Comparing novice programming environments for use in secondary education: App Inventor for Android vs. Alice

Stamatios Papadakis*

Faculty of Education, 
Department of Preschool Education, 
University of Crete, 
Crete, Greece 
Email: stpapadakis@gmail.com 
*Corresponding author

Vasileios Orfanakis

1st Vocational School of Agios Nikolaos, 
Agios Nikolaos, Crete, Greece 
Email: vorfan@gmail.com

Abstract: Coding is part of logical thinking and is one of the basic skills which are known as ‘21st-century skills’. Coding acquisition is necessary as it is used in a wide range of occupations. However, computer programming is difficult to learn and programming courses often have high drop-out rates. Novice programmers suffer from a wide range of difficulties and deficits. Research in teaching and learning programming across different countries and educational contexts reveal that novice programmers face the same challenges in their efficiency of writing, debugging and running programs. These difficulties have led those involved in the teaching of programming to further consider the most effective ways that can facilitate novice programmers in learning the basic programming concepts. Visual programming environments which support the construction of programs through a drag-and-drop interface are among the most popular coding tools for teaching novice programmers. In this paper, we investigate the use of Alice and App Inventor for Android, with regard to their effectiveness for teaching and learning programming in secondary education students.

Keywords: Alice; App Inventor for Android; novice programmers; visual programming environments.


Biographical notes: Stamatis Papadakis is a graduate of the Economics and Business (AUEB) University, Athens, Greece, Department of Information. He received his MSc in Education from the University of the Aegean, Greece, and PhD from the University of Crete, School of Education. He has been working for a series of years as an ICT teacher in public sector secondary education. He has published many articles in journals and has presented several papers in conferences. His research interests include ICT in education, mobile
Comparing novice programming environments for use in secondary education

1 Introduction

Twenty-first century education needs to prepare young people for jobs that don’t exist yet, using technologies that have not even been invented, for which competition will be global (Hampson et al., 2012; Papadakis, 2016). Programming in today’s information society, besides being a learning object, has become a core competence addressed to current students and potential future employers (Papadakis et al., 2016). The Information Technology (IT) sector is characterised by tremendous professional development opportunities (European Schoolnet, 2014). According to the skills framework for the information age foundation, there are 86 different professional skills related to IT which correspond to 290 different specialisations in the IT sector. In Europe, in 2013, more than 4.5 million people worked in IT, in a large number of different professional activities (Papadimitriou, 2014). According to the European Commission, there will be 756,000 unfilled vacancies for ICT professionals by 2020 (European Commission, 2016).

Accordingly, one would expect to see rising enrolments of students in introductory programming courses in the context of formal education. However, inspecting the data suggests that this tendency is not observed. Although it certainly appears to be an increment in the number of IT jobs, the number of student entrants in the IT industry steadily decreased since its peak in the late 1990s (Cooper et al., 2010). One reason is that students consider programming as a boring and tedious process; thus, this attitude finally shapes their future career choices (Cooper, 2010).

Researchers state that although novice programmers are often interested in programming due to their previous experience with video games and multimedia applications (Orfanakis and Papadakis, 2016), they get easily discouraged because they find it extremely difficult and time-consuming to create their own applications (Cooper et al., 2003). By contrast, other studies show that school-aged students show increased interest and ability in programming only if they deal with it in an attractive way (Hylton and Otoupal, 2005; Papadakis et al., 2016; Papadakis and Orfanakis, 2017).

For these reasons, the subject of teaching programming is constantly at the heart of the research community (Guzdial, 2004; Kelleher and Pausch, 2005; Xinogalos et al., 2015). In an attempt to increase students’ interest and participation in computer science, much effort has been made during the last decades in the development of tools and activities to facilitate learning programming for novices in schools and universities (Hsu and Ching, 2013; Papadakis and Orfanakis, 2017; Storey, 2005). These efforts include the
implementation of kinaesthetic learning activities such as computer science unplugged, the unplugged hour of code and the development of programming environments suitable for beginners. Those programming environments, namely novice programming environments (NPEs) like Scratch and Alice and recently App Inventor for Android (AIA), have successfully lowered the barrier of novice programmers’ first programming experience (Meerbaum-Salant et al., 2013; Roy, 2012). NPEs facilitate software development within a fun and interactive framework which does not deter novice programmers from learning to program (Kelleher and Pausch, 2005).

The rest of the article is organised as follows. In Section 2, we discuss the difficulties that novice programmers face as a result of the traditional teaching approach in programming. Sections 3–5 discuss the use of NPEs in introductory programming courses and we present two famous NPEs, the Alice and the AIA. Sections 6 and 7 compare Alice and AIA highlighting their strengths and weaknesses from a pedagogical and technological point of view. This is followed by a section of discussion about the use of NPEs in introductory programming courses and ends with concluding remarks.

2 The difficulties of novice programmers and the traditional teaching approach

Learning to program is a difficult task (Efopoulos et al., 2005; Ivanović et al., 2016; Lahtinen et al., 2005; Robins et al., 2003) as it is an inherently complex mental activity (Spohrer and Soloway, 1986). At all levels of secondary education, the teaching and learning of programming are accompanied by the classical phenomena which characterise the ‘difficult’ courses: students do not even understand the basic programming concepts and have difficulty in solving problems (Gomes and Mendes, 2007).

Novice programmers ‘suffer’ from a wide range of difficulties and cognitive deficits (Mohammed et al., 2015; Nandigam and Bathula, 2013; Robins et al., 2003). Burkhardt et al. (1997) state that programming is difficult, as there is no direct handling and the concepts are abstract. Similarly, Pennington et al. (1995) point out the difficulties that arise when one tries to maintain simultaneously different levels of information while they state that the major problem for novice programmers is to sufficiently coordinate the different problems.

