Scientific Research Publishing

Journal of Computer and Communications, 2022, 10, 197-223
https://www.scirp.org/journal/jcc
ISSN Online: 2327-5227
ISSN Print: 2327-5219

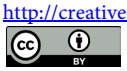
Design and Implementation of a Programming Learning Support System Using the Concepts of Physical Visualization and Serious Gaming

Noriyuki Ishiguro, Yusuke Saitoh, Kazuhiro Yamamoto, Hirohide Haga

Graduate School of Science and Engineering, Doshisha University, Kyoto, Japan
Email: nishiguro@ishss10.doshisha.ac.jp, ysaitho@ishss10.doshisha.ac.jp, kyamamoto@ishss10.doshisha.ac.jp, hhaga@mail.doshisha.ac.jp

How to cite this paper: Ishiguro, N., Saitoh, Y., Yamamoto, K. and Haga, H. (2022) Design and Implementation of a Programming Learning Support System Using the Concepts of Physical Visualization and Serious Gaming. *Journal of Computer and Communications*, **10**, 197-223.
https://doi.org/10.4236/jcc.2022.1011013

Received: October 8, 2022
Accepted: November 27, 2022
Published: November 30, 2022



Abstract

The purpose of this research is the design and implementation of a support system for learning programming. To archive this purpose, in this article, we propose a Puzzle Programming System that uses jigsaw puzzles as an example of the application of physical visualization, which visualizes logical constraints to physical ones. This Puzzle Programming System aims to teach basic programming concepts by presenting the invisible constraints of programming language syntax using the visual constraints of jigsaw puzzle pieces. This system runs on an Apple iPad and was developed using the Unity game engine. We used YAML as a data format for serializing structured data for data management. By inviting high school students to try out a prototype, we could confirm the usefulness of the Puzzle Programming System. The experimental evaluation results also shed light on aspects of the game that need to be redesigned and parts where the visual programming model needs to be modified and expanded.

Keywords

Serious Gaming, Augmented Reality, Visual Programming, Constraint Visualization

1. Introduction

In this article, we discuss the development of a visual programming environment called “Puzzle Programming” where it is possible to write programs using a jigsaw puzzle metaphor, the use of this environment for teaching high school students, and the results of evaluating it with a questionnaire survey. By using the familiar metaphor of a jigsaw puzzle, we were able to get the students moti-

vated and interested in programming. From the results of this evaluation, it became clear that we could prove the effectiveness of our proposed concepts. Furthermore, it becomes clear that we need to redesign the game system and modify/expand its visual programming capabilities.

The system described in this article is based on the concept of using physical entities as a means of visualizing invisible constraints. This is discussed in the results of applying the concept to programming education. This system adopts the concept of "physical visualization". This is our original concpet. This is the most innovative point of our system. Physical visualization is the use of physical constraints to represent invisible constraints such as logical constraint conditions. For example, programming languages and natural languages like English are bound by invisible grammar constraints. However, these constraints can be visualized by representing them as physical constraints, such as the shapes of connections between separate parts. In this article, we describe the application of this concept to the teaching of a programming language. The syntax of a programming language is invisible, but can be visualized using the shapes of jigsaw puzzle pieces. To keep the students interested, we used a combination of programming and serious gaming. Serious gaming refers to the use of a game framework for purposes other than entertainment. For example, it could refer to the use of a game engine to accomplish certain tasks in fields such as education and healthcare. The Puzzle programming proposed in this article is a system that uses a game to teach programming.

2. Related Works

2.1. Visual Programming Languages (VPLs)

A VPL is a language for programming by performing visual operations, as opposed to conventional text-based programming languages [1]. But before students can start learning using a VPL, they have to understand what constitutes a program, and have to consider learning ordinary text-based programming languages in the future. Consequently, there is a need for a new programming language that can make programming simpler and easier to understand, but occupies the space between text statements and existing VPLs.

There are a variety of different types of VPL, but in this study we classify them into the following two types. One is the "diagram" type, where programming is done by joining up arrows between shapes containing text statements in the form of a flowchart, and the other is the "block" type, where programming is done by combining icons or blocks in vertical or horizontal configurations. An example of the former is LEGO Mindstorms [2] as shown in **Figure 1**, and an example of the latter is MIT Media Lab's Scratch [3] as shown in **Figure 2**. A common feature of both is that the flow of a program can be understood at a glance. It can thus be seen that the overall flow can be grasped by representing programming in a visual way. Since a VPL allows basic operations to be performed using other input devices instead of a keyboard, it has the advantage of allowing people without computer experience to concentrate on coding.

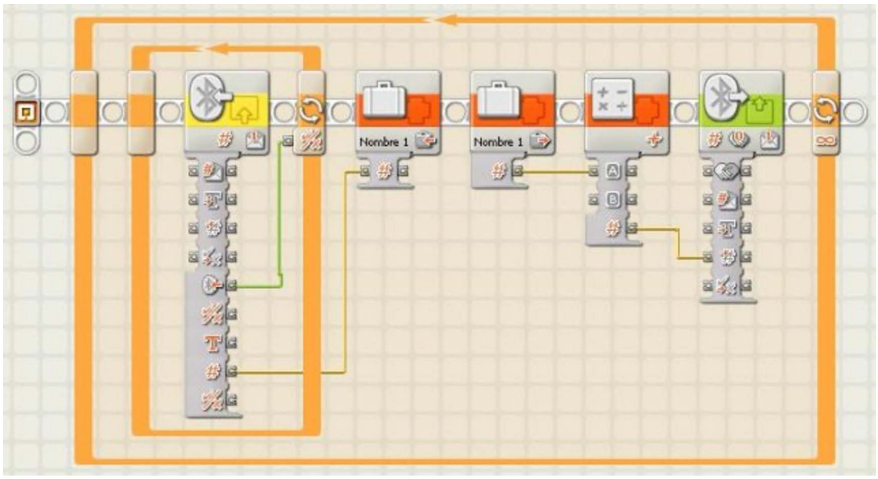


Figure 1. Mindstorm screenshot.

