

Vetle Olav Pettersen Finstad

Parallel Programming with MPI through Gaming

MPI Edutainment Game

Master's thesis in MIDT

Supervisor: Professor Anne C. Elster

June 2022

Vetle Olav Pettersen Finstad

Parallel Programming with MPI through Gaming

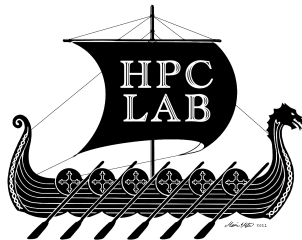
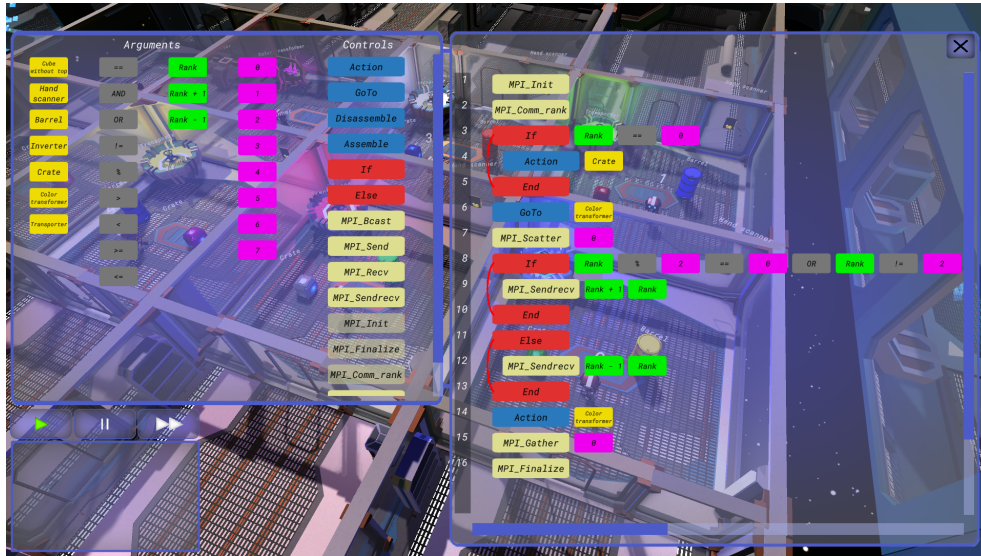
MPI Edutainment Game

Master's thesis in MIDT
Supervisor: Professor Anne C. Elster
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Vetle Olav Pettersen Finstad

Parallel Programming with MPI through Gaming



Master's thesis in Computer Science
Supervisor: Professor Anne C. Elster
June 2022

Abstract

Lack of experience with parallel programming among programmers is a growing concern. This is problematic as parallel programming is required to fully utilize components such as GPUs and multi-core CPUs. We address this problem in our thesis by building an environment for learning parallel programming with MPI. MPI is one of the dominating programming extensions for programming highly parallel systems. Through developing an edutainment game that simulates the MPI environment, we intend to make the process of familiarizing programmers with MPI more fun and interesting.

Our thesis work is an extension of the work done as part of the author's fall project. In enhancing our MPI game, we consider standard qualitative gaming characteristics that might impact players' engagement in the game. These considerations include appropriate game difficulty, limiting new game experiences, and a concise game tutorial. While further developing the game, we are cognizant of our choice and illustration of MPI-functions while, at the same time, implementing several new MPI-functions.

We also make use of the results from a prior user study, as well as a user study conducted for our extended work. The latter tries to determine the game's enjoyability and game difficulty, as well as players' perceived learning and interest in the field of parallel programming.

Some of the technical and conceptual challenges we faced when making an edutainment game, precisely for familiarizing a user with parallel programming, are also described.

In particular, we discuss our choice of using the Unity game engine and how we used an approach where players can visually program inside the game with MPI. The MPI processes are simulated by robots. Furthermore, game-elements are implemented to abstract out code such as inversion of a pixel color. Users are also able to add to and look at the underlying MPI codes that control these robots.

MPI functions simulated include `MPI_Init`, `MPI_Finalize`, `MPI_Comm_rank`, `MPI_Send`, `MPI_Recv`, `MPI_Bcast`, `MPI_Scatter`, `MPI_Gather` and `MPI_Sendrecv`.

Our final user study showed this game has a lot of potential. A discussion of several avenues for future work is also included.

Sammendrag

Mangel på erfaring med parallell programmering blant programmerere er en økende bekymring. Dette er problematisk ettersom parallell programmering kreves for å utnytte komponenter som GPUer og flerkjerne-CPUer fullt ut. Vi tar opp dette problemet i oppgaven vår ved å bygge et miljø for å lære parallell programmering med MPI. MPI er en av de dominerende programmeringsutvidelsene for programmering av svært parallelle systemer. Gjennom å utvikle et edutainment-spill som simulerer MPI-miljøet, har vi til hensikt å gjøre prosessen med å gjøre programmerere kjent med MPI morsommere og mer interessant.

Avhandlingsarbeidet vårt er en forlengelse av arbeidet som er gjort som en del av forfatterens høstprosjekt. For å forbedre MPI-spillet vårt vurderer vi standard kvalitative spillegenskaper som kan påvirke spillernes engasjement i spillet. Disse vurderingene inkluderer passende spillvanskeligheter, begrensede nye spillopplevelser og en kortfattet spillveiledning. Ved utvikling av spillet tar vi også hensyn til vårt valg og illustrasjon av MPI-funksjoner samtidig som vi legger til flere MPI-funksjoner.

Vi bruker også resultatene fra en tidligere brukerstudie, samt en brukerstudie til vårt utvidede arbeid. Sistnevnte prøver å bestemme spillets fornøyelse og spill-evanskelighet, samt spillernes opplevde læring og interesse for feltet parallellprogrammering.

Noen av de tekniske og konseptuelle utfordringene vi møtte ved å lage et edutainmentspill nettopp for å gjøre en bruker kjent med parallell programmering blir også beskrevet.

Spesifikt, diskuterer vi vårt valg av å bruke Unity-spillmotoren og hvordan vi brukte en tilnærming der spillere visuelt kan programmere inne i spillet med MPI. MPI-prosessene simuleres av roboter. Videre er spillelementer implementert for å abstrahere ut kode som, for eksempel, inversjon av en pikselfarge. Brukere kan også legge til og se på de underliggende MPI-kodene som styrer disse robotene.

Simulerte MPI-funksjoner inkluderer `MPI_Init`, `MPI_Finalize`, `MPI_Comm_rank`, `MPI_Send`, `MPI_Recv`, `MPI_Bcast`, `MPI_Scatter`, `MPI_Gather` og `MPI_Sendrecv`.

Vår siste brukerstudie viste at dette spillet har mye potensial. En diskusjon av flere veier for fremtidig arbeid er også inkludert.

Acknowledgement

I would like to thank my advisor Anne C. Elster, for providing me with important feedback along the way. Moreover, I would like to thank her for the great ideas for the game during development she has given me. I would also like to thank her for giving me access to the HPC-Lab, providing me with a possible workspace and hardware to create my application.

I would also like to express my gratitude to my girlfriend, Olga Sandvær, for all her love and support during this difficult time. And my friend, Gabriel Martinussen, for his invaluable discussions around the game and thesis. A final gratitude for all people that took the time to test my game, you know who you are.

Contents

Abstract	iii
Sammendrag	v
Acknowledgement	vii
Contents	ix
List of Figures	xiii
List of Tables	xv
Listings	xvii
Acronyms	xix
Glossary	xxi
1 Introduction	1
1.1 Motivation	1
1.2 Contribution	1
1.3 Research Questions	2
1.4 User Study Questions	2
1.5 Thesis Outline	2
2 Background	5
2.1 Important Video-Game Design	5
2.1.1 Design for an Engaging Game	6
2.1.2 Environment Design	8
2.2 Game Engine	9
2.2.1 Unity	10
2.2.2 Unreal Engine	10
2.3 Parallel Programming	10
2.3.1 Message Passing	11
2.3.2 Message Passing Interface	11
2.3.3 Black/White Image Inversion	12
2.3.4 Border Exchange	12
2.4 Bezier Curve	13
2.5 Related Work	15
2.5.1 Human Resource Machine	15
2.5.2 Scratch	15
3 Developing an MPI Edutainment Game	17
3.1 Choosing a Game Engine (RQ1)	17
3.2 Description of the Game	18

3.3	Game Models and Licensing	19
3.4	Timing	19
3.5	Visual Programming	19
3.5.1	Controlboard	19
3.5.2	Visual Programming Architecture	20
3.5.3	Development Process	22
3.6	Simulation of Processes	23
3.6.1	Development Process	23
3.6.2	Simulation of Functions	24
3.6.3	MPI-Interpreter (RQ2)	28
3.7	Interactable Objects	32
3.8	Conversion from Visual Programming to Programming Language C	33
3.9	Simulating MPI Programs	36
3.9.1	Black/White Inversion	36
3.9.2	Border Exchange	36
3.10	Handling of Exceptions	37
3.11	Interactive tutorial	38
3.12	Local Game Save Data and Audio	39
4	User Study	43
4.1	Interview	43
4.2	Survey	44
4.3	Observation	44
4.4	Quiz	44
4.5	Reflection on our User Study	44
5	Results	47
5.1	Development Environment	47
5.2	Recommended Specifications for Running the Game	47
5.3	Development Results	49
5.3.1	Game Main Menu	49
5.3.2	Settings	49
5.3.3	Level Selector	50
5.3.4	Level Overview	50
5.3.5	Visual Programming	52
5.3.5.1	Displaying an Exception in a Control	54
5.3.6	Console	54
5.3.7	Level Completed	55
5.3.8	Displaying Equivalent C Code	56
5.3.9	Tutorial	57
5.3.10	Interactable Objects	57
5.3.11	Border Exchange	58
5.4	User Study Results	61
5.4.1	Increase of Interest in Parallel Programming	61
5.4.2	Game Difficulty for the Players	61
5.4.3	Learning Effect of the Game	62

5.4.4	Enjoyment Effect of the Game	63
6	Discussions	65
6.1	Development Results	65
6.1.1	Main Menu	65
6.1.2	Settings	66
6.1.3	Level Selector	66
6.1.4	Visual Programming	67
6.1.5	Simulation of Processes	68
6.1.6	Simulation of Functions	68
6.1.7	Tutorial	69
6.1.8	Border Exchange	69
6.2	User Study	70
6.2.1	Increase of Interest in Parallel Programming	70
6.2.2	Game Difficulty for the Players	70
6.2.3	Learning Effect of the Game	71
6.2.4	Enjoyment Effect of the Game	71
7	Conclusion and Future Work	73
7.1	Research Questions	73
7.2	User Study Questions	74
7.3	Future Work	76
	Bibliography	77
A	User Guide	79
A.1	Main menu	79
A.1.1	Gameplay	80
A.2	Objects	84
B	Code Snippets	87
C	Survey and Interview Questions	95
C.1	Survey Questions	95
C.2	Interview Questions	95
D	Quiz Questions	97
E	Poster	107

List of Figures

2.1	Border Exchange between Process 1 and Process 2 [18]	13
2.2	Bezier curve according to the control points [19]	14
2.3	The visual programming inside the game [20]	15
2.4	Scratch's user interface [21]	16
3.1	Architecture for controls and arguments	21
5.1	Main menu in our fall project	49
5.2	Main menu in our current project	49
5.3	Settings	49
5.4	Old level selector	50
5.5	New level selector	50
5.6	Level 6 in our fall project	50
5.7	Level 12 in our current game	51
5.8	The controlboard in the our project	52
5.9	The controlboard in the current project	52
5.10	Overview of arguments for If control in our fall project	52
5.11	Overview of arguments and controls in current project	53
5.12	Closer look at the active controls and arguments in current project	53
5.13	Describes what an Action control does	54
5.14	An exception display in a control in our fall project	54
5.15	An exception displayed in a control in the current project	55
5.16	The console displaying an exception in the current project	55
5.17	Interface when completing a level in our fall project	56
5.18	Interface when completing a level in the current project	56
5.19	Equivalent C code	57
5.20	Tutorial in our fall project	57
5.21	Tutorial in our current project	57
5.22	Crate	58
5.23	Barrel	58
5.24	Cube	58
5.25	Transporter	58
5.26	Transformer	58
5.27	Inverter	58

5.28 Hand scanner	58
5.29 Key	58
5.30 Scattering a cube to other robots	59
5.31 Robots disassembles its cube, and exchanging with its neighbour . .	59
5.32 Robots assembling the cube back together, creating a cube with only one color	60
5.33 The root robot gathers all white cubes from the other robots	60
A.1 Caption	79
A.2 Main menu	80
A.3 Selection of a level	80
A.4 Example of picking up a crate, putting it onto the transporter and initiating transportation	81
A.5 Description of what the Action control do	81
A.6 Showcasing start, pause and fast forward buttons	82
A.7 Run-time exception occurred to robot 0	82
A.8 Panel that opens when player completes a level	83
A.9 Converts visual programming code to C code. This equivalent C code is converted from A.4	83
A.10 Crate	84
A.11 Barrel	84
A.12 Cube	84
A.13 Transporter	84
A.14 Transformer	84
A.15 Inverter	84
A.16 Hand scanner	84
A.17 Key	84
A.18 Robot	85
D.1 Question 1	97
D.2 Question 2	97
D.3 Question 3	98
D.4 Question 4	98
D.5 Question 5	99
D.6 Question 6	99
D.7 Question 7	100
D.8 Question 8	101
D.9 Question 9	102
D.10 Question 10	103
D.11 Question 11	104
D.12 Question 12	105

List of Tables

5.1	Minimum Specifications for development of the game, based on Unity's own website. For more information: Link	48
5.2	Question 4 and 5 from the survey	61
5.3	Question 1 and 6 from the survey	62
5.4	Question 2 from the survey	63
5.5	Question 3 from the survey	63

Listings

3.1	Wrapper for controls	20
3.2	Functions for spawning controls and arguments in Controlboard.cs	20
3.3	Creation of a bezier curve	23
3.4	Robot iterating over controls	24
3.5	Using the control name to determine which function to run	25
3.6	Matching an argument name to an object	25
3.7	Extraction of the three first arguments	26
3.8	Checking whether the sentence is true or false	26
3.9	Checking whether the sentence is true or false with modulo	27
3.10	More arguments imminent and the sentence was true	27
3.11	More arguments imminent and the sentence was false	27
3.12	Else function	28
3.13	Declaring control points	29
3.14	Retrieving bezier points, the highest point of the curve and declaring some variables	29
3.15	Animating the game-element to the highest point, and then to its final destination	29
3.16	Extracting out the receiver	30
3.17	Create handler if handler is null, then subscribe to action and increment number of ready workers	30
3.18	Function in MPIHandler.cs	31
3.19	Creating a new BroadcastHandler object	31
3.20	If owner of <i>MPI_Bcast</i> , subscribe to action and increment, otherwise call the function WorkerReady	31
3.21	Start of interaction with object	33
3.22	Conversion to camel-case	33
3.23	The AND operator in Arguments.json	34
3.24	Example of MPI_Sendrecv in controls.json	34
3.25	Inserting arguments at appropriate indexes	35
3.26	Adding arguments to code output	35
3.27	Conversion to C with an If control	36
3.28	Small cubes being sent to robots in Worker.cs	37
3.29	Creation of an exception	38

3.30	Overridden function in <code>Level1Tutorial.cs</code>	39
3.31	UserAction enum in <code>LevelTutorial.cs</code>	39
3.32	Setting a level button to be interactable or not in <code>MainMenu.cs</code>	39
3.33	Plays and loops a song based on level index	40
3.34	Subscribing to the PlaySound Action in <code>GameHandler.cs</code>	40
3.35	An enum to distinguish what sound effect to play	40
3.36	Playing sound based on what enum	40
3.37	Invoking the Action in <code>GameHandler</code> to play sound effect when a control/argument is dragged in <code>Draggable.cs</code>	41
B.1	Function for instantiating control	87
B.2	Function for instantiating argument	88
B.3	Function for checking whether an If control is true or false	89
B.4	Checking for exceptions in a control	90
B.5	Creation of a small red line	93
B.6	Setting the red line's position and rotation	93
B.7	Calculating the bezier point	93

Acronyms

edutainment educational entertainment. 1, 2, 7, 15, 17, 65, 66, 73, 75

fps frames per second. 32, 47, 66

MP Message Passing. 11

MPI Message Passing Interface. 1, 2, 11, 12, 18, 22, 23, 28, 32–34, 36, 43, 44, 52, 61–63, 65, 67–69, 71, 73–76

MPP Massively Parallel Processing. 11

Glossary

Emergence Defining general global rules to induce emergent gameplay. 8

Scripting Developers hand-craft, anticipate -and script game-objects, interactions and events. 8

Chapter 1

Introduction

Because the limits of CPU frequency scaling have been reached, computers today are equipped with multi-core processors and GPUs. To make use of these components, parallel programming is required. Parallelism offers a huge performance improvement that should be taken advantage of. Despite the high demand for parallel programming skills, many computer scientists will graduate in 2022 with little or no experience in the field according to [1].

This thesis project expands on the work done in the "*fall project*". Several Message Passing Interface (MPI) functions have been added, including illustrating border exchange examples. The visual programming has also been revamped. A user study was done in the fall project, this thesis has the same set-up, but the questions have been changed. The background chapter has been extended and its sections, although also based on the fall project, have been rewritten for readability.

1.1 Motivation

Video games have seized the world by storm, and they are now a common part of people's daily lives. As a result, scientists began researching the world of video games and how it affects individuals. They started looking at how a video game can be both educational and entertaining, coining the phrase "edutainment game." These games are used to educate a wide range of subjects. With today's technology, a video game may provide an immersive and interactive experience, increasing attention span and focus.

1.2 Contribution

We demonstrate how an educational entertainment (edutainment) game may be utilized as an environment for learning parallel programming using MPI. We also show how to simulate MPI functions in a game, as well as how to simulate oversimplified image processing and border exchange, to enable players to become

familiar with and understand how parallelism speeds up a program.

1.3 Research Questions

Four research questions have been induced to collect relevant information about the project's overall solution.

RQ 1: *"What is the best game engine for implementing an MPI edutainment game?"*

RQ 2: *"What are some of the technical advantages and challenges with the approach chosen?"*

RQ 3: *"What are the most useful MPI functions, and how does one illustrate them well in a gaming setting?"*

RQ 4: *"Will there be any timing issues related to such a game?"*

1.4 User Study Questions

Four different user study questions have been produced to demonstrate the quality of the edutainment game, as well as to collect useful information about players' experience with the game.

USQ 1: *"Will the game have an effect on increasing interest in parallel programming?"*

USQ 2: *"Is the game too difficult?"*

USQ 3: *"What is the learning effect of playing the game?"*

USQ 4: *"What is the enjoyment effect of playing the game?"*

1.5 Thesis Outline

The rest of the thesis is outlined as follows:

- Chapter 2 holds all the relevant background and related work relevant for this thesis
- Chapter 3 describes how the game was implemented
- Chapter 4 presents the user study method applied to answer the user study questions
- Chapter 5 holds the development results as well as the results from the user study
- Chapter 6 contains a discussion of the results in chapter 5 for both the development results and the user study
- Chapter 7 concludes this thesis and answers both the research -and user study questions as well as presenting the future work
- Appendix A: User guide
- Appendix B: Code snippets
- Appendix C: Survey and Interview Questions
- Appendix D: Quiz Questions

- Appendix E: Poster

Chapter 2

Background

Nuclear physicist William Higgingbotham created one of the first games called *Tennis for two* in 1958, which was designed to inform visitors of the Brookhaven National Laboratory on the effects of gravity [2].

But the real breakthrough for video games happened in 1961 when MIT student Steve Russell created the game *Spacewar!*. It included elements that are still used in video games today such as a scoring system, in-game settings, and a two-player mode, to mention a few. It was the slow beginning of video-game's growth to the commercial summit [2].

The video-gaming industry has grown huge over the last 60 years. According to [3], in 2020, there were 2.69 billion players playing video games worldwide. Furthermore, the revenue from gaming reached over \$159.3 billion. This shows the immense pull gaming has on people.

The following sections contain the relevant background information for this thesis about important video-game design, parallel programming with the message passing interface, and some general image processing techniques including both independent as well as more complicated features.

2.1 Important Video-Game Design

One of the reasons people play video games is because it is fun. Fun also plays an important role in learning for both adults and children. According to [4], pleasure and enjoyment function as an incentive to obtain the necessary knowledge and skills. Furthermore, fun and enjoyment encouraged concentration, which aided understanding. Some researchers describe how the game's difficulty should be managed: "The player should be able to customize the controls and the gameplay to fit their learning and playing styles or the game should be designed to allow different styles of learning and playing." [5] The following sections describe the elements a developer should keep in mind to create a fun and enjoyable experience for players.

2.1.1 Design for an Engaging Game

In 1980, Tom W. Malone [6] discovered five important characteristics for making educational games fun. Years later, Sweetser and Wyeth [5] studied the flow elements in games, where they identified eight different flow elements. Despite the fact that the authors list five and eight separate features, they can be divided into three groups: *challenge*, *curiosity* and *fantasy*.

Challenge

"Challenge is consistently identified as the most important aspect of good game design." [5]

As discussed in [5, 6], games should have a clear goal and be sufficiently challenging. It is also critical that the game's challenge does not surpass the player's ability, as this may result in players experiencing anxiety. On the contrary, too easy a game might make players feel unmotivated to play.

Players experience enjoyment from the challenge of the game. A game should portray itself as hard through careful level design that triggers a sequence of challenging scenarios - giving an experience that encourages players to keep playing the game. Providing such an experience must first of all present a goal to reach. This goal must be clearly defined with immediate feedback on players' progress towards it. The clearer the goal, the better.

A game can be made challenging by the introduction of variable levels of difficulty. This can, for instance, be determined automatically by the game, chosen by the player, or automatically determined by their opponent's skill level. When the difficulty is determined automatically by the game, it is important that the game gradually increases its difficulty when the player progresses, maintaining the player's interest. Equally important is that the game does not start too difficult, else, players can feel discouraged.

Players can also feel discouraged by experiencing new details and challenges at a too high rate. To sustain the game's challenge and tension, unlocking new mechanics or tougher foes should be done at a steady pace. This also serves to pique the player's curiosity. Another way of making a game uncertain is by introducing several goals for the player to reach. While the player's primary goal is to achieve something in the game world, they may also have additional meta-goals. These meta-goals can come in the form of speeded responses or maximizing scorelines.

