applied sciences

MDPI

Article

Evaluating Usability and Accessibility of Visual Programming Tools for Novice Programmers—The Case of App Inventor, Scratch, and StarLogo

Kamil Żyła [1,*], Kinga Chwaleba [1] and Dominik Choma [2]

[1] Department of Computer Science, Faculty of Electrical Engineering and Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland; k.chwaleba@pollub.pl
[2] Faculty of Electrical Engineering and Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland; dominik.choma@pollub.edu.pl
* Correspondence: k.zyla@pollub.pl

Abstract: The current state of the labor market requires modern engineers to acquire programming skills at different levels of advancement and to apply them in multidisciplinary environments. Not all modern engineers will become fully fledged programmers, but sometimes the possibility to use low-code programming environments like LabView or MIT App Inventor will be sufficient. In order to give good foundations for adulthood, schools use elements of visual programming, e.g., Scratch or StarLogo, to enhance the critical and algorithmic thinking of future engineers. This article attempts to answer the question of whether anyone cares about following general accessibility and usability guidelines in the case of solutions like Scratch, App Inventor, and StarLogo. Another goal is to show a set of tools that is successful in such an assessment. The authors used Nielsen's heuristics, followed by analyzing WAVE output and Web Content Accessibility Guidelines compliance. Especially, the last one provides insights usually omitted when evaluating low-code environments. It was found that Scratch and App Inventor are leading solutions in terms of look and feel, functionality, documentation, interface navigation, and memorization. The StarLogo interface, on the other hand, is less friendly in terms of aesthetics and functionality.

Keywords: visual programming; accessibility assessment; App Inventor; Scratch; StarLogo

check for updates

Citation: Żyła, K.; Chwaleba, K.; Choma, D. Evaluating Usability and Accessibility of Visual Programming Tools for Novice Programmers—The Case of App Inventor, Scratch, and StarLogo. *Appl. Sci.* **2024**, *14*, 9887. https://doi.org/10.3390/app14219887

Academic Editor: João M. F. Rodrigues

Received: 2 August 2024
Revised: 21 October 2024
Accepted: 25 October 2024
Published: 29 October 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Modern engineers are frequently forced to acquire programming skills on different levels of advancement and to apply them in multidisciplinary environments. Not all of these engineers will become full-fledged programmers, but sometimes the possibility to use low-code visual programming environments like LabView [1] or App Inventor [2] will be sufficient.

In order to give good foundations in adulthood, schools use elements of visual programming to enhance critical and algorithmic thinking of future engineers so that they could easier grasp the idea and syntax of full-fledged programming languages used in a particular engineering domain. Kordaki [3] presents diverse categories of programming learning activities that could be performed within Scratch. Harvey and Mönig [4] argue that Scratch can go to the university. A special focus is also put on improving computational thinking concepts learning by Scratch [5], STEM (Science, Technology, Engineering, and Mathematics) and robotics learning [6,7] or even deep learning [8]. Comparing to Scratch, StarLogo has a bit limited achievements in context of STEM and computational thinking learning, but still is important for the field [9,10].

Another solution is providing visual environments that would be usable enough to efficiently aid engineering tasks and democratize [11] application development. Such a trend can be seen in the scientific literature as well. App Inventor might be perceived as such

a visual environment, additionally being a proven tool in computational thinking learning [5]. Among others, it was used for the following engineering-like purposes: building GSM (Global System for Mobile Communications)-based home automation systems [12], building Bluetooth-based smart homes [13], selecting machining parameters [14], building an IoT (Internet of Things) application to collect and analyze sensor data [15], building 3-axis filament winding machine control systems [16], and many others.

In parallel there is a pending discussion about positive role of the visual programming environments in education. Few of examples are worth to be mentioned. Pérez-Jorge and Martínez-Murciano [17] investigate the influence of Scratch and App Inventor on the development of skills and competencies for learning in higher education. Among the skills and competences the following are mentioned: autonomy, attention, motivation, critical thinking, creative thinking, computational thinking, communication, problem solving and social interaction. Boldbaatar and Sendurur [18] attempt to observe how the block-based environment (i.e., StarLogo) can facilitate the problems caused by the abstractness of textual programming languages. Papadakis et al. [19] investigate whether App Inventor and Scratch can contribute to increasing interest in computer science classes. Finally, Papadakis et al. [20] and Chang [21] investigate how App Inventor, Scratch and StarLogo can make computer science education more accessible to everyone.

All of these are the reasons why App Inventor, Scratch and StarLogo were used as the case studies in this article. They are one of the leading visual programming environments and the outcomes of their applications are positively presented in the scientific literature. To give just two examples of popularity scale: in November 2023 Scratch reported ~121 million registered users; on 1 October 2024, App Inventor reported that total number of created projects was 108.2 million. Please follow the links to the official reports [22,23] to see detailed statistics. The proof of these solutions being present in scientific literature are the works cited in the previous paragraphs and Section 2.1. At last, another reasons are missing fair usability and accessibility assessments, and being provided and managed for many years by a well-established entity, i.e., Massachusetts Institute of Technology (MIT), Cambridge, MA, USA .

Due to the popularity of low-code visual programming environments in engineering and educational applications, it is worth considering if their accessibility was properly approached. It gains more importance in the context of democratizing application development that will involve a broad spectrum of people, including those with impairments. In the case of App Inventor accessibility, studies focus on developed applications rather than the visual programming environment itself, to mention [24,25]. MIT performed a usability testing of App Inventor [26] and StarLogo [27], although in different ways than the authors. The work [26] focuses on regular users performing a limited set of tasks and analyzing unstructured opinions. The work [27] concentrates on solving a limited set of issues observed by the author. Both works are far to cover the entirety of the solutions. Some insights into Scratch usability can be found in [28] where interaction problems in Scratch collected by the authors in [28] were used within their custom metric evaluation process.

In general, there is a strong need to provide an assessment of low-code programming environments enhancing critical thinking as well as end-user-programming activities performed by engineers not being IT specialists. As one of the solutions the authors propose Nielsen's heuristics accompanied by WCAG compliance analysis. Such assessment is important, because high quality environments when identified and chosen stimulate good outcomes from their usage, among others by providing an interface that is not discouraging for users and allows them to focus on planned tasks.

Identifying challenges and issues concerning such environments might help making them more inclusive. Novice programmers with disabilities are endangered with exclusion the most. At this stage it can be seen, that programmers with sight problems might count on a lot of aid, as technology is basically ready for such facilitations as reading descriptions, acting on voice commands, applying favorable color schemes, and many more. Another area of big importance and many opportunities is aiding navigation by limiting traditional

equipment usage, e.g., mouse. It is worth to check, if the environments are ready to cooperate in such settings.

The research question is whether the proposed set of assessment tools can be used for an accessibility and usability assessment of visual programming tools for novice programmers. Another question is to what extent the chosen leading platforms for novice programmers fulfil the usability and accessibility guidelines that are widely accepted by the computer science society. The outcomes of the assessment will be used to identify the most problematic aspects and to give recommendations when necessary.

2. Materials and Methods

This section describes the proposed accessibility assessment method of visual programming environments for novice programmers. It is accompanied by descriptions of the visual programming environments, Nielsen's heuristics, and WCAG.

