POLISH-JAPANESE ACADEMY OF INFORMATION TECHNOLOGY

Faculty of Computer Science

Department of Software Engineering

Software and Database Engineering

Jakub Radlak

album number 22558

Orbital Flight Simulator - educational program

Master Thesis
Supervisor
Mariusz Trzaska Ph. D.

Warsaw, February 2023

Abstract

The subject of the work is to present research aimed at developing an author's program that simulates orbital flights. The most important implementation details will also be discussed.

In this paper we will also discuss shortly physical basics of the orbital flight. Then we will move on to the description of the virtual machine, 3D visualization and a simple physical engine created especially for the orbital simulation program.

This work also describes the application from the user's perspective and possible educational or hobby applications of the created simulator. The article ends with an overview of the improvement plans and possible future developments of the application.

Acknowledgments

I would like to thank my family, my parents Marzena and Piotr Radlak, my sister Malgorzata Radlak, as well as my grandmother Krystyna Damek for their constant support and faith throughout my studies.

I would also like to thank my supervisor, Mr. Mariusz Trzaska, for his patience and help in preparing this thesis.

Contents

Chapter 1

Introduction

Spaceflight and rocketry are of interest to many enthusiasts of all ages. There are many hobby organizations that organize rocket building competitions and demonstrations. One of them is Polish Rocket Society (Polskie Towarzystwo Rakietowe) which organizes events, courses and actively promotes rocketry in the society. In author's opinion, such activities have a huge impact on increasing the awareness of critical thinking, creativity and deepening knowledge in the field of exact sciences, including physics [21].

1.1 Physics simulation controlled by the software

However, regardless of the undoubted advantages of such activities as building and launching amateur rockets - there are some obvious constrains. One of them is that it is almost impossible to build an amateur rocket that will fly into space. Therefore, there is a need to create a tool that gives a substitute for such a flight - the simulation software with credible behaviour of physics.

In addition to physics simulation, that software should also enable precise control of the rocket's flight, for example by writing programs that are executed by the simulation of the on-board computer of some kind. That approach will be compatible with actual on-board computers of spacecraft of the Apollo-era [13] [14].

In author's opinion, combining physics simulation with control by a computer program written by the user is not just fun. It gives another level of understanding of applied computer science. Writing computer programs for such a specific purpose with certain explicit constraints and applied under real-world real-time conditions is quite similar to real-time embedded system programming (in cars, for example).

1.2 Purpose of this work

This paper describes the path that the author has traveled to achieve the goals outlined in the above section. We will discuss some of the technical research and development that had to be undertaken to deliver a working prototype.

1.3 Work results

The result of this work is primarily a discussion of the technical and conceptual aspects of the implementation of the Orbital Flight Simulator (OFS) application. Bearing in mind that OFS must provide a reasonable and useful set of functions. This article also discusses some of the technical research that

had to be done to come up with a solution: e.g.: solving the problem of the size of the world. The working application itself is a side effect of these activities.

1.4 Work organisation

This paper is divided into chapters as follows. Each chapter describes a step or steps taken to implement the final solution. Some chapters cover more theoretical aspects, while others focus on describing the exact solution to practical problems during the development process. However, the author tries to avoid describing things that are not specific to the solution or can be found in the attached literature.

In chapter 2, we will discuss some examples of applications that solve similar problems that we are trying to solve with our simulator. In addition, we will present selected, well-known examples of virtual machines and approaches to their implementation.

In chapter 3 based on the discussion in the second chapter, we outline the main ideas and requirements for our solution. In this chapter we will present the basic shape of our solution.

From the chapter 4, we begin to build our toolkit by starting with an intuitive description of the orbital flight problem. In this chapter, we will also derive the equations of motion for our rocket.

The chapter 5 provides a detailed overview of the internal structure and functions of the embedded virtual machine. We will also discuss the proposed assembly language.

Chapter 6 describes issues related to three-dimensional simulation visualization. The chapter also contains an overview of the most specific problems related to our solution. More general topics that can be found in the literature will only be mentioned. We will also explain why the work involves implementing our own graphics engine instead of using one of the existing solutions.

In the chapter 7, we will discuss our simulation physics engine, which solves rocket equations of motion in real time. As in the previous chapter, we will discuss the reasons why we decided to create our own solution instead of using existing ones.

Chapter 8 describes the languages, libraries and other programming tools used to implement the solution. We will also discuss the reasons for choosing this particular set of tools. In addition, we will present a short comparative analysis of the available toolkits.

In the chapter 9, we will discuss the user interface and features of the application from the user's perspective.

And finally, in the last chapter 10, we summarize the whole work and present the final conclusions. We will also describe the possible development paths of the application in the future and the educational aspects of the solution.

Chapter 2

Overview of existing solutions

There are at least two alternative, widely available software solutions to the orbital simulation problem. Both of them approach this problem in a different way. Orbital Flight Simulator is also unique in its own way. In this chapter, we will discuss the similarities and differences between this three solutions.

For a better picture and understanding, in this chapter we will compare some features of the existing solutions with the features implemented in our Orbital Flight Simulator.

2.1 Kerbal Space Program

Kerbal Space Program is an advanced space program simulation software with a touch of self-irony. Technically, KSP is a video game, but it features sophisticated physics simulation and accurate orbital mechanics implementation. The figure 2.1 shows the program's user interface.

Figure 2.1: Kerbal Space Program, source: self-elaboration (screenshot)

Kerbal Space Program contains some features that are absent in Orbital Flight Simulator (OFS) described in this paper. First of all, KSP allows users to build their own spacecraft from a set of prefabricated components. Another feature is the ability to explore the entire solar system, not just orbits near the main planet. Although there are plans to include this feature in OFS as well. KSP is also, understandably, much more sophisticated in terms of visuals (although its graphics are maintained in a cartoon style - OFS focuses on realism) [18].

On the other hand OFS contains features that are absent in KSP. One of them is the presence of the Virtual Machine and the ability to program the flight of the rocket with precision unattainable in Kerbal Space Program. This gives OFS additional educational applications and is supposed to bring our solution closer to real solutions known from spaceships. Real spaceships are almost entirely controlled by machines.

2.2 Orbiter 2016 simulator

Another good example of space flight simulator is a free software called Orbiter 2016. Example image of its user interface is shown in the figure 2.2.

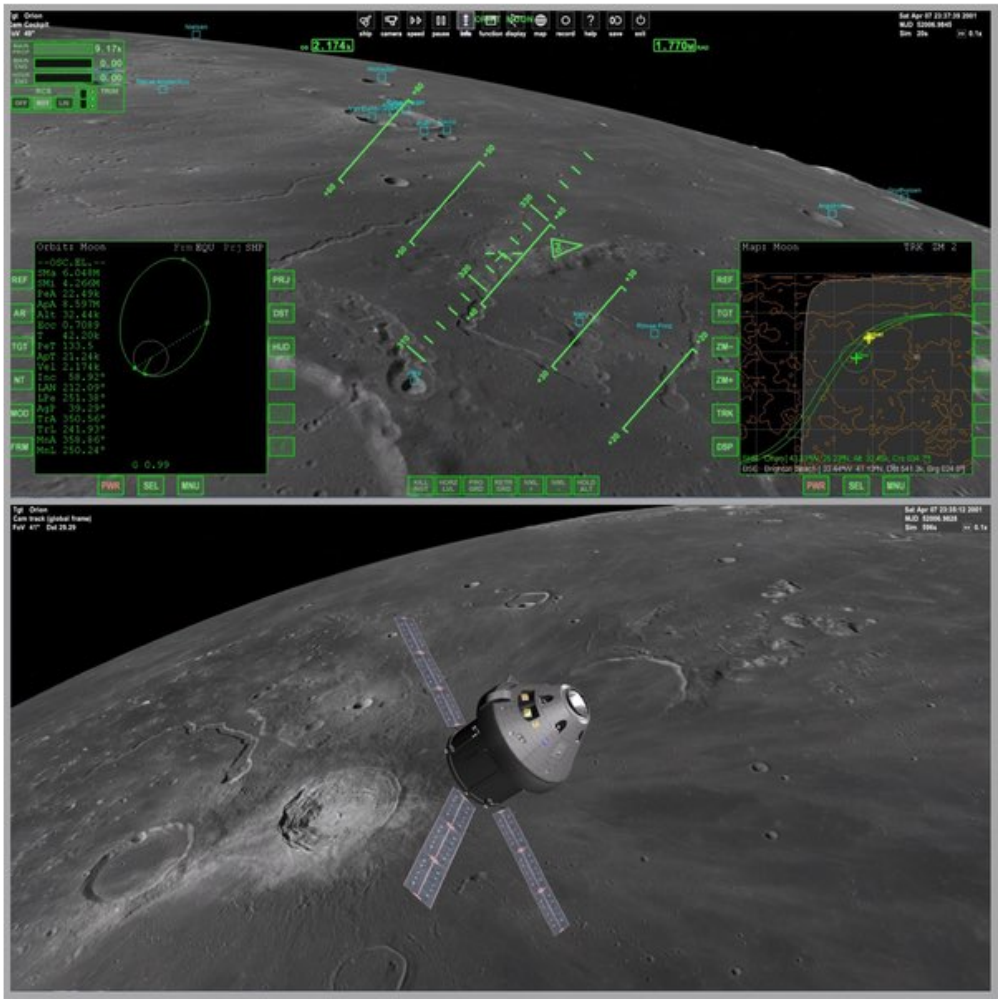

Figure 2.2: Orbiter 2016, source: [19]

This program uses a different approach than the Kerbal Space Program. First of all, it focuses on the realism of graphics. Solar system objects as well as all vehicles and their interiors are modelled with great attention to detail. Believable spacecraft behaviour and Newtonian physics have also been implemented.

Orbiter simulator does not contain a virtual machine similar to our Orbital Flight Simulator, but it does contain an advanced scripting solution - a Lua language interpreter [32]. This scripting engine allows users to control a variety of simulation tasks including autopilots, mission control, physics simulation interaction, e.t.c. It is a modern solution - rather at the level of the application itself - a general-purpose scripting language that allows user to automate various processes (including steering of the spacecraft) using extensions and commands added to the Lua language. For details on Orbiter features, see [19].

OFS by implementing its own Virtual Machine uses a different approach. It try to simulate behaviour, functions and constrains of real on-board computers, especially in the earlier years of the space exploration era - the already mentioned Apollo Guidance Computer. The author believes this approach has interesting implications - especially for educational applications and hobbyists interested in low-level computer architecture, science, physics, and space exploration in general.

2.3 Overview of other virtual machines

The purpose of virtual machines is to define abstraction layers between real hardware and executing software. There are many uses for virtual machines: from emulating outdated hardware on modern computers (for example, an old game console) to allowing software portability (Java Virtual Machine).

The general VM architecture developed in this paper was inspired by the machine described in [5], but differs from it in many details:

- Different language used in the implementation: C++ instead of Python.
- Different instruction set and FPU presence: addition of floating point instructions and registers.
- No stack and no interrupts. Both can be emulated with main memory, comparisons and conditional jump instructions.
- The presence of translator from assembly language into byte-code. In [5] translator was only mentioned without details of its implementation.

A very well-known example of a general purpose virtual machine is the Java Virtual Machine [17]. Despite its complexity (it is on a completely different level than the VM developed in this paper), it has some basic similarities with our solution.

- Both machines use bytecode interpreted by a built-in interpreter (Java VM also has a just-in-time compiler, which our implementation doesn't have because it is redundant in our applications).
- Both machines contain a translation from a higher-level human-readable language into byte-code. The Java VM uses a very high level language called Java and our machine uses low assembly language.

The Java VM tries to be as efficient as possible in terms of memory usage and speed. It's basically a low-cost hardware abstraction layer that is necessary to achieve hardware independence from the programs it runs. The Java VM also includes very powerful features that make it easier to write Java programs and improve stability and resistance to programming errors. One such feature is Garbage Collector. Garbage Collector is a subsystem that automatically releases memory when it is no longer needed. Our VM does not have it - the programmer must take care of allocating and releasing memory resources himself. This is typical for low and middle level languages such as our assembly language.

Chapter 3

Motivation and the proposed solution

The analysis of existing solutions shows that it is possible to create a software that will better serve the goals set in the first chapter. In this chapter we will try to define the requirements that our solution should meet.

We need a universal tool that reliably simulates the physics of rocket flight and gives the user the most precise control possible. In addition, the solution should provide visual feedback of acceptable quality: a reliable representation of the appearance of the rocket, planet, atmosphere and outer space. Spaceflight also needs to happen in real time.

Therefore our solution should meet the following requirements:

- A low-level language that is easy to learn and gives the user the ability to write any algorithm. Existing programming languages can be used, but it is better to create our own, simple language tailored to our requirements.
- The low-level language in terms of its capabilities and functions should correspond to the low-level language that was used to program the on-board computer of the Apollo spacecraft - Apollo Guidance Computer. The choice of this model language is dictated by the fact that the AGC is basically a representative on-board computer. It is advanced enough to fully digitally and algorithmically control the spacecraft in real-time, and at the same time it is simple enough not to overwhelm with its complexity. Of course, we must bear in mind that our language model will be much simpler - at least at the prototype stage. Some basic features of the AGC language as well as the computer itself will be discussed in Chapter 4.
- As a consequence of the above requirement, some instructions in this language should allow assembly language programs to send commands to the racket.
- Commands sent to the rocket should have full control over the thrust vector (direction and magnitude). In addition, the commands must be able to change the orientation of the rocket during the flight phase above the atmosphere when the main engine is turned off (simulation of the reaction control system and/or gyroscopes).
- The rocket itself must have the physical properties of its real-world counterpart: time-varying mass and the right size and shape. The same applies to the properties of the planet and other celestial bodies in the simulation.
- A set of commands should enable programming of any flight: ballistic, suborbital and orbital. A deorbit maneuver should also be possible.

- Our software should give the user the ability to edit, save and load written programs. It should also include a source code editor. This will give us a kind of substitute for a simple integrated development environment in which the user can comfortably develop his/her orbital programs.
- A very desirable feature is the ability to stop and speed up time. Real-time simulated spaceflights can be long-lasting.
- Visual feedback of the state of the rocket is another very welcome feature. The user should know what are the current and historical flight parameters, such as: rocket mass, speed, delta-v, position, altitude, apogee and perigee of the orbit. Historical data should be presented in the form of easy-to-read charts.
- Another welcome feature is the ability to plot a future flight trajectory prediction. The user should know how the flight will proceed with the current parameters (direction and magnitude of the thrust vector, atmospheric pressure, altitude, e.t.c.)

