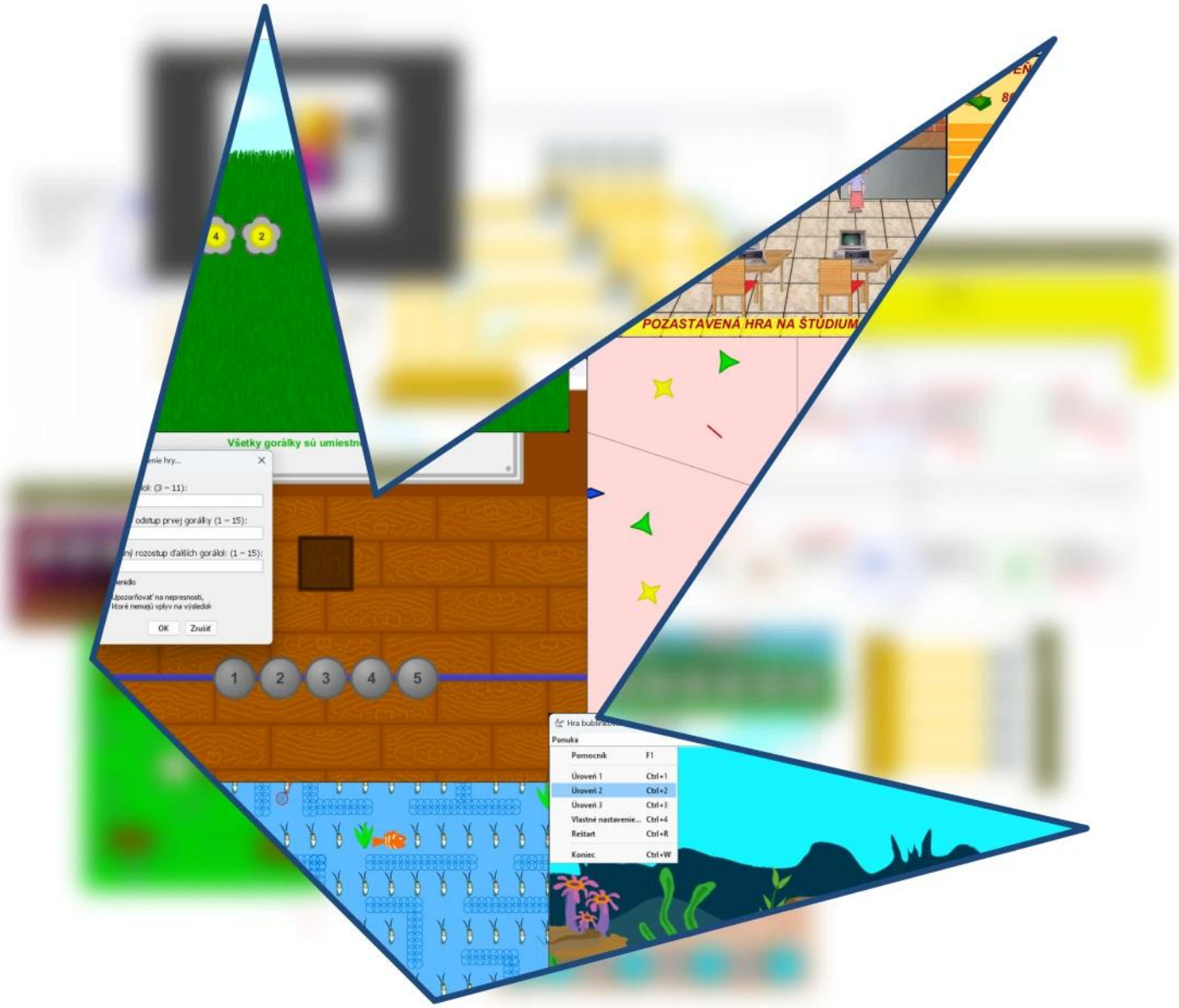
A Graphical Programming Framework for Didactic Purposes

Roman Horváth



2024

A Graphical Programming Framework for Didactic Purposes

Author:	Mgr. Ing. Roman Horváth, PhD.
Reviewers:	doc. Ing. Jana Jurinová, PhD., Ing. Gabriela Križanová, PhD.
Language Proofreading:	prof. PhDr. Anton Pokrivčák, PhD.
Typesetting:	Mgr. Ing. Roman Horváth, PhD.
Year:	2024
Publisher:	Faculty of Education, Trnava University in Trnava

ISBN 978-80-568-0680-7

https://doi.org/10.31262/978-80-568-0680-7/2024

Table of Contents

Introduction

In 2010, the development of a simple library began. At first, it was a single class named 'Korytnačka' (which, translated from Slovak to English, means turtle; its development started approximately in August 2010; in the meantime, the name changed many times; it was called, e.g., "Robot" or "a group of graphical robot classes"). 'Korytnačka' development was about making a tool capable of simplifying the teaching of programming in Java as part of the work on my dissertation [1]. The initial intention was to bring graphics into programming – "I can see what I am creating" – which should have a motivational effect. Another goal was to bring playful elements to programming in the sense that one of the goals of implementing this tool was to provide its potential use in creating simple games. Even though creating a simple game is not always as straightforward as it seems, it can motivate students to learn a lot. The tool underwent several modifications and many years of development until it reached the stage of the GRobot programming framework.

The GRobot Framework and Considerations for Possible Internationalisation

The GRobot framework is a more advanced successor of the former library. It supports graphics-oriented programming that is partially similar to Logo. The framework is written (like its predecessor) in the Slovak language, which means all methods and properties are labelled in the Slovak language, and the whole documentation is written in Slovak. (This is a consequence of a decision made at the outset of developing this tool, due to findings from the preliminary research for the author's dissertation.) This work will describe the process of gradual improvement of the programming framework from the stage of a simple library to its current form, which is still in the process of gradual improvement, refactoring, and restructuring. The documentation was officially (stressing the word "officially") released only for the first versions [2, 3]. The current version [4] is continuously updated, but it is no longer issued as a separate publication because this would "freeze it" (the officially released work must not be significantly edited) and also, no human resources were found to help its publication (language proofreading, opponents…). The reason was its current size – approx. 120 AH. During the years of its use, successes resulting from student feedback and the results achieved by them in the subjects where it was applied [5, 6, 7, 8, 9, 10] have been recorded.

Other tools that perform a similar function to this framework [15, 16, 17, 18] are available in English or multiple languages. That is why I am also considering making the framework available in English with the possibility of its transfer (translation) into any language. However, given the size of the framework and the documentation associated with it (circa 7.7 MB of code from which circa 120 AH makes the documentation), this process is not simple. One of the ways in which it could be implemented is the use of artificial intelligence.

To "translate" (or, in other words, to convert) the source codes into another language, one would need a customized Java language parser, a table of unambiguous translations of various elements (classes, interfaces, methods, fields…), in the proper contexts, and an engine that would connect these elements and apply them in practice. A separate problem would be the translation of the accompanying documentation. This is where the use of artificial intelligence could be advantageous.

Regardless, the Slovak version is here and working. The basal elements of the framework are the classes 'Svet' (World; originally Lúka – Meadow), GRobot (this is the framework's only element name that sounds and is universal and does not require a translation; it is derived from the Slovak 'grafický robot' which translates to a very similar "graphical robot" in English; originally it was 'Korytnačka' – Turtle and meanwhile

just Robot, without the G prefix), ‘ObsluhaUdalostí’ (EventHandler inspired by the concept of one simple library, where this was handled by the EventFactory class), and the associated class ‘ÚdajeUdalostí’ (EventData), which is convenient to import statically.

The class ‘Svet’ (World) became the central element of an application utilising the framework, which is meant to be the application window. Therefore, the class contains many static methods related to setting and getting the properties of an application window, but except that it provides much more. To name some of the features: it provides a general communication interface with the operating system, for example, creating a system tray icon and the associated notification bubbles, couple ways to measure the (system) time, handling an application timer, methods for processing the standard input, methods synchronously suspending the application waiting for various events (for a key, for a click, for user input, etc.), methods to start other processes, send emails, open web pages, create threads, print through the system dialogue, and the like.

The original ‘lúka’ – meadow (now transformed to ‘svet’ – world) did not meet the requirements for some simple games. The graphics always lay behind the turtles (now robots). Sometimes, it was useful to have some graphics in front of the turtles (alias robots), so, with the World, a new class arose: ‘Plátno’ (Canvas). Two static instances are created together with the World: ‘Podlaha’ (Floor) and ‘Strop’ (Ceiling). Each robot draws on the Floor by default, but its drawing pen can be redirected to the Ceiling, and everything painted or drawn on the Ceiling will be displayed in front of all robots.

From Basic Drawing to a Comprehensive Framework

The simplest robot’s feature is that it can draw a line while moving, just like the turtle in Logo language (hence turtle graphics). Still, compared to the original Turtle, it has many other features. The robot can move in relative Cartesian and polar coordinate systems and absolute cartesian systems. It can draw directly to the framework’s images (that can be reused for another drawing – it could be described as creating a stamp), do some primitive automated actions (like autonomous uniform straightforward movement, accelerated movement, and uniform or accelerated rotation), generate some basic (dots, circles, polygons, rotated ellipses, squares, and rectangles, etc.) or more advanced (rounded rectangles, stars, regular n-gons, eggs, etc.) Java shapes, draw the outlines of generated shapes or fill them, draw texts, and so on.

The framework can also play sounds, has many event-handling features (mouse, keyboard, timer, animation control, World window events handling, console events handling, automatic configuration saving and loading…), can work with text files, and much more. Many methods are polymorphic, some are categorised as getters or setters (not having the English prefix in its name – hence the use of special icons within the documentation), and some robot’s methods are intended to be overloaded so they can be invoked automatically in specific situations and work just like the event handlers – the programmer can specify what to do when the robot is passive, when it gets out of the bounds (of the Canvas), when it stops and so on.

Over time, the original focus on a purely didactic application featuring drawing robots on a graphical Canvas shifted towards enabling the creation of more full-fledged applications. In its early stages (before it evolved into a full-fledged framework), the system included various auxiliary objects (classes) such as ‘Obrázok’ (Picture), ‘Zvuk’ (Sound), ‘Farba’ (Colour), ‘Zoznam’ (List), and ‘Súbor’ (File). Gradually, however, other elements were added, inspired by former didactic software such as Logo Imagine [15]. These additions included components like buttons (class ‘Tlačidlo’), context items (class ‘KontextováPoložka’), context menus (class ‘KontextováPonuka’), notepads (class ‘PoznámkovýBlok’), scroll bars (class ‘RolovaciaLišta’), and more.

Ultimately, the framework expanded significantly: from the original 15 embedded classes, extensive refactoring and additions led to over 50 separate classes, nested classes, and interfaces.

In addition to classes serving the initial didactic purpose—many of which were considerably enhanced (for example, with classes like 'Bod' – Point, 'Častica' – Particle, 'Roj' – Swarm, 'Plazma' – Plasma, and others)—new classes were introduced for working with the operating system and its interfaces (such as 'Svet.PríkazovýRiadok' – World.CommandLine, 'Okno' – Window, 'Tlač' – Print, 'Spojenie' – Connection). Furthermore, new classes were created to support internal scripting (e.g., 'Skript' – Script), as well as to extend the framework's capabilities for data import and export (e.g., 'SVGPodpora' – SVGSupport, 'Filtre' – Filters, 'Archív' – Archive). Simultaneously, the documentation grew in scope, incorporating numerous usage examples, illustrative images, explanations of various concepts, and more. This rich documentation further facilitated both educational and practical use of the framework, making it more accessible for users with different needs and levels of expertise.

Event Handling

The framework contains a centralized class that processes all events handleable directly by the framework ('ObsluhaUdalostí' – EventHandler), including the most common events associated with working with the window (that means that the event factory centralizes many more events, other than window-related), and an affiliated class that conveys information associated with the creation of individual events ('ÚdajeUdalostí' – EventData). Most of the events can also be handled by individual robots – by overriding the method intended for that purpose (e.g., 'klik' – any mouse button click).

In addition to handling basic events like mouse clicks, the framework's event system is versatile enough to manage more complex, custom events. This allows developers to introduce new interaction patterns or behaviours into the framework without having to rewrite core components. For instance, events can be triggered by specific conditions in the robot's environment (e.g., 'prekreslenie' – repaint, 'zastavenieAnimácie' – animationStopped, 'spracovaniePríkazu' – commandProcessing, 'ladenie' – debugging, etc.) or through interaction with other objects in the graphical interface (e.g., 'voľbaTlačidla' – buttonPressed, 'zmenaVeľkostiOkna' – windowSizeChange, 'ukončenie' – exit, etc.). The modular design of 'ObsluhaUdalostí' (EventHandler) ensures that adding or modifying event types is straightforward and does not disrupt existing functionality.

Using the 'ÚdajeUdalostí' (EventData) class to encapsulate event details further simplifies the process of passing data between objects, ensuring that each event carries all the necessary information without introducing unnecessary complexity. Specifically, event handlers can relieve themselves from receiving parameters that they may not even need to use and are able to read them additionally through the corresponding methods of the 'ÚdajeUdalostí' (EventData) class. The up-to-dateness of the data passed by the event objects of this class is ensured internally.

Moreover, by decentralizing event management from a dedicated class, the framework enables extended handling of asynchronous inputs and responses. This is particularly useful in scenarios where multiple robots (or their descendants) are active simultaneously, as it allows for smoother coordination and communication between different elements of some implemented application (didactic or other). It is realised through a so-called system of calls to actions ('systém výziev' in Slovak).

The ability to extend event handling to both built-in and custom events enhances the framework's flexibility and provides a solid foundation for building more interactive and dynamic applications. This approach encourages the students (or other programmers) to explore more deeply how robots and their environment can respond to stimuli, making the event-handling system a motivating component in the framework's overall architecture.

Teaching Object-Oriented Concepts with Robots

Regarding the original purpose of the framework – to teach programming – we can use all the features to better explain the object-oriented features, like the method overloading, polymorphism, and the purpose of various object-oriented mechanisms. Here is an example showing how to program a robot to follow the last mouse click position automatically; let me do it in the Slovak language first:

```
rýchlosť(10, false);

new ObsluhaUdalostí()
{
    @Override public void klik()
    {
        cieľNaMyš();
    }
};
```

Now let me translate the code back to English to make the example more understandable:

```
velocity(10, false);

new EventHandler()
{
    @Override public void click()
    {
        targetToMouse();
    }
};
```

The example must be enclosed in the class definition that extends (inherits from) the GRobot class. The fastest way is to put it in the constructor, and it could look like this (bilingually):

```
import knižnica.*;

public class Príklad extends GRobot
{
    private Príklad()
    {
        rýchlosť(10, false);

        new ObsluhaUdalostí()
        {
            @Override public void klik()
            {
                cieľNaMyš();
            }
        };
    }

    public static void main(String[] args)
    {
        Svet.použiKonfiguráciu("Príklad.cfg");
        new Príklad();
    }
}
```

```
import library.*;

public class Examle extends GRobot
{
    private Examle()
    {
        velocity(10, false);

        new EventHandler()
        {
            @Override public void click()
            {
                targetToMouse();
            }
        };
    }

    public static void main(String[] args)
    {
        World.useConfiguration("Examle.cfg");
        new Examle();
    }
}
```

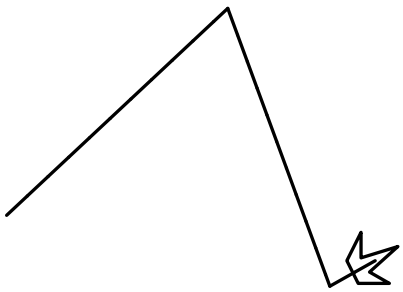


Figure 1. A preview of the possible result (depending on where the user clicks).

It is necessary to mention here that within the framework's application, only one EventHandler can be active simultaneously. (This principle was implemented for efficiency reasons.) To prevent confusion, creating another handler throws an exception by default. This can be configured, and multiple handlers can be created and activated alternately, but it must be explicit.