2.1. Visual Programming Environments

This section provides a brief overview of the visual programming environments that are the focus of this article. To see their full shape and abilities, the homepages of the chosen environments can be visited. Their purpose is to allow development of applications, while using vendor-specific graphical syntax, instead of textual syntax, like in the case of the classical development approach. They are not intended for professional software development, although they are frequently used for developing so-called "do it yourself" applications, and education.

2.1.1. App Inventor

App Inventor is one of the most popular environments for visual programming of mobile applications that support iOS- and Android-based mobile devices. It is managed by MIT and attracts more than one million users daily. Basically, almost all features of mobile devices are supported, even elements of artificial intelligence and Lego Mindstorms [2]. It applies an event-driven programming paradigm and a puzzle-like visual representation of programming concepts (Figure 1). App Inventor can be applied in science and engineering (programs made by engineers not accustomed to professional programming techniques), fast prototyping, and even teaching [29,30].

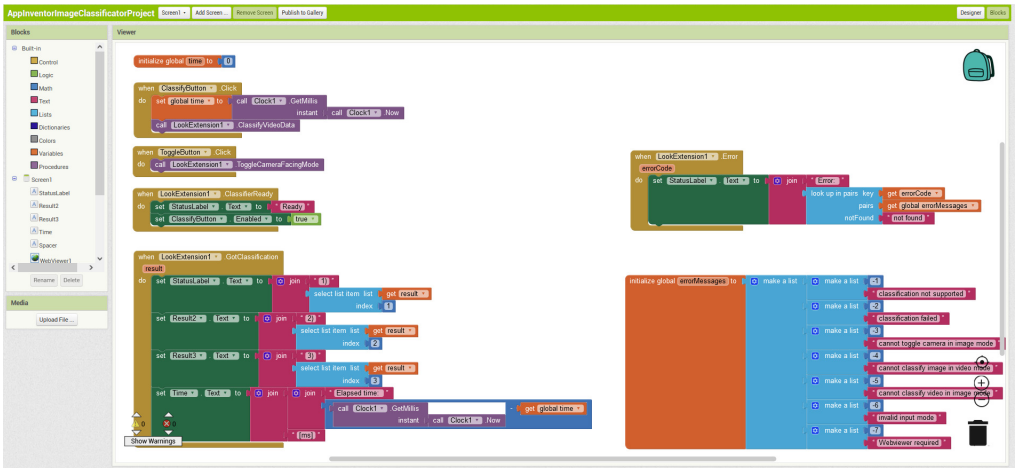


Figure 1. Blocks editor view of the App Inventor visual programming environment.

2.1.2. Scratch

Scratch is another very popular environment for visual programming, managed by the Lifelong Kindergarten Group of the MIT Media Lab. It allows users to create interactive stories, animations, and games using block-based programming and puzzle-like syntax (Figure 2) that is easier to understand and remember than traditional code

writing [31]. With Scratch, users can learn the basics of programming concepts. The main purpose of this environment is edutainment as well as teaching critical thinking and algorithmization [32]. It is simpler than App Inventor in terms of functionality, but more useful in giving foundations for fully fledged programming.

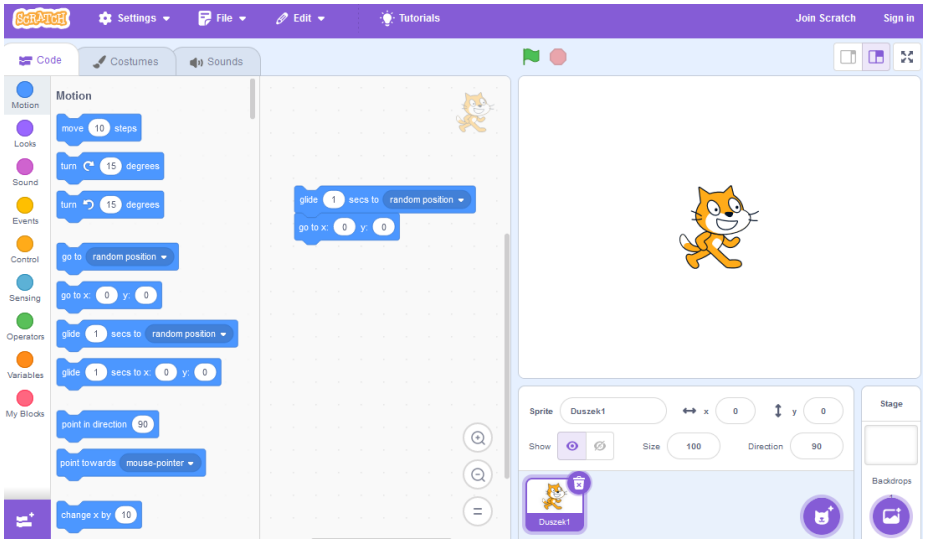


Figure 2. Scratch visual programming environment.

2.1.3. StarLogo

StarLogo is a visual programming environment that was created by Mitchel Resnick and Mitchelle Feinstein from the MIT Media Lab. It allows users to create simulations and models of various systems, such as ecology, financial markets, and traffic-related ones [33,34]. Three-dimensional visualization and game creation are also supported. Block-based programming and a puzzle-like syntax are utilized (Figure 3). StarLogo was primarily intended for teaching critical thinking and algorithmization [10,18,35]. Similar to Scratch, it is simpler than App Inventor in terms of functionality, but more useful in giving foundations for fully fledged programming.

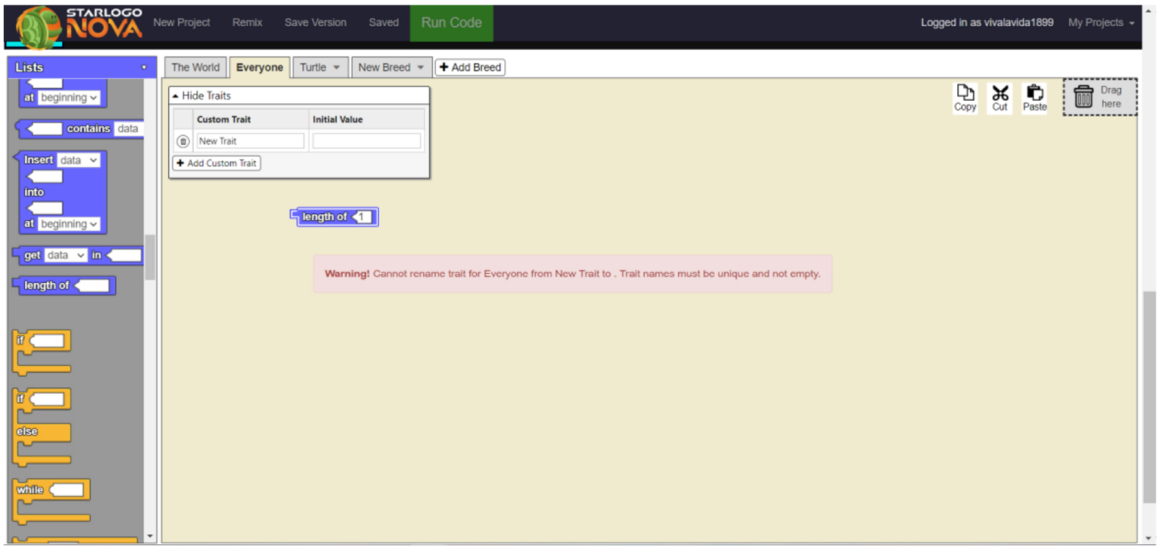


Figure 3. StarLogo visual programming environment.