The game should also be playable without the need for external resources; players should not be expected to read, for instance, a game manual to start playing the game. This may be avoided by including a tutorial that teaches the player how to continue through the game. This typically happens in the earlier stages of the game. However, the tutorial mustn't provide lengthy explanations, as this can result in the player becoming bored. Moreover, it should be obvious to the players when they have done something wrong, and how it can be resolved.

Another valuable contribution to a game's challenge is that the players should feel a sense of control over their actions. The user interface should be intuitive

and easy to use - and its controls should be easy to learn and master. "The game shell menu should be easy to use, intuitively organized, and should not sacrifice readability and functionality for aesthetics." [5].

Fantasy

Another way of making a game engaging can be made possible according to [6], is by implementing *fantasy*, where the game includes images of physical objects or social situations not present. Malone found a relatively easy way of trying to increase the fun of learning where the player progresses toward a fantasy goal or tries to avoid a fantasy disaster. Malone put *fantasy* in two different categories: *extrinsic and intrinsic fantasies*. This can, for instance, be demonstrated by an arithmetic problem game; an extrinsic fantasy could be that hangman is used to indicate whether the player in the game was incorrect - fantasy is dependent on the skill. Whereas for intrinsic fantasies, the skill is also dependent on the fantasy. An arithmetic problem game might be used to explain this. In the game, the player must travel to the proper tile indicating the correct number for the arithmetic problem provided, while wrong responses result in a bomb creeping closer to explosion. He found that intrinsic fantasies are more interesting than extrinsic fantasies.

Curiosity

Motivation to learn is important in an edutainment game as discussed in [6], this is where *curiosity* comes in. *Curiosity* can be evoked by providing environments that are neither too complicated nor too simple in relation to the player's existing knowledge. A good game world/level is not completely incomprehensible, yet still surprising and new. It is also categorized in two different categories: *sensory* and *cognitive*.

Sensory curiosity is the response to changes in the environment in the form of audio, sound, or movement. Malone states different strategies to invoke sensory curiosity. One way is to "decorate" the environment, by putting in audio, regardless of what the player is doing. This is often referred to as background music. "My conjecture is that this kind of effect will enhance the initial interest of a game, but will quickly become boring." [6]

Another strategy is to enhance the fantasy of the game, by using sound or graphics. Rewarding the player for good performance with visuals or music was also discovered to be important, while also adding to the game's difficulty. Lastly, representing things inside a game using sound or graphics. It is a far superior way of representing things inside a game, instead of using text and numbers.

Cognitive curiosity builds upon the fact that people want to expand on existing knowledge. The degree of curiosity is related to the knowledge level one has in a field. A person's curiosity is very low, or non-existent when their knowledge of the field is zero. However, having grasped a subject will often lead to people wanting to broaden their knowledge. For instance, watching a TV series 85% finished,

one is much more likely to finish the series. Compared to a series that has been watched only 15%. Cognitive curiosity is also invoked by finding out information that conflicts with what the person currently has knowledge about.

2.1.2 Environment Design

The paper [7] discusses and defines two divergent approaches to designing game environments:

1. *Scripting*.
2. *Emergence*.

Scripting is the most popular alternative for developing games; the developer designs predefined paths and interactions that the player will follow throughout the game. This constrains the game to its designer's ideas of what is consistent and enjoyable, which can make player interactions appear limited, rigid, and lifeless.

The alternative is *emergence*, where game elements are globally defined with consistent characteristics and behavior. Players' interactions have rules and boundaries, rather than prescribed paths.

When designing a game using *scripting*, there must be an effort in the placement of game objects, puzzles, and sequences to maintain consistency in the game environment, which requires significant time and work from the designers. Furthermore, extensibility in scripted systems poses an issue: because each instance of a game object is unique, there must be a direct link between them for every interaction. Since there can be no uncertainty or unexpected events in the game, the need for testing and quality assurance is high. However, *scripting* is effective for developing simple systems. Additionally, it is straightforward to give feedback as the developer knows how and when the player will interact with various game elements.

However, when designing a game where *emergence* is the desired approach, the developer designs only the types of objects and interactions, resulting in greater efficiency in development and testing. In contrast to a *scripted* system, an *emergent* system scales well, and is easily extended. Making changes to an *emergent* system is more efficient as changes can be made to object types, instead of each instance of a game object. *Emergence* can easily make the game designer feel a loss of creative control, as using this particular approach makes it harder to set up a specific narrative. As a result, telling a story or controlling the flow of the game is not as simple as in the *scripting* approach. On the other hand, *emergence* makes it possible to introduce uncertainty, which is one of the ingredients for making the game feel challenging according to *Malone*. However, it may also allow harmful behavior in the game, which requires extensive testing to ensure it is forbidden. As the game gives rise to numerous actions, assuring the players that they are on the right track with feedback and direction is important.

When it comes to the game's players, the *scripting* technique easily breaks their immersion since it lacks consistency between the real world and the game. However, with *emergent* gameplay, the game environment is closely integrated with the actual world, preserving the players' immersion. Another compelling reason to use the emergent technique is that players will not have to relearn the game's conceivable interactions, keeping it intuitive and satisfying the player's expectations.

The paper states that both approaches hold benefits and drawbacks for game developers, with consequences for the players. Developers of the *scripted* approach must hand-craft, implement and test every aspect of the game individually while keeping full creative control and knowing the game will not break after release. The drawbacks are that players are unable to express their creativity, and inconsistencies may appear. For the *emergent* method, developers are bound to be uncertain of the game's behavior after its release, however, it allows players to express their creativity while keeping the game world consistent and intuitive.

The authors concluded that game development should fall somewhere in the middle. A good balance of planned, narrated gameplay and freedom to interact, allows the player to do their own thing while yet driving the plot forward.

2.2 Game Engine

Before the arrival of the game "DOOM" in 1993, id Software promised that the game would push the boundaries of the processing power of a computer. The game demonstrated significant technological, content, and gameplay advancements. A new term was coined: Doom Engine; the term described the technology for id Software's newest game software and later revolutionized the game industry [8].

With the release of this game engine, it became possible to avoid creating games from scratch, saving valuable time. Texturing, animation, lighting, and 3D rendering, among other things, could now be easily implemented with the help of this game-changing engine [9].

In 2022, game engines are ubiquitous. They are more powerful and feature-rich than the original game engine. The motivation for using a game engine comes from the desire to save time when developing a game from the ground up. Developers can concentrate on game logic and interactions, while the engine handles everything from graphics rendering to collision detection.

There is a time limit for the delivery of this thesis, hence, it is crucial to make use of a game engine to develop the game. However, it is important to choose the right game engine for the system that is going to be developed. Game engines are becoming increasingly similar; however, they differ slightly in some areas, such as graphic quality.

2.2.1 Unity

On May 21, 2002, Nicholas Francis and Joachim Ante began working together on a game engine and made good progress, which piqued the interest of David Helgason, who eventually joined the team. Initially, the three developers intended to make video games for a living, but they quickly realized that they would rather create a tool for making games. When Joachim and Nicholas rented a flat in Copenhagen to become roommates, with David living just down the street, development really started to take shape. Launched in 2005, Unity would become one of the most successful game engines to exist [10].

According to [11], in 2020, there were 5 billion downloaded apps built in Unity and 2.8 billion monthly active end-users who engaged with content created in or operated by Unity. It remains the game engine of choice; 61% of game developers chose Unity as their game engine in 2020.

Unity is a cross-platform game engine, supporting both 2D and 3D graphics, with C# as its scripting language. What makes Unity different from other game engines is its huge target platforms, simplicity, and asset store. The asset store is a place where developers can upload their creations and share them with the community [12].

2.2.2 Unreal Engine

In 1991, 21-year-old Tim Sweeney created a game called ZTZ. It was a simple game, but the approach he used to program that game would germinate into something much bigger. He designed the game in such a way that allowed him to have easy control over gameplay objects without much complexity. This would allow a user to do significant user modifications – which laid out the framework for the idea of a game engine. With Tim Sweeney, Epic Games began working on Unreal, which developed into arguably the best-looking shooter game of its time. As a result, competitors became interested and desired to develop their games on Epic Games' engine [13].

The Unreal Engine is a game engine that, like Unity, supports both 2D and 3D graphics and employs C++ as its scripting language. Unreal Engine's huge potential for creating beautiful graphics is where it truly shines, and it is frequently picked up and used by AAA game companies [14].

2.3 Parallel Programming

Microprocessor performance increased by more than 50% per year on average between 1986 and 2003. This implies that users and software developers could simply wait for the next generation of microprocessors to improve application performance. However, since 2003, single-processor performance has slowly declined. In 2005, the majority of big manufacturers decided that parallelism was

the way to go, and they began putting multiple complete processors on a single integrated circuit [1].

This change had a significant impact on software developers; simply adding more processors would not improve serial-application performance. Applications must be parallelized to take advantage of these multi-core processors. This is where parallel programming comes in [1].

2.3.1 Message Passing

Message Passing (MP) is a parallel programming model and is defined by having the ability to communicate with other processes. It is very limited: only a copy of an item, known as a *message* can be sent from one process to another. Nonetheless, it is effectively capable of cooperating programs having read/write access to each other's local memory. A process that receives a message can write it to its local memory, allowing the sender of the message to modify the receiver's local memory. A Massively Parallel Processing (MPP) hardware using MP, consists of P sequential programs, where every process runs its own program. Each of these processes uses MP instructions to synchronize themselves and access the memory of other processes [15].

2.3.2 Message Passing Interface

MPI was developed in a year (1993-94) of intensive meetings, involving more than 80 people from 40 different organizations, mostly from the US and Europe. Voting at the meetings was done by a single vote per organization, and to vote, the organization must have had a representative at two of the last three meetings. Many vendors of concurrent computers were involved, along with researchers from universities, government, laboratories, and industries. This was how the MPI specification was publicized [16].

MPI uses the MP parallel programming model. It works by moving data from the address space of one process to another through cooperative operations on each process. All of the MPI operations are expressed as functions, subroutines, or methods, in relation to the language bindings. The main advantage of MPI is its portability and simplicity [17].

MPI works by having all processes grouped in a *communicator*. The most common communicator in an MPI program is a communicator called **MPI_COMM_WORLD**. Each process is identified by a unique number, which is retrieved and stored in a variable by calling the function **MPI_Comm_rank**. By doing this, processes can be differentiated and manipulated to do separate things.

An MPI program contains *synchronization points*, these are points in the program where processes synchronize themselves with all or some processes in the program. It means that processes wait for other processes to reach the same point in the program before continuing. An example of this can be made in a simple send and receive program between two processes. The sending process (process 0) has to call the send function, while the receiving process (process 1) has to

call the receive function. When process 0 reaches the send function, it waits until process 1 reaches its receive function and then continues - and vice versa. Another example is where a collective operation is done, the root process (process 0) sends information to all other processes. It then waits until all other processes have reached this point before it can send the information, and then continues further.

Using a lot of communication between processes can slow down an MPI program, which is called **communication overhead**. A program can run faster with eight processes instead of, for instance, sixteen processes. One must be careful when choosing the number of processes used to run the program.

MPI Functions

All of the functions simulated in the game are covered and explained here.

- **MPI_Init**, initializes the MPI execution environment.
- **MPI_Finalize**, terminates the MPI execution environment.
- **MPI_Comm_rank**, retrieves the rank of the process and stores it in a variable.
- **MPI_Send**, sends data to a recipient process.
- **MPI_Recv**, retrieves data from a sending process.
- **MPI_Sendrecv**, both sends and retrieves data.
- **MPI_Bcast**, sends data to all other processes.
- **MPI_Scatter**, splits up and sends an equal amount of array data to all other processes.
- **MPI_Gather**, receives array data from all other processes and stores it in a single array.

2.3.3 Black/White Image Inversion

This is an *independent* parallel programming task, meaning the processes do not need to communicate with each other during the processing. The root process scatters an image to the other processes, who can all independently invert the pixel color of their portion of the image. Afterward, the root process can gather the results from the processes.

2.3.4 Border Exchange

The majority of MPI applications are complex and require domain decompositions (splitting the main system matrix over several processes). Several applications in science and engineering programs then iterate over the distributed domain, but must update the results via "border exchanges." When doing image filtering, for example, the root process scatters the image to all of the processes, and the processes begin the filtering process. When processing its individual image chunk with a kernel, one difficulty arises: It requires data from its neighboring processes

to complete its computation. While performing image filtering, processes receive border pixels from neighboring processes and send border pixels to other processes. *Border exchange* is the common name for this operation [18].

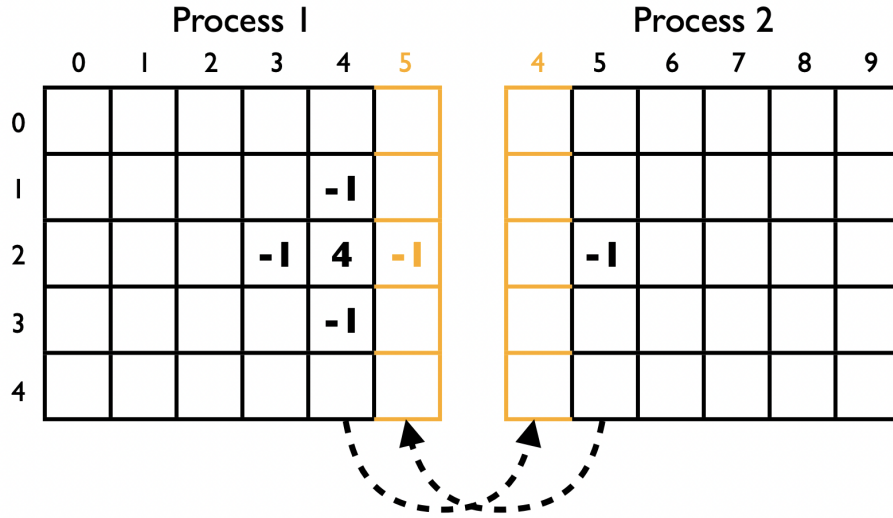


Figure 2.1: Border Exchange between Process 1 and Process 2 [18]

2.4 Bezier Curve

One way of expressing geometric design is by utilizing Bezier curves. By establishing "control points", a smooth and continuous curve that may be scaled endlessly can be created. The first and last "control points" are always the start and end-points of a bezier curve [19]. A bezier curve can be written as:

$$BZ(t) = \sum_{i=1}^n \binom{n}{i} t^i (1-t)^{n-i} P_i, 0 \leq t \leq 1, \quad (2.1)$$

Where n is the degree of the curve, and P_i are elements of $\mathbb{R}^k, k \leq n$, called bezier points. If $P_i = (P_{ix}, P_{iy}, P_{iz}) \in \mathbb{R}^3, 0 \leq i \leq n$, then we have:

$$BZ(t) = \sum_{i=1}^n B_i^n(t) (P_{ix}, P_{iy}, P_{iz}) \quad (2.2)$$

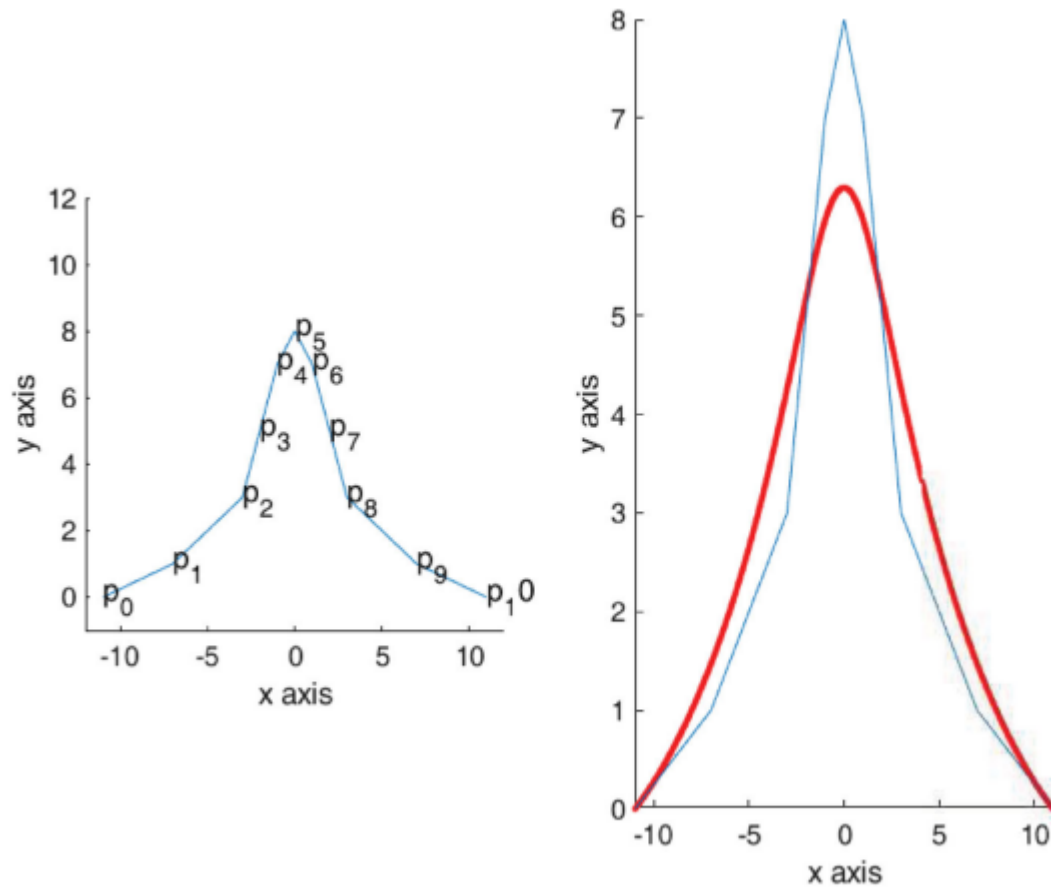


Figure 2.2: Bezier curve according to the control points [19]

2.5 Related Work

We were not able to find work related to parallel programming edutainment games. However, we found work related to an edutainment game in which visual programming was utilized. As well as a visual programming language.

2.5.1 Human Resource Machine

This is a puzzle-type game where you program an office worker to do tasks given by the game. The game takes advantage of visual programming to automate the office worker's tasks. The game starts with only 2 commands available, gradually increasing the game's difficulty as well as the number of commands available after levels are completed. "Commands" are programming functions in which you drag and drop in the game [20]. "The entire language contains only 11 total commands - but they're enough to simulate almost any computer algorithm in the world!" [20].

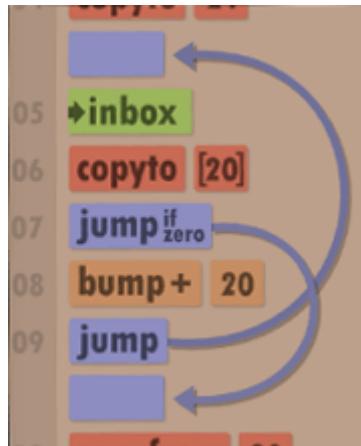


Figure 2.3: The visual programming inside the game [20]

2.5.2 Scratch

Scratch began its development in 2003 and was released in 2007. Their inspiration came from the fact that no tools were able to offer an easy way for children to create interactive animations, digital stories, and games. Now, scratch is the world's largest coding community for children, allowing for the creation of these digital stories, animations, and games by utilizing a visual programming language with a simple interface [21, 22].

"Scratch promotes computational thinking and problem-solving skills; creative teaching and learning; self-expression and collaboration; and equity in computing" [22].

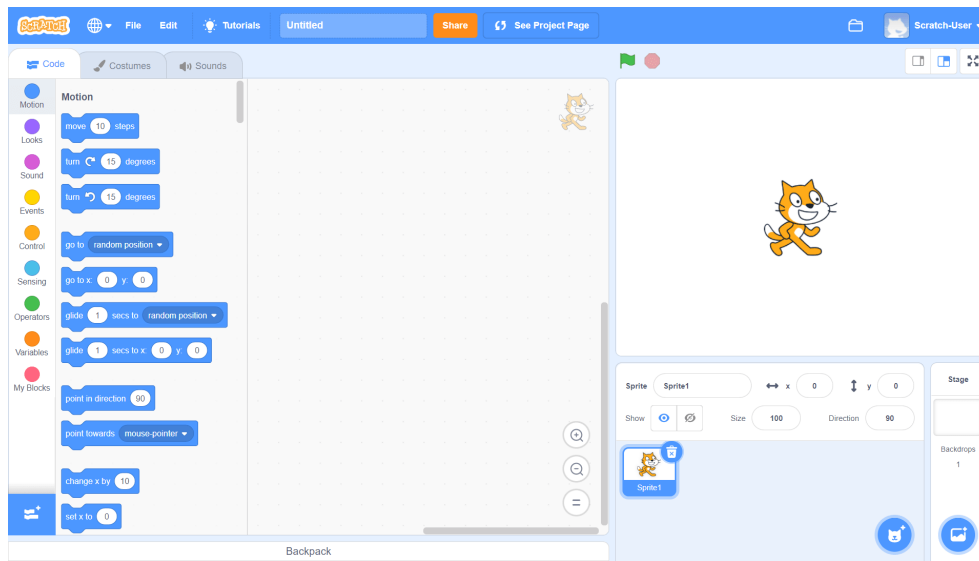


Figure 2.4: Scratch's user interface [21]

Chapter 3

Developing an MPI Edutainment Game

This chapter will present what game engine was chosen to develop the game, a description of the game, and how the edutainment game was implemented, with the reasoning behind the choices we made. Including possible encounters with pitfalls. Not every part of the implementation is covered, only the most relevant and important aspects.

3.1 Choosing a Game Engine (RQ1)

As previously stated in Section 2.2, using a game engine is essential for finishing a game in such a short amount of time. It is also critical to select the *right* game engine, as Abraham Maslow stated:

"If the only tool you have is a hammer, you tend to see every problem as a nail."

According to the quotation, different problems necessitate different tools. The two most popular gaming engines, Unity and Unreal Engine, will be compared and one of them will be chosen as the development platform. They are quite similar, however, there are some significant distinctions between them. In terms of visuals, Unreal Engine has the upper hand because it comes with a lot of capabilities out of the box, such as post-processing and volumetric fog. To get achieve a similar result in Unity, developers must install third-party libraries, which takes more time [14].