The way of handling the events directly by robots is more recent. Programmer can override one of the purpose-built methods of the GRobot class, like 'klik' (click). In this context, the GRobot class has almost an exact "copy" of the original 'ObsluhaUdalostí' (EventHandler) class built-into its code. So, the example above can now be rewritten as follows (the resulting behaviour will be exactly the same):

```
import knižnica.*;

public class Príklad extends GRobot
{
    private Príklad()
    {
        rýchlosť(10, false);
    }

    @Override public void klik()
    {
        cieľNaMyš();
    }

    public static void main(String[] args)
    {
        Svet.použiKonfiguráciu("Príklad.cfg");
        new Príklad();
    }
}
```

```
import library.*;

public class Examle extends GRobot
{
    private Examle()
    {
        velocity(10, false);
    }

    @Override public void click()
    {
        targetToMouse();
    }

    public static void main(String[] args)
    {
        World.useConfiguration("Examle.cfg");
        new Examle();
    }
}
```

By utilising the robots in the class, practical examples used during lectures and exercises have been markedly transformed, too. The transformation was mainly connected to the change of thinking and the way of teaching in parallel because teaching utilising the graphical robot means using and inducing slightly different ways of thinking about the problems. Some kinds of examples were clearly unsuitable for use in the classroom while teaching utilising the robot. Finding equivalents for algorithmic problems that we used (in the class) before and translating them into a graphical form took time and/or intellectual energy so that everything could not be transformed immediately. Several examples were held provisionally in the older ("verified") form. The transformation from computationally/algebraically ("mathematically") based jobs to visually/graphically ("geometrically") based jobs required a long-term collection of experience and searching for information and inspiration in a broader context. However, our experience shows that applying these changes in the programming courses made sense.

Loops:

When searching for didactic elements to teach various basic programming concepts, different visualizations that could facilitate their understanding were devised. For example, this code fragment demonstrates the functioning of two loops do-while and while consecutively:

```
kružnica(180);
farba(ružová);

do
{
    náhodnáPoloha();
    kruh(3);
}
while (vzdialenosťOd(0, 0) >= 180);

farba(šedá);

while (vzdialenosťOd(0, 0) < 180)
{
    náhodnýSmer();
    vpred(15);
}
```

```
circle(180);
color(pink);

do
{
    randomLocation();
    disk(3);
}
while (distanceFrom(0, 0) >= 180);

color(grey);

while (distanceFrom(0, 0) < 180)
{
    randomDirection();
    forward(15);
}
```

The first loop generates random positions for the robot until a position is found that lies within a circle with a radius of 180 points (with the centre in the middle of the canvas). At this point, it can be explained to students that this loop must be of the do-while type because it is necessary generate the position at least once. If it was not used this loop type and used a while loop instead, it would be necessary to generate the position once before the loop and again within the body of the loop (which would be inconvenient).

The second loop rotates the robot in a random direction and moves it forward (in that direction) by 15 points, but only if the robot is present within the same circle mentioned in the description of the first loop. Thus, the robot jumps in a random direction until it exits the circle. Here, students are told that, in principle, both loops could be used. However, if, for example, it is assumed that the robot's presence within the circle is of critical importance (this could be likened to a critical condition, such as one that might lead to writing data into a database), and the robot must not, under any circumstances, move itself while being outside the circle, then the use of a while loop is crucial and irreplaceable. The condition of the loop simultaneously serves as a safety measure to prevent anything unintended from happening, which, in extreme cases, could, for example, result in our application crashing.

Naturally, in this example, there is a double safeguard. The first loop ensures that the robot is inside the circle before it starts moving, so the use of a while loop in the second case is an additional layer of protection to ensure that the robot does not move if it is already outside the circle. However, it can explained to students that this cannot always be relied upon. What if the first loop was in a different module of the application, for which another programmer is responsible, and what if that programmer introduced an error?

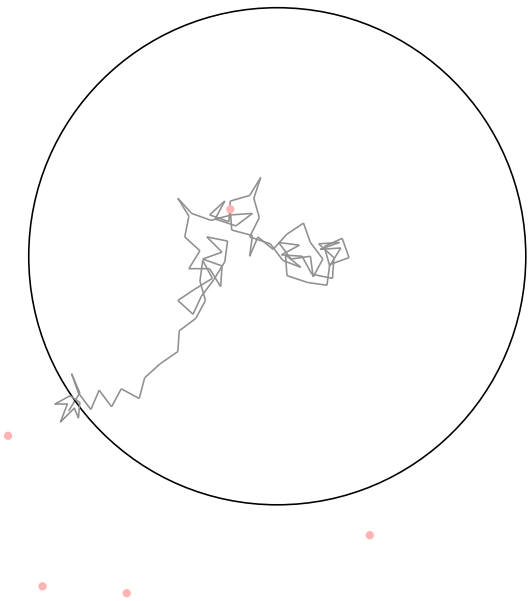


Figure 2. One of the possible results of the example.

This method of teaching – where the results are not given numerically but visually – gives the students a higher level of motivation, which was proven through random interviews with students. Ultimately, students must deal with all mandatory curricula, no matter whether they did or did not learn the programming this way. The "fall" into the environment of a formal expressing and strict syntax is milder when programming is taught using the graphical robot, where students can directly see the graphical results of their effort…

Framework Evolution and New Integer Methods

In 2019, converting the library (group of classes) into a programming framework was started. From now on, the framework files will be delivered in a separate Java package in a compiled state (in a .jar file). During this conversion process, several hidden bugs were uncovered. They were related, for example, to the coincidentally same naming of the parameters of some functions with the naming of the private fields of the original group of classes. The code was translatable but buggy.

In 2019, new quite interesting methods for working with whole numbers were added to the framework, specifically to the class 'Svet' (World; from more recent to older): 'jePrvočíslo' (isPrime), 'celéNaRímske' (integerToRoman) and 'rímskeNaCelé' (romanToInteger).

The method 'jePrvočíslo' (isPrime) allows us to assess whether the specified integer is a prime number. The method is optimised not to check each integer from the range from one to the given integer as the possible divisor. I did a quick survey in this area prior to the start of implementation. I used the most common optimisations, which were: (1) omitting all divisors greater than half the value of the verified number since it is clear that no integer divisor can be present within that range and (2) dividing only by prime numbers less than or equal to half the value (see the reasoning below) of the verified number because the remaining divisors are not applicable based on the principle of decomposition to the prime divisors. In this context, the framework (namely the class 'Svet' – World) creates a map of primes that have already been verified. The map increases proportionally according to the last verified number, although it is cleared after each restart of the application utilising the framework, which means the framework does not persistently store the computed prime numbers.

Let us discuss why my method considers divisors “up to” half the range, rather than just up to the square root. My initial educated guess was that dividing by two is essentially a bit shift operation, whereas calculating the square root in Java is computationally much more demanding. For evaluating small values, I believe it is certainly more efficient to use division rather than the square root. My assumption was also that this method would only be applied to relatively small values of prime numbers. However, I see room for further optimization: for values below a certain threshold (e.g., 500 or 100), division could be used, while for larger values, the square root could be applied instead. Additionally, I see an opportunity to conduct a performance analysis of the algorithm in both versions.

The methods ‘celéNaRímske’ (integerToRoman) and ‘rímskeNaCelé’ (romanToInteger) convert integers to Roman numerals and back. This feature can be used in a variety of creative ways. For example, Roman characters could be replaced with chosen syllables, words, or phrases, allowing us to create innovative words, puns, or simple cyphers.

Example:

Various word games can be created using the three methods: ‘jePrvočíslo’ (isPrime), ‘celéNaRímske’ (integerToRoman), and ‘rímskeNaCelé’ (romanToInteger).

If, for example, we replace the first three characters of the Roman numeral system with the following syllables:

I – to; V – ta; X – ma,

the first 20 (positive) integers will represent the following words:

I – to; II – toto; III – tototo; IV – tota; V – ta; VI – tato; VII – tatoto; VIII – tatototo; IX – toma; X – ma; XI – mato; XII – matoto; XIII – matototo; XIV – matota; XV – mata; XVI – matato; XVII – matatoto; XVIII – matatototo; XIX – matoma; XX – mama.

If we converted only prime numbers to Roman numerals (let’s say from the range 2 to 97) and if we replace the syllables according to the following table (columns represent the position of the numeral within the resulting number):

	if 1st	**if 2nd**	**if 3rd**	**if 4th**	**if 5th**
I	ma	mi	na	*(space)*	zu
V	sta	vi	t	*(space)*	–
X	ro	bo	ti	*(space)*	ha
L	e	vi	–	–	–
C	–	bo	–	–	–

we will get the following result:

II – mami; III – mamina; V – sta; VII – stamina; XI – romi; XIII – romina; XVII – rovina; XIX – romiti; XXIII – robona zu; XXIX – robona; XXXI – roboti; XXXVII – roboti zuma; XLI – rovina; XLIII – rovina zu; XLVII – rovit zu; LIII – emina; LIX – emiti; LXI – ebona; LXVII – ebot zu; LXXI – eboti; LXXIII – eboti zuma; LXXIX – eboti ha; LXXXIII – eboti zumami; LXXXIX – eboti zuro; XCVII – robot zu.

Some words make sense in Slovak language, some in English, other words are made up. The table could be refined to get more meaningful words. But here, the toying with ideas doesn't have to end. Variations are endless. We could continue, for example, by looking for four-digit numbers, where each digit would have the meaning of a word answering the questions: who? where? with whom? what does he do? This would give us sentences. Alternatively, we could also include numbers having fewer digits than four in the game, while we would fill in the missing numbers according to the chosen rules so that the resulting sentences would still make sense. We could also generate images instead of words. Everything depends on the imagination of the teacher.

Recent Improvement – class 'Okno' (Window)

Throughout the years of development, the framework applications worked within just a single application window. There was no option to create an additional window, which meant that an application that used the framework had to either create its own windows using other Java resources or it had to work within the default window. For the academic term 2022/2023, work began on incorporating this long-absent feature. A new class has been added to create and manipulate new application windows – 'Okno' (Window). The primary goal for the basic implementation was straightforward window creation and usage while maintaining the framework's existing functionalities in the new windows.

In Java, of course, any number of windows can be created. Still, the additional "external" windows could not be easily connected with the existing features of the programming framework without in-depth knowledge of the framework's inside processes. The new framework's class 'Okno' (Window) significantly simplifies this process. So, while this feature might seem trivial, it has become essential for some types of applications in particular, if it is needed to implement an application that can work on multiple screens simultaneously. This requirement could not be met in any other way than by creating additional windows for the application. In addition, it makes it easier to connect new windows with the existing features of the framework.

Each new window holds a component containing a picture instance since all constructors of the 'Okno' (Window) class require an 'Obrázok' (Picture) class instance as a mandatory argument. (Note that a series of constructors can also be seen as a single polymorphic constructor.) This seamlessly integrates with the framework's existing functionalities since any robot can draw in the 'Obrázok' (Picture, which was implemented long ago regardless of the new class; the new class just utilised this feature). This setup should clue the framework user (programmer) how to use any robot to draw contents into new windows.

Some constructors display the window automatically after construction. Those that do not explicitly specify whether the window should be automatically displayed or not leave the window hidden after construction. The window must be shown afterwards using the 'zobraz' (show) method. Some constructors allow naming the window, in other words, assigning a unique string identifier for addressing and distinguishing from other windows). This identifier is used to save basic window properties to the configuration file if the use of the configuration is enabled.

Like the main application window—the "world" ('svet' in Slovak) represented by a 'Svet' (World) class instance—the window class comes with various useful features. To name a few: adjusting the desktop colour, the mouse cursor shape, minimizing the window, switching between classic window mode and the full-screen mode, and so on. Methods such as 'farbaPlochy' (desktopColor) allow the programmer to set the colour of the window surface in different ways – the same way as it can be done in the 'Svet' (World) class – by instantiating the java.awt.Color class (or derivatives such as the 'Farba' – Colour frame class), by

implementing the ‘Farebnosť’ (Colourfulness) interface, or by defining the RGB values of a new colour. The shape of the mouse cursor while moving over its surface can be changed using the ‘zmeňKurzorMyši’ (changeMouseCursor) method, also like in the ‘Svet’ (World) class, as long-time users of the framework are used to.

In connection with the full-screen mode, it is necessary to point out the limitation of the Java virtual machine. Java does not allow for existing windows to go to the full-screen mode, which is probably due to limitations on various platforms. This is solved in the programming framework in such a way that when switching to full-screen mode, a new instance of the window (aka frame, aka javax.swing.JFrame) is created, and all components of the original frame are transferred to it. Access to the current window instance is mediated by the ‘Okno’ (window) method, which both the ‘Svet’ (World) class and the new ‘Okno’ (Window) class have implemented.

The full-screen mode change can be performed using different versions of the ‘celáObrazovka’ (fullScreen) method. The ‘oknoCelejObrazovky’ (fullScreenWindow) method returns an instance of the javax.swing. JFrame class if the window is in full-screen mode; otherwise, it returns null. A window may contain its own implementation of the way to change to the full-screen mode. The ‘zmenaCelejObrazovky’ (fullScreenTransition) attribute defaults to null but can be set to any of the default instances of the ‘ZmenaCelejObrazovky’ (FullScreenTransition) class. (There are two default implementations – hardware and software transition, which differ on each system.)

The ‘overujPočiatočnúPolohuOkna’ (checkInitialWindowPosition) global property was introduced in the framework’s earlier versions specifically to manage applications written in the environment with multiple screens. By default, this property is active. Let me explain its primary purpose. As an educator, I habitually developed prototype projects collaboratively with students during classroom exercises and made them available post-lesson for absentee students. Frequently, I encountered the issue where the main application window of such project persisted on my, i.e., “teacher’s,” secondary screen; thus, students downloading and launching the project on a system with a single screen (e.g., at home) could not see the window. If the student was unaware of how to relocate the window to their primary (that means the only) screen (for example, by utilizing OS Windows’ system-wide keyboard shortcuts), they would become disoriented and seek my assistance. Of course, the ‘overujPočiatočnúPolohuOkna’ (checkInitialWindowPosition) feature has been transferred from the main “world” window to the supplementary framework windows. When activated, this property ascertains that each newly generated window is within the confines of the current display devices at initialization; if not, the window is auto-shifted to the primary (assumingly default) screen.

As mentioned in the chapter about event handling, all events (including window-related ones) are centralised in the ‘ObsluhaUdalostí’ (EventHandler) class. Individual robots can also handle many of them. The ‘ÚdajeUdalostí’ (EventData) class helps identify the origin of these events and provides more useful data related to events. All events applying to ‘Svet’ (World) can also be applied to ‘Okno’ (Window). It could be dropping a file into the window, changing the size or position of the window, clicking into the window, etc.

A new method, ‘oknoUdalosti’ (eventWindow), was added to the ‘ÚdajeUdalostí’ (EventData) class to distinguish the origin of a window-related event. The method returns the instance of the last window that invoked any event connected to that window. If an event originates in the main application window (‘Svet’ – World), the method’s return value is null; otherwise, it is an instance of the triggering window. This system extends the framework’s functionality while keeping it compatible with older applications, so they do not

need to be modified. At the same time, this mechanism enriches the framework so that new applications can utilise it.

The following code shows the use of the class ‘Okno’ (Window):

```
import knižnica.*;

public class TestNovéhoOkna extends GRobot
{
    private final Okno okno;

    private TestNovéhoOkna()
    {
        skry();
        text("Klikni sem na zobrazenie " +
            "druhého okna.");

        Obrázok obrázok =
            new Obrázok(800, 600);

        kresliDoObrázka(obrázok);
        skoč();
        text("Toto je druhé okno.");
        odskoč(veľkosť() * 2);
        text("Klikni sem na jeho skrytie.");

        okno = new Okno(obrázok,
            "bočnéOkno");
    }

    @Override public void klik()
    {
        if (null == ÚdajeUdalostí.
            oknoUdalosti())
        {
            if (null == okno ||
                okno.zobrazené())
            {
                Svet.pípni();
            }
            else
                okno.zobraz();
        }
        else
            okno.skry();
    }

    public static void main(String[] args)
    {
        Svet.použiKonfiguráciu(
            "TestNovéhoOkna.cfg");
        new TestNovéhoOkna();
    }
}
```

```
import library.*;

public class NewWindowTest extends GRobot
{
    private final Window window;

    private NewWindowTest()
    {
        hide();
        text("Click here to display " +
            "the second window.");

        Picture picture =
            new Picture(800, 600);

        drawOnPicture(picture);
        jump();
        text("This is the second window.");
        recoil(size() * 2);
        text("Click here to hide it.");

        window = new Window(picture,
            "sideWindow");
    }

    @Override public void click()
    {
        if (null == EventData.
            eventWindow())
        {
            if (null == window ||
                window.visible())
            {
                World.beep();
            }
            else
                window.show();
        }
        else
            window.hide();
    }

    public static void main(String[] args)
    {
        World.useConfiguration(
            "NewWindowTest.cfg");
        new NewWindowTest();
    }
}
```

The code first prints the text “Click here to display the second window.” to the main window, then creates an instance of the picture, prints the text “This is the second window. Click here to hide it.” to the picture, and then creates the secondary window (using the picture). The secondary window is hidden by default. In the event handler ‘klik’ (click), the application checks which window was clicked to (null means the main window) and, accordingly, the secondary window is displayed or hidden. More specifically, clicking in the main window will show the secondary window (or beep if already visible), and clicking in the secondary window will hide it. Figure 3 shows the appearance of both windows after the application is launched and after they are displayed.