2.2. Nielsen's Heuristics

Nielsen's heuristics are a set of good practices formulated by Jakob Nielsen that should be followed when designing a user interface. Heuristic analysis based on Nielsen's works allows us to determine the level of interface ergonomics as well as to identify usability issues that can be fixed to significantly improve human–application interaction [36,37].

Nielsen's heuristics are defined in the following ten points [38,39]:

1. Show system status: It is important that an interface remains in constant communication with users and provides them necessary information about what is going on. Examples: Displaying information about content loading, so users know that their connection to the application is not lost; displaying consecutive stages of placing an order in an online store, indicating which stage users are currently at.
2. Keep the consistency between the system and reality: It is not recommended to use a highly specialized language. Technical terms for an average user are enigmatic and can lead to confusion. It is also recommended to adapt to generally accepted standards, especially in the context of graphical user interface elements, such as icons on buttons. Example: Using icons known from old tape recorders and discmans when creating an element responsible for managing an audio or video player. Such icons are commonly known to users because they are used in most players.
3. Give the user full control: It is recommended to allow a user to navigate through the entire interface. Users frequently choose options by mistake, so it is important to support undo and redo operations. Example: When filling in a form and going to the next page, the user should be able to return to the previous step and make corrections without having to fill in the form fields from scratch.
4. Stick to the standards and be consistent: It is recommended to maintain the consistency of the interface in order to allow the user to navigate the site efficiently. Predictable navigation patterns, uniform naming conventions, and coherent semantics should be pursued. Examples: Elements that enable interaction should change (additionally in a similar manner) when the cursor is placed on them, e.g., highlighting a button; and placing the "Cart" icon anywhere but not in the upper right corner of the screen might be seen as a violation of consistency.
5. Prevent errors: It is recommended to prevent errors by applying careful design or providing appropriate hints and suggestions that will help users to perform actions correctly at once. Such an approach might highly limit displaying errors. Example: Displaying product availability to the user before starting the purchase procedure.
6. Show instead of forcing to remember: It is not recommended to force a user to remember past actions. An interface should be intuitive and actions that have been performed before should be easy to reproduce. Users' memory load should be minimized by making objects, actions, and options visible. Example: Providing a browsing history or "recently viewed" section.
7. Ensure flexibility and efficiency: An interface should be designed to meet the needs (and/or speed up interactions with the interface) of different users—novices as well as experts. Examples: A possibility to change the font size; high contrast mode; a variety of navigation options like menus; search engines; and a sufficient interface performance—loading times acceptable for users.
8. Take care of aesthetics and moderation: An interface should not contain irrelevant (or even rarely needed) information, decreasing the relative visibility of relevant information. Example: Users are more likely to use interfaces they find visually appealing and without an overwhelming amount of information.
9. Ensure effective error handling: An interface should provide messages that help to recognize, diagnose, and recover from errors. Error messages in sophisticated technical language should be avoided. Messages clearly informing the user what error has occurred and indicating what action they should take should be preferred. Example: 404 error pages found on various websites.

10. Take care of help and documentation: Despite following the best design guidelines, there might be still situations when a user will need help in using an interface. It is necessary to provide help and documentation that is user-oriented and easy to find. Examples: providing contact data in the footnote, providing "frequently asked questions" subpages.

2.3. WCAG Overwiev

Accessibility analysis is a process that allows for determining whether the interface of a given program or application is available to the widest possible audience, and also people with different backgrounds or limitations such as visual impairments [40]. One such standard, called the Web Content Accessibility Guidelines (WCAG), was developed by the World Wide Web Consortium (W3C). Its first version, known as WCAG 1.0, was published in 1999. The next iteration, WCAG 2.0, was made in 2008, and W3C updated its guidelines with 61 success criteria, continuing to take into account users with disabilities [41].

WCAG 2.2, which is the current version, gives guidelines for making websites more accessible for all users in general, but also focuses on people with disabilities, including accommodations for users with blindness and low vision, deafness and hearing loss, limited movement, speech disabilities, photosensitivity, learning disabilities, and cognitive limitations. These guidelines address web content on desktops, laptops, tablets, and mobile devices in the following four areas [42]:

1. Perceivable—related to providing information and interface components in ways users can perceive. Examples: Providing text alternatives for non-text content that can be changed to speech or braille; and providing adaptable content that can be presented in different ways without losing information or structure.
2. Operable—related to providing interface components and navigation that is operable for users. Examples: Making all functionality available from a keyboard; providing users with enough time to read and use interface content; avoiding designs known to cause seizures or physical reactions; facilitating users in navigation, searching for content, and determining their current location; and facilitating usage of input devices other than a mouse and a keyboard.
3. Understandable—related to providing information and ways of interface operation that are understandable to users. Examples: Making text context readable and understandable; making interface elements occur and operate in a predictable way; and helping users to avoid mistakes and aiding mistake correction.
4. Robust—related to making interface content robust enough to be interpreted by a variety of user agents, including assistive technologies. Example: Taking care of interface compatibility with commonly used user agents and assistive technologies.

The compatibility of web content with the WCAG guidelines might be automatically assessed, which is the already established approach [40,43,44]. A specialized software takes the link as input and produces a compliance report. Advances in this domain concern even automated webpage refactoring methods that improve web accessibility [45,46].

2.4. Assessment Method

In order to answer the questions posed in this study, it was decided to conduct Nielsens' heuristics assessment and automated WCAG compliance analysis. The subjects of the assessment were the interface areas involved in activities regarding development and running a visual code of an application, i.e., infrastructure for visual code management, project management, running and debugging the code, as well as all kinds of menus and predefined shortcuts.

2.4.1. Usability Assessment

Nielsen [47] investigated heuristics assessment effects, depending on the expertise level of evaluators. They were divided into three groups: novice evaluators with no usability expertise, regular usability specialists, and double usability specialists, who

additionally had experience with the interface being evaluated. According to [47], for double specialists, it is enough to have two or three evaluators to find most problems. The accuracy of detection might even be between 80% and 90%. The authors, therefore, decided that six evaluators, being double specialist, would be enough for the purpose of this article.

An assessment procedure of the chosen environments for visual programming started with Nielsen's heuristics. Six double specialists independently evaluated the interfaces of App Inventor, Scratch, and StarLogo according to the Nielsen's heuristics checklist. They gave points from 1 (very bad, heuristic not fulfilled) to 5 (very good, heuristic fulfilled) in each category (a 5-point Likert scale was used). Then, the average was counted and rounded (not truncated) to a full number. After individual assessment, the evaluators discussed each other's findings and agreed on the final comments in each category. The final grade was counted as the sum of the points from all categories.