The difficulties faced by novice programmers in learning to program are well documented and are identified in the use of variables, logical operators and control structures, the loops, passing the values and running multiple threads simultaneously (Meerbaum-Salant et al., 2013). Both the work of Robins et al. (2003) and the classic book entitled “Studying the Novice Programmer” (Soloway and Spohrer, 1989) outline several of students’ misconceptions about the concept of programming in detail which should be considered in the design of introductory programming courses. For example, they comment that novice programmers usually have a superficial knowledge of programs. Often their software development approaches are line by line instead of using programming constructs. Novice programmers often fail to correctly apply the knowledge they have obtained (Ala-Mutka, 2003). Students may know the syntax and semantics of individual statements, but they do not know how to combine those elements in order to produce valid programs (Costa et al., 2012). Students commonly have misconceptions concerning the creation of a program, such as variable initialisation, loop conditions, indicators, data structures and recursion (Robins et al., 2003; Soloway and Spohrer, 1989; Winslow,
Comparing novice programing environments for use in secondary education

They face similar problems with object-oriented programing paradigms (Guzdial, 2004). Soloway et al. (1983) state that only 38% of novice programmers can write a program to calculate the average of numbers given by a user, an activity that most teachers consider compatible with the cognitive level of secondary education students who have attended introductory programing courses.

The difficulties or challenges for the novice programmer extend beyond the five areas noted by du Boulay (general orientation, the notional machine, notation, structures and pragmatics). A literature review also shows that these problems may result from widespread use of the traditional or classical didactic approach in introductory programing courses (Xinogalos, 2002). As Xinogalos and Satratzemi (2004a, 2004b) state the traditional or classical teaching method means an approach to teaching programing which consists of the following:

a. The use of a general-purpose programing language (Java, Visual Basic, C, C++, etc.).

b. A professional, integrated development environment (IDE) for this language.

c. Solving a set of calculation and number processing problems, and/or the appearance of some messages.

In the traditional didactic approach, more emphasis is placed on learning the programing language itself, which requires a strict discipline in its syntax and semantics rather than the development of a methodology and the capacity to solve problems (Kibby and Hartley, 2014). Novice programmers’ education is typically conducted using professional software development environments or others which are similar and freely available. These environments, despite extensive documentation, are addressed to the professional programmer and are not necessarily the best topic to address in the students’ first programing course (Robins et al., 2003), let alone for teaching programing in secondary education.

Xinogalos and Satratzemi (2004b) point out that the inadequacy of these environments is due to a number of reasons, the main of which are as follows:

a. General-purpose programing languages typically have a large command repertoire and are complex.

b. The students’ attention is focused on language syntax learning instead on developing the ability to apply problem-solving skills.

c. Students generally do not have sufficient support in order to understand the basic operations and control structures, as the programing environment does not usually provide output visualisation features.

d. Commercial compilers do not meet the needs of novice programmers.

e. The intellectual complexity that requires the representation of an algorithm in a programing language is great due to the ‘nature’ of the language.

f. The successful solution of even interesting problems requires that novice programmers must cover a large subset of the language as well as the development of sizeable programs. This finally draws learners’ attention to the formal properties of the language rather than the development of problem-solving skills.
Additionally, Efopoulos et al. (2005) point out that although in introductory programming courses a small portion of these languages is taught to beginners, ignoring the obscure and powerful features of the language, there are several problems arising such as:

a The general-purpose programming language support material (manuals, books, on-line help) is not adapted to novice programmers’ needs.

b The language compiler is not limited to a subset of the language but produces messages that refer to the whole of the language.

In conclusion, novice programmers having a first contact with a professional software development environment face a relatively complex interface and a variety of features, functions and environment-integrated tools which are not initially needed. As a result, those generic, all-purpose languages and the traditional teaching method in general put them under too much cognitive load and make them feel confused and discouraged (Kinnunen and Simon, 2010; Lee et al., 2011).

3 Novice programming environments

Forte and Guzdial (2004) state that the ‘traditional’ teaching approach in programming is more likely to prevent than to attract students. In particular, the lack of motivation and lack of enthusiasm are two of the main reasons for high failure in introductory programming courses (Mendes et al., 2012; Siegle, 2009; Yadin, 2011). For decades, college and high school students tend to drop-out introductory programming courses at very high rates (Kranck, 2012; Winslow, 1996; Xinogalos et al., 2015). It appears that a significant number of students consider programming as a mysterious and complex process, which requires specialised technical training and education (Bryant et al., 2008; Ford, 2008). Students describe the programming courses as too technical, cut off from the real world and devoid of creativity (Khuloud and Gestwicki, 2013). Researchers consider that the main cause of high drop-out and failure rates in introductory programming courses is that students do not perceive those courses as useful and interesting for them (Bennedsen and Caspersen, 2007; Forte and Guzdial, 2004; Sarpong et al., 2013).

In particular, on incentives, if the creation of a ‘Hello, World!’ program intrigued students in previous decades, this is not the case with the current generation of students who are attracted by video game consoles, smart mobile devices and modern interactive multimedia environments (Guzdial and Soloway, 2002; Papadakis et al., 2014). According to Freudenthal et al. (2010), the teaching of algorithmisation and programming should be made so as to minimise the cognitive load while maximising the pedagogical value. Student involvement is often the catalyst for educational improvements and is often successful when the ‘classroom context’ is consistent with student interests and goals, and encourage students to assess what they learn (Gray et al., 2012; Papadakis et al., 2014; Papadakis and Orfanakis, 2017).

Margulieux et al. (2012) point out that the difficulties facing students in learning computer programming skills can be successfully addressed by transforming an introductory programming course from a simple lecture into an easy and fun experience, and there are several strategies for achieving those goals. One way to achieve this is through a gradual reduction of the intrinsic cognitive load that is required from novice programmers to learn programming and with a corresponding reduction in the amount of
the information which is used for problem solution (Robins et al., 2003). In order to effectively reduce the amount of information, multiple programing components can be isolated so that students do not try to learn multiple subjects simultaneously. Students could first learn to solve a problem on a theoretical level, creating mental models, which should focus on building solutions, without being particularly involved with the syntax of obscure or complex commands - statements themselves (Resnick et al., 2009). Papert (1980) argued that programing languages should have a ‘low floor’ (easy to get started) and a ‘high ceiling’ (opportunities to create increasingly more complex projects over time). In addition, languages need ‘wide walls’ (supporting many different types of projects so people with many different interests and learning styles can all become engaged) (Greher and Heines, 2014; Guzdial, 2004; Harvey and Monig, 2010).