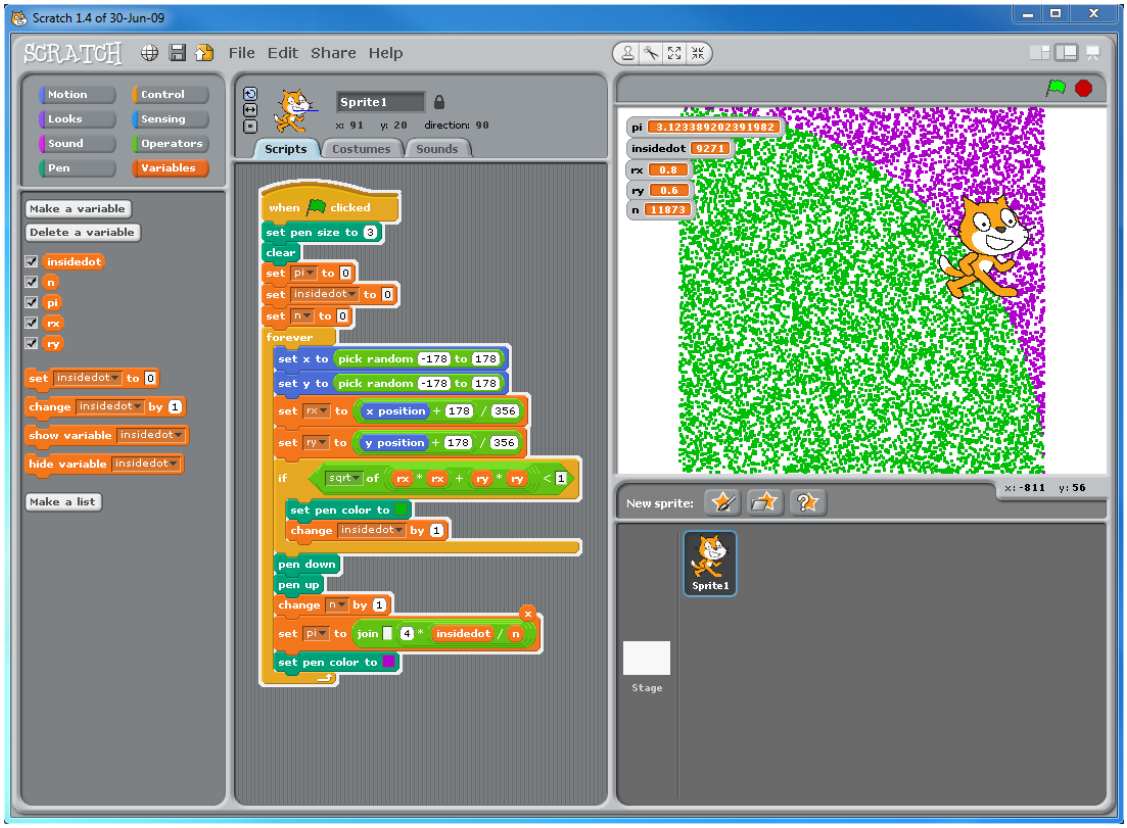


Figure 2. Scratch screenshot.

Unlike an ordinary text-based programming language, a VPL is a language designed with limited functions and represents each process and program visually to compensate the user's programming skills.

2.2. Serious Gaming

A serious game is defined as a digital game used to solve problems in education and various other areas of society [4]. The term "serious game" originates from

the 1970 book "Serious Games" written by a social scientist Clark Abt [5], which advocates the use of games for education and for conveying information. The definition of serious games also includes the following basic premises:

- That digital games can be used for other purposes besides entertainment, and are useful for solving social issues.
- That digital games can also be considered for use in applications outside of education, beyond the framework of conventional educational games.
- That it is essential not only to develop game software but also to develop methods for using this software.

For the "digital native generation" that has grown up surrounded by information technology, serious games are recognized as an effective way of improving motivation, concentration and consistency. There are already several examples of serious gaming for various fields such as healthcare, environment, SDGs and so on [6].

3. Basic Concepts of Puzzle Programming

In this section, we discuss Puzzle programming and the design of two serious games—a Maze game, and a Robot game.

3.1. Overview of Puzzle Programming

Puzzle programming is an educational tool that allows people to experience programming by combining jigsaw puzzle pieces. The syntax constraints of programming languages are represented by the shapes of jigsaw puzzle pieces. This system thus embodies the concept of "physical visualization" [7]. In the following, we refer to this system as "Puzzle programming".

Puzzle programming is an application designed to run on smart phones and tablets. Using the touch screen interfaces of these devices, it gives players the impression that they are actually solving a puzzle. By making a design that more closely resembles a real puzzle, we aim to borrow the entertainment value of puzzles and reduce the feelings of resistance that novices tend to experience when programming for the first time.

The Puzzle programming operation methods, the puzzle piece matching effects, and a comparison with existing VPLs are discussed below.

3.1.1. Operation

Figure 3 and Figure 4 show the system's main screen. The interface consists of three parts: a selector for choosing pieces on the right (blue part), the gray cells (squares) where pieces can be placed on the left, and a red pop-up menu at the top. Puzzle programming has three games: a "programming" game that outputs a log, and two serious games (a "maze" game and a "robot" game). Before starting the programming game, the user can use the game type menu at the top of the screen to specify the type of game to use for programming.

The selector (blue part on the right) contains puzzle pieces of various shapes and colors arranged into three categories (functions, conditions, and variables).

Figure 3. Puzzle programming look and feel.

Figure 4. Puzzle programming screenshot.

The user can switch between categories by tapping the blue button at the top of the selector. As shown in **Figure 5**, the functions category consists of "function" pieces, to which are assigned APIs (application program interfaces) corresponding to each game type (programming, maze game, robot game), and "parameter" pieces corresponding to these function pieces. The conditions category consists of an "if" piece that performs conditional judgment processing, and a "while" piece that performs iterative processing. The variables category consists of "variable substitution" pieces that are used when using variables, and "computation" pieces, that perform arithmetic operations using variables and numbers.

When a piece is dragged from the selector, the color of cells where the piece can be dropped changes to green (**Figure 6**), and pieces can be assembled through the simple operation of dropping them onto these green cells. When a piece that has been dropped onto a cell is dragged again, it can be moved to a different cell or removed from the puzzle by dropping it outside the cell area. Using the pop-up menu at the top of the screen (**Figure 4**), it is possible to delete all of the pieces that have been placed in the puzzle. The pop-up menu also includes functions for saving and loading programs assembled from puzzles (**Figure 4**).

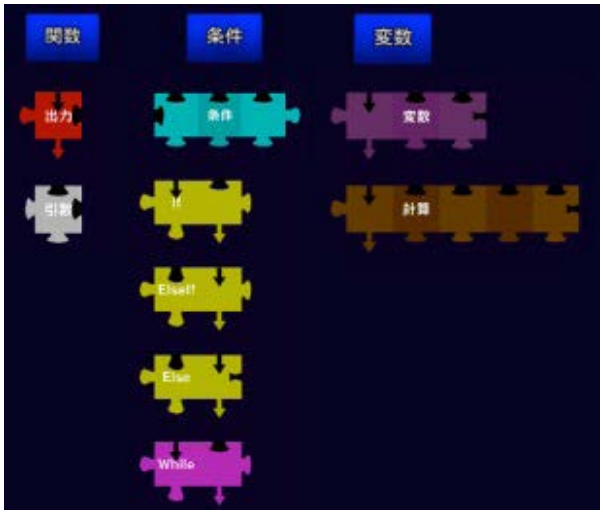


Figure 5. Categorization of pieces.

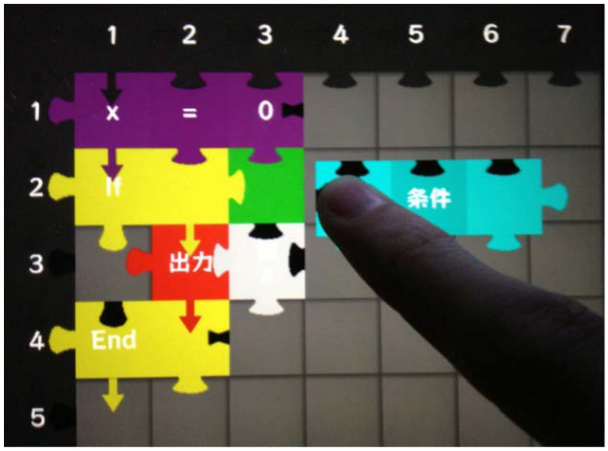


Figure 6. Green cells available for placement.