Both game engines have out-of-the-box support for networking, but Unreal Engine's networking is far superior. Unity has the upper hand for creating 2D games, while Unreal Engine has the upper hand in the field of AI. Unreal Engine supports only C++, whereas Unity supports C#. Unity has far more user-created content with Unity's asset store than Unreal Engine has with UE Marketplace. Thus, it is easier to find assets that you require in Unity's asset store [14].

The author chose Unity, which may seem counter-intuitive given that Unreal Engine outperforms Unity in many areas. The author's experience with Unity was a major factor in our decision, as the author has been developing games in Unity for 9 years and only 1 month in Unreal Engine. Not having to learn the inner workings of a game engine saves a lot of time. Additionally, C# is the language the author is most comfortable with, and the game will not be using a network solution. Finally, the game does not need groundbreaking graphics and AI.

3.2 Description of the Game

To create a context for the sections below, a description of the game is presented. First of all, the game has a *controlboard*, where the player can build a program using visual programming. In the levels, there are robots that simulate processes. They gather the controls (functions) that have been visually programmed from the controlboard and start interpreting them. The robots also interact with several different game elements; the goal of the levels is to make the robots transport out *crates*, *barrels* and *cubes*, depending on the current level the player is in. This is done by placing them onto a *transformer*. However, the *transformer's* color must match the *crate* for instance, to be able to put them onto the *transformer*. Game-elements such as the *crate* can also be put onto a *transformer*, to change its color to white. Furthermore, putting a *crate* on a *inverter*, inverts the *crate's* color. The *transformer* also has a color, similar to the *transformer*, representing that it only can receive game elements that match its color.

To be able to complete levels, the player will have to make use of MPI -and traditional programming with a visual programming interface to make robots communicate with each other, and also use different game elements in the levels to match the game elements' color to the *transformer's* color. The player must complete 12 different levels, each of which increases in difficulty. There are 2 unique levels that portray simplified versions of image manipulation and border exchange.

Every level unlocks new controls, arguments, and sometimes new objects. When a level has been completed by the player, the player receives 0 to 3 stars based on how optimized the program is. Additionally, run-time and compilation exceptions can occur, which are presented in a console with a description of the error. An interactive tutorial is present in the game, guiding players through the early levels, explaining the goal of the game and the various game mechanics.

The following sections describe in more detail the implementation choices we made.

3.3 Game Models and Licensing

All game models in the game are downloaded from Unity's asset store. These include the license type: *Extension Asset*, and falls under the license agreement: *Standard Unity Asset Store EULA*¹.

3.4 Timing

In every part of the game where game elements move, one must make use of `Time.deltaTime`². This makes sure that the operation does not move faster or slower depending on the frame rate of the game. This is often referred to as *frame-independence*.

3.5 Visual Programming

Given that the game provides an environment for learning parallel programming, it was reasonable to include some type of programming for the player to make use of. We looked into creating one's own programming language and using it to program in the game. However, the idea was quickly abandoned because it would be time-consuming and we had no prior experience with language construction. We began considering various methods of programming within the game and chose visual programming. The Human Resource Machine game described in Section 2.5.1 became a driving force behind the implementation of this sort of programming. Additionally, as described in Section 2.1.2, a game environment should be balanced between **scripting** and **emergence**. With players programming in the game, they can complete a level however they want. This induces the **emergent** gameplay, while still having a defined goal in place, preserving the **scripting** part of the environment.

3.5.1 Controlboard

`Controlboard.cs` handles the spawning of arguments and controls. It retrieves the controls from a JSON file called `controls.json`. Arguments is retrieved from `arguments.json`. At the start of a level, all arguments and controls reside in a *side-panel*, where the player can drag and drop them into an *active-panel* to start building a program.

The system uses a `JSONUtility`³ class with official support from Unity. The class converts Unity objects to and from JSON format. To deserialize and serialize lists and arrays, a "wrapper" is needed. The wrapper is serialized instead of serializing the list or array directly.

¹https://unity3d.com/legal/as_terms

²The interval in seconds from the last frame to the current one. <https://docs.unity3d.com/ScriptReference/Time-deltaTime.html>

³<https://docs.unity3d.com/Manual/JSONSerialization.html>

Listing 3.1: Wrapper for controls

```
[System.Serializable]
public class ControlsWrapper
{
    public Control[] controls;
}
```

Listing 3.2: Functions for spawning controls and arguments in Controlboard.cs

```
public GameObject InstantiateControl(Control c, Transform parent, bool isActive)

public GameObject InstantiateArgument(Argument argument, bool isActive,
Transform parent = null, int index = -1,
bool placeholder = false)
```

The *isActive* parameter denotes whether the argument/control has been dragged and dropped into the active-panel or not. Arguments and controls can have three different parents; one for its side-panel field where you select it from. Another one for when they are being dragged, and need to have a parent that does not clip their image. The last parent is where it is active in the active-panel. Some controls can have several arguments, the *index* parameter represents which sibling index it should have. The last parameter *placeholder* represents if the argument is being dragged. As the functions are relatively long, the content of the functions can be found here: Listing B.1 and B.2.

3.5.2 Visual Programming Architecture

The overall architecture for the visual programming used in our game design is illustrated in Figure 3.1.

As shown, the controls are functions for the robots to simulate. Similar to real programming, some functions have parameters. There are several classes defining which type of control or argument it is. Depending on if it is an argument or a control it uses either *ArgumentDraggable.cs* or *ControlDraggable.cs* respectively; both inherit from the superclass *Draggable.cs*.

If the control has the class *InActiveControlDraggable.cs*, it resides in the side-panel, ready to be dragged and dropped into the active-panel. However, if it has the class *ActiveControlDraggable.cs*, it is in the active-panel. Both *ArgumentControlDraggable.cs* and *BracketControlDraggable.cs* inherit from *ActiveControlDraggable.cs*. The first one indicates that the control has available parameters, while the other one simulates a bracket. *ArgumentControlWithBracketDraggable.cs* inherits from *ArgumentControlDraggable.cs* and specifies that the control should have a bracket. The last one, *ElseControlDraggable.cs* simulates an *else* in real programming, and inherits from *ArgumentControlWithBracketDraggable.cs*. These classes makes it possible to be able to drag and drop the arguments/controls. For arguments, the architecture is simpler. It is in the side-panel if it has *InActiveArgumentDraggable.cs* or *ActiveArgumentDraggable.cs* if it resides in the active-panel.

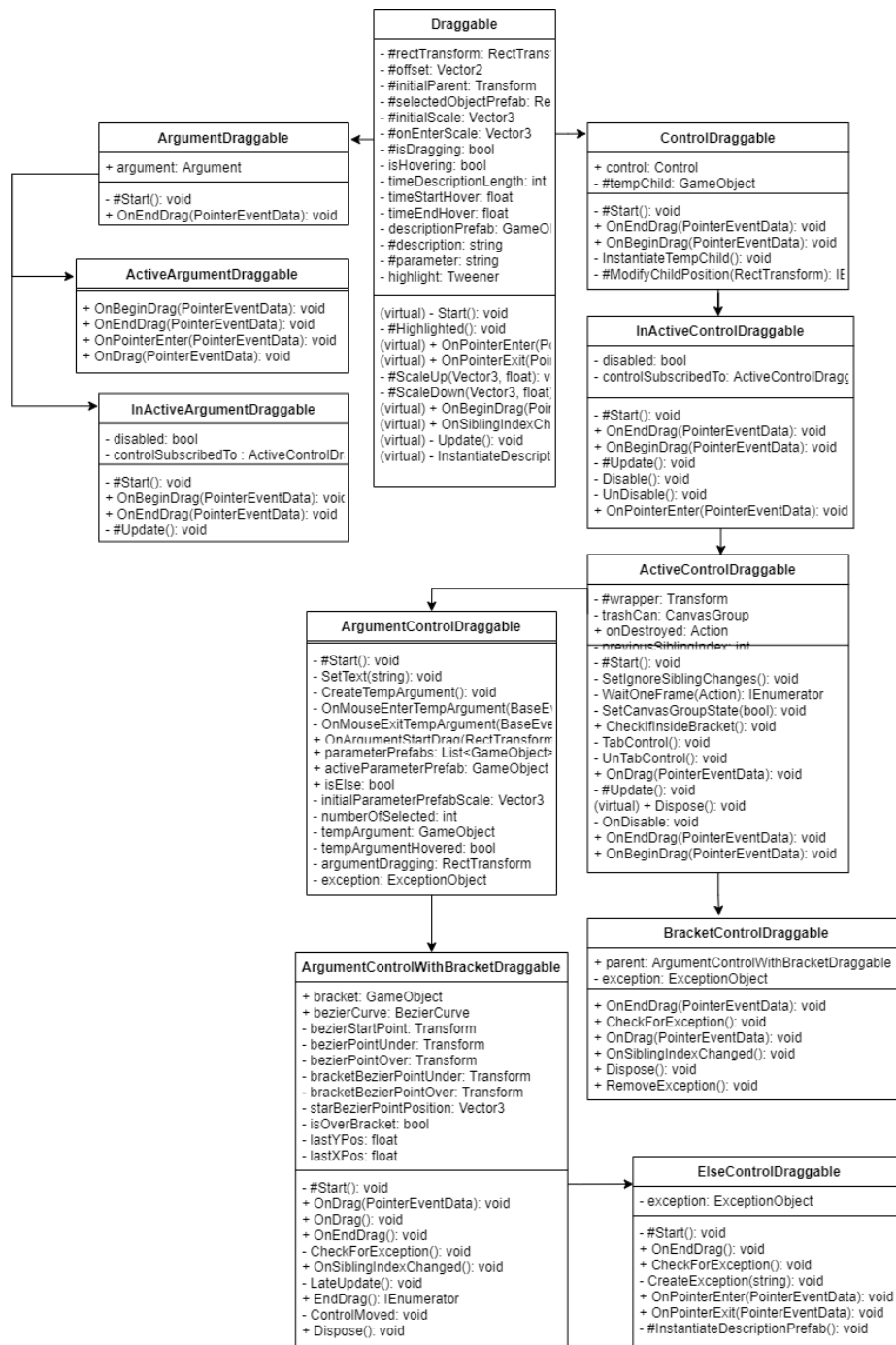


Figure 3.1: Architecture for controls and arguments

3.5.3 Development Process

A hurdle we encountered was when the very first control was developed, called: Action. It enabled a robot to interact with a game element. When the player picked the Action control an argument-panel opened with options to what game element to interact with. For example, if each robot had a crate-object in its local memory, there would be *four* crate-objects displayed when the argument panel appeared. In a parallel setting, having four unique options for the same object type did not make sense to us. Therefore, we changed it to name matching; if the object's name matches, only one object should be visible.

If Sentence

In a real MPI program, processes must frequently be distinguished, necessitating the use of if statements. The game required an If control, to be able to simulate an if sentence, brackets had to be simulated. By using an End control, players could drag it where the If control should end. Controls between the If and End controls were now only executed by processes for which the If control is true. Additionally, if there are several If controls, one would need to differentiate which End control is to which If control. The Else control, placed beneath an End control, works in the same way, simulating whether an If sentence has an Else. Bezier curves as described in Section 2.4 was utilized to connect the End and the If controls; it was done by creating a lot of small red lines at every point in the bezier curve.

The CalculateCubicBezierPoint function Listing 2.2, uses the equation from Section 2.3.4. MakeLine function creates a small red line, and the SetRectTransform function sets the previous created line to its correct position and rotation. The functions can be found here: Listing B.5, B.6 and B.7.

Listing 3.3: Creation of a bezier curve

```

public void DrawCurve()
{
    Vector3 lastPos = controlPoints[0].position;
    for (int j = 0; j < curveCount; j++)
    {
        for (int i = 1; i <= segmentCount; i++)
        {
            float t = i / (float)segmentCount;
            int nodeIndex = j * 3;
            Vector3 pixel = CalculateCubicBezierPoint(t, controlPoints
            [nodeIndex].position, controlPoints [nodeIndex + 1].position,
            controlPoints [nodeIndex + 2].position, controlPoints
            [nodeIndex + 3].position);
            if(lines.Count != segmentCount)
            {
                lines.Add(MakeLine(lastPos.x, lastPos.y, pixel.x, pixel.y,
                Color.red).gameObject);
            }
            else{
                SetRectTransform(lines[i - 1].GetComponent<RectTransform>(),
                lastPos.x, lastPos.y, pixel.x, pixel.y);
            }
            lastPos = pixel;
        }
    }
}

```

3.6 Simulation of Processes

Processes are simulated and represented in the game as robots, using the class `Worker.cs`. Their rank is identified by an integer floating above them. Every robot is placed in its own cubicle, representing its local memory. If a robot attempts to interact with an object outside of its cubicle, the game will be paused and a null-reference exception will be printed. When pressing "Play", the robots gather the controls and arguments into their local memory and start iterating over them.

3.6.1 Development Process

One of the first obstacles we had to solve was how one could limit game elements to robots (processes). As MPI does not deliver shared memory between processes, every robot should not have access to all game elements in the game. For instance, the root robot (process 0), should only be able to send its game element, because the game element resides only in the root robot's cubicle.

This was solved by having the robots put all of its game-element children with the class `Interactable.cs` in a local array called `interactables`. This way, it was easy to distinguish what game element should be limited to what robot.

Listing 3.4: Robot iterating over controls

```

private void NextIteration()
{
    currentIteration++;

    if(currentIteration >= controls.Count || runTimeError || mpiFinalized)
        return;

    var currentControl = controls[currentIteration];

    if(currentControl.type == Control.ControlType.Action)
    {
        new ActionInterpreter(currentControl, this, interactables.ToArray(),
            NextIteration);
    }
    else if(currentControl.type == Control.ControlType.Logic)
    {
        new LogicInterpreter(currentControl, this, NextIteration, ModifyIteration,
            lastIfSentenceFalse);
    }
    else if(currentControl.type == Control.ControlType.MPI)
    {
        new MPIInterpreter(currentControl, this, NextIteration);
    }

    lastIfSentenceFalse = false;
}

```

When the game starts, robots retrieve all of the controls visually programmed and interpret them. They are interpreted by one of three classes: `ActionInterpreter.cs`, `LogicInterpreter.cs` and `MPIInterpreter.cs`. All of these inherit from the superclass `Interpreter.cs`. The robot passes a `System.Action`⁴ delegate to each interpreter; when it is invoked, the robot continues its iteration over the controls.

3.6.2 Simulation of Functions

The following subsection will cover and explain how the different controls are interpreted.

Action-Interpreter

The `ActionInterpreter.cs` class is responsible for interpreting object-interactive functions. It receives the robot's interactables array and uses the control and the interactables array to determine if the argument name provided to an Action control matches an object in the interactables array; if it does not, the robot generates a run-time error and exits. Depending on the control name, it can run several functions.

⁴<https://docs.microsoft.com/en-us/dotnet/api/system.action-1?view=net-6.0>

Listing 3.5: Using the control name to determine which function to run

```

switch(control.Name)
{
    case "Action":
        Action();
        break;

    case "GoTo":
        GoTo();
        break;

    case "Disassemble":
        Disassemble();
        break;

    case "Assemble":
        Assemble();
        break;
}

```

Listing 3.6: Matching an argument name to an object

```

var selection = control.arguments[0].name.ToLower();

var interactableObject = interactables.
FirstOrDefault(x => x.Name.ToLower() == selection);

if(interactableObject == null)
{
    ExceptionHandler._instance.
    RuntimeError("Can't find object " + selection + ".", caller,
    ExceptionObject.ExceptionType.NullReferenceException);

    Controlboard._instance.nextTutorialIteration?.
    Invoke(LevelTutorial.UserAction.ActionError);
    return;
}

```

Logic-Interpreter

LogicInterpreter.cs interprets logic controls: If and End controls. It determines which function to run similarly to ActionInterpreter.cs. The class takes two actions, onTrue and onFalse to determine if the If or End control was true or false. An If control can have endless arguments, therefore a recursive function IfSentence(List<Argument> arguments) was created to check if the If control is true or false for a robot. The entire function can be found here: Listing B.3.

Creating a Simulation of an If Sentence

The If control was checked for any faults by the ArgumentControlDraggable.cs before the level was started, for example, if the If control had less than three arguments. Alternatively, if the control's first parameter is an operator. An If sentence

can be incorrect in a variety of ways. Because *ArgumentControlDraggable.cs* examines the If control, there is no need for any checks in the *LogicInterpreter.cs* class.

The first argument for an If control in this context is always a variable or an integer value.

Listing 3.7: Extraction of the three first arguments

```
var variableOrInteger = arguments[0];
var argumentOperator = arguments[1];
var variableOrInteger2 = arguments[2];

var variableOrIntegerValue = variableOrInteger.argumentType ==
Argument.ArgumentType.Variable ?
caller.GetType().GetField(variableOrInteger.name.ToLower()).GetValue(caller).
ToString() : variableOrInteger.name;

var variableOrIntegerValue2 = variableOrInteger2.argumentType ==
Argument.ArgumentType.Variable ?
caller.GetType().GetField(variableOrInteger2.name.ToLower()).GetValue(caller).
ToString() : variableOrInteger2.name;

arguments.RemoveRange(0, 3);

bool sentenceTrue = false;
```

The first variable or integer is extracted first, followed by the operator, and finally the variable or integer. The initial half of the If control has now been retrieved. Then, the list removes these arguments. If there are no more arguments left, we have extracted everything and can check whether the sentence is true or false.

Listing 3.8: Checking whether the sentence is true or false

```
private bool CheckSentenceTrue(string arg, string value1, string value2)
{
    switch(arg)
    {
        case "==":
            return value1 == value2;
        case "!=":
            return value1 != value2;
        case ">":
            return int.Parse(value1) > int.Parse(value2);
        case "<":
            return int.Parse(value1) < int.Parse(value2);
        case ">=":
            return int.Parse(value1) >= int.Parse(value2);
        case "<=":
            return int.Parse(value1) <= int.Parse(value2);
        default:
            return false;
    }
}
```

However, if there are more arguments, we must determine which operator is being used. If the operator is a modulo operator, we must calculate the value

already extracted with the modulo operator. We also need to figure out the operator to use to check the modulo value, as well as which variable or integer to use. Furthermore, the arguments must be removed from the list.

Listing 3.9: Checking whether the sentence is true or false with modulo

```

if(argumentOperator.name == "%")
{
    var moduloValue = int.Parse(variableOrIntegerValue) %
    int.Parse(variableOrIntegerValue2);

    var operatorValue = arguments[0].name;
    var lastArgValue = arguments[1].name;

    arguments.RemoveRange(0, 2);

    sentenceTrue = CheckSentenceTrue(operatorValue, moduloValue.ToString(),
    lastArgValue);
}

```

If there are still more arguments imminent, it is guaranteed that it is not a modulo operator. We will have to check whether it is an OR or a AND operator. If the sentence was true already, and the next operator is an OR operator, we can invoke the onTrue action and exit. However, if the operator is an AND operator, we have to check whether that sentence is true, calling the function again - with the modified list.

Listing 3.10: More arguments imminent and the sentence was true

```

if(sentenceTrue && arguments[0].argumentType == Argument.ArgumentType.Operator)
{
    if(arguments[0].name.Equals("AND"))
    {
        arguments.RemoveAt(0);
        IfSentence(arguments);
    }
    else if(arguments[0].name.Equals("OR"))
        onTrue?.Invoke();
}

```

If, on the other hand, the sentence was false and the next operator is an OR operator, we have to check if that sentence is true. But, if the operator is a AND operator, we can call onFalse, because there is no need to check if it is true since both sentences would have to be true. When calling onFalse, we have to pass how many iterations the robot has to skip.

Listing 3.11: More arguments imminent and the sentence was false

```

else if(arguments[0].name.Equals("OR"))
{
    arguments.RemoveAt(0);
    IfSentence(arguments);
}
else
onFalse?.Invoke(Mathf.Abs((control.bracket.EndIndex - control.bracket.StartIndex)));

```

Creating the Else control logic is much simpler; it checks whether the last If control was true or false. If it was true, it invokes onFalse, onTrue otherwise.

Listing 3.12: Else function

```
private void Else()
{
    if(lastIfSentenceFalse)
    {
        onTrue?.Invoke();
    }
    else
    {
        onFalse?.Invoke(control.bracket.EndIndex - control.bracket.StartIndex);
    }
}
```

3.6.3 MPI-Interpreter (RQ2)

`MPIInterpreter.cs` simulates all of the MPI functions described in Section 2.3.2. The motivation for choosing these is that the author is most confident with these functions, also it seems that these functions are the most used and basic when starting to learn MPI. Additionally, Anne C. Elster uses these functions to teach MPI in her parallel computing subject and the book [1] uses these functions to get started with MPI.

To simulate these functions, several classes are used. A `GatherHandler.cs`, `BroadcastHandler.cs` and a `SendReceiveHandler.cs`. They all derive from the superclass `MPIHandler.cs`. Most of the MPI functions contain several parameters: these are abstracted out, and only the most relevant ones are kept: *sender*, *root* and *dest*. This means that, for instance, `MPI_Send` only has one parameter, the receiver. This is to avoid players feeling overwhelmed and is not essential to be able to understand the logic of parallel programming/MPI. Additionally, when sending the game elements, animations are used to give visual appeal and to simulate communication overhead. These animations also use a bezier curve to achieve this. To be able to animate something over several frames coroutines⁵ are essential to achieve this.

For the animations, first we declare some controlpoints. We choose the start point where the game-element is now, the second control point we set as 25% to the destination on the xz plane, while adding 5 to its y value. Same with the third control point, but inverted. Finally, the end point is at the hands of the robot receiving.

⁵Coroutines are excellent when modeling behavior over several frames. <https://docs.unity3d.com/ScriptReference/MonoBehaviour.StartCoroutine.html>

Listing 3.13: Declaring control points

```

Vector3[] controlPoints = {
    startPos,
    Vector3.Lerp(startPos, pickupPos.transform.position, 0.25f) + Vector3.up * 5f,
    Vector3.Lerp(pickupPos.transform.position, startPos, 0.25f) + Vector3.up * 5f,
    pickupPos.transform.position
};

```

This is then used in the `BezierCurve.cs` class, as shown in Listing 3.3. However, a function was created with parameters defining how many segments to create, while also only returning the points and not actually creating a curve with red lines. We calculated and retrieved three bezier points. Since it is only three points, we know that the second one is at the highest point, and in the middle. Also, we declared some variables, `sendTimeT` and `velocity`.

