MINISTRY OF EDUCATION AND SCIENCE OF REPUBLIC OF KAZAKHSTAN
SULEYMAN DEMIREL UNIVERSITY
ENGINEERING FACULTY

Department of Computer Engineering

UDC 004.9:373.5

Zhumagulova Dana Galymzhanovna

Using the Programming Environment Scratch in Kazakh Schools for Development of Children

6M070400– «Computing systems and software» speciality

Kaskelen, 2013

MINISTRY OF EDUCATION AND SCIENCE OF REPUBLIC OF KAZAKHSTAN
SULEYMAN DEMIREL UNIVERSITY
ENGINEERING FACULTY

SULEYMAN DEMIREL UNIVERSITY

Department of Computer Engineering

«Accepted for defense»
responsible director of department
d.t.s professor Amirgaliyev E. N.
«___» 06 2013 year

Director of department
for postgraduate education
Ph.s.c. Sh. Aydogdu
«___»_____ 2013 year

Master dissertation

Using the Programming Environment Scratch in Kazakh Schools for Development of Children

6M070400– «Computing systems and software» speciality

Master student _____ Zhumagulova Dana Galymzhanovna

Supervisor _____ d.t.s., proffesor Niyazi Ari

Kaskelen, 2013

АҢДАТПА

Дайындалған диссертацияның негізгі мақсаты: қазақ мектептерінде оқушылардың ой-өрісін дамыту мақсатында Scratch программалау ортасын пайдалану үшін оқулық жасау.

Оқу құралы ретінде туториал-оқулық дайындалды. Оқулық 15 зертханалық жұмыстан тұрады. Әрбір зертханалық жұмыс жаңа тақырыптар түсіндіріле отырып, мысал жаттығулар орындауға арналған. Зертханалық жұмыс орындалғаннан кейін, артынша сабақтың қорытындысы, сол сабаққа қатысты сұрақтар мен тапсырмалар берілген.

Оқушы сабақ уақытында сол күнгі сабақты дұрыс түсінбеген жағдайда немесе белгілі себептермен қатыспаған болса, оқушы бос уақытында қосымша сабақ ретінде оқу құрáлын қолданып дайындала алады.

Тақырыптар кіші топтарға бөлініп оқушыларға оқу материалдарын меңгеруін тездетуге және оңайлатуға мүмкіндік жасап, ең маңызды, керекті жақтары алынып және оларға жақсы мысалдар беру арқылы түсіндірілген.

Нәтижеде оқушылар оқу құралын пайдалана отырып, Scratch программалау ортасында жұмыс жасау арқылы программалауға деген қызығушылығы артады. Олар өздігімен түрлі анимациялар, ойындар және кіші мультфильмдер құрастыруға қабілеттері ашылады.

АННОТАЦИЯ

Основная цель подготовленной диссертации: изготовить учебный материал о среде программирования Scratch для использования в казахских школах в целях развития детей.

В качестве учебного материала подготовлен учебник-туториал. Учебник состоит из пятнадцати лабораторных работ. Каждая лабораторная работа составлена так, что выполняя работу ученик изучает новую тему. После каждой лабораторной работы даются краткое заключение урока и контрольные вопросы и задания.

В случае, если ученик пропустил занятие или не освоил урок полностью, он может в свободное время пользоваться учебным материалом и готовиться самостоятельно.

Темы разделены на группы, что дает возможность ученикам ускорить усвоение учебных материалов и оптимизировать процесс обучения, и все разделы разъяснены по приведенным примерам.

В итоге, используя учебный материал, в то же время программируя в среде программирования Scratch у учеников повысится интерес к программированию. Они самостоятельно смогут составлять разные анимации, игры и мини-мультфильмы.

ANNOTATION

The main aim of this thesis paper is to produce educational material about Scratch programming environment for use in Kazakh schools for development kids.

As a teaching material was prepared tutorial-book. The tutorial consists of fifteen laboratory works. Each lab is constructed so that doing the work a student is studying a new subject. After each lab there are conclusion of the lesson and lesson quizzes and assignments.

If a student did not understand the given topic of school material for that day or he did not participate at the classes due to certain circumstances then he can use this tutorial as an additional class and prepare at home.

Topics are divided into small groups and explained with the most important and relevant information with examples so that students could learn the course material more quickly and easily.

As the result, students will get more interested in programming after programming on Scratch. They will have abilities to crate animations, games and short cartoons by themselves.

CONTENT

ABBREVIATIONS

MIT – Massachusetts Institute of Technology

ICT – Information and Computer Technologies

HTML – HyperText Markup Language

IDE – Integrated Development Environment

OS – Operation System

URL – Uniform Resource Locator

INTRODUCTION

Computing has become the framework of modern daily life. From personal computers to telephones to cars and cameras, much of what we use for work and play, communication and entertainment is controlled by software. To find jobs, housing or education and others, we turn to the web for information and resources. Employment of computer systems analysts, engineers and scientists is expected to double as organizations become increasingly dependent on advancing technologies and the workers who design and implement them.

Within a few generations, computers have come to underlie the foundations of all industries from natural sciences to health care to finance and engineering. International commerce, banking, airline flights and local subway schedules are now created, recorded and analyzed via software. The field of biology has completely changed with the advent of software and hardware to process genome data.

As many other fields become dependent on computing, professionals in those fields must not only acquire expertise in their own fields, but competency in computing as well. In the globally connected world of the new millennium, software technology serves as a driving force behind the innovation that propels the international economy forward. Fields as diverse as social sciences, economics, computational finance and marketing rely on the digital computing power of software to collect, analyze and transform data sets far too large for humans to manage.

Today's world is full of interactive objects. Walk up to a door, and it opens automatically. Play with a toy, and it flashes, beeps, and talks in response to your actions.

Connect to a website, and animations react to the movements of your mouse. Log into an online game, and you can interact with fantasy creatures and other virtual objects. Boot up a science simulation, and you can explore the behavior of a bird flock or motions of the planets.

Young people interact with these objects everyday. But very few of them can create these objects. The creation of interactive objects requires the ability to program, but traditional programming languages are notoriously difficult to learn and understand. As a result, most young people are not fully fluent with digital media – they can "read" but not "write." While Web 2.0 has opened up opportunities for everyone to create and express themselves with text, audio, and video, only a small subset of the population can create and express themselves with interactive media.

And today's computer science educator's goal is to move the current generation from being users of technology to creators of technology. To do this, we must teach this generation computing.

Computing offers well-paying careers in positions as varied as software developer, systems analyst, database architect and web developer and etc.

In today's rapidly changing world, people must continually come up with creative solutions to unexpected problems. Success is based not only on what you know or how much you know, but on your ability to think and act creatively. In short, we are now living in the Creative Society.

Unfortunately, few of today's classrooms focus on helping students develop as creative thinkers. Even students who perform well in school are often unprepared for the challenges that they encounter after graduation, in their work lives as well as their personal lives. Many students learn to solve specific types of problems, but they are unable to adapt and improvise in response to the unexpected situations that inevitably arise in today's fast-changing world.

New technologies play a dual role in the Creative Society. On one hand, the proliferation of new technologies is quickening the pace of change, accentuating the need for creative thinking in all aspects of people's lives. On the other hand, new technologies have the potential, if properly designed and used, to help people develop as creative thinkers, so that they are better prepared for life in the Creative Society.

In this thesis, I will discuss about the technology developed by Mitchel Resnick's research group at the MIT Media Lab with the explicit goal of helping people develop as creative thinkers. The technology, called Scratch, is designed to support the "creative thinking spiral." In this process, people imagine what they want to do, create a project based on their ideas, play with their creations, share their ideas and creations with others, and reflect on their experiences—all of which leads them to imagine new ideas and new projects. As students go through this process, over and over, they learn to develop their own ideas, try them out, test the boundaries, experiment with alternatives, get input from others, and generate new ideas based on their experiences.

1 COMPUTER PROGRAMMING

1.1 What is computer programming?

Computer programming (often shortened to **programming, scripting,** or **coding**) is the process of designing, writing, testing, debugging, and maintaining the source code of computer programs. This source code is written in one or more programming languages (such as C++, C#, Java,Python, Smalltalk, etc.). The purpose of programming is to create a set of instructions that computers use to perform specific operations or to exhibit desired behaviors. The process of writing source code often requires expertise in many different subjects, including knowledge of the application domain, specialized algorithms and formal logic.

Within software engineering, programming (the *implementation*) is regarded as one phase in a software development process.

Different programming languages support different styles of programming (called *programming paradigms*). The choice of language used is subject to many considerations, such as company policy, suitability to task, availability of third-party packages, or individual preference. Ideally, the programming language best suited for the task at hand will be selected. Trade-offs from this ideal involve finding enough programmers who know the language to build a team, the availability of compilers for that language, and the efficiency with which programs written in a given language execute. Languages form an approximate spectrum from "low-level" to "high-level"; "low-level" languages are typically more machine-oriented and faster to execute, whereas "high-level" languages are more abstract and easier to use but execute less quickly. It is usually easier to code in "high-level" languages than in "low-level" ones.

Allen Downey, in his book "How To Think Like A Computer Scientist", writes:

The details look different in different languages, but a few basic instructions appear in just about every language:

- **input**: Gather data from the keyboard, a file, or some other device.
- **output**: Display data on the screen or send data to a file or other device.
- **arithmetic**: Perform basic arithmetical operations like addition and multiplication.
- **conditional execution**: Check for certain conditions and execute the appropriate sequence of statements.
- **repetition**: Perform some action repeatedly, usually with some variation."

A *program* is a set of step-by-step instructions that directs the computer to do the tasks you want it to do and produce the results you want.[1]

There are at least three good reasons for learning programming:

- Programming helps you understand computers. The computer is only a tool. If you learn how to write simple programs, you will gain more knowledge about how a computer works.
- Writing a few simple programs increases your confidence level. Many people find great personal satisfaction in creating a set of instructions that solve a problem.
- Learning programming lets you find out quickly whether you like programming and whether you have the analytical turn of mind programmers need. Even if you decide that programming is not for you, understanding the process certainly will increase your appreciation of what programmers and computers can do.

1.2 Importance of programming

The question most of the people ask is why should we learn to program when there are so many application software and code generators available to do the task for us. Well the answer is as give by the Matthias Felleisen in the book "How to design programs": "The answer consists of two parts. First, it is indeed true that traditional forms of programming are useful for just a few people. But, programming as we the authors understand it is useful for everyone: the administrative secretary who uses spreadsheets as well as the high-tech programmer. In other words, we have a broader notion of programming in mind than the traditional one. We explain our notion in a moment. Second, we teach our idea of programming with a technology that is based on the principle of minimal intrusion. Hence, our notion of programming teaches problem-analysis and problem-solving skills without imposing the overhead of traditional programming notations and tools." Hence learning to program is important because it develops analytical and problem solving abilities. It is a creative activity and provides us a mean to express abstract ideas. Thus programming is fun and is much more than a vocational skill. By designing programs, we learn many skills that are important for all professions. These skills can be summarized as:

- Critical reading
- Analytical thinking
- Creative synthesis.

Today's society sees Information and Communications Technology (ICT) almost exclusively via useful application software, with little understanding of what goes on behind the screen".

We argue that schools of general education should emphasize fundamental, timeless concepts underlying ICT, and that personal experience with writing small programs is a good way to introduce such concepts.

Computers permeate the technical infrastructure our society takes for granted: microwave ovens, washing machines, vcrs, ticket machines, electronic payment or online flight reservation systems are just some examples. The internet and world wide web have linked humanity.

A sound general education is necessary to keep up with the challenge of understanding these fast-changing technologies and to be able to cope with their increasing complexity. Schools face a difficult decision: what aspects of computer science to try to convey to students in the limited time available? Computer science comprises many different levels, but the general public only sees its applications. Educators may be tempted to concentrate on short-lived, product specific aspects of applications; to focus on low-level skills, such as "how to do it", rather than on understanding, on "why and how it works". Yet mastery of today's release may not be helpful a few years later when confronted with a new system or a different application. Moreover, teachers often focus on application-specific skills not only because these skills are of immediate use, but also because they themselves are just users of computers and do not know much about what is going on "behind the screen".

Schools that provide general education must promote an understanding of the fundamental concepts of computer science. Later, on the job, there is rarely time to study fundamentals. In language courses, for example, students learn not only vocabulary and grammar, but also basic concepts of communication, such as: How do I structure and formulate my thoughts for a specific audience? Physics courses teach the basic laws of nature, such as conservation of energy, and their consequences.

Such concepts of general education form the foundation which helps us keep up with life-long learning.

What are some of the fundamental concepts of computer science, and how can they be taught in school? One of the most fundamental aspects is that all computer systems are controlled by programs, i.e. by rigorously defined algorithms expressed in a formal notation. Modern society relegates an ever-growing number of every-day tasks to machines. These machines act as controllers that initiate actions based on their current state and on received inputs. The number of possible behaviors, of sequences of actions triggered by different environmental conditions, is usually so huge as to be impossible to enumerate. Yet, we aim to be sure that

each and every possible behavior, of which only a tiny fraction will ever be played out, is „correct“ in some precise sense. The way to do that is to write a specification that captures the practical infinity of processes that may evolve over time, depending on received inputs. A program is such a formal specification, and "program" is surely among the most fundamental concepts required to understand computers.[2]

Programming is a core activity of computing because it enables the user to access and release the potential of the computer they are using. Computer programming can be likened to playing chess – although there is a relatively small set of simple rules, it is the strategic and sustained application of those rules that can create interesting games between children or intellectual fights between grand masters. The same with programming, the first applications of the rules can produce the interesting results, fun play on graphics, numbers or words. But, there is no boundary preventing the learner moving all the way to being the grand master of computer programs. Once you can do it, the sky's the limit over what you can make computers do.

1.3 Teaching programming at school

It has become common place to refer to young people as "digital natives" due to their apparent fluency with digital technologies. Indeed, many young people are very comfortable sending text messages, playing online games, and browsing the Web. But does that really make them fluent with new technologies? Though they interact with digital media all the time, few are able to create their own games, animations, or simulations. It's as if they can "read" but not "write." As we see it, digital fluency requires not just the ability to chat, browse, and interact but also the ability to design, create, and invent with new media. To do so, need to learn some type of programming.

The ability to program provides important benefits. For example, it greatly expands the range of what you can create (and how you can express yourself) with the computer. It also expands the range of what you can learn.

In particular, programming supports "computational thinking," helping us learn important problem-solving and design strategies (such as modularization and iterative design) that carry over to nonprogramming domains. And since programming involves the creation of external representations of your problem-solving processes, programming provides you with opportunities to reflect on your own thinking, even to think about thinking itself.

When personal computers were first introduced in the late 1970s and 1980s, there was initial enthusiasm for teaching all children how to program. Thousands of schools taught millions of students to write simple programs in Logo or Basic.

Seymour Papert's 1980 book Mindstorms presented Logo as a cornerstone for rethinking approaches to education and learning. Though some children and teachers were energized and transformed by these new possibilities, most schools soon shifted to other uses of computers. Since that time, computers have become pervasive in children's lives, but few learn to program. Today, most people view computer programming as a narrow, technical activity, appropriate for only a small segment of the population.

What happened to the initial enthusiasm for introducing programming to children? Why did Logo and other initiatives not live up to their early promise?

There were several factors: "Early programming languages were too difficult to use, and many children simply couldn't master the syntax of programming; programming was often introduced with activities (such as generating lists of prime numbers and making simple line drawings) that were not connected to young people's interests or experiences; and programming was often introduced in contexts where no one could provide guidance when things went wrong—or encourage deeper explorations when things went right".

Papert argued that programming languages should have a "low floor" (easy to get started) and a "high ceiling" (opportunities to create increasingly 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). Satisfying the triplet of low-floor/high-ceiling/wide-walls hasn't been easy. In recent years, new attempts have sought to introduce programming to children and teens. Some use professional programming languages like Flash/ActionScript; others use new languages (such as Alice7 and Squeak Etoys5) developed specifically for younger programmers. They have inspired and informed our work on Scratch. But we weren't fully satisfied with the existing options. In particular, we felt it was important to make the floor even lower and the walls even wider while still supporting development of computational thinking. To achieve these goals, we established three core design rinciples for Scratch: Make it more tinkerable, more meaningful, and more social than other programming environments. In the following sections, we will discuss how each of these principles guided our design of Scratch.[3]

So Why teach programming?

Most of educators believe that teaching programming is important for two core reasons:

- firstly, it is a form of digital literacy that is of growing importance within society; and
- secondly, it promotes intellectual development and the development of problem-solving skills in a way that is applicable to many other subjects and in many other areas of life.