As shown in **Figure 7**, a program assembled from a puzzle can be started up by pressing the "Run" button in the pop-up menu (**Figure 4**). When the game type is "Programming", a log output screen is displayed (**Figure 8**), and the logs of the assembled log program are output sequentially. The outputs of the "Maze" and "Robot" games are described in Section 3.2 ("Cooperating with serious games").

3.1.2. Piece Matching Effects

To write a program consisting of text statements, the programmer must first remember the programming rules in order to write the program based on the language's grammar. These rules and grammar are invisible constraints. For a beginners, they are said to present the biggest barrier to learning how to program. Many learners fail to overcome this barrier and become discouraged. However, Puzzle programming can solve this sort of problem.

Puzzle programming uses a puzzle as an interface for the creation of programs. Apart from their entertainment value, puzzles also have the function of

Figure 7. A program assembled from a puzzle.

Figure 8. Log output screen after running the program (game type: Programming).

playing the role of a piece matching function, which plays an important role in programming. Here, a piece matching function refers to the effect of conveying programming rules visually by representing the invisible constraints of language grammar and programming rules using the physical constraints of the shapes of puzzle pieces.

Figure 9 shows an example. An "If" piece and a "Condition" piece used for conditional judgment processing are designed with shapes that fit together. When two pieces fit together in the Puzzle programming game, it means that the combination of these pieces conforms to the rules of programming. As a result, these two pieces can be placed next to each other. On the other hand, pieces that do not match are naturally incapable of being placed next to each other. In Puzzle programming, being unable to place pieces next to each other means that

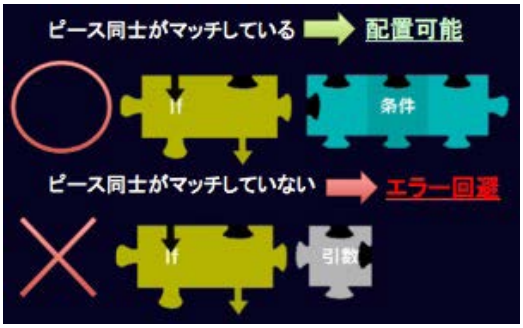


Figure 9. Puzzle programming example.

they break the rules of programming. Therefore, it becomes possible for users to learn programming while gaining an unconscious understanding the concepts of grammar. Eliminating error messages helps to keep novice programmers motivated, and provides a short cut to learning programming.

3.1.3. Comparison with Existing VPLs

Compared with existing VPLs, Puzzle programming is designed with the idea of helping people to learn ordinary text-based programming languages in the future. As stated above, unlike existing VPLs that make programming as simple and easy to understand as possible, this application is a programming education tool that exists between text-based programming and VPLs. In the following, we discuss the novel aspects of Puzzle programming compared with existing VPLs.

We will use "Scratch" [3] as an example of an existing VPL. In the iterative processing block (**Figure 10**, right side) for a "for" statement that means iteration (**Figure 10**, left side), the "condition" for specifying how many times to repeat the loop is already attached. Since the user doesn't have to provide the condition, it may become harder to associate the visual programming model with original programming rules such as the need for a condition statement whenever performing iterative processing.

On the other hand in Puzzle programming, as shown in **Figure 11**, the user is provided with a While piece that performs iterative processing, and a blue Condition piece (different from the While piece), and can perform iterative processing by combining the two. Therefore, from the act of combining pieces together, it is possible to strengthen the associations with the original programming rules, such as the necessity of a condition piece for every While piece, by visually conveying the relationships between two pieces. In the future, as the learner moves towards programming languages based on text statements, if the need for a condition in iterative processing is firmly established in the learner's mind, then the learner should be able to create conditional statements without difficulty.

In Puzzle programming, as shown in **Figure 12**, the "If" piece is designed in a similar way to the While piece. Also, output pieces that correspond to functions can be provided with separate "Parameter" pieces instead of making functions that are pre-associated with parameters. This results in a design that is more like a real programming language with text statements.

```
for (int i = 0; i < v.length; i++) {
    v[i]++;
}
```

Figure 10. The iteration programs (Java and Scratch).

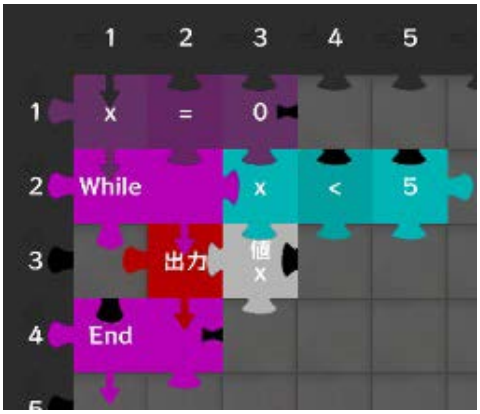


Figure 11. An iteration program (Puzzle programming).

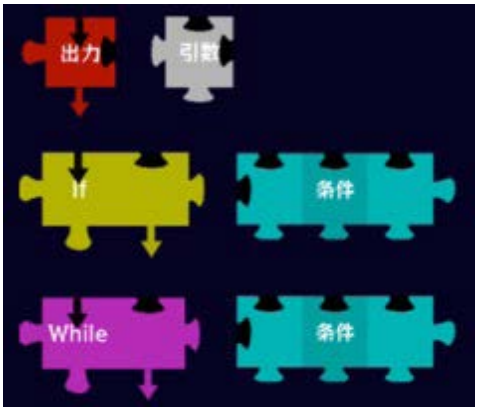


Figure 12. Design of pieces in the Puzzle programming system.

From game design measures such as these, we consider that the barrier to transitioning to text-based statements may be reduced compared with existing VPLs.

3.2. Cooperation with Serious Gaming

In programming education, the introduction of serious gaming has a tremendous effect on aspects such as improving motivation, concentration and persistence. Therefore, with the cooperation of Puzzle programming, we developed two serious games called "Maze game" and "Robot game". By using a serious game to display the execution of programs crafted by the user as the actions of a character, we aim to impress the user and inflate the image of how characters that appear in games around the world are driven by programs. The contents of the Maze and Robot games are described below.

3.2.1. Maze Game

The Maze game (Figure 13) is a problem-solving game that trains the user in the logical thought that underpins a program. The gameplay consists of clearing a series of stages by guiding the character around obstacles between a starting point (the blue panel) and a goal (the red panel) (Figure 14). If the character collides with an obstacle while moving, or travels outside the game area, then the word "Fail" is displayed at the top of the screen (Figure 15).

As an operating procedure, first in Puzzle programming (Figure 16), while checking the stage at the bottom left of the screen, programming is performed by

Figure 13. Maze game screenshot.

Figure 14. Clear screen.

Figure 15. Fail screen.