To address these challenges, the research community has over time suggested several alternative solutions and approaches. For this purpose, it is still looking for teaching tools, including programing languages, supplementary library collections, software development environments, software templates, textbooks and other means (Kelleher and Pausch, 2005; Orfanakis and Papadakis, 2014; Vidžiūnas and Vitkutė-Adžgauskienė, 2012). Some approaches include computational models that simulate and provide immediate and relevant feedback, which enables novice programmers to learn from their successes and mistakes (Repenning, 2011). Other approaches include the use of micro-worlds and micro-languages, the improvement of compilers’ diagnostic capabilities, the use of understandable error messages, the use of virtual programing languages, etc. (Spyropoulos et al., 2005). Further suggested are the use of interactive software that enables data visualisation, teaching the algorithm in a computer-based collaborative learning environment and the use of diagrammatic representation of repeated structures (Laakso et al., 2009).

One of the most successful strategies for teaching novice programmers is through the development and use of suitable programing environments for novices (Krul, 2012; Roy et al., 2012; Utting et al., 2010). Popular NPEs, such as Scratch and Alice, have successfully lowered the barrier of initial entry to programing (Roy et al., 2012), offering opportunities to students to practice their creative imagination and skills in activities that are relevant to their interests as well as authentic and valuable (Federici, 2011). NPEs are visual programing languages which use a ‘drag-and-drop’ code creation system, replacing the text-based code with sliding components (codeblocks), an approach that reduces the cognitive load associated with strict command syntax (Mason et al., 2015), allowing users to focus on the development of problem-solving skills (Ma et al., 2011). These block-based visual programing tools have increased student interest in programing by successfully lowering the barriers and challenges faced by novice programmers (Weintrop and Wilensky, 2015). Also, these programing environments are considered easy by users of all ages, cognitive backgrounds and interests, allowing novice programmers to practice programing just like putting LEGO bricks together (Resnick et al., 2009). NPEs facilitate software development in a context that is fun and not ‘threatening’. This way of learning programing removes the thorny syntax issues that traditionally plagued much of novice programing in a playful way (Nijholt, 2015; Roy et al., 2012).

During the last two decades, a significant number of varied programing learning environments have been created. These include open micro-worlds environments, like Scratch, narrative-centred environments, such as Alice, specialised outputs such as Logo and Lego Mindstorms NXT, three-dimensional (3D) virtual worlds such as MS Kodu and
various other environments, such as GameMaker, GameFoot, DesignBlocks, Greenfoot, etc. (Brennan and Resnick, 2012; Giannakos et al., 2013; Maloney et al., 2010). NPEs, such as Scratch, Alice and Lego Mindstorms NXT, have seen great acceptance and publicity in recent years, as research has shown that those visual programming environments play an important role in attracting and retaining novice programmers (Federici, 2011). The hope is that students will no longer suffer from low self-esteem and experience fear and anxiety when confronted with code, making them more receptive to further deepening their knowledge in programming (Olabe et al., 2011). However, we must not forget that the use of an NPE does not solve the problem of command syntax, but postpones it until the student understands the basics of program development (Wilson and Moffat, 2010). The learner will have to deal with code syntax later, during secondary, professional language learning, while trying more advanced courses. Figure 1 shows the difference in the syntax in one of the simplest programs (Hello World!) in a traditional programming language (Java) and 2 NPEs (Scratch and AIA).

**Figure 1** Differences between coding in Java, Scratch and AIA (see online version for colours)

---

In recent years, smart mobile devices such as smartphones and tablets have flooded the market and increasingly penetrated the world of young people (Ng, 2015). Young people are fanatical users, both of these devices and their applications (Papadakis et al., 2017; Papadakis and Kalogiannakis, 2017; Wagner et al., 2013; Zaranis et al., 2013), while many of them express their interest in developing their own mobile applications (Roy, 2012). In this case, if NPEs’ wish to remain at the heart of youth and student interest, they should be adapted targeting a mobile-centric audience. The last entry in the field of NPEs is AIA, which by taking advantage of the rising popularity of mobile devices and student preference for mobile apps has made a dynamic penetration in a variety of school environments (Papadakis et al., 2014; Roy et al., 2012).
Comparing novice programming environments for use in secondary education

4 The Alice programming environment

During the past years, efforts have been made to make programming easier for novices. These efforts culminated in Alice (www.alice.org), a programming environment that makes developing interactive applications simple. Alice is a programming environment created by Carnegie Mellon University in the United States that addresses many of the issues associated with the difficulty of learning to program with traditional programming languages (Ali and Smith, 2014; Robins et al., 2003). Alice was first released on January 2009 with Alice 2.2 version and then returned with 2.3.5 and 2.4.2 on 2013 and 2014, respectively. In addition, Storytelling Alice, another version of Alice, was created. The difference between the two versions is that Alice 2.x was designed for high school and college students. Storytelling Alice is designed for middle-school students. Storytelling Alice was created by Caitlin Kelleher as part of her doctoral work in Computer Science at Carnegie Mellon University. Finally, version 3.x was first released in August 2014 and the final version 3.3.0.0 was released in August 2016 (Alice.org, 2016). Pedagogically, Alice owes its existence to the work of Papert with the Logo language and the other educational programming language for beginners Karel the Robot. It should be noted that Alice is not an acronym. ALICE is not A.L.I.C.E. and does not mean anything. The development team called the environment ‘Alice’ in honour of Lewis Carroll, who wrote the novel Alice’s Adventures in Wonderland (Daly and Wrigley, 2015).

