The Greenfoot Programming Environment

MICHAEL KÖLLING
University of Kent

Greenfoot is an educational integrated development environment aimed at learning and teaching programming. It is aimed at a target audience of students from about 14 years old upwards, and is also suitable for college- and university-level education. Greenfoot combines graphical, interactive output with programming in Java, a standard, text-based object-oriented programming language. This article first describes Greenfoot and then goes on to discuss design goals and motivations, strengths and weaknesses of the system, and its relation to two environments with similar goals, Scratch and Alice.

Categories and Subject Descriptors: K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education*

General Terms: Design, Human Factors, Languages

Additional Key Words and Phrases: Greenfoot, programming environment, programming education

ACM Reference Format:
Kölling, M. 2010. The greenfoot programming environment. ACM Trans. Comput. Educ. 10, 4, Article 14 (November 2010), 21 pages. DOI = 10.1145/1868358.1868361.
http://doi.acm.org/10.1145/1868358.1868361.

1. INTRODUCTION

Greenfoot is an integrated educational software development environment aimed at learning and teaching programming to young novices. The target user group starts at pupils from about 14 years of age, and also includes introductory university education.

Figure 1 shows Greenfoot's main window, with a scenario—Greenfoot's term for a project—open in the environment. The main part of the window shows the Greenfoot world, the area where the program executes. The world is of variable, user-defined size, and holds the scenario's objects.

On the right we see a class diagram that visualizes the classes used in this scenario and their inheritance relations. The two superclasses visible here (World and Actor) are part of the Greenfoot system and are always present.

Author's address: M. Kölling, School of Computing, University of Kent, Canterbury, UK; email: mik@kent.ac.uk.

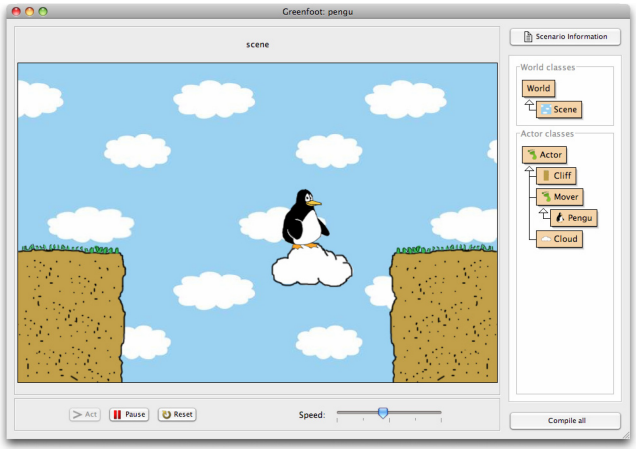

Fig. 1. The Greenfoot main window.

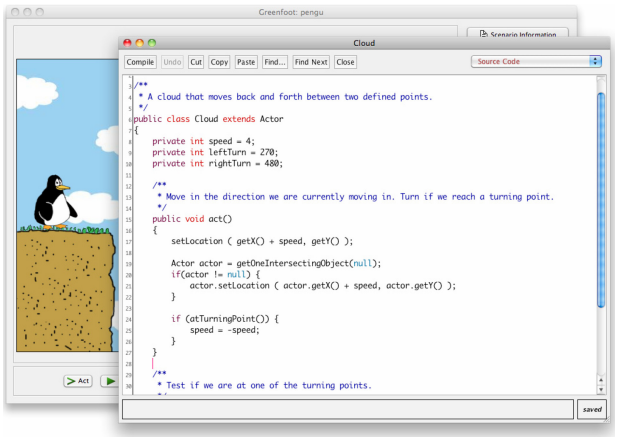

Fig. 2. The Greenfoot editor.

The subclasses (all others classes visible here) are part of this particular example and will vary with different scenarios. A scenario will always have at least one world subclass, representing the actual world (the rectangular execution area) used. It will also have one or more Actor subclasses. Actors are those objects that are present in the world and exhibit behavior to implement the scenario's objective.

Below the world view are some execution controls that allow running or single-stepping the scenario.

Double-clicking a class in the class diagram (or right-clicking to choose from a popup menu) opens a text editor, showing the class's source code (Figure 2).

The language used to program is standard Java—Greenfoot internally uses the standard Java compiler and the standard virtual machine (JVM) to ensure full conformance with current Java specifications. However, although the language is Java, the environment supports use of the language in simpler ways

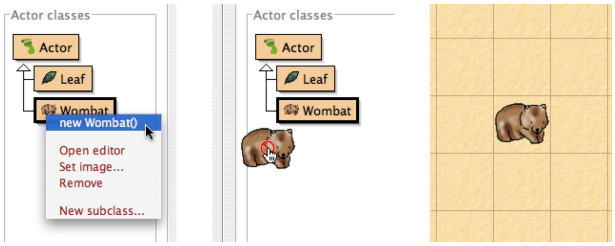

Fig. 3. a) Create object, b) Move to world, c) Place.

than normally available. For example, no static main method needs to be written, and programming can start with simple, one-line methods. An example of this is given in Section 3 below.