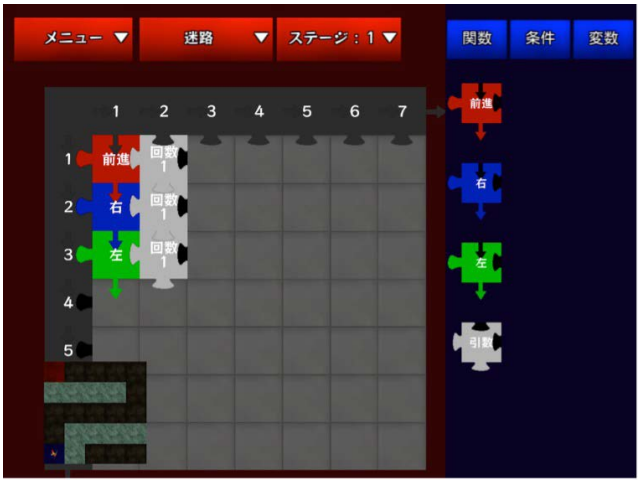


Figure 16. Puzzle programming screen (game type: Maze game).

combining three function pieces (move forward, turn right, turn left) that move the character. By placing a white parameter piece beside each function piece, it is possible to specify the number of times the character should move forward, or the number of times it should turn 90 degrees to the left or right. After finishing programming, pressing the Run button in the menu switches from the Puzzle programming screen to the Maze screen (**Figure 16**), where the character starts moving.

At the current state, four stages of increasing difficulty (stages 1 - 4) are provided to gradually improve the logical thinking.

- In Stage 1 (**Figure 17**), the key is the concept of "sequential execution", which is one of the three main components of programming, and if this is kept in mind then the problem can be solved.
- At Stage 2 (**Figure 18**), it is possible to use While (iteration) pieces without making improper use of the Function pieces.
- At Stage 3 (**Figure 19**), variables are set as the parameters of Function pieces for forward movement, and While pieces can be used to increment the value of these variables one at a time.
- At Stage 4 (**Figure 20**), an If piece (conditional judgment) is used in combination with the While piece to judge whether or not the iteration is odd-numbered or even-numbered, and a decision can be made to turn the character to the right or left accordingly.

By gradually increasing the level of problems in this way, we can keep the user interested by stimulating the user's ambition to solve all the problems, thereby naturally improving the user's programming ability and logical thinking skills.

3.2.2. Robot Game

The Robot game (**Figure 21**) was produced by developing LEGO Mindstorms [1] in a 3D game environment. Like the Maze game, the user runs a program assembled in Puzzle programming to make a robot perform actions such as moving forward or turning to the left or right. Since the obstacles in the robot's path

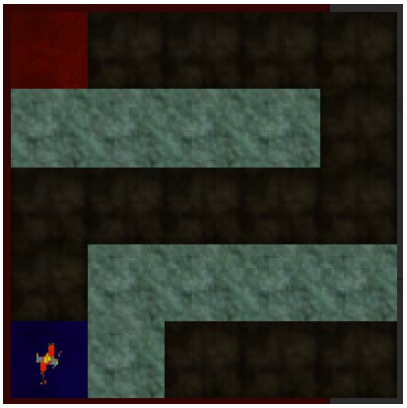


Figure 17. Stage 1.

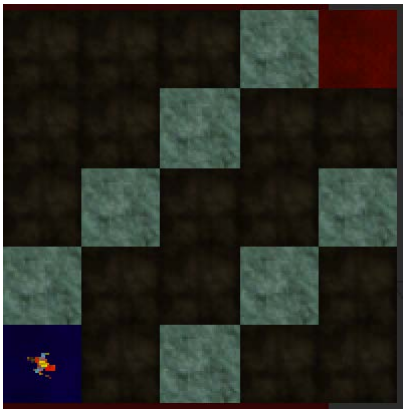


Figure 18. Stage 2.

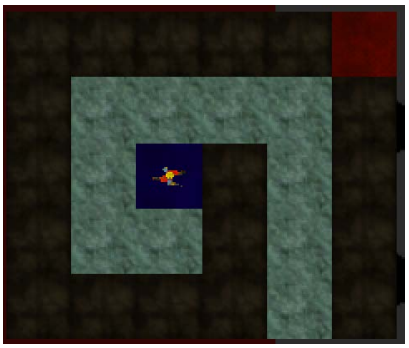


Figure 19. Stage 3.

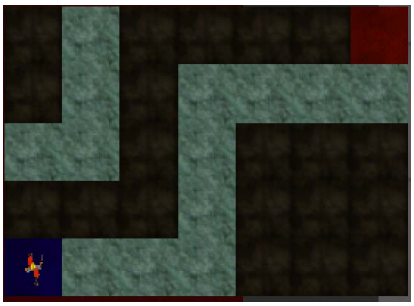


Figure 20. Stage 4.

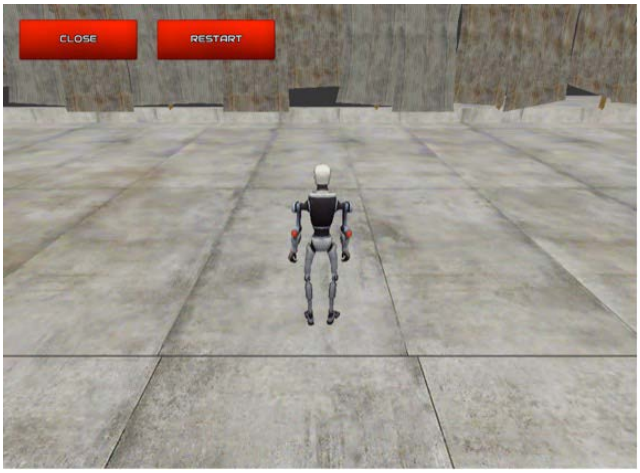


Figure 21. Robot game screenshot.

are not placed around it, this game allows greater freedom of movement than the Maze game. As shown in **Figure 22**, by placing a white Parameter piece beside a Function piece, it is possible to specify parameters such as whether the robot should move forwards or backwards, how fast it should turn, for how many seconds, and so on. Movements that the robot is capable of performing include circular movements, square movements, zigzag movements and figure-of-eight movements, and by using an infinite loop (While piece), it is also possible to repeat the same movements endlessly.

4. Implementation

In this section, we discuss the Puzzle programming development environment and system configuration. Puzzle programming was developed using the Unity game development engine [6]. Unity is an interactive 3D application game development engine provided by Unity Technologies in the United States.

4.1. System Configuration

Figure 23 shows a functional block diagram of Puzzle programming. Puzzle programming consists primarily of puzzle pieces and cells on which these pieces can be placed. The puzzle pieces are associated with code information, and the cells are associated with information about which pieces can be placed on them.

4.1.1. Yaml

In this study, we performed data management using the Yaml text format [8]. Yaml is derived from XML, but is geared more towards data management than to markup languages like XML. Examples of how Yaml is used to represent lists and hashes are shown below.

- List: series of objects separated by "," and sorrounded by bracket.
  example: [apple, orange, grape].
- Hash: pair of *key* and *value* separated by ":"
  example: name: Tom Johns.