The growing popularity of Alice as an introductory programming environment is due to the many advantages offered over traditional or general programming languages (Al-Jarrah and Pontelli, 2014). Alice provides an environment where students can create virtual worlds on their computers and populate them with some really interesting 3D objects in a creative sense, and use/modify these objects and write programs to generate animation (Al-Linjawi and Al-Nuaim, 2010). Ali and Smith (2014) point out the advantages of using Alice to introductory programming courses such as the absence of strict syntax, the existence of an IDE which allows the usage of familiar examples, as well as the presence of a game-like environment which allows for more interaction during program development. In Alice, students do not need to write any code because they simply ‘drag-and-drop’ command blocks into the editor. Additionally, Alice’s graphical interface makes it easier for novice programmers to monitor the execution of a program while at the same time significantly reduces code complexity (Herbert, 2010).

Therefore, Alice is widely used in introductory level programming courses both at the upper end of secondary and the first years of tertiary education. In the Alice environment, novice programmers are able to watch their programs executed in animated form, as a turtle and robot, respectively, so as to directly observe how each command alters the flow of execution of the program (Cooper, 2010). However, Alice is differentiated from similar environments since it includes a 3D graphics engine and a large library of 3D models (Parsons and Haden, 2007). In Alice, students can create their own virtual worlds and animations in the same way objects move in a flight simulator or a video game. Many of the same techniques are used in Alice to give the illusion of motion as are used by animators to create animated cartoons for film studios such as Disney and Pixar (Taffe, 2004).
The main purpose of Alice’s creation was the introduction of novice programmers to object-oriented (OO) programming and objects-first strategy (Cooper et al., 2003). Alice supports the pedagogical goals of teaching OO concepts, such as a fundamental introduction to objects, methods, decision statements, loops, recursion and inheritance (Al-Linjawi and Al-Nuaim, 2010). The initial target audience for Alice was students who were computer science majors, so it was necessary to consider transitioning students to Java, or some other object-oriented language (Cooper, 2010). Alice provides novice programmers with an environment, which encourages the students to think in terms of objects, and represent their relationships graphically using 3D class models (Al-Linjawi and Al-Nuaim, 2010). The objects would come from the everyday world such as trees, animals, etc. The students-novice programmers learn the basic programming concepts while creating a virtual world through a mental situation that Randy Pausch calls a ‘headfake’ approach to learning. ‘Headfake’ is defined as a situation where students encouraged to learn something believe that they are learning something different (Dann and Cooper, 2009). In Alice, this can be achieved in one of two ways: first Alice includes structured programming constructs like loops and conditionals and, second, provides a ‘Java-like’ command syntax, without students having to deal with wayward syntax such as the use of semicolons. Alice not only impresses students with its rich multimedia environment in which novice programmers feel comfortable with allocating and handling objects, but it emphasises object-oriented programming methodology and concepts exceptionally useful for students wading into object-oriented programming (Daly, 2011). Alice’s third edition provides better support for this transition. It is possible for a project in Alice to be used as a library in Java, using NetBeans IDE (Dann et al., 2010).

4.1 Advantages of using Alice in introductory programming courses

Alice offers computer science instructors an approach to introducing fundamental concepts to novice programmers that allows them to quickly identify and learn from mistakes (Al-Linjawi and Al-Nuaim, 2010). The main advantage of using Alice in introductory programming courses is that it successfully treats the four problems that make learning a programming language a difficult task for novice programmers (Ali and Smith, 2014). First, the difficulty of writing a computer program was resolved in a very simple way. In Alice, there are no syntax problems as the novice programmer does not need to type code. In Alice, users place and manipulate objects according to predefined commands (Powers et al., 2007). When the user selects a particular object, a drop-down menu appears asking the user to select an option or multiple options in a single command. In this approach, the syntax errors are avoided but of course not the runtime errors, and/or the logic errors. Instead, novice programmers’ efforts are directed towards understanding the concepts and mechanisms of programming (Al-Tahat, 2010). Second, Alice allows programmers to see the effects of their programs in real time during the development phase after their completion (Adams, 2008). As objects are selected from the corresponding library, the programmer can choose to ‘work’ partially with them before program completion in order to check their behaviour. As a result, novice programmers who use Alice are able to literally ‘see’ the behaviour of their program during its development phase.
Comparing novice programing environments for use in secondary education

The third difficulty is associated with lack of motivation when coding and it is considered the most serious possible reason for students’ repulsion towards introductory programing courses (Daly, 2011). Part of the difficulty stems from the objective assessment that learning programing is a boring task and the time it takes an inexperienced programmer to develop a simple program is quite long (Jenkins, 2002; Powers et al., 2007). In other words, the time/output ratio is large enough to cause someone to believe that it is not worth investing the time in learning about programing. Coding in Alice, however, solves these particular programing problems. Alice uses a visual output. The objects in Alice are 3D models. This output is visually appealing to most students (Ali and Smith, 2014).

Fourth, the coding time needed for writing simple Alice programs is minimum compared to other popular programing languages. Furthermore, Alice engages novices both in the development and the testing phase. Using real-world metaphors, Alice is not limited to printing simple texts on the screen as with many other popular programing languages. Instead, the environment handles objects that jump, talk, change colour or perform similar other actions widely used in game development. In other words, the use of Alice helps to weaken the sense that programing is boring, while strengthening novices’ incentives to programing. Student’s behaviour changes as they are able to create interesting programs in less time (Guibert et al., 2004).

4.2 Disadvantages of using Alice in introductory programing courses

Surprisingly, the main disadvantage of using Alice in introductory programing courses is that Alice is strictly a teaching/learning tool (Ali and Smith, 2014). Alice is not used in industrial applications such as employee payroll software programs and other critical software applications. In other words, just Alice cannot guarantee a job in the software industry, even on an introductory level. Students with some programing experience do not consider Alice an interesting and challenging environment (Powers et al., 2007). Indeed, some students may think that their participation and attendance in Alice programing courses will turn them away from a deep understanding of programing (Adams, 2008). Furthermore, a serious disadvantage is that after their initial enthusiasm fades, students realise they are not programing in a ‘real’ programing language so their interest dissipates (Lewis et al., 2014). Lewis (2010) concluded that student in primary and secondary education who program in languages that require them to write code, such as Logo, display greater confidence in problem-solving than those who program in Scratch. Moreover, Alice does not address the difficulties that arise in a course using a standard programing language such as Java. Students-novice programmers will sooner or later come up against problems in syntax and logic of a general-purpose language (Ali and Smith, 2014).