Listing 3.14: Retrieving bezier points, the highest point of the curve and declaring some variables

```

Vector3[] bezierCurvePoints = new BezierCurve().
CreateBezierCurvePoints(controlPoints, 3);

Vector3 highestPoint = bezierCurvePoints[1];

float sendTimeT = 0;
float velocity = 1;

```

Listing 3.15: Animating the game-element to the highest point, and then to its final destination

```

while(sendTimeT < 1)
{
    sendTimeT += velocity * Time.deltaTime;

    velocity = Mathf.Lerp(3, 1, sendTimeT);

    transferObject.transform.position = Vector3.Lerp(startPos, highestPoint,
    sendTimeT);

    yield return null;
}

sendTimeT = 0;

while(sendTimeT < 1)
{
    sendTimeT += velocity * Time.deltaTime;

    velocity = Mathf.Lerp(1, 3, sendTimeT);

    transferObject.transform.position = Vector3.Lerp(highestPoint,
    pickupPos.transform.position, sendTimeT);

    yield return null;
}

```

These variables are used to set the velocity of the animation, and also to know at all times how far along one is to the destination. Then the `sendTimeT` float is

added by the velocity times `Time.deltaTime`, to make sure the animation is frame-independent. Furthermore, the velocity is changed so that the animation is fastest at the start and the end. The game element is first animated to the highest point, and then to the final destination.

Simulation of `MPI_Send` and `MPI_Recv`

To accomplish a simulation of `MPI_Send` and `MPI_Recv`, a list of `SendReceiveHandler.cs` called `sendReceiveHandlers` is declared as a static field, because several send/receive can occur in the game simultaneously. For a robot sending something, we have to extract out who is the receiver of the send in the `EnterSend` function.

Listing 3.16: Extracting out the receiver

```
var receiver = ConvertFromArgumentToInt(control.arguments[0]);
```

Furthermore, we will have to check if there is a `sendReceiveHandler` already with the same receiver and same sender. This is because the robot who is on the receiving end could enter the function `EnterReceive` before the sender, creating the handler. If it is not created - create a handler, and declare the receiver and sender. Further, we subscribe to a `onExecute` action, and increment the number of ready workers. When the number of ready workers hit 2, the action will be invoked and the function `ExecuteSendAndReceiveOwner` will be executed. In this function, a dictionary of workers along with its `onFinished` action is supplied as an argument, to be able to manipulate other robots.

Listing 3.17: Create handler if handler is null, then subscribe to action and increment number of ready workers

```
var handler = MPIInterpreter.sendAndReceiveHandler.
FirstOrDefault(x => x.receiver.Equals(receiver)
&&
x.sender.Equals(caller.rank));

if(handler == null)
{
    MPIInterpreter.sendAndReceiveHandler.Add(new SendAndReceiveHandler());
    handler = MPIInterpreter.sendAndReceiveHandler.Last();

    handler.receiver = receiver;
    handler.sender = caller.rank;
}

handler.onExecute += ExecuteSendAndReceiveOwner;
handler.ReadyWorkers++;
```

The `EnterReceive` function calls a function `WorkerReady` in the `SendAndReceiveHandler` object which increments the `WorkersReady` integer, and adds the robot to a dictionary along with its `onFinished` action.

Listing 3.18: Function in MPIHandler.cs

```
public void WorkerReady(Worker worker, Action onFinished)
{
    workersReady.Add(worker, onFinished);
    ReadyWorkers++;
}
```

Simulation of MPI_Bcast

MPI_Bcast only need a single function for its simulation: EnterBroadcast, and one static field: broadcastHandler. First, we have to check if the handler is null, if it is, create a new BroadcastHandler object.

Listing 3.19: Creating a new BroadcastHandler object

```
if(MPIInterpreter.broadcastHandler == null)
    MPIInterpreter.broadcastHandler = new BroadcastHandler();
```

An *MPI_Bcast* has one sender, which is denoted as the owner in this context. So, we have to check if the robot is the owner the *MPI_Bcast*. If it is the owner, we subscribe to the onExecute action, and increment the readyWorkers integer. However, if it is not the owner, we only call the function: Listing 3.18.

Listing 3.20: If owner of *MPI_Bcast*, subscribe to action and increment, otherwise call the function WorkerReady

```
if(ConvertFromArgumentToInt(control.arguments[0]).Equals(caller.rank))
{
    if(caller.PickedUpObject == null)
    {
        ExceptionHandler._instance.RuntimeError("Does not have an item picked up.",
        caller, ExceptionObject.ExceptionType.RuntimeError);
        return;
    }

    MPIInterpreter.broadcastHandler.onExecute += ExecuteBroadcastOwner;
    MPIInterpreter.broadcastHandler.ReadyWorkers++;

    return;
}

if(caller.PickedUpObject != null)
{
    ExceptionHandler._instance.RuntimeError("Does have an item picked up.", caller,
    ExceptionObject.ExceptionType.RuntimeError);
    return;
}

MPIInterpreter.broadcastHandler.WorkerReady(caller, onFinished);
```

MPI_Gather and *MPI_Scatter* are very similar to how they were implemented in regards to *MPI_Bcast*, so these will not be explained.

Development process

The very first MPI function simulated in the game was *MPI_Send* and *MPI_Recv*. In a real MPI program, when sending a message, the process will still have the message data in its local memory. It was originally simulated exactly as the MPI function, the robot sends an object and another robot receives it, and the sender still had the object in its local memory. This was not intuitive behavior for the game, therefore it was changed so that the sender removes it from its local memory. The sender robot needs to have the object picked up to be able to send it, and the receiver must have no object picked up.

MPI_Bcast was simulated next. It broadcasts a game element to all other processes, and the sender still keeps it in its local memory. The most common usage of *MPI_Bcast* is to send configuration parameters or user input to other processes. In the fall project, the broadcast served no real purpose except for demonstrating its functionality to the player. To make actual use of such a function, the game would have to include an object that can be duplicated and still be relevant in the game. As a result, our solution was to construct a game element that activates a game element while yet allowing the broadcaster to keep the game element in its local memory. *MPI_Scatter* and *MPI_Gather* behaves the same as *MPI_Bcast* in the game, however, it sends chunks of objects.

We also encountered a pitfall regarding the time it took to complete a level - the time was not always consistent. Even though the program was the exact same. Our first thought was that it could have something to do with the rounding up to the first decimal. However, the problem still occurred. Then we started looking at all of the animations, are they all frame-independent? This problem became clear as day when we used a slower machine to run the game. The animation to send game elements between robots took much longer to complete. Our first animation to send game elements was created by still using a bezier curve, but with a lot more points, to create a smooth curve. First, we created a for loop, looping every point in the curve, then we interpolated between the points with a nested while loop. However, this method was not only causing the bad performance, but it was also frame-dependant. As frames per second (fps) is not always consistent during runtime, the smallest change in fps can affect the time an animation takes to complete.

3.7 Interactable Objects

Interactable objects are game elements that can be interacted with by robots. All interactable objects derive from the `Interactable.cs` superclass. Game-elements that can be picked up use the class `InteractablePickup.cs`. Other game element classes are more tailored to a single item. For instance, `InteractableInverter.cs` is made to invert a `InteractablePickup.cs` object's color. By using a game element to do this inversion of color, we can abstract out code. This is done by hand in a genuine MPI program, however, creating such advanced visual pro-

gramming would be far too complex and time-intensive. Another abstraction of code is the `InteractableTransformer.cs` class. It transforms a game element's color to white.

The `Interpreter.cs` class has a virtual function to begin interaction with an object.

Listing 3.21: Start of interaction with object

```
public virtual void Interact(Action onFinished)
{
    this.onFinished = onFinished;

    StartCoroutine(worker.Move(transform.position, FinalInteraction,
        stoppingDistance));
}
```

It takes the `onFinished` action from the robot we have seen before as a parameter. Additionally, it tells the robot to move to itself. After the robot has reached its destination, the protected abstract void `FinalInteraction()` function is executed. This is where the derived classes can do individual logic.

3.8 Conversion from Visual Programming to Programming Language C

As C is mostly used for programming with MPI, it was reasonable to be able to see what the equivalent code would be in C. This was also a request from advisor Anne C. Elster. To be able to do this conversion, we would have to have the same syntax for both arguments and controls as C. Camel cases would also be needed. We created a `CodeName` property in both `Argument.cs` and `Control.cs`. It converts the control or argument's name to camel-case.

Listing 3.22: Conversion to camel-case

```
public string CodeName {
    get {
        if(overrideCodeName != null)
            return overrideCodeName;

        var splittedName = name.Split(' ');

        if(splittedName.Length == 1)
            return name.ToLower();
        else
        {
            var upperLastName = splittedName[1][0].ToString().ToUpper() +
                splittedName[1];

            return splittedName[0].ToLower() + char.ToUpper(splittedName[1][0]) +
                splittedName[1].Substring(1);
        }
    }
}
```

There may also be a need for overriding the argument or control's name when the syntax is different. For instance, in our game, the && operator is AND. Overriding it was done by setting the `overrideCodeName` variable to &&.

Listing 3.23: The AND operator in Arguments.json

```
{
  "name": "AND",
  "argumentType": 0,
  "levelAccess": 10,
  "description": "And operator",
  "overrideCodeName": "&&"
}
```

Because most MPI function parameters are not utilized but still have to be displayed in the conversion, we must ensure that the parameters we have are in the proper position for the functions. First, we add default parameters and select a communicator. We used the most used communicator: `MPI_COMM_WORLD`. For the parameters using a type, we always select `MPI_INT`. Lastly, we have to create an array of indices of where to put the arguments called `parameterIndex`.

Listing 3.24: Example of `MPI_Sendrecv` in controls.json

```
"name": "MPI_Sendrecv",
"type": 2,
"levelAccess": 9,
"description": "A combination of MPI_Send and MPI_Recv",
"parameter": "First parameter specifying the rank of the receiver,
second one specifying the rank of the sender",
"maxParameters": 2,
"parameterIndex": [3, 8],
"defaultParameters": [
  "&pickedUpObject",
  "1",
  "MPI_INT",
  "0",
  "&pickedUpObject",
  "1",
  "0",
  "MPI_COMM_WORLD",
  "MPI_STATUS_IGNORE"
],
"validArguments": [1, 3]
```

Now we have everything in place to do the conversion. The `ConvertToCode.cs` class is responsible for the conversion. For a control with no brackets, the `ControlWithNoBrackets` function is used. First, we create a new list containing the default parameters for the control. If the control has arguments, we loop through the `parameterIndex` array and insert the arguments at the appropriate indices.

Listing 3.25: Inserting arguments at appropriate indexes

```

List<string> controlParameters = new List<string>(control.defaultParameters);
if(control.arguments.Count > 0)
{
    int i = 0;
    if(control.parameterIndex != null)
    foreach(var index in control.parameterIndex)
    {
        if(index == controlParameters.Count - 1)
            controlParameters.Add(control.arguments[i].CodeName);
        else
            controlParameters.Insert(index, control.arguments[i].CodeName);

        i++;
    }
}

```

If the list contains more than one element, we can loop through the created list and add it to the code output with the appropriate color. However, if the list does not contain more than one element, it does not have default parameters, and we can add one argument to the code output. The code variable is a `StringBuilder`⁶.

Listing 3.26: Adding arguments to code output

```

if(controlParameters.Count > 1)
{
    for(int i = 0; i < controlParameters.Count; i++)
    {
        var parameter = controlParameters[i];

        if(i != 0)
            code.Append(", ");

        code.Append("<color=lightblue>");
        code.Append(parameter);
        code.Append("</color>");
    }
}
else if(control.arguments.Count > 0)
{
    code.Append("<color=lightblue>");
    code.Append(control.arguments[0].CodeName);
    code.Append("</color>");
}

```

For a control containing brackets i.e. `If control`. We use the `ControlWithBrackets` function. Here we have to add brackets as well as looping through its arguments and append them to code output.

⁶<https://docs.microsoft.com/en-us/dotnet/api/system.text.stringbuilder?view=net-6.0>

Listing 3.27: Conversion to C with an If control

```

private static void ControlWithBrackets(List<int> brackets, Control control)
{
    brackets.Add(Mathf.Abs((control.bracket.EndIndex - 1 -
        control.bracket.StartIndex)));

    for(int i = 0; i < control.arguments.Count; i++)
    {
        var arg = control.arguments[i];
        if(i != 0)
        {
            code.Append(" ");
        }

        code.Append("<color=lightblue>");
        code.Append(arg.CodeName);
        code.Append("</color>");
    }
}

```

3.9 Simulating MPI Programs

In the fall project, none of the levels were closely connected to standard MPI programs. Advisor Anne C. Elster suggested we try to simulate independent tasks, as well as collective tasks.

3.9.1 Black/White Inversion

To be able to do black/white inversion, it was vital to be able to abstract out the inversion as mentioned previously. *MPI_Scatter* and *MPI_Gather* was also essential to implement in order to make the inversion easier. Using *MPI_Scatter* while holding an object sends chunks of the object to the other robots. Every robot can then put its chunk in the Inverter, inverting the chunked object's color. Then when *MPI_Gather* is used, all the chunks are collected by the owner and are joined into the original object with its inverted color.

3.9.2 Border Exchange

This was a difficult subject. The initial concept we came up with was for the root robot to have a container that was half green and half magenta. When scattering the object, robots 0, 2, 4, and 6 would receive the green chunk, while the others would receive the magenta chunk. A Transformer object has a point-light on it, indicating which color it can invert. The robots receiving the green chunks would have a Transformer with a magenta point-light, and the opposite for the other robots. To solve this problem, robots would have to trade chunks with their neighbors. In a true MPI application including filtering-type image processing, however, the processes do not exchange their entire image with each other. This may mislead the players about the purpose of border exchange.

Our solution then became to create a 4x2 cube, made up of small cubes. Every small cube has a color, but its top is a different color. The 4x2 cube would have the class `InteractablePickupWithChunks.cs`. When scattering the cube, if the object is a `InteractablePickupWithChunks.cs` the small cubes being sent should have two different colors on it depending on the robot rank. One for the top part and one for its body part.

Listing 3.28: Small cubes being sent to robots in `Worker.cs`

```
foreach(var worker in workers)
{
    if(PickedUpObject is InteractablePickupWithChunks interactablePickupCube)
        scatterObject.GetComponent<InteractablePickup>().splittedColors =
            interactablePickupCube.SplittedColors[worker.Key.rank % 2];
    if(worker.Key != null)
        worker.Key.TransferAndPickupItem(scatterObject.gameObject, worker.Value,
            scale, PickedUpObject.transform.position);

    // The owner of the scatter
    if(PickedUpObject is InteractablePickupWithChunks interactablePickupCube2)
        scatterObject.GetComponent<InteractablePickup>().splittedColors =
            interactablePickupCube2.SplittedColors[rank % 2];
    RemovePickedUpItem(false);
    InstantiateAndPickupInteractable(scatterObject.gameObject);
    onFinish?.Invoke();
}
```

The goal is to transform the color of an object into white. To be able to do this, the cube chunk has to be the same color. After scattering, the small cubes are made up of two colors. We then created a new control called `Disassemble`, which disassembles the small cube if the robot has it in its hands, leaving the body part on the ground, and the top part in the robot's hands - simulating that the player extracts out a "border" of an image. Next, the player can exchange the top with its neighbor, retrieving the matching top for the body of the small cube. However, the body part is on the ground, therefore an `Assemble` control needed to be implemented. It assembles both the parts to reassemble the small cube. Now the small cube's color is ready to be transformed into white. Afterward, the small cubes can be gathered to the root robot. Additionally, robot 0's `Transformer's` point-light is both magenta and green. Representing that the level can be completed by only having robot 0 doing the processing, to show the difference in speed between the alternatives.

3.10 Handling of Exceptions

Exceptions occur when players have done something wrong in the game, and are vital to implement to give feedback to players about when this happens and how they can resolve the error. "Games should help players recognize, diagnose, and recover from errors" [5]

In regular programming, exceptions can occur. In C, if you write code that cannot be built, a compile exception arises when you try to build the program.

We wanted to be able to simulate run-time compilation where the visually programmed code is checked at every change for any exceptions, resulting in a better workflow, which allows the player to correct their code in the process. Exceptions are handled in `ExceptionHandler.cs`. To create an exception, an `ExceptionObject.cs` object is created.

Listing 3.29: Creation of an exception

```
public ExceptionObject CreateException(string description, Transform transform,
ExceptionObject.ExceptionType type)
{
    var exceptionPrefab = Instantiate(Resources.Load("Controls/Exception"),
transform) as GameObject;

    var exception = new ExceptionObject(description, exceptionPrefab, type);

    exceptions.Add(exception);

    UpdateConsole();

    exceptionPrefab.transform.SetSiblingIndex(transform.childCount - 2);

    return exception;
}
```

There are two types of exceptions available. A `NullReferenceException` arises when, for instance, a robot is not able to find an object in its cubicle, and a `CompilerException`, for when a control has invalid arguments. Checking to see if a control has invalid arguments is done in the function `CheckForException`, and can be found here Listing B.4. Looking at the function, the most work is done checking whether an `If` control has the correct arguments in the correct places.

When a real program is being run, sometimes a `NullReferenceException` is not enough to stop the program. Therefore, when a `NullReferenceException` occurs in the game it is paused and can be continued if the player wants to. However, the robot that created the exception will never be able to continue. Exceptions are always displayed in a console describing the error, along with the rank of the worker, so the player knows which robot received the exception.

3.11 Interactive tutorial

A tutorial implemented in the game was essential for players to progress through the game. Without a tutorial, players would most likely be clueless about how a level would be completed; Sweetser and Wyeth [5] reinforce that a game should be playable without external resources.

In the first levels, players are guided by the tutorial step by step. As the game progresses, the tutorial gradually allows the players to use their imagination to beat the levels. At the same time, it is essential to maintain the player's skill level, as mentioned in Section 2.1.1.

The logic of the tutorial resides in the `LevelTutorial.cs` super class. When players do the correct action, the tutorial iterates to the next step. However, as

each level has a different set of "correct actions", different classes, such as `Level1Tutorial.cs`, must be implemented, inheriting from `LevelTutorial.cs`. In order for a player to advance to the next iteration of the tutorial, the `Level1Tutorial.cs` overrides the `NextTutorialIteration` function from the `LevelTutorial.cs`.

Listing 3.30: Overridden function in `Level1Tutorial.cs`

```
protected override void NextTutorialIteration(LevelTutorial.UserAction action)
```

`LevelTutorial.UserAction` is an enumerable created in `LevelTutorial.cs` to be able to know if the player did the right action to iterate the tutorial.

Listing 3.31: `UserAction` enum in `LevelTutorial.cs`

```
public enum UserAction {
    Started,
    OpenControlBoard,
    Stop,
    Play,
    Pickup,
    SelectedParameter,
    PlacedActionControl,
    PlacedMPI_InitControl,
    PlacedMPI_FinalizeControl,
    PlacedMPI_BcastControl,
    PlacedIfControl,
    PlacedMPI_Comm_rankControl,
    ActionError
}
```

3.12 Local Game Save Data and Audio

Players' progress through the game is saved locally to their disk using `PlayerPrefs`⁷ - with official support from Unity. To check if a level has been completed we check if the key for the level, such as `Level 1` has been set. The value for the key is a float which represents the number of stars received by completing the level.

Listing 3.32: Setting a level button to be interactable or not in `MainMenu.cs`

```
levelButton.transform.Find("Panel").GetComponent<ClickableUI>().Interactable =
PlayerPrefs.HasKey("Level " + (i - 1));
```

As described in Section 2.1.1, background music can enhance the initial interest in the game. Players also receive feedback from dragging and dropping controls/arguments with a "clicking" sound. When completing a level, a sound will accompany the appearance of the stars. "Games should reward players with feedback on progress and success" [5].

To implement background music, some songs were downloaded from the asset store. A `BackgroundAudioBehaviour.cs` class was made to play, and loop, a song based on which level the player is in.

⁷<https://docs.unity3d.com/ScriptReference/PlayerPrefs.html>

Listing 3.33: Plays and loops a song based on level index

```

public void PlaySong(int index)
{
    if(loop != null)
        StopCoroutine(loop);

    if(index != 0)
        index = index % 3 + 1;

    audioSource.clip = audioClips[index];
    audioSource.Play();

    loop = StartCoroutine(LoopSong((int)audioSource.clip.length));
}

private IEnumerator LoopSong(int songLength)
{
    yield return new WaitForSeconds(songLength);

    PlaySong(songIndex);

    loop = StartCoroutine(LoopSong((int)audioSource.clip.length));
}

```

For sound effects, an `FXAudioBehaviour.cs` class was made. The class subscribes to a `System.Action` in `GameHandler.cs` called `PlaySound` with a function with the same name `PlaySound`. A `Sounds` enumerable was then created to distinguish what sound effect to play. To facilitate retrieval, the sounds are saved in a folder with the same name as those in the enum.

Listing 3.34: Subscribing to the `PlaySound` Action in `GameHandler.cs`

```

GameHandler._instance.PlaySound += PlaySound;

```

Listing 3.35: An enum to distinguish what sound effect to play

```

public enum Sounds {
    PlacedControl,
    DraggedControl,
    StarSound
}

```

Listing 3.36: Playing sound based on what enum

```

private void PlaySound(Sounds obj)
{
    var soundObject = new GameObject("Sound Object");
    var audioSource = soundObject.AddComponent<AudioSource>();
    audioSource.clip = Resources.Load("Sounds/" + obj.ToString()) as AudioClip;
    audioSource.volume = volume;
    audioSource.Play();

    Destroy(soundObject, 10);
}

```

With this architecture in place, a sound effect can easily be played by invoking the Action in `GameHandler.cs`.