1.3.1 Programming as digital literacy

Computers are now instrumental to our society and the need for pupils to attain a form of 'digital literacy' is now generally accepted. This is currently interpreted as the need to be able to use standard applications, such as office-type software within a windows environment interface, proficiently. We agree that this is important.

However, the use of computers is changing rapidly. They are now as much mechanisms for social communication, as they are office tools. As this connectivity expands to every aspect of our lives, the ability to exercise control over the information becomes crucial. Controlling information is one of the fundamental skills of programming. If students master this skill, they will be able to engage successfully, not just with today's applications, but also with uses of technology that have yet to be devised.

Programming offers the ability to create new uses for computers. Whereas a competence in office-type software allows the production of new documents, programming allows the creation of new behaviours, rather than just the consumption of behaviours provided for us by others.

Computer programming is carried out by many people and is a hobby, pastime, leisure pursuit, interest, diversion, relaxation... For our pupils, it could be a way of enabling them to "enjoy and achieve" .

"In some senses, computer programming itself is one of the best computer games of all. In the 'computer programming game', there are obvious goals and it's easy to generate more. The 'player' gets frequent performance feedback (that is, in fact, often tantalizingly misleading about the nearness of the goal). The game can be played at many different difficulty levels, and there are many levels of goals available, both in terms of the finished product (whether it works, how fast it works, how much space it requires, etc.) and in terms of the process of reaching it (how long it takes to program, etc.). Self-esteem is crucially involved in the game, and there is probably the occasional emotional or fantasy aspects involved in controlling so completely, yet often so ineffectively, the behaviour of this responsive entity. Finally the process of debugging a program is perhaps unmatched in its ability to raise expectations about how the program will work, only to have the expectations surprisingly disappointed in ways that reveal the true underlying structure of the program",Malone, 1980.

Computer programming is also a vocational pursuit and may enable our pupils to "achieve economic wellbeing", Pupils discovering their proficiency in handling syntax, algorithm, logic and analysis may find they can enter an industry in which those skills are highly valued. This teaching resource is designed to bring

computer programming to every pupils' experience because it contributes in a powerful way to their ability to learn, conceptualise and understand. Computer programming is a subset of a computing curriculum and, together with other knowledge, experience, skills and understanding of computing, it exercises skills that are valuable in other aspects of learning, work and leisure. It also gives an insight into why computers behave as they do and therefore puts an understanding into the ICT curriculum.

Hence, programming:

- enables pupils to enjoy and achieve;
- develops problem-solving skills through both individual endeavour and team work;
- provides experience of a powerful way to "learn, conceptualise, and understand".

"I'd add that programming is a tremendous "motivator" for students, because it makes computers "come alive" and "dance to their tune". Maybe the language of "computational thinking" belongs here? One way to put it is this is "programming makes abstraction concrete". "Abstraction" is a tremendously powerful idea that we use again and again but it is, by definition, abstract. Programming gives tangible, concrete form to the act of abstraction, and repeatedly shows how useful it is. You don't have to say "we're going to learn about abstraction". You just do it, repeatedly, and then afterwards say "look at the common pattern that we have used over and over"." Simon Peyton-Jones, from "Teaching Programming for Cognitive Benefits". Teaching this key skill of computational thinking, through teaching programming, allows pupils to develop a new cognitive tool.[4]

1.3.2 Aspects of programming

Educators of University of Southampton made an analysis of the pedagogic value of programming identifies 6 important areas: ***accuracy of expression, understanding algorithms, visual representation of concepts and procedures, analysis of situations, data structures and application of logic.***

Accuracy of expression

Through computer programming we can insist upon and, importantly, demonstrate the need for accuracy and precision in what we do. However, this does not prevent creativity. Like a chess player has to obey the rules of the game, the

imaginative and thoughtful implementation of rules can create structures and procedures that are unique and valued.

At the *character level* it is akin to spelling in the English curriculum - color and colour are significantly different.

To change the colour of text to red use <font color="#FF0000"> [HTML]

But the accuracy at character level is more than just an Americanisation of spelling; it is also "paying attention to detail" and realising that spaces, punctuation marks and the case of letters are important. In many areas, computer programming is case sensitive.

At the *syntax level* the pupils become aware of the structures of instructions with operands and operators and the need to match each with the other in much the same way as there is an object-verb relationshiprogramming For example, FORWARD 10, WAIT 20, REPEAT 5, PRINT "hello world".

Structures such as IF THEN ELSE ENDIF emphasise the importance of accuracy at a level higher than the individual character.

At the *instruction level* the pupils have to be aware of structure in the same way as the grammar of an English sentence has rules of structure. All sentences begin with a capital letter and end with a full stop, exclamation or question mark. They follow the rules of grammar and punctuation. Each sentence has a subject and a verb. In the same way, computer instructions have a precise structure with common rules called syntax. For example, in many languages the end of a line of code is indicated by a semi-colon.

if (aName == bName) {System.out.println('== the same')};

At the *module, procedure or program level* the concept of wholeness is introduced. The computer program is complete within itself and usable. When pupils are asked in history to write about the reason for the rise of nationalism in the 1930s, they are expected to present their ideas as a sequence of connected statements that follow each other logically to build a sustained argument, frequently as a paragraph of text.

In the same way, pupils develop the skills of coherent thought through sequencing the instructions, beginning by setting the context, carrying out the operation and concluding in a formal manner. Writing a complete module, procedure or program is like writing a formal essay, poster, recipe or invitation. The product has a wholesomeness or completeness.

Two examples, the first is a simple program in BASIC to print out a "times table":

```
10 A=7
20 M=12
30 FOR K = 1 TO M
```

```
40 P=K*A
50 PRINT A;" times "; K;" equals "; P
60 NEXT K
70 END
```

Lines 10 and 20 set the context; 30 to 60 carry out the repeated operation to print 1 times 7 equals 7, 2 times 7 equals 14, etc. and the final line formally ends the program releasing the computer to do other things.

The second example completes the same task in Pascal:

```
a := 7;
m :=12;
for k := 1 to m do begin
p := p * a;
writeln(a, " times ", k, " equals ", p);
end;
```

"The rigorous nature of prógramming and the need for absolute precision seem to surprise a fair few of our (undergraduate) students when they encounter programming in the first few weeks of their Computing/Computer Science degrees, and some of them clearly struggle to come to terms with it. Perhaps this need for discipline and precision is something we should be promoting more strongly and at a much earlier stage in their education - but do we then risk stifling the creative element and making programming seem that much more dull and difficult?" Nick Efford "For many years there have been interactive programming development environments (including several developed by Artificial Intelligence researchers, as well as others) that support incremental development and testing, and are perfectly capable of giving sensible feedback if the programmer makes a trivial mistake, for example, and allowing the programmer (e.g. a learner) to find out what went wrong, correct it and continue - without having to start all over again. Of course, deep mistakes, where your program runs, but does the wrong thing, are another matter. Experiencing that is an important form of education. Having devices that prevent all such errors is the last thing we need in our educational environments", Aaron Sloman

"To me programming at that stage (key stage 2) is a way of exploring the virtual world. It's practical mathematics. Theory is fine, but it needs a base of experience to build on and be relevant. I still remember the thrill of writing a program that actually (eventually) did something, even if only to print 'Hello World' ", Jack Lang.

Understanding algorithms

Algorithm can be considered a sequence of instructions, a finite set of commands or a method of working.

Algorithm can be considered synonymous with program but algorithm encompasses the whole domain of carrying out instructions in a predefined and accurate way. Algorithm is to computer program as writing is to story. It is not limited to programming. However, through computer programming, pupils can gain a better understanding of the value of predefined sequences of action to more efficiently and effectively achieve an outcome.

Two initial definitions for pupils are:

1) *an algorithm can be represented as a sequence of instructions to be carried out until an end point is reached;*

2) *algorithms are the rules, conditions or sequence by which the computer or people tackle a problem or situation.*

Other keywords to be used when discussing algorithm are: steps, instructions, commands sequence, flow, decisions, branches, jumps, if then, conditional, if then else, true false repeat, until, condition, iteration. Algorithm takes many forms. It can be the rules by which you drive a car. It can be the way in which you eat from a buffet. It can be the way in which you carry out a science procedure.

"You are approaching a traffic queue: which lane do you take? Always going to the shortest line is a greedy algorithm. You might consider the shortest queue but always err to the right because you think that the fastest drivers are there. That algorithm is more strategic or refined. You may rely upon local knowledge of the road and queues and make different decisions in different places. The algorithm is very specialised and contains or makes reference to information. In a similar way, we program the computers to obey a set of predetermined rules - the algorithm." John Woollard

Analysis of situations

The act of creating a computer program usually requires a deeper and more rigorous analysis of the context of the program than a pupil might otherwise undertake, for example, the sequence of traffic lights or the input/output requirements of a heating system or the data requirements of a video shoprogramming The use of techniques such as systems analysis gives pupils an insight into the ways in which precision can be obtained. The analysis of system diagrams help pupils to understand how complex systems work. Those systems

may be anything from biological population, mechanical devices, businesses, through to the impact of social policy and regulations. The synthesis of system diagrams for familiar situations helps the pupils understand the detail and the complexity of systems – for instance, the school as a system. At the highest level (simplest) view of a system it is similar to a block diagram, showing its inputs, outputs and processes.

System diagrams and data flow are particularly helpful in showing how a change in one factor may impact elsewhere. Importantly, a good diagram might show how changing a factor may feed back to affect itself!

Drawing a system or data flow diagram is a good way of starting to build a computer model. The technique helps you to map out the structure of the system to be modelled. It shows the factors and relationships that are important, and helps you to start quantifying the linkages between factors.

Visual representation of concepts and procedures

The skills of visual representation of pupils' understanding are acknowledged in many areas of the curriculum. In English there is the story board, in mathematics is the chart, in science the symbolic representation of atomic and subatomic particles, in geography is the map at different levels of scale and symbolism and in humanities are the icons of religion, consumerism and politics. In computing there are visual representations of what the computer is doing when running a program: flow diagram, variables table and structured English instructions. Each aids the computer programmer but also, each is a tool that learners can apply in other areas of study to help them more efficiently and effectively represent their knowledge and understanding.

Data structures

In computing, a data structure is a particular way of storing and organising information so that it can be accessed efficiently. Different kinds of data structures are suited to different kinds of applications and different types of data. Three examples that pupils will be familiar with are: classification keys, indexes and labels. In science, classification keys in biology or chemicals are often based on a tree structure. Paper-based shopping catalogues use indexing and sequencing of the information. Car registration number plates represent the systematic labelling of items. Pupils can learn to interpret (decode) a number plate and begin to understand the process of labelling for computerised systems such as the bar codes on goods for sale, the structure of URLs or the unique identifiers such as National Insurance numbers and the use of check digits.

Some data structures are specifically designed for efficient computer processing, for example, B-trees are particularly well-suited for implementation of

databases, while compiler implementations usually use hash tables to look up identifiers. Data structures are used in almost every program or software system. Specific data structures are essential ingredients of many efficient algorithms, and make possible the management of huge amounts of data, such as large databases and internet indexing services. Some formal design methods and programming languages emphasize data structures, rather than algorithms, as the key organizing factor in software design.

Application of logic

Logic as a topic encompasses the understanding and application of the Boolean algebra AND, OR, NOT truth values in keyword searching, electronic circuits, logic circuits and truth tables. This rigorous application of rules again supports the need for accuracy of expression and clear thinking around novel concepts. Logic is the application of methods and criteria of validity of inference, reasoning and knowledge. The following are examples of the application of logic.

Boolean logic is used in search engines to retrieve items on the web, for example, when searching in Google, if you enter words adjacent to each other, say monty python, the interface automatically inserts an AND operator between the words and returns documents or items which contain both of these words, not necessarily adjacent to each other. The search for monty OR python would give many more hits whereas the search for "monty python" would give much fewer hits.

As an alternative analysis, Roger Broadie observes, "I believe ICT (computing) has a unique contribution to offer in that it brings ways of thinking that are not provided for or by any other areas of the curriculum...

These ways of thinking certainly include (and there may be others):

- *Programming*; analysis of processes in order to produce complete and correct programs that will provide the desired result. This is an important skill whether or not ICT is involved;
- *Interface design*; this is about human interactions with information and other humans, and the ways in which information and the development of the interaction are presented to stimulate and guide the interaction. Other subjects touch on this but none with the depth and effectiveness that ICT can;
- *Information structuring*; this includes hierarchical, relational and hyper-linked structures, and while some of these are covered in say science with biological keys, relational and hyper info structures can only be satisfactorily worked on with ICT;

- *Networked communication*; even simple examples such as how to use the asynchronous nature of email effectively are hardly covered in the English subject curriculum, and where social networking is taking us most certainly is not;
- *Language structure and semantics*; while this is shared with human language studies when they look at grammar, there is a broad range of semantic structures used in programming languages that human language does not use;
- *Data*; including coding of data, data redundancy and issues around compression of data;
- *Search and information validation.*

While some of these might be beyond the school curriculum except for the brightest who specialise in ICT, I would maintain that the first four at least should be studied to some level by all, as they are vital to how life and work will operate this century. And ICT as a separate subject is the way to do this."

1.3.3 Programming as part of general education

Why should students be taught programming? After all, for just about every conceivable application, ready-made software packages provide tools much more powerful than what almost any user could write – and we are all users of software such as word processors, spreadsheets, search engines etc. If computer users no longer program, does it follow that the art of programming should only be taught to computing professionals? If school was only about the productivity of future office-application users, one might agree with this view. Instead of teaching programming, teachers might concentrate on how to use "Word" more proficiently.

But this point of view, introducing computer science as a set of application specific skills, has a minor and a major flaw. The minor flaw is that most introductions to office-application packages concentrate only on short-lived details of the package being used, making any transfer of knowledge to a different package difficult. Concepts which might be useful across applications, such as how information is encoded and structured, are often neglected, even though they facilitate mastery of new software.

The major flaw of a computer education focused on skills only is harder to explain.

Let us consider an analogy with mathematics teaching. Many professions in science and engineering require the use of mathematical results. Scientists and engineers are users of mathematics in the same sense that many people are users of computers: they check the preconditions (input) of a theorem or formula and derive

the conclusions (output). Thus, one might argue that scientists and engineers only need to learn how mathematical theorems are applied; that the concept of „proof" is only relevant to professional mathematicians. But centuries of experience show that any math instruction involves significant time devoted to proofs, even though most dents are unlikely to ever prove a theorem outside their math courses. The reason is plain: we do not trust a „mathematics user" to apply formulas or mathematical software in a reliable and responsible manner if he has never understood the concept of "proof". Applying mathematical results is a matter of understanding, not just of pattern matching. Mathematics users do not need to know the proof of every formula they use; they need to understand mathematical thinking, based on the concepts of theorem and proof.

Similarly, applying computers should be a matter of understanding, not just of pushing the right keys. Computer users do not need to know the source code of every application they use, but they should have an intuition of what constitutes a program.

This calls for personal experience with writing, testing and debugging a number of small programs. In the days of ready-made application software, we do not need programming as a tool, but rather as background knowledge that helps us understand what computers can and cannot do. A similar statement can be made for any kind of general education. General education is rarely applied for immediate use, but helps us put details of transient importance into a larger perspective.

How to introduce students to the fundamental concepts of programming? If a teacher wants to teach programming, what language should he use? C++, Java, Delphi?

These languages are made for the professional programmer; they are powerful and complex. They are also object-oriented, posing an additional challenge for the teacher: whereas object-orientation is relevant to "programming in the large", it is a non-trivial additional hurdle for a beginner. And last but not least, programming environments with their project-management and debugging facilities are complicated to handle and not suited for an easy introduction to programming. There is no need to introduce beginners to the complexities inherent in professional programming languages and environments, where you need a manual just to display "hello world". Programming practiced as an educational exercise, free from utilitarian constraints, is best learned in a toy environment, designed to illustrate selected concepts in the simplest possible setting. The fundamental concepts of programming may be intellectually demanding, but they are not complex in the sense of requiring mastery of lots of details. The main ideas can be conveyed with a few selected, simple examples. [4]

2 SCRATCH PROGRAMMING ENVIRONMENT

2.1 What is Scratch?

Scratch is developed by the Lifelong Kindergarten group at the MIT Media Lab. Here is how Scratch website describes Scratch:

"Scratch is designed to help young people (ages 8 and up) develop 21st century learning skills. As they create Scratch projects, young people learn important mathematical and computational ideas, while also gaining a deeper understanding of the process of design. With Scratch, kids can create their own interactive stories, games, music, and animation and share their creations with one another on the Web." [5]

Scratch is a programming environment, based upon LOGO, which allows young programmers to easily create multimedia applications and simple games. Block-programming eliminates the frustrations of syntax errors which afflict novices learning traditional computer programming languages. The Scratch environment allows the creation and control of graphical objects, known as sprites, which move around the screen and integration of graphics or photos and sound files to create multimedia applications.