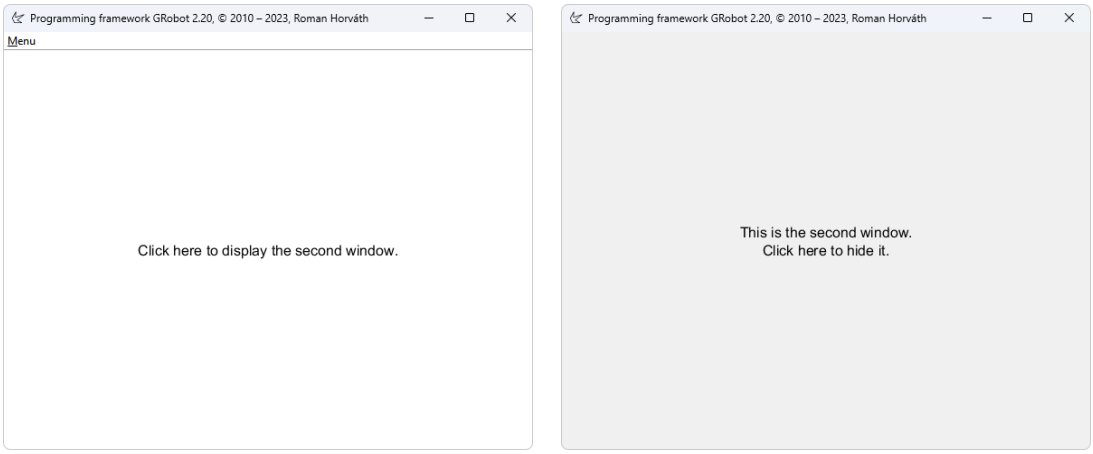


Figure 3. Appearance of windows after they are displayed.

Implementing the 'Okno' (Window) class was, as expected, more time-consuming than would be estimated by some independent observer if provided just the outline descriptive information about the class's purpose. It required over 3,000 lines of code and documentation written over a couple of weeks. Considering that the class uses already existing Java classes (from the Swing package), the observer could be surprised. However, the changes were not limited just to writing this class. A lot of code had to be added or adapted in the rest of the framework to make the class work with the rest and make it easy to use (which the example above tries to show). Nevertheless, I believe it was worth the effort, making the framework more powerful and flexible.

A brief report about the development progress was published in [38]. Since then, the class has been improved. Now, it supports moving existing framework controls (e.g., buttons or notepads created in the main window by default) from the main window to a specified additional window. This significantly enhances the possibilities of windows.

Use Cases

After establishing the basic capabilities and structure of the GRobot framework, it's essential to consider how it has been practically applied in various projects. In fact, the best way to test, extend or refine a framework is to create some projects utilising it [20, 21]. For our purposes, I will call such projects *use cases*. Over the years, countless such use cases have arisen. I will select and describe only some of them that I consider illustrative. I am the author of most of them, but some of them were created by my students.

The GRobot programming framework does not only provide tools for creating a virtual world and for creating, controlling, and influencing the behaviour of graphic robots in this world but also tools for the creation and management of various supporting elements with the help of which practically any graphic desktop (potentially didactic) application can be created. The concept may seem strange to Java programmers who are used to different programming styles. The world – the main window of the application in which the robots move – is created completely autonomously and implicitly by the first robot. The latter is then designated by the term "main robot." The class 'Svet' (World) is then rather a class used to manage various properties of the world. We can look at the class World as some "system class" (because, in addition to methods for managing the main application window, it also contains methods for closing the application, timer control, communication dialogues, data type conversion, pseudorandom number generation, memory management, basic sounds, and much more). Everything is summarized in the continuously updated documentation mentioned earlier. The documentation of the class 'Svet' (World) can be found specifically at this address: https://pdfweb.truni.sk/horvath/GRobot/Svet.

Due to the concept chosen in this way, the GRobot class contains constructors that allow defining the size of the canvas (or canvases) of the world and the title of the application window. It makes sense to use these constructors only for the first (main) robot; the other robots ignore those parameters (the default constructor is enough for them). Graphic robots can then move around in the world, draw lines or different objects, check their collisions with each other, and otherwise communicate. The robot can change its shape, size, speed of autonomous movement and rotation, generate areas serving different purposes, and so on. The robot's movement can be influenced by various methods, from which the particular groups look at the robot from different points of view. The basis is a movement using the concept of the so-called turtle graphics [19]. This concept moves the robot using a relative polar coordinate system, where the centre and orientation of the coordinate system are always adjusted to the robot's current position. Still, we can also use groups of methods working in the absolute or relative Cartesian coordinate system to move a robot. Continuously updated documentation of this class can be found at the following address: https://pdfweb.truni.sk/horvath/GRobot/GRobot.

Student's Didactic Computer Game – SuperShop

In 2015, student Monika Chnápková (now Ondreičková) created the didactic computer game SuperShop utilising the framework [22]. The game aimed to manage a computer cafe while learning about computer hardware. Customer service resembles the management of a mass service system. Satisfied customers generate profit, but when the "café owner" wants to advance to a higher level, he must pass a knowledge test on hardware, the curriculum of which is hidden in the "library of the store." The game created is quite complex, and its development remained essentially open. However, the game is another example of the usability of the programming framework:

Figure 4. A few screenshots from the computer didactic game SuperShop. Top left: customer payment; right: game paused for study purposes; bottom left: processing the customer's request; right: the test required to advance to the next level.

In addition to the primary didactic goal, the application supports the development of entrepreneurial thinking. It allows the player to gain or support the growth of the foundations of financial literacy. The owner of the "café" – shop, assigns seats to incoming customers interested in various services – writing and printing documents, playing games, sending e-mails, and the like. The owner must maintain the shop. He buys newer hardware and takes care of its operations. At any time, the player can switch to the study section and study the theory that he will need to advance to the next level of the game. The transition to the next level is conditional on passing a simple knowledge test composed of questions from the theory available to the player.

So, the game is mostly controlled by the mouse. The player clicks on controls, characters, hardware components, etc. The exception is the test. The player enters the answers in the form of a character in the input line of the programming framework. The player selects the customers who come to the shop by clicking on them, and they announce their requests. The player may seat a customer at the table if the shop is equipped sufficiently. Clicks also replenish printer consumables and collect payments from customers.

If the teacher invites the student to do so, then he can analyse the principle of the game while playing and try to figure out what the game develops in him, what helps him understand and how it moves him further, i.e., in what area, and what are the game's objectives, aims, and principles. This may cause the application to become a tool for self-reflection. Students can also be tested on the knowledge (contained in the game) outside the game's interface – by a test or a written work created and assigned by the teacher. (The game itself does not contain a test editor; however, a skilful teacher may edit the test's configuration file to use his own tests within the game. But the game does not save the results.)

Semestral Project – Aquarium

This project was created by a student, Zuzana Radošovská, as a semester project for the object-oriented programming course. It is not as extensive as the SuperShop project, but the student designed and implemented an exemplary object model within its scope. This is precisely how it should be done. If I were a beginner like her, I certainly could not have done it better myself (maybe even today, at least as a prototype).

Upon its first opening, I was pleasantly surprised, as I was not accustomed to such an approach. The student demonstrated a proper understanding of OOP concepts, so, unsurprisingly, I had no comments at first glance. (Today, with the benefit of hindsight, there might be some minimal comments, but in her category—beginner level—I probably would not find a better project.)

The student proposed four basic game objects: Fish, Wall, Food, and Obstacle. Fish is the player's character, Wall is an object you cannot pass through but does not harm the character, Food is an object the player needs to collect, and Obstacle is an object to be avoided as it deducts life. Furthermore, Obstacle is the parent class of three derived classes: Algae, Net, and Rod (fishing rod). Thanks to their common parent, all enemy objects can be evaluated in a single loop. The student also created her own graphics for the game. The object model and a game sample are shown in Figure 5.

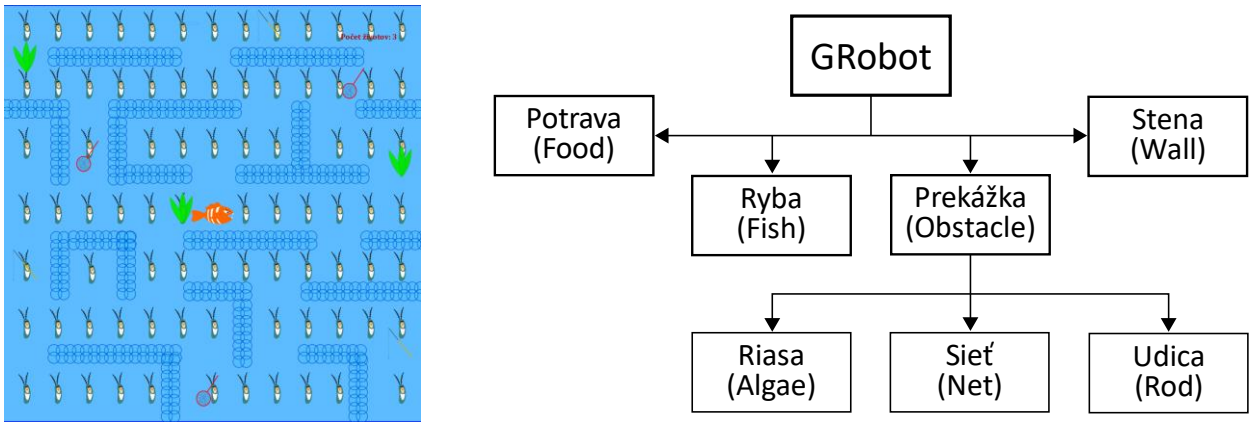


Figure 5. Game preview (left) and the object model created for game implementation (right).

Various Mini-applications

Several mini-applications were also created for didactic purposes. Some were made by the author of the programming framework, while to others, he was just an assistant. The latter was created as part of a student's final work – Daniel Kuszý's bachelor's thesis: Using Modelling and Simulations in the Interactive Educational Materials (2019).

Figure 6. Central mini-application [11].

A series of mini-applications is published at [11]. The central mini-application (Figure 6) launches seven didactic mini-applications focused mainly on the binary system (basically except for one). Six of them form pairs dedicated to one topic. Bitwise Addition and Bitwise Multiplication (Figures 7 and 8) focus on bitwise operations, the Convert to Decimal and Convert to Binary on conversions between binary and decimal systems, and Assembly from Decimal and Assembly from Binary on exploring the connections between the decimal and binary systems. The last one is the Typing Racer, which is aimed at practising fast typing on the keyboard – it can be used to practice touch typing either by self-discipline or under supervision.

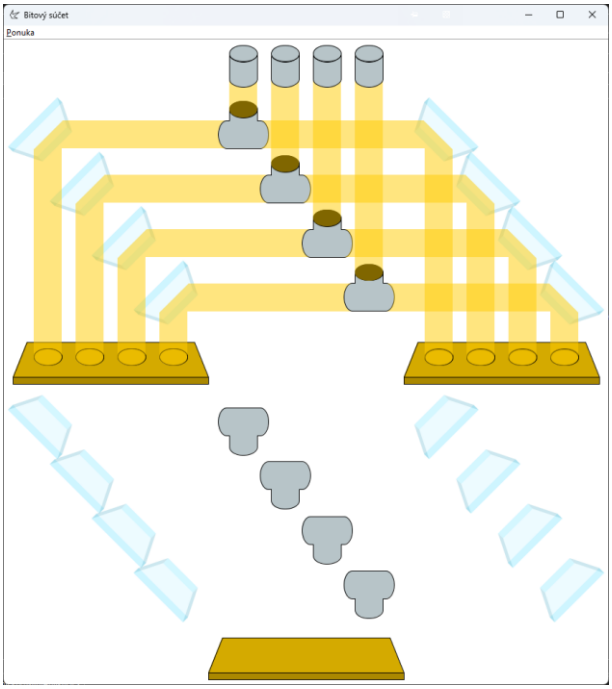


Figure 7. Mini-application Bitwise Addition [11].

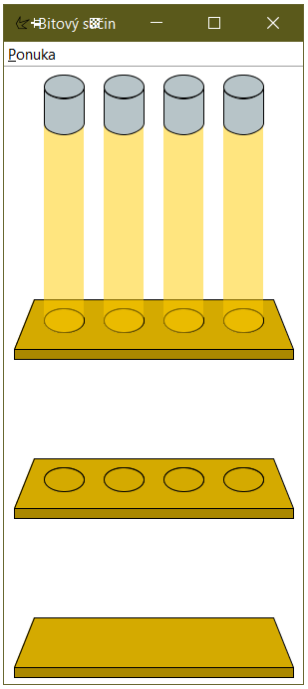


Figure 8. Mini-application Bitwise Multiplication [11].

Figure 7 shows a light-mirror simulation of the bitwise summation. If needed, the user can turn off and on the four lamps at the top to simplify the simulation. However, the most important elements of the simulation are the holes in the plates in the middle. The user can open and close them by clicking (the opening symbolizes a bitwise one; the closed hole symbolizes a bitwise zero). The result of the bitwise summation is instantly projected on the bottom plate, while the falling light beam symbolizes a bitwise one, its absence a bitwise zero.

Figure 8 shows a simulation of the bitwise multiplication. As with the bitwise summation, the user can turn on and off the four lamps at the top (thus simplifying the simulation if needed). Also, in this simulation, the most important elements are the holes in the plates in the middle, which the user clicks may open (set them to bitwise one) and close (set them to bitwise zero). Also, in this simulation, the result of the bitwise multiplication is projected on the bottom board (with the light beam, again, symbolizing the bitwise one and its absence zero).

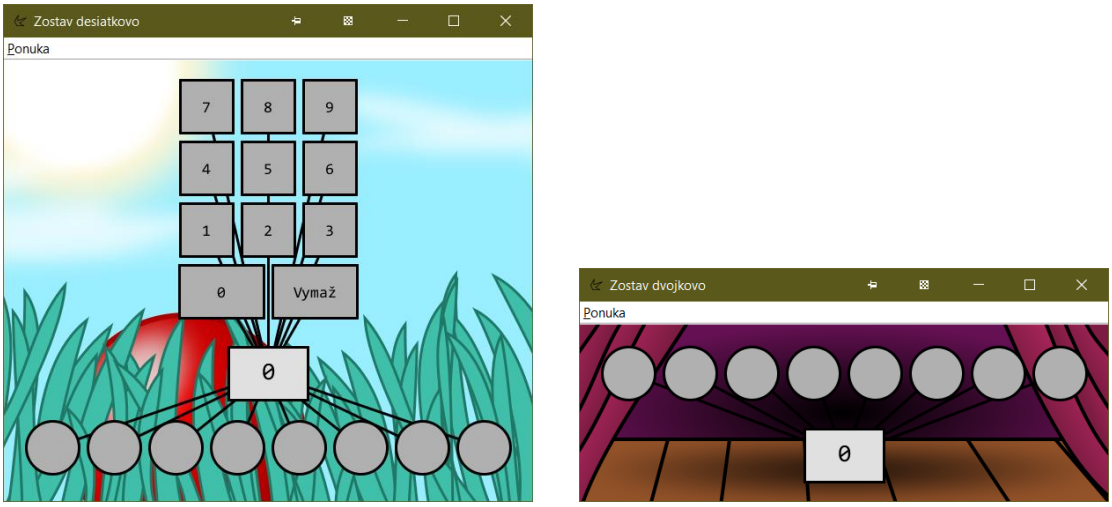


Figure 9. Two mini-applications: Assembly from Decimal (left) and Assembly from Binary (right) [11].

The pair of mini-applications in Figure 9 – Assembly from Decimal (left) and Assembly from Binary (right) – are used to explore the differences between the two number systems and find connections between them. Although in Figure 6 are buttons for those two mini-applications listed later (because they were finished later), I introduce them first because they are dedicated to learning, while the Convert to Decimal and Convert to Binary are rather for self-testing and improving the knowledge.