Chapter 4

Key issues of orbital flight simulation

The main idea of the application is to simulate ballistic rocket flights. This includes orbital flights. Our application was created to simulate flights in the gravitational field of the planet. The parameters of the planet (size, mass, e.t.c.) are similar to those of Earth. The rocket is guided by a virtual on-board computer of the original design. Controlling the rocket is about writing programs in the assembly language of the virtual machine. Flight programs written in assembly language have access to virtual machine memory where telemetry data such as rocket speed, position, time, mass, etc. is stored. Based on this data, the flight program can send commands to the rocket.

4.1 Overview of orbital flight in intuitive terms

First, let's define the orbital flight problem. In this section, we'll take a more informal approach - we'll try to define it intuitively. Later in this chapter, we will introduce more formal mathematics.

As we know, a simple ballistic flight looks like this: take a stone and throw it straight into the air. It will go up to the certain point, and after reaching maximum altitude, it will start to fall until it collides with the ground. If we throw that stone at some angle, the flight will be in parabolic shape. This shape is the resultant of the gravitational force that pulls the stone to the ground and the initial velocity of the stone given by the throwing force. We can assume that the gravitational force is constant and always directed towards the center of the planet. By changing the throwing force (its angle and magnitude) we can affect the parabolic shape of the stone's flight.

There is another very important force - the force of atmospheric drag, which is especially important when the speed of the stone is large enough. The magnitude of this force is directly proportional to the velocity of the stone and inversely proportional to the altitude above the ground. At a certain height, this force disappears completely.

Now imagine that we throw this stone with a very large initial force. It is possible that this stone will leave the atmosphere and continue its ballistic flight. If the angle and speed are right, the stone will never fall to the ground because the planet is round and the gravitational force that bends the stone's trajectory is always directed towards the center of the planet. Suppose there is no atmospheric drag because the flight is above the atmosphere. If the above conditions are met, such a flight is called an orbital flight. As we can see, orbital flight is a special case of ballistic (or suborbital) flight. Typically, the orbit is ellipse-shaped. There are two particular points of such an ellipse: the apogee - where the altitude is highest, and the perigee - where the altitude is lowest. The orbit is stable if perigee is above the upper atmosphere.

Of course, under real conditions, the so-called "low Earth orbit" is not stable - there is still some atmospheric drag. Therefore, space vehicles must periodically start their engines to stay in orbit.

However, for the purposes of our simulation, we will assume that above a certain altitude there is no more atmospheric resistance.

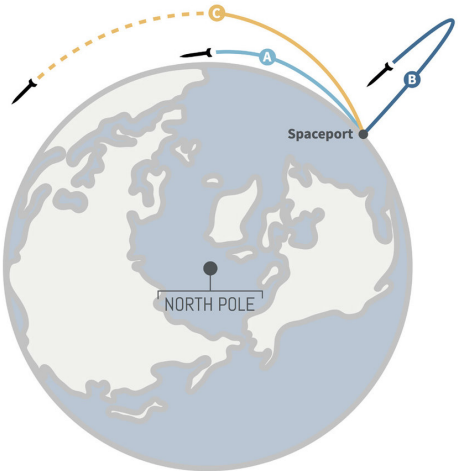

Figure 4.1: Comparison of suborbital and orbital flights, source: [15]

The figure 4.1 shows the difference between suborbital and orbital flights. In the figure, flight B corresponds to parabolic flight, flight A to suborbital (but still parabolic), and finally flight C to regular orbital flight.

4.2 Overview of basic orbital parameters

In this section we need to say a few words about orbits in general. In his work, Kepler defined a number of parameters that define orbits. In addition to perigee and apogee, we can also distinguish, among others:

- Eccentricity - a quantity characterizing the shape of the orbit.
- Inclination - the angle between the orbital plane and the reference plane.
- Semi-major axis - half of the long axis of the ellipse.
- True anomaly - a term used to describe an angular parameter that determines the position of a body moving in a orbit.

In further considerations in this work, we will mainly use the perigee and apogee of the orbit. A more detailed discussion of the orbital parameters can be found in [4]. A more general description of determining the orbits of celestial bodies by observation can be found in [3]

Another schematic figure 4.2 shows the key points of the orbit that will be of interest to us.

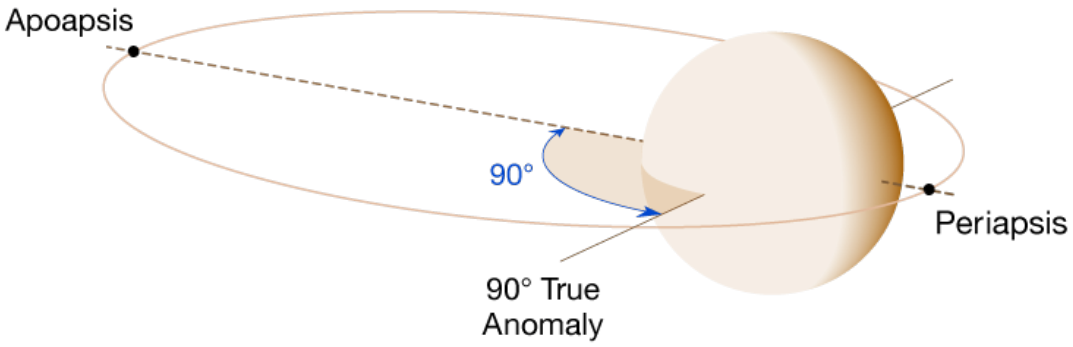

Figure 4.2: Key points of the orbit, source: [4]

In the figure 4.2 apoapsis refers to the apogee in our case, similarly periapsis is the perigee in the case of orbits around the earth. The concept of true anomaly was briefly described above in the text.

The reasoning described at the begging of the previous section is similar to the thought experiment called *Newton's Cannonball*, where throwing a stone has been replaced by firing a cannonball with sufficient force, and the atmospheric drag force has been neglected. Newton's original engraving is shown in 4.3

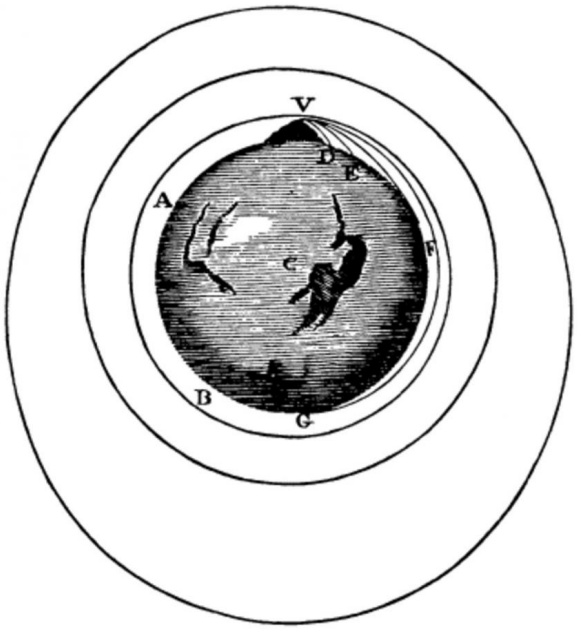

Figure 4.3: Newton's cannonball, (original engraving) source: [16]

In the figure 4.3 the letter V denotes the cannonball launch point, the letter A denotes the surface of the planet, the letters F and G are variants of ballistic flight, and the letter B is a stable orbital flight.

4.3 Newtonian Physics in relation to rigid body moving in gravity field

In the previous sections we made an informal introduction to orbital flight, now we will introduce some formalisms. Our stone becomes the rocket, and the throwing force becomes the thrust of the rocket's motor (which is also a force).

To calculate the trajectory of the rocket, we will use Newton's second law of motion:

- *In an inertial frame of reference, if the forces acting on a body are unbalanced (i.e. resultant force $\vec{F}_{\text{res}}$ is not zero), then the body moves with an acceleration directly proportional to resultant force and inversely proportional to body mass.*

We can write the above law as formula (1.1)

$$\vec{a} = \frac{1}{m}\vec{F}_{\text{res}} = \frac{\vec{F}_{\text{res}}}{m}. \tag{1.1}$$

where $\vec{a}$ velocity and $\vec{F}_{\text{res}}$ resultant force are three-dimensional vectors, and m mass is a scalar value.

In our case, the resultant force $\vec{F}_{\text{res}}$ is the sum of several forces, which we can write as formula (1.2).

$$\vec{F}_{\text{res}} = \sum_{i=1}^{n} \vec{F}_i, \quad n = 2 \tag{1.2}$$

The first force $\vec{F}_1$ is the thrust of the variable-geometry rocket engine. The final $\vec{F}_2$, second force is what we call *dynamic atmospheric pressure*, which requires further explanation.

The dynamic atmospheric force is a function of the rocket's current velocity and the current altitude above the planet's surface. This force can be determined in equation (1.3).

$$\vec{F}_{3(t)} = aA_{(t)}b\vec{v}_{(t)}, \quad a, b \in \mathbb{R} \tag{1.3}$$

Where $A_{(t)}$ is a scalar variable representing the altitude at time t. $\vec{v}_{(t)}$ is the velocity vector at t and a, b are arbitrary scalar constants.

For the purposes of our simulation, we will assume that the magnitude of the gravitational force vector is constant. The mass of the rocket is negligibly small compared to the mass of the planet, and our orbits are not too far from the planet's surface. For this reason, the gravitational term of the acceleration equation should not be divided by the mass of the rocket. Our equation (1.1) takes the form presented in (1.4).

$$\vec{a} = \vec{F}_g + \frac{\vec{F}_{\text{res}}}{m}. \tag{1.4}$$

where $\vec{F}_g$ is our gravitational force vector pointing towards the center of the planet.

Knowing the acceleration vector, we can calculate the velocity. From now on, we will treat all quantities as instantaneous ($\Delta t \to 0$). So the velocity and position equations of the rocket take the form (1.5) and (1.6), respectively.

$$\Delta\vec{v} = \vec{a}\Delta t, \quad \Delta\vec{v} = \vec{v}_{t(i+1)} - \vec{v}_{t(i)}, \quad i >= 0, i \in \mathbb{N} \tag{1.5}$$

$$\Delta\vec{r} = \Delta\vec{v}\Delta t, \quad \Delta\vec{r} = \vec{r}_{t(i+1)} - \vec{r}_{t(i)}, \quad i >= 0, i \in \mathbb{N} \tag{1.6}$$

where $\Delta\vec{r}$ is the rocket's position change. So finally our position change equation takes the form (1.7).

$$\Delta\vec{r} = \left(\vec{F}_g + \frac{\vec{F}_{\text{res}}}{m}\right)\Delta t^2, \quad \Delta \to 0 \tag{1.7}$$

And the iterative equation of motion takes the form (1.8).

$$\begin{cases} r_{t(0)} = r_0 \\ r_{t(i+i)} = r_{t(i)} + \Delta\vec{r}, \quad i >= 0, i \in \mathbb{N} \end{cases} \tag{1.8}$$

We must remember that $\vec{F}_{\text{res}}$ in (1.7) is actually a function and must be computed for each time step. In the following chapters we will look at the numerical solution of this equation. We introduce a simple physics engine, an algorithmic numerical solver that is part of our simulation engine.

The reasoning in this section is based on descriptions of the laws of motion and Newtonian mechanics found in [1], especially in Chapters 4 and 5.

4.4 The problem of flight simulation in the context of the flight of a simple single-stage rocket

As it turns out, equation (1.4) is sufficient to reliably simulate a ballistic (including orbital) flight of a rocket relatively near a planet surface. Our rocket is simple rigid body with thrust vector mounted at the one of it's ends. Rotating the rocket changes the direction of the thrust vector. A rocket burns fuel, so its mass decreases over time.

The main purpose of our application is to allow the user to control the rocket using programs written by himself. For this to be possible, we must have four foundations of our application:

- Ballistic flight physics simulation.
- 3D graphics visualisation.
- A programming language and model for a virtual machine that interprets and executes user-written programs.
- Some kind of I/O interface that allows programs written in assembly language to interact with the racket: sending commands and responding to telemetry data.

We have briefly outlined the mathematical foundations of physics simulation. It's time to say a few words about the on-board computer - the virtual machine and its language.

4.5 On-board computer based on Apollo Guidance Computer

Flying a rocket is a complicated task. The problem becomes even more complex when we need to achieve an orbit with exact parameters. Manual control becomes cumbersome and inefficient, and sometimes even impossible. For this reason, spacecraft are usually directly controlled by machines.

One such machine was the Apollo Guidance Computer - [13] the on-board computer of the Apollo spacecraft. It was an extremely advanced machine and far ahead of its time. Suffice to say, it had a cooperative multitasking operating system. It could run up to 8 programs simultaneously. In addition, it supported two programming languages: a low-level assembly-like language for most time-critical tasks, and a high-level interpreted language for more tedious use cases. All of this in just 2048 16-bit

words of random access memory [14]. The Apollo Guidance Computer (AGC) controlled almost every aspect of flight: including orbital maneuvers, the moon landing, e.t.c.

The on-board computer concept developed in this paper is largely based on the AGC. For obvious reasons, the technical execution and most of the functions differ from the original. In the next chapter, we will discuss the detailed implementation of our on-board computer.

Chapter 5

On-board computer model - Virtual Machine

The on-board computer is implemented as a sub-module of our simulation application. This is called a "Virtual Machine" with its own virtual memory, registers and instruction set. In addition, the VM has a byte-code interpreter and a translator from assembler to byte-code. This is all written in C++ and runs asynchronously in a separate thread. The application includes a simple assembly language code editor and offers the ability to load and save programs.

The figure 5.1 shows a general diagram of the Virtual Machine. In the following sections, we will discuss in detail the objects shown in this figure. That is: assembly code, translator, byte-code, interpreter, registers and main memory.