Scratch is easily learnt by an adult in a couple of hours and children from the age of seven can be productive from the very first lesson. A typical lesson plan might require six sessions for a young programmer to learn the main programming concepts and begin creating their own games or multimedia projects. Scratch applications are designed to be shared and the Scratch web site has thousands of applications which can be downloaded and remixed into new applications. Some applications that can be created by novice Scratch programmers are detailed below:

- A simple game where the player controls a sprite chasing another sprite around the screen.
- An animation with sprites representing the letters of the programmer's name with each letter spinning or otherwise moving around the screen and playing a sound when clicked.
- A quiz style application showing several pictures and a question which responds to the user clicking on the selected answer.
- A multimedia presentation incorporating photographs with sound recordings providing explanations as the user clicks on each item.

Scratch programming involves several IT skills and creation of a typical application might involve:

- Drawing a sprite within the Scratch application;
- Importing a background picture from a photograph or a file created in another graphics application;
- Recording a sound file or incorporating an MP3 file into the application;
- Adding text with different fonts to an application;
- Moving sprites around the screen with turtle graphics commands;
- Manipulation and digital effects on graphics objects.

Scratch programming also involves an understanding of some of the principles of computer programming:

- Creating a sequence of commands.
- Using an IF conditional statement to determine a course of action.
- Repeating a sequence of commands.
- Moving a sprite around the screen and responding to the environment.
- Responding to keyboard and other events.
- Using variables in an application.
- Using messages and broadcasts to control the application.

One of the main advantages of this system is the availability of versions for different operating systems: Windows, Linux and Mac OS.

2.2 Scratch as an educational platform

Scratch was originally created as an educational platform, based on the constructionist ideas behind Logo, where students create knowledge themselves, rather than just absorbing it from teachers. Maloney give a detailed summary of the Scratch environment. The "wide walls, high ceiling and low floor" principle of making an environment easy to approach, of being capable of allowing diverse projects and of accepting projects with increasing complexity is described by Resnick.

Studies have found various results using Scratch to teach core computing concepts. Colleen Lewis reports on a study comparing S. and Logo for teaching programming to 6th graders. Students spent a total of 36 hours over 12 days learning programming constructs shared by Scratch and Logo. She studied both their learning outcomes and attitudes toward computing, but found little difference

of statistical significance between the two platforms. One area where students using Scratch outperformed students in the Logo group was in comprehending conditional statements. Surprisingly, however, students in the Scratch group did not show greater comprehension of loops.

In "Habits of Programming in Scratch," Meerbaum-Salant found that while Scratch encourages self-directed learning, students only really learned programming concepts when explicitly taught them. The habits they found in novice programmers learning with Scratch include bottom-up programming and extremely fine-grained programming. The students took the bottom-up approach to the extreme, starting with the small components to build up to top-level system and not approaching the system from an algorithmic standpoint. With extremely fine-grained programming, the students took functional decomposition to the extreme as well, breaking scripts up more than necessary and accidentally introducing concurrency bugs.

Rivzi describe a new S.-based undergraduate course (CS0) inserted before the traditional first programming course (CS1) and aimed at student retention. One set of students enrolled directly in CS1, while the others enrolled first in CS0. Amongst the CS0 set, the researchers found increased interest in computer science as well as improved learning outcomes.

A study of Scratch use by younger students (ages 8 – 18) at a Computer Clubhouse is described by Maloney. The researchers studied over 530 projects created in S. by youth at the center. Rather than explicitly teaching participants programming concepts, researchers allowed the youth to work on projects of their own choosing. A significant number of projects used loops (51.8%) or conditional statements (26.1%), but fewer used variables (9.6%).

Another study investigated learning results of computer science concepts by students learning in the S. environment . A Scratch-based curriculum was developed for middle-school-aged children. Middle school teachers taught the course during regular school hours. Pre, interim and post tests were used to test students' comprehension of concepts including loops, variables and lists. Students performed well with loop concepts, but less so with variables.[6]

2.2.1 Creating with Scratch

People have access to an incredible variety of interactive games, stories, animations, simulations, and other types of dynamic, interactive media on their computers today.

But, for the most part, these programs are a one-way street: you can only browse and click what others have created; you can't design and create your own.

Scratch changes that, broadening the range of what you can design and create on the computer, making it easier to combine graphics, photos, music, and sound into interactive creations. With Scratch, you can create characters that dance, sing, and interact with one another. Or create images that whirl, spin, and animate in response to movements of the mouse. Or integrate images with sound effects and music clips to create an interactive birthday card for a friend, or an interactive report for school.

The name Scratch comes from the Scratching technique used by hip-hop disc jockeys, who spin vinyl records back and forth with their hands to mix music clips together in creative ways. We can do something similar with Scratch, mixing different types of media clips (graphics, photos, music, sounds) in creative ways.

At the core of Scratch is a graphical programming language that lets you control the actions and interactions among different media. Coding in Scratch is much easier than in traditional programming languages: to create a script, you simply snap together graphical blocks, much like LEGO bricks or puzzle pieces.

Once you've created a Scratch project, you can share it on the S. website, the same way you might share videos on YouTube or photos on Flickr. Or you can embed your Scratch project in any other webpage – for example, embedding an interactive Scratch animation on your MySpace or Facebook homepage.

You can get new ideas for S. projects by browsing through projects on the Scratch website. If you like one of the characters or images or scripts in another project, simply download the project and use parts of it in your own Scratch project.[7]

2.2.2 Learning with Scratch

What do students learn as they create interactive stories, animations, games, music, and art with Scratch?

For one thing, they learn mathematical and computational ideas that are built into the Scratch experience. As students create programs in Scratch, they learn core computational concepts such as iteration and conditionals. They also gain an understanding of important mathematical concepts such as coordinates, variables, and random numbers.

Significantly, students learn these concepts in a meaningful and motivating context. When students learn about variables in traditional algebra classes, they usually feel little personal connection to the concept. But when they learn about variables in the context of Scratch, they can use variables immediately in very meaningful ways: to control the speed of an animation, or to keep track of the score in a game they are creating.

As students work on Scratch projects, they also learn about the process of design. Typically, a student will start with an idea, create a working prototype, experiment with it, debug it when things go wrong, get feedback from others, then revise and redesign it. It's a continuous spiral: get an idea, create a project, which leads to new ideas, which lead to new projects, and on and on. This project-design process combines many of the 21st century learning skills that will be critical to success in the future: thinking creatively, communicating clearly, analyzing systematically, collaborating effectively, designing iteratively, learning continuously.

Creating projects in Scratch also helps students develop a deeper level of fluency with digital technology. What do we mean by fluency? To be considered fluent in English, Spanish, or other language, you must learn not only how to read but also to write – that is, how to express yourself with the language. Similarly, to be fluent with digital technology, you must learn not only how to interact with the computer but also to create with it.

Of course, most students will not grow up to become professional programmers, just as most will not become professional writers. But learning to program offers benefits for everyone: it enables students to express themselves more fully and creatively, helps them develop as logical thinkers, and helps them understand the workings of the new technologies that they encounter everywhere in their everyday lives.[8]

2.2.3 Learning by designing

When people think about learning and education, they often think about one person transmitting information to another, like this: teacher comes to the class and tries to explain lesson to everyone, he talks, talks, talks again and again. And he wants all of the pupils to understand what he says. But sometimes teachers don't attach importance to that pupils differs by their abilities to understand.

Increasingly, educators are recognizing that this "transmission approach" doesn't work very well. Research has shown that people learn best not when they are passively receiving information, but when they are actively engaged in exploring, experimenting, and expressing themselves (sometimes known as the 3 X's).

More and more schools are focusing on learning-by-doing, engaging students in hands-on activities. But founders of Scratch want to go a step further. Their goal is to help students learn to design, create, and invent things. It's not just learning-by-doing; it's learning-by-designing.

Why Design?

- Design projects engage kids as active participants, giving them a greater sense of control and responsibility for the learning process;
- Design projects encourage creative problem-solving.;
- Design projects are often interdisciplinary, bringing together ideas from art, technology, math, and sciences;
- Design projects help kids learn to put themselves in the minds of others, since they need to consider how others will use the things they create;
- Design projects provide opportunities for reflection and collaboration;
- Design projects set up a positive-feedback loop of learning: when kids design things, they get new ideas, leading them to design new things, from which they get even more ideas, leading them to design yet more things, and so on.[7]

2.2.4 Constructivism and constructionism

The learning-by-designing approach is inspired by two important theories of learning and education.

The constructivist theory of learning, developed by Swiss psychologist Jean Piaget, views learning as a very active process in which people continually construct new knowledge from their experiences in the world.

According to this theory, people don't get ideas, they make them. Constructivist theory is the underpinning for many educational reform initiatives.

The constructionist approach to education, developed by MIT professor Seymour Papert, is based on two types of construction: it argues that people construct new knowledge especially well when they are engaged in constructing things in the world. They might be constructing sand castles, LEGO machines, or computer programs. What's important is that they are actively engaged in creating something meaningful to themselves or others around them.[9]

2.2.5 XXI century learning skills

Scratch supports the development of 21st Century Learning Skills , as described by the Partnership for the 21st Century7.The report Learning for the 21st Century identifies nine types of learning skills, divided into three key areas. This handout highlights the ways Scratch supports the development of these 21st Century learning skills.

1) Information & Communication Skills:

– *Information and Media Literacy Skills:*

By working on Scratch projects, students learn to select, create, and manage multiple forms of media, including text, images, animation, and audio recordings. As students gain experience creating with media, they become more perceptive and critical in analyzing the media they see in the world around them.

– *Communication Skills:*

Effective communication in today's world requires more than the ability to read and write text. Scratch engages young people in choosing, manipulating, and integrating a variety of media in order to express themselves creatively and persuasively.

2) Thinking and Problem-Solving Skills:

– *Critical Thinking and Systems Thinking:*

As they learn to program in Scratch, young people become engaged in critical reasoning and systems thinking. In order to build projects, students need to coordinate the timing and interactions between multiple "sprites" (programmable moving objects). The ability to program interactive input provides students direct experience with sensing, feedback, and other fundamental systems concepts.

– *Problem Identification, Formulation & Solution:*

Scratch supports problem finding and solving in a meaningful design context. Creating a Scratch project requires thinking of an idea, then figuring out how to break the problem into steps and implement them using Scratch programming blocks. Scratch is designed to be "tinkerable": students can dynamically change pieces of code and immediately see the results (e.g., doubling a number to see how it changes a graphic effect). Throughout the design process, students engage in experimenting and iterative problem-solving.

– *Creativity and Intellectual Curiosity:*

Scratch encourages creative thinking, an increasingly important skill in today's rapidly changing world. Scratch involves young people in seeking innovative solutions to unexpected problems—not just learning how to solve a predefined problem, but being prepared to come up with new solutions as new challenges arise.

3) Interpersonal & Self-Directional Skills:

– *Interpersonal and Collaborative Skills:*

Because Scratch programs are built of graphical blocks, the programming code is more readable and shareable than other programming languages. The visual objects

and modular code supports collaboration, enabling students to work together on projects and exchange objects and code.

- *Self-Direction:*

Taking an idea and figuring out how to program it in Scratch requires persistence and practice. When young people work on project ideas they find personally meaningful, their ideas provide internal motivation for overcoming challenges and frustrations encountered in the design and problem-solving process.

- *Accountability and Adaptability:*

When students create Scratch projects, they have an audience in mind, and need to think about how other people will react and respond to their projects. Since Scratch projects are easy to change and revise, students can modify their projects based on feedback from others.

- *Social Responsibility:*

Because Scratch programs are shareable, students can use Scratch to provoke discussion of important issues with other members of their immediate learning environment, as well as with the wider international Scratch community.[10]

2.3 Three core design principles for Scratch

Three core design principles for Scratch are: make it more tinkerable, more meaningful, and more social than other programming environments.

2.3.1 Tinkerability

Lifelong Kindergarten research group at the MIT Media Lab (http:// llk.media.mit.edu) has worked closely with the Lego Company (http://www.lego.com/) for many years, helping develop Lego Mindstorms and other robotics kits. They have always been intrigued and inspired by the way children play and build with Lego bricks.

Given a box full of them, they immediately start tinkering, snapping together a few bricks, and the emerging structure then gives them new ideas. As they play and build, plans and goals evolve organically, along with the structures and stories.

They wanted the process of programming in Scratch to have a similar feel.

The S. grammar is based on a collection of graphical "programming blocks" children snap together to create programs. As with Lego bricks, connectors on the blocks suggest how they should be put together. Children can start by simply tinkering with the bricks, snapping them together in different sequences and combinations to see what happens. There is none of the obscure syntax or punctuation of traditional programming languages. The floor is low and the experience playful.

Scratch blocks are shaped to fit together only in ways that make syntactic sense. Control structures (like forever and repeat) are C-shaped to suggest that blocks should be placed inside them. Blocks that output values are shaped according to the types of values they return: ovals for numbers and hexagons for Booleans. Conditional blocks (like if and repeat-until) have hexagon-shaped voids, indicating a Boolean is required. The name "Scratch" itself highlights the idea of tinkering, as it comes from the S.ing technique used by hip-hop disc jockeys, who tinker with music by spinning vinyl records back and forth with their hands, mixing music clips together in creative ways. In Scratch programming, the activity is similar, mixing graphics, animations, photos, music, and sound.

Scratch is designed to be highly interactive. Just click on a stack of blocks and it starts to execute its code immediately. You can even make changes to a stack as it is running, so it is easy to experiment with new ideas incrementally and iteratively. Want to create parallel threads? Simply create multiple stacks of blocks. Founder's goal is to make parallel execution as intuitive as sequential execution.

The scripting area in the Scratch interface is intended to be used like a physical desktoprogramming You can even leave extra blocks or stacks lying around in case you need them later. The implied message is that it's OK to be a little messy and experimental.

Most programming languages (and computer science courses) privilege top-down planning over bottom-up tinkering. With S., we want tinkerers to feel just as comfortable as planners.

The emphasis on iterative, incremental design is aligned with their own development style in creating S..

Scratch creaters selected Squeak as an implementation language since it is well-suited forrapid prototyping and iterative design. [11]

2.3.2 Meaningfulness

We know that people learn best, and enjoy most, when working on personally meaningful projects. So in developing Scratch, it's founders put a high priority on two design criteria:

- *Diversity*. Supporting many different types of projects (stories, games, animations, simulations), so people withwidely varying interests are all able to work on projects they care about;
- *Personalization*. Making it easy for people to personalize their Scratch projects by importing photos and music clips, recording voices, and creating graphics.

These priorities influenced many of the design decisions. For example, Scratch designers decided to focus on 2D images, rather than 3D, since it is much easier for people to create, import, and personalize 2D artwork. While some people might see the 2D style of S. projects as somewhat outdated, Scratch projects collectively exhibit a visual diversity and personalization missing from 3D authoring environments. The value of personalization is captured nicely in this blog post from a computer scientist who introduced Scratch to his two children: "I have to admit that I initially didn't get why a kids' programming language should be so media-centric, but after seeing my kids interact with Scratch it became much more clear to me. One of the nicest things I saw with Scratch was that it personalized the development experience in new ways by making it easy for my kids to add personalized content and actively participate in the development process. Not only could they develop abstract programs to do mindless things with a cat or a box, etc... but they could add their own pictures and their own voices to the S. environment, which has given them hours of fun and driven them to learn."

It's so amazing that the diversity of projects appear on the Scratch web site every day. As expected, there are lots of games, ranging from painstakingly recreated versions of favorite video games (such as Donkey Kong) to totally original games. But there are many other genres, too. Some Scratch projects document life experiences (such as a family vacation in Florida); others document imaginary wished-for experiences (such as a trip to meet other Scratchers). Some Scratch projects (such as birthday cards and messages of appreciation) are intended to cultivate relationships. Others are designed to raise awareness on social issues (such as global warming and animal abuse).

Some Scratch projects grow out of school activities. For an Earth-science class, a 13-year-old boy from India created a project in which an animated character travels to the center of the Earth, with a voice-over describing the different layers along the way. As part of social-studies class, a 14-year-old boy from New Jersey created a simulation of life on the island of Rapa Nui, designed to help others learn about the local culture and economy.

2.3.3 Sociality

Development of the Scratch programming language is tightly coupled with development of the Scratch Web site.

For Scratch to succeed, the language needs to be linked to a community where people can support, collaborate, and critique one another and build on one another's work. The concept of sharing is built into the Scratch user interface, with a prominent "Share" menu and icon at the top of the screen. Click the Share icon

and your project is uploaded to the Scratch Web site where it is displayed at the top of the page, along with the "Newest Projects." Once a project is on the Web site, anyone can run it in a browser (using a Java-based player), comment on it, vote for it (by clicking the "Love It?" button), or download it to view and revise the scripts. (All projects shared on the site are covered by Creative Commons license.)