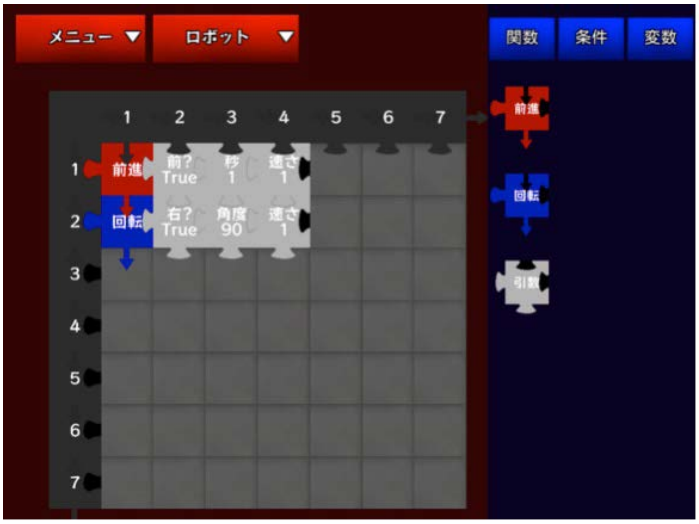


Figure 22. Puzzle programming screenshot (game type: Robot game).

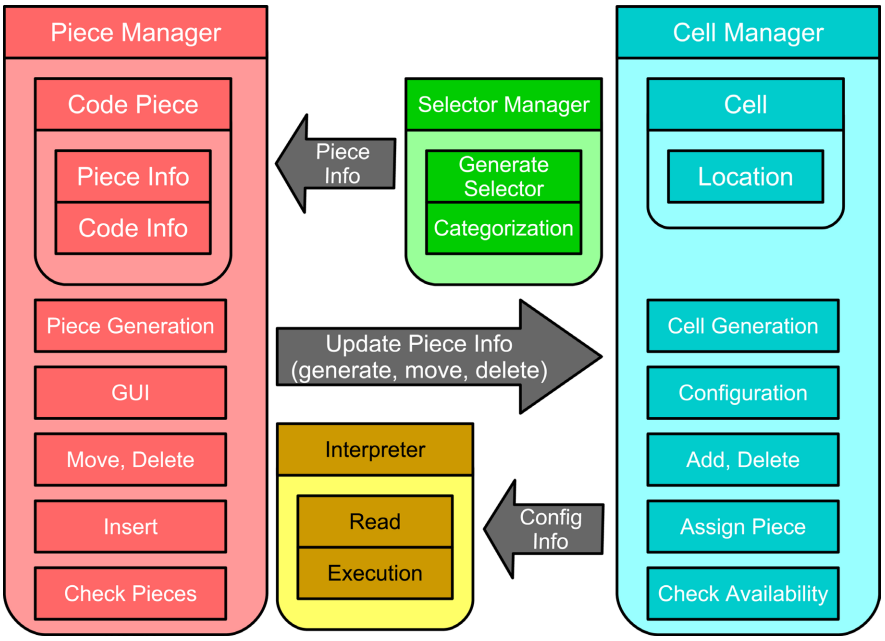


Figure 23. Functional block diagram.

As these examples show, Yaml is highly readable because it is written in text form, making it useful for the management of data that is not too complicated. In this study, we developed and used our own Yaml parser to enable the use of Yaml data within Unity.

4.1.2. Puzzle Pieces

In Puzzle programming, the shapes of pieces play a very important role. A jigsaw puzzle piece as shown on the left of **Figure 24** is actually assembled from parts with peg and gap shapes as shown at the right of the figure. This piece configuration information is managed in Yaml. The text data for the pieces of **Figure 23** is shown below.

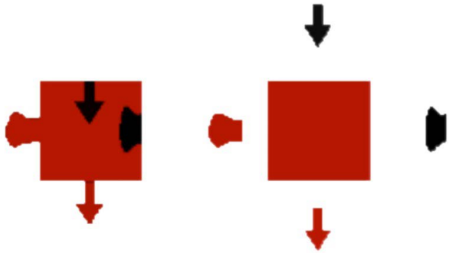


Figure 24. Puzzle piece configuration.

```
id: 4 #Unique ID
condition: [false, false, true, true] # Peg or gap
type: color: [3, 1, 3, 2] # Peggap types/
color: [180, 0, 0, 255] # Color by RGB
```

The keys "condition" and "type" in the above example respectively refer to the top, right, bottom and left edges of a piece. It is possible to create a wide variety of new pieces by providing type data for new shapes.

In this study, a puzzle piece to which code has been added in this way is called a code piece. In practice, the pieces placed on the screen are already code pieces as shown in Figure 24, which are managed with Yaml in the same way as puzzle pieces, and the examples of Figure 24 are represented as text data as shown below.

```
name: output # name of code
piece: [4] #puzzle piece id
class: DebugCode # name of active class
method: Debug # name of active function
p_name: [value] # parameter name
p_type: [System.Single] # data type of parameter
p_def: [1.0] # Default value of parameter
return_type:  # data type of return value
category: 0 # type number
```

When code pieces are run in Puzzle programming, the "class", "method" and "p_type" keys in the above example are used to call functions dynamically. Also, the "category" key is respectively assigned a value of 0, 1 or 2 according to the types of functions, conditions and variables in the piece selection menu at the right of the screen.

4.2. Interpreter

Figure 25 shows a flowchart of the Puzzle programming interpreter processing.

In the Puzzle programming interpreter, based on the current position, the code is executed while referring to the cell at this position and the code piece placed in it.