Once some code has been written for a class, the class can be compiled and objects can immediately be interactively created through a context menu on the class. There is no need to first complete a whole project, or even for all classes to compile. As soon as the world class and one of the classes representing an actor in the world compile, objects can be created and the behavior can be tested. There is no need to write test drivers or a main method.

All actors in the world have a method called act (inherited from the Actor superclass). Clicking the Act button in the execution controls invokes this act method once on each actor in the world. Using the Run button causes the act method to be called repeatedly, until the user pauses execution again. This is how Greenfoot scenarios are executed. Programmers simply define the behavior of every actor in the act method, and Greenfoot ensures that each actor gets called to act appropriately. Thus, students can concentrate on programming logical behavior of the actors and do not need to write graphics code. Greenfoot does the graphical animation of the actors implicitly.

2. INTRODUCING CONCEPTS

One of the goals of Greenfoot is a design that explicitly visualizes important concepts of object-oriented programming. While the source code editor has been mentioned in the previous section, students do not typically start by manipulating source code.

A typical first exercise for students might involve presenting them with a scenario that has already been implemented and compiled. Students then start by creating objects, which is achieved by selecting the object's constructor from the class's menu.

Once the object has been created, it can be placed into the world (Figure 3).

When the object is in the world, a right-click shows all the object's public methods (all available actions it can perform) in a popup menu (Figure 4).

Once a method has been selected, it executes and the effect is visible on-screen immediately. If the method expects parameters, a dialogue window pops up prompting users to enter the parameter values. Similarly, possible result values are displayed in a dialogue window.

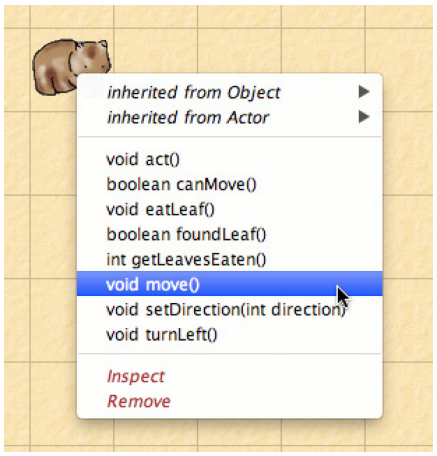

Fig. 4. An actor's context menu.

Of course, multiple objects can be created from the same class. Users can create several actors (wombats, in our illustration) and place them into the world. Methods can be executed on each of them individually.

The state of each actor can be examined by selecting the Inspect option from the actor's menu. This will open an object inspector window showing all internal instance variables and their values. This illustrates the independence of objects (all objects have different values) and their relation to their class (all objects of the same class have the same kinds of fields).

Pedagogically, these kinds of interactions are extremely valuable. They allow teachers to introduce some of the most important and fundamental concepts of object orientation in an easily understandable manner. These concepts include the following.

—A program consists of a set of classes.
—From classes, we can create objects.
—Multiple objects can be created from one class.
—All objects of the same class offer the same methods and have the same fields.
—Each object holds its own values for its fields (each object has individual state).
—We communicate with objects by calling one of its methods.
—Methods may have parameters. They also may return values.
—Parameters and return values have types.

All of these concepts are fundamental to understanding modern object-oriented programming, and they are traditionally very hard to teach. One reason for the difficulty in teaching them is that these concepts are very abstract. Greenfoot makes them concrete by creating tangible experiences, explicitly visualizing them and guiding users through interactions involving concrete manifestations

of these concepts. Students learn from these experiences without the need for much explicit instruction.

This enables the introduction of the fundamental concepts first, before the need to deal with syntax and source code. Students do not immediately get distracted from important concepts by having to worry about placement of semicolons and parentheses.

Of course, fairly soon (typically in the second lesson) we do get students to manipulate source code, but when they do this, they do so in the context of a conceptual framework of object orientation: They know that what they are doing is modifying or specifying the behavior of an object (or more precisely: a class of objects).

3. CODE EXAMPLES

Before discussing underlying design goals in more detail, we give a few short examples of typical Greenfoot source code to provide a feel for the level of complexity involved in writing Greenfoot scenarios.

When creating new actor classes, each class is assigned an image to represent it in the world. The image can be chosen from a built-in image library, or it can be specified by the user from their own available images. Objects created from the class start with their class's image as their default image.

Each actor in Greenfoot has three elements of state that are automatically visualized on screen: a *position* (specified in x,y-coordinates), a *rotation*, and an *image*. Each of these can be manipulated through method calls to create an animation effect.
For example, the call:

```
setRotation(90);
```

rotates the object to 90 degrees.
The method:

```
getRotation()
```

is an accessor method that returns the current rotation (in degrees, 0-359). Thus, we can quite easily make the actor rotate around its axis by writing the following code:

```
setRotation( getRotation()+2 );
```

This code segment reads the current rotation, adds a little bit to it, and sets the result as the new rotation. When this statement is added to the body of the act method, executing the scenario will cause it to be repeatedly called, resulting in continuous rotation of the actor.