In the 27 months following the S. launch, more than 500,000 projects were shared on the Scratch Web site.

For many Scratchers, the opportunity to put their projects in front of a large audience—and receive feedback and advice from other Scratchers—is strong motivation. The large library of projects on the site also serves as inspiration. By exploring projects there, Scratchers get ideas for new projects and learn new programming techniques. Marvin Minsky once said that Logo had a great grammar but not much literature. Whereas young writers are often inspired by reading great works of literature, there was no analogous library of great Logo projects to inspire young programmers. The Scratch Web site is the beginning of a "literature" for Scratch. The site is also fertile ground for collaboration. Community members are constantly borrowing, adapting, and building on one another's ideas, images, and programs. Over 15% of the projects there are remixes of other projects on the site. For example, there are dozens of versions of the game Tetris, as Scratch ers continue to add new features and try to improve gameplay. There are also dozens of dress-up-doll projects, petitions, and contests, all adapted from previous Scratch projects. At first, some Scratchers were upset when their projects were remixed, complaining that others were "stealing" from them. That led to discussions on the Web site's forums about the value of sharing and the ideas behind open source communities. The goal is to create a culture in which Scratchers feel proud, not upset, when their projects are adapted and remixed by others. Scratch founders have continually added new features to the site to support and encourage this mind-set. Now, when someone remixes a project, the site automatically adds a link back to the original project, so the original author gets credit. Also, each project includes links to its "derivatives" (projects remixed from it), and the "Top Remixed" projects are featured prominently on the S. homepage.

Some projects focus on the site itself, providing reviews and analyses of other projects there. One early example was called SNN, for Scratch News Network, featuring the Scratch cat (the default character in S.) delivering news about the Scratch community, much like a CNN anchor. At first, we saw it as a "simulated newscast" but then realized it was a real newscast, providing news of interest to a real community—the Scratch online community. The SNN project inspired others, leading to a proliferation of online newsletters, magazines, and TV shows, all programmed in Scratch, reporting on the Scratch community.

Other Scratchers formed online "companies," working together to create projects that their individual members could not have produced on their own. One company got its start when a 15-year-old girl from England, with screen name BeeBop, created a project full of animated sprites and encouraged others to use them in their projects or place special requests for custom-made sprites. She was setting up a no-fee consulting business. A 10-year-old girl, also from England, with screen name MusicalMoon, liked BeeBop's animations and asked if she'd be willing to create a background for one of her projects.

This collaboration gave rise to Mesh Incorporation, a self-proclaimed "miniature company" to produce "top quality games" in Scratch. A few days later, a 14-yearold boy from New Jersey, screen name Hobbit, discovered the Mesh Incorporation. gallery and offered his services, saying, "I'm a fairly good programmer, and I could help with debugging and stuff."

Later, an 11-year-old boy from Ireland, with screen name Marty, was added to the Mesh Inc. staff due to his expertise in scrolling backgrounds.

Such collaborations open opportunities for many different types of learning.

To encourage international sharing and collaboration, there placed a high riority on translating Scratch into multiple languages. Scratch designers created an infrastructure that allows the Scratch programming blocks to be translated into any language with any character set. A global network of volunteers has provided translations for more than 40 languages.

Children around the world now share Scratch projects with one another, each viewing the S. programming blocks in their own language.[12]

2.4 Scratch programming

With traditional computer and Internet applications, users are limited to working with applications in the way the programmers who developed the applications designed. Scratch turns things around by letting users become programmers. Scratch is designed to meet the needs of young people between 8 and 16, helping to introduce them to computer technology and to improve their learning skills while at the same time facilitating creativity and personal expression.

Many people regard computer programming as a mysterious and complex process that requires advanced technical training and education. This is a misperception.

Programming languages like BASIC have been around for decades and were developed expressly for the purpose of teaching first-time programmers how to program. In recent years, a new crop of programming languages has appeared,

specifically geared towards helping children and students learn to program. One of the very best and newest of these languages is Scratch.

Scratch is a visual programming language that is made up of a graphic interface that supports application development in which new projects are created by mixing together images, sound, and video under the control of scripts, which specify the application's programming logic. Scripts are created by snapping blocks together, much in the same way that Lego blocks are snapped together to create all sorts of unique creations. Each block represents a different command or action that tells the application how to execute.

Scratch also provides programmers with access to all kinds of media, including graphics and sounds as well as tools that can be used to create new graphics and sound files.

Scratch is an interpreted programming language. This means that application projects are not precompiled (turned into executable code that can be run as a stand-alone application) before their execution. Instead, the code blocks that make up Scratch application projects are interpreted and processed each time the application project is executed. Scratch is also a dynamic programming language. It allows changes to be made to application projects even while the projects are executing. As such, Scratch lets programmers experiment by making application changes on the fly in order to see what type of effect the changes may have on the application's execution.

The S. application is used to create projects containing media and scripts. Images and sounds can be imported or created in Scratch using a built-in paint tool and sound recorder. Programming is done by snapping together colorful command blocks to control 2-D graphical objects called sprites moving on a background called the stage. S. projects can be saved to the file system or shared on the Scratch Web site.

The original design of Scratch was motivated by the needs and interests of young people (ages 8 to 16) at after-school computer centers such as the Intel Computer Clubhouses. Scratch added programmability to media-manipulation activities that are popular in youth culture, and it encouraged young people to learn through exploration and peer sharing, with less focus on direct instruction than other programming languages. Initially, Scratch was used primarily in informal learning settings such as community centers, after-school clubs, libraries, and homes, but increasingly it is used in schools as well.

The Scratch project began in 2003, and the S. software and Web site were publicly launched in 2007. Scratch is free, available in nearly 50 languages, and more than two million copies have been downloaded from the Scratch Web site. In addition, Scratch software is often redistributed by school systems and educational organizations. For example, Scratch is distributed by One Laptop Per Child and

has been shipped on hundreds of thousands of XO laptop computers. Since the launch, more than a million Scratch projects have been uploaded to the Web site by more than 120,000 users. Roughly 1500 new projects are uploaded to the Web site every day—on average, more than one new project every minute.

Scratch builds on the constructionist ideas of Logo (Kafai and Resnick 1996; Papert 1980) and Etoys (Kay 2010; Steinmetz 2002). To help users make their projects personally engaging, motivating, and meaningful, Scratch makes it easy to import or create many kinds of media (images, sounds, music). The Scratch Web site provides a social context for Scratch users, allowing users to share their Scratch projects, receive feedback and encouragement from their peers, and learn from the projects of others.

A key goal of Scratch is to introduce programming to those with no previous programming experience. This goal drove many aspects of the Scratch design. Some of the design decisions are obvious, such as the choice of a visual blocks language, the single-window user interface layout, and the minimal command set. Others are less obvious, such as how the target audience influenced the type system and the approach to error handling. This article explores aspects of the Scratch programming environment and language design that make it easier for young people to explore, express themselves, and learn.

Scratch's slogan is Imagine—Program—Share! It is designed to encourage teens' creativity by providing them with an easy to learn yet powerful programming environment in which they can unleash the power of their imagination. Scratch encourages and facilitates the development of application projects using a mixture of media, graphics, sound, and video in order to create something new.

Scratch provides new programmers with everything needed to create and execute new application projects. Its programming language is designed to make it as easy as possible for new programmers to jump in and get their feet wet and to receive immediate feedback on their progress. Scratch promotes an understanding of programming concepts, including conditional and iterative logic, event programming, the use of variables, mathematics, and the use of graphics, and sound effects. By learning to program with Scratch, new programmers develop an understanding and appreciation of the design process, from idea generation to program development, then testing and debugging and the incorporation of user feedback.

People, especially kids, love to share, as demonstrated through the amazing success of websites like YouTube, which allows people to share home video.

Sharing is a fundamental part of the Scratch programming experience. Scratch application projects can not only be run on the programmer's desktop but can also be uploaded to the Scratch website, where they can be viewed, executed

online, and commented on by other Scratch programmers from around the world. By posting their Scratch application projects on the Scratch website, kids share their experiences and learn from one another and gain gratification and confidence from the experience.[13]

Most computer languages use text only to communicate commands. The text can be very hard to initially comprehend because it looks something like this (Java):

```
public class Helloworld
{
  public static void main(String args[])
{
        System.out.println("Hello world!");
  }
}
```

Whereas Scratch will produce something much similar, much easier as Figure 1 shows:

Figure 1 simple program on Scratch

To repeat code in Java you would need to write something like this:

```
for(int i = 0; i < 10; i++)
{
  commands
}
```

Whereas Scratch will produce something much similar, much easier as figure 2 shows:

Figure 2 Looping in Scratch

All the program above does, when executed, is display *"hello, world!"* on the user's screen. You might have guessed as much just by looking over the code—and ignoring anything that didn't make sense! But what's with all the curly braces ({ and })? What's *System.out*? What does *class Hello* mean? And *public static void main(String [] args)*? Let's not even go there. Suffice it to say that, when it comes

to learning to program, there's quite a learning curve with languages like Java. Before you can begin to solve problems, you must first learn to read and write a new language, even if the task at hand is relatively simple (*e.g.*, "hello, world!"). And whereas you might still understand a foreigner who mispronounces some English word, computers aren't so forgiving when it comes to mistakes. Leave out a semicolon, and the program above won't even work!

Learning to program is ultimately about learning to think logically and to approach problems methodically. The building blocks out of which a programmer constructs solutions, meanwhile, are relatively simple. Common in programming, for instance, are "loops" (whereby a program does something multiple times) and "conditions" (whereby a program only does something under certain circumstances. Also common are "variables" (so that a program, like a mathematician, can remember certain values).

For many students, the seemingly cryptic syntax of languages like Java tends to get in the way of mastery of such relatively simple constructs as these. Before we tackle a language like Java, then, with its curly braces and semicolons, we turn our attention to Scratch, which is just as useful (and fun) for budding computer scientists. By representing programs' building blocks with color-coded blocks (*i.e.*, puzzle pieces), Scratch "lowers the bar" to programming, allowing budding computer scientists to focus on problems rather than syntax, to master programmatic constructs rather than syntax. Syntax, of course, will come later. But, for now, we focus on programming itself. It just so happens that programming, for now, will be more like putting together a puzzle.

As we know most people view computer programming as a tedious, specialized activity, accessible only to those with advanced technical training. And, indeed, traditional programming languages like Java and C++ are very difficult for most people to learn.

In text-based programming languages, code statements are formulated by following a complex set of syntax rules. Failure to precisely follow these rules when writing statements leads to syntax errors that prevent applications from running.

Scratch, on the other hand, uses a different approach. Scratch application projects are built by selecting and snapping together graphical programming blocks, as demonstrated in figure 3.

By using code blocks in place of complex program text statements, Scratch significantly simplifies application development while still making use of the same basic programming logic and concepts implemented in other programming languages. As figure 3 demonstrates, each code block represents a different command or action. Blocks fit together like pieces in a puzzle. You can only snap

together blocks in ways that make syntactic sense, completely eliminating syntax errors that proliferate in other programming languages.

Some code blocks are configurable, allowing you to specify things like the number of times an action should execute, text that is to be displayed, or the color to be used when displaying something on the screen. Despite its use of graphical code blocks, Scratch supports the same basic set of programming techniques and constructs as do other traditional programming languages. For example, Scratch supports variables, conditional and iterative logic, and eventdriven programming. Scratch also supports the manipulation of graphics and the integration of sound into application projects.

Figure 3 Example of simple Scratch program

Scratch aims to change that. Scratch takes advantage of advances in computing power and interface design to make programming more engaging and accessible for children, teens, and others who are learning to program. Key features of Scratch include:

- ***Building-block programming.*** To create programs in Scratch, you simply snap graphical blocks together into stacks. As figure 4 shows the blocks are designed to fit together only in ways that make syntactic sense, so there are no syntax errors. Different data types have different shapes, eliminating type mismatches. You can make changes to stacks even as programs are running, so it is easy to experiment with new ideas incrementally and iteratively.

Figure 4 Building-block programming example

- ***Media manipulation.*** With Scratch, you can create programs that control and mix graphics, animations, music, and sound. Scratch extends the media-manipulation activities that are popular in today's culture – for example, adding programmability to Photoshop-style image filtering. You can see an example of media manipulation on figure 5 given below:

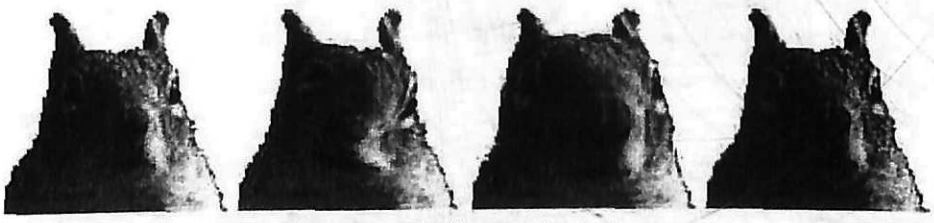


Figure 5 Media manipulation

- ***Sharing and collaboration***. The Scratch website provides inspiration and audience: you can try out other people's projects, reuse and adapt their images and scripts, and post your own projects. The ultimate goal is to develop a shared community and culture around Scratch. In the figure 6 you can see the screenshot of official site of Scratch where Scratch ers share with their projects.

Scratch offers a low floor (easy to get started), high ceiling (ability to create complex projects), and wide walls (support for a wide diversity of projects). In developing Scratch, we put high priority on simplicity, sometimes even sacrificing functionality for understandability.

As students work on Scratch projects, they have opportunities to learn important computational concepts such as iteration, conditionals, variables, data types, events, and processes. Scratch has been used to introduce these concepts to students of many different ages, from elementary school through college. Some students transition to traditional text-based languages after getting introduced to programming with Scratch.

Scratch is built on top of the Squeak programming language. It was inspired by previous work on Logo and Squeak Etoys, but it aims to be simpler and more intuitive.

Scratch is an open-source but closed development project. The source code is freely available, but the application is developed by a small team of researchers at the MIT Media Lab.[6]

As a beginner's programming language, Scratch teaches children a number of critical programming concepts that they will be able to rely on should they decide to make the jump to other more traditional and industrial-strength programming languages like Microsoft Visual Basic, Cþþ, JavaScript, and AppleScript.

Figure 6 S. official site screenshot

The programming concepts that puplis can learn from Scratch include:

- ***Sequential Processing.*** This involves the processing of application code blocks, in the order that they are laid out, starting at the beginning of a script file and continuing to the end of the script.
- ***Conditional Programming Logic.*** This involves the conditional execution of code blocks based on data collected during application execution.
- ***Use of Variables.*** This involves the storage, retrieval, and modification of data during application execution.
- ***Iterative Processing.*** This involves the repeated execution of code blocks to process large amounts of information or to control the repeated execution of code blocks required to direct the execution of a game or application.
- ***Boolean Logic.*** This involves the application of programming logic that executes based on the analysis of true/false data provided by Scratch during program execution.
- ***Interface Design***. This involves the development of user-friendly and intuitive application stage layout, making it easy for users to interact with applications.
- ***Program Synchronization.*** This involves the passage and receipt of messages between application scripts for the purpose of coordinating the execution of different parts of an application.

- ***Event Handling.*** This involves the initiation of script execution based on the occurrence of predefined events, such as the pressing of keyboard keys, the pressing of the green flag key, or the receipt of a synchronization message.
- ***Application and Game Development.*** This involves the creation of different types of computer application projects.
- ***Sprite Programming.*** This involves the use of sprites as the basis for developing graphical programs.
- ***Application Troubleshooting.*** This involves the identification, location, and elimination of programming errors, or bugs, that prevent applications from executing as they are supposed to.

2.4.1 Programming environment

2.4.1.1 Single-window user interface

Many users learn Scratch as they go, trying commands from the palette or exploring code from existing projects. To encourage such self-directed learning, the Scratch programming environment was designed to invite scripting, provide immediate feedback for script execution, and make execution and data visible.

The Scratch user interface strives to make navigation easy. It uses a singlewindow, multi-pane design to ensure that key components are always visible.

Scratch avoids floating palettes, which can get buried, and minimizes the use of panes that show only on demand.

Appendix A shows the S. window, which has four main panes. The left pane is the command palette with buttons to select categories. The middle pane shows the scripts for the currently selected sprite, with folder tabs to view and edit the costumes (images) and sounds owned by that sprite. The large pane on the upper right is the stage, where the action happens. The bottom-right pane shows thumbnails of all sprites in the project, with the currently selected sprite highlighted.To invite scripting, the command palette is always visible.[14]

Scratch provides access to over 100 code blocks. These code blocks are organized into eight categories and are made available on the blocks palette. Each of these categories of code blocks is described in the following list:

- **Motion**.Code blocks that control sprite placement, direction, rotation, and movement.
- **Looks.** Code blocks that affect sprite and background appearance and provide the ability to display text.