Listing 3.37: Invoking the Action in GameHandler to play sound effect when a control/argument is dragged in Draggable.cs

```
public virtual void OnBeginDrag(PointerEventData data)
{
    GameHandler._instance.PlaySound?.Invoke(FXAudioBehaviour.Sounds.DraggedControl);
}
```


Chapter 4

User Study

To be able to answer the user study questions, we used a combined approach by utilizing both the qualitative and quantitative method. Rahi in his paper [23] explains how the quantitative method ignores the individual's emotions and feelings, in addition to the environmental context. For the qualitative method, Rahi explains that the feelings and emotions of a person are equally important to interpret. This approach is used when a researcher wants to observe or interpret an environment with the intent of developing a theory. The quantitative method was represented by a quiz and a survey, while the qualitative data constitutes the use of observation and interview. Eight participants were picked, within the age range of 18-34, and where half of them were programmers and none with experience with MPI. The following section will present the user study questions, and then information about interviews, surveys, quizzes, and our observations of the participants.

User Study Questions

USQ 1: *"Will the game have an effect on increasing interest in parallel programming?"*

USQ 2: *"Is the game too difficult?"*

USQ 3: *"What is the learning effect of playing the game?"*

USQ 4: *"What is the enjoyment effect of playing the game?"*

4.1 Interview

Interviews were done to gain insight into the players' perspectives on the game. Interviews are a type of qualitative user research that provide more information and detailed responses than, say, a survey. It also allows for the introduction of follow-up inquiries which may provide us with useful information on the user interface and the game's overall experience. Additionally, we may receive new and improved game ideas. The outcomes of USQ1, USQ2, USQ4, and partly USQ3

were explained using data from the interviews. The interview questions were as follows: Appendix C.2

4.2 Survey

A survey was conducted to acquire quantitative data. The questions were single-choice and allowed participants to choose an option using the *Likert's scale* from strongly disagree to strongly agree. Because surveys provide accurate information about people's opinions and behaviors, they may be utilized to make key game design and mechanics decisions. The survey data was used to explain the USQ1 and USQ4 outcomes. Survey questions can be found here: Appendix C.1.

4.3 Observation

Observing the players while they play the game offers valuable information on how they interact with the game. For example, the speed with which people comprehend and use the user interface might indicate how well it is developed. A player who is having difficulty with a gaming mechanism may be able to give important input on where changes might be made. This is a significant indicator of whether they understand the game or feel it is too difficult.

4.4 Quiz

An MPI quiz was the final research method. This was necessary to evaluate if the game had any effect on training the players on MPI and if they grasped the ideas so that we could explain USQ3. The quiz was taken both before and after playing the game to showcase how the game affected the players' knowledge. Appendix D contains the quiz questions. There are single-choice questions and multiple-choice questions, the participants must have every right answer in the multiple-choice to be able to receive a point for that question. There are 12 questions, which equals 12 points maximum. Half of the quiz contains text-based questions and the other half contains code-based questions. In the fall project, there were a lot fewer code-based questions than text-based questions. We observed that participants managed to get the correct answer to the text-based questions based on the name of the MPI functions. Therefore, more code-based questions are included in this user study.

4.5 Reflection on our User Study

We chose participants that were easily accessible to us, which could invoke a selection bias. According to [23] this can provoke "convenience sampling". This means

that participants could be biased towards us; not giving honest opinions on the game, ultimately skewing our results.

Moreover, the survey should perhaps have included more questions pinpointed to elements that have been researched to affect enjoyment, game difficulty, and interest to get more quantitative data for drawing conclusions. This means that to research, for instance, if the implementation of a high score had an effect on their motivation for the game; we could ask: "I felt more motivated by receiving a high score after completing a level". From this question, we could have concluded that a high score does or does not increase motivation.

Additionally, from looking at the interview and survey questions, we feel that the interview and survey questions occasionally were synonymous; the questions overlapped. Furthermore, the interview questions should have been better phrased to prevent yes or no responses. Although interview questions are useful for gathering qualitative data, they are ineffective when the responses are either yes or no.

Chapter 5

Results

An executable of the game can be found and downloaded from: <https://github.com/ancelster/MPI-Edutainment-Game/tree/executables>. It features executable for Windows, macOS, and Linux. A user guide for the game can be found in Appendix A. This chapter will present both the results from the development of the game and also from the user study.

5.1 Development Environment

The game is developed and built using Unity 2020.3.18f1. The source code for the game can be found and downloaded from: <https://github.com/ancelster/MPI-Edutainment-Game/tree/main>, and must be opened in the Unity version stated. Minimum specifications to develop the game can be found in Table 5.1. Specifications used for the development of the game:

- **OS:** Windows 11 Home
- **CPU:** Intel(R) Core(TM) i5-8600K CPU @ 3.60GHz
- **GPU:** NVIDIA GeForce GTX 1060 6GB
- **RAM:** 16,0 GB

5.2 Recommended Specifications for Running the Game

The game was tested on our machine, which was used for the development of the game, running smoothly at over 300 fps. It has also been run on CPUs with integrated GPU, however, running slightly under 60 fps. This was on Windows 10, 8.0 GB RAM, and i5 8250U @ 3.40 GHz CPU. Furthermore, it was also tested on a machine using NVIDIA GeForce GTX 760 as the GPU. Where the CPU was Intel(R) Core(TM) i5-3770K CPU @ 3.90GHz, with 8,0 GB of RAM. The game appears to consume just about 750 MB of RAM, where it operated well at roughly 100 frames per second. As a result, the recommended specifications should be somewhere in the middle, with at least 4,0 GB of RAM.

Table 5.1: Minimum Specifications for development of the game, based on Unity's own website. For more information: [Link](#)

	Windows	macOS	Linux
Operating system version	Windows 7 (SP1+), 10, 11, 64-bit versions only	High Sierra 10.13+	Ubuntu 16.04, Ubuntu 18.04, and CentOS 7
CPU	X64 with SSE2 instruction support	X64 with SSE2 instruction support	X64 with SSE2 instruction support
Graphics API	DX10, DX11, and DX12-capable GPUs	Metal-capable Intel and AMD GPUs	OpenGL 3.2+ or Vulkan-capable, NVIDIA and AMD GPUs.
Additional requirements	Hardware vendor officially supported drivers	Apple officially supported drivers	X11 Windowing system and official NVIDIA -and AMD Mesa graphics driver.

5.3 Development Results

The following subsections will provide the game's outcomes, along with images of the game parts that were implemented. The images will then be compared to those from the fall project to see how they differ.

5.3.1 Game Main Menu

Players can now delete their game save data in the current project. The settings may also be accessed from the main menu. The players in our fall project were unable to change anything about the game, hence there were no available options. In addition, the UI style has been altered.



Figure 5.1: Main menu in our fall project

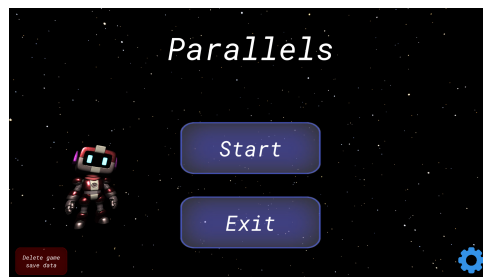


Figure 5.2: Main menu in our current project

5.3.2 Settings

The current project's settings panel may be seen here. Players may toggle between full screen and no full screen, modify the game's quality, and adjust the music and effects loudness.

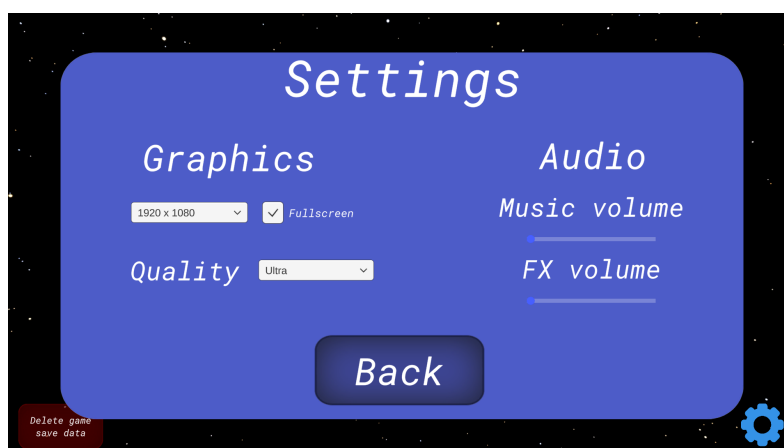


Figure 5.3: Settings

5.3.3 Level Selector

This user interface shows all of the game's levels. Players in the fall project did not obtain stars for their level performance, which are now visible in the level picker. Additionally, a screenshot is provided for every level and the user interface changed.

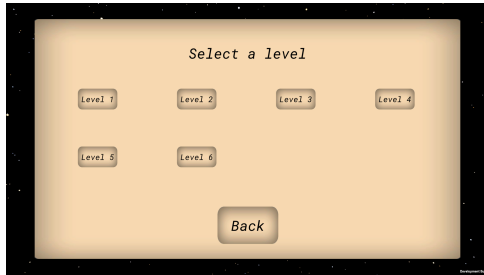


Figure 5.4: Old level selector



Figure 5.5: New level selector

5.3.4 Level Overview

Both versions display their last level. The current project has more objects and robots visible, and the console has been scaled down to allow a better perspective of the level.



Figure 5.6: Level 6 in our fall project

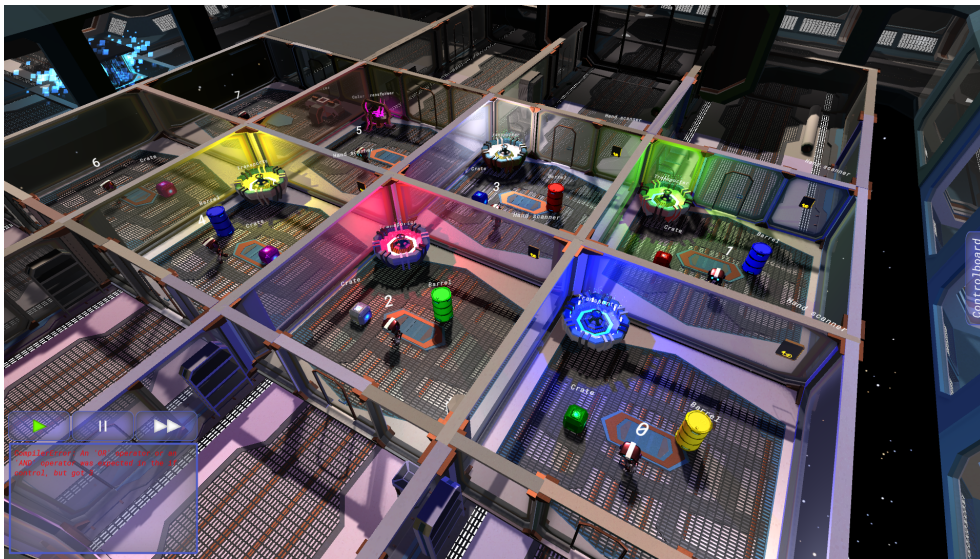


Figure 5.7: Level 12 in our current game



Figure 5.11: Overview of arguments and controls in current project



Figure 5.12: Closer look at the active controls and arguments in current project

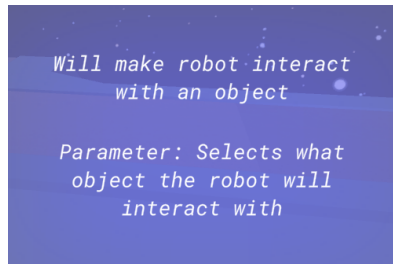


Figure 5.13: Describes what an Action control does

5.3.5.1 Displaying an Exception in a Control

As stated previously in Section 3.2, exceptions can occur in the game. In the first image, we can see that the End control is above the If control, which is illegal, producing a compiler exception. In the second image, from the current project, there is an integer placed as the fourth parameter, where an operator is expected.

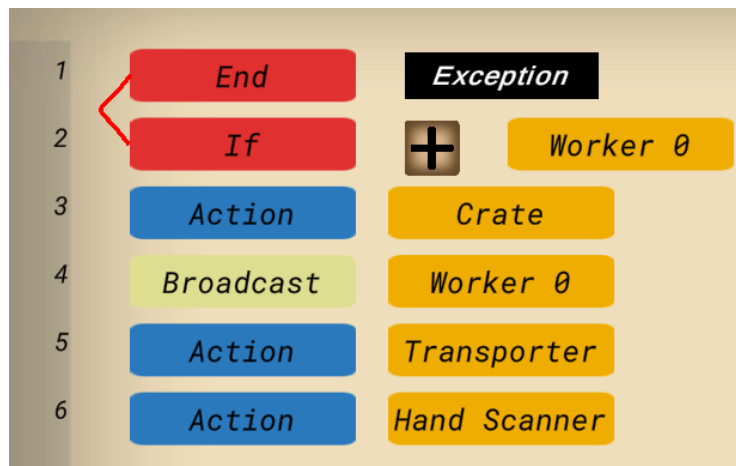


Figure 5.14: An exception display in a control in our fall project

5.3.6 Console

The exception is relevant to the error done previously in Section 5.3.5.1 in the current project. The player will not be able to start the game while having one or more exceptions. Additionally, the console allows the player to start, pause and fast forward the game.

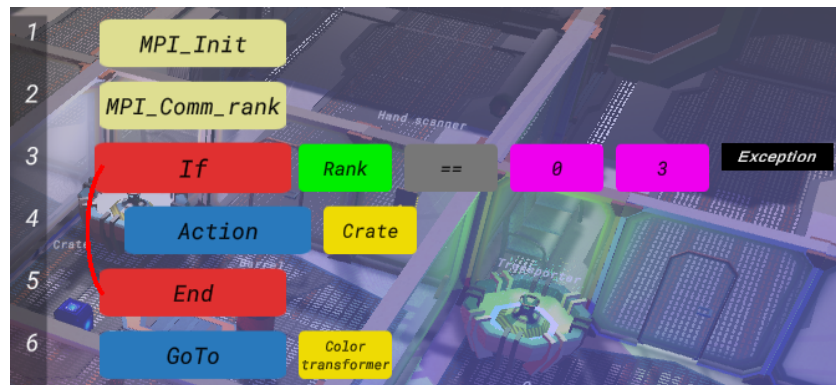


Figure 5.15: An exception displayed in a control in the current project



Figure 5.16: The console displaying an exception in the current project

5.3.7 Level Completed

When a level is completed, the player is offered three options: showing the comparable C code, which is unique to the current project, replaying the level, and proceeding to the next level. In addition, the time it took the program to finish the level, as well as the high score, will be shown. In the current project, the player will also receive stars based on how long it took for the program to complete the level.

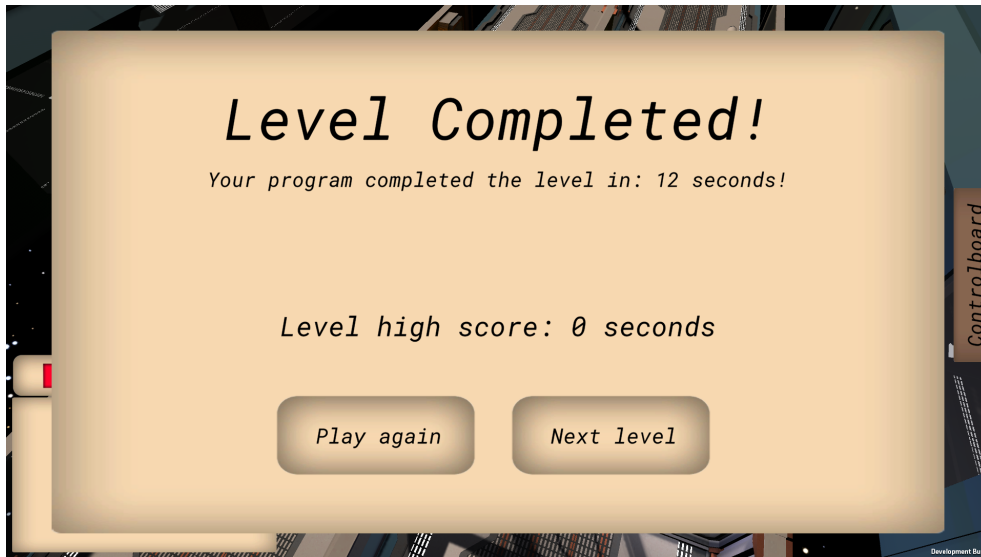


Figure 5.17: Interface when completing a level in our fall project

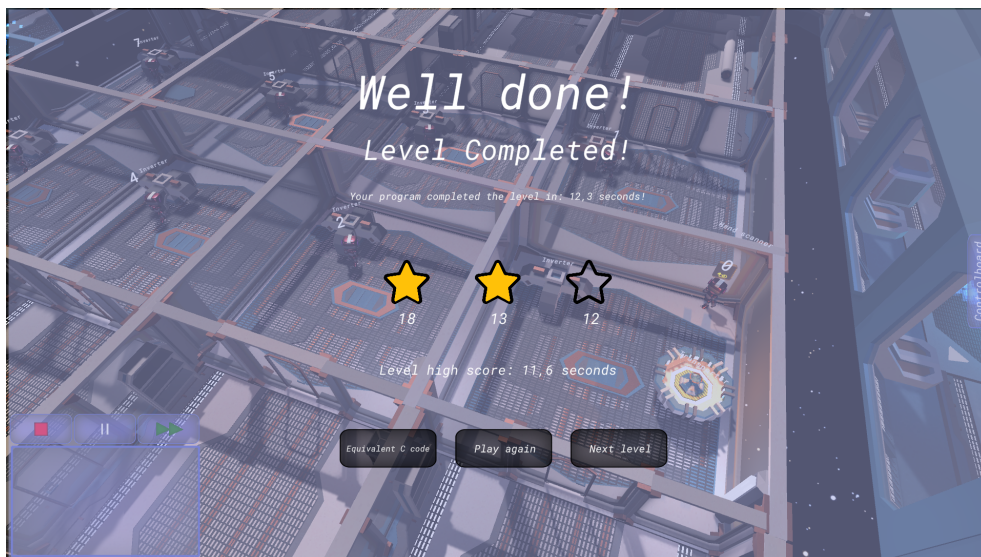
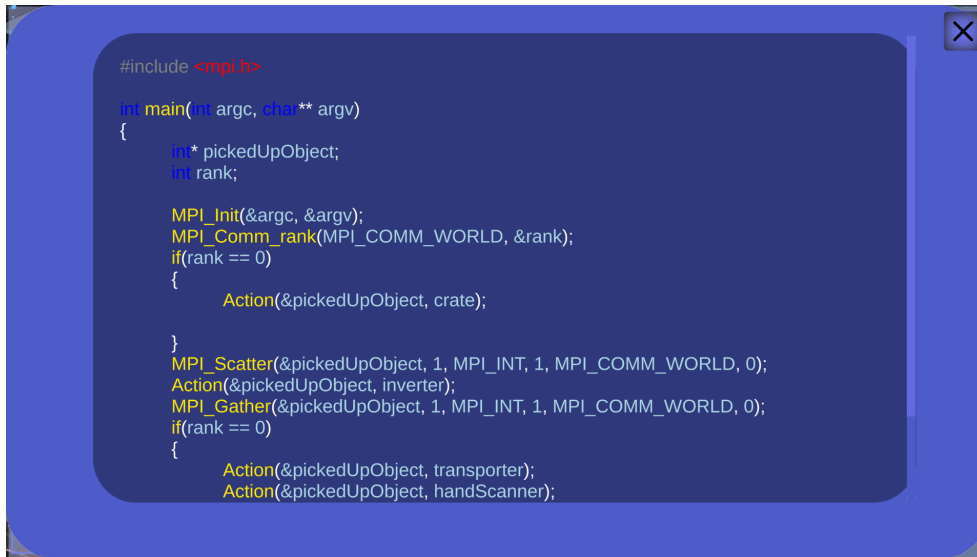


Figure 5.18: Interface when completing a level in the current project

5.3.8 Displaying Equivalent C Code

The code displayed is the equivalent C code for completing the black/white inversion level.



```

#include <mpi.h>

int main(int argc, char** argv)
{
    int* pickedUpObject;
    int rank;

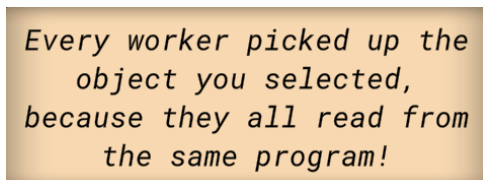
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if(rank == 0)
    {
        Action(&pickedUpObject, crate);
    }
    MPI_Scatter(&pickedUpObject, 1, MPI_INT, 1, MPI_COMM_WORLD, 0);
    Action(&pickedUpObject, inverter);
    MPI_Gather(&pickedUpObject, 1, MPI_INT, 1, MPI_COMM_WORLD, 0);
    if(rank == 0)
    {
        Action(&pickedUpObject, transporter);
        Action(&pickedUpObject, handScanner);
    }
}

```

Figure 5.19: Equivalent C code

5.3.9 Tutorial

In the current project, players can close the tutorial. They are also able to iterate back and forth in the tutorial.



Every worker picked up the object you selected, because they all read from the same program!

Figure 5.20: Tutorial in our fall project

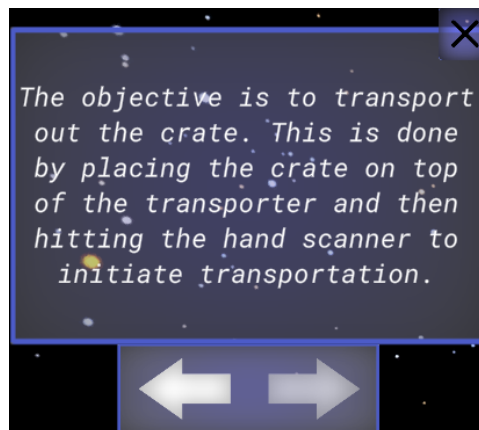


Figure 5.21: Tutorial in our current project

5.3.10 Interactable Objects

In this section the game-elements that are able to be interacted with by the robots are presented. In the current project four new objects have been implemented: Figure 5.24 Cube, Figure 5.26 Transformer, Figure 5.27 Inverter and Figure 5.29 Key.

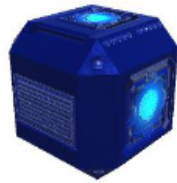


Figure 5.22: Crate



Figure 5.23: Barrel

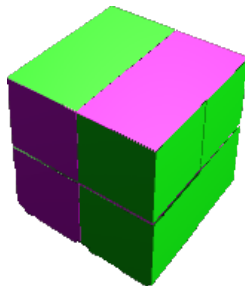


Figure 5.24: Cube



Figure 5.25: Transporter

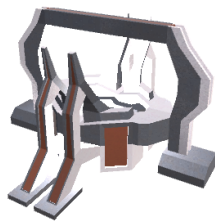


Figure 5.26: Transformer



Figure 5.27: Inverter



Figure 5.28: Hand scanner



Figure 5.29: Key

5.3.11 Border Exchange

First, the root robot picks up the multi-colored cube and scatters it to the other robots. The robots receive a small cube made up of two colors.