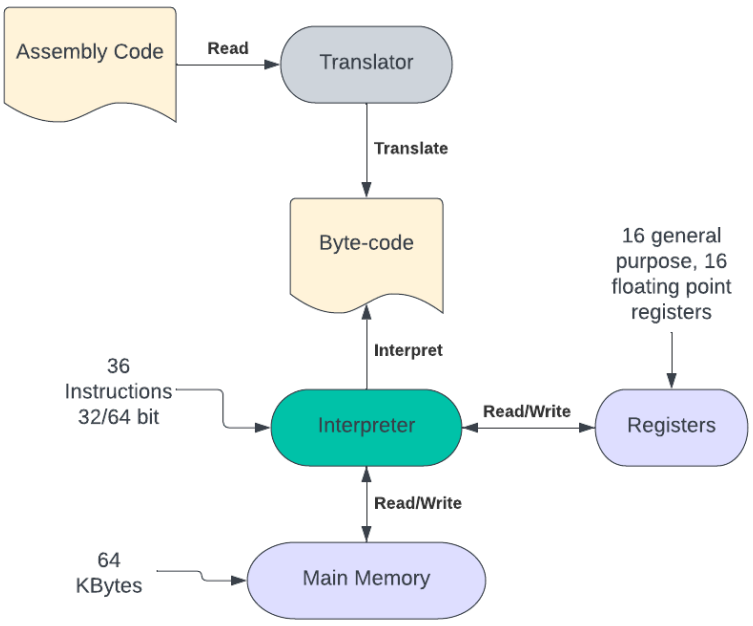

Figure 5.1: Virtual Machine Schema, source: self-elaboration

5.1 Main memory organisation

The virtual machine contains 64 kilobytes of random access memory. Memory has no specific structure: it is represented as single-dimensional array of bytes. The interpretation of a series of bytes in memory depends on the context of the instruction. For example: it may be a floating point number, an integer number, a string, e.t.c. The memory class written in C++ contains several methods that enables to store and fetch data. Listing (5.1) shows methods for storing and retrieving double-precision floating-point numbers. Each number of this kind is 8 bytes in size.

Listing 5.1: Fetch and store dword into memory

```
double Memory::fetchDWord(unsigned int addr) {
        assertConditions(addr + 8);
        unsigned char result[8] = { };
        memcopy(mem, result, addr, 0, 8);
        double val = *reinterpret_cast<double*>(result);
        leaseSemaphore();
        return val;
}

void Memory::storeDWord(unsigned int addr, double dword) {
        assertConditions(addr + 8);
        unsigned char* result
          = static_cast<unsigned char*>(static_cast<void*>(&dword));
        memcopy(result, mem, 0, addr, 8);
        leaseSemaphore();
}
```

The universal function *memcopy* copies a series of bytes between the requested locations in memory. Both methods use the *assertConditions* and *leaseSematore* functions, which provide multi-threaded security in concurrent resource access situations.

The example in Listing 5.2 shows a VM instruction that uses the *fetchDWord* function to fetch a number from memory and store it in a floating point register. We will talk about the instructions in the later sections of this paper.

Listing 5.2: Method that load from the memory into the register

```
void Instructions::fld(unsigned char* args) {
        unsigned char r_src_addr = args[1];
        unsigned char r_dst_addr = args[0];
        unsigned int src_addr = registers[r_src_addr];
        double value = memory.fetchDWord(src_addr);
        registers.fl(r_dst_addr, value);
}
```

There are regions of memory that have special interpretation and treatment. The figure 5.2 shows the main memory organization.

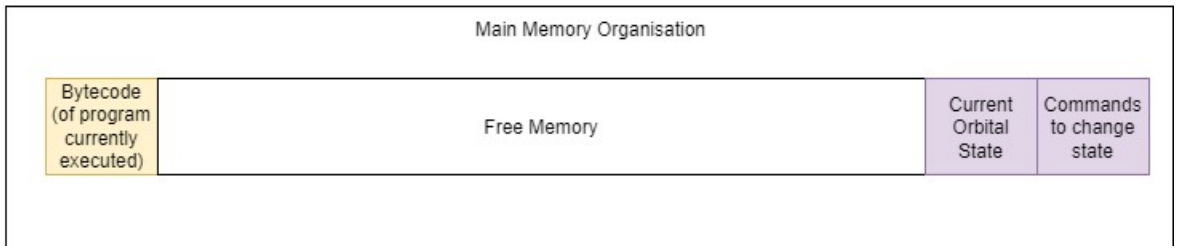

Figure 5.2: Map of the memory, source: self-elaboration

The initial part of the memory is used to store the translated byte-code of the currently executing program. Usually it is from a few to several thousand bytes depending on the complexity of the program. Byte-code is represented as binary stream of bytes and it is very compact. After byte-code there is the "free space" where programs can store their data, e.t.c.

Counting from the end, a few dozen bytes are used to store rocket flight telemetry information. This memory section is structured as follows:

```
Rotation:
        z       65528
        y       65520
        x       65512
Velocity:
        z       65504
        y       65496
        x       65488
Position:
        z       65480
        y       65472
        x       65464

        mass              65456
        thrust magnitude  65448
        altitude          65440
        timestamp         65432
```

The last few bytes are reserved to store temporary command data sent to the rocket.

5.2 Registers organisation

The virtual machine has 16 floating point registers and 16 general purpose registers. Floating point registers can store 64 bit double precision floating point numbers. The general purpose registers are 32 bits. So we can say that the VM has a hybrid 32/64 bit architecture and has a built-in FPU (Floating Point Unit).

In addition, there are three special-purpose registers:

- Zero Flag Register (ZF) - is used in compare and jump instructions. When the two compared values are equal, it takes the value 1.
- Carry Flag Register (CF) - is used in comparisons and jump instructions. When the first of the compared values is greater than the second, the register is set to 1.
- Program Counter (PC) - used to store the address of the currently executed instruction.

All arithmetic operations, comparisons and jumps are performed only on registers. Special VM instructions are needed to move data from memory to registers and vice versa.

5.3 Instructions set

We can divide instructions set into these seven categories:

- Data copy operations.
- Saving and loading data to and from memory.
- Arithmetic operations.
- Logical operations.

- Comparisons and conditional jumps.
- Unconditional jumps.
- Special instructions.

We will discuss each of those groups separately. Table 5.1 lists each supported operations. Instructions with the letter "h" at the beginning operate on floating-point registers. the target register is always specified first.

Table 5.1: List of instructions

Instruction name	Description	Arguments	Arguments size
Data copy operations			
mov, fmov	Moves data between two registers	ids of destination and source registers	2 x 8 bits
set, fset	Store value in the register	id of register, 32 or 64 bit value	1 x 8 bits + 32 or 64 bits
Load and store data from and to memory			
ld, fld, bld	Load value from the memory and store it into register	id of the destination register, id of the register in which address of value in memory is stored	2 x 8 bits
st, fst, bst	Store value from the register into memory	id of the register in which address of the destination in memory is stored, id of the register storing source value	2 x 8 bits
Arithmetical operations			
add, fadd, sub, fsub, mul, fmul, div, fdiv, mod	Add, subtract, multiply, divide and modulo two values and store result into the destination register	ids of the source and destination registers	2 x 8 bits
Logical operations			
vor, vand, vxor	Logical or, and, xor on two values and store result into the destination register	ids of the source and destination registers	2 x 8 bits
vnot	Logical not of single value	id of register which value must be negated	1 x 8 bits
vshl, vshr	Shift bits in left or right direction	id of destination register, id of register in which number of bits to shift are stored	2 x 8 bits

Comparisons and conditional jumps			
cmp, fcmp	Compare values of two registers. Sets ZF register to 1 if values are equal, and CF register if second register is greater than first	ids of registers to compare	2 x 8 bits
jz, jnz	Jump if ZF flag is set to 1 (jz) or is set to 0 (jnz)	id of register with address to jump	1 x 8 bits
jc, jnc	Jump if CF flag is set to 1 (jv) or is set to 0 (jnv)	id of register with address to jump	1 x 8 bits
jbe, ja	Jump if both ZF and CF flags are set to 1 (jbe) or CF is set to 1 and ZF is set to 0 (ja)	id of register with address to jump	1 x 8 bits
Unconditional jumps			
jmp	Unconditional jump	Address to jump	32 bits
jmpr	Unconditional jump	id of register in which address to jump is stored	1 x 8 bits
Special instructions			
cmd	Send command to the rocket	id of register in which command code is stored, id of floating point register in which value of command is stored	2 x 8 bits
halt	Stops Virtual Machine		

5.4 Assembly language

The assembly language used by our on-board computer is a low-level language, similar in some respects to the Z80 or C64 assembly language described in [25] and [26]. Main difference is that our language supports floating point numbers and registers. The number of registers in our implementation is also much larger. The assembly syntax is very simple. Each non-blank row has the following structure:

```
(db str)|label: |(instr (label|(idreg1 [(,)idReg2|value])))

where
   a | b  means alternative
    (a)   means requirement
    [a]   means optionality

   db     defines string of characters
   instr  is instruction name
```

```
label  is any string of characters (excluding white spaces)
value  is any number
idReg1 is first register identifier
idReg1 is second register identifier
```

As it turns out, a simple language and a set of instructions listed above is enough to code any algorithm. This means it is Turing-Complete. Comparison statements and conditional jumps are enough to implement branches and loops of any kind.

For example, the program in listing 5.3 copies the string "Hello world" from program code memory to another address in VM memory.

Listing 5.3: Hello World assembly program

```
; store address of data in register (Hello world literal)
set r4, data

; set up registers for memory addresses
vxor r0, r0
set r1, 1
set r3, 256

print_loop:
  ; fetch byte from addres stored in r4
  bld r2, r4

  ; if zero, exit the loop
  cmp r2, r0
jz .end

; otherwise store byte in desired address in memory
bst r3, r2
add r3, r1

; move r4 on another character and go back
; to the beginning of the loop
add r4, r1
jmp print_loop

.end:
  halt
data:
  db Hello world
```

Lines starting with ";" are ignored and act as comments. Comments in Listing 5.3 describe the function of each block of code. "db" is not a instruction. This is a special directive for the translator, which means that the following string must be treated as a compact piece of memory and stored right after the translated program byte-code.

5.5 Translator to byte-code

Virtual Machine does not run assembly language code directly. Regardless of how cryptic the reading may seem, assembly language was designed to be used by a human, not a virtual machine interpreter. There are a two main reasons for this:

- Assembly language written as text is still very verbose compared to other methods of storing computer programs in memory. This is all the more important since we only have 64 kilobytes of main memory.

- The interpreter can become unnecessarily complicated if it has to translate and decode every single line of code individually.

We need to translate our assembly code into something more convenient for execution by a relatively simple interpreter. Therefore, our virtual machine has a supporting subsystem that is able to generate a stream of binary data corresponding to our assembly source code. This binary data is called byte-code.

The bytecode has a relatively simple structure: each line is translated one-to-one into the following stream of bytes:

```
opCode+argumentBytes

where
        opCode          is a numerical identifier
                        of the instruction

        argumentBytes   is a string of bytes representing
                        the arguments of the instruction

        +               is an empty space - there is no
                        byte separation between
                        nstrCode and argumentBytes
```

Each pair like above is stored next to other. It produces very compact stream: considering that each *opCode* is exactly one byte, length of *argumentBytes* is variable between 1 and 8 bytes. The interpreter knows the length of each *opCode's argumentBytes* because it uses a built-in table of instructions similar to the list we provided in the section above.

Listing 5.4 shows an example translator method. It translates an instruction class that takes a register identifier and a floating point number as arguments. Such an instruction could be, for example, fset: (fset f9, -1.27)

Listing 5.4: Translate instruction

```
void Translator::trnsl_fconstant_to_register(
std::tuple<unsigned int, unsigned int> instr, std::string line) {
  // decode opcode and arguments size
  line = trim(line);
  unsigned int opcode = std::get<0>(instr);
  unsigned int instrSize = std::get<1>(instr) + 1;

  // decode register number:
  int pos = line.find(" ") + 2;
  int pose = line.find(",");
  std::string regNumber = line.substr(pos, pose - pos);
  unsigned char reg = stoi(regNumber);

  // decode floating point constant number:
  pos = line.find(",") + 2;
  pose = line.size();
  std::string constant = line.substr(pos, pose - pos);
  double cnst = 0;
  if (std::isdigit(constant[0]) || constant[0] == '-') {
    cnst = stod(constant);
  } else {
    // label pointing to address
    cnst = labelDict[constant];
  }

  // calculate current instruction addres,
  // and copy opcode, registry and decoded number
```

```
  // into memory
  unsigned int addr = instr_addr - instrSize;
  code[addr++] = opcode;
  code[addr++] = reg;

  unsigned char* word = static_cast<unsigned char*>
  (static_cast<void*>(&cnst));

  Memory::memcopy(word, code, 0, addr, 8);
}
```

The comments in the 5.4 listing describe the functions of each section of code. The sample method in 5.4 is one of several types of decoding methods, all other types are listed below:

- trnsl_constant_to_register similar to trnsl_fconstant_to_register above but refers to integer numbers
- trnsl_register_to_register - translate instructions that operates on two registers
- trnsl_constant - translate instructions that operates only on constants or labels
- trnsl_register - translate instructions that operates only on one register
- trnsl_halt - translate halt instruction

5.6 Byte-code interpreter

The byte-code interpreter is a simple subsystem of the virtual machine, and on the other hand one of the most important. It contains the main execution loop. It sequentially fetches instructions one by one, decodes them, and executes them. The special register PC (Program Counter) indicates the memory address of the next instruction to be executed. PC is incremented on each iteration of the loop if there was no jump. Jump instructions can modify the PC register.

Thanks to the transfer of part of the work to the language translator, the interpreter may be simple and fast. Execution loop method is shown in the listing 5.5

Listing 5.5: Execution loop

```
void VMachine::executionLoop() {
  // fetching first opcode:
  threadFinished = 0;
  unsigned int opcode = memory->fetchByte(0);
  while (opcode != opcodes->getOpcode("halt") && !shouldStop) {
    while (pause); // wait

    // decode and execute instruction:
    unsigned int args_size = opcodes->getInstrSize(opcode);
    unsigned char* args = new unsigned char[args_size];
    Memory::memcopy(memory->mem, args, pc + 1, 0, args_size);
    instructions->call(opcode, args);
    delete[] args;

    pc = registers->pc();
    if (oldpc == pc) {
        // there was no jump - update program counter:
        pc += args_size + 1;
        registers->pc(pc);
    }
    oldpc = pc;

    // fetch another opcode
```

```
            opcode = memory->fetchByte(pc);
        }

        threadFinished = 1;
    }
```

Comments in the code in 5.5 describe the behaviour of the method. There is another interesting observation: conditional and unconditional jump instructions can change the program counter (PC) register. In this case, the execution loop method cannot increment PC - this is reflected in the "if" statement in the method.