When creating new actors, a simple but complete class template is automatically provided. This forms the starting point for students' work. It looks as follows:

```
import greenfoot.*;
/**
 * Write a description of class MyActor here.
```

```
 *
 * @author (your name)
 * @version (a version number or a date)
 */

public class MyActor extends Actor
{
   /**
    * Act - do whatever the actor wants to do.
    */
   public void act()
   {
       // Add your action code here.
   }
}
```

Providing this template follows a philosophy we first formulated during the design of the BlueJ environment [Kölling et al. 2003]: never start with a blank screen. Starting from a blank screen requires design, and is an advanced exercise. It is something students encounter later, but not as the first contact.

Students start their work by adding code to the method body of the *act* method. Adding the one line of code shown above, placing an actor into the world and clicking the Run button, results in an actor that rotates around its center point.

Similar effects can be achieved by changing the location. For example, the statement

```
setLocation( getX()+2, getY() );
```

has the effect of moving the actor across the screen from left to right.
Naturally, statements can be combined. The combination:

```
setRotation( getRotation()+2 );
setLocation( getX()+2, getY() );
```

combines movement and rotation, and results in an actor that rolls over the screen. Thus, students can create visible animated behavior with just one or two lines of code.

The Greenfoot API provides methods that make common tasks in programming simple graphical animations and games relatively easy. Following is an example to handle collision detection:

```
a = getOneIntersectingObject(Car.class);
```

This call will check whether the current actor intersects with another actor of class *Car* and return the intersecting object if there is one (returning null otherwise).

The Greenfoot API is defined in a total of five classes with a relatively modest number of methods. The complete API documentation can be printed on

two pages of standard letter size paper; it is also available from within the Greenfoot environment.

Students typically learn quickly to read this documentation and work with it for their projects.

4. DESIGN GOALS

A number of important design goals underpin the interaction design of Greenfoot. At the highest level, they can be summarized in two points, from two different perspectives:

(1) From the student's perspective, the goal is to make programming engaging, creative and satisfying.
(2) From the teacher's perspective, the goal is for the environment to actively help in teaching important, universal programming concepts.

When we investigate these design objectives further, each results in a number of more detailed goals that support and enable the top level aims.
The first goal, engagement of students, leads to the following sub-goals.

1.1 *Ease of use*. The system must be easy to use. It must allow students to achieve their goal without them getting stuck in unnecessary, mundane administrative tasks.
1.2 *Discoverability*. The system must be discoverable. Learners must be able to find out about functionality needed to achieve their goal.
1.3 *Support engaging functionality*. The system must allow to implement engaging, attractive functionality. This is possible in different ways, but certainly must include the easy use of graphics, animation and sound.
1.4 *Flexibility*. Since the judgement of what students might find engaging and exciting varies with age and with individual interest, the system must be flexible enough to support a wide variety of possible scenarios.
1.5 *Quick feedback loop*. The system must allow for quick and frequent success experiences. This means that it must support development in small steps with frequent opportunities at execution, observation, and visual feedback.
1.6 *Availability*. The system must be easily available (including cost) on a wide range of commonly used systems, on average hardware. Only then can learners play at home—an essential part of creative, explorative learning.
1.7 *Social interaction/sharing*. The system must support sharing and communication between learners. Social interaction is a strong driver for engaging in creativity.
1.8 *Extendibility*. It would be an advantage if the system can be designed to be extendable, so that it can connect to reasonable existing interests and systems. This may be use of existing hardware (such as standard game controllers) or software (such as connecting to servers on the internet, databases, or other functionality of interest).

The second high-level goal—supporting the teaching—leads to the following more specific requirements.

2.1 *Visualization*. Important concepts of the programming paradigm (in our case: object orientation) should be directly visualized.
2.2 *Interaction*. Interactions in the environment should illustrate programming concepts (e.g., instantiation, method calls).
2.3 *Consistent mental model*. All representation of principles in the environment must correctly reflect the underlying programming model. This allows students to draw conclusions from experimentation and observation, and to arrive at correct interpretations.
2.4 *Concepts before syntax*. Syntax should not be the very first hurdle that students have to cross to achieve their first success.
2.5 *Avoid cognitive overload*. Interaction in many modern development systems is at a level of complexity that makes learning difficult for beginners. Cognitive load theory tells us that the capacity for mental processing is limited [Miller 1956], and the number of simultaneous cognitive challenges strongly influences the ability to learn. To keep cognitive load at a manageable level, all extraneous complexity (this is: complexity not intrinsic to the task at the focus of the learning) should be avoided, all peripheral tasks automated and hidden.
2.6 *Support for teachers*. Teachers at secondary schools often do not have much time for the development of material or for off-site professional development and training. Thus, we need infrastructure to support the system that provides explicit support for teachers, such as discussion and sharing of teaching material.

These design goals have strongly influenced the interface and interaction design of Greenfoot. Below, we discuss selected characteristics of Greenfoot's functionality that result from our attempt at reaching those goals.

5. EASE OF USE

When thinking about ease of use, the most important considerations are not about the design of the elements of functionality that are included, but about what to leave out. Ease of use can only be achieved by restricting the scope of functionality, and design decisions in the ongoing tension between flexibility and simplicity have the highest impact.