Figure 5.30: Scattering a cube to other robots

After this is done, the robots disassemble the cube, keeping the top part of the cube, and dropping the other part to the ground. Then the top part is exchanged with the robot's neighbor.

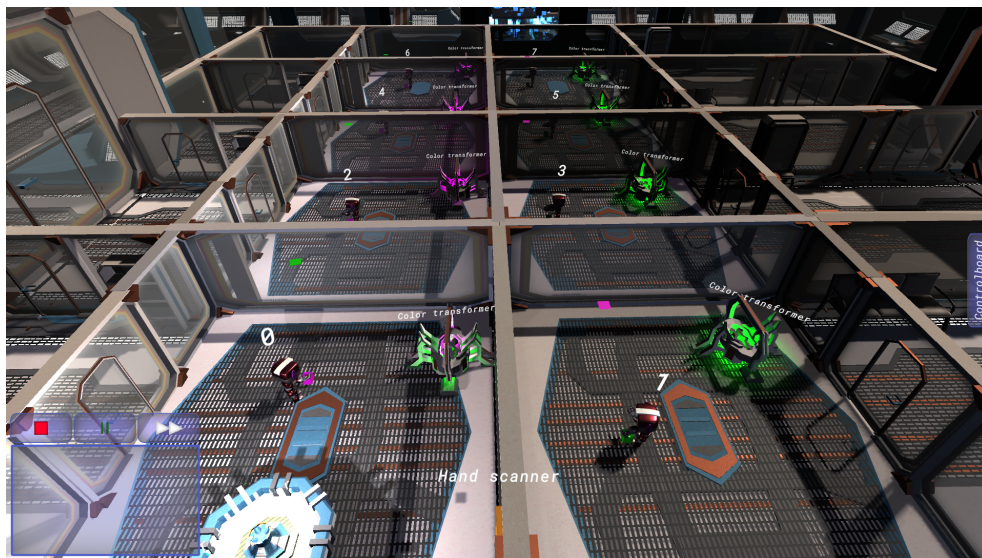


Figure 5.31: Robots disassembles its cube, and exchanging with its neighbour

When the exchange is done, the robots assemble the top part with the received top part - creating a cube with a uniform color.

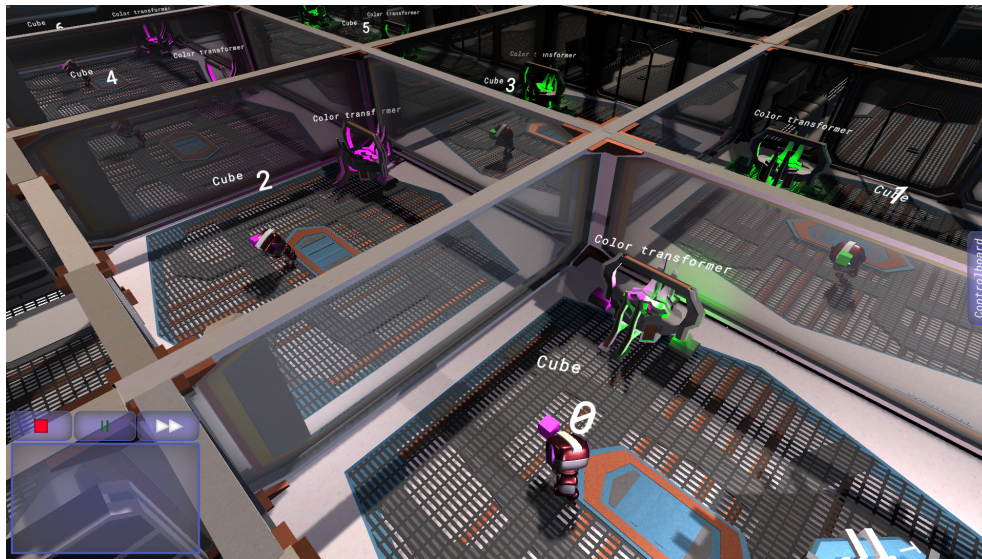


Figure 5.32: Robots assembling the cube back together, creating a cube with only one color

The robots are then ready to begin the color transformation. Every robot puts it into a Transformer object, changing the color to white. When this is done the root robot gathers all of the cubes.

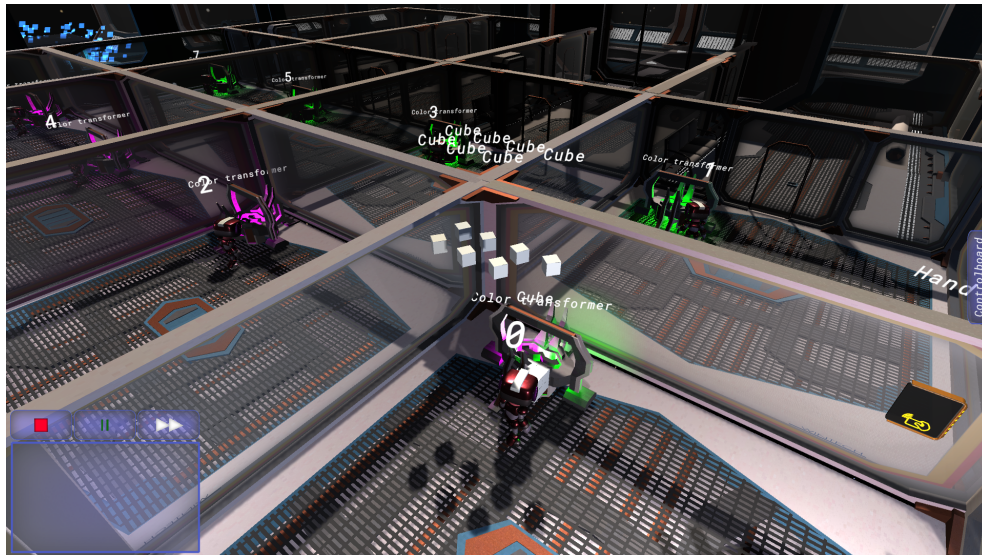


Figure 5.33: The root robot gathers all white cubes from the other robots

5.4 User Study Results

This section will present the results of the user study. The word "Strongly" has been shortened to "Str." for better readability in the tables.

5.4.1 Increase of Interest in Parallel Programming

From the survey, 25% of players agreed that they gained an increased interest in parallel programming. From the interviews, most of the participants communicated that they gained an increase or an initial increase of interest in parallel programming. A few participants did not gain any interest in parallel programming. Their explanations included previous unfavorable encounters with it and a general lack of interest in programming. This can also be illustrated by looking at Table 5.2 from the survey. One participant added that they believe this would be a good starting point for anybody interested in learning about parallel programming/MPI.

Table 5.2: Question 4 and 5 from the survey

Statement	Str. Dis-agree	Disagree	Neutral	Agree	Str. Agree
Before playing the game I was interested in Parallel programming	50%	12.5%	12.5%	25%	0%
After playing the game I was interested in Parallel programming	25%	0%	25%	25%	25%

5.4.2 Game Difficulty for the Players

It was a clear distinction from the observations that programmers advanced through the levels at a higher pace than the non-programmers. The non-programmers had a slower start, but it was clear that it ultimately clicked for them. However, one non-programming participant was perplexed and needed continual assistance from the observers. Yet, after the first few stages, the struggling non-programmer began to feel more at ease with moving the controls and comprehending the notion of communication between processes. Some participants added that they experienced difficulty with various definitions such as *if sentence* and *arguments*. We also noticed that every single one of the participants struggled at one point, and needed assistance. This was especially apparent in the border exchange level.

Everyone seemed to understand the goal of the levels - to transport the crates, barrels, and cubes. One participant made a funny comment; wishing they did not have an annoying observer watching them. Regarding that, a few participants expressed that they felt stressed being observed. A participant found the `If` controls to be confusing, coming from a non-programmer. Additionally, another participant added that they found the `MPI_Comm_rank` control to be "very magical".

Most participants also stated that they had to ask the observers questions to progress through the stages, but that if they had spent some more time, they would have made it on their own.

Table 5.3: Question 1 and 6 from the survey

Statement	Str. Dis-agree	Disagree	Neutral	Agree	Str. Agree
I felt I needed help to complete the game	12.5%	37.5%	37.5%	12.5%	0%
I felt the tutorial taught me everything needed	12.5%	25%	12.5%	25%	12.5%

5.4.3 Learning Effect of the Game

When asking question 11 from Appendix C.2, it was clear that most participants understood what the intentions of the levels were. Although not everyone could provide an example, knowing the concept is evidence of learning. Some added that they understood that dividing up the problem set into several processes could make programs more efficient. One participant eagerly remarked: "*The order of operations is important, almost like a symphony! Additionally, no process must be idle.*". Some participants found it hard to translate what they have learned from the game into real MPI programming - wishing the tutorial would explain what problem the level solved. From Table 5.4 one can observe that the participants understood the MPI functions, some more than others. Concerning the quiz, the participants' average points increased from 5,875 to 7,375, after playing the game, where the maximum score possible was 12. Some participants mentioned that they wish the game would give information about what tasks have been solved, for instance in the border exchange level. Another participant thought that the game provided little information about its purpose.

Table 5.4: Question 2 from the survey

Statement	Str. Dis-agree	Disagree	Neutral	Agree	Str. Agree
I felt I understood the MPI functions	0%	0%	0%	62.5%	37.5%

5.4.4 Enjoyment Effect of the Game

Table 5.5 describes the result of the enjoyment effect on the players. 62.5% of players strongly agreed that the game was fun. A significant percentage of comments from interview participants indicated that they had a wonderful time. One participant added that they felt the game gave satisfying interactions between the robots and a sense of accomplishment when beating the levels. Participants added that receiving stars for how fast they completed the levels became a motivating factor. However, one participant found receiving stars to be stressful. Several participants during observations were motivated to try to beat their own highest score in the levels. A few mentioned that they want to beat the world record for this level. A participant felt that functions were missing from the visual programming: "A lot of duplicated codes can be avoided with functions". Furthermore, the participant noted that templates would be a useful addition because they found themselves duplicating a lot of the same code.

The majority of participants found it annoying that after they exited the tutorial, it did not reappear at a later time to provide more information. They closed the tutorial because it was cluttering up the user interface. We also discovered that players did not read every tutorial panel based on our findings. One participant felt the game would have a smoother experience if it could be sped up even further. Another suggestion was to implement copy & paste, for faster visual programming. For a better overview of the code, one participant suggested zooming. Additionally, one mentioned that being able to choose favorites, would speed up the visual programming.

Table 5.5: Question 3 from the survey

Statement	Str. Dis-agree	Disagree	Neutral	Agree	Str. Agree
I found the game fun	0%	0%	12.5%	25%	62.5%
I felt the user interface was easy to use	0%	0%	0%	62.5%	37.5%

Chapter 6

Discussions

The findings from the development game will be discussed first, followed by the findings of the user research.

6.1 Development Results

In this section, we will discuss the result of our game development. This was motivated by the following research questions:

RQ 1: *"What is the best game engine for implementing an MPI edutainment game?"*

RQ 2: *"What are some of the technical advantages and challenges with the approach chosen?"*

RQ 3: *"What are the most useful MPI functions, and how does one illustrate them well in a gaming setting?"*

RQ 4: *"Will there be any timing issues related to such a game?"*

The rest of this section discusses the results in Chapter 5, main menu, settings, level selector, visual programming, tutorial, and border exchange. Furthermore, simulation of functions and simulation of processes from 3 will also be discussed.

6.1.1 Main Menu

In regards to the main menu of the game, we have not found a reason to change much from the fall project. We gave the game a somewhat fitting title "Parallels" because we wanted to emphasize that the game is centered around parallelism. The only substantial changes made since the fall project were the implementation of a settings button and also an option to remove the game save data. The option to remove game save data is a standard feature in games, but it was also convenient for us regarding testing the game for the user study. In future versions of the game, we would want to put more effort into the main menu. We feel that captivating the audience from the get-go is a powerful tool to create a good first impression which hopefully will result in more engagement from the player. However, Sweetser and Wyeth [5] highlights the value of having a pragmatic main menu, where you do not sacrifice usability for aesthetics. One way to balance this

line could be by having the integrity of the main menu intact while showcasing some of the endgame content in the background of the main menu art.

6.1.2 Settings

The settings panel is a wholly new feature in the current project. It contains settings regarding the game's graphics and audio. These are the standard quality of life game options that the player has at his disposal. We did not want to over-complicate this part of the game, but we were keen to make the game's fps as optimal as possible no matter what computer setup a player has at their disposal. With the game still being in its early stages, we did not find any reason to implement any more options for the player. In Section 2.1.1, Sweetser and Wyeth highlight how players should be able to customize the controls and the gameplay to fit their learning and playing styles, if not, the game should be designed to allow different styles of learning and playing.

Regarding the ability to customize controls, this is not relevant to our game since it is in such an early stage. One could customize the camera sensitivity, but we did not feel like the reward matched the resources required for such a trivial feature. We do, however, recognize the need for such customization of controls in potential future iterations of the game, where we can imagine more complex controls. More interestingly, we want to address the claim that the gameplay, or game, should be designed to allow different styles of play and learning. On the surface this seems like a non-controversial claim, but when you factor in that it is an edutainment game being developed things get a bit murky. There is a friction present between the player's ability to complete the game and our ability as developers to teach the designated curriculum to the player. This is one of the reasons why we have not implemented different difficulty settings. We were not able to relieve this tension between teaching the curriculum and engaging all players, so we prioritized the integrity of the information that we communicated.

6.1.3 Level Selector

The level selector interface has seen some minor changes since the fall project. We revamped the user interface to make it more aesthetically pleasing to the player, and we also added stars to represent the scores the player achieved on each level. The latter was done to provide appropriate feedback to the player. Furthermore, a preview of the levels is now displayed on each level, even the locked ones. This was done to give the players a sneak peek at what was awaiting them in the next level, as well as to make the interface more visually pleasing. In future iterations of the game, we would want to use this interface to convey more information to the players. One idea is to include a star counter that displays how many of the potential stars have been obtained. There are also possibilities regarding the transition from the level selector interface to the game, or other graphical considerations that was limited by the time constraint.

6.1.4 Visual Programming

As stated, in our fall project, when controls were chosen, an options screen would pop up and the player was able to select the argument. Participants found it cumbersome because if they chose the wrong argument, the whole control would have to be scratched, as there was no way to change the argument. After looking for inspiration from Section 2.5.2, we found that they have an elegant solution, where arguments are already visible and can be dragged to functions. Now, both arguments and controls already reside on the screen, thus, arguments can easily be dragged and dropped to the controls. Players are now able to both remove and move arguments between controls with ease. Furthermore, having the user interface intuitive is a central point in [5]. In the fall project, players were only given instructions on what a control did by the tutorial, and had to restart the tutorial to be able to check what the control did. User feedback from the fall project made it clear that this was not an intuitive behavior. Making it more intuitive was done by implementing a description of each control and argument.

Moreover, the `If` control was very limited, lowering the complexity ceiling of the visual programming. It could only have a maximum of three arguments, where only "Robot 0", "Robot 1", "Robot 2", and "Robot 3" were the arguments capable of selection, the control implicitly added an `AND` operator between them. It did not behave similarly to a real if sentence. It was improved by introducing `MPI_Comm_rank`, a `rank` variable, integers and operators. Players now have to run `MPI_Comm_rank`, to retrieve the ranks of the robots and then use the `If` control similar to a real if sentence. Additionally, the `If` control no longer has any limitations to how many arguments it can have. Because of user input from the fall project, the user interface was also altered to translucent blue; participants complained that the controlboard obscured their view of the robots resulting in constant closing and opening of the controlboard. There were no complaints from participants concerning this issue after the user interface was modified.

We have noticed that after the first few levels, a large number `If` controls must be placed to complete the levels, which may be laborious to place. In a real MPI program although if sentences are used, they are not as often used compared to our game. In a genuine MPI application, only the root process performs anything that the other processes do not. However, in this game, the processes (robots) do a lot of different things. This can skew the expectations of the participants about real MPI programming. When simulating the border exchange level, we discovered that fewer `If` controls were utilized to accomplish the level, and this is how it should be portrayed to players. The game should have featured more real-life examples of MPI programming to create a more exciting environment for the players and their expectations.

As stated earlier in Section 3.2, every new level unlocks new controls, arguments and game-elements. We did this because it maintains the game's difficulty, and keeps the game interesting. It is far simpler to deal with fewer things initially than to be assaulted with controls, arguments, and game elements. This is

underpinned by both [5, 6].

Given the ground structure of the visual programming, it is straightforward to implement new controls/arguments to, for instance, add a new MPI function. By adding a new object in the `controls.json` or `arguments.json` file, a new control/argument will appear on the controlboard. It automatically adds the `Draggable.cs` class on it when the game starts, making it capable to be dragged. Implementation of new game elements was also structurally programmed to be straightforward. With the `Interactable.cs` class, any game element can easily be manipulated to be either a game element to abstract out code or an element to be sent between robots.

6.1.5 Simulation of Processes

When we started this project we were aware that we would need to implement a lot of levels. Since this is time-consuming work, we were keen to figure out a solution that would enable us to easily create new levels from scratch. The parent-child relationship between game elements was found to be an excellent solution to this problem. This made it possible for us to place robots out in a scene, then insert game elements under them as a child. When the game has then started the robot automatically gains a local connection to the game element, making this an efficient solution and, in the process, a valuable time-saver regarding the creation of new levels. Furthermore, in real MPI programs, processes have local memory. By dividing the robots into cubicles, players can explicitly observe what game element has a local connection to the robots - simulating local memory for processes.

6.1.6 Simulation of Functions

Our supervisor made remarks about not having the ability to use `MPI_Init` and `MPI_Finalize` in the fall project. We initially thought about adding these functions in the fall project, but decided that it would be tedious to use those controls for every level. However, as it is used in every MPI program it is highly relevant in a game where MPI is being educated. Therefore, these functions were implemented. Another function that is highly relevant for MPI is `MPI_Comm_size`. It is used in every MPI program. However, it was scratched, because we could not find any usage for it. The function is used to retrieve how many processes are active in an MPI program. In the game, the player knows at all times how many processes (robots) are present, thus, it was difficult to find a purpose for it.

In a real MPI program the MPI functions have a lot more arguments than the controls, we felt like it would be very tedious to have to drag all of these arguments into the controls. Additionally, we felt that players would have better odds of understanding the controls with fewer arguments. However, this approach comes with its drawback; the players' expectations of these functions will be distorted in relation to real MPI programming. Despite that, when players complete a level,

they can choose to display the equivalent C code. Here, their expectations can be restored.

MPI_Sendrecv behaves the same as in a real MPI program. Though, we took notice of something regarding the simulated functions *MPI_Send* and *MPI_Recv*; *MPI_Sendrecv* should be able to be replicated by only using *MPI_Send* and *MPI_Recv*. This is not possible in the game, because both robots pick up an object and the receiver must have its hands free. This could, in future work, be changed to be able to send an object without holding it.

6.1.7 Tutorial

Based on user feedback in the fall project, the tutorial was changed. Players felt frustrated that they had to have the tutorial panels open at all times. Furthermore, they had to constantly restart the level to be able to check what information the tutorial had in its previous iteration. Its functionality was also changed. In the fall project, no matter what action the player did, the tutorial would increase its iteration. Players could do the complete opposite of what the tutorial said the player should do, and the tutorial would iterate anyway. This was changed in the current project; players will have to do the correct action to be able to continue the tutorial iteration. These are all the changes that have been made to the project from the fall version to the current version.

In several of the tutorial panels, the present version of the project suffers from long explanations. We have reason to believe that this will have a detrimental impact on player engagement since they will become bored with reading large blocks of text [5]. Making the game tutorial briefer would improve future editions of the game. This might be accomplished by breaking up the tutorial panels and explaining things more progressively. Another method to cut the tutorial down is to leave out some of the definitions. This approach raises the issue, "For whom is the game designed?" In the user study part that follows, we will go deeper into this subject.

6.1.8 Border Exchange

In level 9, we wanted to explain the process of border exchange. Specifically to highlight one key aspect; that something has to be exchanged between processes to solve a problem. To make this feasible within the limitations of our game, we made a simplified simulation of the border exchange process. Actual border exchange in an MPI program is a lot more complex than our representation of the process. However, we wanted to convey the product of border exchange, to give the player an understanding of its utility. We wanted to show that by using border exchange one could solve the level more efficiently and speedily. This was done by making it apparent to the player that a chunk of the processes' local game element had to be exchanged between their neighbor to maximize the efficiency of the processes. We made this point more visceral by allowing the players the slower

alternative where border exchange was not utilized. The players could then experience the decreased speed that followed not using border exchange to solve the level.

6.2 User Study

In this section, we will discuss the results of our user study. The following subsections are motivated by the following user study questions:

USQ 1: *"Will the game affect increasing interest in parallel programming?"*

USQ 2: *"Is the game too difficult?"*

USQ 3: *"What is the learning effect of playing the game?"*

USQ 4: *"What is the enjoyment effect of playing the game?"*

6.2.1 Increase of Interest in Parallel Programming

Results from the interview and quiz showed that most participants gained interest in parallel programming. Evoking the curiosity of players' motivates them to learn and be interested according to [6]. Malone stated that environments should be neither too complicated nor too simple - in relation to the player's existing knowledge. According to our observations of the players, we did not feel like most players got overwhelmed with the complexity, nor found it too easy. This may have contributed to an increase in interest in the subject. However, there were a few participants who denied an increase in interest in parallel programming, but their reasons were not about the game. With the benefit of hindsight, we can see that it is hard to draw conclusions about players' interest in parallel programming when their only experience of it comes through an oversimplified game version of the concept. We will explore this issue deeper below in Section 6.2.3.

6.2.2 Game Difficulty for the Players

As non-programmers do not have the same background as programmers, they do not know from their background what arguments or what an if sentence means. From the observation and the survey, we can see that non-programmers had a slow start compared to the programmers. This means that the tutorial needs to provide more in-depth explanations of these concepts. However, this method comes with its trade-off: even more lengthy explanations. Thus, in future work, it should be defined what the primary user group should be.