5.7 Input - output and ODDMA subsystem

Any computer is useless if it doesn't have some kind of I/O subsystem. In our case, the I/O subsystem is closely related to a specific use case: rocket flight control. To better understand this, we will use the figure 5.3

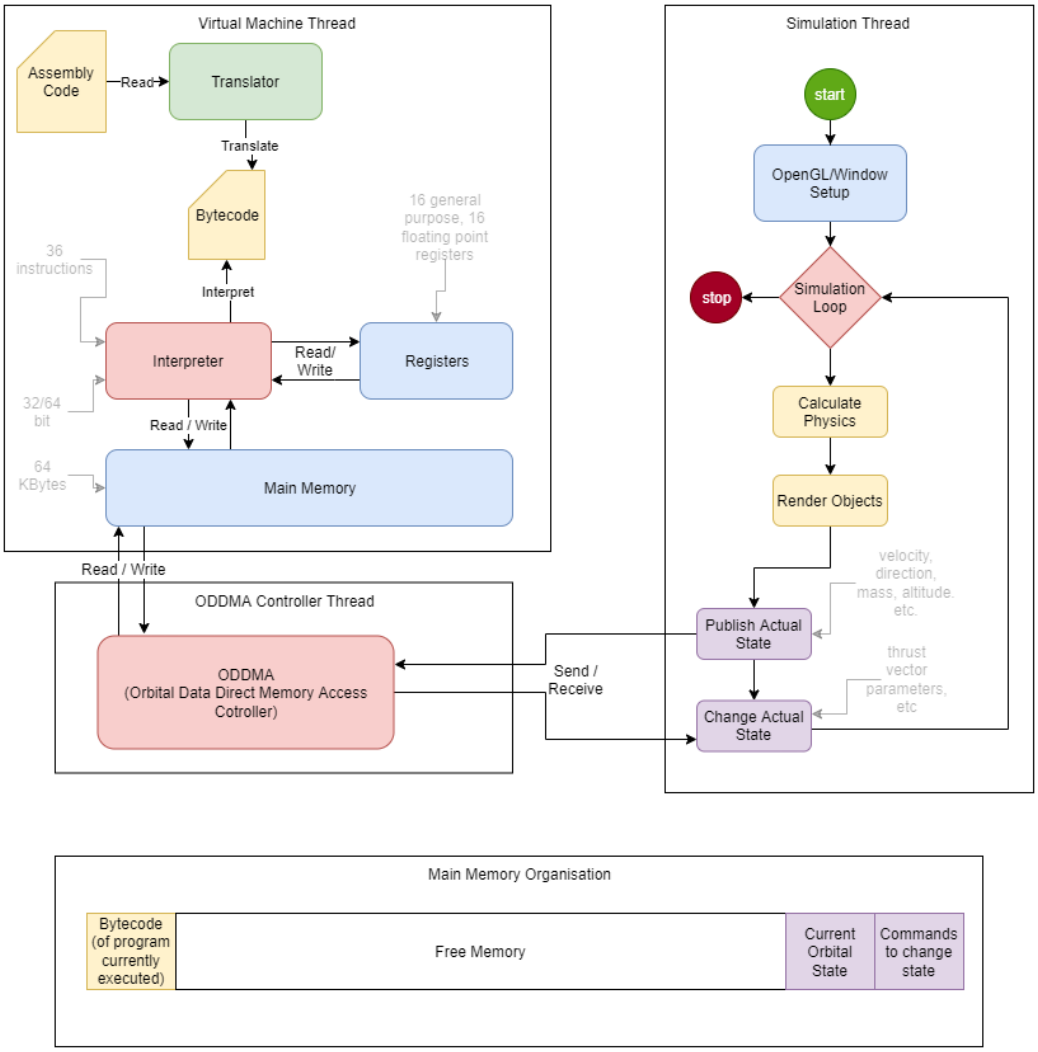

Figure 5.3: Solution schema, source: self-elaboration

As we can see in 5.3, the virtual machine and the main simulation loop run in separate threads. To some extent, this fact alone represents the real world: a separate unit that is supposed to communicate with the rocket is the on-board computer. To make this possible, we need a separate thread that handles the communication between the rocket and the virtual machine. Technically, this is implemented as three separate threads wrapped in one subsystem, which we call ODDMA - Orbital Data Direct Memory Access controller. At its heart, ODDMA has two queues: *rocket status Queue* and *commandBus*, which is buffered and thread safe. In addition to this, ODDMA includes the three previously mentioned threads:

- State Producer - it reads the current state of the rocket, wraps it in a compact data structure and sends it to the rocket *rocket status queue*.
- State Consumer - reads data from the *rocket status queue*, deserializes it, and stores it in memory as discussed in the section on memory structure.
- Command Listener - reads the command from *commandBus* and sends it to Rocket in a form understandable to it.

The *cmd* VM instruction can write directly to *commanBus*. Main memory as well as both queues are thread-safe, which means that simple semaphores are implemented to prevent complications when accessing shared resources concurrently.

Let's take the code snippet 5.6 as an example. This code implements the *Command Listiner* loop.

Listing 5.6: Command listener

```
void ODDMA::commandListener() {
    while (threadsStarted) {
        if (commandBus->anyCommands()) {
            RocketCommand rocketCommand
                = commandBus->getCommad(runningTime);
            executeInstruction(rocketCommand.code(),
                rocketCommand.value());
        }

        takeANap();
    }
}

void ODDMA::takeANap(){
    std::this_thread::sleep_for(
        std::chrono::milliseconds(4));
}
```

The *takeANap* method simulates the inertia of the system and its non-instantaneous response time. The *while* loop continuously checks if there is a command on the command bus, and if so, it delegates its execution.

Chapter 6

Rendering a 3D world

This chapter deals mainly with the visual aspects of the simulation. We need to place our simulation in a believable space imitating real conditions. Therefore, our simulation world is rendered as a set of properly lit 3D objects. The scale of the world is roughly equal to the scale of our solar system limited by the distance of the earth from the sun. The only moving part of our 3D world is the rocket, if we don't include the rocket engine smoke path, trajectory prediction paths, and camera (actually: there is no concept of camera in our simulation - more on that later). In the following sections, we will discuss in detail how the 3D world of our simulation is built.

The application works in three main modes. These modes determine what objects are rendered on the screen. In this chapter, we will describe these modes only briefly. A more detailed description can be found in the chapter on the user interface.

- **Simulation Mode** - default application mode. In this mode, the user can load, edit, store and run programs. The physics simulation is running and the internal clock is running. The rocket is displayed and the camera rotates around the rocket.
- **Trajectory Prediction Mode** - rocket is not displayed in this mode. Instead, the application displays the history of the rocket's trajectory and its predictions for the future.
- **Demo Mode** - in this mode, the application shows a demonstration flight towards the moon. The flight ends when the camera reaches the moon.

In all these modes, the application displays the earth and the sun. In the last mode, the application also displays the Moon.

6.1 Technologies used to render a 3D representation of the world

The application uses the OpenGL API to visualize 3D graphics. OpenGL has a very long history [12]. It was originally developed as a specification by Silicon Graphics, Inc. (SGI). Since 2006, OpenGL has been managed by the Khronos Group consortium as a free and open source specification. OpenGL has been widely used in the fields of computer-aided design (CAD), virtual reality, scientific visualization, information visualization, flight simulation, and video games. Examples of applications using OpenGL include: Autodesk AutoCAD, 3D Studio MAX, Blender, Google Earth, Stellarium, SciLab, Universe Sandbox.

Technically, OpenGL is not an API: rather, it is an API specification that describes how software and hardware work together to achieve desired results in terms of computer graphics, animation, e.t.c. However, in this article, we will call OpenGL an API for simplicity. Since OpenGL is an open standard,

there are several implementations in many modern operating systems. Typically, programs that use OpenGL are written in C or C++. However, there are implementations using other languages: Java, Rust, and even JavaScript (via the WebGL standard).

OpenGL is a low-level API, which means the developer has extensive control over the most important, hidden aspects of the application. This includes programming the GPU directly through shaders, transferring data from main memory to graphics card memory, sending commands from the CPU to the GPU, e.t.c.

It may be beyond the scope of this paper to discuss programming interactive real-time 3D graphics applications, especially when using low-level tools such as OpenGL. The simulator's source code contains sections that rely heavily on concurrent programming as well as the use of mathematical concepts in linear algebra (vector spaces, quaternions, matrices, e.t.c.) and geometry (coordinate systems, translations, rotations, e.t.c.). For this reason, a detailed discussion of these issues is not the subject of this article. The author refers to literature items that discuss the topic in more detail. The basics of mathematics are covered in detail in [2]. A more academic approach to OpenGL is represented by [7]. However, we will briefly discuss more important issues specific to the application that is the subject of this paper.

There is at least one important concept that we need to understand first if we want to talk about graphics programming in OpenGL. This concept is the graphics processing pipeline and the collaboration and role sharing between CPU and GPU.

Figure 6.1 sheds light on this issue.

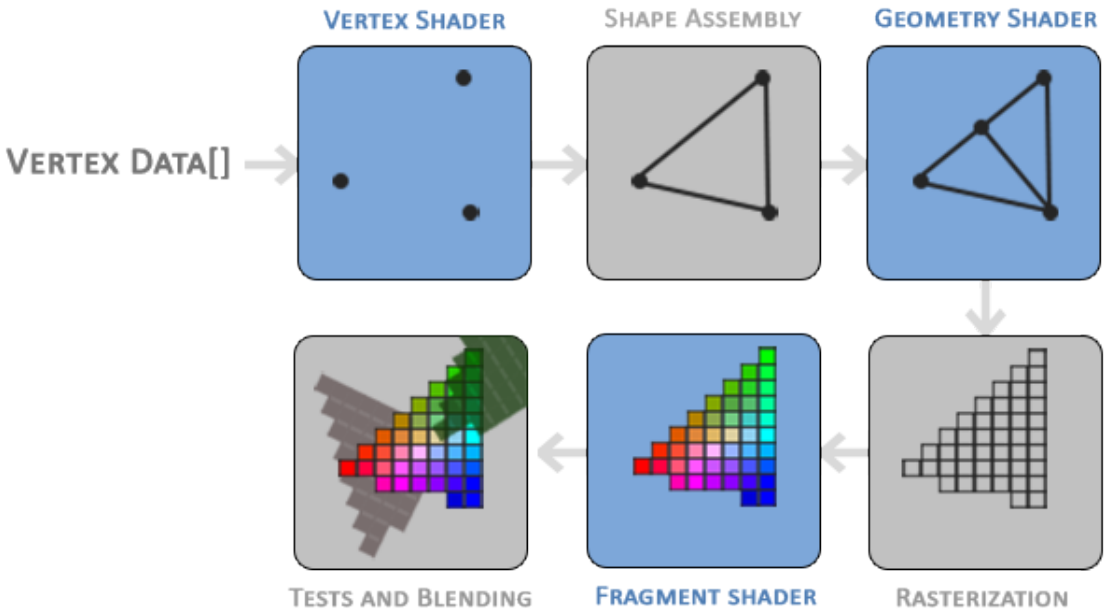

Figure 6.1: Graphics Processing Pipeline, source [8]

Typically, the CPU prepares the basic geometry of the world: it loads the 3D models into memory, converts them to internal, convenient data structures, determines their positions in the world, etc. The CPU also sends the vertex data to the graphics card's memory and sends commands to the GPU, telling it when and how these objects are to be displayed. The GPU usually does the rest of the work: with small programs called shaders: transforms the coordinates of the vertices between spaces (more on that later) *(Vertex Shader)*, creates triangles from the vertices, then triangles into more complex shapes, and performs operations on them *(Geometry Shader)*, and finally perform operations on a

single pixel or group of pixels *(Fragment Shader)*. By the way, it performs other important operations, such as rasterization, which we will not discuss here. The aspects discussed in this section are described in [7], [8] and [9].

Shaders are small programs written in a language called GLSL (OpenGL Shading Language) and run directly on the GPU hardware - more specifically, on all its stream processors (e.g. CUDA cores in NVIDIA GPUs). This means that it has its own special limits on available memory, processing power, etc. The GPU is actually a massive parallel processor with hundreds of relatively simple cores. Vertex and fragment shaders are mandatory and have no default implementations. The programmer has to program them himself if he/she wants to display something on the screen. Examples of such shaders will be shown in the following sections of this chapter. Shaders and GLSL are described in detail in [7], [8] and [9].

6.2 3D world objects

Our simulation consists of three main groups of objects:

- Celestial bodies: this includes the planet, moon and sun. Celestial bodies are modeled as spheres. The spheres are made of triangles. The program contains a simple generator code that creates a set of vertices of triangles of a sphere, taking the number of stacks and sectors of the sphere and their normal vectors. This generator code is based on [20].
- Rocket itself - 3D model loaded from *obj* file.
- Background objects such as trees and clouds - also 3D models loaded from a file.

The table 6.1 shows the sizes and relative positions of celestial bodies:

Table 6.1: Celestial bodies

Celestial body name	Radius (km)	Distance from the Earth (km)
Earth	6371	0
Moon	1737	384400
Sun	1392700	149600000

In the next section, we'll briefly discuss the coordinate systems and how world size affects the rendering process.

6.3 Coordinate system and solution to the world's size problem

In OpenGL, each vertex (x, y, z) that will be displayed on the screen must be in *Normalized Device Coordinates* (NDC), which means that each x, y, z coordinate must be within range from -1.0 to 1.0.

Therefore, the vertices of each object must be transformed to NDC before they become pixels on the computer screen. As stated in [8] it usually takes several steps - the vertices are transformed step by step to intermediate coordinate systems (so-called *spaces*):

- Local space (Object space)
- World space
- View space (Eye space)

- Clip space
- Screen space

We will not discuss coordinate systems in detail. There is an excellent book [8] that covers this in the *Coordinate System* chapter. In this paper, we will only mention that the transformation from one space to another is carried out by multiplying the coordinates by special matrices. The most important of which are: model, view and projection matrix. The figure 6.2 shows the transformation process to successive spaces.