Greenfoot, even though it makes use of a standard programming language, excludes many tools commonly found in more generic IDEs, such as version control, unit testing, refactoring, etc. The goal is that learners become familiar with the complete user interface within a few days.

Another area of leaving things out is in the design of the APIs. Greenfoot attempts to strike a balance between flexibility—giving users enough power to implement interesting functionality— and simplicity—restricting the number of methods in the API to an easily learnable set.

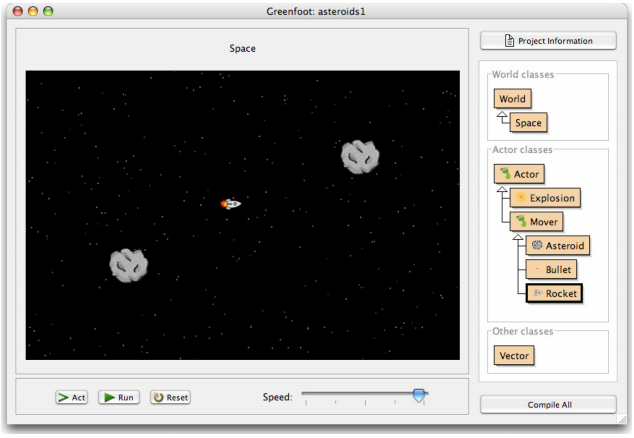

Fig. 5. An "asteroids" game.

6. FLEXIBILITY

At first glance, and looking only at one or two initial examples, Greenfoot appears to be a traditional micro-world (in the tradition of Turtle Graphics [Lukas 1972; Caspersen and Christensen 2000], Karel the Robot [Pattis et al. 1994] or the AP Gridworld Case Study [CollegeBoard 2010], built into an integrated development environment with a tight edit/compile/execute cycle. However, it is much more flexible than that.

Greenfoot can be seen as a micro-world meta framework that allows the quick and easy creation of specific micro-worlds. Existing systems, such as Turtle Graphics or Karel the Robot, can easily be implemented in Greenfoot, and Greenfoot goes much beyond the restrictions of those systems.

Traditional systems integrate the programming functionality (move, turn, etc.) with a specific scenario (a turtle with a pen, for example, or a robot collecting "beepers"[1]). As a result, if students start with robots and beepers, after some weeks of work they are still stuck with robots and beepers. If students get bored with this idea, or are not interested in robots in the first place, there is no easy way out for them.

Greenfoot separates these two aspects: The available programmed functionality is separated from a specific scenario, allowing the implementation of a wide variety of examples. This supports the goals of engagement and flexibility (goals number 1.3 and 1.4 above).

As a result, many substantially different scenarios can be created, covering many different topics. Examples include games (Figure 5), simulations (Figure 6 and Figure 7), music (Figure 8) and other visualizations (Figure 9).

The opportunity to cover widely different contexts with the same framework has a number of advantages. Examples and context can be tailored to different age groups or different interests, either by a teacher or individually by students themselves. If a learner is most interested in games, they can start

[1]"Beepers" are objects for robots to collect in the Karel the Robot micro-world.

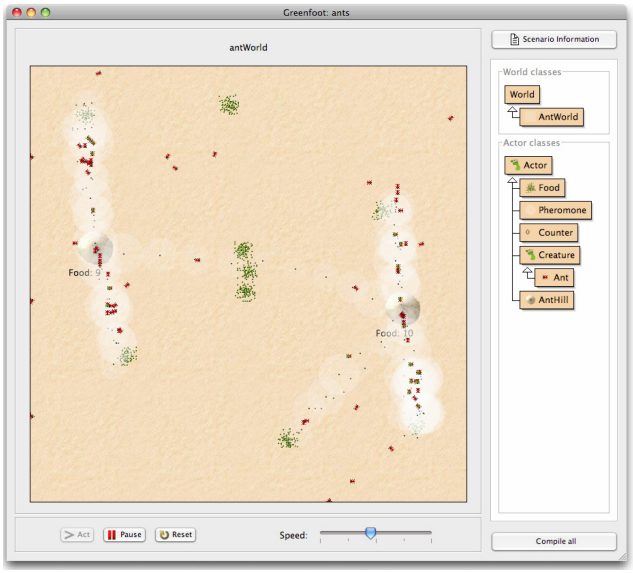

Fig. 6. A simulation of ants collecting food.

by building a game. If another student looks for a more useful application of computing, they can work on simulations with an environmental theme, for example. More scientifically minded users can work with biological, physics or chemistry-inspired simulations, and teachers can easily adapt examples to their context.

Some of the technical aspects that make this flexibility possible is a framework, embodied in the Greenfoot API, that does not specialize too strongly in a specific small subclass of simulation, and a world model with flexible resolution of the world cells.

As a result, many programs that use two-dimensional, animated graphics as their main user interface can easily be implemented in Greenfoot.

7. COMPLEXITY OF DEVELOPMENT

A typical method of using Greenfoot initially, with complete novices, is to provide students with a partially implemented scenario that they then modify. Student activities often start with executing and experimenting with the provided material, followed by implementing extensions of the existing functionality.