The Assembly from Decimal mini-application is for experimenting with how a decimal value looks in binary. It is a good idea to instruct users to try entering powers of two (1, 2, 4, 8, 16…). That's how they can see a single light bulb on (cyan means the bulb is on, which is a bitwise one, and grey is off, which is a bitwise zero). It should be further explained to them that when they enter any other value (that is, one that is not a power of two), then exactly such a combination of light bulbs will light up, which is the sum of the corresponding powers of two. If they imagine it in their head in parallel and recalculate, they should come up with some connections – new knowledge. The 'Vymaž' (Delete) button works like the backspace key on a computer keyboard.

The Assembly from Binary mini-application is for experimenting with how the individual bits (again represented by light bulbs that glow cyan when they represent bitwise one) affect the decimal value projected on display below them. Like Assembly from Decimal, the user needs to be properly instructed, and it should be explained to him that if he lights on just a single light bulb, he will get the power of two (1, 2, 4, 8, 16…). Lighting any combination of bulbs will get a value that is the exact sum of the "bulb values." To get visually and numerically more exact image, the right mouse button click can show and hide the numerical values of the bits (the powers of two they represent) – either by right-clicking on the bulbs individually or by right-clicking on the decimal display to show/hide all the values at once.

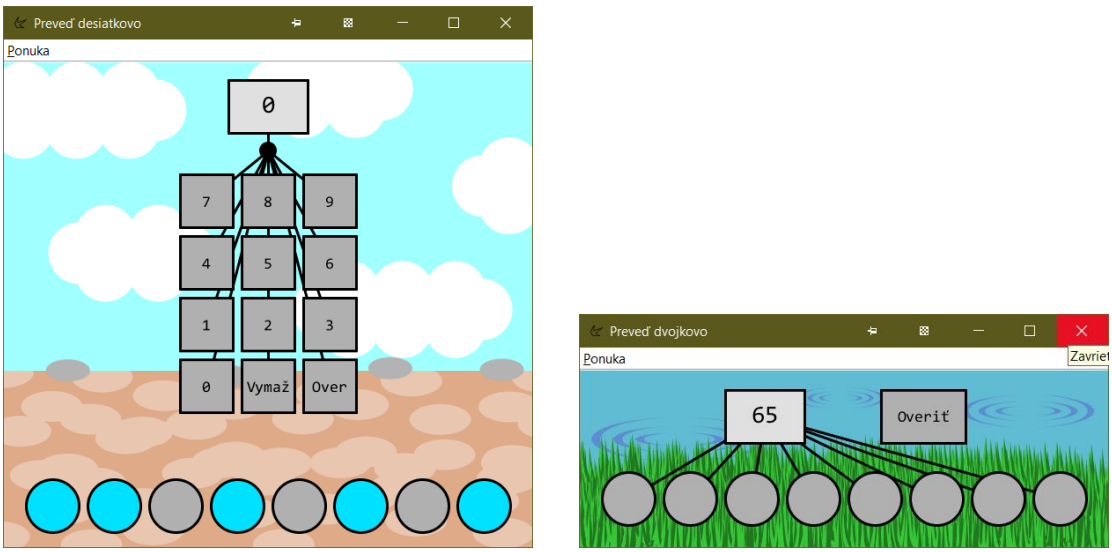


Figure 10. Two mini-applications: Convert to Decimal (left) and Convert to Binary [11].

The mini-applications in Figure 10 – Convert to Decimal (left) and Convert to Binary (right) – allow training conversions between these numerical systems. The Convert to Decimal is intended to allow the user to convert the number from binary to decimal system. At the bottom are randomly lit bulbs (cyan means on, again). The user must count the bit values and type in the result using a visual keyboard (keypad). The user must go blind this time. No numeric clues (powers of two) are available in this mini-application. The user can verify the correctness of what he typed with the 'Over' (Check) button, which then changes to the 'Znova' (Again) button and lights the decimal display to green if the solution is right or red if it is not. The button 'Znova' (Again) may then be used to generate a new combination of light bulbs. The 'Vymaž' (Delete) button works like the backspace key on a computer keyboard.

With Convert to Binary, the direction of conversion is opposite. Above the light bulbs, which are off by default, a small "display" shows a randomly generated value in the decimal system. The user must divide it into bits (i.e. into the binary system) and light the bulbs under the display accordingly (again, cyan means the bulb is on, which is a bitwise one, and grey means off, which is a bitwise zero). The user verifies the correctness of the solution with the 'Over' (Check) button, which changes to the 'Znova' (Again) button after the correctness/incorrectness of the solution is visualised – by lighting the decimal display to green/red. Using the 'Znova' (Again) button the user can generate another combination of bits.

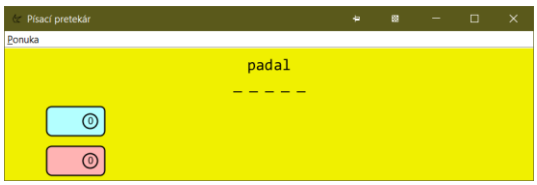


Figure 11. Mini-application Typing Racer [11].

Although Typing Racer (Figure 11) is unrelated to bitwise operations and the binary system, the application was created in parallel with them and, therefore, was assigned to this group. Its purpose is to provide a tool for practising fast typing on the keyboard, ideally with respect to finger placement. There are two "cars" in the gaming race. One (the red on the bottom) is controlled by the computer (it is moved forward in random steps), and the other (light-blue above the red one) is moved by entering the correct letters of the displayed

word. The game starts immediately (which should be corrected in the future) and is endless (which may or may not be corrected). When the “cars” reach the right edge of the screen, they will be projected back to the left part and the “beacon” on the “car” displaying a number will be increased. The number represents the count of laps the “car” has completed. So, anyone can immediately distinguish who is winning – the computer (red) or the player (light-blue). The words to be entered can be edited in the TextPretekára.txt file, but the game only selects words with the required length (5 characters).

Small Utilities for Screen Shading

Other small use cases include two small yet practical utilities that have been developed to address specific user needs: the “White Window” [12] and the “Any Colour Top Window” [13]. These utilities provide an unobtrusive method for overlaying portions of the screen with a window that lacks standard decorations, making it effectively invisible when properly positioned.

The “White Window” was originally created in response to an immediate request from a colleague who required a way to obscure part of the screen during the recording of an educational video. Thanks to the flexibility and capabilities of the GRobot framework, this tool was developed in less than 30 minutes.

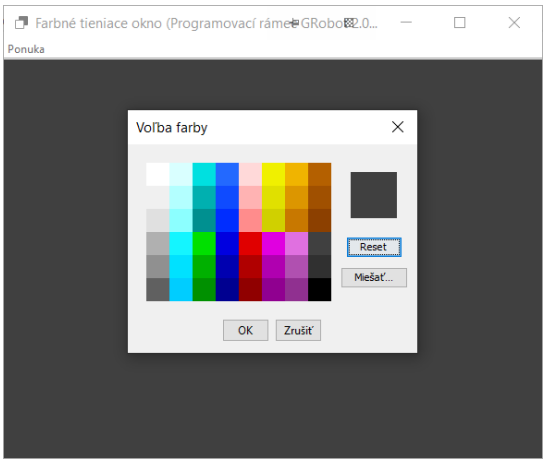


Figure 12. The window after the first launch and the invocation of the dialogue for changing the colour [13].

Following its initial success, the utility was further enhanced to allow customizable colour options, resulting in the “Any Colour Top Window” (Figures 12 and 13). This upgraded version, developed within the next few hours, introduced additional configuration possibilities, improving its versatility. The rapid development and refinement of both utilities exemplify the efficiency and adaptability of the GRobot framework in creating practical solutions for real-time demands.

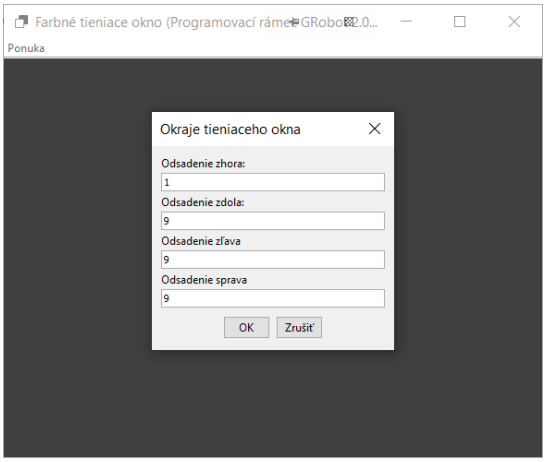


Figure 13. The window after the first launch and the invocation of the dialogue for adjusting the offset of the shading window relative to the main window (which allows for configuring the settings of the shading window) [13].

Small Game with Public Source to Study – Pažravec (Glutton)

The game **Pažravec (Glutton)** presents a graphically pleasing, interactive environment where player controls a yellow, spiked character (the 'Pažravec' – Glutton) with the objective of consuming all the food scattered across a green lawn while avoiding obstacles and enemies represented by amoebas and logs. This game demonstrates core principles of object-oriented design, graphical rendering, and event-driven interaction.

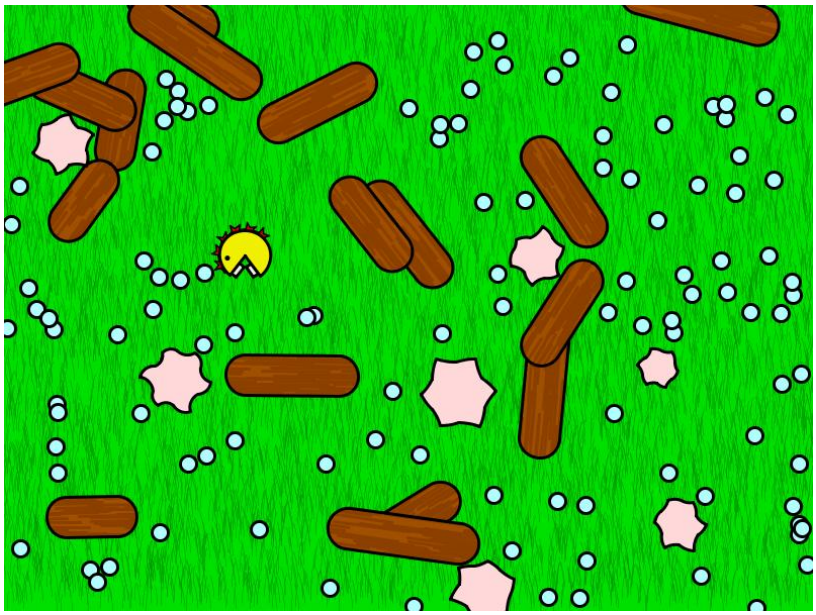


Figure 14. A preview of the appearance of a new game started [14].

The following sections break down the essential components and mechanisms of the game, which's public source code serves as educational tool for understanding graphical programming in Java, particularly in environments suitable for teaching or learning.

Game Overview

Pažravec (Glutton) is an insatiable creature controlled via keyboard and mouse. Its primary task is to consume all visible food on the field while avoiding being consumed by amoebas. The game utilises various input methods, allowing for flexibility: the player can steer the Pažravec (Glutton) using keyboard arrow keys or

drag the mouse cursor to guide the character. Pažravec's (Glutton's) secondary function is to defend itself against the amoebas by shooting spiked projectiles generated from its spiked crest. The number of available spikes is visually represented, and as the Pažravec (Glutton) consumes food, it accelerates spike generation.

Should the Pažravec (Glutton) be consumed by an amoeba, it is regenerated at a random position on the field, while the difficulty increases by spawning an additional amoeba. The game terminates in defeat if the amoebas overpower the Pažravec (Glutton) or in victory if all food is consumed.

Object-Oriented Design and Classes

The game's source code is structured around several classes, each handling distinct aspects of gameplay. This modular design provides a clear view of object-oriented principles and how different objects interact within the game world.

- **'Hlavná' (Main):** This serves as the primary controller for the game logic. It manages the field layout and initialises the Pažravec (Glutton), food ('Krmivo' – Food), enemies ('Príšera' – Monster), and obstacles ('Stena' – Wall). The class also defines methods to reset the game state, spawn new food and enemies, and monitor interactions between the Pažravec (Glutton) and other entities. This centralisation of control makes it an excellent educational example of how to handle game flow and logic.
- **'Pažravec' (Glutton):** The main character – Pažravec (Glutton) – is an instance of this class. It handles player input, movement, and interaction with the game environment. Additionally, the class is responsible for the generation and control of projectiles ('Pichliač' – SpikeyProjectile), and it defines the graphical rendering of the Pažravec's (Glutton's) appearance, including the open-and-close animation of its mouth as it moves. This class illustrates how to extend graphical rendering capabilities in a game.
- **'Príšera' (Monster):** The enemies, represented by amoebas, are instances of this class. Each amoeba is autonomous and moves within the game world based on randomised behaviour patterns. The 'Príšera' (Monster) class manages their size, position, and interactions with both the Pažravec (Glutton) and other obstacles.
- **'Krmivo' (Food):** Food is an essential part of the gameplay, and its visibility and position are managed by this class. Food objects spawn randomly across the field, and their consumption by the Pažravec (Glutton) triggers corresponding changes in game logic.
- **'Stena' (Wall):** Walls, actually wooden logs, act as barriers scattered around the field. They are designed to obstruct movement and create a strategic challenge for the player. The Pažravec (Glutton) can destroy the logs, but only after eliminating all enemies, adding a layer of complexity to the gameplay. (This ability was added due to a small chance that some food occurs inside an inaccessible space circled by the logs.)
- **'Pichliač' (**SpikeyProjectile**):** This class defines the spiked projectiles that the Pažravec (Glutton) shoots to defend itself against the amoebas. Each projectile is created with a limited lifespan (causing limited projectile range) and interacts with enemies and obstacles within the game world.
- **'Pozadie' (Background):** The graphical background, a grass-like texture, is dynamically generated using this class. The background contributes to the game's visual appeal while ensuring that the game world is vibrant and engaging.

Game Mechanics and Interactions

The game operates on a **tick-based system,** where each *tick* advances the game state. The 'Hlavná' (Main) class manages the periodic updates, checking for collisions, food consumption, and enemy interactions. When the Pažravec (Glutton) consumes food, the game updates the status of food items and checks whether the game is complete. Similarly, interactions between the Pažravec (Glutton) and amoebas, or between the Pažravec (Glutton) and walls, are evaluated at each tick.

Projectile generation is a crucial defensive mechanism for the Pažravec (Glutton). The game tracks the number of available projectiles, allowing the player to use them strategically. Projectiles can destroy both enemies and obstacles, but only under specific conditions, as outlined by the game's rules.

Educational Value

The game Pažravec (Glutton) serves as a didactic tool by demonstrating essential programming concepts in a practical, hands-on way. Students learning object-oriented programming can examine how inheritance, encapsulation, and polymorphism are applied. In particular, the game's use of graphical programming demonstrates how graphical objects can be controlled and rendered in real time, making it an excellent case study for understanding user interfaces, animations, and game design principles.

The public source nature of the game further encourages students to experiment with the code, adding new features, altering mechanics, or enhancing the graphical presentation, which deepens their understanding of both programming and game development.

Conclusion

Pažravec (Glutton) is more than just an entertaining game; it is a sophisticated example of how complex programming concepts can be taught through engaging, interactive means. The GRobot framework allows for efficient development, providing an environment where educators can teach key concepts of graphical programming, object-oriented design, and event-driven interaction.

Set of Educational Games for Demonstrating Sorting Algorithms

In the academic year 2023/2024, I was a consultant for a bachelor's thesis titled *Interactive Didactic Game for Demonstrating (Sorting) Algorithms,* written by Barbora Hodálová and supervised by prof. Veronika Stoffová. My role was to guide and support the student throughout the development of this project, which aimed to create an educational tool for teaching sorting algorithms in an interactive and engaging manner. This chapter outlines my contributions and the collaborative process involved in bringing this project to fruition.

Project Overview

The thesis focused on developing interactive educational games designed to help students understand and practice various sorting algorithms. The games aimed to make learning these algorithms more accessible and enjoyable, addressing the challenges students often face when dealing with abstract and complex concepts in computer science.