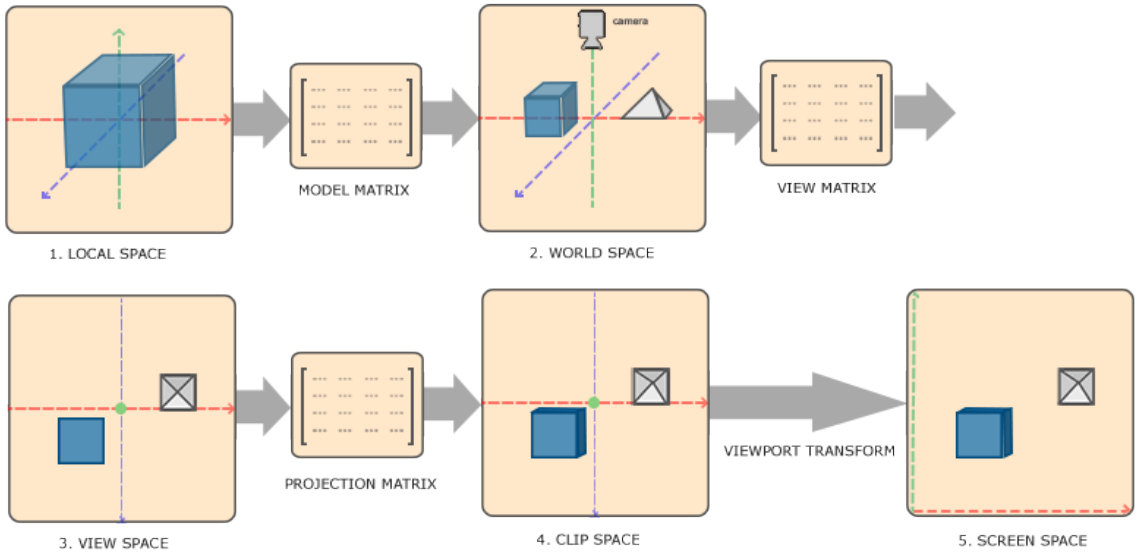

Figure 6.2: Coordinate transformations, source: [8]

From our point of view, the transformation from local space to world space is particularly important. In this step, our object's local spatial coordinates are placed in the global world-size coordinate system relative to some global origin. As we know, the size of our world is very large, and the sizes of objects range from relatively small (rocket) to really large (planet). Therefore, very small or very large numbers may appear int the model matrix, later - in the screen space, we get smaller values - which actually fit on the screen.

At this point we have to make a small digression about the precision of floating point numbers - single precision (32 bit) floating point arithmetic. Only over 4 billion different values can be stored in 32 bits. It is easy to calculate that if we take 384,400,000 (the distance of the Earth from the Moon in meters) as the largest possible number, then the greatest possible "resolution" of such a world will be about 10 cm. With a rocket size of several meters, this means a completely unstable image and simulation. The problem is of course more serious at even greater distances.

Today's popular GPUs do not support double precision operations (except maybe professional CPUs like NVIDIA Quadro). In the normal case: in the processing pipeline, the GPU takes over the space transformations described above in this chapter.

One of the solutions to this problem is to transfer from the GPU to the CPU the matrix multiplication operations associated with transformations to successive spaces. This approach was adopted in this work. Modern CPUs have the ability to do matrix multiplication quickly using FPU SIMD operations such as AVX. And the library used for this at the software layer (instead of the hardware GPU) is GLM [11]. The code snippet in Listing 6.1 shows an example of such use.

Listing 6.1: Rendering method

```
void ObjectRenderer::renderWithRotation(
  glm::dmat4& projection,
  glm::dmat4& view,
  double size,
  glm::dvec3 position,
  glm::dvec3 rotation) {
  shader->use();
  shader->setFloat("logDepthBufFC", logDepthBufFC);

  glm::dmat4 model = glm::dmat4(1.0);
  model = glm::translate(model, position);

  // calculate rotations:

  model = glm::rotate(model,
    glm::radians(rotation.x), glm::dvec3(1.0, 0.0, 0.0));
  model = glm::rotate(model,
    glm::radians(rotation.y), glm::dvec3(0.0, 1.0, 0.0));
  model = glm::rotate(model,
    glm::radians(rotation.z), glm::dvec3(0.0, 0.0, 1.0));

  model = glm::scale(model, glm::dvec3(size));
  shader->setMat4("model", glm::mat4(model));

  // calculate transformation matrix and
  // and passing it to the shader program:

  glm::mat4 transformation
    = glm::mat4((projection * view * model));
  shader->setMat4("transformation", transformation);

   [...]
}
```

The rendering method in 6.1 takes the projection and view matrices as parameters. Creates a model matrix and performs calculations according to the position and rotation of the model. Finally, multiply all three matrices and pass them to the shader. All these operations are performed by the CPU in the main rendering thread.

6.4 Camera movement

OpenGL does not have the concept of a camera in the common sense. We can only simulate the behavior of the camera by moving all objects in the scene in the opposite direction to "we" (or rather the simulated camera). There are basically three types of camera movement in our simulation:

- Free run - in trajectory prediction mode.
- Orbiting around the rocket - in simulation mode
- Moving along a straight line - in demonstration "Flight to the Moon" mode

In our case, we'll start with the "free run" camera type and derive two other types of camera movement based on that. We need to take a closer look at the *view space* concept mentioned briefly in the previous sections. The view-space transformation matrix transforms all world-space coordinates into view-space coordinates. When we talk about view space, we mean the coordinates as seen from the camera's perspective. Again, an excellent book [8] covering it in detail in the *Camera* chapter. This book mentions that we need four vectors to define a camera:

- vector describing position of the camera - *position*

- vector describing front of the camera - *front*
- vector describing up of the camera - *up*
- vector describing the right side of the camera - *right*

There is another interesting vector - the vector describing the direction of the camera, which is simply the difference between the camera position and the vector pointing to the target.
We can summarize what we said above with figure 6.3

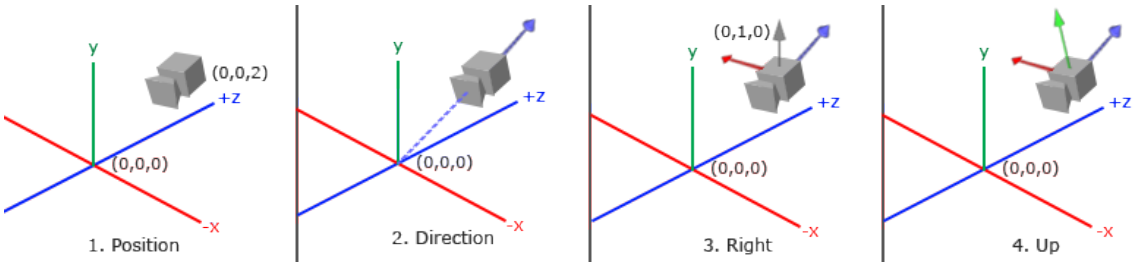

Figure 6.3: Camera axes, source: [8]

Knowing all the needed vectors, we can derive the view transformation matrix. Fortunately, we don't have to do it manually. The GLM library provides the *lookUP* function. This function takes the camera position, target position and camera up vector as parameters. In our case, the call of this function might look like the 6.2 listing.

Listing 6.2: Camera view matrix

```
glm::dmat4 Camera::getViewMatrix() {
  return glm::lookAt(position, position + front, up);
}
```

When we want to "walk around" with the camera, all we need to do is modify the *position* vector, but "looking around" is much more complicated. To look around the scene, we need to modify the vectors *front*, *up*, and *right* according to the pitch and yaw angles. Pitch and yaw are numbers (scalars) and represent: pitch - how much we look up or down, yaw - how much we look left or right. Using school trigonometry and vector algebra in (6.1) we can compute the vectors we need.

$$\begin{aligned} \vec{f_x} &= cos(\alpha)cos(\beta) \\ \vec{f_y} &= sin(\beta) \\ \vec{f_z} &= sin(\alpha)cos(\beta) \end{aligned} \tag{6.1}$$

$$\vec{r} = \vec{f} \times \vec{w}, \ \vec{u} = \vec{r} \times \vec{f}$$

where α is yaw, β is pitch, $\vec{f}$ is a front vector, $\vec{r}$ is a right vector, $\vec{u}$ is a up vector, and $\times$ is a cross product of two vectors. The corresponding code in C++ using GLM library is in the listing 6.3. The method is based on the solution described in [8].

Listing 6.3: Camera vector recalculation

```
void Camera::updateCameraVectors() {
  glm::dvec3 front;
  front.x = cos(glm::radians(Yaw)) * cos(glm::radians(Pitch));
  front.y = sin(glm::radians(Pitch));
```

```
    front.z = sin(glm::radians(Yaw)) * cos(glm::radians(Pitch));
    Front = glm::normalize(front);

    Right = glm::normalize(glm::cross(Front, WorldUp));
    Up = glm::normalize(glm::cross(Right, Front));
}
```

where *WorldUp* is a vector that points to the up direction of the world - in our case (0, 1, 0).

In our application, the *position* vector of the camera is controlled by the keyboard (wasd keys) and the "look around" (pitch and yaw angles) is controlled by mouse movement.

All this gives us the ability to freely move the camera around the scene. This camera guidance is used in trajectory prediction mode to conveniently move the camera to view the trajectory graph. In simulation mode, the position of the camera is closely related to the position of the rocket. In addition, in this mode the camera automatically 'orbits' at a certain distance around the rocket. Achieving this effect is quite simple if we have a solution to the problem of free camera movement. We need to add a function 6.4 that will be called periodically.

Listing 6.4: Update camera position

```
void Camera::updateCamPosition(glm::dvec3 newRocketPosition,
    double rotationAngle,
    float radius) {
      double camX = cos(glm::radians(rotationAngle)) * radius;
      double camZ = sin(glm::radians(rotationAngle)) * radius;

      rotationPosition = glm::dvec3(-camZ / 2.0, camX, camZ)
        + newRocketPosition;

      position = newRocketPosition;
}
```

where *newRocketPosition* is the current position of the rocket, *rotationAngle* is the camera rotation angle (increased periodically), and *radius* is the distance between the rocket and the camera. The result of *rotationPosition* is the new position of the camera.

Finally, we need to change the behavior of *getViewMatrix* as shown in listing 6.5.

Listing 6.5: Camera view matrix alternative version

```
glm::dmat4 Camera::getViewMatrix() {
    return glm::lookAt(rotationPosition
     + glm::dvec3(0.016, 0.0, 0.012), position, up);
}
```

where *position* is the position of the rocket, *rotationPosition* is the position of the camera, and *glm::dvec3(0.016, 0.0, 0.012)* is the initial distance between the camera and the rocket. And that's it - we have a camera orbiting the rocket.

The last mode of our application is a special presentation mode that takes the camera on a journey towards the moon. In this mode, we use the overloaded *updatePosition* method of the Camera class, with a fixed angle and radius set to 0.020, to change the distance between the rocket and the camera step by step. We need to calculate the direction vector from the rocket to the moon (the *toTheMoon* vector) and using it as a "guide" iterate through the distance until we meet the moon. The code in listing 6.6 solves the problem of iteratively updating the camera position.

Listing 6.6: Presentation mode camera movement

```
if (presentationMode) {
    [..]
    toTheMoon = SolarSystemConstants::moonPos - rocket->getPosition();
    camera->updatePosition(rocket->getPosition() + (toTheMoon * radius),
```

```
    rocket->getRotation());

  if (radius < 0.52) {
    step *= 1.005;
  } else {
    step /= 1.005;
  }

  radius += step;
}
```

6.5 Illumination and shaders

We use a combination of ambient and diffuse lighting in our application. Ambient lighting is a global illumination caused by the reflection of many rays of light from surfaces in the environment. This gives more or less even illumination from all sides. In more advanced applications, the global lighting system (or ambient occlusion) takes local environmental conditions into account. In our case, a simple uniform light is enough.

Diffused lighting is more interesting. Diffused lighting makes objects brighter the greater the angle ϕ between the surface of the object and the light source.

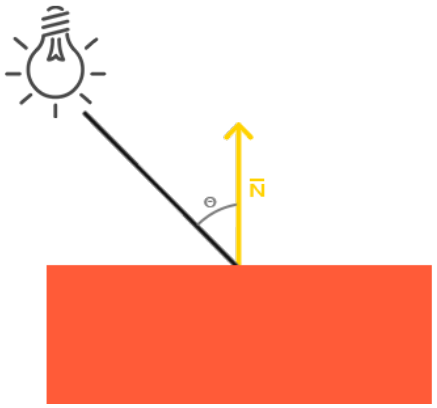

Figure 6.4: Diffuse illumination, source: [8]

As we can see in the picture 6.4 to calculate the angle ϕ we need *normal vectors* $\vec{N}$ for the surface of each object. The normal vector is a vector perpendicular to the surface. Typically, normal vectors come with 3D models. For other objects, normal vectors are computed by internal algorithms. In our case, these are only spherical objects (planets, points on trajectories, smoke "clouds"), which are generated programmatically - normal vectors are calculated for each of them. Regardless of the method, the normal vectors and vertex coordinates are transferred to the fragment shaders, where the actual illumination is computed as a mixture of ambient and diffuse illumination. An example of this shader written in GLSL is shown in listing 6.7, which is based on the solution in [8], chapter Basic Lighting.

Listing 6.7: Fragment shader - illumination, based on: [34]

```
#version 330 core
out vec4 FragColor;

in vec3 Normal;
```

```
in vec3 FragPos;

[..]

uniform vec3 lightPos;
uniform vec3 lightColor;
uniform vec3 objectColor;

void main() {
    // ambient
    float ambientStrength = 0.1;
    vec3 ambient = ambientStrength * lightColor;

    // diffuse
    vec3 norm = normalize(Normal);
    vec3 lightDir = normalize(lightPos - FragPos);
    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = diff * lightColor;

    vec3 result = (ambient + diffuse) * objectColor;
    FragColor = vec4(result, 1.0);

    [..]
}
```

As we can see, the GLSL syntax is very similar to C/C++. On the output 6.7 the fragment shader returns *FragmentColor* which is nothing but the color of the pixel. As input parameters, our shader takes the *Normal* vector and the *FragPos* vector, which is the position of the currently processed pixel.

Figure 6.5 shows that proper lighting is an important aspect of 3D visualization. This adds another level of depth to 3D scenes and sometimes makes them look quite realistic.