All evaluators had at least 6 months of experience in using the assessed visual programming environments. They were accustomed with the topics concerning the foundations of visual programming, that include building and using visual languages. They were also users of the evaluated visual programming environments, developing applications fulfilling their own needs. While choosing the evaluators, the highest experience was aimed at. Moreover, persons that could cause conflicts of interest concerning this article were excluded.

The execution of usability assessment was preceded by training. The evaluators were asked to read and understand the heuristics. They were provided with the NN Group [48] and invesp [49] guidelines. A collaborative evaluation of the UML online editor was performed to practice the procedure of usability assessment. Next, the visual programming environments to be evaluated were introduced to the evaluators. No comments concerning the usability of these environments were made at this stage.

Each evaluator performed their evaluation independently of the others, as the goal was to capture observations with no influence from other evaluators. During the evaluation, use of the evaluated web-based visual programming environments for the purpose of visual code management, application projects management, running and debugging code as well as visiting all kinds of menus and using shortcuts was encouraged. These, in general, are typical activities needed for the purpose of developing and running an application code. Moreover, the evaluators were told that they should keep in mind the purpose of the environments, which is not professional application development, but rather developing applications for personal use or education.

The task of each evaluator was to give marks on the Likert scale for each of the heuristics. Again, guidelines [48,49] were provided to help. The evaluators were also asked to provide comments justifying the grade and comments concerning facts/aspects that seemed to be significant for them. Finally, this feedback was discussed and processed for the purpose of presentation as the final results.

2.4.2. Accessibility Assessment

Accessibility analysis is a process that allows for determining whether the interface of a given program or application is available to the widest possible audience, including users with various limitations.

As part of the analysis, the Scratch, App Inventor, and StarLogo environments were automatically analyzed against compliance with the WCAG guidelines, using the WAVE tool [50]. This was possible as all of the visual programming environments were accessible through the web. The authors used WAVE as the tool for the automatic accessibility assessment, checking the compliance of the chosen visual programming environments with the newest WCAG guidelines. WAVE is a proven tool used by other authors for such types of analysis [43,45]. WAVE produced a compliance report that indicated, e.g., errors and alerts related to particular guidelines. The environments were compared for the number of errors and their priority.

The authors decided to check the WCAG compliance with and without the application code embedded (displayed) in the visual programming environment code editor. Testing the empty code editor allowed us to check the influence of displaying visual syntax symbols on the overall results obtained from the compliance checking tool. Moreover, each visual programming environment was tested with the code of three different applications embedded in the code editor area of the interface. Applications published by the community or the visual programming environment provider were selected, making the results more reproducible, as the visual code of applications can be easily found. Another goal was to select a balanced variety of applications to cover the wide spectrum of syntax offered by the visual programming environments, as it is not possible to test all the possible application codes that can be developed using the provided visual syntax.

In the case of the App Inventor (http://appinventor.mit.edu/) web-based visual environment, the following accessibility test cases were addressed:

1. (AP) Visual environment without any application code embedded.
2. (AC1) Visual environment with the "Where is my car?" application code embedded.
3. (AC2) Visual environment with the "CreativeProject" application code embedded.
4. (AC3) Visual environment with the "pongpub" application code embedded.

In the case of the Scratch (https://scratch.mit.edu/) web-based visual environment, the following accessibility test cases were addressed:

1. (SP) Visual environment without any application code embedded.
2. (SC1) Visual environment with the "Avocado Juggle v1.2" application code embedded.
3. (SC2) Visual environment with the "Maze Starter" application code embedded.
4. (SC3) Visual environment with the "Greeting Card" application code embedded.

In the case of the StarLogo (https://www.slnova.org) web-based visual environment, the following accessibility test cases were addressed:

1. (LP) Visual environment without any application code embedded.
2. (LC1) Visual environment with the "Game starter project" application code embedded.
3. (LC2) Visual environment with the "First person action game" application code embedded.
4. (LC3) Visual environment with the "Biology Simulation" application code embedded.

The results obtained from the compliance check were discussed keeping in mind the following four WCAG areas: perceivable, operable, understandable, and robust. Moreover, the number of distinct problems categories and total number of detected problems in each test case were calculated.

3. Results and Discussion

This section presents the results of using the assessment method. First, the results of Nielsen's heuristics regarding particular environments for visual programming are presented; this is followed by WCAG compliance analysis.

3.1. Nielsen's Heuristics

Table 1 presents the results of a Nielsen's heuristics evaluation for the chosen visual programming environments, i.e., App Inventor, Scratch, and StarLogo. It has to be remembered that this work concerns accessibility and usability assessment, so the exact purpose and advancement of the environment does not significantly affect the assessment.

The comments provided in the subsequent sections are not only for the purpose of describing issues, but also for describing the current shape of the assessed environment. This might be helpful for people who are thinking of taking advantage of the visual programming concept, but still looking for a suitable solution.

It can be seen that ensuring flexibility and efficiency is the most problematic issue for all of the assessed solutions. The authors suppose that this is the result of visual syntax usage, which affects flexibility due to the higher abstraction level of visual syntax elements. It is also supposed that the reasons behind the mediocre score regarding effective error handling

are syntax simplifications, and a common fact of tradeoffs between editor simplicity and providing its mandatory functionality. At the same time, these inconveniences might not be perceived as problematic by novice programmers, as the joy of rapidly seeing usable results (i.e., applications) may cover/outweigh the negative aspects of their development. In that matter, further research is needed. The very low score of StarLogo regarding help and documentation is surprising, although it might be an unexpected outcome of some policy of the provider. In general, the high popularity of the environments, continuous provider support, and thriving communities find reflection in high usability scores and the small number of issues described in the comments.

Table 1. Aggregated results of Nielsen's heuristics evaluation.

	Heuristics	Visual Programming Environments		
		Scratch	**App Inventor**	**StarLogo**
1.	Show system status	5	5	4
2.	Keep the consistency between the system and reality	5	5	3
3.	Give the user full control	4	4	4
4.	Stick to the standards and be consistent	5	5	4
5.	Prevent errors	4	4	4
6.	Show instead of forcing to remember	4	4	3
7.	Ensure flexibility and efficiency	3	3	3
8.	Take care of aesthetics and moderation	4	4	2
9.	Ensure effective error handling	3	3	3
10.	Take care of help and documentation	5	5	1
	Total	42	42	30

3.1.1. Show System Status

Scratch fully informs the user about the system status. Simple and clear messages are used. Users are informed about the section they are in. In addition, after pointing the mouse cursor at a particular element, they receive feedback about the function of this element. An important element of indicating the system state to users is informing them about unsaved changes when they try to perform an action that could potentially lead to the loss of work done.

App Inventor informs the user to a similar degree as Scratch. Warning and error messages are displayed as well, although from a programmer's point of view debugging options might be more advanced. The heuristic was rated 5 despite the tool being the most complex, which is a noteworthy achievement of App Inventor.

StarLogo informs the user about the status of the system by the means of, e.g., simple and clear messages, information on the user's location within the interface, information on projects, messages in the case of actions potentially leading to data loss, and warning and error messages. Unfortunately, some features were absent, e.g., no function indication of a given element after pointing at it with the mouse cursor.