- **Sound.** Code blocks that control the playback and volume of musical notes and audio files.
- **Pen**. Code blocks that can be used to draw using different colors and pen sizes.
- **Control.** Code blocks that trigger script execution based on predefined events, repeatedly execute programming logic using loops, and perform conditional logic.
- **Sensing**. Code blocks that can be used to determine the location of the mouse-pointer, its distance from other sprites, and whether a sprite is touching another sprite.
- **Operators.** Code blocks that perform logical comparisons, rounding, and other arithmetic operations.
- **Variables.** Code blocks that can be used to store data used by applications when they execute. [15]

You can view the code blocks belonging to a given category by clicking on one of the eight labeled button controls at the top of the blocks palette which you can see on figure 7. Note that each category of code block is color coded, making it easy to distinguish between code blocks from different categories.

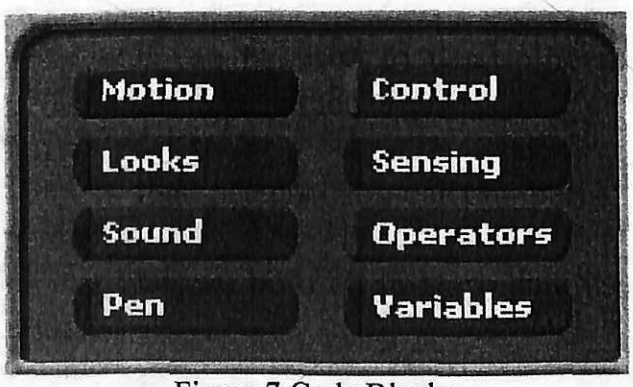


Figure 7 Code Blocks

Each of these categories of code blocks is reviewed in the sections that follow.

This review covers Scratch's entire collection of code blocks, indicating which ones support monitors and providing a brief description of each block's usage.

This avoids long, potentially overwhelming, lists of commands: in most palettes, all the commands can be viewed without scrolling. In each category, the most self-explanatory and useful commands appear near the top of the command

palette. Command blocks are color-coded by category, helping users find related blocks.

2.4.1.2 Liveness and tinkerability

A key feature of Scratch is that it is always "live". There is no compilation step or edit/run mode distinction. Users can click on a command or program fragment at any time to see what it does. In fact, they can even change parameters or add blocks to a script while it is running. By eliminating potentially jarring mode switches and compilation pauses, Scratch helps users stay engaged in testing, debugging, and improving their projects. We say that Scratch is tinkerable because it lets users experiment with commands and code snippets the way one might tinker with mechanical or electronic components. Tinkerability encourages hands-on learning and supports a bottom-up approach to writing scripts where small chunks of code are assembled and tested, then combined into larger units.

Tinkerability helps users discover the functionality of blocks. A block can be tested by clicking on it, even in the palette. Function blocks show their return value in a cartoon-like "talk bubble". We can see it on Figure 8. To help users more easily explore what blocks do, each block comes with default parameters that give an illuminating demonstration of what that block does. Scratch has help screens for every command, accessible via the right-button menu, but many users learn about commands just by trying them.

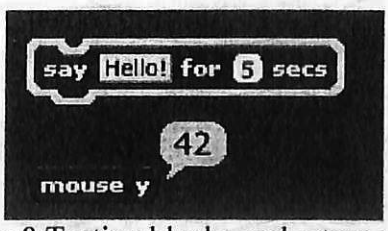


Figure 8 Testing blocks and return value

Scratch does not require that the user create complete scripts before running the projects. Program fragments can be left in the scripting pane and are saved with the project. Such fragments play a role similar to commented-out code in a text-based language. When troubleshooting, a long script can be broken into pieces and each piece tested independently.

2.4.1.3 Visible execution and error messages

Scratch provides visual feedback to show script execution. When a script is running, it is surrounded by a glowing white border as Figure 8 shows. This feedback helps he user understand when scripts are triggered and how long they

run. If a script encounters an error (e.g., dividing by zero), the border turns red and the block that caused the error is highlighted in red, in Figure 9. Scratch can also show command sequencing and flow of control. Enabling single-stepping (selected from a menu) causes blocks to flash as they run . Even when single-stepping mode is not enabled, Scratch updates the display after every command. Seeing the effect of every command, even if only as a brief flash on the screen, provides important visual clues when trouble hooting.

When people play with LEGO R _ bricks, they do not encounter error messages. Parts stick together only in certain ways, and it is easier to get things right than wrong. The brick shapes suggest what is possible, and experimentation and experience teaches what works.

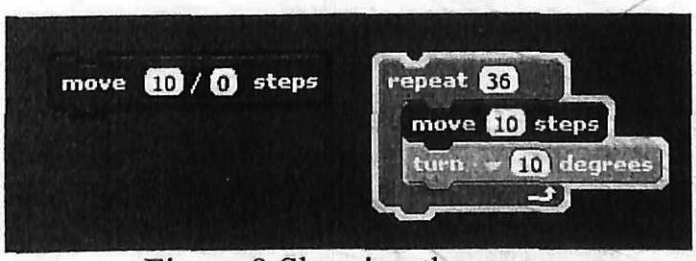


Figure 9 Showing the error

Similarly, Scratch has no error messages. Syntax errors are eliminated because, like LEGO R _ bricks, blocks fit together only in ways that make sense. But Scratch also strives to eliminate run time errors by making all blocks be failsoft. Rather than failing with an error message, every block attempts to do something sensible even when presented with out-of-range inputs. For example, the "set size" block bounds the range of its parameter so that it cannot make the sprite excessively large (possibly exceeding system limits) or invisibly small.

Of course, eliminating error messages does not eliminate errors. The user must still think carefully to write scripts that do what they want and must trouble-shoot scripts that do not work as expected. However, even when a script does not do the right thing, it does something, and that is a good start. A program that runs, even if it is not correct, feels closer to working than a program hat does not run (or compile) at all. [16]

2.4.1.4 Data concrete

In most text-based programming languages, variables are invisible, abstract, and difficult to understand. Like earlier systems such as Boxer, Scratch turns variables into concrete objects that the user can see and manipulate, making them easier to understand through tinkering and observation. In Scratch, a variable can

appear on the stage as a variable monitor , you can see it in Figure 10. Monitors allow users to see the effect of commands such as "change x by 1", helping them build a mental picture of how variables work. But monitors are not only an aid to understanding; they are also useful for their own sake as readouts e.g., to display the score in a game) or, using the optional slider, as controls. Scratch also has monitors for lists. When a list monitor is on the stage, quick animations show the effects of list operations. For example, when a list element is accessed, the index of that element flashes. Small design details can make a big ifference. In early versions of Scratch, a newly created variable was not displayed on the stage, and the gesture to make that happen was not obvious. Many users did not discover variables, even when they needed them. After changing the design so that a newly-created variable is immediately displayed, many more users started using variables.[10]

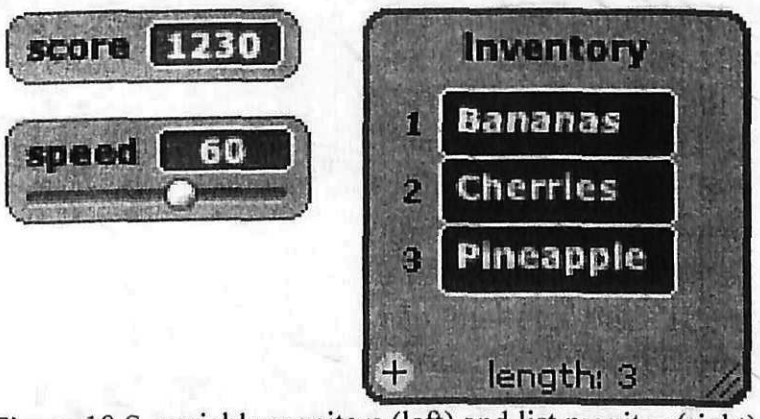


Figure 10 S. variable monitors (left) and list monitor (right)

2.4.1.5 Minimizing the command set

Scratch strives to minimize the number of command blocks while still supporting a wide range of project types. One might argue that flexibility, programmer convenience, and extra features are more important than a small command set.

However, in Scratch, unlike a text-based language, every command consumes screen space in the command palettes, so there is a higher "cost" to increasing the command set. Adding more commands requires either adding more categories or forces the user to scroll down to see all the commands within a given category. Either way, a larger command set makes it harder to find a given command in the palettes.

Scratch 1.0 had 92 command blocks. As S. has evolved, it has been a constant struggle to keep down the number of commands. Every release adds new features and new commands. Occasionally, a command is withdrawn.

Withdrawn commands, called obsolete blocks, are still supported by the runtime system to allow old projects to run.) However, more commands have been added than removed. Scratch 1.4 has 125 command blocks, although some of them do not appear until needed. One strategy for reducing the number of command blocks is to group a set of related operations into a single block with a drop-down menu to select the specific operation. Examples of this technique include the scientific math function block (a dozen math functions) and the image effect blocks (seven image effects), both of which were originally collections of individual blocks.

Another strategy is t omake sets of command blocks appear on demand when they are first needed, the blocks to access variables and lists appear only after a variable or list has been created. [10]

2.4.2 Programming language

This section examines the design of the Scratch programming language: its syntax (i.e., visual blocks), type system, object model, inter-object communications, and approach to concurrency.

2.4.2.1 Syntax

Scratch scripts are built by snapping together blocks representing statements, expressions, and control structures. The shapes of the blocks suggest how they fit together, and the drag-and-drop system refuses to connect blocks in ways that would be meaningless. In Scratch, the visual grammar of block shapes and their combination rules play the role of syntax in a text-based language.

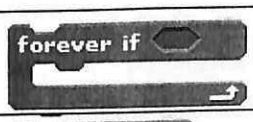
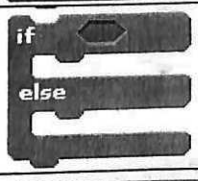
There are seven kinds of Scratch blocks: *command blocks, reporter block, trigger blocks, loop block, conditional block, predicate block, operator block*, as shown in table 1. When command blocks are snapped together to create a sequence of commands, or stack, the notches and bumps fit together like puzzle pieces.

Trigger blocks allow for starting scripts based on a given event – the green flag being pressed, a specified key being pressed, or a message broadcast fromanother sprite. Command blocks perform some functionality, such as moving or changing the color of a sprite. These blocks click together to form a sequence of commands called a "stack." Reporter blocks return a value from either an expression or a variable. Predicate blocks are like reporter blocks, but only return true or false from the evaluation of a conditional expression. Each block is colored to match its category, and the shapes of blocks suggest how to use them. Trigger blocks, which initiate execution of a script, have a curved top that does not interlock with other blocks. Loop blocks are C-shaped and fit around other control

blocks. Other control blocks have slots at the top and bottom to fit together with other control blocks. Both predicate and reporter blocks are shaped to fit in to inputs to other blocks. [17]

Control structure blocks are a kind of command block with one or more nested command sequences. The form of control structure blocks makes them easy to use. In most text-based languages, the closing delimiters for control structures can be omitted or misplaced, leading to errors. In Scratch, a control structure block is an indivisible unit. The closing arm of a loop or conditional block is part of the block itself—it cannot be misplaced—and the nesting of the enclosed command sequence is manifest. Instructors using Scratch as a quick introduction to programming before switching to a text-based language report that some students continue to "think in Scratch blocks" as a form of pseudocode, even after moving to the text-based language.

Table 1.- Block Types

№	Block Type	Explanation
1		*Trigger (hat) block* – beginnig of execution of a stack of command blocks
2		*Reporter block* – returns the current value of a variable
3		*Loop block* – repeats commands enclosed witin
4		*Conditional block* – if/else statement executes commands in one section or the other based on given predicate
5		*Predicate block* – returns true or false
6		*Command block* – performs some action
7		*Operator block* – perform math and text operations

Command blocks are like the statements of a text-based language; function blocks are like operators. Function blocks are not joined in linear sequences like command blocks. Instead, they are used as arguments to commands and nested together to build expressions.

Trigger blocks connect events (such as startup, mouse clicks, and key presses) to the stacks that handle those events. For example, all stacks starting with a green flag trigger block are run when the user clicks the start button.

Some blocks have embedded parameter slots. The shape of a parameter slot shows the parameter type: number, string, boolean, etc. Some parameter slots (ones with a white background) allow the user to enter a value from the keyboard. Others have drop-down menus or color choosers. Most parameter slots can accept a function block.

When assembling scripts, Scratch only allows blocks to be connected in meaningful ways. A command block connects when dropped into command sequence, but a function block will not connect if dropped in the same place.

As the user drags a block, Scratch gives visual feedback showing possible sequence insertion points (command blocks) or parameter slot targets (function blocks).

Disassembling stacks is easy. Grabbing the top block of a stack drags the entire stack; grabbing a block in the middle of a stack detaches that block and any blocks below it. Using the blocks editor feels natural and easy, and users often discover how to use it without being told.

2.4.2.2 Data types

Scratch has three first-class data types: boolean, number, and string. These are the only data types that can be used in expressions, stored in variables, or returned by built-in functions. In the Scratch visual language, the shape of a parameter slot indicates the data type expected and the shape of a function block indicates the type returned as in Figure 11. While there are three parameter slot shapes, there are only two function block shapes: boolean and number/string.

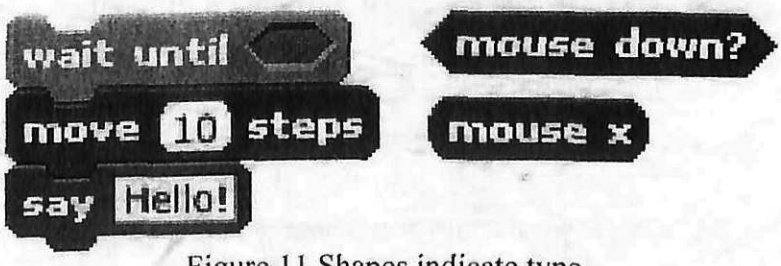


Figure 11 Shapes indicate type.

This is a consequence of the fact that Scratch variables are untyped and can contain either numbers or a strings.

The Scratch editor only allows a function block to be inserted into a parameter slot if the result would not violate the data typing constraints (Figure

10). Boolean parameter slots are the most strict, accepting only boolean function blocks. Number and string parameter slots are less strict. They accept a function block of any type, coercing the parameter to the target type if necessary.

A Scratch variable can hold values of any data type. This avoids requiring that the user specify the type of a variable when it is created. Scratch automatically converts between numbers and strings depending on context. For example, if the string "123" is passed to an arithmetic operation, it is converted to a number, while if a number is passed to the "say" command, it is converted to a string. Given the goals of Scratch, automatic conversion is preferable to requiring the user to explicitly convert between types.

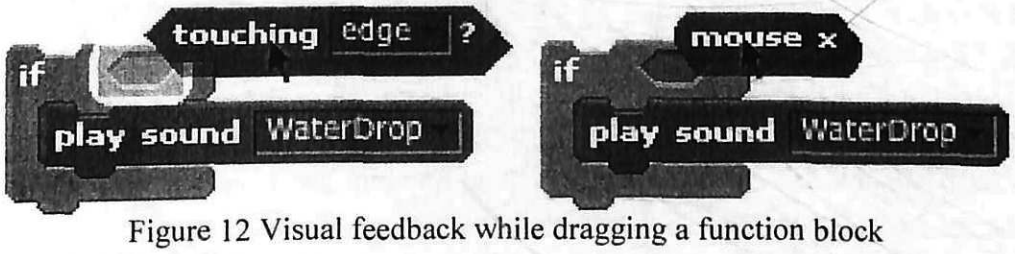


Figure 12 Visual feedback while dragging a function block

Scratch currently supports only three first-class data types. How might the visual grammar be extended to handle additional first-class types in the future? One approach would be to create a new slot/function shape for each new type. But if many types were added, that approach could lead to visual clutter and potential confusion. An alternative approach is to have all new types share the same rounded shape already used for numbers and strings, consistent with Scratch's untyped variables. This design choice mirrors the choice between static and dynamic typing in text-based languages.

One very important concept that you need to understand when working with variables is variable scope. A variable's scope identifies the location within an application where the variable's value can be modified. Scratch supports two levels of variable scope, as outlined here:

- *Local.* Variables that can be modified only by scripts belonging to the sprite in which the variable is defined.
- *Global.* Variables that can be modified by any script in an application.