Consultation and Guidance

As the consultant, my primary responsibility was to provide expert advice on the theoretical and practical aspects of the project. This involved several key areas:

- **Initial Concept Development:** I assisted Barbora in refining the initial concept of the games. We discussed the educational goals, target audience, and the types of sorting algorithms to be included. Our aim was to ensure the games would be both pedagogically valuable and technically feasible.
- **Algorithm Selection and Implementation:** One of the crucial aspects of the games was the accurate implementation of various sorting algorithms, such as bubble sort, insertion sort, selection sort, and planned merge sort. Each of the games focused on a single algorithm. The merge sort was not completed due to time constraints. I provided detailed explanations and pseudocode for these algorithms, helping Barbora understand their underlying principles and how to translate them into interactive game mechanics.
- **Educational Design Principles:** To maximize the educational impact of the games, we incorporated several pedagogical principles. I advised on how to structure the games levels to gradually increase in difficulty, how to provide feedback to students, and ensure the games remained engaging while educational. We also discussed the importance of visual and interactive elements to help students visualize the sorting process.
- **Technical Support:** Throughout the development process, I provided technical support and troubleshooting assistance. This included reviewing code, suggesting optimization techniques, and helping to integrate various functionalities into the games. My experience in programming and software development was instrumental in overcoming technical challenges and ensuring the smooth progression of the project.
- **Testing and Evaluation:** To ensure the games' effectiveness, we conducted several rounds of testing with students from the intended audience. I helped design the evaluation criteria and analyse the feedback to make necessary adjustments. This iterative process played a key role in improving the games and boosting their educational value.

Outcomes and Achievements

The final product was a comprehensive educational tool that successfully demonstrated sorting algorithms in an interactive and user-friendly manner. The games included multiple levels of difficulty, real-time feedback, and a variety of sorting scenarios that helped students grasp the concepts more effectively.

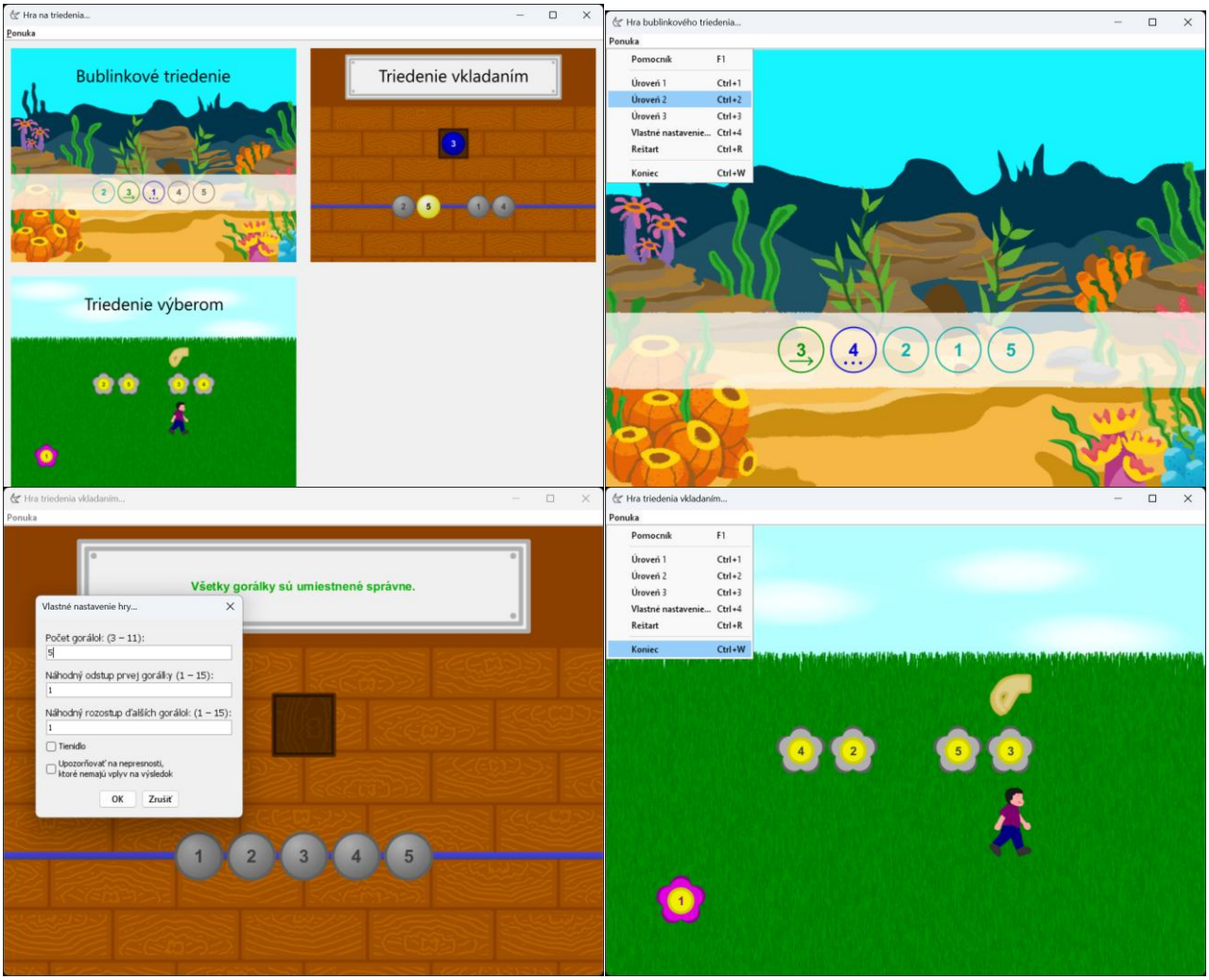


Figure 15. Central screen (top left) and individual games (bubble sort, insertion sort, and selection sort).

Barbora's thesis was well-received, demonstrating a robust understanding of educational theories and technical implementation. The project not only fulfilled its academic requirements but also contributed a valuable resource for teaching computer science.

Conclusion

My involvement in this bachelor's thesis project was a rewarding experience, allowing me to apply my expertise in a meaningful way and contribute to the development of an innovative educational tool. The collaboration with Barbora Hodálová was productive and inspiring for both of us, showcasing how the fusion of pedagogical insights and technical skills can lead to powerful learning experiences.

This chapter highlighted the importance of mentorship and collaboration in academic projects, emphasizing how expert guidance can significantly enhance the quality and effectiveness of student work. As I advance in my academic career, I look forward to more opportunities to mentor and support students in their academic journeys.

Mandalarian

In 2023/2024, I supervised another bachelor's thesis, *A Programming Guide for Creation Chosen Mini Application as Part of a Bigger Educational Content,* by Jarmila Peldová. This chapter presents an overview of the thesis where the student applied the GRobot programming framework. It is crucial to emphasize that the student focused on creating educational materials based on a **pre-existing programming template I provided** rather than developing a novel application from scratch. This work aimed to create a structured guide for programming a small application aligned with the educational content already developed for other students learning programming. The student meticulously documented her process, culminating in a comprehensive educational resource that not only offers step-by-step instructions but also provides alternative solutions to specific challenges.

Assignment Overview

The student, Jarmila Peldová, was tasked with selecting a mini-application to implement using the GRobot framework. The selected mini-application would be developed as part of an interactive educational package, integrating lessons and principles already familiar to students through previously developed content. The student's progress was carefully documented throughout, resulting in a teaching guide on implementing the mini-application. The guide included chapters on programming principles, goals, summaries, questions, and even a glossary. It also proposed alternative ways to approach specific tasks, thereby offering flexibility and enhancing the learning process for future students.

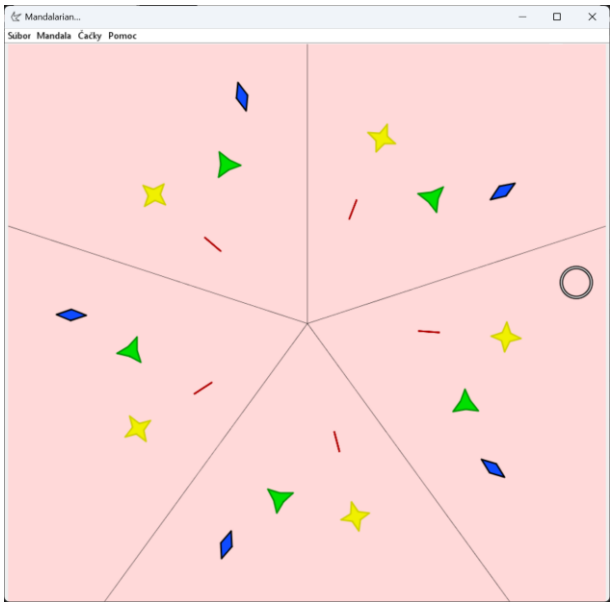


Figure 16. A view of the appearance of the Mandalarian's environment and work in it.

Jarmila's bachelor's thesis serves as a guide for programming the selected mini-application, contributing to a broader educational initiative aimed at teaching programming skills. This project aimed to create an effective and interactive method for introducing students to the world of programming as part of their school curriculum. In today's digital age, programming is an indispensable skill, and this work provides a beginner-friendly guide that focuses on teaching the fundamentals of coding while helping students develop additional skills such as design and data management.

By using practical examples and real-world resources, the guide is structured for students new to programming. Step by step, it explores the basics of coding, offering clear and straightforward instructions on how to create simple applications. This educational material is not only useful in the school environment but also provides students with a skill set beneficial for their future careers.

Because the ability to code is still important, by creating a programming guide for the chosen mini-application, students are given the opportunity to gain hands-on experience and improve their coding skills. This guide aids students in solving problems, encourages critical thinking, and introduces them to various programming concepts.

The focus of this project was to guide the development of a mini-application using Java, a language known for its versatility and cross-platform compatibility. The student structured the project into well-defined components, creating a clear blueprint for other students to follow when developing similar applications. Along with practical instructions, the student ensured the material was engaging, encouraging problem-solving creativity and promoting interest in IT disciplines.

Educational Framework and Student Role

The student's role was primarily to create educational materials using the framework developed for years. This meant working within the confines of the GRobot framework, adhering to the design patterns and previously developed templates. The educational material produced includes instructions for building the mini-application while presenting various potential solutions to common challenges, thus enhancing the learning process for other students.

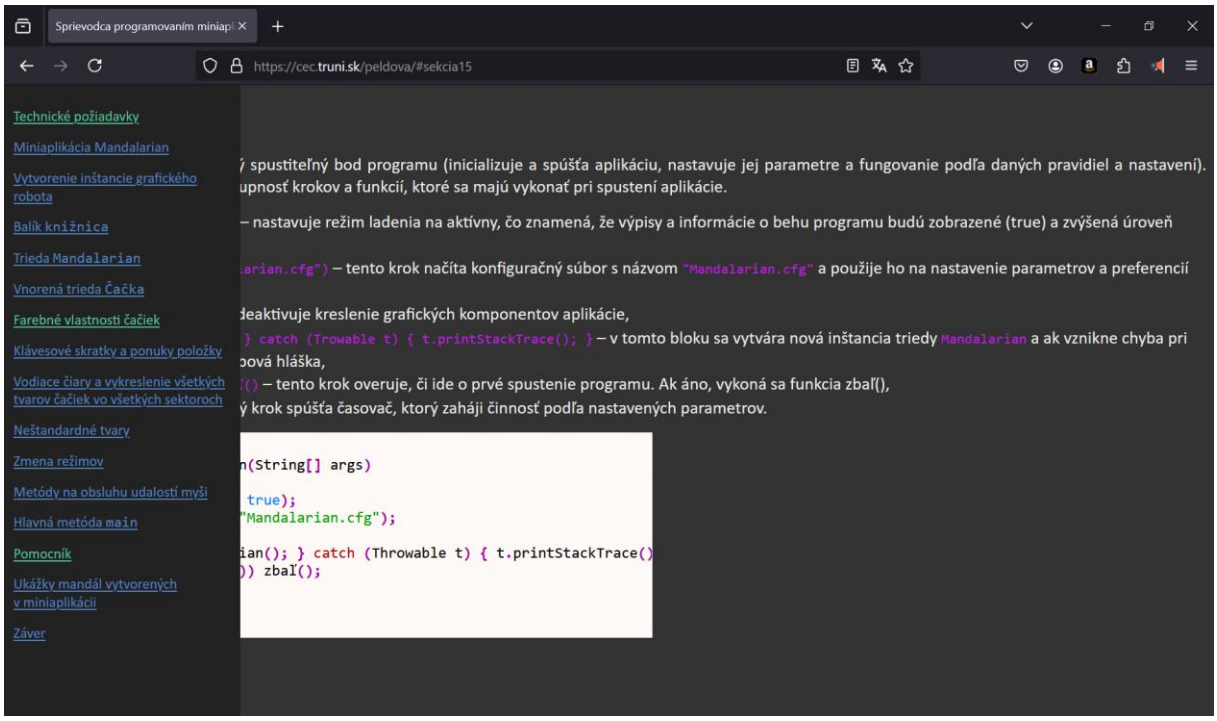


Figure 17. A preview of the resulting educational material.

The project also highlighted the importance of documenting alternative approaches. By offering different ways to solve programming problems, the material should promote creativity and critical thinking in students who use it as a learning resource. The student effectively developed a modular and adaptable guide, offering a solid foundation for future programming exercises.

Conclusion

This chapter illustrates how systematic work and the careful documentation of the programming process can provide an invaluable learning resource for students. Through her work, the student developed educational content that introduces the fundamental principles of programming and encourages experimentation and problem-solving. By using the GRobot framework and following the template provided, the student successfully created a guide that supports learning and creativity in a structured, supportive manner.

The completed guide, while primarily targeting beginners, is also useful for more advanced students, thanks to its comprehensive coverage of alternative methods and solutions. The result is a robust teaching aid that complements the GRobot framework and reinforces key programming concepts in a practical, user-friendly way.

Queue System Simulator

In the year 2021, the initiation of a specialized queueing system took place, targeting its application primarily for pedagogical aims in courses focusing on modelling and simulation [23]. The Queue System Simulator is among the larger ongoing projects. It is a software tool that allows users to configure various queue systems within its interface visually. The overarching goal centred around creating a conducive environment for students and educators to engage in efficient simulations and analyses, particularly in logistics and production contexts.

I began addressing the properties of this system in an article [24]. The description will be slightly expanded here since it could not be done in the original article. Although the initial conceptualization of the system was rooted in educational contexts, there is an ongoing consideration regarding its versatility, speculating that its utility might be adaptable for applications that transcend the educational realm. The lines can currently be one of seven types – each type serves a different purpose; each has different settings. Except that, the lines offer three customer modes for selecting the next line, variable customer lists (for improved tracking), multiple visual settings, and more. The system, as a whole, supports smooth simulation launching, stepping, toggling display of debugging information, and other features.

Background and Literature Review

While conducting a literature review on the subject matter in Slovak and English, distinct presentation styles were observed even though the fundamental content appeared similar. This observation led me to surmise that the field has historically evolved somewhat independently in the context of the former Czechoslovakia (or so-called “easter block”) compared to the “western world.” The phrase, *“When two do the same thing, it is not the same thing,”* appeared more than apt. Faced with the challenge of developing a straightforward approach for implementing the internal simulation engine, I found that none of the methodologies outlined in the consulted resources [25, 26, 27, 28, 29, 30, 31] met the criteria for elemental simplicity. Consequently, I opted to “invent” my own way of implementation.

Had this manuscript been penned a year prior, my observations would have remained largely intuitive—a mere “feeling.” However, the availability of advanced tools has since altered my investigative landscape. To substantiate my background research, I solicited the assistance of two artificial intelligence systems—ChatGPT and Bard. Even though I was initially sceptical and anticipated “excuses” for the inability to generate relevant data, an impression based on recent experiences with artificial intelligence, both systems proved

unexpectedly competent. Intriguingly, the insights they provided were strikingly parallel, defying prior expectations.

In a comparative examination of methodological approaches between the former Czechoslovakia and Western contexts, distinct predilections emerged. Within the scientific community of the former Czechoslovakia, there was a strong inclination towards utilizing mathematical methods like analytical probability theory and stochastic differential equations. This emphasis on mathematical rigour suggests that the focus was often tilted towards theoretical frameworks. It should be noted that these methodologies were considered novel and complex during the period of their application. Conversely, in Western academia, research leaned towards applied facets of study. Statistical methods such as Monte Carlo simulation and Markov chains were commonly employed. These methods were relatively well developed and more easily comprehensible during the same timeframe.