3.1.2. Keep Consistency

Scratch uses many elements in its interface that clearly suggest their function, e.g., a button marked with a magnifying glass with a "+" or "−" sign. These types of markings are used on many sites and users are already familiar with them. An important aspect of maintaining compatibility between the system and reality is to avoid technical jargon that is incomprehensible to the average user. Scratch uses a very simple and understandable language. Descriptions of specific programming blocks are formulated in such a way that they clearly describe how they function.

App Inventor also uses elements in its interface that suggest their function. Elements from a visual language palette remind users of real-life equivalents, e.g., an element handling camera activities reminds users of a camera, an element handling voice calls

reminds users of a handset, etc. Despite addressing a complex domain of mobile application development, it was possible to keep the language user- and domain-oriented.

StarLogo also uses suggestive elements in its interface, e.g., a button with scissors for the "cut" action or a button with a basket for removing an element. These types of markings are used on many sites and users are already familiar with them. An important aspect is avoiding language that is incomprehensible to the average user. Nevertheless, StarLogo uses a slightly complicated language, especially in describing some programming blocks, which can cause some difficulties, especially for beginners.

3.1.3. Give the User Full Control

Scratch and App Inventor offer the user almost complete control. However, the rating was downgraded due to the fact that some possible actions are not presented to the user in any way. For example, undoing the last action performed can be achieved using a standard keyboard shortcut that is present in many other applications—"Ctrl + Z". Although the action can be performed by a keyboard shortcut, the user does not have this option in the graphical interface. Consequently, only users with a certain degree of familiarity with the environment are really given full control.

StarLogo offers the user almost full control, which is given to both users with a certain degree of familiarity with the environment and those just starting out. For example, users can cut a given element by using a standard keyboard shortcut "Ctrl + X" or by selecting the "cut" option within the graphical user interface.

3.1.4. Stick to the Standards and Be Consistent

Scratch adheres to certain standards. The tools are on the left and the navigation bar is at the top. The app logo that takes the user to the home page is in the top left corner, and the navigation for user features is in the top right corner. The canvas is in the middle, and the layers and assets information is on the right. Such solutions are familiar to users, because this is the most common arrangement for visual programming/modeling environments.

App Inventor offers a similar content/functionality distribution over the user interface. It provides two views when creating an application—Designer and Blocks editor. In both cases, the canvas is in the middle and the navigation patterns are consistent and predictable. In the case of Designer, properties of application elements are accessible on the right. When it comes to the Blocks editor, the list of properties that can be set programmatically is presented on the left, which is an acceptable solution, keeping in mind App Inventor's domain specificity.

StarLogo adheres to certain standards as well. The tools are on the left, and the navigation bar at the top. The app logo that links to the home page is in the top left corner, and the navigation for user features is in the top right corner. The working area is in the middle, which is typical for such solutions.

3.1.5. Prevent Errors

Scratch prevents errors. For example, it is not possible to type a string from a form field intended for a number. However, its rating was downgraded due to a lack of user feedback parts, e.g., flaws in providing hints and suggestions. This can lead to frustration, as users may not be aware that the action they take is wrong, which, in turn, can lead to the erroneous belief that the program does not function correctly.

App Inventor also prevents errors, e.g., by checks of data types, preventing combination of some incompatible elements, and highlighting problematic elements. Although, due to its complicated nature, the user might be better informed which design leads to erroneous applications, which is sometimes clear after its installation on a mobile device. Simple error cases are correctly addressed.

StarLogo prevents errors in a similar way to Scratch. It quite frequently provides error warnings, although not in all cases, which might lead to confusion for users.

3.1.6. Show Instead of Forcing to Remember

Scratch satisfies this heuristic. Navigating the editor is intuitive. Every action is easy to replay. Code blocks are color-coded to indicate their purpose, so the user can easily find the item they need without having to search through all the sections.

App Inventor satisfies this heuristic just like Scratch. What is more, when providing the Designer and Blocks editor views, all useful information from a different view is repeated.

StarLogo does not fully meet this heuristic, so it is rated lower. The names of some blocks require the user to be familiar with the environment. What is more, navigating the editor can cause some difficulties, as it has been divided into three main screens and the user is not fully supported with the relevant information from the other screens.

3.1.7. Ensure Flexibility and Efficiency

Scratch works efficiently, and the user is not forced to wait for individual elements of the interface to load. However, the issue of flexibility poses a problem. Despite a large number of selectable interface languages, several other basic accessibility features are missing: there is no option to enlarge the interface font (workspace window has to be zoomed), a dark mode and a high contrast mode are also missing (the former is expected to be available in the near future).

App Inventor works efficiently in most cases. Although, when dealing with large applications, the area of a model is so large that it needs to be scrolled. When working with big models, the user might wish for smaller elements in order to see as much of the model on a single screen as possible. Some flexibility issues known from Scratch are present as well. App Inventor offers many language versions that address universal languages and languages of smaller but active societies.

StarLogo works efficiently, and the user is not forced to wait for individual elements of the program to load. However, flexibility poses similar problems as in the case of Scratch. Moreover, only an English version is available.

3.1.8. Take Care of Aesthetics and Moderation

The Scratch interface is aesthetically pleasing and minimalistic and keeps all the useful information on the page. Both the color palette and the arrangement of individual elements are planned at a high level. The heuristic is fully met.

App Inventor is aesthetically pleasing and keeps all the useful information on the page. Nevertheless, working with big models and a large number of element types on pallets might be unfavorable, although, due to App Inventor's domain specificity, it seems to be unavoidable.

StarLogo received a noticeably worse rating, as its interface is not minimalistic and aesthetically pleasing. Mismatching high saturation colors were selected. In addition, the page layout is vertical, which also affects its unfavorable appearance.

3.1.9. Ensure Effective Error Handling

This heuristic has been downgraded in the case of all environments due to improper error handling. Although certain actions that might be erroneous are not possible to be performed, the user does not always receive feedback suggesting what error was made and how to fix it. What is more, in the case of App Inventor, the user sometimes only learns after installation that an application design is flawed.

3.1.10. Take Care of Help and Documentation

Scratch has a lot of help and documentation. Users can access a lot of tutorials showing step by step how to deal with the environment. They are enriched with video materials and screenshots showing individual elements of the interface and options that should be selected to achieve the expected effect. In addition, there is a Scratch Wiki page and a forum where users can receive help from others.

App Inventor is also accompanied by a large amount of help, documentation, and tutorials. Audio-visual content is made and distributed by a thriving community gathered around the environment. What is unique is that App Inventor use cases can be found in books and the scientific literature.

StarLogo has a very low rating, as its creators have provided almost no user support. Its documentation and tutorials (that are unavailable) can cause serious problems to some users. Some of the content is outdated, and the links direct to non-existing web pages.

3.2. WCAG Compliance Analysis

The results obtained from the compliance check are presented in Table 2 and discussed taking into account four WCAG areas: perceivable, operable, understandable, and robust. Moreover, the number of distinct problems categories and total number of problems detected in each test case were calculated. The authors understand that the figures presented in Table 2 closely depend on the content of the webpages; nevertheless, they show the common accessibility problems that should be expected for the particular environments.

Table 2. Results of the WCAG analysis—errors and alerts.