Cooperation with Serious Gaming

The "Maze" and "Robot" serious games were created and developed as separate

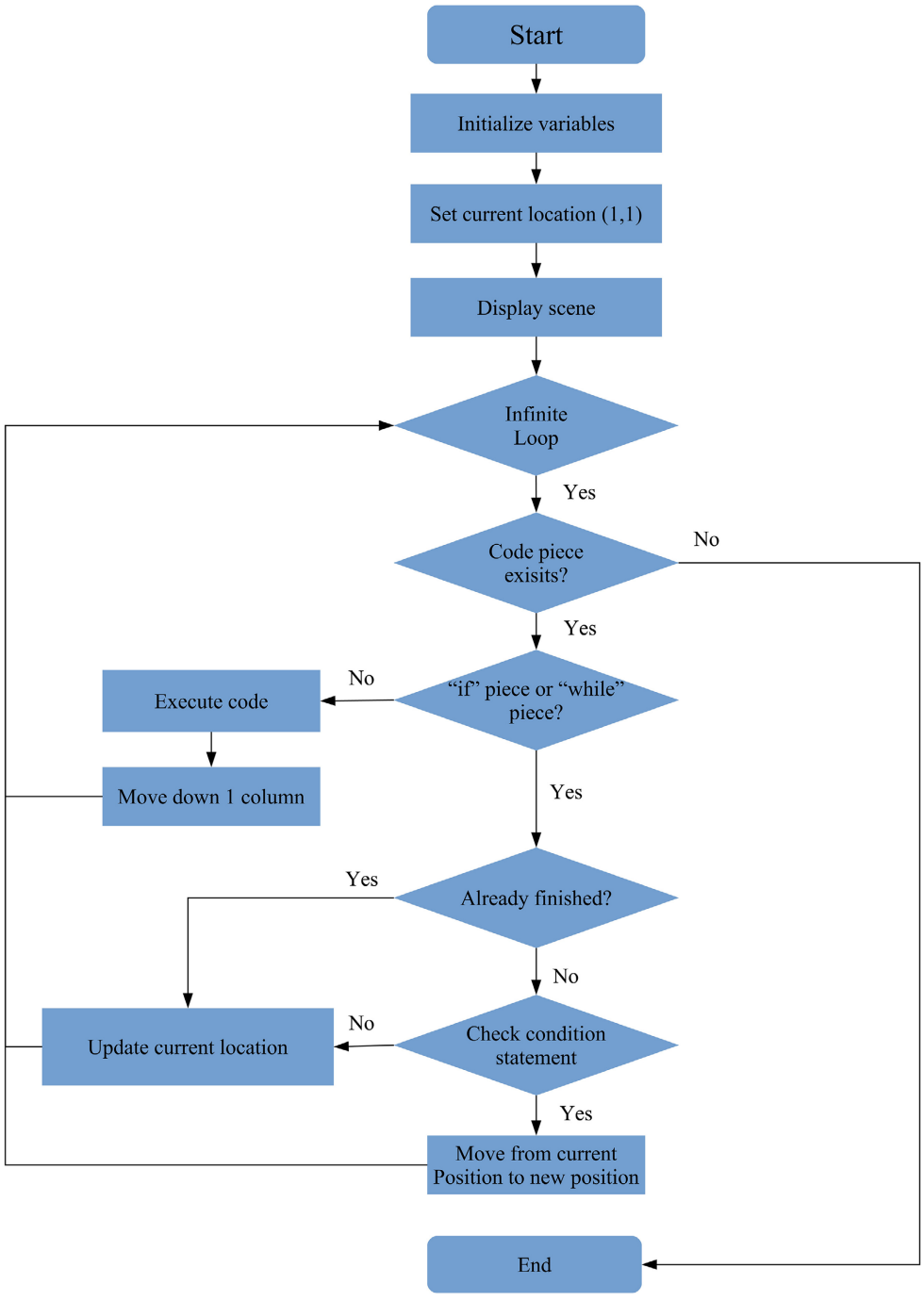


Figure 25. Flowchart of interpreter processing.

scenes in the aforementioned Unity game engine. This is because developing scenes separately makes it easier to debug the game units, and because it was judged that putting all the games in one scene would be a large waste of memory. The scenes created in this way are incorporated into the Puzzle programming scene when the game type is modified in the screen menu. Also, when the game type is changed again, the scene shown before the change is deleted before incorporating the new scene.

The game type data is also managed in Yaml. For example, the Maze game is represented by the following text data.

Maze game Yaml example

```
name :maze # name of game
no_use_code: [RobotMove, etc] # unused pieces
load_screens: Maze # name of screen to be incorporated
stage_button: true # display stage button
stage_names: [1,2,,3,4] # stage names
class: MasterManagerM # class name of stage change
method: ResetStage # function name of stage change
p_type: System.Int32 # stage change parameter
```

The "no_use_code" key in the above example specifies a code piece name that is not used in this game type. This prevents such pieces from appearing on the piece selection screen on the right of the screen. Also, when the stage changes, as in the maze game, there is no need to include the information after the "stage_button" key.

5. Evaluation Experiment

This section discusses an experimental evaluation of Puzzle programming. In the evaluation experiment, the Puzzle programming system was used by high school students who were then asked to fill out a questionnaire. Quantitative assessment is virtually impossible. Only qualitative assessment is possible.

5.1. Experimental Environment and Evaluation Procedure

Followings are experimental environment and evaluation procedure.

Location: Doshisha International High School;

Sample: 27 second year high school students and 33 third-year high school students;

Time: 11:00-12:40 and 13:00-15:10.

The total number of students are 60 and it seems sufficient for reliable evaluation fromt the statistical point of view. The procedure of the experiment (programming experience lesson) is as follow:

1) Pre-test questionnaire for checking participants' knowledge of programming.

2) Short lecture about programming (15 minutes).

3) Demonstration and operation guide for puzzle programming system (20 minutes).

4) Exercise (1.5 hours).

5) Post-test questionnaire for checking the learning effect.

From a preliminary survey conducted to clarify the students' programming knowledge before starting the experiment, we found that almost all of them were novices. Therefore, before explaining the Puzzle programming system, they were

given a brief lecture of the fundamentals of programming, especially its three main components (sequential execution, conditional processing, and iteration). After this lecture, students were asked to solve prepared programming questions in order to improve their logical thinking. At the end of the lesson, a final post-questionnaire was conducted to investigate how their understanding of programming had improved.

5.2. Results

5.2.1. Pre-Questionnaire and Answers

The results of the pre-questionnaire on the test subjects' programming knowledge and background are summarized below.

- **Q1.1:** The question is "*How would you rate your knowledge of programming*?" and the choice options are as follow:
  (a) I know a lot about it.
  (b) I know a little bit.
  (c) I've heard of it.
  (d) I don't know anything about it.
  Answer is in Table 1.
- **Q1.2:** The question is "*Do you have any experience of programming*?" and the result is in Table 2:

A few students in both grades had some programming experience, and were expected to have little difficulty with the Puzzle programming. On the other hand, for students with no experience, we expected to be able to measure the extent to which they became able to do programming after this lesson, or to put it another way, the efficacy of the Puzzle programming system.

- **Q1.3:** The question is "*At the present time, how interested are you in programming*?" and option of choices are shown below.
  (a) Very interested.
  (b) Somewhat interested.
  (c) Not very interested.
  (d) Not at all interested.
  (e) Don't know.

The result of this questionnaire is in Table 3. Overall, there were no students with a strong interest in programming, and we were able to evaluate the extent to which the use of serious gaming can increase interest in programming.

- **Q1.4:** The question is "*Are you interested in making your own apps for smart phones or tablets*?" and the selectable answers are as below (Table 4).
  (a) Very interested.
  (b) Somewhat interested.
  (c) Not very interested.
  (d) Not at all interested.
  (e) Don't know.

It seems that the rapid spread of smart phones in recent years has resulted in many students wanting to develop apps of one sort or another.

Table 1. Results for **Q1.1**.

Student	(a)	(b)	(c)	(d)
2nd year	0	3	14	10
3rd year	0	3	24	6

Table 2. Results for **Q1.2**.

Student	Yes	No	N/A
2nd year	3	24	0
3rd year	1	31	1

Table 3. Results for **Q1.3**.

Students	(a)	(b)	(c)	(d)	(e)
2nd year	0	9	6	3	9
3rd year	2	11	11	4	5

Table 4. Results for **Q1.4**.

Students	(a)	(b)	(c)	(d)	(e)	N/A
2nd year	9	7	4	1	6	0
3rd year	1	13	8	5	15	5

5.2.2. Post-Questionnaire

The results of post-questionnaire are shown as follow.

- **Q2.1:** The question is "*Was it easy to understand how to use* '*Puzzle Programming*'", and the result is in Table 5.

  (a) Very easy.

  (b) Relatively easy.

  (c) Neutral.

  (d) Not so easy.

  (e) Very hard.

Overall, many students answered that the our "Puzzle Programming System" was "easy to understand". This may be because our system enables programming by simply assembling puzzles, rather than text-based programming.

We also gave them additional question "What specific points were easy or difficult to understand? Followings are some sample answers to this question.

- Responses of those who answered "very easy to understand" and "easy to understand".

✧ The fact that programming is done just by assembling puzzles made it fun and easy to understand, even for someone like me who is not good at numbers and calculations. I am not good at numbers and calculations, but it was fun and easy to understand.