2.4.2.3 Sprites

Scratch application projects are made up of objects called sprites. A sprite is a two-dimensional bitmap image drawn on a transparent background. Sprites can

be moved around and made to interact with one another. Sprites consist of three primary components, as outlined here:

- **Scripts.**Collections of code blocks that outline the programming logic that controls the operation of sprites.
- **Costumes.** Images that are used to display the sprite on an area of the Scratch IDE, referred to as the stage. Sprites can consist of any number of costumes.
- **Sounds.** Sound effects that are played during application execution when certain events occur or as background audio.

A sprite's appearance can be changed by assigning it different costumes. To move a sprite and control its behavior, you snap together code blocks to create scripts.

Sprites can have any number of scripts associated with them. Scripts can be run by double-clicking the code blocks that make them up, in which case each block in the script is executed in top-down order. You can also set things up so that scripts automatically run when various events occur. For example, you can configure script execution to occur when a sprite is clicked or when it interacts with other sprites.

Sprites are displayed and interact with one another on a stage. As such, sprites are often referred to as actors. Scratch's stage is located in the upper-right corner of its graphical interface.Sprites are objects: they encapsulate state (variables) and behavior (scripts).

However, since Scratch has neither classes nor inheritance, it is an object-based language but not an object-oriented one.

Commands operate only on the sprite in which they appear; one sprite cannot invoke a command such as "move" on a different sprite. In object-oriented terms, the implicit receiver of every command is the sprite in which it appears. An early prototype of Scratch allowed cross-sprite commands, but users found that confusing.

Every sprite has its own independent set of scripts. This design involves a tradeoff. On one hand, it is easy to understand. The scripts in a given sprite tell the entire story about that sprite's behavior; the user need not look up the class hierarchy or follow a prototype chain to find inherited scripts. Furthermore, code changes are localized: editing a script affects only the sprite in which that script appears, whereas in other languages editing an inherited method can have far-reaching and possibly unexpected consequences. On the other hand, without classes or some other code-sharing mechanism, it is more work to manage multiple sprites with identical behavior, such as the bricks in a Breakout game. In such cases, the user typically creates a single sprite with the desired behavior, then uses

the stamp tool to make as many copies as desired. If the behavior of the sprites needs to be changed after the copies have been made, the user can either make the same change in every copy or delete all but one of the copies, edit that sprite's scripts, and then make a new set of copies.

Scratch provides ready access to all kinds of sprites, organized into the following six folders:

- Animals
- Fantasy
- Letters
- People
- Things
- Transportation

2.4.2.4 Inter-sprite communications and sharing

As mentioned, sprites cannot call each other's scripts directly. Instead, Scratch uses a broadcast mechanism to support inter-sprite communication and synchronization.

Any sprite can broadcast a message (an arbitrary string). A broadcast triggers all scripts in all sprites that begin with a matching "when I receive <msg>" trigger block. For example, in response to the broadcast message "scene two," several sprites representing the characters in a story might start acting out that scene as shown in Figure 13.

The Scratch broadcast model is one-to-many, because a given broadcast can trigger many scripts (possibly in multiple sprites); loosely-coupled, because it does not matter how many receivers there are; and asynchronous, because the "broadcast" command does not wait until the triggered scripts complete.

A "broadcast" can start scripts that loop forever, similar to starting a thread. Scratch also has a synchronous variant of broadcast that waits until all triggered scripts have completed.

There are two benefits to not allowing sprites to control each other directly. First, when trying to understand why a sprite does something (e.g., moves in a certain way), the scope of possibilities is limited to scripts within that sprite itself. Second, and more importantly, because sprites are self-sufficient and only loosely coupled to other sprites, they can be moved between projects without breaking dependencies. This allows sprites to be shared.

Sharing sprites encourages code reuse and collaboration. For example, a sprite with generic arrow key motion controls could be shared so that other Scratch users can import that sprite into their own projects. In the process of adapting a

sprite for their own purposes, users are exposed to new commands and programming techniques. Sharing sprites also fosters collaboration: when users are working together, each of them can independently develop sprites and then combine those sprites to create the final project. [18]

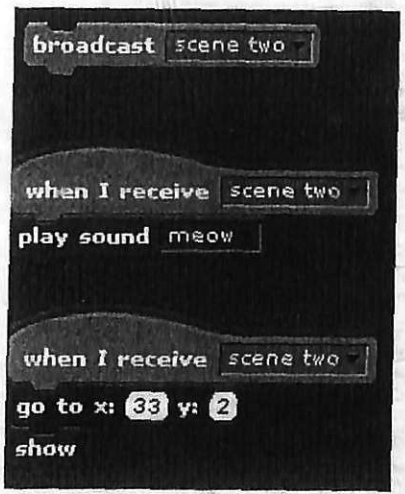


Figure 13 A broadcast command and two scripts it triggers.

2.4.2.5 Paint editor

In addition to using the sprites supplied with Scratch and graphics that you acquire from the Internet, you can always create your own sprite using any graphic/paint program. Although it does not have all of the bells and whistles that applications like Corel Paint Shop Pro or Adobe Photoshop have, Scratch's built-in Paint Editor, shown in appendix B, offers everything needed to draw or modify graphics for use as sprites and backgrounds.

As appendix B demonstrates, Scratch's Paint Editor is divided into multiple components. Thanks to Scratch's cross-platform design, the Paint Editor looks and operates identically on both Microsoft Windows and Mac OS X.

Links to the Paint Editor program are located just under the stage and within the Costumes and Backgrounds tabs located in the scripts area. The Paint Editor program can be used to create or modify new sprites, costumes, and backgrounds. Most of the space on the Paint Editor's window is dedicated to a drawing canvas. To draw on the canvas, you select different drawing commands from the toolbar and then use the mouse to draw on the canvas. You can work with different colors and apply a range of special effects.

If the size of the graphic being worked on exceeds the available area, scrollbars are enabled on the right-hand side and the bottom of the drawing canvas, allowing you to view all parts of the graphic. You can also use the Zoom In and

Zoom Out buttons located at the bottom of the Paint Editor window to temporarily increase or decrease the magnification of the drawing canvas.

2.4.2.6 Procedures

Early versions of S. had a mechanism for creating procedures. In early field tests, however, many users were confused by procedures since they seemed very similar to broadcasts—both involved associating a name with a collection of commands. In the interest of simplicity and minimalism, procedures were removed from the language before Scratch was officially released, and S. has gotten along surprising well without them. Yet procedural abstraction is one of the "powerful ideas" of computer science and procedures have practical value as a way to structure code as projects grow. The Scratch team is considering reintroducing procedures as a way for users to define their own command blocks. Other researchers have been developing a variant of S. that supports not only procedures but first-class functions, closures, and a complete functional programming system.

2.4.2.7 Concurrency

Concurrency (or "multi-threading") is often considered an advanced programming technique. Yet our everyday world is highly concurrent, so Scratch users are not surprised that a sprite can do several things at once. For example, in a simple paddle game, one script might move the ball, another might make it bounce when it hits the paddle, and another might end the game when the player misses the ball as in Figure 14. In Scratch, all of these scripts can run concurrently.

Scratch lacks the explicit concurrency control mechanisms often found in other programming languages, such as semaphores, locks, or monitors. Instead, Scratch builds concurrency control into its threading model in a way that avoids most race conditions, so that users do not need to think about these issues. This is done by constraining where thread switches can occur.

In many preemptive threading models, a thread switch can occur between any two instructions. In the Scratch model, a thread switch can occur in only two places: (1) on a command that waits explicitly (e.g., "wait 1 second") or (2) at the end of a looprogramming A thread switch cannot occur in the middle of a sequence of non-waiting statements, or between the test of an "if" command and its body.

The S. threading model allows users to reason about a script in isolation, giving little or no consideration to potential side effects from the interleaved execution of other scripts. For example, in the code shown in Figure 15, the command that sets the "busy" variable is guaranteed to be executed immediately after the test, without any other thread getting control. As this example shows, the

Scratch threading model supports a kind of implicit "critical section" that, in theory, allows the user to create their own synchronization mechanisms. In practice, such mechanisms are seldom necessary.

Although the Scratch threading model avoids most race conditions, it does not eliminate all concurrency issues. The most common one that arises in Scratch is when multiple scripts are triggered by an event or broadcast, and the ordering of those scripts is not what the user expects. However, once users understand that multiple scripts triggered at the same time are run in an arbitrary order, they readily understand the solution, which is to have the event trigger a single script which then triggers the other scripts in the desired order.

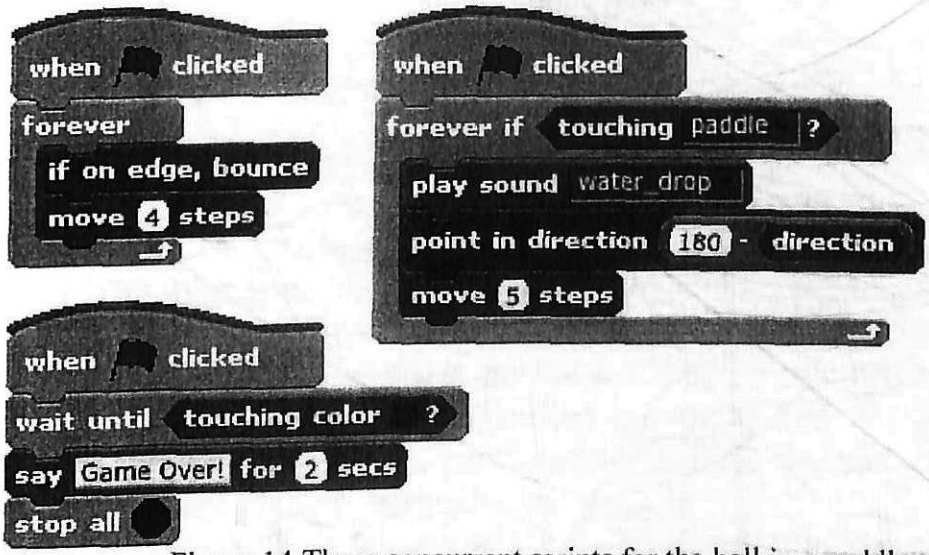


Figure 14 Three concurrent scripts for the ball in a paddle game

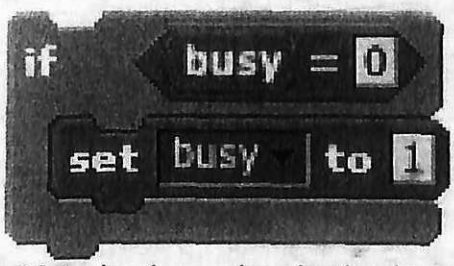


Figure 15 A simple synchronization lock in S..

2.4.2.8 Mathematical calculations

Scratch provides robust support for performing mathematical calculations. This gives you the ability to develop applications that can manipulate numeric data in a variety of ways. Scratch provides this support through numbers code blocks.

create a number guessing game that automatically generates a random number and then challenges the player to try to guess it. Once the random number is generated and stored in a variable, the player needs to be prompted to try to guess it (perhaps by clicking on one of 10 buttons with numbers printed on them). Once the player's guess is captured, the application needs to compare the player's guess against the value of the variable that stores the game's random number to determine whether the player's guess is correct. To facilitate this type of comparison operation, S. provides access to the three code blocks shown in figure 20.

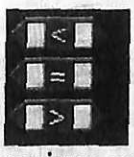


Figure 20 Comparison operation blocks

The first and last code blocks shown in figure 20 allow you to compare one value against a range of values. The first code block checks to see if the numeric value specified in its first input field is less than the value specified in its second input field. The third code block does the opposite, checking to see if the numeric value specified in its first input field is greater than the value specified in its second input field. The middle code block is used to determine if two values are equal.

In addition to all of the mathematical operations that you can put together using the numbers code blocks previously discussed in this chapter, Scratch provides one additional multi purpose code block.

This code block is designed to perform any of 12 different mathematical functions, which can be selected from the code block's drop-down list. The functions that this code block can perform are outlined in the following list:

- **abs.** Returns the absolute, non-negative value of a number.
- **sqrt.** Returns the square root of a number.
- **sin.** Returns a value representing the sine of an angle.
- **cos.** Returns a value representing the cosine of an angle.
- **tan.** Returns a value representing the tangent of an angle.
- **asin.** Returns the arc sine for the specified numeric value.
- **acos.** Returns the arc cosine for the specified numeric value.
- **atan.** Returns the arc tangent for the specific numeric value.
- **ln.** Returns the inverse of the natural exponent of a specified value (i.e., the opposite of e^).
- **log.** Returns the natural log of a number.
- **e^.** Returns the natural exponent of a specified value.

Specifically, Scratch evaluates an expression using a top-down approach. When applied to the example shown in figure 18, Scratch evaluates it as follows:

1) First, it calculates the value of the two top code blocks. Therefore, 3 is multiplied by 3, yielding a value of 12, and 5 is multiplied by 2 , yielding a value of 10.

2) Next, the expression located in the second level code bock (the addition block) is evaluated. Therefore, 12 is added to 10, yielding a value of 22.

3) Finally, the lowest level code block is evaluated, dividing 22 by 3 and resulting in a final value of 7,3.

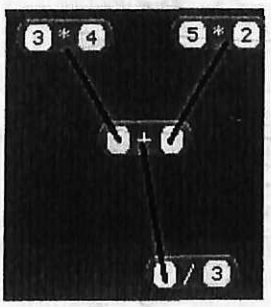


Figure 18 Embedding the code blocks

Some applications, such as computer games, require an element of randomness or chance. For example, a game that needs to simulate the rolling of dice needs to be able to create a pair of random numbers in the range of 1 to 6. S. provides the capability through the code block shown in figure 19.

Figure 19 Random numbers code block

This code block provides a means of generating random integer (whole) numbers using any specified range of numbers. The default range is 1 to 10, but you may change the input fields to suit your needs. If needed, you can generate negative numbers. In addition to hard coding a numeric range into the control, you can substitute variable blocks by dragging and dropping them into either or both of this code block's input fields.

In order to work with numbers, you often need to mathematically manipulate them as demonstrated in the previous section. Doing so will ultimately leave you with a result. Typically, you will want to do something with this result once it has been calculated. For a simple application, all you may need to do is display its value. However, more often than not, you are going to end up using it to guide the execution of your application in some manner. For example, suppose you want to

Numbers code blocks are reporter blocks and therefore can only be used in conjunction with stack code blocks. [19]

Like all modern programming languages, Scratch provides programmers with the ability to add, subtract, multiply, and divide numeric data. This capability is offered through the code blocks shown in figure 16.

The use of these code blocks is quite intuitive, with each code block clearly identifying its usage. These code blocks can be embedded within any S. code block that accepts numeric input.

Figure 16 Mathematical calculations code blocks

As is the case with all programming languages, Scratch allows you to string together different combinations of numbers code blocks in order to create more complicated numeric calculations. For example, take a look at the following script shown on figure 17.

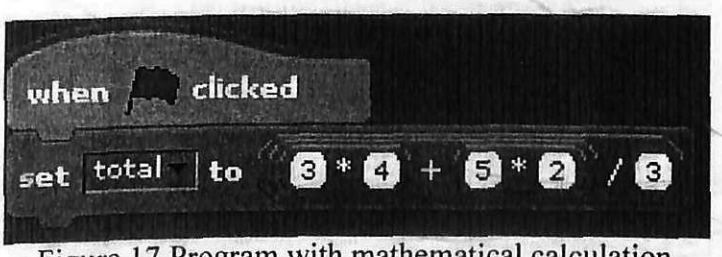


Figure 17 Program with mathematical calculation

Here, a small script has been created that evaluates a numeric expression and assigns the result to a variable named Total. This equation was created by embedding a series of numbers code blocks within one another. Specifically, the equation was created by embedding the code blocks shown in Figure 18 into one another.

As shown in Figure 18, the equation was assembled by embedding the division code block into a variable block. Next, the addition code block was embedded within the left-hand side of the division code block. Finally, a multiplication code block and a subtraction code block are embedded within the input fields of the addition code block.

Like all programming languages, Scratch evaluates the components of mathematical expressions by following a specific order, referred to as the order of precedence.

– **10^.** Returns the value of a number raised to the 10th power.

These code blocks can be real time savers when developing applications that require the use of any of the mathematical functions supported by the code block, saving you the trouble of implementing the underlying programming logic yourself to retrieve similar results. As a result, not only will you spend less time working on the development of your application, but the programming logic that you have to develop will be simplified and easier to maintain, since this code block can do most of the heavy lifting for you.

To specify which function you want to work with, all you have to do is select it from the code block's drop-down list.

2.4.2.9 Running Scratch applications

The stage is the area on the Scratch IDE, located in the upper-right side, as shown in appendix C, where your Scratch applications execute. The stage provides a place for the sprites that make up your applications to interact with one another and the user.