We may have implicitly set the primary user group as programmers, as we have not provided explanations around these concepts. Consequently, the non-primary user group could feel overwhelmed and ultimately lose self-confidence in themselves playing the game. "Failure in a challenging activity, like a computer game, can lower a person's self-esteem and—if it is severe enough—decrease the person's desire to play the game again" [6]. However, we have observed that even

programmers struggled with the border exchange level - requiring more explanations around this level. This is supported by the fact that most participants did not strongly agree to the tutorial being sufficient enough.

As the game is a type of puzzle game, the player must think hard about the task at hand. Since the author was observing the participants playing the game, this might be a cause for increased stress levels for the players, which in turn might impact their performance.

We believe by hiding controls, arguments, and game elements from the start, and then selectively revealing them, contributed to keeping the game difficulty lower, as expressed by [5]. The tutorial also contributed to this, though it should be improved, as mentioned earlier.

6.2.3 Learning Effect of the Game

It is clear when looking at the results of the quiz and the interview that the participants felt like they learned something. The quantitative data from the quiz suggests that they received a broader knowledge field from the game. However, it is not indicative that this will translate to real MPI programming. And not sufficient evidence to conclude that it will make it easier for the participants to create a real MPI program. The users did not try real programming but simply played a game. Nevertheless, most participants understood the concept of level 8 (Black/white inversion) in the game, indicating some learning.

A more descriptive tutorial could be an advantage for the learning effect. Explaining concepts such as border exchange, and come with an example of this with image manipulation. But, we think by playing around with the simulated MPI functions that the participants have done, will make it much easier to begin with MPI in the first place. They can see what the functions do, for instance; *MPI_Scatter* splitting up chunks and sending them to the robots. Whereas in real MPI programming, this can not be seen explicitly. In our personal experience, some of the challenges when beginning with MPI is trying to understand what the functions do.

In a real MPI-program, the *MPI_Comm_rank* behaves slightly different from our simulated function: It takes in an argument, which can then be used to retrieve a process' rank. Our implementation of this function has no parameter, thus, can be experienced as "magical" as one participant mentioned. We suggest that this can be changed to take in the argument Rank. This may lead to players understanding that the function is directly connected with the Rank argument.

6.2.4 Enjoyment Effect of the Game

We believe that an important element that decides the game's enjoyability, is the participants' degree of autonomy regarding how to complete the level, while at the same time striving towards a preconceived goal. As [7] believed, a game should be balanced between emergent -and scripting gameplay.

This does not mean that the players are limited only to the narrow goal of achieving the level. We want, even encourage, the players to have additional meta-goals. By implementing stars that relate to the player's efficiency in completing the level, we allow the players to beat their high score or even the world record. This is in line with [5, 6], where they have found features with these attributes might affect the enjoyment of games.

Additionally, we were keen on keeping the pace of the game optimal; starting with few controls, arguments, and game elements, then selectively revealing them to the players. We believe this might have contributed to keeping their interest in the game. This is reinstated by [5, 6].

However, we believe the game could be even more enjoyable by reevaluating the tutorial. It gives lengthy explanations which according to [5] can result in players becoming bored. Also, we found evidence that players did not read all tutorial panels - making an even stronger argument to change the tutorial. The user interface should be intuitive and easy to use so that players feel a sense of control of their actions according to [5]. Following this logic, one quick way to make the game more enjoyable is to make the tutorial more intuitive. At this moment in time, the tutorial panels pop up in inappropriate places, and the tutorial gets closed for the rest of the level if the player decides to close the tutorial panel.

Chapter 7

Conclusion and Future Work

This thesis focused on creating an MPI edutainment game to provide an environment for players to learn parallel programming with MPI. The thesis proved that simulating such an environment can be challenging. Challenges arose such as tedious visual programming with an overload of if sentences and illustrations of real MPI program tasks. However, we showed that it is possible to develop an edutainment game that is enjoyable for both programmers and non-programmers. We found it hard to determine whether players gained interest and learned the different concepts in the field of parallel programming as a direct result of playing the game. In the following sections, we will address both the research -and the user study questions. Additionally, we will discuss our suggestions for future work regarding the development of MPI edutainment games.

7.1 Research Questions

Following is a summary of how we addressed each research question.

What is the best Game Engine for Implementing an MPI Edutainment game? (RQ1)

We have found that game engines as of 2022 are very similar to each other, and it mostly comes down to personal preference. However, there are some differences between them. Such as how graphically advantageous they are, their AI capabilities, and networking to name a few. For instance, for games that draw in players on being very visually appealing, developers should probably favor the Unreal Engine. Since this game did not need to have AI, networking, or superior graphics, it came down to personal preference and we settled on the Unity game engine. But most substantially, this is the engine the author has had the most experience using, and we felt that this was the natural choice for our project.

What are some of the Technical Advantages and Challenges with the Approach Chosen? (RQ2)

We recognized several technical advantages that came with the approach chosen for this project. By creating super-classes and by embracing emergence as a concept, we, and future developers of the game, were able to effortlessly create and simulate MPI -and general programming functions including new game elements for the robot (process) to both interact with and to abstract out code, such as the inversion of pixel color. Furthermore, by creating a parent-child relationship between the robots and game elements, we reduced the time needed to create new levels.

However, there were some technical challenges with the approach chosen. These include the simulation of real MPI programs and implementation of `MPI_Comm_size`. Moreover, in a real MPI-program, `MPI_Send` and `MPI_Recv` are able to replicate `MPI_Sendrecv`. Given how the sending and receiving of game elements between robots were made in the game, the simulated functions: `MPI_Send` and `MPI_Recv`, were not able to replicate `MPI_Sendrecv`. This shows some of the difficulties of trying to simulate an MPI environment. Balancing the game's intuitiveness with the structural rules that guide the MPI framework, leads to one having to compromise either the former or the latter.

Another challenge that arose, was connected to the way the levels were set up to be completed. Most levels required a huge amount of if sentences to be able to complete them, whereas in actual MPI programs they are not used nearly with the same frequency. This results in tedious visual programming for the player.

What are the most Useful MPI-Functions, and how does one Illustrate them Well in a Gaming Setting? (RQ3)

We recognized the most useful MPI-functions to simulate in the game from the author's experience and our advisor's use of these functions in her parallel programming subject. An additional reason for choosing these particular functions were our familiarity with them, from the book [1]. The functions we deemed as most useful are as following: `MPI_Init`, `MPI_Finalize`, `MPI_Comm_rank`, `MPI_Send`, `MPI_Recv`, `MPI_Bcast`, `MPI_Scatter`, `MPI_Gather` and `MPI_Sendrecv`.

To illustrate them we reduced the total amount of arguments in the functions. This resulted in them being easier to grasp and increased their usability in the game. Furthermore, our approach led MPI functions to be explicitly seen by the players. This was done through a visual representation of the game element being transferred to other robots. This animation represents the idea of communication overhead, hopefully making it easier for the player to understand this concept.

7.2 User Study Questions

Following is a summary of how we addressed each user study question.

Will the Game affect Increasing Interest in Parallel Programming? (USQ1)

We found it difficult to determine players' interest in parallel programming as a field. We suspect that players might conflate their enjoyment of the game with their interest in parallel programming as a field. In hindsight, this user study question might have been too vague, because you can not confidently claim that any qualitative judgments the players make about the parallel programming field, based on their experience playing a game that simulates parallel programming, is representative of actual interest in said field.

Nevertheless, we tried to determine their interest in the field of parallel programming by using the edutainment game as a proxy for parallel programming. By this standard, we found that participants' interest in the field of parallel programming increased. We estimate that this came from the fact that the environment was neither too simple nor complicated in relation to the player's existing knowledge. However, a few participants felt that the game did not increase their interest in the field, but the cause for this was not about the gameplay.

Is the Game too Difficult? (USQ2)

We think that the primary user group of the game should be explicitly defined in any future development. If non-programmers are the primary user group, the tutorial should explain concepts around general programming. However, this comes with its trade-offs where the tutorial includes even more lengthy explanations.

As expected, non-programmers found the game harder compared to programmers. The greatest cause of difficulty regarding the game came in the border exchange level. The tutorial should be improved to explain concepts around this level. This is to make sure the players do not lose self-confidence in themselves and also make sure they can beat the level on their own. Nevertheless, we found that the game is not too difficult to play.

What is the Learning effect of Playing the Game? (USQ3)

The players gave feedback that they learned something from the game. But it is not indicative that this is in relation to real MPI programming. A more detailed and descriptive tutorial that explains MPI concepts should have a positive effect on players' learning. Additionally, a more in-depth study is required to be able to answer this user study question. This can for instance involve having the participants do real MPI programming, after finishing the game.

What is the Enjoyment effect of Playing the Game? (USQ4)

There was strong evidence of the players enjoying themselves playing the game. We believe that this came down to players having full control regarding how they wanted to complete the levels while having a well-defined goal, while at the same

time having the option to form side-goals, such as beating their high score. Additionally, selectively revealing game elements to players may have contributed to an optimal game pace, enhancing and keeping their interest. Moreover, we have identified the tutorial panel as an area with potential for improvement. We believe making it more intuitive and more concise will increase the enjoyment effect of the players.

7.3 Future Work

Based on user feedback, we have a strong belief that featuring more levels closely connected to real MPI programs will enhance players' experience of the game positively. Further work should also focus on shortening the amount of `If` controls needed to complete a level. Also, the primary user group of the game should be defined before any further development is made. Asserting if the game is intended for only programmers or for everybody, changes the entire manuscript of the tutorial, particularly regarding definitions of terms. Nevertheless, the tutorial should be more concise indifferent to what user group it is intended for, and the border exchange level should be explained better. Additionally, even though players close a tutorial panel, the next tutorial panel should pop back up after another iteration.

Relating to the technical aspect of things, future work should figure out a way to implement `MPI_Comm_size`, as it is a vital function for real MPI programs. Additionally, the behaviour around simulating communication between robots (processes) should maybe be changed, as the simulated functions `MPI_Send` and `MPI_Recv` are not able to replicate `MPI_Sendrecv`, as they should be able to. Simulation of `MPI_Comm_rank` should be improved and perhaps more similar to the real function, so it is not experienced as mysterious to players.

We believe future work on this topic would benefit from implementing more MPI-functions to be simulated, such as `MPI_Reduce`, `MPI_Barrier`, `MPI_IRecv` and `MPI_IRecv`. Lastly, we would be interested to see an extension of our project through the use of other parallel programming extensions, such as openMP or Pthreads. We would be excited to see if any of the myriads of challenges we faced undergoing this project could be solved more efficiently with a rivaling parallel programming extension.

Bibliography

- [1] P. S. Pacheco and M. Malensek, Eds., *An Introduction to Parallel Programming (Second Edition)*, Second Edition. Philadelphia: Morgan Kaufmann, 2022, p. 1, ISBN: 978-0-12-804605-0. DOI: <https://doi.org/10.1016/B978-0-12-804605-0.00003-8>.
- [2] G. Howard and K. Berens, *The Rough Guide to Videogames*. Rough Guides; 1st edition, 2008, ch. 2.
- [3] Financesonline. “Number of gamers worldwide 2022/2023: Demographics, statistics, and predictions.” (), [Online]. Available: <https://financesonline.com/number-of-gamers-worldwide/>.
- [4] D. Lucaride, “The impact of fun and enjoyment on adult’s learning,” *Procedia - Social and Behavioral Sciences*, vol. 142, pp. 439–446, 2014, The Fourth International Conference on Adult Education, Romania 2014, ISSN: 1877-0428. DOI: <https://doi.org/10.1016/j.sbspro.2014.07.696>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1877042814046242>.
- [5] P. Sweetser and P. Wyeth, “Gameflow: A model for evaluating player enjoyment in games,” *Computers in Entertainment*, vol. 3, p. 3, Jul. 2005. DOI: 10.1145/1077246.1077253.
- [6] T. W. Malone, *What makes things fun to learn? Heuristics for designing instructional computer games*. SIGSMALL ’80: Proceedings of the 3rd ACM SIGSMALL symposium and the first SIGPC symposium on Small systems, 1980, pp. 163–165.
- [7] P. Sweetser and J. Wiles, “Scripting versus emergence: Issues for game developers and players in game environment design,” 2005.
- [8] H. Lowood, “Game engines and game history,” *Kinephanos*, 2014, History of Games International Conference Proceedings, January 2014, ISSN: 1916-985X.
- [9] F. Sanglard, *The Rough Guide to Videogames*. Independently published, 2019, pp. 391, 393, ISBN: 1099819776.
- [10] J. K. Haas, “A history of the unity game engine,” 2014.
- [11] Unity, “2021 gaming report,” 2021. [Online]. Available: <https://create.unity.com/2021-game-report>.

- [12] F. Jerga, "What is the unity game engine- all you need to know," 2021. [Online]. Available: <https://medium.com/eincode/what-is-the-unity-game-engine-all-you-need-to-know-d4ce77a1b7d2>.
- [13] M. Thomsen, "History of the unreal engine," 2012. [Online]. Available: <https://www.ign.com/articles/2010/02/23/history-of-the-unreal-engine>.
- [14] A. Eldad, "Unity vs unreal, what kind of game dev are you?," 2021. [Online]. Available: <https://www.incredibuild.com/blog/unity-vs-unreal-what-kind-of-game-dev-are-you>.
- [15] O. A. McBryan, "An overview of message passing environments," *Parallel Computing*, vol. 20, no. 4, pp. 1–2, 1994, Message Passing Interfaces, ISSN: 0167-8191. DOI: [https://doi.org/10.1016/0167-8191\(94\)90021-3](https://doi.org/10.1016/0167-8191(94)90021-3). [Online]. Available: <https://www.sciencedirect.com/science/article/pii/0167819194900213>.
- [16] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker, "An introduction to the mpi standard," USA, Tech. Rep., 1995.
- [17] M. P. I. Forum, "Mpi: A message-passing interface standard version 3.1," Tech. Rep., p. 1. [Online]. Available: <https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf>.
- [18] F. B. Kjolstad and M. Snir, "Ghost cell pattern," in *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, ser. ParaPLOP '10, Carefree, Arizona, USA: Association for Computing Machinery, 2010, ISBN: 9781450301275. DOI: [10.1145/1953611.1953615](https://doi.org/10.1145/1953611.1953615). [Online]. Available: <https://doi.org/10.1145/1953611.1953615>.
- [19] S. Baydas and B. Karakas, "Defining a curve as a bezier curve," *Journal of Taibah University for Science*, vol. 13, no. 1, pp. 522–528, 2019. DOI: [10.1080/16583655.2019.1601913](https://doi.org/10.1080/16583655.2019.1601913). eprint: <https://doi.org/10.1080/16583655.2019.1601913>. [Online]. Available: <https://doi.org/10.1080/16583655.2019.1601913>.
- [20] TomorrowCorporation. "Human resource machine." (2015), [Online]. Available: <https://tomorrowcorporation.com/about> (visited on 06/08/2022).
- [21] Scratch. "History." (2003), [Online]. Available: <https://en.scratch-wiki.info/wiki/Scratch#History> (visited on 06/08/2022).
- [22] Scratch. "About." (2003), [Online]. Available: <https://scratch.mit.edu/> (visited on 06/08/2022).
- [23] S. Rahi, "Research design and methods: A systematic review of research paradigms, sampling issues and instruments development," *International Journal of Economics Management Sciences*, vol. 6, Jan. 2017. DOI: [10.4172/2162-6359.1000403](https://doi.org/10.4172/2162-6359.1000403).

Appendix A

User Guide

Here, the user guide will be presented. It will explain how to start, customize and play the game. All the various objects in the game will be explained as well as how to utilize the visual programming in the game to complete the levels. As there is a tutorial imminent in the game, every aspect of the game will not be covered such as how every control (programming-function) and arguments works.

A.1 Main menu

When the game is loaded up, the main menu appears. Here, the game save data can be deleted, able to customize settings and exit/start the game.



Figure A.1: Caption

In the settings, a various of things can be customized; enable or disable fullscreen, change the quality of the game and set sound volume.

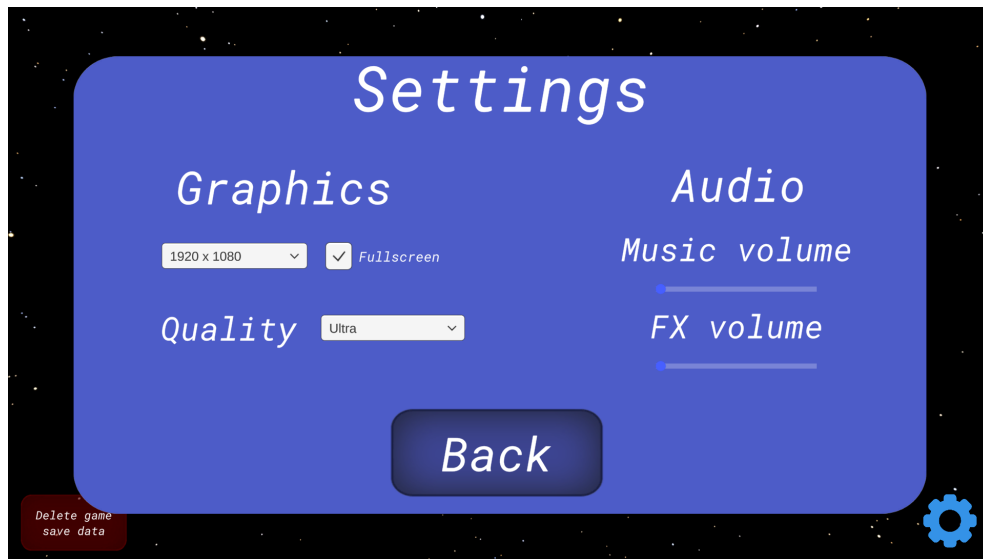


Figure A.2: Main menu

To start the game, the start button can be clicked. Here, all the levels will be displayed. If a level is greyed out, it means the level has not been unlocked yet. Additionally, how many stars have been received for each level is displayed.



Figure A.3: Selection of a level

A.1.1 Gameplay

Players can move the camera with WASD, right mouse button and right click. In the levels, there are robots. The objective is to program the robots to transport out

all crates, barrels and cubes. This is done by robots picking them up and placing them onto a Transporter, and then initiating transportation by robots interacting with a Hand Scanner. A Transporter can be disabled in the game. When disabled, game-elements can not be put onto it. However, it can be activated by using a Key on it. Furthermore, a *transporter* emits light, this light represents which color of an object it can transport. A green *crate* can not be put onto a *transporter* emitting magenta-light.

To start programming them, the Controlboard has to be opened. Here, the visual programming can start. The left-side panel holds all of the controls and arguments a player can use to program the robots. Controls are programming-functions which can be dragged and dropped into the right side panel of the Controlboard to make them active. Most controls must have arguments. An Action control interacts with an object specified by the argument. For instance, if an Action control has the argument *Crate*, the robot picks up the crate. Furthermore, a description of both controls and arguments can be read by hovering over them.



Figure A.4: Example of picking up a crate, putting it onto the transporter and initiating transportation

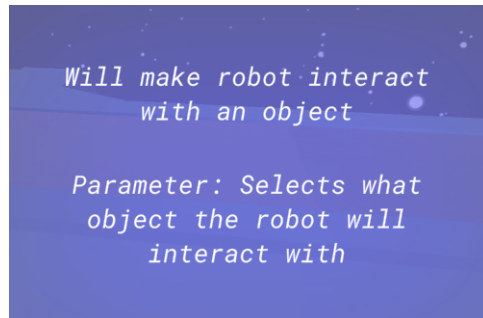


Figure A.5: Description of what the Action control do

To run the program, the start button must be pressed. When pressed, it transforms to a stop button, and can be pressed to stop the game. Additionally, the game can be paused and also fast forwarded.



Figure A.6: Showcasing start, pause and fast forward buttons

Exceptions can occur, for instance, a robot trying to interact with a Transformer object while having no object in its hands. The robot is trying to put an object onto the Transformer object, creating a run-time exception. The integer inside the parentheses represents which robot rank it occurred to.



Figure A.7: Run-time exception occurred to robot 0

When a level is completed, stars is received based on the performance (How long the program took to complete the level). Additionally, it shows the highest score received. From here, one can go to the next level, play the level again or display the equivalent C code. The "Display equivalent C code" converts the visual programming code to actual C code.

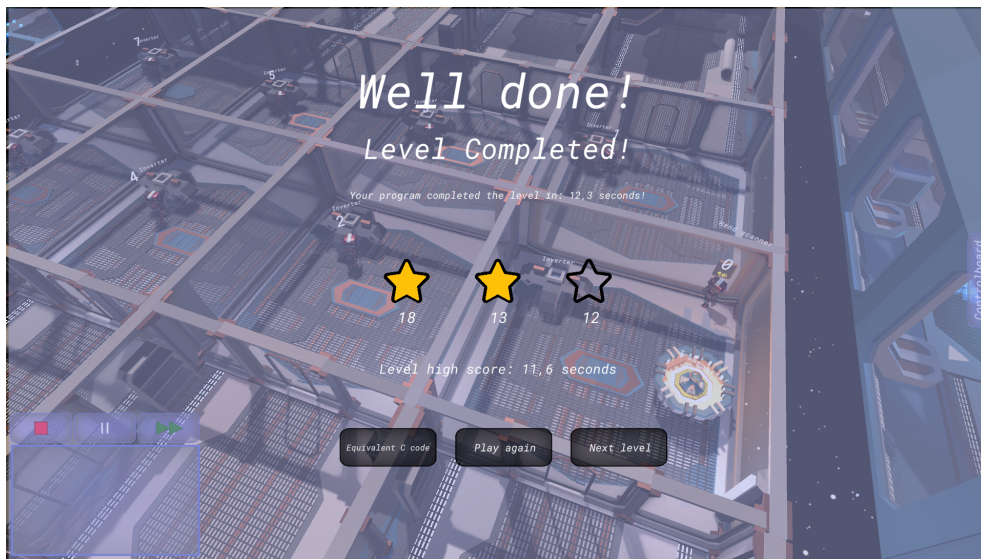


Figure A.8: Panel that opens when player completes a level

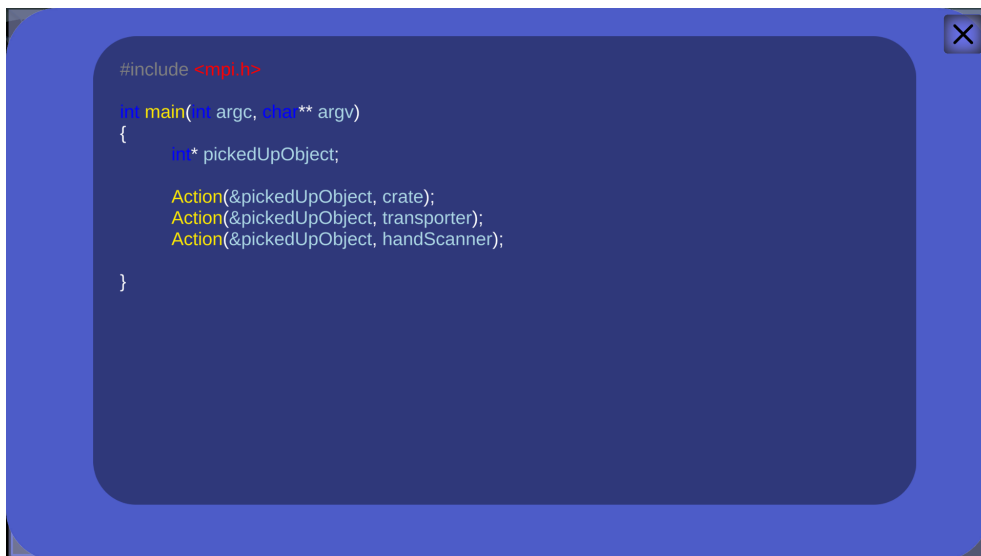


Figure A.9: Converts visual programming code to C code. This equivalent C code is converted from A.4

In level 1, there are only one robot, one control and three arguments unlocked. As one progresses through the levels, more controls, robots and arguments are unlocked. Additionally, the levels will be increasingly difficult. After level 1, one will learn that robots read from the same program. So one will have to be able to distinguish between the robots and send messages with MPI.