✧ It was very easy to understand in terms of operability, because I only had to assemble the pieces as I saw them. I found it very easy to understand in terms of operation.

Table 5. Result of **Q2.1**.

Students	(a)	(b)	(c)	(d)	(e)	N/A
2nd year	2	11	10	3	0	1
3rd year	17	14	1	0	0	1

✧ Since iPad is used, it was easy to operate.

- Responses from those who answered "neutral".

✧ I could understand it when it was explained to me, but it is difficult when I actually operate it myself.

This result shows that programming using puzzles is possible even for students who are not good at numbers and calculations. Some students, such as those who answered "undecided," actually found the operation difficult, so it will be necessary to simplify the interface in the future.

- **Q2.2:** The question is "*How did you feel the look and feel of* '*puzzle programming system*'?"

(a) Very good.
(b) Good.
(c) Neutral.
(d) Not so good.
(e) Bad.

The answer is in Table 6.

Since "Puzzle Programming" was developed for beginners in programming, various "user considerations" were added to the interface. As a result, we believe that these positive results were achieved. We furthermore gave them an additional question "*What specific points were easy or difficult to understand*?"

Followings are some sample answers of above question.

- Responses of those who answered "very easy to understand" and "easy to understand"

✧ The pieces were color-coded, and it was easy to see where a piece could be inserted. The pieces are color-coded, and it is easy to understand.

✧ There are not too many different types of pieces, and they are divided into categories, so it was easy to remember each piece. It was easy to memorize each piece.

✧ The shapes of the arrows and other puzzle pieces were easy to understand and familiar.

- Responses from those who answered "neutral".

✧ When the game type is a maze, the maze stage located at the lower left of the screen is in the way when you assemble the pieces downwards. When the game type is a maze, the maze stage located at the bottom left of the screen is in the way, making it difficult to program the game. I could understand it when it was explained to me, but it is difficult when I actually operate it myself.

The good points that were raised in the responses are the parts that were more

Table 6. Result of **Q2.2**.

Students	(a)	(b)	(c)	(d)	(e)	N/A
2nd year	12	9	6	0	0	1
3rd year	10	17	5	0	0	1

persistent to make the interface simple in the design stage. As for the "neutral" response, it is necessary to take measures such as making the stage screen draggable so that it can be moved freely.

- **Q2.3:** The question is "*After using* "*Puzzle Programming*", *do you understand the three major components of a program* (*sequential execution*, *conditional decision*, *and repetition*)?"
  (a) Well understood.
  (b) Fairly understood.
  (c) Neutral.
  (d) Not understood much.
  (e) Not understood at all.
  The answer is in Table 7.

  This result means that it can be said that the students understood the three major components of a program to some extent. However, while the students seemed to smoothly understand sequential execution and conditional judgments throughout the class, many students stumbled in the incrementing of variables necessary for repetition.
- **Q2.4:** The question is "*After using* '*Puzzle Programming*', *do you understand the rules of the program*, *such as* '*while statement needs a condition*'?"
  (a) Well understood.
  (b) Fairly understood.
  (c) Neutral.
  (d) Not understood much.
  (e) Not understood at all.
  The answer is in Table 8.

  From these results, we can conclude that students naturally learned the "rules of programming" as they became accustomed to programming on "Puzzle Programming". The effect of the ingenious design of the puzzle pieces was significant.
- **Q2.5:** The question is "*Would you like to download Puzzle Programming and try it out yourself when it becomes available*?"
  (a) Absolutely YES.
  (b) Probably YES.
  (c) May be.
  (d) Not so interested.
  (e) Absolutely NO.
  The answer is in Table 9.

  As a developer, we are very pleased to see the large number of third-year students

Table 7. Result of **Q2.3**.

Students	(a)	(b)	(c)	(d)	(e)	N/A
2nd year	1	13	8	5	0	0
3rd year	9	20	3	0	0	1

Table 8. Result of **Q2.4**.

Students	(a)	(b)	(c)	(d)	(e)	N/A
2nd year	1	13	6	7	0	0
3rd year	10	17	4	1	0	1

Table 9. Result of **Q2.5**.

Students	(a)	(b)	(c)	(d)	(e)	N/A
2nd year	3	9	7	5	3	0
3rd year	4	16	6	6	0	1

said "want to try it". We will continue to improve "Puzzle Programming" to enhance its functionality or to improve the gameplay of serious games, with the goal of distributing it in the future.

- **Q2.6:** The question is "*After this class, would you like to try actual programming without puzzles yourself*?"

  (a) Absolutely YES.
  (b) Probably YES.
  (c) May be.
  (d) Not so interested.
  (e) Absolutely NO.

  The answer is in **Table 10**.

Compared to **Table 4** of the pre-questionnarie, the number of respondents who wanted to learn actual programming and develop applications decreased slightly. The reason for this can be attributed to the fact that the respondents learned the difficulty of programming through "Puzzle Programming". However, even taking this cause into consideration, the number of respondents who answered that they "would like to try it" may be regarded positively as a large number.

- **Q2.7:** The question is "*Did this class increase your interest in programming*?"

  (a) Absolutely YES.
  (b) Probably YES.
  (c) May be.
  (d) Not so interested.
  (e) Absolutely NO.

  The answer is in **Table 11**.

Table 10. Result of **Q2.6**.

Students	(a)	(b)	(c)	(d)	(e)	N/A
2nd year	1	7	9	9	1	0
3rd year	1	11	10	7	2	2

Table 11. Result of **Q2.7**.

Students	(a)	(b)	(c)	(d)	(e)	N/A
2nd year	3	15	9	0	0	0
3rd year	8	20	3	1	0	1

Compared to Table 3 of the preliminary questionnaire, it is clear that the number of students interested in programming increased. Therefore, it can be said that this hands-on programming class was a "success" in terms of increasing students' interest in programming, partly due to the effect of the serious games.

- **Q2.8:** The question is "*How did you feel the level of the problems you solved in this class*?"

  (a) Fit to my understanding level.

  (b) Easy.

  (c) A little bit difficult.

  (d) Very difficult.

  The answer is in Table 12.

Since this was our first attempt to conduct a programming class for beginners, we could not accurately predict the level of programming understanding of the students, and thus created the wrong level of problems. As a result, we believe that many students found the problems "difficult".

- **Q2.9:** The question is "*Please write your opinions and impressions throughout the entire experience.*

  Some selected answers are as follow:

- I was very interested in programming while using this "puzzle programming system". I thought it was very cool to be an app developer, considering that the apps we use on our smartphones today are made in such a complicated way.
- At first, I did not even know what programming was, but through this class, I was able to feel the fun of programming and the joy of solving it. I regretted a little that I should have chosen a science course.
- I always wanted to study programming, but this trial class made me want to study programming even more.
- It would be more interesting if there was a function to convert an application created by puzzle programming into an actual programming language.
- At first, I was not very good at programming languages that use ordinary text descriptions, but puzzle programming is fun and easy to use because you can program as if you were playing a game. I have never been conscious of programming in my daily life, but through the class, I think I understand a little more about what programming is.

Table 12. Result of **Q2.8**.

Students	(a)	(b)	(c)	(d)	N/A
2nd year	0	0	22	5	0
3rd year	17	4	11	0	1

- Students of the university kept an eye on the progress of the class and used simple language to explain things to us amateur high school students, which made the class very easy to understand and enjoyable. I would like to take such a class again.