Furthermore, the analysis extends to the broader contextual variables that influenced these divergent methodological tendencies. For instance, the distinct economic landscapes prevailing in the respective countries during that era shaped the focus on specific types of problems warranting solutions. Another influential factor was the disparity in access to publication resources available to researchers in these settings. Additionally, terminological variances emerged as a direct consequence of the research communities' linguistic backgrounds, reinforcing the observed methodological differences.

The Core of the Simulation Engine

Central to the system's architecture is a set of 13 foundational classes. Beyond these classes, the system is integrated with the GRobot framework, a considerably more intricate entity. I opt not to delve into the intricacies of the GRobot framework in this monograph as it is exhaustively documented, partially in my former publications [1, 5, 6, 7, 8, 9, 10, 20, 21, 22] and in more detail in the framework's official documentation [2, 3, 4]. Among the core set of 13 classes, the most pivotal are termed 'Linka' (representing the production line), 'Zákazník' (equivalent to the customer), and 'Systém' (symbolizing the simulation system). Given that the original nomenclature of these classes and their respective elements are in Slovak, they will be presented in their native terminology, accompanied by parenthetical translations.

In the system's architectural design, particular emphasis is given to the 'Linka' (Link) class, which is conceptualized to model a single production line or link within a broader logistics framework (or production system). Within the construct of the simulation, each line signifies an individual stopping point or stage. Interconnections between multiple 'Linka' instances are permissible, facilitating a complex, multi-node simulation. Conceptually, the 'Zákazník' (Customer) within this setting can be interpreted as goods processed during production, subject to varying time constraints for completion (this is referred to as "the customer is served"). This could manifest as a conveyor belt, a queue of products awaiting processing, or even a robotic station designed to execute specific tasks. Functionally, the 'Linka' class inherits behaviours from the 'Činnosť' (Activity) interface. This interface serves as a common denominator across multiple classes within the system, underpinning functionalities related to mass processing. The key attributes of the 'Linka' class include:

- Visual adaptability – among the key attributes of this class is its capacity for visual adaptability because visual representation is essential with this type of software (i.e., simulation software). It enables dynamic graphical representation through method implementations such as 'zmeňNaElipsu' (changeToEllipse) and 'zmeňNaObdĺžnik' (changeToRectangle), which is helpful for intuitive

visualization of the simulated elements. Further, methods like ‘spojnica’ (connector), ‘zrušSpojnicu’ (removeConnector), and ‘aktualizujSpojnice’ (updateConnectors) are instrumental in visualizing the interconnectivity of the ‘Linka’ instances. These methods collectively enable a flexible and dynamic representation of the production line. Additionally, functionalities like ‘kopíruj’ (copy) and ‘blikni’ (blink) are incorporated, allowing the class to replicate the current visual state of the current link instance to a specified another instance and provide visual feedback, such as blinking, to signal status changes.

- Mode and type flexibility – another integral attribute of the ‘Linka’ (Link) class centres on its mode and type flexibility. The class is engineered to support a multitude of operational ‘modes’ or ‘states,’ notably ‘režimVýberuZákazníkov’ (customersSelectionMode) and ‘režimVýberuLiniek’ (linesSelectionMode). These functionalities render the class highly adaptable to diverse situational requirements and scenarios. Furthermore, methods prefixed with ‘zmeňNa…’ (changeTo…), such as ‘zmeňNaEmitor’ (changeToEmitter) or ‘zmeňNaZásobník’ (changeToStorageBin), facilitate dynamic role alterations for the link within the broader production system. For example, it can transform into an emitter, serving as a customer source, or metamorphose into a storage bin with a defined capacity, capable of storing customers for an indefinite duration.

The class designated as ‘Zákazník’ (Customer) serves as a representative entity for a customer or an item within a logistics or production line system. It is architecturally designed to include several key features that enable rich functionality. These features are facilitated through a diverse array of methods, which allow easy access to the class’s core capabilities, as elaborated below:

- Object pooling – the class incorporates static methods such as ‘nový’ (new), ‘počet’ (count), and ‘daj’ (get) that are integral to its object pooling mechanism. This allows for efficient management of instances while also optimizing memory usage. An instance method, termed ‘reset,’ is specifically designed to consistently initialize and reinitialize objects within this pooling shell.
- Visualization – methods such as ‘upravCieľPodľaLinky’ (adjustTargetByLink) are geared towards fine-tuning the visual aspects of the simulation. They ensure that a customer’s visual representation stays congruent with its link (or production line) internal states, thereby accurately reflecting the internal state of the whole simulation. Event handlers like ‘dosiahnutieCieľa’ (targetAcquired) serve to animate instances that have reached specific target areas, typically representing their destination production line.
- Naming – the methods ‘meno’ (short for getName) and ‘pomenuj’ (setName) facilitate the retrieval and assignment of names to instances of customers. While naming may be optional, depending on the specific use case, it can be pivotal for differentiating between otherwise indistinguishable objects. For example, in scenarios requiring the distribution of students into various groups (which can be one of the use cases of the system), naming becomes not just useful but essential.
- Time management – the class incorporates methods like ‘pridajInterval’ (addInterval), ‘nastavInterval’ (setInterval), and ‘čas’ (time) that act upon the internal clock of each customer instance. This clock is a critical element in governing the remaining time for a state change for a customer. For instance, it can dictate how long a customer remains at a specific stage of a production line.

- Link association – methods such as ‘priraďKLinke’ (attachToLink) and ‘vyraďZLinky’ (detachOfLink) enable the attachment or detachment of the customer/item to or from a specific link within the system. These functionalities underscore the class’s role in representing the customer or item in a complex, interconnected production system.

In the architectural schema, the class designated as ‘Systém’ (System) acts as the centralized hub that coordinates the workings of the entire simulation milieu. This class serves a multi-faceted role as it interferes with both the ‘Linka’ (Link) and ‘Zákazník’ (Customer) multiple instances while also integrating the functionalities offered by the remaining ten classes within the project. Its purview is expansive, serving to:

- Coordinate the presentation: the class bears responsibility for dictating the visibility and graphical representation of simulation entities.
- Lifecycle management: it oversees the entire lifecycle of each object within the simulation, right from instantiation to termination.
- Simulation control: ‘Systém’ encompasses features for pausing and resuming the simulation, offering granularity in control during different phases of the simulation process.
- User interface mediation: the class serves as the intermediary layer between the user interface and the underlying simulation model.
- Data processing utilities: it also encapsulates tools specifically designed for data manipulation and processing, enhancing the system’s utility for analytics.

The features of the ‘Systém’ (System) class also offer the following distinct functionalities. Visualization tools:

- Information display – methods like ‘zobrazInformácie’ (displayInfo) and ‘prepniZobrazenieInformácií’ (toggleInfo) are designed to control the visibility of information at a system level. These prove invaluable for debugging or performance scrutiny.
- Grid layout management – methods such as ‘mriežkaX’ (gridX) and ‘mriežkaY’ (gridY) allow the use of grid-based positioning that eases setting the spatial layout of production lines. It’s worth noting that this grid system has no influence on customer objects.

Simulation control tools:

- Pause and resume – methods like ‘repauzuj’ (repause – a neologism derived from the word pause the same way as the word reset is derived from the word set) and ‘pauza’ (pause) facilitate pausing and resuming of the simulation. These contribute to granular control over the simulation process.
- Timer and events – methods like ‘setTimer’ (exceptionally, some methods are named in English), ‘tik’ (tick), ‘klik’ (click) cater to time-sensitive functionalities and user events, further enriching simulation control.

Data management tools:

- System operations – methods such as ‘newSystem’ (again, some methods are named in English), ‘openSystem’, ‘saveSystem’, ‘undo’, ‘redo’ provide the capability to manage the state of the entire system. They permit operations like creating, opening, saving, and altering system states.
- Selection management – methods like ‘selectAll’, ‘deselectAll’, ‘selectNext’, ‘selectPrevious’ offer advanced interactive selection capabilities, possibly targeting ‘Linka’ (Link) and ‘Zákazník’ (Customer) objects (even though the user cannot select single customers through the user interface).
- Connector management – with ‘newConnector’, ‘deleteConnectors’, ‘jeZačiatokKonektora’ (isStartOfConnector), the class has the ability to manage the link connectors, thereby defining the interplay between different production links in the current simulation configuration.

In sum, the ‘Systém’ class functions as the veritable “control room” of the simulated environment. It consolidates all disparate elements, offering a centralized interface for simulating, managing, and interacting with logistics or production line systems and emerges as an all-encompassing hub that encapsulates an array of functionalities, from visualization and simulation control to data management, thereby offering a structured yet dynamic simulation environment. The result is a structured yet agile platform where objects and entities can seamlessly interact, operate, and comply with predefined rules and conditions.

Features and Functionalities

When establishing a new line in the simulation environment, users have the option to input its name, signifying the first step in a richly customizable process. Following this:

- Line Typology: Users designate the functional type of the line, whether it is an emitter, a conveyor, a processor, or what is termed a “releaser,” which means the consumer entity in this simulation engine. Details on this are given later.
- Parameter Configuration: Users can tailor specific line attributes based on the typology. For example:
    - Quantitative parameters: Timer configurations like processing time, generation times, or other telling intervals – the name is contextual according to the line type, line capacity and customer selection modes or other specific parameters depending on the line type.
    - Visual Adjustments: Parameters like dimension, rotation, and shape are modifiable, allowing the line to be visually coherent with its function.
    - Next Line Selection Mode: The modality for customers exiting the line can also be configured, offering a higher level of dynamism in line-to-line transitions.

Throughout this phase, one can establish or dismantle connections between lines, dictating the paths along which customers will be allowed to traverse. (It is worth noting that the transfer of customers is only permissible between connected lines.)

The emitter lines act as the genesis points for customers. These emitters have the capability to generate both named and unnamed customers. Moreover, the system allows for cyclic rotation of names. However, certain limitations exist:

- Name Shuffling: If a specific arrangement of customer names is required, such as randomization, the existing system default behaviour may fall short. One workaround could be generating all customers upfront, relegating them to a "waiting room," and subsequently applying a randomized customer selection mode.
- Event Scripting: Currently, the framework does not extend support for event scripting, constraining the level of automation that can be achieved in the simulation events.

The simulation system currently recognizes seven distinct types of lines: emitter, storage bin, waiting room, halting room, conveyor, converter, and releaser. While theory suggests that the number of line types could be minimized significantly because the fundamental principles in different line types are the same and can be distinguished just parametrically, it was decided, for simplicity and user comfort, to implement multiple line types, each with its pre-programmed behaviour. The user can change the line type at any point.

Among types of lines, two stand out for their fundamental roles: **emitter,** responsible for generating customers entering the simulation, and **releaser,** which acts as the endpoint for the customers, correctly releasing those who have been served properly. It is worth noting that an "incorrect release" occurs when a customer attempts to exit a line but finds no connected output lines ready for further processing. These customers are marked as "unserved," which is logged in the internal report available for further processing.

Four support line types help prevent incorrect releases: The **storage bin** serves as a simple holding area until the next connected line becomes available. **The waiting room** is similar to the storage bin but assigns a time interval during which the customer is willing to wait. The customer proceeds (leaves the room) if the next line becomes available before this interval expires. **The halting room** also assigns a time interval but holds customers even if a connected line becomes available. The three types of lines have limited capacity. Once a line reaches its maximum capacity, it becomes unavailable until the number of customers inside is reduced. **The conveyor** operates similarly to the halting room but differs in its unlimited capacity and in visually indicating the movement of customers through the line.

The final category in the taxonomy of line types is the **converter,** which serves as a versatile unit capable of customer service or product processing. Figure 18 illustrates a simulation chain of lines in both inactive and active states, showcasing an emitter, a conveyor, a halting room, and a releaser. All parameters of the lines are set to default. Interestingly, the halting room in this example stands in for a machine, which is not its usual role. Usually, the converter fulfils this task.

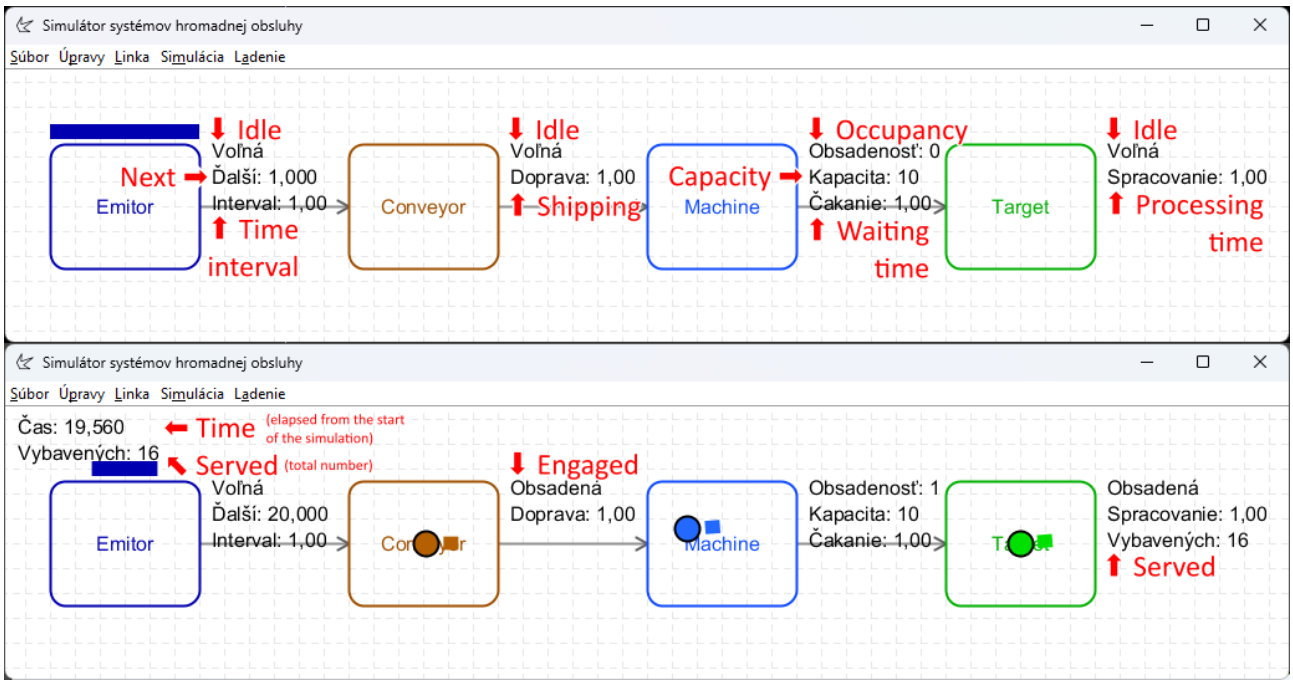


Figure 18. Unassuming chain of production lines. Displays the simulation in both inactive and active state. (Translations of selected Slovak texts are provided within the image.)

Typically, the converter is employed for machine-like tasks. The key distinction between the halting room and the converter lies in capacity constraints. The halting room has a defined capacity, while the converter does not have one, which means that it can handle a single customer at a time. According to theory, all types of lines can be reduced to just three or four types. However, I implemented more types to simplify the simulation setup, as many existing line types can be used interchangeably. If it better fits the simulation's needs, it is suggested that any type of lines be used in any role.

Further enriching the simulation capabilities, I've also explored various randomisation methods. Based on [32], I have implemented a pseudo-random number generator customized for my simulation needs. This feature adds another layer of realism and unpredictability, enhancing the simulation's robustness.

Visual and User Experience Aspects

Line configuration parameters are, in the main, dictated by the type of line under consideration. Figure 19 shows a side-by-side comparison of dialogues for setting the basic properties of different line types. On the far left is the most general case—a dialogue box for a line with no designated purpose. The central dialogue pertains to the waiting room, while the dialogues on the right display settings for an emitter and a releaser, stacked one above the other. Figure 20 focuses on the dialogue for adjusting the selection of the next line for exiting customers. This feature complements the customer selection mode, glimpsed at the bottom of the general and waiting room dialogues in Figure 19.

Note: Texts on images 19–21 are not translated (in this monograph) because, from my point of view, they show some less critical aspects of the simulation system.