Also, as students learn and use some advanced programing concepts they will need to spend more time working individually instead of with a team for software development. As a consequence, some of the advantages of using Alice can potentially can act as a barrier, putting students at a disadvantage in more advanced programing courses (Powers et al., 2007). The added value that Alice provides to a student’s CV is not important, especially if the student chooses programing for professional rehabilitation as employers are looking for specific skills and qualifications for staff developer jobs (McLean, 2006; Scott-Bracey, 2013).
5 The programing environment App Inventor for Android

As with Alice, AIA provides a visual programming environment in which students can create their applications simply by dragging and connecting different blocks of commands like pieces in a puzzle. The development environment is fairly recent, as it was created as a pilot project in Google’s lab, in 2009. AIA was made available through request on July 2010, and released publicly on December 2010. In the second half of 2011, Google released the source code, terminated its server and provided funding for the creation of The MIT Centre for Mobile Learning, led by App Inventor creator Hal Abelson and fellow MIT professors Eric Klopfer and Mitchel Resnick. The MIT version was launched in March 2012. In December 2013, MIT released AIA 2, renaming the original version ‘App Inventor Classic’. The major difference between these two versions is that the block editor in the original version ran in a separate Java process, using the Open Blocks Java library for creating visual blocks (Wikipedia, 2016).

However, unlike Alice, in AIA, the students-novice programmers do not create applications that run on conventional computers, but on smart mobile devices with the Android operating system (smartphones/tablets) (Zaranis et al., 2013). Research findings show that mobile technologies support student engagement and that carefully designed educational activities with the use of smart mobile devices, encourage a deeper understanding of programming (Bradley and Holley, 2011; Duffy, 2012; Foti et al., 2014; Sung et al., 2016). Essentially, AIA takes advantage of the rapid adoption of mobile devices by young people in order to increase students’ interest in programming. With AIA, students do not need any prior knowledge of programming (Papadakis et al., 2014).

The AIA interface strongly resembles the interface of Scratch, the free visual programming language developed by the MIT Media Lab. However, AIA is more sophisticated than Scratch and like Alice it is possible to convert App Inventor apps into Java Android apps. Java Bridge allows users to use AIA components in Java programs. That means that a programming teacher can use AIA to introduce novices to programming in a simple pleasant way and as students mastered the basics of coding to ‘transfer’ them to a general-purpose programming language as Java (Roy, 2012). The AIA provides a real-time (or almost real time) adaptation to the user’s actions. In general, changes that a user makes in the interface or the code of the program appear to take effect immediately either in users’ smart mobile devices or in the AIA embedded emulator. This is a completely different operating mode in comparison to traditional software development environments in which programs should be compiled and executed again after each new modification. When a user completes a project in AIA, he can generate the final program in .apk format (Android Application Package) in order to install it on an Android device, or to store it locally on a computer so as to later distribute it on other Android devices or even to upload it in the Google store (Google Play) either free or with a subscription.

In order for someone to create or to access their own application using the AIA environment, they just need to visit the AIA website (http://appinventor.mit.edu/). Figure 3 shows the development structure of the AIA environment.
Figure 3  The structure of the App Inventor environment (see online version for colours)

Adapted from: App Inventor, 2016

One advantage of using the AIA in the school laboratory is that if an Android device is not available, the student is able to create and control the operation of an application using the embedded Android emulator (Figure 4). App Inventor provides an Android emulator, which is a software running on a computer and acts as a mobile phone. The Android emulator supports most features of a device but does not include virtual hardware for advanced usage such as Wi-Fi, Bluetooth, NFC, etc. In any case, it is appropriate for teaching introductory programing in secondary education. The only disadvantage of using the mobile device emulator is that it is relatively slow (as it simulates both software and hardware), compared to a real mobile device.
Figure 4  Emulator various stages of preparation for simulating an Android phone (see online version for colours)

The AIA programming environment consists of two distinct parts:

- **Designer**: The programmer selects the necessary components for the application and adjusts their properties.
- **Blocks editor**: The programmer places the pieces of code so as to ‘tell’ program components how to ‘behave’.

Figures 5 and 6 show the designer and the block editor in the AIA environment during an app development phase.

Figure 5  The designer provides an intuitive interface for choosing the app’s components and their properties (see online version for colours)
Comparing novice programming environments for use in secondary education

5.1 Advantages of using AIA in introductory programming courses

App Inventor for Android is a programming environment which is based on the ‘low floor, high ceiling’ philosophy first espoused by Logo developers (Martin and Sherman, 2015). AIA has the advantage that its generated products (mobile apps) can have practical use in students’ real life. For example, with AIA students can create note apps, personal diaries, Twitter apps, maps, etc. The strongest point of AIA lies in the nature of students’ products since their mobile applications can be tested or used directly on a mobile device. This gives a sense of practicality and reality in achievement, which can be a very satisfying experience in novice programmers learning. Like Scratch and Alice, the process of creation with AIA can stimulate fun, creativity and learning about programming (Hsu and Ching, 2013).

According to Morelli et al. (2011), the advantages of using AIA in introductory programming courses are as follows:

- **Simplicity**: The AIA environment is very simple. Teachers can use it effectively in just 2 weeks. Novice programmers do not need previous programming knowledge to use it, as they just have to drag-and-drop and connect command blocks.

- **Installation ease**: AIA is a web-based environment, and as a result, its setup is much easier than other Android development environments (Eclipse, IntelliJ).

- **Target audience**: AIA enables young students and novice programmers to create their own mobile apps, without having to know what a garbage collector is for example.