A.2 Objects

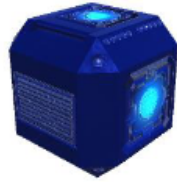


Figure A.10: Crate



Figure A.11: Barrel

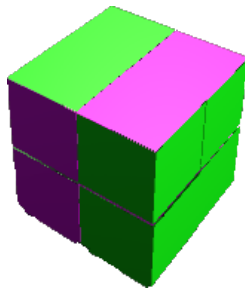


Figure A.12: Cube



Figure A.13: Transporter

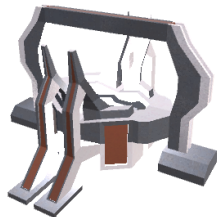


Figure A.14: Transformer



Figure A.15: Inverter



Figure A.16: Hand scanner



Figure A.17: Key



Figure A.18: Robot

A.10, A.11, A.12 are objects the robots are able to pick up. A.14 turns any object placed on it to the color white. A.15 on the other hand, turns any object placed on it to the inverse of its color. The A.13 transports a game-element. It can either be activated or deactivated. It is represented as deactivated if its point-light is not emitting light. A.17 activates a transporter.

Appendix B

Code Snippets

Listing B.1: Function for instantiating control

```
public GameObject InstantiateControl(Control c, Transform parent, bool isActive)
{
    Control control = new Control(c);

    var prefab = Instantiate(Resources.Load("Controls/Control"), parent)
    as GameObject;

    if(isActive)
    {
        var wrapper = Instantiate(Resources.Load("Controls/Wrapper"), parent)
        as GameObject;

        prefab.transform.SetParent(wrapper.transform, true);
        prefab.transform.SetSiblingIndex(0);

        switch(control.type)
        {
            case Control.ControlType.Action:
                Component component =
                prefab.AddComponent<ArgumentControlDraggable>();
                break;

            case Control.ControlType.Logic:

                if(control.Name == "If")
                {
                    component =
                    prefab.AddComponent<ArgumentControlWithBracketDraggable>();
                }
                else
                {
                    component = prefab.AddComponent<ElseControlDraggable>();
                    ((ArgumentControlDraggable)component).isElse =
                    control.Name != "If";
                }

                break;
        }
    }
}
```

```

        case Control.ControlType.MPI:
            if(control.Name == "MPI_Init" || control.Name ==
               "MPI_Finalize" || control.Name == "MPI_Comm_size" ||
               control.Name == "MPI_Comm_rank")
            {
                component = prefab.AddComponent<ActiveControlDraggable>();
            }
            else
            {
                component = prefab.AddComponent<ArgumentControlDraggable>();
            }
            break;

        default:
            prefab.AddComponent<ActiveControlDraggable>();
            break;
    }

    InstantiateCodeLines();
}
else prefab.AddComponent<InactiveControlDraggable>();

prefab.GetComponent<Image>().color = Control.GetColor(control.type);
prefab.transform.GetChild(0).GetComponent<TextMeshProUGUI>().text =
control.ToString();

prefab.GetComponent<ControlDraggable>().control = control;

return prefab;
}

```

Listing B.2: Function for instantiating argument

```

public GameObject InstantiateArgument(Argument argument, bool isActive,
Transform parent = null, int index = -1, bool placeholder = false)
{
    if(parent == null)
        switch(argument.argumentType)
        {
            case Argument.ArgumentType.Operator:
                parent = operatorArgumentsContent;
                break;

            case Argument.ArgumentType.Object:
                parent = objectArgumentsContent;
                break;

            case Argument.ArgumentType.Variable:
                parent = variableArgumentsContent;
                break;

            case Argument.ArgumentType.Integer:
                parent = integerArgumentsContent;
                break;
        }

    var argumentPrefab = Instantiate(Resources.Load("Controls/Argument"), parent)

```



```

as GameObject;

if(!placeholder)
{
    if(isActive) argumentPrefab.AddComponent<ActiveArgumentDraggable>();
    else argumentPrefab.AddComponent<InactiveArgumentDraggable>();

    argumentPrefab.GetComponent<ArgumentDraggable>().argument = argument;

    if(index != -1)
        argumentPrefab.transform.SetSiblingIndex(index);
}

argumentPrefab.transform.GetChild(0).GetComponent<TextMeshProUGUI>().text =
argument.name;

argumentPrefab.transform.GetComponent<Image>().color = argument.GetColor();

return argumentPrefab;
}

```

Listing B.3: Function for checking whether an If control is true or false

```

private void IfSentence(List<Argument> arguments)
{
    if(arguments.Count < 3)
        return;

    var variableOrInteger = arguments[0];
    var argumentOperator = arguments[1];
    var variableOrInteger2 = arguments[2];

    var variableOrIntegerValue = variableOrInteger.argumentType ==
Argument.ArgumentType.Variable ?
caller.GetType().GetField(variableOrInteger.name.ToLower()).
GetValue(caller).ToString() : variableOrInteger.name;

    var variableOrIntegerValue2 = variableOrInteger2.argumentType ==
Argument.ArgumentType.Variable ?
caller.GetType().GetField(variableOrInteger2.name.ToLower()).GetValue(cal
ler).ToString() : variableOrInteger2.name;

    arguments.RemoveRange(0, 3);

    bool sentenceTrue = false;

    if(argumentOperator.name == "%")
    {
        var moduloValue = int.Parse(variableOrIntegerValue) %
int.Parse(variableOrIntegerValue2);
        var operatorValue = arguments[0].name;
        var lastArgValue = arguments[1].name;

        arguments.RemoveRange(0, 2);

        sentenceTrue = CheckSentenceTrue(operatorValue, moduloValue.ToString(),
lastArgValue);
    }
    else sentenceTrue = CheckSentenceTrue(argumentOperator.name,

```

```

variableOrIntegerValue, variableOrIntegerValue2);

if(arguments.Count > 0)
{
    if(sentenceTrue && arguments[0].argumentType ==
Argument.ArgumentType.Operator)
    {
        if(arguments[0].name.Equals("AND"))
        {
            arguments.RemoveAt(0);
            IfSentence(arguments);
        }
        else if(arguments[0].name.Equals("OR"))
            onTrue?.Invoke();
    }
    else if(arguments[0].name.Equals("OR"))
    {
        arguments.RemoveAt(0);
        IfSentence(arguments);
    }
    else onFalse?.Invoke(Mathf.Abs((control.bracket.EndIndex -
control.bracket.StartIndex)));
}
else
{
    if(sentenceTrue)
        onTrue?.Invoke();
    else onFalse?.Invoke(Mathf.Abs((control.bracket.EndIndex -
control.bracket.StartIndex)));
}
}
}

```

Listing B.4: Checking for exceptions in a control

```

private void CheckForException()
{
    List<Argument> arguments = GetArguments();
    string description = "";
    bool foundException = false;

    switch(control.Name)
    {
        case "If":
            for(int i = 0; i < arguments.Count; i+=4)
            {
                var addedIndexes = 0;

                var variableOrInteger = arguments[i];
                if(variableOrInteger.argumentType != Argument.ArgumentType.Variable
                &&
                variableOrInteger.argumentType != Argument.ArgumentType.Integer)
                {
                    description = "A variable or an integer was expected in the if
control, but got " + variableOrInteger.argumentType.ToString();
                    foundException = true;
                    break;
                }
            }
        }
    }
}

```

```
        if(arguments.Count < (i + 2))
            break;

        var argumentOperator = arguments[i + 1];
        if(argumentOperator.argumentType != Argument.ArgumentType.Operator)
        {
            description = "An operator was expected in the if control,
                but got " + argumentOperator.argumentType.ToString();
            foundException = true;
            break;
        }

        if(arguments.Count < (i + 3))
            break;

        var variableOrInteger2 = arguments[i + 2];
        if(variableOrInteger2.argumentType != Argument.ArgumentType.Variable
            &&
            variableOrInteger2.argumentType != Argument.ArgumentType.Integer)
        {
            description = "A variable or an integer was expected in
                the if control,
                but got " + variableOrInteger2.argumentType.ToString();
            foundException = true;
            break;
        }

        if(arguments.Count < (i + 4))
            break;

        var argumentOperator2 = arguments[i + 3];
        if(argumentOperator2.name != "%" &&
            !argumentOperator2.name.Equals("OR")
            &&
            !argumentOperator2.name.Equals("AND") ||
            argumentOperator2.name == "%" &&
            (argumentOperator2.name.Equals("OR") ||
            argumentOperator2.name.Equals("AND")))
        {
            description = "An 'OR' operator or an 'AND' operator was
                expected in the if control, but got " + argumentOperator2.name;
            foundException = true;
            break;
        }

        if(argumentOperator2.name.Equals("%"))
        {
            if(arguments.Count < (i + 5))
                break;

            var variableOrInteger3 = arguments[i + 4];

            if(variableOrInteger3.argumentType !=
                Argument.ArgumentType.Integer &&
                variableOrInteger3.argumentType !=
                Argument.ArgumentType.Variable)
            {
                description = "A variable or an integer was
```

```

        expected in the if control, but got " +
        variableOrInteger3.argumentType.ToString();
        foundException = true;
        break;
    }

    addedIndexes++;

    if(arguments.Count < (i + 6))
        break;

    var argumentOperator3 = arguments[i + 5];
    if(!argumentOperator3.name.Equals("OR")
    && !argumentOperator3.name.Equals("AND"))
    {
        description = "An 'OR' operator or an 'AND' operator was
        expected in the if
        control, but got " + argumentOperator3.name;
        foundException = true;
        break;
    }

    addedIndexes++;

    i += addedIndexes;
    }
}

if(exception != null)
    ExceptionHandler._instance.RemoveException(ref exception);

if(foundException)
    exception = ExceptionHandler._instance.CreateException(description,
    wrapper.transform,
    ExceptionObject.ExceptionType.CompilerError);

break;

default:

    if(exception != null)
        ExceptionHandler._instance.RemoveException(ref exception);

    foreach(var arg in arguments)
    {
        var argType = arg.argumentType;
        if(!control.validArguments.Contains(argType))
        {
            exception = ExceptionHandler._instance.CreateException("Invalid
            argument inserted
            in 'Action'", wrapper.transform,
            ExceptionObject.ExceptionType.CompilerError);
            return;
        }
    }

    break;
}
}
}

```

Listing B.5: Creation of a small red line

```
private RectTransform MakeLine(float ax, float ay, float bx, float by,
Color col) {
    GameObject NewObj = new GameObject();
    NewObj.name = "line from "+ax+" to "+bx;
    Image NewImage = NewObj.AddComponent<Image>();
    NewImage.sprite = lineImage;
    NewImage.color = col;
    RectTransform rect = NewObj.GetComponent<RectTransform>();
    rect.SetParent(transform);
    rect.localScale = Vector3.one;

    SetRectTransform(rect, ax, ay, bx, by);

    return rect;
}
```

Listing B.6: Setting the red line's position and rotation

```
private void SetRectTransform(RectTransform rect, float ax, float ay, float bx,
float by)
{
    Vector3 a = new Vector3(ax, ay, 0);
    Vector3 b = new Vector3(bx, by, 0);

    rect.position = (a + b) / 2;
    Vector3 dif = a - b;
    rect.sizeDelta = new Vector3(dif.magnitude, 2);
    rect.rotation = Quaternion.Euler(new Vector3(0, 0, 180 *
Mathf.Atan(dif.y / dif.x) / Mathf.PI));
}
```

Listing B.7: Calculating the bezier point

```
private Vector3 CalculateCubicBezierPoint(float t, Vector3 p0, Vector3 p1,
Vector3 p2, Vector3 p3)
{
    float u = 1 - t;
    float tt = t * t;
    float uu = u * u;
    float uuu = uu * u;
    float ttt = tt * t;

    Vector3 p = uuu * p0;
    p += 3 * uu * t * p1;
    p += 3 * u * tt * p2;
    p += ttt * p3;

    return p;
}
```


Appendix C

Survey and Interview Questions

C.1 Survey Questions

1. I felt the tutorial taught me everything needed
2. I felt I understood the MPI functions
3. I found the game fun
4. Before playing the game I was interested in Parallel programming
5. After playing the game I was interested in Parallel programming
6. I felt I needed help to complete the game
7. I felt the user interface was easy to use

C.2 Interview Questions

1. Did you feel that the tutorial was enough to understand the content of the game? If no, what can be explained better?
2. Did you find the game entertaining?
3. Would you say that you have an increase or gained interest of parallel programming after playing the game?
4. Did you feel that you learned what parallel programming is about?
5. Was there anything in the game you did not understand?
6. Is there some improvements to the game that can be made?
7. Is there anything you wish were implemented?
8. Did you feel that you understood that all processes read from the same program, and how you can manipulate them to do other things?
9. Did you feel more motivated to get stars when you cleared a level?
10. Was there any MPI-functions you did not understand, and must be explained better?
11. In level 8, MPI_Scatter and MPI_Gather were used to send chunks of an object to other processes and then get them back after the processes have done something to the chunks. Can you think of how this can be used in a real MPI-application?

12. Generally, what do you think of the game?
13. Did you feel that you learned something from the game?

Appendix D

Quiz Questions

⋮

What is the MPI_Init function used for?

- Initializes MPI
- Terminates MPI
- Pausing MPI

Figure D.1: Question 1

What is the MPI_Finalize function used for?

- Terminates the MPI environment
- Initializes MPI
- Determines the size of the group, i.e. retrieves the number of processes

Figure D.2: Question 2

What does MPI_Comm_rank do?

- Retrieves a specific process
- Ranking the best processes
- Retrieves the rank of a process

Figure D.3: Question 3

⋮

What is true for MPI_Bcast?

- Sends a message to all other processes
- Must be run by all processes
- Must only be run by one process
- Other processes must run MPI_Recv, to receive the message

Figure D.4: Question 4

What is true for MPI_Send?

- Sends a message
- Must have a receiver
- Continues if it does not find a receiver
- Must be run by all processes
- Must be run by only one process

Figure D.5: Question 5

What is true for MPI_Scatter and MPI_Gather

- MPI_Scatter must be run by all processes, while MPI_Gather must not
- MPI_Gather must be run by all processes, while MPI_Scatter must not
- Both must be run by all processes
- MPI_Scatter is used to create more processes
- MPI_Gather is used to gather all processes
- MPI_Scatter divides up and sends chunks of an array
- MPI_Gather is used to gain information about other processes

Figure D.6: Question 6

⋮

What is wrong with this code?

```
int rank;  
  
MPI_Init();  
  
if(rank == 0)  
    MPI_Send(3)  
else if(rank == 3)  
    MPI_Recv(0);  
  
MPI_Finalize();
```

- MPI_Comm_rank has not been used
- Wrong argument for MPI_Send
- MPI_Init is used before MPI_Send and MPI_Recv
- There is nothing wrong

Figure D.7: Question 7

⋮

How will this program run?

```
int rank;

MPI_Init();

MPI_Comm_rank(&rank);

if(rank == 0)
    MPI_Send(3)
else
    MPI_Recv(0);

MPI_Finalize();
```

- Absolutely fine
- Process 1 and 2 will deadlock (Never continue)
- Process 0 and 3 will deadlock (Never continue)

Figure D.8: Question 8

⋮

What is wrong with this code, to be able to use MPI_Bcast?

```
int rank;

MPI_Init();

MPI_Comm_rank(&rank);

if(rank == 0)
    MPI_Bcast(0);
else MPI_Recv(0);

MPI_Finalize();
```

- There is nothing wrong
- Wrong argument in MPI_Bcast
- Wrong argument in MPI_Recv
- MPI_Recv should be removed
- MPI_Bcast should be run by all processes
- MPI_Bcast should be run by all processes except process 0
- if statement and else should be removed

Figure D.9: Question 9



What is wrong with this code?

```
int rank;

MPI_Init();

MPI_Comm_rank(&rank);

MPI_Scatter(0);

// Do some stuff

MPI_Gather(0);

MPI_Finalize();
```

- There is nothing wrong
- MPI_Scatter has the wrong argument
- MPI_Gather has the wrong argument
- MPI_Scatter should only be run by process 0
- MPI_Gather should only be run by process 0
- Missing MPI_Recv

Figure D.10: Question 10

⋮

What is wrong with this code?

```
int rank;

MPI_Init();

MPI_Comm_rank(&rank);

if(rank == 0)
    MPI_Sendrecv(1, 0);
else if(rank == 3)
    MPI_Sendrecv(0, 3);

MPI_Finalize();
```

- There is nothing wrong
- Process 0 should send to process 3, not 1
- Nothing wrong after changing from rank == 3 to rank == 1
- All other processes deadlock (Never continue)

Figure D.11: Question 11

⋮

Suppose this program has 4 processes, how will the program run?

```
int rank;

MPI_Init();

MPI_Comm_rank(&rank);

if(rank % 2 == 0)
    MPI_Sendrecv(rank + 1, 0);
else
    MPI_Sendrecv(rank - 1, 1);

MPI_Finalize();
```

- There is nothing wrong
- Process 0 and 1 deadlocks
- Process 2 and 3 deadlocks
- Process 1 and 2 deadlocks

Figure D.12: Question 12

Appendix E

Poster

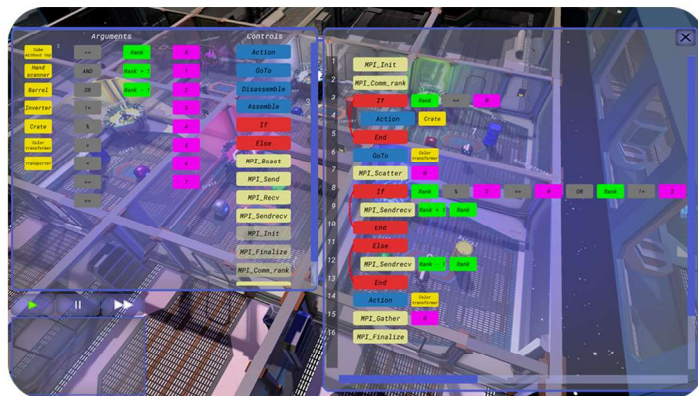
Motivation

- Video games have seized the world by storm
 - Scientists began researching video games
- Researching if video-games can both be educational and entertaining
 - Coining the term «Edutainment Game»
- Games can provide an immersive and interactive experience

Contribution

- Environment for learning parallel programming with the Message Passing Interface (MPI) [1]
 - Simulate MPI-functions in a game
- Simulation of oversimplified image processing
 - Simulation of Border Exchange [2]
- Enable players to understand how parallelism speeds up a program

This is an extension of the work done as part of the author's fall project. Consider standard qualitative game characteristics such as appropriate game difficulty, limiting new game experiences, and concise game tutorial to enhance our MPI game. Cognizant of choice and illustration of MPI-functions, while implementing several new MPI-functions. Also make use of results from a prior user study, as well as a user study for extended work. Study for the extended work determines the game's difficulty and game difficulty. As well as player's perceived learning and interest in field of parallel programming. Technical and conceptual challenges faced when making edutainment game for familiarizing a user with parallel programming, are also described. We discuss our choice of the Unity game engine and how we used an approach where players can visually program inside the game with MPI. Processes are simulated by robots. Game-elements are implemented to abstract out code such as inversion of a pixel color. Users are also able to add and look at underlying MPI-code that controls the robots. MPI-functions simulated include MPI_Init, MPI_Finalize, MPI_Comm_rank, MPI_Send, MPI_Recv, MPI_Bcast, MPI_Scatter, MPI_Gather and MPI_Sendrecv.



Future Work

- Feature levels more closely connected to real MPI problems
 - Reducing number of if-sentences to finish a level
 - Define primary user group
- Tutorial should explain border exchange level more in-depth
 - Tutorial should be more concise
- More in-depth study to evaluate the game's learning effect
 - Find out how to simulate MPI_Comm_size
- Simulate MPI_Comm_rank more similar to the real function
 - Simulate more MPI functions
- Simulate more parallel programming frameworks

References

- [1] M. P. I. Forum, "Mpi: A message-passing interface standard version 3.1," Tech. Rep., p. 1. [Online]. Available: <https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf>.
- [2] F. B. Kjolstad and M. Snir, "Ghost cell pattern," in Proceedings of the 2010 Workshop on Parallel Programming Patterns, ser. ParaPLoP '10, Carefree, Arizona, USA: Association for Computing Machinery, 2010, ISBN: 9781450301275. DOI: 10.1145/1953611.1953615. [Online]. Available: <https://doi.org/10.1145/1953611.1953615>.