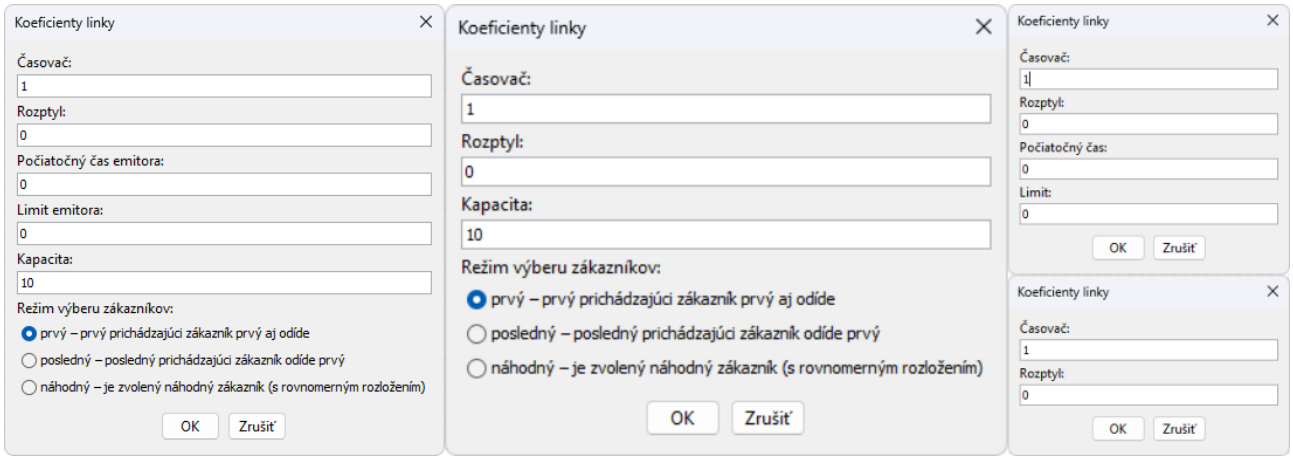


Figure 19. Dialogues for setting the basic properties of lines.
(Strings translations are in the text that follows.)

Because Figures 19 and 20 lack the space for a direct translation of the interface strings on the picture, this section aims to offer clarifications. For this text, translations are primarily pulled from the general variant of the dialogue in Figure 19. This is because, across different line types, the strings generally retain their meaning, undergoing at most minor shifts in nuance.

Core Parameters:

- *'Časovač'* – timer – this parameter dictates the operating time intervals for different line types. Its semantic meaning varies: for an emitter, it sets the customer generation interval; for a conveyor, it denotes transport time; and for a releaser, it is not applicable.
- *'Rozptyl'* – variance – this parameter decides the random fluctuation of timer values. It is useful for adding randomness to the simulation. More about it later. Only a uniform distribution method is currently implemented to generate these random values.
- *'Počiatočný čas (emitora)'* – (emitter) initial time – this parameter is unique to the emitter and sets an initial delay before customer generation begins.
- *'Limit (emitora)'* – (emitter) limit – this is another emitter-specific parameter. It designates the maximum number of customers an emitter can generate before becoming "exhausted."
- *'Kapacita'* – capacity – this parameter determines the maximum number of customers that can be stored at one time in the line. This parameter is available for lines like storage bins, waiting rooms, and halting rooms.
- *'Režim výberu zákazníkov'* – customer selection mode – three customer selection modes are available:
  - *'prvý […]'* – first – the first customer to arrive is the first to leave;
  - *'posledný […]'* – last – the last customer to arrive leaves first;
  - *'náhodný […]'* – random – a customer is chosen at random, based on a uniform distribution.

Variance means the random spread of the timer values in both directions where the timer value serves as the central point value. This approach was chosen due to a more straightforward interpretation than entering an interval in the form of parameters representing the lower and upper bounds and due to the possibility of implementing more distribution methods of pseudo-random values in the future.

The emitter limit can be utilized to simulate “waves” of customers by setting different time delays (emitter initial times) and limits for multiple emitters connected to the same line, through which the customers will continue further into the system. (Until scripting is available, various situations will depend on user creativity.)

The user interface is vital in configuring line parameters and setting customer transition logic between connected lines. While I won’t delve into visual elements like figures, I will describe the dialogue interfaces that help to configure these essential functionalities.

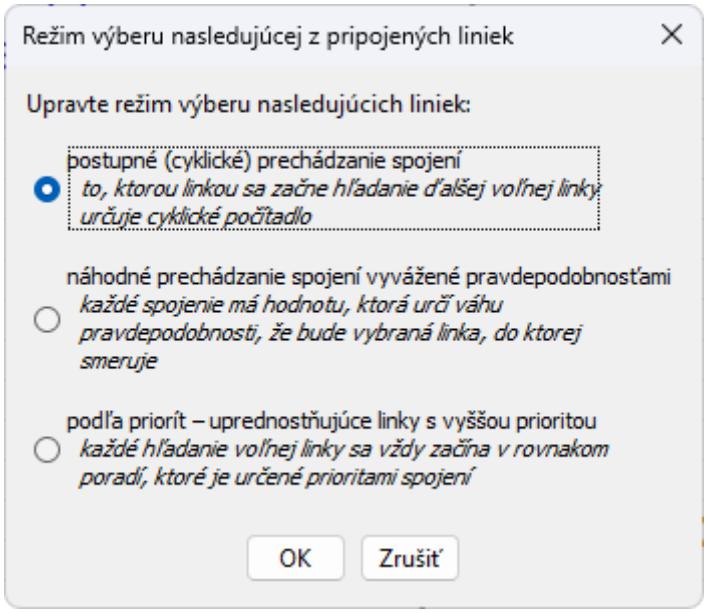


Figure 20. Dialogue for selecting the mode of the next line choice from the connected ones. (Strings translations are in the text that follows.)

Line selection modes dialogue in Figure 20 shows the following options:

- *‘postupné (cyklické) prechádzanie spojení […]’* – sequential (cyclic) switching of connectors: In this mode, a cyclic counter controls the starting point for searching the next available line for outgoing customers. This cyclic nature increases the likelihood of rotating through different lines. (It is worth noting that the start of the search for idle line shifts each time. This significantly increases the probability that the lines would cycle evenly.)
- *‘náhodné prechádzanie spojení vyvážené pravdepodobnosťami […]’* – random traversal of links balanced by probabilities: Here, each connector has a probability weight, dictating how likely it is that the connected line will be chosen as the next destination for the customer.
- *‘podľa priorít – uprednostňujúce linky s vyššou prioritou […]’* – according to priorities – giving priority to lines with a higher priority (weight): In this setting, priority is given to lines with a higher weight. The search for a free line commences in the same order every time, guided by these pre-set priorities.

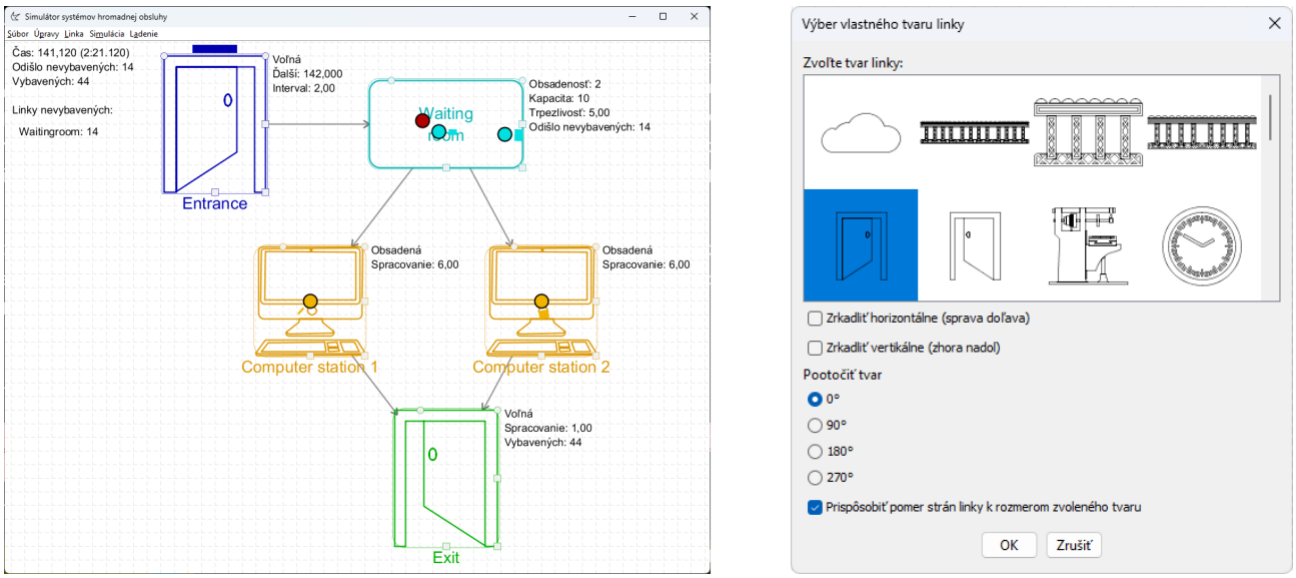


Figure 21. An example of a slightly more complex network of production lines (left) and dialogue of production line custom shape selection (right).

In Figure 21, we see an example system that might represent a computer cafe. The appearance of the lines can be adjusted via this dialogue. Even though the visual setting is often a primary concern for users, I consider it marginal from a data-processing standpoint.

Future Work

While the system is essentially complete, there are specific areas I am contemplating for further improvement. Let us delve into these aspects. Adding an undo/redo feature is a consideration, although it is primarily for user comfort and not critical to the functioning of the software.

The existing simulation engine operates based on the computer system's nanosecond clock. It has its own timer that measures the nanoseconds passed between application ticks. These nanoseconds are then divided into smaller intervals according to the simulation settings to advance the elements' internal states. The initial assumption was that this method would be universal and not affected by speeding up or slowing down the simulation. However, testing has revealed inconsistencies in simulation results when altering the speed, leading to reconsidering the engine's architecture.

The engine's current configuration stems from not foreseeing the system timer behaviour when combined with the application timer. Specifically, it is susceptible to errors during manipulation of a simulation speed. In an ideal scenario, repeated simulations with precisely set parameters should always yield the same outcome. Currently, this is not the case—results vary when changing the simulation speed. I suppose that refactoring the engine's core will correct these discrepancies. The goal is to develop an engine where changing the speed of time passage does not alter the simulation results. This correction is pivotal and surpasses the priority of additional user interface features like undo/redo.

To explain it in more detail: In our efforts to validate the existing simulation engine, I conducted tests on a straightforward deterministic system. The hypothesis was that if the engine's behaviour remains consistent, then the simulation results at normal speed should align precisely with those obtained at accelerated speeds. Contrary to expectations, the results started to deviate at four times the original speed and continued to differ as the acceleration factor increased. This outcome was far from satisfactory. Upon examining the engine algorithm, no apparent bugs were found within its original design. This led me to

conclude that the issue likely originates from utilising two different timing systems within the engine—the application's own timer and the computer system's nanosecond clock. Given these findings, my future plan aims to eliminate dependency on the system timer. Instead, I will advance the simulation states using a more deterministic approach to ensure consistency in the results across different speeds.

Conclusion

My initial ambition was to craft a system flexible enough for multiple applications, particularly in education. To a significant extent, the system I have developed meets this goal. It provides a comprehensive simulation environment adaptable to intricate logistics or production line scenarios. The system is replete with functionalities that make it user-friendly and technically sound.

It also comes equipped with its own pseudo-random number generator and supports various line types—from emitters to releasers—each with unique behaviour and settings. Although my platform is not nearing professional completion, it is still a work in progress and might be useful for small simulations in the future.

One of the primary focus areas for upcoming development is the internal simulation engine. My testing has exposed inconsistencies, particularly when altering the simulation speed. I have ruled out flaws in the algorithm, suggesting the problem likely lies in using multiple timers. This necessitates a future update to shift towards a more deterministic timing approach. While some features like undo/redo are still on the drawing board, they are aimed mainly at enhancing user comfort and experience. That said, these features are secondary compared to the core issues identified in the simulation engine.

Lessons from Development and Future Directions

The development of the framework and its predecessors overlaps with the writing of detailed documentation. From there came a valuable experience in finding out that writing the documentation can lead to discovering hidden inaccuracies, malfunctions, and bugs in the code itself. Writing documentation can, and often does, serve the same purpose as debugging the code. This experience has been confirmed many times over. Later, I randomly came across the testimonies of other colleagues who had the same experience, proving further that this finding reflects a real phenomenon.

However, the improvements went two ways. It is not just a way of debugging the code but also improving the documentation, not in the sense of adding information about new features. The whole documentation had been continuously revised, for example, in the context of improving the terminology used within it. It is natural that over the years, the framework's author had better understand some terms used in mathematics and programming. There were many findings and improvements [33, 34, 35, 36, 37].

One of the interesting findings was the clarification of the difference between the terms parameter and argument. These two terms were often used interchangeably, and it was not easy to find any references about the differences in later years. The difference is subtle but fundamental. If we look at them from a programming point of view, then a parameter is a quasi- "special kind of variable" whose value is initialised "outside" the method (at the time and place of the method call) and which I use in the body of the method. So, in short, *a parameter is an identifier used in the body of a method.* Its value is initialised at the time the method is called.

If we want to know what the value of the parameter is initialised with, we must understand the term argument. An argument is an expression (i.e., a combination of literals, constants, variables, operators, etc.) that is put at the point of the method call. So, the argument is exactly the expression whose resulting value initialises the parameter. In short, *an argument is an expression listed in the argument list at the point of the method call.* The resulting argument value initialises the corresponding parameter. The following self-explanatory pseudocode statement could be written to explain it in a more practical manner: *«parameter»* = *«argument»*;

Another interesting finding relates to the importance of seemingly subtle aspects such as file extensions. For years, this has been an overlooked issue, considered inconsequential as it does not directly affect the content of a file, and therefore deemed unimportant. However, I eventually realised (fortunately without significant consequences) that it is critical to correct description, identification, and classification. The following example serves to illustrate the significance of this issue.

Imagine a programmer writing code and registering a configuration file at the outset, typically using the same name as the main class. Due to fatigue or a lack of morning coffee, and in an attempt to simplify (or speed up) the process, the programmer copies the class name from the 'Save as…' dialogue and neglects to change the extension from `.java` to `.cfg`. Without noticing the error, they run the program. Upon exiting, they are astonished to discover that the source code of the class has been discarded, and the configuration data has been written over it. If this occurs at the start of developing the class, the loss may be minor. If the source

code was backed up, the damage could be negligible. However, by enforcing the correct suffix, such risks of data loss can be entirely avoided.

After recognising this, I implemented the following rule: when saving standard files, such as configuration files, logs, and similar types with a recognised extension, always ensure that the correct extension is appended to the filename entered by the user. If absent, it must be added automatically. This serves as an essential safety measure.

Various implementations of this rule can be observed within the framework's source code. In cases where multiple extensions are permissible (e.g., when saving SVG data, either `.svg` or `.html` may be used), a check is conducted to ensure that the filename includes one of the allowed extensions. If it does not, the file will not be saved.

The visual principle on which the framework is based also brought in another dimension of debugging. The typical JUnit tests cannot be written for a program whose output is purely visual. Such a result is best to be judged and assessed visually because it also involves imagination and logic. It means the same for some attempts to assess the assignments automatically. Unless you have some sufficiently advanced artificial intelligence, the automated assessment is very difficult and, in most cases, practically impossible.

In this context, logging becomes an indispensable tool for tracking the internal workings of the framework, especially when visual outputs alone are insufficient to identify or solve issues. This is where the `knižnica.log.Log` class proves to be highly useful. Initially designed for a different project (SHO – the Queue System Simulator), it was adapted into the framework to streamline the creation of debugging outputs (logs) without requiring the extensive overhead of a full debugging process. The class consists entirely of static elements, making it lightweight and easy to integrate into various application parts. It enables developers to insert logs at critical points within their code, capturing method entries, parameter values, return values, and even stack trace elements. This is particularly helpful when diagnosing complex issues related to state changes, sequence errors, or performance bottlenecks.

By using the `logIn()` and `logOut()` methods, programmers can precisely track the flow of execution through their methods, and by customising the logging output, it becomes possible to gather all the essential information needed for effective debugging. While visual debugging is critical for evaluating output, logging enhances the process by providing insight into the program's execution flow, offering a more complete approach to debugging.

Plans for Improvements

There is a further development plan that contains a few items, and some have been there for a long time. Here are the most interesting items (from oldest planned to newer):

- the possibility of working with kinematic models;
- the possibility of creating structural graphs with options for breadth-first search (BFS) and depth-first search (DFS);
- the possibility of working with Fourier transformation to decompose a (sound) signal into the frequencies, possibly also adding the frequencies to create a (sound) signal;

- the possibility of working with neural networks, including the possibilities of their visualization.