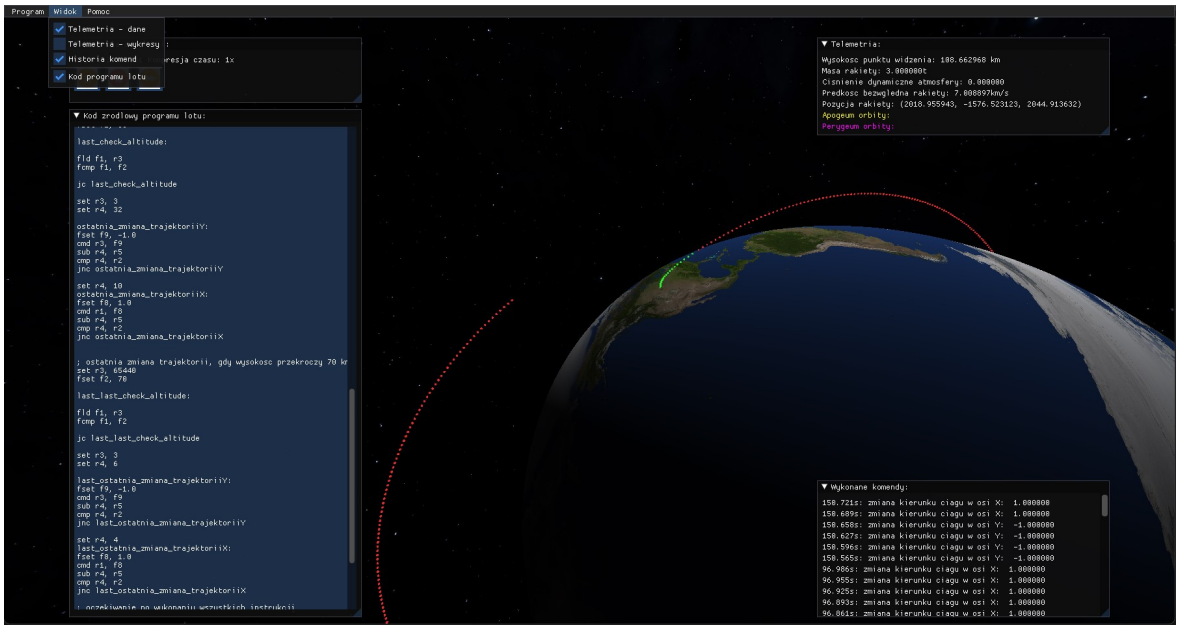

Figure 6.5: Usage of diffuse illumination, source: self-elaboration

6.6 Rocket orientation and rotation

It is important to ensure the correct starting orientation of the racket. We have to place the rocket on the surface of the planet in the chosen place. Since the surface of the planet is a sphere, we need to rotate the rocket at the right angles. To do this, we provide a method called *pointAboveTheSurface* in the listing 6.8. This method is a member of the CelestialBody class. An instance of an object of a class inheriting from CelestialBody is our planet.

Listing 6.8: Point above the surface method

```
glm::dvec3 CelestialBody::pointAboveTheSurface(double theta,
  double phi, double distance) {
  // theta - polar angle
  // phi - azimuth angle
  double r = diameter / 2.0 + distance;
  double x
    = r * cos(glm::radians(theta)) * sin(glm::radians(phi));
  double z
    = r * sin(glm::radians(theta)) * sin(glm::radians(phi));
  double y
    = r * cos(glm::radians(phi));

  glm::dvec3 result = position + glm::dvec3(x, y, z);
  return result;
}
```

This method takes the angles *theta* and *phi* as parameters. These angles correspond to the polar and azimuthal angles of the celestial body, starting from one of the planet's poles. The method then calculates the position of the point "above" the surface based on the distance (height) from the surface, the position and diameter of the planet itself.

In the second step, the method described above is used to calculate the so-called "towards" point and the initial position of the rocket. The "towards" point is a point below the surface of the planet (negative distance parameter) about 50 kilometers. This point is now used to calculate the rocket's direction of rotation as a three-dimensional vector. Then the direction vector is represented as a single *quaternion* representing a rotation in three dimensions. And in the last step, this quaternion is converted into a rotation vector expressed as the so-called Euler angles. This rather complicated chain of transformations is written in the following lines of the Listing 6.9. The conversion to a quaternion is done using the helper function *Geometry::gLookAt* with quite complicated math behind described in the literature [2].

Listing 6.9: The rocket's rotation calculation

```
  glm::dvec3 direction
    = glm::normalize(rocket.getPosition() - towards);
  glm::quat qlook
    = Geometry::gLookAt(direction, glm::dvec3(0.0,1.0,0.0));
  glm::dvec3 rotation
    = glm::eulerAngles(qlook) * 180.0f / 3.14159265f;

  thrustVector = direction * thrustMagnitude;
  [..]
  rocket.updateRotation(rotation);
```

Listing 6.9 shows a fragment of code, that calculates the *direction* vector, the described quaternion *qLook*, the *rotation* and the *thrustVector* vectors.

These transformations give us two things:

- the rotation of the rocket expressed as a vector of euler angles (x - angle along the x axis, y - e.t.c...), coupled with:

- the direction of the thrust vector.

The rotation of the rocket along any axis should entail the rotation of the thrust vector along the same axis. The rotation of a thrust vector can be done with a relatively simple function in the listing 6.10

Listing 6.10: Rotate vector method

```
glm::dvec3 Geometry::rotateVector(glm::dvec3 v,
   glm::dvec3 k, double fi) {
  fi = glm::radians(fi);
  return v * cos(fi)
    + glm::cross(k, v) * sin(fi)
    + k * glm::dot(k, v) * (1.0 - cos(fi));
}
```

where *fi* is the rotation angle, *v* is the vector to rotate, and *k* is the vector representing the rotation axis (e.g. (1.0, 0.0, 0.0) means that we rotate along the 'x' axis)

6.7 The 3D Model Loader

The 3D model is represented as a collection of meshes. Each mesh is a set of vertices. The software described in this article uses the Assimp [30] library to load *obj* files into in-memory data structures and convert them to a convenient representation of meshes and vertices. A snippet of such code displays listing 6.11.

Listing 6.11: The model loading method, based on: [35]

```
void Model::loadModel(std::string path) {
    Assimp::Importer importer;
    const aiScene* scene = importer
      .ReadFile(path, aiProcess_Triangulate
      | aiProcess_GenSmoothNormals
      | aiProcess_FlipUVs
      | aiProcess_CalcTangentSpace);

    // Error handling
    [...]

    directory = path.substr(0, path.find_last_of('/'));
    processNode(scene->mRootNode, scene);
}

void Model::processNode(aiNode* node, const aiScene* scene) {
    for (unsigned int i = 0; i < node->mNumMeshes; i++) {
        aiMesh* mesh = scene->mMeshes[node->mMeshes[i]];
        meshes.push_back(processMesh(mesh, scene));
    }

    for (unsigned int i = 0; i < node->mNumChildren; i++) {
        processNode(node->mChildren[i], scene);
    }
}
```

The method uses the *importer.ReadFile* function to load and build a data structure that represents the in-memory representation of the loaded object. The recursive helper method *processNode* creates a collection of meshes. A detailed discussion of loading 3D models can be found in [8].

6.8 Chapter summary

Three other aspects of 3D world rendering have not been covered in this chapter: texturing, sky area generation, and object instantiation. These problems are very important in the visualization subsystem and have a huge impact on the final appearance of the program, but their solutions are quite typical and can be easily found in the literature.

The visualization of our application is by far the most complex part of the solution. Although this is not the central point or the main goal of this work. Everything we said in this chapter was just scratching the surface and picking out the most important and interesting aspects of visualization. A more detailed discussion of the topics covered in this chapter can be found in the literature: [2] [7] [8]. The source code of the simulator itself can also provide extended explanations about the issues raised.

Chapter 7

Rocket flight physics implementation

In the first chapter of this work, we derived the equations of motion. In this chapter, we will discuss a simple implementation of the physics solver in our application. At this stage of implementation, the solver (or engine) supports only one rigid body - the rocket, but there are plans for further improvements - we will discuss them in the summary chapter. Our solver is simple enough to be implemented in its entirety instead of using external tools like Bullet Physics Engine e.t.c.

7.1 Numerical integration of the equation of motion

For convenience, let's recall our equations of motion from Chapter 1:

$$\vec{F}_{\text{res}} = \sum_{i=1}^{n} \vec{F}_i, \quad n = 2 \tag{1.2}$$

$$\Delta\vec{r} = \left(\vec{F}_g + \frac{\vec{F}_{\text{res}}}{m} \right) \Delta t^2, \quad \Delta \to 0 \tag{1.7}$$

$$\begin{cases} r_{t(0)} = r_0 \\ r_{t(i+i)} = r_{t(i)} + \Delta\vec{r}, \quad i >= 0, i \in \mathbb{N} \end{cases} \tag{1.8}$$

First of all, we need to find all the forces, so we'll start with Equation 1.2. In our computer program, there is a *addForce* method that adds force to an internal collection.

Listing 7.1: Add force

```
void addForce(glm::vec3 force);
```

This method is used in two methods of our solver:

Listing 7.2: Main methods in the solver

```
\begin{lstlisting}[language=C++]
unsigned __int64 calculateForces(
```

```
    unsigned __int64 timeInterval
);

void calculateAtmosphericDragForce();
```

The first of these methods *calculateForces* is used to determine all "internal" forces acting on the rocket. By *internal* we mean all forces that are independent of external physical conditions (e.g. atmospheric pressure, gravity, e.t.c.). We only have one such force: it is the thrust vector force that can be driven by the programs executed by our Virtual Machine described earlier.

In addition, this method calculates several variables such as deltaPosition and calls the *updatesPhysics* method with the calculated time interval, and finally calls the *calculateAtmosphericDragForce* method. A code snippet of the key operations that are performed in *calculateForces* is shown in the listing 7.3.

Listing 7.3: Fragment of the calucalte forces method

```
[...]
updatePhysics(MS_PER_UPDATE / 1000.0f);
timeInterval -= MS_PER_UPDATE;
deltaP = rocket.getPosition() - lastPos;
lastPos = rocket.getPosition();
[...]
calculateAtmosphereGradient();
calculateAtmosphericDragForce();
[...]
```

The variable *timeInterval* should be understood as Δt from our equations of motion. The method also calculates the change in the position of the rocket - the *deltaP* variable.

The second method mentioned above is *calculateAtmosphericDragForce*, this method determines the dynamic pressure of the atmosphere acting on the rocket during flight and must be calculated at each time step. The method will be discussed in the next section.

Finally, when we have established all the necessary forces, we can calculate the position and velocity of the rocket, i.e. solve equation 1.7 and its iterative form 1.8. The *updatePhysics* method in the 7.4 list is quoted in its entirety.

Listing 7.4: Update physics method

```
void PhysicsEngine::updatePhysics(double deltaTime) {
    glm::dvec3 velocity = rocket.getVelocity();
    glm::dvec3 position = rocket.getPosition();

    glm::dvec3 gravityForceVector
      = glm::normalize(rocket.getPosition()
      - celestialBodyCenter(celestialBodySize)) * GConst;

    glm::dvec3 sumOfForces = glm::dvec3(0.0);
    for (unsigned int i = 0; i < forces.size(); i++) {
        sumOfForces += forces[i];
    }

    velocity += (gravityForceVector
      + (sumOfForces / rocket.getMass())) * deltaTime;
    position += velocity * deltaTime;

    rocket.updatePosition(position);
    rocket.updateVelocity(velocity);

    resetForces();
}
```

With the combined power of C++'s operator overloading and GLM, our key method code looks similar to real equations and needs no further explanation. It is important to understand that the solver runs continuously in the main simulation loop and is based on the deltaTime calculated since the last time the forces were calculated.

7.2 Atmospheric pressure simulation

There is one more force we must consider:

$$\vec{F}_{3(t)} = aA_{(t)}b\vec{v}_{(t)}, \quad a, b \in \mathbb{R} \tag{1.3}$$

As we mentioned earlier, *dynamic atmospheric pressure* is a function of rocket velocity and altitude above the ground. Below is a fragment of the *calculateAtmosphericDragForce* method that solves this problem. The method shown in the 7.5 calculates the force from equation 1.4

Listing 7.5: Fragment of the method calculating the drag force

```
if (glm::length(rocket.getVelocity()) > 0.0) {
  glm::dvec3 forceDirection = glm::normalize(rocket.getVelocity()) * -1.0;
  double velocityMagnitude
    = vFactor * glm::length(rocket.getVelocity());
  double altitudeMagnitude = 1.0 / (altitude * aFactor);
  if (altitude > hLimit) {
    altitudeMagnitude *= 1.0 / altitude;
  }
  glm::dvec3 dragForce
    = forceDirection * velocityMagnitude * altitudeMagnitude;
  [..]
}
```

aFactor and *vFactor* are constants taken into equation 1.3 ($a, b \in \mathbb{R}$). By default they are 1.1 and 1.2 respectively. So as we can see, the velocity factor in the equation is a little more important than the altitude. In addition, starting from *hLimit* height, the method changes the way the drag force is calculated - from then on, the force drops off sharply. (simulates the upper atmosphere)

7.3 Trajectory prediction mode

Our application includes one more special feature that is closely related to the physics solver. We give the user the opportunity to see the future forecast of the current rocket trajectory. This is done by some king of sample-and-holding method. When a user requests a trajectory prediction, our algorithm stores the rocket's current position and velocity and removes all forces except gravity. Now, using a similar solver described in the section above, the application calculates all the future positions of the rocket. The code in listing 7.6 performs the deltaTime calculation used for each prediction step.

Listing 7.6: Demo mode delta time calculation

```
elapsedTime /= 1000;
int n = 512;
double currentTime = 4000.0 - elapsedTime;
double deltaTime = (double)currentTime / n;
```

As you can see, the prediction takes place in 512 steps and a maximum time span of up to 40,000 seconds. *elapsedTime* is the elapsed time since the simulation started. This gives the user a prediction of the free ballistic trajectory and whether orbit has already been reached or how far away it is. The

visual aspect of the trajectory prediction mode and other functions of the application will be discussed in a separate chapter.

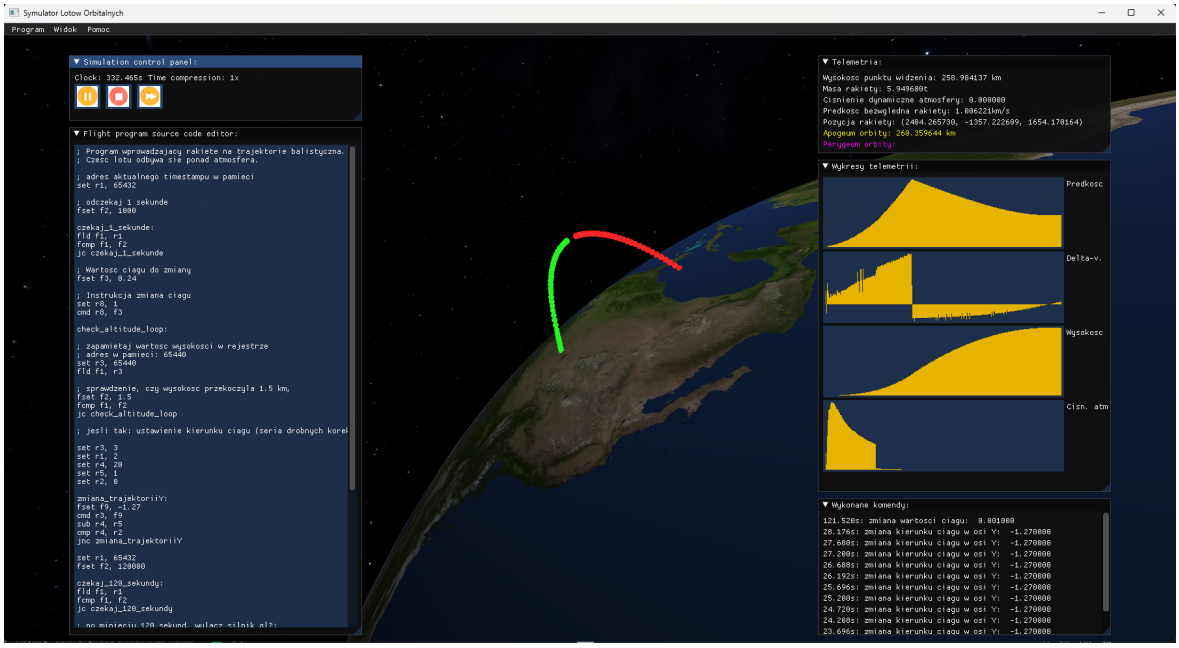

Figure 7.1: Trajectory prediction mode, source: self-elaboration (screenshot)

In 7.1 we can see an example plot of the predicted trajectory (red) and the route traveled so far (green). The example is for ballistic flight.