The stage provides the canvas upon which sprites are displayed and interact with one another The stage is 480 units wide and 360 units high. The stage is mapped out into a logical grid using a coordinate system made up of an X-axis and a Y-axis.

The X-axis runs from coordinates 240 to –240, and the Y-axis coordinate runs from coordinates 180 to –180. The middle of the stage has a coordinate location of (0, 0). Scratch keeps you informed of the pointer's location whenever it is moved over the stage by displaying its (X, Y) coordinate position in the mouse x: and mouse y: fields just beneath the bottom-right side of the stage.

The stage can be assigned one or more backgrounds, allowing you to change its appearance during application execution. By default, all Scratch applications are assigned a blank background. You can add new backgrounds by clicking on the Stage thumbnail, located on the left-hand side of the sprite list, and then clicking on the Backgrounds tab located just above the scripts area. Like sprites, the stage can be assigned its own scripts and sound effects.

As we know Scratch runs our applications on the stage within the IDE by default. However, if you click on the Presentation Mode button, located just beneath the bottom-left corner of the stage, you can run your Scratch application project in Presentation mode. To see how this works, click on the Open button located at the top of the Scratch IDE and then locate and open the Hello World project. Next, click on the Presentation Mode button to switch to full-screen mode.

Once in Presentation screen mode, single-click on the sprite representing the kitten and watch as your application executes.

You can exit Presentation mode at any time either by clicking on the Exit Presentation Mode icon located just above the upper-left side of the stage or by pressing the Escape key.

Whether running your application from the IDE's stage or in Presentation mode, you can automatically start any scripts that begin with the green flag control block by clicking on the green flag button located in the upper-right corner of the IDE.

This same button is also available in Presentation mode. By clicking on the red stop button located right next to the green flag button, you can stop the execution of your applications any time you finish working with them.

2.4.2.10 Comparison with Alice and Greenfoot

Like Scratch, Alice, and Greenfoot are intended to introduce programming to those without prior experience and, as a result, the three systems share many of the same design goals. For example, all three systems support rich graphics and sound and let users create projects that connect to their interests. Both Alice and Greenfoot target older students than Scratch, introduce class-based object-oriented programming, and emphasize Java or Java concepts, making them suitable as preparation for the Advanced Placement Exam in Computer Science. Greenfoot, since it is Java, also allows students to explore high-performance computation (e.g., complex simulations or cryptography problems) and to extend the system. Alice is the only one of these systems that supports 3-D graphics.

Scratch targets younger users than the other two systems, focuses on selfdirected learning, and emphasizes tinkerability. While all three systems allow media to be imported, Scratch includes tools to draw images and record sounds. The 2-D images used by Scratch and Greenfoot are easier to create and edit than the 3-D models used by Alice. The Scratch command set is designed to encourage a wider variety of project types than the other two systems, and the ability to easily share projects on the Scratch Web site, with its active user community, provides motivation and opportunities to learn from others.

Scratch has been used as an introduction to both Alice and Greenfoot. The transition from Scratch to Greenfoot is especially smooth, since both are 2-D systems and Scratch's sprites and stage map directly to Greenfoot's actors and world.

2.5 ScratchR

ScratchR is the web-based platform that lets kids share their Scratch creations on-line. ScratchR gives kids access to an on-line community of Scratch programmer like themselves. One of the goals of ScratchR is to help the creative process by fostering collaboration. To achieve collaboration, ScratchR tries to provide a wide range of entry points: from the simple act of interacting with a project to commenting and uploading new projects. The platform is a repository of user-generated content that then serves as a source of inspiration and appropriable objects. Lastly, ScratchR allows members to connect with each other, forming a social network of creators. The immediate result of this work is the public release of the Scratch on-line community which is as an important element of the whole Scratch experience. The platform is being evaluated with a small group of middle-school students and a larger group of beta testers. The result of this research will provide a theoretical framework for analyzing and designing on-line communities that engage kids in learning from each other by sharing their creations.

2.5.1 Programmable media and Scratch

The easiest way to start describing the term programmable media is by giving examples of what it is not. Videos, audio, images and text are not programmable media. Programmable media is created when images, audio and text, are controlled according to certain behavior. Scratch is an authoring environment for creating programmable media. For example, a kid can use the image of a cat (non-programmable media) and then define a behavior like "move the cat 10 steps when the right arrow key of the computer is pressed". That mix of behavior and non-programmable media is what we call programmable media. It's shown as a scheme on figure 21.

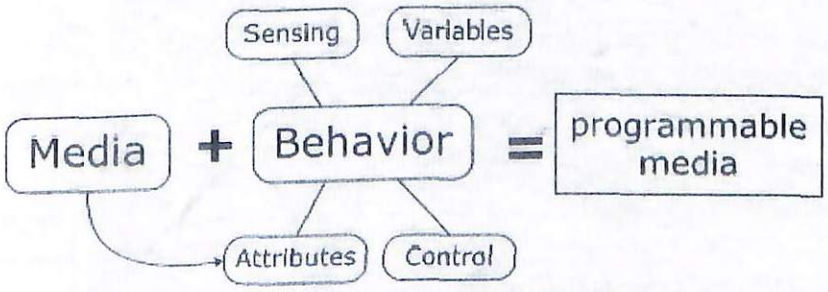


Figure 21 Programmable media scheme

Platforms for sharing user-generated content are not new; in fact, web sites like that have gained a lot of attention in recent years. For example, YouTube and

Flickr are well-known platforms for sharing non-programmable media. In these sites users get different types of feedback from other members of the community and find inspirational ideas by looking at other people's creations (e.g. pictures and videos). However, there is not a platform for sharing programmable media and especially not one that focuses on youth and children.

Due to the nature of programmable media, it is very useful to let people reuse other people's work (e.g. blocks of code and sprites). That is what we call creative appropriation: the utilization of someone else's creative work in the making of a new object. There are very few platforms that allow creative appropriation; one of them is the OPENSTUDIO web site which focuses on sharing digital paintings. There seems to be no platform that allows people to engage in creative appropriation of programmable media. S.R is a unique platform that gives young programmers access to the power of collective intelligence. [20]

2.5.2 Architecture

ScratchR is composed of three basic elements: a repository of Scratch projects, a collection of user-generated metadata about those projects and a community of socially networked people. Members of the community can share (upload) or appropriate (download) the original building blocks of the Scratch projects they interact with on the web site. Internally, the platform and the Scratch files themselves keep a record of those activities. Users can also participate in the community by tagging, commenting, grouping and rating other people's projects. These activities occur in the context of a social network where members can connect with each other by establishing friendship links. You can see components of ScratchR platform on appendix D.

The Scratch programming system strives to help users build intuitions about computer programming as they create projects that engage their interests. The user interface layout, with its prominent command palette and central scripting area, invites users to program. The Scratch blocks language eliminates syntax errors, allowing users to focus on interesting problems right away, rather than struggling merely to get their program to compile. Block shapes and visual feedback while dragging help the user learn how to assemble programs and use data types. Runtime error messages are avoided through failsoft commands, while a carefully designed concurrency model avoids race conditions.

The Scratch programming language emphasizes simplicity. The type system and object model were designed to work smoothly without prior explanation, yet to make perfect sense on closer examination. Scratch has a surprisingly small number of commands, especially given the wide range of project types it supports. A

loosely-coupled inter-object communication system allows sprites to be exchanged without breaking dependencies, fostering collaboration and code sharing.

The system is always live, with no run/edit switch, so commands or code snippets can be run with a click, and graphical feedback shows execution. Variables and lists have concrete visualizations, so the effect of data operations can be seen immediately. These features support and reward discovery through tinkering.

The Scratch programming environment and language work together to create a system that is exceptionally quick to learn—users can be programming within fifteen minutes—yet with enough depth and variety to keep users engaged for years. [21]

3 SCRATCH TUTORIAL

3.1 About tutorial

So how teach pupils to Scratch in our kazakh shcools? As a teacher of Computer Science this question was one of my main problems in my research. I have been teaching computing for 3 years, and this short time was enough to be faced with the problem of shortage of books in kazakh to teach for computing.

As we know in our counrty pupils are taught for computing from 7th grade, i.e. at the age of 13-14. And books for schools begins from this grade. But nowadays more and more schools began to teach Computer Science from early 3rd or 4th grades, without having special books. Teaching pupils for computing at the early ages is nessecity of our life today. Almost every family has a computer at home and pupils start to work on it and using this chance, as technology is developing every hour today, we need to begin teaching puplis for computing at the age of 8-9.

As a teacher of primary school I had to meet with a problem of shortage of books for this generation. So I decided to prepare a tutorial about how to start work on S..

A tutorial is a method of transferring knowledge and may be used as a part of a learning process. More interactive and specific than a book or a lecture; a tutorial seeks to teach by example and supply the information to complete a certain task.

Depending on the context a tutorial can take one of many forms, ranging from a set of instructions to complete a task to an interactive problem solving session (usually in academia).

I made a tutorial book in our native language, in kazakh in order to be useful to our kazakh schools. Tutorial consists of 15 lessons such as:

1. *Fisrt Steps to Scratch*
2. *Sprites. Working with several characters*
3. *Drawing on Scratch*
4. *Using Drawings in the Program*
5. *Synchronous and coherent events*
6. *Working with Stage Background*
7. *Animation*
8. *Animation (continuation)*
9. *Using Sounds in Program*
10. *Coordinates*
11. *Variables and Operaters*

12. *Storytelling*
13. *Pen block*
14. *First Game*
15. *Car Racing Game*

Each lesson begins with a step by step explanation of how to program. After each lab there is a short conclusion of the lesson and tasks and questions.

The first lesson (Fisrt Steps to Scratch) describes how to start working on Scratch .This lesson explains how to use Scratch tools, introduces with Scratch environment and how to write fisrt program on Scratch.

Lesson 2 (Sprites. Working with several characters) lets children to discover working with several characters and to change sprites properties.

On the Lesson 3 (Drawing on Scratch) puplis learn to draw on Scratch editor and in the 4th lesson (Using Drawings in the Program) they learn to use their drawings in the program.

Lesson 5 (Synchronous and coherent events) describes concurrency and coherent events in programming.

On the lesson 6 (Working with Stage Background) pupils are allowed to introduce with the concept of Background. They will learn how to change background and it's settings.

It will be more interesting to learn animating with Scratch on lesson 7 (Animation) and lesson 8 (Animation (continuation)). In these lessons pupils will be able to animate with templates and also to make their own animations by drawing animation costumes. They will also learn to ccompany their projects with sounds on lesson 9 (Using Sounds in Program).

We said that with Scratch pupils learn mathematical and computational ideas that are built into the Scratch experience.Pupils will introduce with this ideas on lesson 10 (Coordinates) and 11 (Variables and Operaters). Also they will learn how to make conditions in different situations. Next 12th lesson (Storytelling) teachs pupils to make stories on Scratch.

Lesson 13 about Pen Block, how to use pen and what we can do with pen. Lessons 14 and 15 demonstrates how to do games with Scratch.

At the next section we will find a fisrt lesson as an example from my tutorial.

3.2 Example lesson from tutorial

Lesson 1: Getting Started with Scratch.

This is the Scratch window on figure 22:

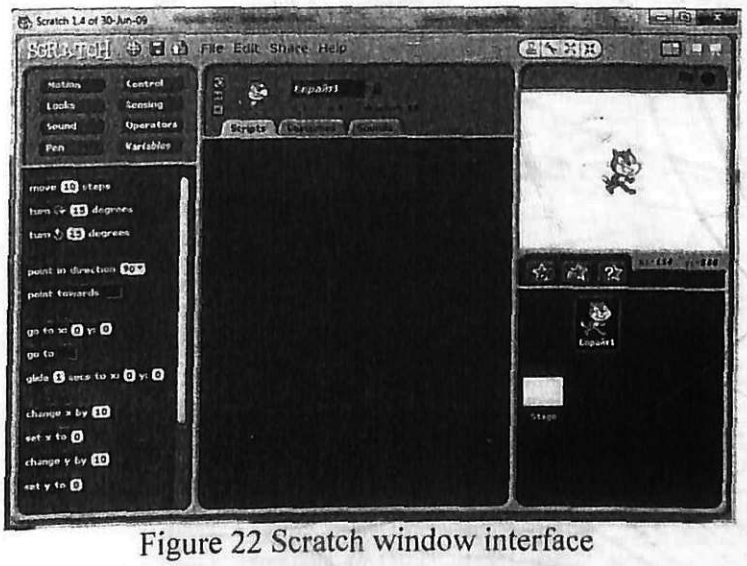


Figure 22 Scratch window interface

Here we can create our own stories, games, animations.
As you see on figure 23 the Scratch interface consists of three parts:

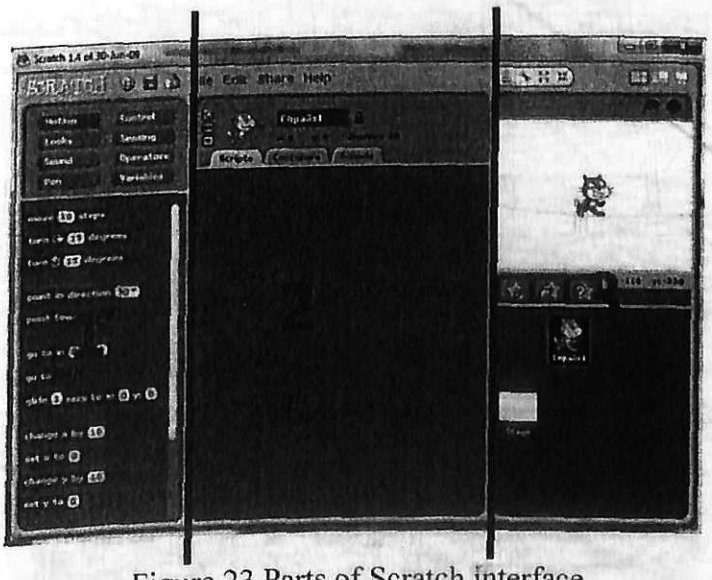


Figure 23 Parts of Scratch interface

Part 1: blocks palette
Part 2: sprite info and scripts area
Part 3: stage and sprites list.

Now we can see only one character on the stage, which you can see on figure 24 given below:

Figure 24 Scratch character

But we can also change this character to an another one.

So let's start doing our first program.

When we first open the S. environment we can see only Motion blocks on the blocks palette. But in S. there are 8 blocks: Motion, looks, sound, pen, control, sensing, operators, and variables:

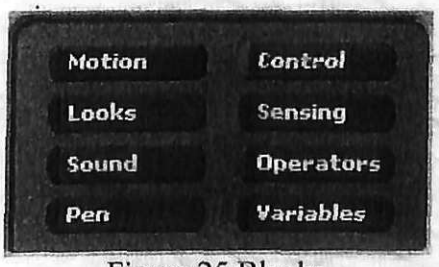


Figure 25 Blocks

So let's choose the script move 10 steps from the Motion blocks. Choosing scripts is doing by dragging and dropping block to the scripts area as given in figure 26:

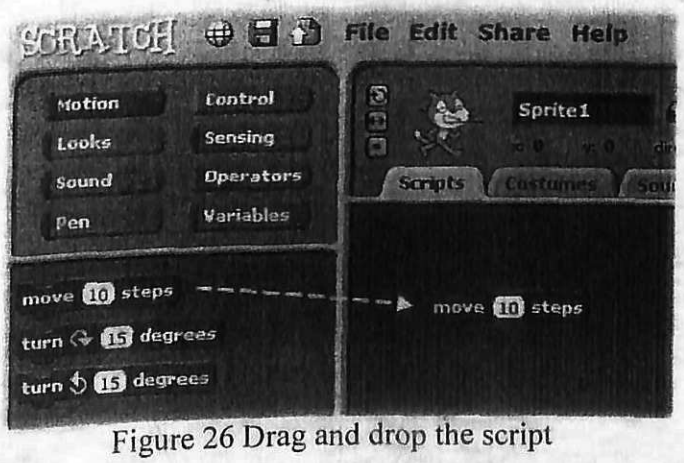


Figure 26 Drag and drop the script

Now each clicking on this script will let our character move 10 steps forward. But, of course, we can change the number of steps. So even this one script we can call program.

In Scratch programs must run by clicking the green flag. But now it's impossible, because our program is not complete. To complete our program it needs one more script. It's a "when green flag clicked" from Control blocks. This

script allows the program character to understand what commands to do when the green flag clicked.

So let's join "when green flag clicked" to "move 10 steps", also by dragging and droppping as given in figure 27:

Figure 27 Fisrt steps

Now our program will start by clicking green flag.