Paradoxically, the earlier the item was included in the plan, the lower the priority for its implementation today. Therefore, the current situation is that the work on the last item has already started; on the other hand, the first item will likely never be included in the framework. Not that it did not make sense. In fact, each new feature of the framework enriches it as a whole because the functionality is mostly connectible with the other framework components. This means that implementing the kinematic model could be connected to a neural network model to visualise it and/or explore its interrelationships and regularities better. In essence, the constraints are driven by the limitation of available time.

Known Issues and Limitations

One of the known bugs of the programming framework is hidden in the tone generation subsystem. In the course of its refinement, an unknown bug was introduced into the system, and a system with a strictly deterministic result started to behave chaotically. The bug was introduced during the implementation of the so-called tone generation scheduler (something that could be compared to a "programmatic pre-preparation of a MIDI playback"), and it could not be traced even after a long time. The system is supposed to be deterministic, but from a particular moment of implementation, it is not able to consistently generate the same result within a single application launch, and the results differ even between separate application launches when the system should always be in the same initial state, which defies any logic. The easiest solution would probably be to write the entire system from scratch.

The second known "bug" is not an actual "bug." Let me call it an "inconsistency." It is a situation that results from the programming framework's design. It occurs when the robot moves during initialisation as part of drawing its own shape. The framework supports two different approaches: simple mode for beginners and advanced mode for more complex projects. These two modes are not compatible. In simple mode, it is not recommended to use the custom robot's shape drawing by overriding the 'kresliSeba' (drawSelf) or 'kresliTvar' (literally drawShape, but the final translation will probably be sketchSelf) methods, and in advanced mode, turning off automatic redraw during initialisation is required. If the framework itself would turn off the automatic redrawing during the initialisation, the simple mode would lose its meaning (the drawing must be enabled; otherwise, the pupils would not see the gradual drawing of the robot). Conversely, if automatic redrawing is not turned off during initialisation in advanced mode and the robot moves while drawing its own shape (if it draws more complex shapes), it will collide with simple mode, and the robot may end up in the wrong position after the initialisation.

The third shortcoming is the implementation deficiency of the ExpressionProcessor helper class, which is capable of parsing and interpreting simple mathematical expressions. During the development, the possibility of working with lists of values was incorporated into this class. Still, since this possibility was not implemented from the start and into the original core (but is just something like a "superstructure"), problems connected to interpreting some syntactic constructs arose. The empty list is one of the problems that could not be adequately bypassed and/or resolved. The solution would be to rewrite the class so that lists would become part of its core. (However, this process is hard to imagine since the class lists about 7,200 lines of code, and its development roots go back to 2003.)

Conclusions

The GRobot framework's evolution from a simple graphical library into a comprehensive programming framework demonstrates the path many programmers take during their careers, not once, even several times while doing different projects. In this monograph, there is an effort to show the utility of such tools in both didactic and practical programming contexts. Designed initially to aid in teaching programming concepts through visual feedback and interaction, the framework has matured into a robust platform capable of supporting complex applications beyond its original intent. The gradual expansion of features, from basic turtle graphics to advanced functionalities such as event handling, object-oriented principles, and simulation capabilities, has significantly enriched its educational value.

As with any evolving software framework, continuous refinement and feedback have been essential to its development. The various extensions—whether through enhanced integer manipulation methods or expanded graphical capabilities—broaden the framework's applicability and improve the learning experience for students. Despite the occasional challenges, such as the limitations in speed manipulation or subtle inconsistencies in the tone generation system, the framework's overall architecture has proven flexible and resilient.

Looking ahead, the potential for further improvements—such as integrating neural networks or more advanced visualisation techniques—promises to expand the GRobot framework's utility in both educational and research contexts. However, the limitations of time and resources naturally constrain the pace of these advancements. Nonetheless, the framework, in its current form, remains a valuable tool for introducing students to core programming concepts while also providing opportunities for more advanced exploration in applied computing and software development.

In conclusion, the GRobot framework achieves its didactic aims and presents a flexible, modular foundation for future educational and technical endeavours. Its ongoing development will undoubtedly continue to contribute meaningfully to both programming education and the broader field of software design.

Recommended Reading and References

[1] Horváth, Roman. (2013). *Hľadanie nových postupov vo výučbe programovania na VŠ.* Dizertačná práca. Bratislava : Fakulta matematiky, fyziky a informatiky Univerzity Komenského. 162 s.

[2] Horváth, Roman. (2014). *Dokumentácia skupiny tried grafického robota pre Javu – 2. prepracované vydanie.* Trnava : Faculty of Education, Trnava University in Trnava. [27.67 AH]. ISBN 978-80-8082-796-0. Retrieved from: ⟨http://cec.truni.sk/horvath/GRobot/⟩ (access date: 17/08/2023).

[3] Horváth, Roman. (2012). *Dokumentácia skupiny tried grafického robota pre Javu.* Trnava : Faculty of Education, Trnava University in Trnava. [20.55 AH]. ISBN 978-80-8082-537-9. Retrieved from: ⟨http://cec.truni.sk/horvath/Robot/⟩ (access date: 17/08/2023).

[4] Horváth, Roman. (2023). *Dokumentácia programovacieho rámca GRobot.* Retrieved from: ⟨https://pdfweb.truni.sk/horvath/GRobot/⟩ (access date: 17/08/2023).

[5] Horváth, Roman. (2018). The Past Seven Years of Development of the Framework for Teaching Programming and the Students' Results. In *ICETA 2018.* Danvers : IEEE, pp. 185–189. ISBN 978-1-5386-7912-8.

[6] Horváth, Roman – Javorský, Stanislav. (2014). New Teaching Model for Java Programming Subjects. In *Procedia social and behavioral sciences. Vol. 116(2014).* 5th World Conference on Educational Sciences – WCES 2013 (konferencia), WCES 2013, pp. 5188–5193. ISSN 1877-0428. ⟨https://doi.org/10.1016/j.sbspro.2014.01.1098⟩. Retrieved from: ⟨http://www.sciencedirect.com/science/article/pii/S187704281401115X⟩ (access date: 18/08/2023).

[7] Horváth, Roman. (2012). Porovnanie hodnotenia študentov z predmetu PA1 s bodovaním získaným v prijímacom konaní na PdF TU. In *DidInfo 2012.* Banská Bystrica : Univerzita Mateja Bela, Fakulta prírodných vied, Katedra informatiky, pp. 83–86. ISBN 978-80-557-0342-8.

[8] Horváth, Roman. (2011). Nový model vyučovania programovania na Pedagogickej fakulte TU. In *Acta Facultatis Paedagogicae Universitatis Tyrnaviensis.* Trnava : Trnavská univerzita, Pedagogická fakulta, pp. 98–102. ISBN 978-80-8082-514-0. Retrieved from: ⟨http://pdf.truni.sk/actafp/2011/c/#HORVATH⟩ (access date: 21/08/2023).

[9] Horváth, Roman. (2011). Teaching one Language in More Depth is better than Many Languages Superficially. In *ICETA 2011.* Stará Lesná, The High Tatras, Slovakia : Óbuda University, IEEE, pp. 71–74. ISBN 978-1-4577-0050-7. ISBN 978-1-4577-0051-4. ⟨https://doi.org/10.1109/ICETA.2011.6112588⟩.

[10] Horváth, Roman. (2009). Inovácia vyučovania predmetov programovania na Pedagogickej fakulte Trnavskej univerzity. In *Acta Facultatis Paedagogicae Universitatis Tyrnaviensis.* Trnava : Trnavská univerzita v Trnave, Pedagogická fakulta, pp. 64–68. ISBN 978-80-8082-320-7.

[11] Horváth, Roman. (2023). *Vzdelávacie MiniAplikácie.* Available at: ⟨https://pdfweb.truni.sk/horvath/materialy?miniaplikacie⟩. Last accessed: 5. 3. 2024.

[12] Horváth, Roman. (n.d.). *Biele okno.* Available at: ⟨https://pdfweb.truni.sk/horvath/softver?white--window⟩. Last accessed: 5. 10. 2024.

[13] Horváth, Roman. (n.d.). *Farebné tieniace okno.* Available at: ⟨https://pdfweb.truni.sk/horvath/softver?any-colour-top-window⟩. Last accessed: 5. 10. 2024.

[14] Horváth, Roman. (2020). *Pažravec.* Available at: ⟨https://pdfweb.truni.sk/horvath/materialy?pazravec⟩. Last accessed: 4. 10. 2024.

[15] *Imagine.* (2003). Available at: ⟨https://imagine.input.sk/⟩. Last accessed: 17. 8. 2023.

[16] *JPAZ2 framework.* (2022). Available at: ⟨https://ics.science.upjs.sk/paz1a/wp-content/uploads/sites/6/2022/08/JPAZ2.pdf⟩. Last accessed: 17. 8. 2023.

[17] *JPAZ2 1.1.1 API.* (2017). Available at: ⟨https://paz1a-old.ics.upjs.sk/storage/jpaz/doc/index.html⟩. Last accessed: 17. 8. 2023.

[18] *Educational framework for Java beginners (based on turtle graphics).* (2017). From GitHub – ics-upjs/JPAZ2. Available at: ⟨https://github.com/ics-upjs/JPAZ2⟩. Last accessed: 17. 8. 2023.

[19] Papert, Seymour. (1980). *Mindstorms : Children, computers, and powerful ideas.* New York : Basic Books, Inc. ISBN 978-0-85527-163-3.

[20] Horváth, Roman – Fialová, Jana. (2020). The Creation of Simulation with an Algorithm Optimisation in Java for the Teaching Process. In *ICETA 2020.* Danvers : IEEE, pp. 160–166. ISBN 978-0-7381-2366-0.

[21] Horváth, Roman. (2019). Empirical Study of Minkowski–Bouligand and Hausdorff–Besicovitch Dimensions of Fractal Curves. In *ICETA 2019.* Danvers : IEEE, pp. 231–238. ISBN 978-1-7281-4966-0.

[22] Stoffová, Veronika – Horváth, Roman. (2017). Didactic Computer Games in Teaching and Learning Process. In *eLSE 2017 – eLearning and Software for Education Conference.* Bucharest : Carol I National Defence University Publishing House, pp. 310–319. ISSN 2066-026X, ISSN 2066-026X, ISSN 2343-7669.

[23] Horváth, Roman. (2023). *Simulátor systémov hromadnej obsluhy – SHO; Simulator of Queuing Networks – QN.* Available at: ⟨https://github.com/raubirius/SHO/⟩. Last accessed: 2019-12-05.

[24] Horváth, Roman. (2023). Developing a User-Centric Queueing Simulation Engine for Educational Purposes. In *ICETA 2023. «in press».*

[25] Dvořák, Jiří. (2002). *Systémy hromadné obsluhy.* Available at: ⟨http://www.uai.fme.vutbr.cz/~jdvorak/vyuka/tsoa/PredO11.ppt⟩. Last accessed: 2019-12-05.

[26] Ullrich, Oliver – Lückerath, Daniel. (2017). An Introduction to Discrete-Event Modeling and Simulation. In *Simulation Notes Europe SNE 27(1), 9–16.* ⟨https://doi.org/10.11128/sne.27.on.10362⟩. Last accessed: 2019-12-05.

[27] Sundarapandian, Vaidyanathan. (2009). *Probability, Statistics and Queuing Theory.* India : Prentice Hall India Pvt., Limited.

[28] Rábová, Zdeňka – Zendulka, Jaroslav – Češka, Milan – Peringer, Petr – Janoušek, Vladimír. (1992). *Modelování a simulace.* Brno : Vysoké učení technické v Brně. ISBN 80-214-0480-9.

[29] Law, Averill M. – Kelton, W. David. (1991). *Simulation Modeling and Analysis.* 2nd Edition. New York : McGraw-Hill. ISBN 0-07-100803-9.

[30] Dorda, Michal. (2009). Modelování systému hromadné obsluhy M/M/1/1. In *Perner's Contacts, 4(1), 50–58.* Available at: ⟨https://pernerscontacts.upce.cz/index.php/perner/article/view/1091⟩. Last accessed: 2023-09-15.

[31] Važan, Pavel – Križanová, Gabriela – Červeňanská, Zuzana – Znamenák, Jaroslav. (2017). *Modelovanie a simulácia systémov. Simulátor Witness : Návody na cvičenia.* Trnava : AlumniPress. ISBN 978-80-8096-252-4.

[32] Horváth, Roman. (2022). Optimisation of Algorithms Generating Pseudorandom Integers with Binomial Distribution. In *ICETA 2022.* Danver : Institute of Electrical and Electronics Engineers, pp. 197–201. ISBN 979-8-3503-2032-9.

[33] Mathur, Shikhar. (2022). *Argument vs Parameter in Java.* From: GeeksforGeeks. Available at: ⟨https://www.geeksforgeeks.org/argument-vs-parameter-in-java/⟩. Last accessed: 14. 8. 2023.

[34] Alam, Bashir. (2022). *Java Arguments vs Parameters [Syntax & Examples].* From: GoLinuxCloud. Available at: ⟨https://www.golinuxcloud.com/java-arguments-vs-parameters/⟩. Last accessed: 14. 8. 2023.

[35] Baeldung. (2022). *The Difference Between an Argument and a Parameter.* From: Baeldung on Computer Science. Available at: ⟨https://www.baeldung.com/cs/argument-vs-parameter⟩. Last accessed: 14. 8. 2023.

[36] Vaish, Priyanshu. (2022). *Difference Between Argument and Parameter. Argument vs Parameter.* From: BYJU's Exam Prep. Available at: ⟨https://byjusexamprep.com/difference-between-argument-and-parameter-i⟩. Last accessed: 14. 8. 2023.

[37] Diwan, Amit. (2022). *What is the difference between arguments and parameters in Python?* From: Tutorialspoint. Available at: ⟨https://www.tutorialspoint.com/what-is-the-difference-between-arguments-and-parameters-in-python⟩. Last accessed: 14. 8. 2023.

[38] Horváth, Roman. (2023). Report on the implementation of the additional screen components of the GRobot programming framework. In *36th DidMatTech 2023, Aktualne wyzwania współczesnej edukacji.* (Eds.) Sałata, Elżbieta – Ziębakowska-Cecot, Katarzyna – Feszterová, Melánia [Recenzent]. Radom (Poland) : Kazimierz Pulaski University of Technology and Humanities in Radom. ISBN 978-83--7351-991-6. ISSN 1642-5278, s. 149–156.[0.47 AH].

[39] Viljanen, Susanna. (2021). *Answer to What's zero in the Roman numerical system?* From: Quora. Available at: ⟨https://qr.ae/pymPgP⟩. Last accessed: 15. 8. 2023.

[40] Špendla, Lukáš – Tanuška, Pavol – Štrbo, Milan. (2013). Model proposal for performance testing of safety-critical systems. In *Proceedings of the Third International Conference on Control, Automation and Systems Engineering (CASE-13)*. Paris (France) : Atlantis Press. Advances in Intelligent Systems Research, pp. 42–45. ISSN 1951-6851. ISBN 978-90786-77-81-9.

[41] Stoffová, Veronika – Gabaľová, Veronika. (2023). Interactive simulation models for teaching and learning of computer science. In *INTED2023 – International Technology, Education and Development Conference 2023.* ⟨https://doi.org/10.21125/inted.2023.1050⟩. Barcelona (Spain) : IATED, pp. 3945–3953. ISSN 2340-1079. ISBN 978-84-09-49026-4.

[42] Horváth, Roman. (2019). Zhodnotenie dvojsemestrálneho predmetu OOP vyučovaného metódou projektového vyučovania – Evaluation of the Two-term Course OOP taught by the Project-based Teaching Method. In *32th DidMatTech 2019.* Trnava : Trnavská univerzita v Trnave. 7 s. [0.66 AH]. ISBN 978-80-568-0398-1. Available at: ⟨http://didmattech.truni.sk/2019/proceedings/#horvath⟩. Last accessed: 18. 8. 2023.

A Graphical Programming Framework for Didactic Purposes

Author:	Mgr. Ing. Roman Horváth, PhD.
Year:	2024
Publisher:	Faculty of Education, Trnava University in Trnava
Printing and Distribution:	Online publication
Edition:	First edition
Number of pages:	48

ISBN 978-80-568-0680-7

https://doi.org/10.31262/978-80-568-0680-7/2024


9 788056 806807