Chapter 8

Technologies and tools used

The selection of appropriate technologies was the subject of long-term research. In this chapter, we will discuss the reasons for choosing certain tools over others.

8.1 Usage of C++ and OpenGL

The most important tool used in the project is the C++ language. Before starting work on the implementation, the author compared the capabilities of various programming languages and libraries. The following combinations were considered:

- C++ and OpenGL. Strong language, open, mature and cross-platform API.
- C++ and DirectX. Same as above, but a little easier to use API.
- Java and OpenGL. Easier to use language.
- JavaScript and WebGL (with a high-level ThreeJS library). Possibly the easiest combination to use, but with strong undesirable features: huge memory and performance overhead.
- Rust and OpenGL: Recently fashionable and rapidly gaining popularity language with performance similar to C++, but putting more emphasis on security.

Ultimately, the author decided to use C++ and the OpenGL API. The following features of this combination have proven particularly useful:

- Easily accessible documentation and support. C++ is a very popular and mature language. As well as the OpenGL API. A lot of proven and very valuable software has been written in these technologies.
- Independence from the operating system. Programs written in C++ and OpenGL (if written correctly) can be compiled and run on various operating systems: Windows, Linux, MacOS, FreeBSD. This is not possible when combining C++ or C# and DirectX.
- Excellent integrated development environment - Visual Studio 2022 and Visual Studio Code. Among other languages, only Java has an IDE at a similar level. The use of a modern visual debugger made the work much easier.
- Operator overloading. This feature of the C++ language allowed you to write code that reflected mathematical equations in a more elegant and easy to understand way.

- A promise of higher performance than languages like Java or JavaScript. Although we have to admit that with modern hardware capabilities it matters less than it used to. However, in the author's opinion, it is still important, especially if this software is to be used in schools.

Operator overloading and other advanced C++ features used in this paper are described in [6].

The use of pre-build game engines such as Unreal Engine or Unity was also considered. Unfortunately, none of them can successfully create a world of the size we need [27]. It is possible, but difficult, and requires "hacking" the existing mechanisms of the engine [28]. Therefore, the author decided to write his own low-level 3D visualization engine. The issues related to this process were described in the previous chapter.

8.2 Other important libraries

In addition to OpengGL, the following libraries were used during development:

- OpenGL Mathemacics (GLM) - linear algebra (matrix-based operations) and other math function libraries extensively used in 3D visualization and physics calculations.
- Dear ImGui - A graphical user interface library for drawing a window layout in the main application window. Several controls are used, including: buttons, menus, text boxes, e.t.c. [29]
- Assimp library - Open Asset Import Library - a tool for reading 3D models of objects such as: rocket, clouds, trees, e.t.c. [30]
- STB Image library - stb_image.h. A single file header library used to load textures and other images into memory [31].

8.3 Assets used

All fallowing assets were shared as free of charge and free to use for everyone.

- 3D model of the rocket in obj format [22].
- Models of clouds, trees and launch tower. Models taken from the library included with the free (comes with Windows) program from Microsoft called 3D Builder.
- Planet surface textures and sky space textures [23] [24].

8.4 Integrated Development Environment

In the initial phase of the project, when there was time to experiment and test various tools, the prototype was written in the gcc compiler under the Linux operating system. IDE was Code::Blocks. However, as the work progressed, the need for an advanced application building system became more and more visible. Therefore, the author decided to temporarily switch to the Windows operating system and start using Visual Studio 2022. Currently, the plans are to use the CMake build automation system to create a universal project that will target both the Windows MSVC compiler and the Linux gcc compiler.

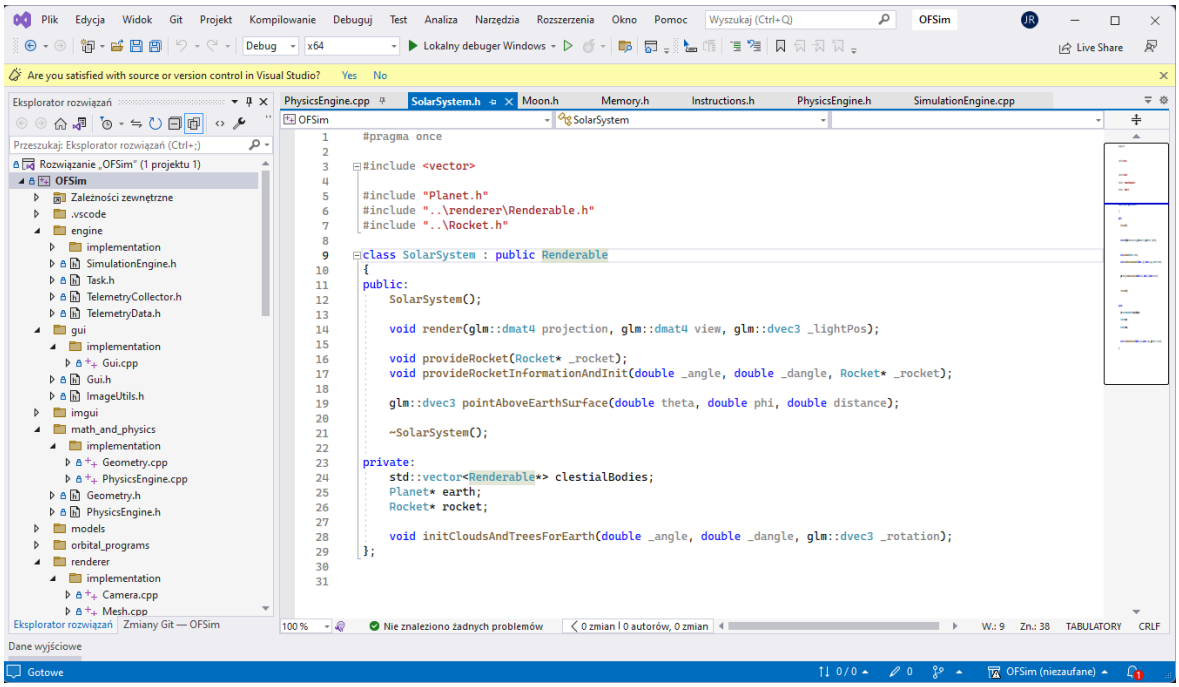

Figure 8.1: Visual Studio 2022

Chapter 9

Application from the user perspective

The user interface of the application has been designed to be as ergonomic and informative as possible. However, thanks to the use of the ImGi library and simple but effective techniques such as texturing and dynamic lighting, we also managed to achieve a fairly aesthetic overall appearance of the application.

9.1 Graphical User Interface

The application is organized as a multi-window application within one “main” program window. The screenshot 9.1 shows the main application window and most of the available internal windows.

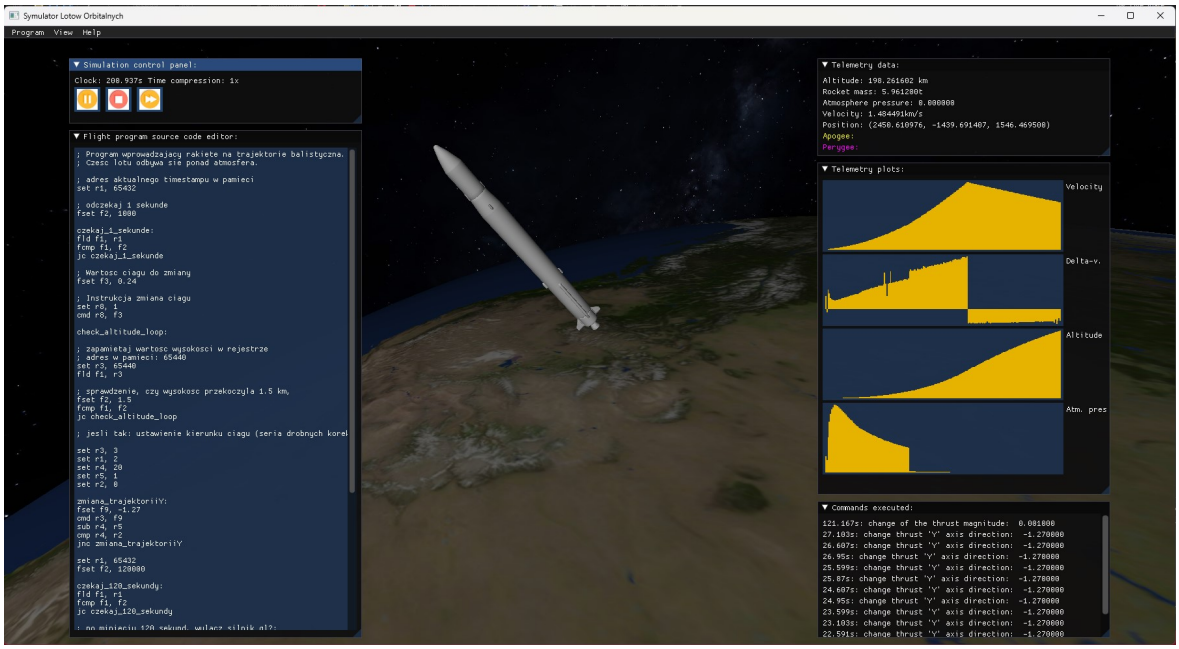

Figure 9.1: Application main window

On this screen we can see several important windows that correspond to the functions of the ap-

plication, which will be discussed below. Each of these windows can be enabled or disabled in the main menu of the application. The main 3D visualization appears under the application windows (as a background): depending on the application mode, it will be a simulation, trajectory prediction or demonstration.

9.2 Functions and possibilities

In the following sections, we will cover the key features of the application.

9.2.1 Simulation control

Figure 9.2: Simulation control panel

In the simulation control window, the user can start, pause and stop the simulation. Clicking the stop button will restart the simulation and save the program written in the code editor window to disk. One of the more useful features is time compression: clicking the "forward" button several times will speed up the simulation clock up to 16 times.

9.2.2 Source code editor

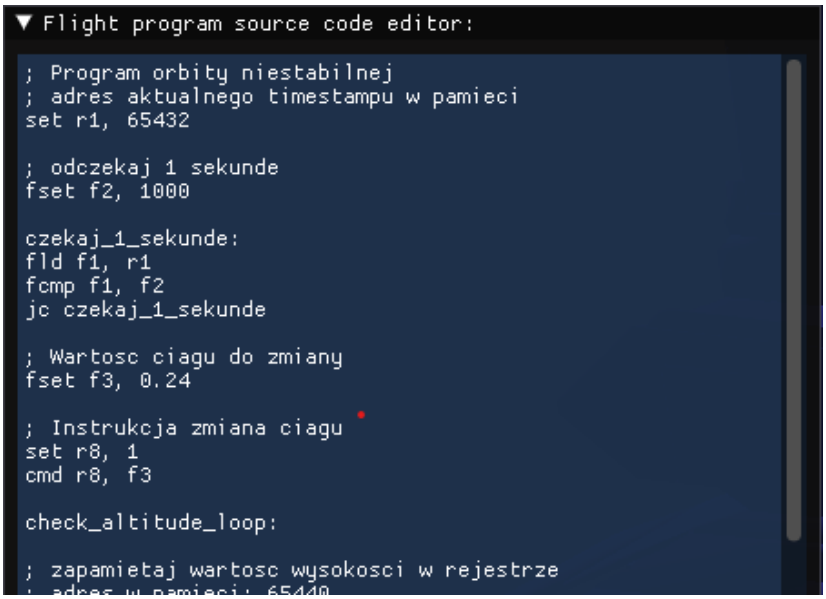

Figure 9.3: Source code editor

In this window, the user can view and edit the source code of the currently loaded flight program. Programs written or edited in this window are executed by the virtual machine after being translated into byte-code.

9.2.3 Telemetry data window

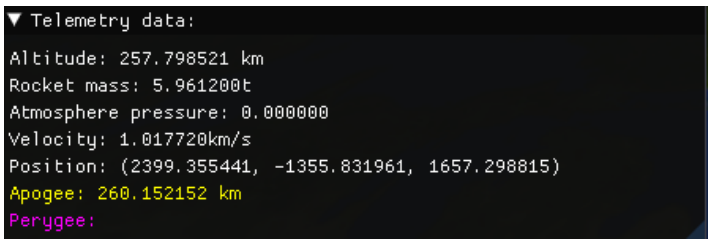

Figure 9.4: Telemetry window

This window dynamically displays the most important flight parameters of the rocket. This includes the perigee and apogee of the orbit (if orbit has been reached).

9.2.4 Telemetry plots

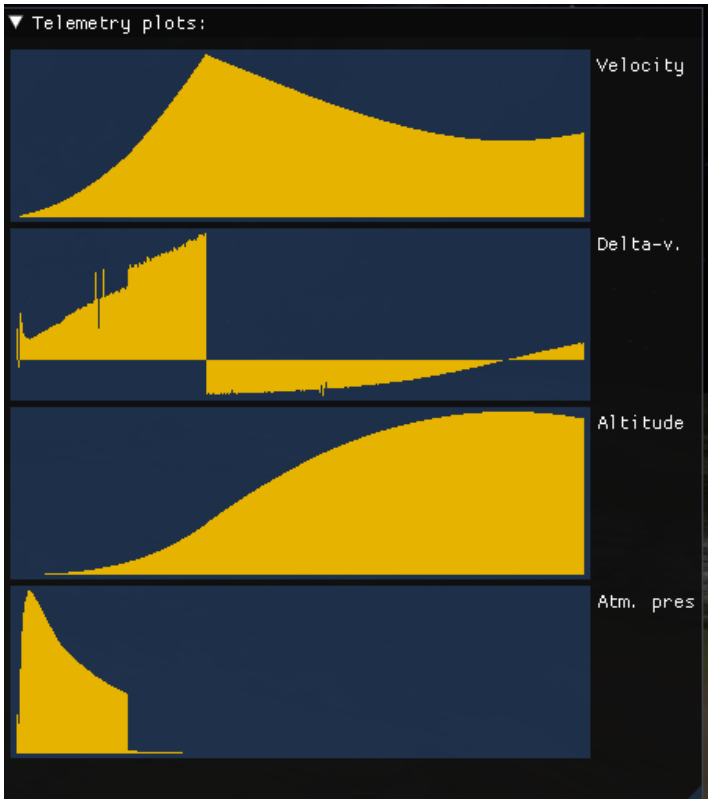

Figure 9.5: Telemetry plots

The Telemetry plots window displays the history of selected telemetry data in a graphical form.