These impressions indicate that this hands-on programming class was a very good experience for the participants. It is difficult to say that high school students who learn programming will go on to pursue IT careers, as was the case in this class. However, it is of great value in the selection of IT human resources. Even for those who do not go into IT, it is important to understand how applications, software, games, etc. used in daily life are structured and developed. Therefore, we will continue to plan such hands-on programming classes in the future to help many students, especially junior high and high school students, grasp the image of programming, even if only a little.

6. Discussion and Future Works

In this section, we discuss our work based on the experimental evaluation results presented in Section 5.

6.1. Learning Effect

Before carrying out the evaluation experiment (programming trial lesson), we set ourselves the following criteria in order to judge its learning effect: "Were the test subjects able to understand the three main components of programming (sequential execution, conditional processing, and iteration), and the two rules of programming?"

Regarding the first criterion (understanding the three main components of programming), based on the post-experimental questionnaire results of Table 7, we can say that the students accomplished this to some extent. However, considering the state of the students during the lesson, they couldn't really be said to have understood these components without question. This is because although nearly all the students seemed to have a firm grasp of sequential execution and conditional processing, many of them were found to have difficulties with the need to increment variables when performing iteration, and found it hard to answer the problems where iteration was required. This is reflected in the post-experimental questionnaire results of Table 12, where the students responded that the problems they had been set were "difficult". This response was at least partly due to the fact that since this was our first attempt at programming lessons for beginners, we were unable to accurately predict how well the students would be able to understand programming, and set the problems at the wrong

level. It seems that this issue can only be resolved by providing more lessons and allowing the students to spend more time programming so they become more familiar with it. Therefore, when we evaluate the learning effect of the next lesson, we will need to provide more lesson time than in the current experiment.

Regarding the second evaluation criterion (understanding the rules of programming), based on the post-experimental questionnaire results of Table 8, we can say that most students did understand the rules of programming. This can be associated with the fact that the students were able to remember the rules of programming naturally as they familiarized themselves with the use of Puzzle programming. This is probably because instead of having conditions pre-associated with while pieces, we produced separate Condition pieces to match the While pieces so that they could eventually by used to build a program.

We can thus say that our intended learning effect was not achieved in this evaluation test because although the student test subjects understood the rules of programming, they did not fully comprehend the three major components of programming.

6.2. Effects of Serious Gaming

In this experiment, we used two serious games called "Maze game" and "Robot game" in conjunction with the puzzle programming. However, in the initial lesson for second-year students, we didn't make much use of serious games and instead conducted the lesson while the students mainly worked on programming (game type) problems where the assembled program is simply output to a log. As a result, there were many students who felt that the problems were difficult, and none of the students thought the level of difficulty was just right (Table 12).

In the next class with the third-year students, we modified our lesson plan to use serious games in order to focus on fixing this shortcoming. As a result, the students showed more interest in seeing a robot move according to the programs they had created, and found the programming experience more interesting and enjoyable. Since we were able to achieve high levels of motivation and concentration, the third-year students understood the class better than the second-year students, as can be seen from Table 7 and Table 8. Also, as can be seen from Table 12, our data shows that they found the problems less difficult than the second-year students.

It can thus be seen that the introduction of serious games helped not only by increasing the motivation, concentration and persistence of the students, but also by creating a learning environment with a more positive effect on learning.

6.3. Interface

For question 1.1 of the preliminary questionnaire, which asked about the difficulty level of the Puzzle programming operating method, most of the students responded that it was easy to understand, as shown in Table 5. When we asked the students to explain the reasons for this, many of them said it was because they were able to create programs just by putting a jigsaw puzzle together. It was

also said that the device was easy to use because the software ran on an iPad. This is probably because they were able to build the jigsaw puzzle using simple drag-and-drop operations.

There were also many students who responded that the Puzzle programming interface was "very good", as shown in Table 6. As specific reasons for this, the students raised three main points:

- The pieces are color-coded and separated into categories.
- When a piece is being dragged, the possible locations for it are changed to a green color.
- The holes in the pieces have shapes that are familiar and easy to understand, such as arrow shapes.

Aside from the three points mentioned by the students, we also ensured that different types of pieces were clearly differentiated by displaying the name of each piece in Japanese, using as little jargon as possible.

In the future, we hope to continue simplifying the interface wherever possible by concentrating on improving the operability of the system with the needs of such users in mind.

6.4. Future Works

Further work is needed to expand the programming language of Puzzle programming. The current version of Puzzle programming offers little freedom as a programming language, and its capabilities are limited to those of existing VPLs. We therefore think it is necessary to increase the range of pieces to include, for example, iteration control pieces such as "Break" and "Continue". However, instead of just blindly adding new code pieces, we should add only those pieces that are necessary for basic programming knowledge.

Since we were unable to create collaborative serious games in this study, we think it is necessary to reconsider the game performance from first principles. For example, in the "Maze" game, it would be possible to create more complex tasks by providing function pieces that return information such as whether or not there is a wall directly ahead, or the color of the current cell. In the "Robot" game, the robot should be given greater latitude of movement by implementing a 3D game space with various sensors that are found in Mindstorms to perform functions such as tracing lines and detecting objects and walls.

In this study, we evaluated Puzzle programming in a classroom environment. However, this evaluation alone is not thought to be sufficient for measuring its educational effects. Therefore, since it is necessary to measure the amount of programming knowledge gained after using a VPL, the effects of programming education must be measured by trying out different evaluation experiment methods, scales, and durations.

Acknowledgements

Sincere thanks to Mr. Shinji Yamamoto of Doshisha International High School for his support to coducting experimental usage of prototype system.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Boshernitsan, M. and Downes, M. (2004) Visual Programming Languages: A Survey. UC Berkeley Report, UCB/CSD-04-1368. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2004/CSD-04-1368.pdf

[2] Vallance, M., Wiz, C. and Schaik, P. (2009) LEGO Mindstorms Proceedings of the 2009 British Computer Society Conference on Human-Computer Interaction. BCS-HCI 2009, British Computer Society, Cambridge, UK, 159-162. https://www.scienceopen.com/hosted-document?doi=10.14236/ewic/HCI2009.17

[3] Marji, M. (2014) Learn to Program with Scratch. No Starch Press, San Francisco, CA.

[4] Avila-Pesantez, D., Rivera, L.A. and Alban, M.S. (2017) Approaches for Serious Game Design: A Systematic Literature Review. *Computers in Education Journal*, **8**, 1-11.

[5] Abt, C.C. (2002) Serious Games. University Press of America, Lanham.

[6] Growth Engineering Group (2022) 16 Serious Games that Changed the World! https://www.growthengineering.co.uk/serious-games-that-changed-the-world/

[7] Moere, A.V. and Patel, S. (2009) The Physical Visualization of Information: Designing Data Sculptures in an Educational Context. In: Huang, M., Nguyen, Q. and Zhang, K., eds., *Visual Information Communication*, Springer, Boston, MA. https://doi.org/10.1007/978-1-4419-0312-9_1

[8] Telang, T. (2020) Introduction to YAML: Demystifying YAML Data Serialization Format. Independently Published.