- **Aesthetically pleasing**: AIA enables students to design apps more efficiently without feeling daunted when they look at countless lines of code from other programming environments.

The purpose of introducing a block-based programming environment to novices is not for them to learn the more advanced parts of coding, rather the use of it is as a learning scaffold for the first real programming language that students will learn. Finally, unlike Alice, AIA can also be used by a programmer to make real software products. Several of the applications that are available on Play Google (either free or with subscription) may have been created with AIA.
5.2 Disadvantages of using AIA in introductory programming courses

App Inventor for Android is an excellent mobile app development tool for novice programmers. However, students become frustrated when they try to create more complex applications with a better user interaction, user interface design, multiple screens, etc. These limitations include the following:

- **Look and feel:** In terms of software design, the app’s user interface, created with AIA is reminiscent of the poor user interface design of Android 2.3 Gingerbread or earlier versions of the Android operating system and has no features of modern design patterns like Google’s material design. It also does not offer ready-made templates, so the design is left solely to the user’s imagination.

- **Functionality:** In comparison with other app development environments, app development with AIA shows limitations in terms of what can be done/achieved. For example, AIA does not support the use of broadcast receivers, or cannot execute background services and finally there are several restrictions on the use of graphics.

6 A comparison between AIA and Alice in their educational value

Unlike with traditional approaches to teaching programming, both AIA and Alice motivate students through the development of programs based on their interests. For this reason, both environments are equally attractive and productive for both male and female students (Cooper, 2010) as well as ‘high-risk’ students, namely students who experience difficulties in understanding and mastering math concepts and have little to no programming experience (Dann et al., 2012). The common objective in the design of Alice and AIA is the provision of a rich and attractive environment for novice programmers (Cooper, 2010).

Both software development teams consider that syntax errors (the errors in the program that make it impossible to parse/interpret, e.g., the error of missing a semicolon) are substantial during the learning phase, in contrast to semantic errors (the errors in the program that make it do something other than what the programmer intended, e.g., the error in the use of methods). It is important for a novice programmer to focus on the semantic difficulties and it is not necessary to focus on accidental syntax errors which derived from the use of a specific programming language (Cooper, 2010). In both environments, the built-in editor with ‘drag-and-drop’ features prevents those errors that beginners frequently make, such as forgetting brackets and semi-colons (Cooper, 2010). AIA and Alice support the use of high-level and low-level commands such as the use of loops. Also, the result of the code execution is immediately visible on a computer monitor or a mobile device. Additionally, both environments can support the transition from primary to secondary and/or tertiary education through the schema “Scratch → Alice/AIA → Object-oriented programming language”. A differentiation between the two environments is that Alice outweighs AIA in relation to the program execution monitoring, which in many cases is superior compared to most professional programming languages. Possibly, Alice provides the best debugging support in comparison to the other NPEs (Cooper, 2010).
Additionally, Alice is the only programming environment for novices that supports the use of 3D graphics. The use of 3D graphics is ideal for certain types of projects such as stories, games and virtual worlds. AIA, although supporting the creation of stories and games, cannot support the creation of virtual worlds. However, other types of projects such as simple games or quizzes are easier to implement in the two-dimensional (2D) world. It is very easy for a student to use 2D images instead of creating or seeking three-dimensional models. AIA is superior compared to Alice, in the range of projects that students regardless of gender are willing and motivated to do. AIA looks a more ideal learning environment compared to Alice as it can motivate students due to the portability and the practical use of the mobile applications they create.

Both environments are theoretically equivalent to the extent that the suggested learning topics in an introductory programming course are covered. Additionally, AIA and Alice can introduce students to more advanced concepts of computer programming including procedures, iterative and recursive thinking, structured project analysis, conditional and logical thinking, and debugging (Shuchi and Roy 2013). But the existence of an embedded emulator in AIA makes it more flexible than its competitors since it does not require the use of mobile devices in the school computer lab.

A disadvantage of Alice usage is that its special graphical user interface can be counterproductive when Alice is used as just an animation and multimedia presentation tool and not as a teaching tool (Parsons and Haden, 2007). In their research, Parsons and Haden (2007) found that some students focused on the creation of 3D animated virtual worlds and failed to recognise the programming capabilities of Alice. A similar example is the Scratch community. Although there are more than 3 million projects available online it has been found that in their majority Scratch users are focusing mainly on characters’ design and/or modification, placing less emphasis on the development of their programming skills. For example, only 10% of the projects in the online Scratch community use the programming structure ‘repeat until (condition)’ and only 20% of the projects use variables (Meerbaum-Salant et al., 2013). It is important for an informative teacher to investigate whether the use of such programming environments facilitates the learning of basic programming concepts or if the student is disoriented in the multimedia capabilities of the environment.

7 A comparison between AIA and Alice in system-hardware requirements

Due to their different nature, the two environments differ in hardware requirements. The hardware requirements for Alice, according to the official Alice site, are at least 1 GB RAM and 600 MB hard disk space. These are the typical characteristics of a 5 to 10-year-old computer. In fact, although several PCs are designed and work well with 2D graphics, it is not certain that they can produce 3D graphics. As for the RAM, although 1 GB is the minimum, for a best experience the user should aim at 2 GB or more. The video card must have enough memory to support 32-bit colours with a minimum resolution of 1024 × 768 pixels. 3D graphics cards offer the best performance although theoretically not required. Alice 3 can run on all of the recent versions of the known operating systems.
Although the above hardware requirements do not seem demanding for a home computer, however, they can act as a deterrent for the satisfactory use of Alice in the outdated computer labs of most schools. Especially for older computers, there is a special rendering version of Alice that will run code through the machine’s CPU to render Alice on the screen rather than trying to take advantage of the machine’s video card. The disadvantage is that it is slower than the hardware accelerated version of Alice, but it is more reliable, especially on machines with older video cards. In any case, Alice developers recommend that a user to try the standard version of Alice first, and if it does not run well, then try the slow and steady version (Herbert, 2010, p.312).