We have argued above that scenarios can easily be adapted to match the interests of students. For this to work, it must be feasible for teachers to develop these teaching scenarios themselves. Teachers at secondary school level typically do not have much time for development of material, and sometimes lack specific expertise in technical details. Thus, the necessary complexity of well-made, completed scenarios has a large influence on the feasibility of custom development of scenarios.

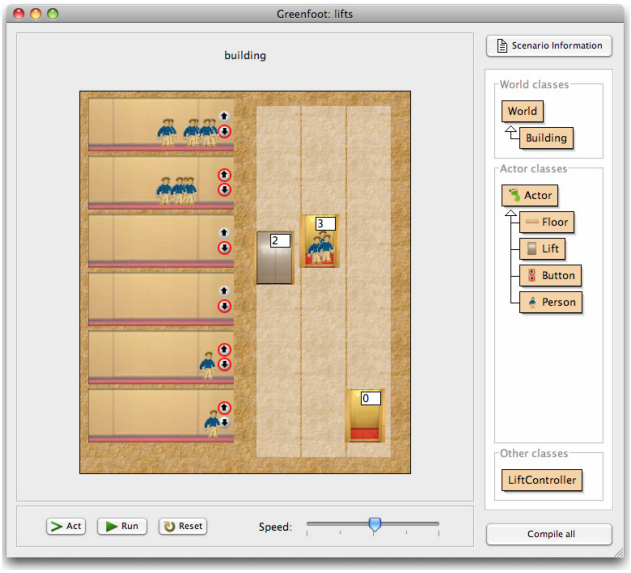

Fig. 7. A lift simulation.

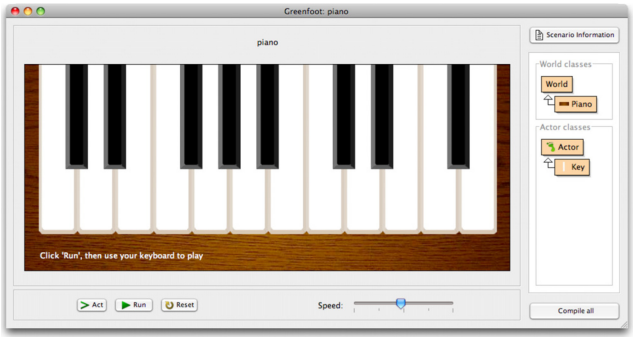

Fig. 8. An on-screen piano.

Creating interesting, good looking interactive animations and games in Greenfoot is relatively straightforward. A typical initial scenario, commonly used as a framework in the first few lessons, consists of about 40 to 70 lines of code. More complex and sophisticated completed scenarios typically consist of only a few hundred lines of code. For example, a full implementation of the Asteroids game (Figure 5) has about 620 lines of code, while a sophisticated ants simulation (Figure 6) consists of about 490 lines of code.

This is a scale and level of complexity where many teachers are able to develop these scenarios themselves. Of course, not every teacher needs to develop new scenarios. The more common case for most teachers is that they browse through a pool of existing teaching scenarios and choose one that suits their needs. However, enough teachers are in a position to develop scenarios that

Fig. 9. A weather map, with real data from the Internet.

such a pool can be created and built up. What is necessary for this to work is a mechanism for sharing these scenarios among the teaching community. This is discussed further in Section 11, below.

The development of prepared teaching scenarios is only necessary in the early stages of learning. After some time (depending on age, engagement, and talent, some days, weeks, or a small number of months) learners will be able to design and develop entire new scenarios from scratch without needing the scaffolding to get started.

The initial starting scenarios are typically provided by a teacher if the learning takes place in a classroom. For self-directed study, teaching material is provided online, with references provided directly in the Greenfoot environment, so that learners can work independently. The material includes both educational scenarios and tutorials. This, too, is discussed further in Section 11.

8. TARGET AUDIENCE

The target audience of users of Greenfoot includes pupils from about 14 years of age, and scales up into introductory university education. (Age, in this context, should be seen as a rough approximation. We have seen Greenfoot successfully used with classes of 13-year-olds—what is needed is a level of literacy and maturity that individual children will naturally reach at varying stages in their development).

While the output of Greenfoot scenarios is graphical, its programming interaction is textual. This imposes a higher hurdle of competence than that needed for Scratch [Maloney et al. 2008] or Alice [Dann et al. 2008].

Users of Greenfoot may be complete programming novices within that age group, or they may have prior programming experience in Scratch, Alice, or comparable systems. Concepts learned in Alice and Scratch transfer well into Greenfoot. Where concepts correspond, they are presented in a similar manner, supporting a similar mental model, while Greenfoot introduces some additional concepts that build on the material learned in those systems. This is discussed further in Section 15.

The purpose of using Greenfoot with novices may be similar to that of using Scratch and Alice—generating interest and enthusiasm in programming (and

maybe computer science in general) and conveying fundamental programming principles—but can also be more formal. Higher level software development concepts, such as good class design, separation of concerns, encapsulation and cohesion, coupling, and additional levels of abstraction can also be taught.