	Accessibility Problem	**Visual Programming Environment**														
		Scratch					**App Inventor**					**StarLogo**				
		SC1	**SC2**	**SC3**	**SP**	**Total**	**AC1**	**AC2**	**AC3**	**AP**	**Total**	**LC1**	**LC2**	**LC3**	**LP**	**Total**
1.	(Error) Empty form label	0	0	0	0	0	36	10	8	7	61	0	0	0	0	0
2.	(Error) Empty link	0	0	0	0	0	0	0	0	0	0	1	1	1	1	4
3.	(Error) Empty table header	0	0	0	0	0	0	0	0	0	0	1	1	1	1	4
4.	(Error) Language missing or invalid	0	0	0	0	0	1	1	1	1	4	1	1	1	1	4
5.	(Error) Linked image missing alternative text	1	1	1	1	4	0	0	0	0	0	5	5	5	5	20
6.	(Error) Missing alternative text	51	35	38	34	158	148	244	168	147	707	1	1	1	1	4
7.	(Error) Missing form label	3	4	3	3	13	153	160	153	153	619	174	175	203	156	708
8.	(Error) Spacer image missing alternative text	0	0	0	0	0	93	81	81	81	336	0	0	0	0	0
9.	(Error) Very low contrast	3	3	3	3	12	4	5	4	4	17	199	198	228	175	800
10.	(Alert) Device-dependent event handler	0	0	0	0	0	0	0	0	0	0	2	2	2	2	8
11.	(Alert) Image with title	2	2	2	2	8	0	0	0	0	0	0	0	0	0	0
12.	(Alert) Layout table	0	0	0	0	0	286	318	291	286	1181	1	1	3	0	5
13.	(Alert) Missing fieldset	0	0	0	0	0	0	0	0	0	0	1	1	1	1	4
14.	(Alert) Missing first level heading	0	0	0	0	0	1	1	1	1	4	1	1	1	1	4
15.	(Alert) No heading structure	1	1	1	1	4	0	0	0	0	0	0	0	0	0	0
16.	(Alert) No page regions	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0
17.	(Alert) Noscript element	2	2	2	2	8	1	1	1	1	4	0	0	0	0	0
18.	(Alert) Orphaned form label	1	1	1	1	4	0	0	0	0	0	7	7	7	7	28
19.	(Alert) Possible list	0	0	0	0	0	0	0	0	0	0	1	1	1	1	4
20.	(Alert) Redundant link	0	0	0	0	0	13	13	13	13	52	7	7	1	1	16
21.	(Alert) Redundant title text	0	0	0	0	0	9	8	1	0	18	3	3	4	2	12
22.	(Alert) Select missing label	0	0	0	0	0	4	10	2	2	18	37	36	57	17	147
23.	(Alert) Tabindex	0	0	0	0	0	0	0	0	0	0	1	1	1	1	4
24.	(Alert) Unlabeled form control with title	0	0	0	0	0	2	3	0	0	5	0	0	0	0	0
25.	(Alert) Very small text	37	20	23	19	99	0	10	0	0	10	3	3	15	3	24
Number of distinct problem categories		7	9	9	9	9	13	14	12	11	15	18	18	18	17	18

Each column within the "visual programming environments" group contains the number of problems (errors and warnings) detected during the WCAG assessment. The columns holding the results for test cases, where an application visual code was embedded in the code editor of a visual programming environment, include problems related to the visual code presentation and problems related to the environment interface. Each section devoted to the particular environment includes a "total" column, which is the

sum of the four preceding cells in each row. The rows marked from 1 to 25 are devoted to particular problem categories. The last row of the table is called "number of distinct problem categories", and it holds the number of non-zero cells within the columns holding numbers of problems. Descriptions of the problem categories, and the threats they pose to accessibility, can be found within the WAVE tool documentation [50].

Scratch seems to be the least problematic environment in terms of accessibility. Nine distinct categories of problems were observed: linked image missing alternative text, missing alternative text, missing form label, very low contrast, image with title, no heading structure, noscript element, orphaned form label, and very small text. The biggest number of problem occurrences was observed in the categories missing alternative text and very small text, that belong to the WCAG perceivable area. Occurrences in the remaining categories were incidental.

In the case of App Inventor, problems were detected in 15 different categories. The biggest number of problem occurrences (significantly exceeding the results for Scratch) was registered for the WCAG perceivable area within the categories: missing alternative text, missing form label, spacer image missing alternative text, layout table, and empty form label. What is new is that the assessment revealed problems in the WCAG operable area (especially the redundant link category) and the understandable area (language missing or invalid, and missing labels). Occurrences in the remaining categories were incidental. Despite these results, it has to be concluded that App Inventor is a quite successful environment, with millions of users that have created millions of applications. The numbers presented may suggest that it is implemented using a style or technology that is more prone to accessibility errors than in the case of Scratch.

In the case of StarLogo, problems were detected in 18 distinct categories, which is the highest score among the tested environments. The biggest number of problem occurrences was noted in the WCAG perceivable area within the categories missing form label, very low contrast, and select missing label. Occurrences in the remaining categories were incidental, although categories of problems from the WCAG operable area are broadly affected.

In general, the main problems of all environments are missing or improper labels/alternative texts, a distorted page structure, redundant links, and problems with a low contrast of graphical interface elements. What are the expected outcomes? If an alternative text/label, e.g., in the case of images, buttons, and links is missing, screen readers for blind users are not able to describe them properly, the purpose of an interface element. Missing headings and headings with no content make navigation problematic for the user, e.g., skipping empty headings when using a keyboard to navigate might be tiresome for the user. Low contrast and very small text deteriorate interface accessibility for visually impaired users. Contrast problems basically only concerned StarLogo and visual symbols within the palette and work area. Redundant links mean that many links on the same webpage direct to the same resource or interface section. This can cause redundant navigation when using screen readers or keyboard navigation. An interesting case is the presence of noscript content, which is available only if JavaScript is disabled. Because the vast majority of users, including those using assistive technologies, are script-enabled, noscript elements cannot be used to deliver content. Finally, a tabular layout detected in the case of App Inventor might introduce order and navigation issues.

4. Conclusions

The trend of using visual programming environments is gaining importance these days. The affected areas of human life are engineering activities, that nowadays very frequently require advanced programming skills. Low-code technologies can bridge the gap between these requirements and the actual knowledge and programming skills distribution.

The authors investigated the cases of Scratch and StarLogo as successful solutions for STEM learning and developing computational thinking competences for future engineers. App Inventor was additionally recognized as a tool able to produce mobile applications answering the actual needs of their creators.

The authors identified the need for accessibility and usability assessments of visual programming platforms like the chosen ones. For this purpose, Nielsen's heuristics were applied, followed by the WCAG compliance analysis. This set of methods proved to be successful. The authors were able to complete their assessment process. What is more, these methods turned out to be inexpensive. In the case of Nielsen's heuristics, it was proven by Nielsen himself that three evaluators being double experts can detect even up to 90% of problems. He also stated that cost effectiveness decreases if there are more than five evaluators. In the case of the WCAG analysis, free automated tools are available. These prove the usefulness of the assessment tools proposed by the authors.

