIGHT, "R", &valueRight, &prevRight, &timeRight);
  btnHandler(PIN_BTN_B, "B", &valueB, &prevB, &timeB);
  btnHandler(PIN_BTN_C, "C", &valueC, &prevC, &timeC);
}
```

```
/*
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
*/
int potHandler(int pin, int prevValue, String prefix)
{
  int res = prevValue;
  int value = analogRead(pin);
  if (value >= POT_MAX)
  {
    value = POT_MAX;
  }

  int n = abs(value - prevValue);
  if (n >= POT_TOLERANCE)
  {
    res = value;
    int scaled = value / POT_TOLERANCE; // Scaled value -> 0..66
    if (readyToGo)
    {
      if (scaled >= mnLedLow && scaled <= mnLedHigh)
      {
        digitalWrite(PIN_LED, HIGH); // Brake light on
      }
      else
      {
        digitalWrite(PIN_LED, LOW);  // Brake light off
      }
      Serial.print(prefix + scaled + END_CHAR);
    }
  }
  return res;
}

/*
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
*/
void btnHandler(int pin, String prefix, int* value, int* prev, unsigned long* timeBtn)
{
  *value = digitalRead(pin);
  if (*value != *prev)
  {
    *prev = *value;
    if (*value == LOW)
    {
      *timeBtn = millis();
    }
    else if (*value == HIGH && millis() - *timeBtn >= BTN_TOLERANCE_MS)
    {
      if (readyToGo)
      {
        Serial.print(prefix + END_CHAR);
      }
      *timeBtn = millis();
    }
  }
}
```