9. JAVA AS A LANGUAGE

The programming language used in Greenfoot is standard Java. This has a strong effect on many of the characteristics of Greenfoot—some of them beneficial, others imposing limitations and complexities.

Firstly, the use of a textual language in general (as opposed to the Scratch/Alice model of drag-and-drop instruction blocks) is considerably harder for young learners to master. This imposes a minimum maturity level. While Scratch and Alice can be used with very young kids, Greenfoot does not usually work for pre-teens.

This is mostly a result of the language, not the conceptual model of Greenfoot. Informal experiments with Greenfoot and 10-year-olds have shown that learners of that age can understand the underlying model and concepts quite easily, but struggle with the syntax of the Java programming language to a degree that they cannot progress on their own.

The second aspect is the use of Java specifically. Other textual languages are available that are easier to master. Here, the main problems are not with the object-oriented nature of Java (as some teachers sometimes speculate), but with the choice of syntax. The concepts of object orientation seem to impose no conceptual problems even for learners at the lower boundary of our target age group. The mental model of independently acting and communicating objects, the concept of classes and even the idea of inheritance seem to pose no overly difficult challenge. The syntax of Java however, with its nested textual block structure coupled with poor error messages, presents a hard challenge for young novices.

Using Java in Greenfoot is considerably easier than using Java in a standard professional environment (be it a text editor/command line environment or a standard professional IDE). Since actors can be created independently and interactively and methods called individually, no testing framework or main method is needed. This, coupled with the automatic provision of simple class templates, leads to the ability of starting by writing and executing single lines of code, with immediate visual feedback. This makes Java in Greenfoot suitable for the target age group, but the Java syntax still provides a problem for younger children.

On the positive side, using a standard, industry strength programming language has a number of advantages. Firstly, Greenfoot scales well into older age groups. More sophisticated and technically demanding ideas can be implemented than is possible in systems with custom-made educational languages. Language constructs and libraries are available for advanced topics.

Secondly, performance is good enough to develop and run fairly complex scenarios with hundreds of objects.

Thirdly, a wealth of infrastructure is available for Java, including teaching material, textbooks, and technical development to port the virtual machine to new platforms, improve performance, and many other ongoing improvements. Greenfoot can tap into those, making many improvements to our platform easier than it would be if we had to develop these elements from scratch. Some of these benefits are discussed in more detail in the next section.

10. EXTENDABILITY AND PERFORMANCE

The fact that Greenfoot uses Java, while presenting problems for young learners, has benefits at the older end of our target age group. In effect, there is no age limit at the upper end.

Since Greenfoot supports the full Java language, including use of all libraries, very sophisticated scenarios can be created. And since execution is carried out on the standard Java VM—a highly optimized professional platform—performance is good for many cases. Scenarios such as the ants simulation, with several hundred objects on screen, each performing collision detection against hundreds of other objects, perform smoothly without problems on typically available medium range hardware.

Educationally, this means that Greenfoot can easily be used in introductory university courses. All concepts that are commonly introduced there have a natural representation in Greenfoot. Greenfoot is also occasionally used in advanced university agent modelling and AI modules. In these cases, teachers are mostly interested in the ease of creating graphical animations of the agents' behavior, while the actual implementation of each agent class may be highly complex.

For motivation of learners, this means that they do not easily hit a brick wall where their ambition outstrips the capabilities of the programming system. With systems based on simpler educational languages and graphical code editing, especially older teenagers often reach a point where their ideas cannot easily be realized within that system. The problem might be in the expressiveness of the language or in performance. At that time, graduating to Greenfoot can give them a welcome path forward. The price we pay in the added complexity for early beginners pays off when students reach a level of competence and ambition to aim for more advanced and demanding projects.

In addition to performance, the Java infrastructure also provides a wealth of specialized functionality through its standard libraries. More advanced users can connect to databases, use networked communication, build multi-user scenarios, communicate with mobile phones, and integrate any other functionality that is supported by a Java library.

Lastly, the use of a standard language commonly used in programming education eases transfer out of the system into more general development environments. Since all educational IDEs are interim tools—the goal is, after all, to motivate students to learn more, and move on to other systems—transfer out of the system should not be neglected. While we often discuss the challenges of getting started in a system, moving out poses its own set of challenges.

Ideally, concepts learned in an educational IDE carry over smoothly into future systems.

The use of Java in Greenfoot makes this connection more direct and immediate. Progression to the BlueJ IDE [Kölling et al. 2003] is specifically well supported. BlueJ was designed by the same development team as Greenfoot, and consistency of representation of the same concepts in the two environments is an explicit design goal. Thus, transferring between the two, including parallel use in a course with frequent switches between them, is easily possible. We have used this approach in our own introductory programming course at university level with good success.

Transfer to other IDEs using class-based, object-oriented languages is also fairly straightforward, as all concepts encountered in Greenfoot should easily transfer. As such, Greenfoot may provide an ideal stepping stone between Scratch or Alice on one hand, and more general IDEs on the other.

11. ECOSYSTEM

Greenfoot is supported by the usual collection of material, including a Web site [Greenfoot 2010], tutorials [GF Tutorial 2010], and a full textbook [Kölling 2009].