App Inventor for Android do not need to be installed on each computer, as it runs through a web browser. All known browsers are supported except Microsoft Internet Explorer. For mobile application testing (in case the user does not want to use the embedded emulator), a smart mobile device with Android OS 2.3 or later is required. Additionally, in case of internet connectivity problems, there are two unofficial offline versions of AIA, AiLiveComplete (https://sourceforge.net/projects/ailivecomplete/) and App Inventor 2 Ultimate (https://sourceforge.net/projects/ai2u/). Table 1 shows the information about the minimum hardware and software requirements that are needed for Alice and AIA.

Table 1  Minimum hardware and software requirements for Alice and AIA

<table>
<thead>
<tr>
<th>Specification</th>
<th>Alice</th>
<th>AIA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>&gt;=1GB</td>
<td>-</td>
</tr>
<tr>
<td>Hard disk</td>
<td>590MB</td>
<td>-</td>
</tr>
<tr>
<td>Video card</td>
<td>32 Bit colour depth</td>
<td>-</td>
</tr>
<tr>
<td>Resolution</td>
<td>1024 × 768</td>
<td></td>
</tr>
<tr>
<td>Operating system</td>
<td>Windows (&gt;XP)</td>
<td>Windows (&gt;XP)</td>
</tr>
<tr>
<td></td>
<td>Mac OS X (&gt;10.5)</td>
<td>Mac OS X (&gt;10.6)</td>
</tr>
<tr>
<td></td>
<td>Ubuntu (&gt;8)</td>
<td>Ubuntu (&gt;8)</td>
</tr>
<tr>
<td></td>
<td>Red Hat</td>
<td>Debian (&gt;5)</td>
</tr>
<tr>
<td>Web browser</td>
<td>-</td>
<td>Mozilla Firefox (&gt;3.6)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Safari Apple (&gt;5.0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Google Chrome (&gt;4.0)</td>
</tr>
<tr>
<td>Mobile device</td>
<td>-</td>
<td>Operating System Android (&gt;2.3)</td>
</tr>
</tbody>
</table>

AIA, App Inventor for Android

Figure 7 shows the processing power needed (and particularly the graphics card) when Alice runs on a typical 2-year-old computer system, as resulting from the application Microsoft Process Explorer v16.05. Figure 8 shows the respective results using AIA.
Comparing novice programming environments for use in secondary education

Figure 7  Graphics card performance when using Alice (see online version for colours)

![System Information Screenshot](image1.png)

Figure 8  Graphics card performance when using AIA (see online version for colours)

![System Information Screenshot](image2.png)

Note:  AIA, App Inventor for Android

It is obvious that the use of AIA in terms of hardware requirements is an optimal choice as it requires fewer computational resources. In an obsolete school computer lab, the use of Alice seems unrealistic. This comparison is addressed mainly in the Greek context but many of the problems mentioned above may be transferable to other educational contexts with the same socioeconomically conditions.
8 A case study

The main purpose of the case study was to examine which of the two programing environments contribute more effectively to the learning of basic programing concepts for novice programmers. Additionally, we wanted to investigate the effect of the programing environment in students’ attitudes about programing.

8.1 Participants

In the 2016–2017 school year, 31 young people aged 15–16 (13 females and 18 males) in a General Lyceum (Upper Secondary School) in Heraklion, Crete, Greece participated in the study. The first-year students attended the subject entitled ‘Computer Applications’. The study consisted of a series of lessons as well as a game or story creation project as part of students’ IT lessons. The pupils were spread over two classes; both were taught the same material by the same IT teacher. The IT teacher had more than 17 years of teaching experience. The pupils in the study had previously learned the same programing language (LOGO) as all of them came from the same school (Gymnasium) and had not taken part in any project activities during previous school years. The teacher in collaboration with the researchers created the learning material that was distributed to students so as to be compatible with the requirements of the present study as well as the Greek Ministry of Education’s curriculum policy. In all, 16 students (52%) constituted the experimental group, and 15 students (48%) the control group. Both groups had a very nearly equal number of boys and girls. The project ran over one and a half-term, in fourth five minute on average lessons, two times per week. As a result, pupils across both classes spent approximately 16 h on the study.

8.2 Teaching intervention

The main learning objective of the teaching intervention was that students understand the basic programing concepts using either the Alice or the AIA environment. The experimental group of students worked with AIA, whereas the control group students worked with Alice. During the first 8 h in both settings, participants were given demonstrations by the teacher, introducing them to the key tools and functionality of each programing language. These demonstrations took the form of instructional videos which the presenter explained as they went along and included basic programing features. Additionally, students were given specifically designed worksheets that guided them in the implementation of tasks of escalating difficulty. The worksheets were identical in the two groups of students differentiating only in the learning programing environment (AIA - experimental group/Alice - control group). Through these worksheets, students had the opportunity to understand the additional value of programing. Pupils worked in pairs (with one group of three). In the fifth week, the students of both groups were asked to create a project (e.g. a game) of their choice either on Alice or in the AIA programing environment. The duration of this phase was 8 h. The research protocol conforms to the ethical guidelines of the European Union.
8.3 Results

At the beginning of the first lesson, students of both groups answered a pre-test which was comprised of questions related to participants’ conceptions about programming as well as their knowledge about basic programming concepts. The same test was also administered as a post-test at the end of the study. The duration of both the pre-test and post-test was 15 min each. The formats of the questions were multiple-choice, right/wrong-choice and “put the commands/instructions in the right order”. Additionally, at the end of the project, semi-structured interviews were carried out with the 31 pupils. As noted above, during the interviews, the participants were asked to discuss their usage of the programming environment and explain various components of the environment interface, in order to gauge their understanding. Additionally, they were asked about their attitudes towards the programming environment they used. Of the experimental group pupils interviewed, the majority noted that they had managed to make things happen in their application using AIA. Of the control group, some students noted that they had used Alice but had not been able to create a working project as their code did not work as expected. When the students who had used AIA were asked about ease of use, in their majority they described it as ‘really easy’ to use. The students who had used Alice said that it was ‘between hard and easy’, and they further explained that there were both hard and easy parts both in the interface and the code. According to their previous programming experience with the LOGO language, students in both groups report that numerous factors contribute to making block-based programming easy, including the natural language description of blocks, the drag and drop composition interaction, and the ease of browsing the language. Of the 31 responses collected, all the students included ease-of-use as a major difference between the graphical and text-based environments. Students in both groups noted that what makes block-based programming easy is the visual nature of the blocks and the graphical cues that each block provides for how and where they can be used.