9.2.5 Executed commands list

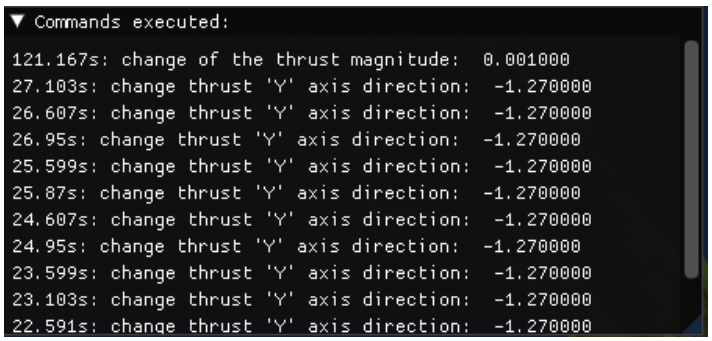

Figure 9.6: Executed commands history

This window shows the history of executed commands. Each entry begins with a timestamp that indicates the exact execution time.

9.2.6 Orbital programs loader

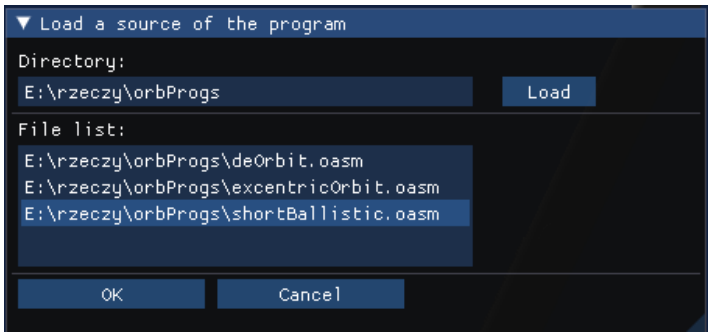

Figure 9.7: Program loader

The Loader gives the user the ability to load created programs from a specific directory.

9.3 Examples of the Virtual Machine programs

Three demo flight programs are provided with the application.

- The first program is the control program for simple ballistic flight, the highest phase of which is above the atmosphere.
- The second program controls orbital flight. The orbit is highly eccentric and unstable. In the flyby phase near the perigee of the orbit, the rocket enters the upper atmosphere, which causes the rocket to gradually decelerate. The apogee and perigee of the orbit decrease with each successive orbit - the rocket will eventually fall to the Earth.
- The third program is a modification of the second program. At some point before reaching apogee, the control program inverts the rocket and start the main engine again. This causes a significant deceleration of the rocket (the so-called de-orbit maneuver). Eventually, the rocket fails to reach orbit and crashes into the ocean.

Listing 9.1 displays short ballistic program written in the Orbital Flight Simulator assembly language.

Listing 9.1: Ballistic program

```
; wait one second before starting the engine:

fset f2, 1000
set r1, 65432
wait_1_second:
  fld f1, r1
  fcmp f1, f2
jc wait_1_second

; set the engine thrust value and send the command to the rocket:

fset f3, 0.24
set r8, 1
cmd r8, f3

; wait for the rocket to reach an altitude of 1.5 km:

check_altitude_loop:
  set r3, 65440
  fld f1, r3
  fset f2, 1.5
  fcmp f1, f2
jc check_altitude_loop

; set the register values needed to execute
; the trajectory change commands:

set r3, 3
set r1, 2
set r4, 20
set r5, 1
set r2, 0

; a loop responsible for sending a series of commands
; to the rocket about changing
; the direction of the thrust vector:

change_trajectory_Y:
  fset f9, -1.27
  cmd r3, f9
  sub r4, r5
  cmp r4, r2
jnc change_trajectory_Y

; a series of instructions responsible for
; waiting 120 seconds:

set r1, 65432
fset f2, 120000
wait_120_seconds:
  fld f1, r1
  fcmp f1, f2
jc wait_120_seconds

; after exceeding the time, sending a command
; to cut off the ignition
; (in practice, reducing it to the value of 0.001):

fset f3, 0.001
```

```
set r8, 1
cmd r8, f3

; the last loop waiting for the altitude to exceed 100 km,
; after exceeding it, the program stops:

set r3, 65440
fset f2, 100
finish_check_altitude:
  fld f1, r3
  fcmp f1, f2
jc finish_check_altitude

halt
```

Comments in the program code describe the actions of individual sections of the code. The idea of this program is to use short loops that send commands to change the rocket's trajectory or allow the program to wait for external conditions to occur. Trajectory changes occur at exact points in time or at exact points in altitude. The current altitude and time points are read from the main memory from specific addresses. These addresses are described in the memory organization section of this article.

The other two demo programs work in a similar way. However, there are more critical points in time and space to consider in their case. Rocket flight programming is often limited to checking the occurrence of certain conditions and, if they occur, performing appropriate calculations in order to decide on the commands transmitted to the rocket. The key elements to make a decision are: the real-time clock and telemetry data saved in real-time in the memory of the virtual machine.

Chapter 10

Summary

Orbital Flight Simulator - the software described in this paper is the result of research and development work motivated by the desire to create a tool useful for learning programming and physics for people interested in rockets and space exploration. The author believes that the provided prototype can be used in high schools or by hobbyists. A working application has potential for future development. In this chapter, we will discuss possible applications and directions of development.

10.1 Discussion of educational aspects of the application

We can think of this solution as a development environment combined with a physics simulator. The target group of application users is to be high school students interested in physics, programming and computer architecture. As well as anyone interested in programming in a more low-level way.

Integrated assembly language is a low-level language: that is, it provides only the most basic instructions (so-called orders) that translate almost directly into VM op-codes. The consequence of this is that the programmer has to write much more code to achieve similar results as in higher-level languages. This approach has its positive and educational aspects. This type of programming teaches the user to better understand the architecture of the computer and better break down the problem into very small steps. On the other hand, the assembler included in the application, as well as the VM architecture itself, are relatively simple - much simpler than the architecture of modern computers. This gives a relatively smooth learning curve.

The included demo programs are quite simple, for example: one launches a rocket into an eccentric, asymmetric orbit. Writing programs that launch a rocket into an *any* orbit is much easier than launching a rocket into an orbit with *specific* parameters. This requires a better understanding of Newtonian physics and more advanced mathematical calculations. Our application naturally enables this because it contains a sufficiently accurate simulation of real-world physics. Therefore, the software that is the subject of this work may be suitable for use in physics lessons in high school.

10.2 Future improvement plans

The Orbital Flight Simulator featured in this paper can be quite a useful application, but there are some areas for improvement. In this section, we will discuss the author's plans for future improvements to the application.

- Physics simulation improvements. Adding some chaotic behaviour to the atmosphere simulation. This includes: winds, turbulence and weather simulation. The upgrade can also include rocket physics. At the moment it is a one-piece rigid body. In the future, it may be a system of connected rigid bodies with uneven and time-varying mass distribution.

- Addition of simulation of the entire solar system instead of just the earth, moon and sun. This includes calculating the actual orbits of celestial bodies and their own rotations. Any celestial body should obey the same laws of physics as a rocket. Therefore, the physics simulation engine should be extended to include all rigid bodies. This opens up completely new possibilities: e.g. writing interplanetary mission programs.
- Adding the ability to change the parameters of the rocket (mass, size, aerodynamics, e.t.c.) and the parameters of celestial bodies in the solar system.
- Adding higher-level languages than assembly language. Although these languages do not have to be that complicated, a good example in the author's opinion is the PL/0 language extended with the ability to define functions. Another interesting example would be the use of a high-level visual language built on algorithm flow diagrams.
- Improvements to the visual aspect: adding dynamic shadows and more believable atmosphere visuals.
- Adding procedural generation of planet surfaces or preparing 3D models of planets in software such as Blender. Considering the very large size of the planets and the need to model their surfaces with sufficient detail, a solution with procedural generation may turn out to be better.
- Adding usage of the CMake automatic build system. This will allow user to build the project using compilers other than MSVC under the Windows operating system. Finally creating versions for Linux, MacOS and Windows.

Bibliography

[1] J. Walker, D. Halliday, R. Resnick: *Fundamentals od Physics Extended 10th Edition*. John Wile & Sons. Inc. 2014

[2] F. Dunn. I. Parberry: *3D Math Primer for Graphics and Game Development Second Edition*. A K Peters/CRC Press, Taylor & Francis Group 2011

[3] A. Branicki: *W Strone Nieba - Interaktywna Szkola Astronomii* PWN 2017

[4] NASA Solar System Exploration *Basics of Space Flight* Chapter 5-1 https://solarsystem.nasa.gov/basics/chapter5-1/ [Accessed: 09-Feb-2023]

[5] G. Coldwind: *Zrozumiec programowanie*. PWN 2015

[6] J. Grebosz: *Opus Magnum C++* Wydanie II. Helion 2020

[7] P. Kiciak: *OpenGL i GLSL (nie taki krótki kurs)*. PWN 2019

[8] J. de Vries: *Learn OpenGL: Learn modern OpenGL graphics programming in a step-by-step fashion*. A K Peters/CRC Press, Taylor & Francis Group 2011

[9] Khronos Group *OpenGL specification reference pages*. https://registry.khronos.org/OpenGL-Refpages/gl4/ [Accessed: 24-Jan-2023]

[10] OpenGL Tutorials *Rotations and Quaternions*.http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-17-quaternions [Accessed: 24-Jan-2023]

[11] OpenGL Mathematics (GLM) *Manual*. https://github.com/g-truc/glm/blob/0.9.9.2/doc/manual.pdf [Accessed: 24-Jan-2023]

[12] Khronos Group *OpenGL history* https://www.khronos.org/opengl/wiki/History_of_OpenGL [Accessed: 02-Feb-2023]

[13] Computers in Spaceflight: The NASA Experience *The Apollo guidance computer: Hardware* https://history.nasa.gov/computers/Ch2-5.html [Accessed: 31-Jan-2023]

[14] F. O'Brien Ars Techina *A deep dive into the Apollo Guidance Computer* https://arstechnica.com/science/2020/01/a-deep-dive-into-the-apollo-guidance-computer-and-the-hack-that-saved-apollo-14/ [Accessed: 31-Jan-2023]

[15] Thomas G. Roberts *Spaceports of the World - Comparing Sub-Orbital and Orbital Trajectories*. https://www.researchgate.net/publication/332211453_Spaceports_of_the_World [Accessed: 24-Jan-2023]

[16] Michael J. I. Brown *Explainer: How do satellites orbit the Earth?* https://phys.org/news/2014-08-satellites-orbit-earth.html [Accessed: 27-Jan-2023]

[17] Oracle Corp. *Java Virtual Machine Guide* https://docs.oracle.com/javase/9/vm/java-virtual-machine-technology-overview.htm [Accessed: 30-Jan-2023]

[18] Kerbal Space Program *Manual* https://static-files.kerbalspaceprogram.com/pd/KSPedia-XB1.pdf [Accessed: 31-Jan-2023]

[19] M. Sweiger: *ORBITER User Manual*. http://orbit.medphys.ucl.ac.uk /download/Orbiter.pdf [Accessed: 24-Jan-2023]

[20] Song Ho Ahn *OpenGL Sphere* http://www.songho.ca/opengl/gl_sphere.html [Accessed: 02-Feb-2023]

[21] Polish Rocket Society *Polskie Towarzystwo Rakietowe* https://rakiety.org.pl/o-nas/ [Accessed 29-Jan-2023]

[22] Free 3D Rocket model: https://free3d.com/3d-model/rocket-v1–735447.html [Accessed: 25-Jan-2023]

[23] Milky Way texture: https://commons.wikimedia.org/wiki /File:Solarsystemscope_texture_8k_stars_milky_way.jpg [Accessed: 25-Jan-2023]

[24] Earth's surface texture: https://commons.wikimedia.org/wiki /File:Solarsystemscope_texture_8k_earth_daymap.jpg [Accessed: 25-Jan-2023]

[25] Zilog, Inc. *Z80 CPU User Manual* https://www.zilog.com/docs/z80/um0080.pdf [Accessed: 02-Feb-2023]

[26] Mos Technology, Inc. *MCS6500 Microcomputer Family Programming Manual, Revision A* MOS TECHNOLOGY, INC. 1976

[27] Tech Art Pub *What is the Maximum Size of an Unreal Engine 5 Map?* https://www.techarthub.com/what-is-the-maximum-size-of-an-unreal-engine-5-map/ [Accessed: 02-Feb-2023]

[28] Unity *Unite 2013 - Building a new universe in Kerbal Space Program* https://www.youtube.com/watch?v=mXTxQko-JH0 [Accessed: 02-Feb-2023]

[29] Dear ImGui *Guide* https://github.com/ocornut/imgui [Accessed: 02-Feb-2023]

[30] Assimp Library *Guide* https://assimp-docs.readthedocs.io/en/v5.1.0/ [Accessed: 02-Feb-2023]

[31] STB Image Library https://github.com/nothings/stb [Accessed: 02-Feb-2023]

[32] Lua language *Language description* https://www.lua.org/about.html [Accessed: 09-Feb-2023]

[33] *Example of illumination shader* https://learnopengl.com/code_viewer_gh.php? code=src/2.lighting/2.2.basic_lighting_specular/2.2.basic_lighting.fs [Accessed: 14-Feb-2023]

[34] *Example of illumination shader* https://learnopengl.com/code_viewer_gh.php? code=src/2.lighting/2.2.basic_lighting_specular/2.2.basic_lighting.fs [Accessed: 14-Feb-2023]

[35] *Example of model loading code* https://learnopengl.com/code_viewer.php ?code=model&type=header [Accessed: 14-Feb-2023]

List of Tables

List of Figures

Listings