In addition, two of the design objectives of Greenfoot were to provide social interaction and sharing for learners (listed as design goal 1.7 in Section 4) and support for teachers (goal 2.6). Both of these require infrastructure beyond the immediate development environment.

11.1 The Greenfoot Gallery

For students, two paths of social interaction are available. The first is an online discussion forum, *Greenfoot Discuss*, provided as a group on the Google Groups platform.[2] Here, learners can ask for help, ask technical questions, and discuss ideas.

The more interesting path is the Greenfoot Gallery [Gallery 2010]. This is a Web site where Greenfoot scenarios can be published, and other people can execute those scenarios and leave ratings and comments for them. Greenfoot scenarios run directly on the site within the Web browser. Scenarios can be published to the Greenfoot Gallery directly from within the Greenfoot environment with just a few mouse clicks.

To publish scenarios, comment or rate them, users must create an account on the Greenfoot Gallery site. Simply playing the scenarios is possible without an account.

The Gallery serves as a highly motivating element in the Greenfoot ecosystem, with frequent discussion threads attached to scenarios where other users provide feedback, report errors or suggest improvements [Fincher et al. 2010]. As a result, students in organized teaching courses often spend additional time after an assignment has finished improving and extending their projects.

[2]Available at http://groups.google.com/group/greenfoot-discuss.

11.2 The Greenroom

The second Web-based system external to the Greenfoot environment itself—but linked into its infrastructure—is the Greenroom [Greenroom 2010]. The Greenroom is an online community site exclusively for teachers. It provides functionality for discussions, resource sharing, and news announcements.

The Greenroom allows new teachers adopting Greenfoot to ask questions, get advice, and find teaching material. Existing resources include worksheets, test questions, assignment and project ideas, handouts, slides, teaching scenarios, written tutorials, and tutorial videos.

The Greenroom has so far avoided the fate of many other resource repositories—quickly waning interest after a short burst of initial use. It has proven highly popular, with more than 800 teachers subscribed, many active discussions, dozens of resources, and daily activity.

12. STRENGTHS

For the purpose of comparison with Alice and Scratch, we discuss three of the main strengths of the Greenfoot system.

(1) Good illustration of OO concepts.
(2) Good scaling up (number of objects, performance).
(3) Relatively simple start.

We will discuss each of these in more detail.

12.1 Good Illustration of OO Concepts

While students play with creating games or simulations, they are implicitly observing and interacting with many major concepts of object-oriented programming, as present in many modern OO programming languages. Concepts that are implicitly illustrated include the following.

—A program consists of a collection of classes.
—From classes, we can instantiate objects.
—One class can create many objects.
—Objects have separate state.
—We can communicate with objects via method calls.
—Methods may have parameters and return values.
—Several more.

These are concepts that are traditionally hard to learn and teach. In Greenfoot, these become implicitly apparent without much verbal explanation, and they transfer well to other widely used programming languages which students may transfer to later.

12.2 Good Scaling Up (Number of Objects, Performance)

Greenfoot scales well when students become more competent and want to create more sophisticated programs. The creation of many objects is not a problem, since programmatic instantiation is part of the language. Thus, games or simulations with many objects (traffic simulations, ant simulations, etc.) can be created easily. This enables the creation of scenarios that display true emergent behavior, which is highly motivating. The performance is good (scenarios with several hundreds of objects, all checking collisions between each other, run smoothly). Also, the user language is standard Java, and any Java library can be used in Greenfoot. This means that very sophisticated algorithms can be written (e.g., AI algorithms for actor behavior) and access to external technologies (databases, network access, external devices such as game controllers) can be implemented.

12.3 Relatively Simple Start

In contexts other than comparing with Scratch and Alice, we claim that it is easy to get started in Greenfoot. Early examples are very easy to write and interesting results are quick to achieve. In comparison with most programming systems, especially producing animated graphics is very easy in Greenfoot. In comparison with Scratch and Alice, Greenfoot gets beaten by those systems in terms of ease of entry. However, in context of its design goals—use of a standard programming language, ability to scale up to sophisticated programs in scope and performance—it provides an extremely easy path of entry for young programmers.

13. LIMITATIONS

As with all systems, there are also clear weaknesses and limitations—situations where Greenfoot is not a good solution. They include:

(1) Tinkering by younger kids.
(2) Error handling and reporting.
(3) 3D.

13.1 Tinkering by Younger Kids

The use of Java as the user language brings with it an unavoidable minimum of complexity. Java is a text-based, syntax-oriented language that requires a certain level of conceptual and linguistic maturity to cope with its structure. Younger children often get caught in structural syntax errors (such as unmatched scope brackets) which they find hard to diagnose and fix. Exploration and tinkering—which we highly value and encourage—requires reading of textual documentation of APIs, which requires an ability and willingness to work with written documentation or sample programs. All of these impose a fairly hard minimum of required maturity, which many children reach around the age of 14.

13.2 Error Handling and Reporting

Another side effect of using Java as the language is the comparatively poor quality of error messages. Firstly, syntax errors are common, as is the case in most text-based programming languages, and every programmer knows the deep feeling of frustration that occasionally sets in when struggling with errors over some period of time. Secondly, the messages used to report the errors are often not especially helpful, and interpreting the error messages is a skill that needs time to acquire. This occasionally wastes time in a non-productive state and has the potential to frustrate learners.