The students’ code project analysis showed that novice programmers in the experimental group used a relatively large number of conditionals statements and loop structures in comparison to the control group. For example, in the experimental group, all of the participants succeeded in creating at least one full script with an event, while more than half successfully added conditionals and repeating blocks in their code. In contrast, the control group students used the selection and loop structures to a lesser extent, and were limited in the use of sequence structures. Additionally, events seemed to cause problems for some students of the control group. During the course of the study, the teacher observed that the control group of students (with respect to the experimental group) mostly used the command blocks incorrectly. In the control group, a few users tried to drag the wrong type of block into the conditional slot, but could tell that something was wrong, as the block would not snap into place. This caused momentary frustration to them, but in each case, it appeared to alert them to the fact that the block was not of the right type to fit. The successful completion of the projects in both groups showed that students were able to identify a problem, select the most effective solution on the basis of and drawing inspiration from the introductory part, and finally create their own solution. The presence of specific blocks in the final applications confirms that students in the experimental group used the corresponding computational concept to a higher degree compared to the students of the control group.
To check whether Alice or AIA facilitated students’ development and understanding of basic programming concepts, respectively, we compared novice responses in pre-test and post-test. Students answer in the pre-test showed that the groups were not equal. ANOVA test showed a significant difference at the \( p < 0.05 \) level, \( F(1,29) = 3.77, p = 0.01 \). The control group scored higher (\( M = 42.21, SD = 5.10 \)) compared to the experimental group (\( M = 37.21, SD = 4.70 \)). At the end of the course, ANOVA tests showed no significant difference at the \( p < 0.05 \) level for both the Alice and the AIA groups, regarding student answers in the post-test \( F(1,29) = 4.00, p = 0.69 \). Previous analyses showed that the two groups were not equivalent in their scores in the pretest. As a result, it cannot be ascertained to what extent the observed differences between the means after the intervention were due to the effectiveness of the intervention itself, or if they simply reflected existing differences between the groups before the intervention. An analysis of covariance in the performance of students showed that the main effect of the independent variable (group type) was found to be statistically significant, \( F(1, 29) = 4.13, p < 0.01 \) partial \( \eta^2 = 0.11 \). After removing the covariate effects, the highest adjusted mean was scored by the experimental group (AIA) (EMM = 67.86, SD = 4.92) followed by the control group (EMM = 52.17, SD = 4.47). EMM refers to the estimated marginal means with pre-test score of the dependent variable. The result shows that students with higher performance in the pre-test did not have higher performance in the post-test compared to the students with lower performance in the pre-test. Besides, AIA and Alice assessment on student knowledge acquisition was interested in finding out the degree of student satisfaction towards the programming environment. The majority of the students, in their assessment at the end of the activity, noted that they enjoyed the activities. Students of both groups were able to recognise their own potential to use technology in their path, which means that they proposed other examples of possible usages of CS and programming in their career. However, students’ answers showed that the experimental group was more excited than the control group. Creating a mobile application helped to increase the interest and curiosity of students, as they belong to an age range that makes strong use of devices like cellular phones or tablets. Some of the students’ answers are presented in Figures 9 and 10.

**Figure 9** Experimental group answers (see online version for colours)
Comparing novice programming environments for use in secondary education

Figure 10  Control group answers (see online version for colours)

9 Discussion and conclusion

Programming appears to be the hardest faculty to master when dealing with computers (Efopoulos et al., 2005). It is clear that novice programmers face a very difficult task. Learning programming includes, besides the acquisition of new complex knowledge, the cultivation of students’ related strategies and practical skills. Teaching programming has many peculiarities that are not conducive to traditional teaching approaches, such as frontal teaching, the strict structure of the content and conventional student activities. However, even today, teachers follow the same unproductive teaching methods and strategies, i.e. traditional programming curricula with a traditional language (Utting et al., 2010) through activities that do not intrinsically interest students.

In connection with the application of constructivist pedagogical approaches to teaching computer programming in high school, the aim is not the teaching of a programming language, but the help that should be given to students in order to build the necessary conceptual frameworks to practice programming (Wulf, 2005). An overview of educational programming environments shows a variety of available schemes that can cover the entire range of student ages (Utting et al., 2010). Each programming environment adopts a different model for approaching the basic programming concepts. The choice of an educational environment that will support the teaching and learning of novice programmers becomes necessary in order that an exploratory approach to teaching and learning be promoted, as well as collaborative learning and skills development that promote the development of students’ computational thinking skills.

This paper deals with a description of two NPEs, Alice and AIA. A comprehensive evaluation of those two NPEs requires research using systematic classroom observation. However, from a technological point of view, AIA seems to emerge as more powerful NPE for adoption in the secondary education in comparison with Alice. The clearly fewer requirements to technological resources of AIA compared to Alice but also the scope for Bring Your Own device (BYOD) solutions make AIA an optimum solution for each school and especially for technologically obsolete school computer labs. From a pedagogical point of view, both NPEs have their strong and weak points. AIA is considered an ideal tool to engage young students compared to Alice using students’
mobile devices for learning purposes. It is not a coincidence that AIA is considered the right tool at the right time, satisfying young people’s thirst for smart mobile devices.

We hope our study to be a useful guide in the hands of every secondary education teacher, who teaches or designs introductory programming courses in an attempt to attract more students to programming.

References


Comparing novice programming environments for use in secondary education


Herbert, C. W. An introduction to programming using Alice 2.2. Cengage Learning, 2010. Boston, Massachusetts, USA.


Comparing novice programming environments for use in secondary education