It was found that the visual programming environments, Scratch and App Inventor, are well-made solutions in terms of look and feel, functionality, documentation, interface navigation, and memorization. The identified WCAG problems mainly concerned low contrast, missing labels and alternative text, and link redundancy. The StarLogo interface, on the other hand, was less user-friendly in terms of documentation, aesthetics, and functionality. These aspects answer the question about how platforms fulfil the accessibility guidelines that are widely accepted by the computer science society.

During the usability assessment it was revealed that the evaluated environments had noticeable problems with ensuring flexibility and efficiency and ensuring effective error handling. This can be the result of some kind of tradeoff between simplification of the visual syntax and the editor, while still providing facilities mandatory for supporting the application development process. The most urgent aspect to be improved is debugging the visual code of an application, in order to ease and speed up user work. Finally, in the case of StarLogo, documentation should be better maintained. It would be also useful to take care of the community created around this environment, as the community is usually a valuable source of help for other users and a source of fresh ideas.

In term of accessibility, it is an obvious recommendation to repair common issues that were identified for the evaluated environments. The WCAG area with the biggest number of problems was each time related to providing information and interface components in ways users can perceive. The second problematic area, although to a smaller degree (a lower number of issues), was related to providing interface components and navigation that is operable for users. The authors think that addressing these areas should be of high priority for the environments' providers, in order to make their solutions more inclusive. In parallel, many assistive technologies (e.g., reading descriptions, acting on voice commands, etc.) are ready and waiting to be implemented within such environments, especially technologies aiding people with sight and movement problems, which might be an interesting topic for future work.

Author Contributions: Methodology—K.Ż.; funding—K.Ż.; writing final text—K.Ż.; data acquisition—K.Ż., K.C. and D.C.; comments on heuristics and WCAG analysis—K.Ż., K.C. and D.C.; experiment execution—K.C. and D.C.; text review—K.C. and D.C.; writing draft—K.C. and D.C.; visualization—K.C. and D.C. All authors have read and agreed to the published version of the manuscript.

Funding: This article was financed by the Lublin University of Technology.

Data Availability Statement: The original contributions presented in the study are included in the article, further inquiries can be directed to the corresponding author.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. LabView. LabView Homepage. Available online: https://www.ni.com/pl-pl/shop/labview.html (accessed on 1 July 2024).
2. App Inventor. MIT App Inventor homepage. Available online: https://appinventor.mit.edu/ (accessed on 1 July 2024).
3. Kordaki, M. Diverse Categories of Programming Learning Activities Could Be Performed Within Scratch. *Procedia Soc. Behav. Sci.* **2012**, *46*, 1162–1166. [CrossRef]
4. Harvey, B.; Mönig, J. Bringing "No Ceiling" to Scratch: Can One Language Serve Kids and Computer Scientists? In Proceedings of the Constructionism 2010, Paris, France, 16–20 August 2010.

5. Park, Y.; Shin, Y. Comparing the Effectiveness of Scratch and App Inventor With Regard to Learning Computational Thinking Concepts. *Electronics* **2019**, *8*, 1269. [CrossRef]
6. Fidai, A.; Capraro, M.M.; Capraro, R.M. "Scratch"-ing Computational Thinking With Arduino: A Meta-analysis. *Think. Ski. Creat.* **2020**, *38*, 100726. [CrossRef]
7. Plaza, P.; Sancristobal, E.; Carro, G.; Blazquez-Merino, M.; Garcia-Loro, F.; Munoz, M.; Albert, M.J.; Morinigo, B.; Castro, M. Scratch as Driver to Foster Interests for STEM and Educational Robotics. *IEEE Rev. Iberoam. Tecnol. Aprendiz.* **2019**, *14*, 117–126. [CrossRef]
8. Alturayeif, N.; Alturaief, N.; Alhathloul, Z. DeepScratch: Scratch Programming Language Extension for Deep Learning Education. *Int. J. Adv. Comput. Sci. Appl.* **2020**, *11*. [CrossRef]
9. Cottone, A.M.; Yoon, S.A.; Coulter, B.; Shim, J.; Carman, S. Building System Capacity With a Modeling-Based Inquiry Program for Elementary Students: A Case Study. *Systems* **2021**, *9*, 9. [CrossRef]
10. Hsiao, L.; Lee, I.; Klopfer, E. Making Sense of Models: How Teachers Use Agent-based Modeling to Advance Mechanistic Reasoning. *Br. J. Educ. Technol.* **2019**, *50*, 2203–2216. [CrossRef]
11. Zavery, A. How No-Code and the Democratization of Application Development Are Helping Businesses. Forbes. 20 July 2021. Available online: https://www.forbes.com/sites/googlecloud/2020/06/25/how-no-code-and-the-democratization-of-application-development-are-helping-businesses/ (accessed on 1 July 2024).
12. Jivani, M.N. GSM Based Home Automation System Using App-Inventor for Android Mobile Phone. *Int. J. Adv. Res. Electr. Electron. Instrum. Eng.* **2014**, *3*, 12121–12128. [CrossRef]
13. Adiono, T.; Anindya, S.F.; Fuada, S.; Afifah, K.; Purwanda, I.G. Efficient Android Software Development Using MIT App Inventor 2 for Bluetooth-Based Smart Home. *Wirel. Pers. Commun.* **2019**, *105*, 233–256. [CrossRef]
14. Mikolajczyk, T.; Fuwen, H.; Moldovan, L.; Bustillo, A.; Matuszewski, M.; Nowicki, K. Selection of Machining Parameters with Android Application Made Using MIT App Inventor Bookmarks. *Procedia Manuf.* **2018**, *22*, 172–179. [CrossRef]
15. Munasinghe, T.; Patton, E.W.; Seneviratne, O. IoT Application Development Using MIT App Inventor to Collect and Analyze Sensor Data. In Proceedings of the 2019 IEEE International Conference on Big Data (Big Data), Los Angeles, CA, USA, 9–12 December 2019; pp. 6157–6159. [CrossRef]
16. Quanjin, M.; Rejab, M.R.M.; Idris, M.S.; Zhang, B.; Merzuki, M.N.M.; Kumar, N.M. Wireless Technology Applied in 3-axis Filament Winding Machine Control System Using MIT App Inventor. *IOP Conf. Ser. Mater. Sci. Eng.* **2019**, *469*, 012030. [CrossRef]
17. Pérez-Jorge, D.; Martínez-Murciano, M.C. Gamification with Scratch or App Inventor in Higher Education: A Systematic Review. *Future Internet* **2022**, *14*, 374. [CrossRef]
18. Boldbaatar, N.; Şendurur, E. Developing Educational 3D Games With StarLogo: The Role of Backwards Fading in the Transfer of Programming Experience. *J. Educ. Comput. Res.* **2018**, *57*, 1468–1494. [CrossRef]
19. Papadakis, S.; Kalogiannakis, M.; Zaranis, N.; Orfanakis, V. Using Scratch and App Inventor for teaching introductory programming in secondary education. A case study. *Int. J. Technol. Enhanc. Learn.* **2017**, *8*, 217–233. [CrossRef]
20. Papadakis, S.; Kalogiannakis, M.; Orfanakis, V.; Zaranis, N. Novice Programming Environments. Scratch & App Inventor: A first comparison. In Proceedings of the 2014 Workshop on Interaction Design in Educational Environments (IDEE '14), Albacete, Spain, 9 June 2014; pp. 1–7. [CrossRef]
21. Chang, C.K. Using Computational Thinking Patterns to Scaffold Program Design in Introductory Programming Course. In Proceedings of the 5th IIAI International Congress on Advanced Applied Informatics (IIAI-AAI), Kumamoto, Japan, 10–14 July 2016; pp. 397–400. [CrossRef]
22. JetLearn. Scratch Statistics: Examining the Popularity of Scratch 2024. Available online: https://www.jetlearn.com/blog/scratch-statistics (accessed on 1 October 2024).
23. MIT. App Inventor Stats. Available online: http://ai2.appinventor.mit.edu/stats (accessed on 1 October 2024).
24. Fakrudeen, M.; Yousef, S.; Hussein, A.H. Analyzing App Inventor for building usable touch screen courseware for blind users. In Proceedings of the International Conference on eBusiness, eCommerce, eManagement, eLearning and eGovernance 2014 (IC5E 2014), London, UK, 30–31 July 2014; pp. 9–14.
25. Solecki, I.; Porto, J.; Da Cruz Alves, N.; Von Wangenheim, C.G.; Hauck, J.; Borgatto, A.F. Automated Assessment of the Visual Design of Android Apps Developed with App Inventor. In Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20), Portland, OR, USA, 11–14 March 2020; pp. 51–57. [CrossRef]
26. Colter, A.J. Evaluating and Improving the Usability of MIT App Inventor. 2016. Available online: https://dspace.mit.edu/handle/1721.1/106027 (accessed on 1 July 2024).
27. Suzuki, A. Improving the Ability of StarLogo Nova Users to Save and Organize Their Work. Available online: https://dspace.mit.edu/handle/1721.1/113100 (accessed on 1 July 2024).
28. Kölling, M.; McKay, F. Heuristic Evaluation for Novice Programming Systems. *ACM Trans. Comput. Educ.* **2016**, *16*, 1–30. [CrossRef]
29. Kowalczyk, R.; Turczyński, Ł.; Żyła, K. Comparison of App Inventor 2 and Java in creating personal applications for Android on example of a notepad. *Adv. Sci. Technol./Postępy Nauk. I Tech.* **2016**, *10*, 247–254. [CrossRef]
30. Żyła, K. Simplified graphical domain-specific languages for the mobile domain–perspectives of learnability by nontechnical users. *Appl. Comput. Sci.* **2017**, *13*, 32–40. [CrossRef]
31. Scratch. MIT Scratch Homepage. Available online: https://scratch.mit.edu/ (accessed on 1 July 2024).