Let's develop our program. Now our program let's the character to move 10 steps and stoprogramming To move the character permanently choose the script "forever" and join it to the program code as shown below on figure 28:

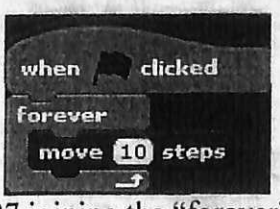


Figure 27 joining the "forever" script

As a result we see that the character starts to move by clicking green flag and it must stop only by clicking red button . But we see that the character stops at the end of the stage not waiting clicking the red button. It means we must complete our program. Join the script "if on edge, bounce" from the Motion blocks like on figure 29:

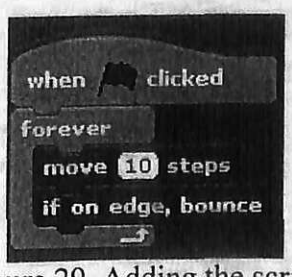


Figure 29 Adding the script

What is the result? The cat moves on the stage and when it gets the end of the stage turns back and moves again. But here is a problem: when cat goes back it moves head over heels as in figure 30:

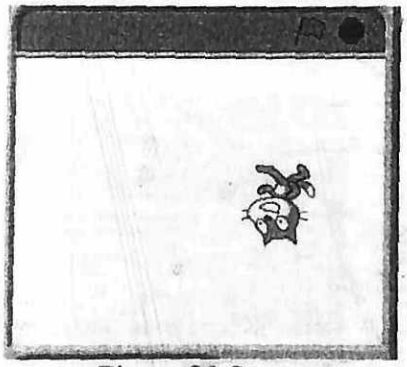


Figure 30 Stage

Of course we can solve this problem. Go to the sprite info, it's located in the middle at the top of the window. Figure 31 shows that:

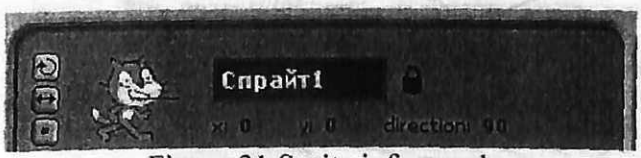


Figure 31 Sprite info panel

Here we can see 3 buttons.Now we see that the button "can rotate" is choosen. So why cat moves head over heels. We must choose "only face left-right" to let the cat to move correctly as in figure 32:

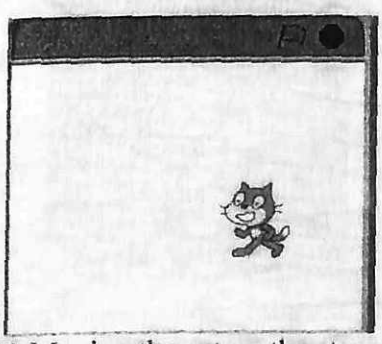


Figure 32 Moving the cat on the stage correct

Now cat moves on the stage permanently. So this is our little, but perfect first program on Scratch.

What did we learn?

So we can write programs in Scratch environment. To write programs we use special blocks called scripts. The result of our program we can see by how characters responds on the stage.

This is our first program on Scratch in figure 33:

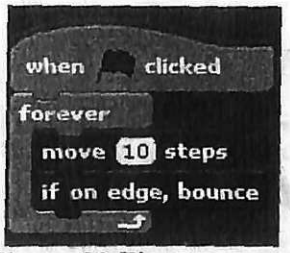


Figure 33 Fisrt program

The meaning of this program is when we click the green flag cat moves on the stage, when it gets to the end of the stage it goes back and so on. This proccess will continue untill we click the red button.

There is a one cycle on this program. We use cycle when we need repeating of some actions. In our program the cycle is the script forever, because it lets the 2 scripts including it to repeat permanently.

Task

1) Write this program and explain how it works:

Figure 34 Example

2) Make the following changes to the program given in Task 1: may the cat after each 20 steps wait 1 second.

Conclusion

Of course, most students will not grow up to become professional programmers, just as most will not become professional writers. But learning to program offers benefits for everyone: it enables students to express themselves more fully and creatively, helps them develop as logical thinkers, and helps them understand the workings of the new technologies that they encounter everywhere in their everyday lives.

CONCLUSION

Everyone ought to be able to read and write; few people within the global mainstream would argue with that statement. But should everyone be able to program computers? The question is becoming critically important as digital technology plays an ever more central role in daily life. The movement to make code literacy a basic tenet of education is gaining momentum, and its success or failure will have a huge impact on our society.

What are some of the fundamental concepts of computer science, and how can they be taught in school? One of the most fundamental aspects is that all computer systems are controlled by programs, i.e. by rigorously defined algorithms expressed in a formal notation. Modern society relegates an ever-growing number of every-day tasks to machines. These machines act as controllers that initiate actions based on their current state and on received inputs. The number of possible behaviors, of sequences of actions triggered by different environmental conditions, is usually so huge as to be impossible to enumerate. Yet, we aim to be sure that each and every possible behavior, of which only a tiny fraction will ever be played out, is „correct“ in some precise sense. The way to do that is to write a specification that captures the practical infinity of processes that may evolve over time, depending on received inputs. A program is such a formal specification, and „program“ is surely among the most fundamental concepts required to understand computers.

The Scratch programming system strives to help users build intuitions about computer programming as they create projects that engage their interests. The user interface layout, with its prominent command palette and central scripting area, invites users to program. The Scratch blocks language eliminates syntax errors, allowing users to focus on interesting problems right away, rather than struggling merely to get their program to compile. Block shapes and visual feedback while dragging help the user learn how to assemble programs and use data types. Runtime error messages are avoided through failsoft commands, while a carefully designed concurrency model avoids race conditions.

The Scratch programming language emphasizes simplicity. The type system and object model were designed to work smoothly without prior explanation, yet to make perfect sense on closer examination. Scratch has a surprisingly small number of commands, especially given the wide range of project types it supports. A loosely-coupled inter-object communication system allows sprites to be exchanged without breaking dependencies, fostering collaboration and code sharing.

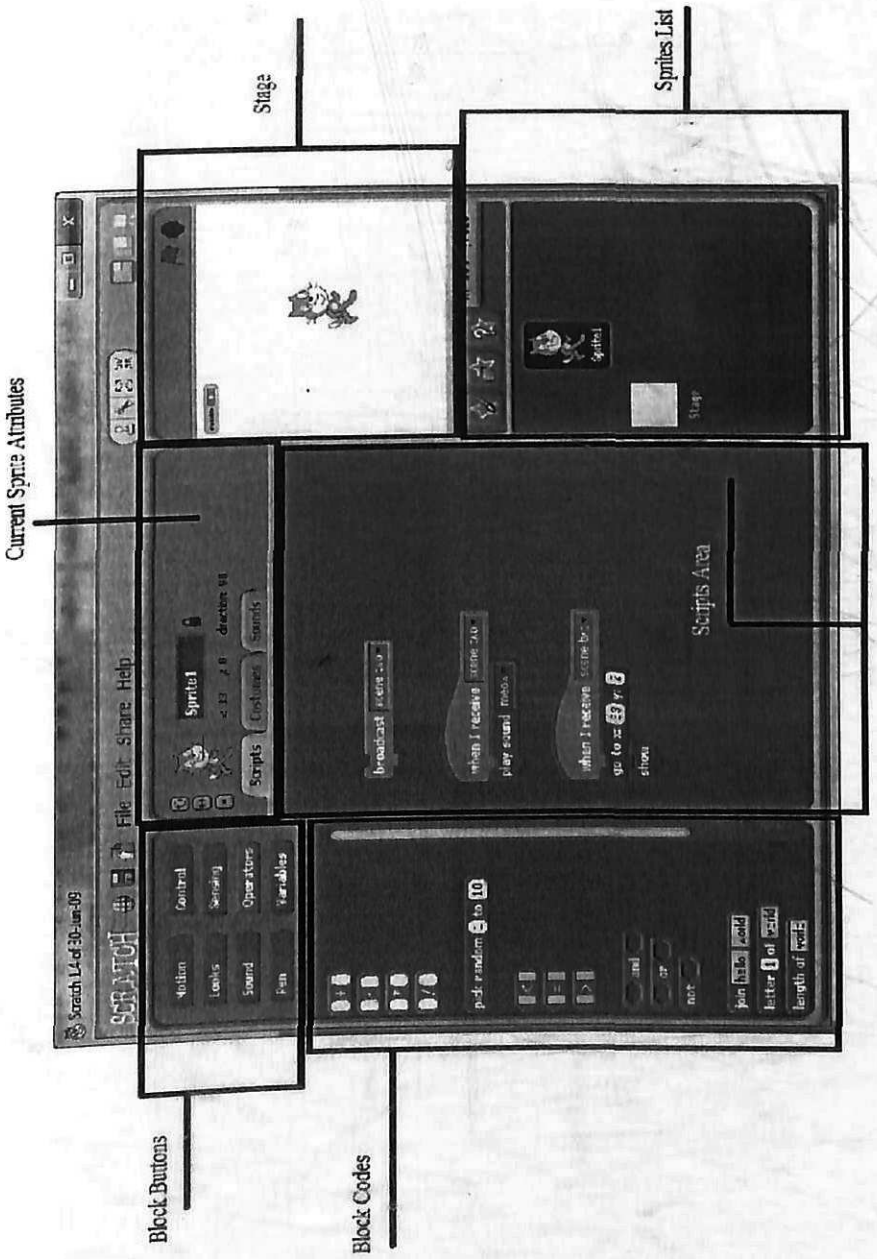
The system is always live, with no run/edit switch, so commands or code snippets can be run with a click, and graphical feedback shows execution.

Variables and lists have concrete visualizations, so the effect of data operations can be seen immediately. These features support and reward discovery through tinkering.

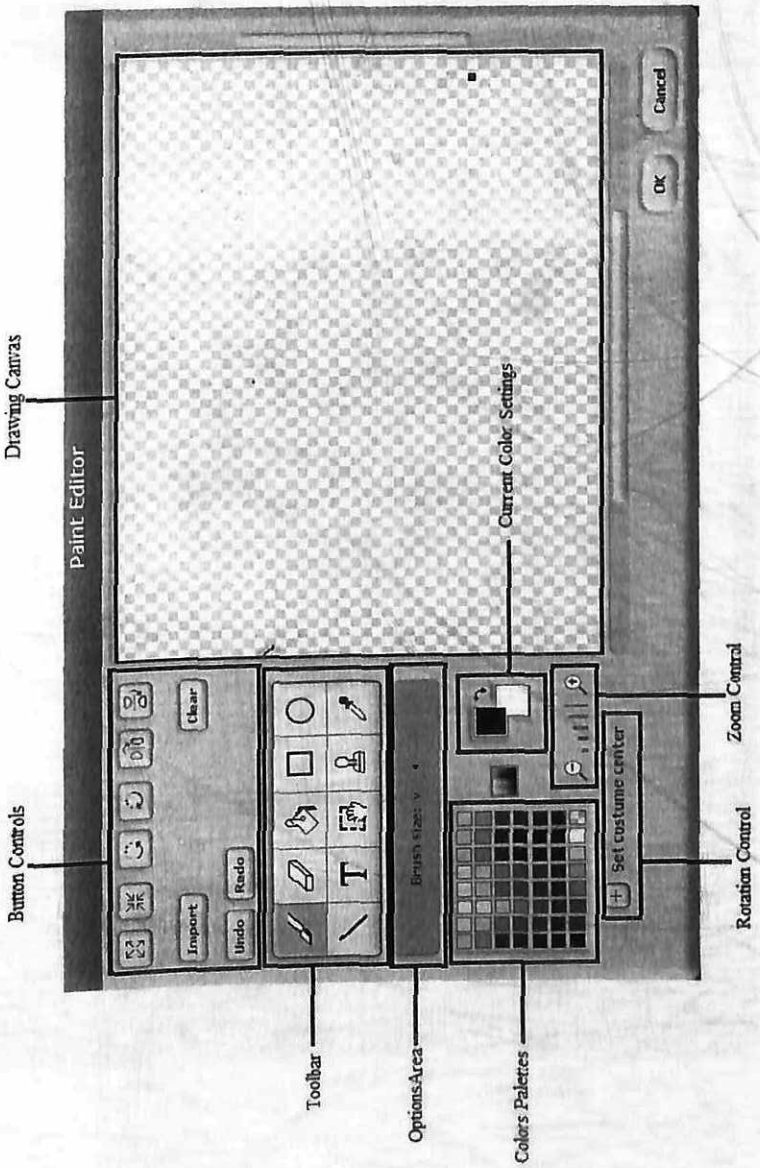
The Scratch programming environment and language work together to create a system that is exceptionally quick to learn—users can be programming within fifteen minutes—yet with enough depth and variety to keep users engaged for years.

REFERENCE

1 http:// en.wikipedia.org/wiki/Computer_programming
2 Raimond Reichert Programming in schools – why, and how?. – Zurich, 2001
3 Mitchel Resnick Scratch:Programming for All. - communications of the acm, november 2009
4 John Woollard Computer Programming inKey Stage . - School of Education, University of Southampton, September 2009.
5 Anne Beug Teaching Introductory Programming Concepts: A Comparison of Scratch and Arduino. - San Luis Obispo, June 2012
6 http://www.Scratch.mit.edu
7 Mitchel Resnick Rethinking Learning in the Digital Age.- MIT Laboratory, 2002.
8 Natalie Rusk 21st Century Learning Skills. - Oxford University Press, 2003.
9 Jerry Lee Ford,Jr. Scratch Programming for Teens. - Canada, 2009
10 John Maloney The Scratch Programming Language and Environment. - ACM Transactions on Computing Education, No. 4 November 2010.
11 Andres Monroy-Hernandez ScratchR: sharing user-generated programmable media. - Aalborg, Denmark, June 6-8, 2007
12 Freeth, Tony, Jones, Alexander Calendars with Olympiad display and eclipse prediction on the Antikythera Mechanism. – Nature 454(7204) July 31, 2008 . p.-614–617
13 Chun, Wendy On Software, or the Persistence of Visual Knowledge. - Grey Room 18. Boston, 2004, p.-34-35
14 James L. Elshoff , Michael Marcotty Improving computer program readability to aid modification. - Communications of the ACM №8, Aug 1982, p.-512-521
15 A.K. Hartmann Practical Guide to Computer Simulations.- Singapore: World Scientific, 2009
16 A. Hunt, D. Thomas, W. Cunningham The Pragmatic Programmer.- Amsterdam: Addison-Wesley Longman, From Journeyman to Master, 1999
17 Brian W. Kernighan The Practice of Programming. – Pearson,1999
18 Weinberg, Gerald M. The Psychology of Computer Programming.- New York: Van Nostrand Reinhold,2000
19 Monroy-Hernández, A. and Resnick, M. Empowering kids to create and share programmable media. - ACM interactions 15, 2, March 2008,p- 50–53
20 http://inst.eecs.berkeley.edu/~cs10/fa11/
21 http://www.chirp.scratchr.org/blog/?m=201105

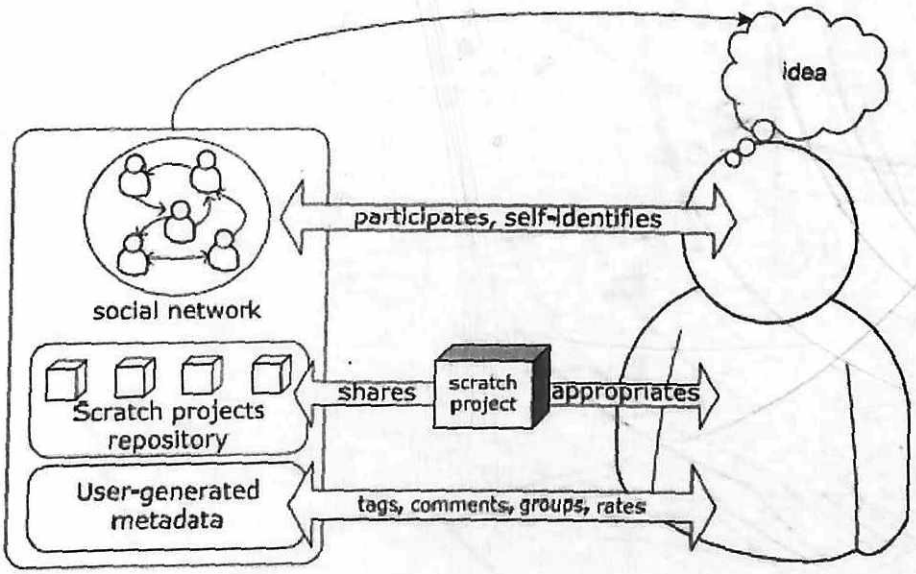


Scratch using interface



Scratch paint editor

Scratch stage



Components of the ScratchR platform