32. Sarasa-Cabezuelo, A. Use of Scratch for the Teaching of Second Languages. *Int. J. Emerg. Technol. Learn.* **2019**, *14*, 80. [CrossRef]
33. Rholl, D. Complex Systems in a Complex Time: Two Flexible Activities Using Online Models to Study Infectious Disease Transmission and Code Complex Systems Simulations. *J. Microbiol. Biol. Educ.* **2021**, *22*. [CrossRef]
34. StarLogo. MIT StarLogo Homepage. Available online: https://education.mit.edu/project/starlogo-tng/ (accessed on 1 July 2024).
35. Klopfer, E. The Complex Evolution of Technologies and Pedagogies for Learning About Complex Systems. *Systems* **2021**, *9*, 31. [CrossRef]
36. Gonzalez-Holland, E.; Whitmer, D.; Moralez, L.; Mouloua, M. Examination of the Use of Nielsen's 10 Usability Heuristics &Amp; Outlooks for the Future. *Proc. Hum. Factors Ergon. Soc. Annu. Meet.* **2017**, *61*, 1472–1475. [CrossRef]
37. Hermawati, S.; Lawson, G. Establishing Usability Heuristics for Heuristics Evaluation in a Specific Domain: Is There a Consensus? *Appl. Ergon.* **2016**, *56*, 34–51. [CrossRef] [PubMed]
38. Nielsen, J. Enhancing the explanatory power of usability heuristics. In Proceedings of the 1994 SIGCHI Conference On Human Factors in Computing Systems, Boston, MA, USA, 24–28 April 1994; pp. 152–158. [CrossRef]
39. NNG. Nielsen Norman Group Homepage—World Leaders in Research-Based User Experience. Available online: https://www.nngroup.com/people/jakob-nielsen/ (accessed on 1 July 2024).
40. Doush, I.A.; Sultan, K.; Al-Betar, M.A.; Almeraj, Z.; Alyasseri, Z.A.A.; Awadallah, M.A. Web Accessibility Automatic Evaluation Tools: To What Extent Can They Be Automated? *CCF Trans. Pervasive Comput. Interact.* **2023**, *5*, 288–320. [CrossRef]
41. Ballantyne, M.; Jha, A.; Jacobsen, A.; Hawker, J.S.; El-Glaly, Y.N. Study of Accessibility Guidelines of Mobile Applications. In Proceedings of the 17th International Conference on Mobile and Ubiquitous Multimedia, Cairo, Egypt, 25–28 November 2018. [CrossRef]
42. W3C. WCAG 2.1 Homepage. Available online: https://www.w3.org/TR/WCAG21/ (accessed on 1 July 2024).
43. Sarita, K.; Kaur, P.; Kaur, S. Accessibility of Healthcare Sites: Evaluation by Automated Tools. In Proceedings of the International Conference on Data Science and Applications, Kolkata, India, 26–27 March 2022; Lecture Notes in Networks and Systems. Springer: Singapore, 2021; pp. 625–636. [CrossRef]
44. Vigo, M. Benchmarking Web Accessibility Evaluation Tools: Measuring the Harm of Sole Reliance on Automated Tests. In Proceedings of the 10th International Cross-Disciplinary Conference on Web Accessibility, Rio de Janeiro, Brazil, 13–15 May 2013. [CrossRef]
45. Csontos, B.; Heckl, I. Improving Accessibility of CMS-based Websites Using Automated Methods. *Univers. Access Inf. Soc.* **2020**, *21*, 491–505. [CrossRef]
46. Ikhsan, I.N.; Candra, M.Z.C. Automatically: An Automated Refactoring Method and Tool for Improving Web Accessibility. In Proceedings of the 2018 5th International Conference on Data and Software Engineering (ICoDSE), Mataram, Indonesia, 7–8 November 2018; pp. 1–6. [CrossRef]
47. Nielsen, J. Finding usability problems through heuristic evaluation. In Proceedings of the 1992 SIGCHI Conference On Human Factors in Computing Systems (CHI '92), Monterey, CA, USA, 3–7 May 1992; pp. 373–380. [CrossRef]
48. NN Group. How to Conduct a Heuristic Evaluation. Available online: https://www.nngroup.com/articles/how-to-conduct-a-heuristic-evaluation/ (accessed on 1 July 2024).
49. Invesp. Heuristic Evaluation: Your Complete Guide to Heuristic Evaluation for Conversion Optimization. Available online: https://www.invespcro.com/blog/heuristic-evaluation-your-complete-guide/ (accessed on 1 July 2024).
50. Utah State University. WAVE Tool Homepage. Available online: https://wave.webaim.org/ (accessed on 1 July 2024).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.