13.3 3D

Greenfoot is designed as a 2D system. All APIs are aimed towards supporting 2D animation. It is possible to create 3D scenarios, as several impressive projects submitted to the Greenfoot Gallery have demonstrated. However, doing so is hard work, and Greenfoot offers little specific support for making this easier than it is in standard Java. The existence of these projects in the Gallery, however, shows that there is an interest—at least for some users—to create those kinds of games. This is not well supported in Greenfoot.

14. EXPERIENCE

Greenfoot has been used extensively at high school level, at colleges and in introductory university courses. Lengths of courses stretch from single-day outreach activities with novices (sometimes as short as two hour sessions) to full semester formal modules.

A large amount of feedback and anecdotal data exists; however, no formal study of the effects of teaching Greenfoot has been carried out. Thus, firm conclusions are hard to draw.

Some of the data that is available is interesting nonetheless and provides a basis for further work on the system. It may also form a possible basis for research questions for more formal studies of the effectiveness of Greenfoot in learning situations.

Accurate user numbers are not collected for Greenfoot, but some lower boundaries can be established. The Greenroom has, at the time of writing, over 800 members, almost all of which are instructors using Greenfoot. About two thirds of these are at a secondary school, and most of the remainder are at college or university. Since its first release in mid 2006, Greenfoot has been downloaded more than half a million times from our Web site.

Feedback from teachers often emphasises motivation and engagement of students:

> "Greenfoot worked really well with my students, I was very impressed how absorbed they were."

> "Greenfoot is a brilliant instructional tool, and it's setting off sparks in my student's heads like I've never seen before."

> "My students love Greenfoot! They worked harder than ever on this challenge."

> "We are having a great time working with Greenfoot. The students can relate to the scenario and are anxious to add additional capabilities."

Another aspect regularly mentioned in user feedback is ease of use. Greenfoot seems to succeed in providing fewer stumbling blocks than other systems teachers had experience with.

> "I've been enjoying both the system and the book; it's great to see my students making real progress on their projects, free from the normal development obstructions and roadblocks."

The Greenfoot system (software and material) also seems successful in providing help for less experienced teachers.

> "I had my first class yesterday, and having done no programming whatsoever before, I was worried that I would not be able to get into it, however using Greenfoot to program the 'little crab' has reassured me, and I feel that I will be able to do this well."

At high school level, Greenfoot is regularly used both in curricular instruction and in extracurricular activities, such as after-school clubs.

At university level, most use is in introductory programming courses. Here, it is sometimes used for an initial part of the first semester (e.g., the first six weeks), and sometimes throughout the semester in parallel or alternating with other environments (this pattern seems to work especially well with BlueJ, because of the commonalities in interface and programming model). Universities also use Greenfoot for school outreach and engagement activities.

15. RELATION TO SCRATCH AND ALICE

Greenfoot is a teaching system designed to make the creation and understanding of computer programs easier. In the taxonomy presented by Kelleher and Pausch [Kelleher and Pausch 2005], it is grouped in the Teaching Systems/Mechanics of Programming category, and within this, covers the following sub-categories: Simplify Typing Code, Making New Models Accessible, Tracking Program Execution, Make Programming Concrete and Models of Programming Execution.

Greenfoot shares many design objectives and teaching philosophies with Scratch and Alice, such as the encouragement of creative exploration, tinkering, experimentation, discovery, and social interaction. All three systems make use of graphics and sound as motivators, while teaching fundamental programming concepts.

The main target age groups differ. Both Scratch and Alice can be used with younger children than Greenfoot, while Greenfoot scales better for more proficient users. Scratch has a target user group almost entirely complementary to

Greenfoot (8- to 16-year-olds, to Greenfoot's 14+), while Alice aims at a broader range from 12 to about 19.

Greenfoot can follow on well from either Alice or Scratch. However, because of the good complementary fit in target age group, as well as the common focus on 2D graphics, a sequence of Scratch-to-Greenfoot is especially interesting.

Concepts between those systems transfer well. Scratch has a simpler object model without classes which allows a very quick and easy start, but has drawbacks for some kinds of more advanced projects, especially those involving many objects of the same kind. This is an ideal opportunity to motivate a switch to Greenfoot once learners reach a stage where they aim for more ambitious projects. Concepts learned in Scratch can then be generalized and extended. The move to Greenfoot's text-based editing and class-based object model adds complexity, but the functionality gained in return provides more power for the programmer. This presents a well working, realistic trade-off in a move to a more powerful system as learners progress in their programming ability. Some resources exist in the Greenroom to explicitly facilitate this transition from Scratch to Greenfoot.

Alice's object model is, like Greenfoot's, based on Java[3], and the Alice-equivalent Java code can be produced and edited. Thus, Alice's mental model is very close to that of Greenfoot. This would allow smooth transition between the two systems. However, the Alice team also propose use of Alice at introductory university level, covering much of the same age group that Greenfoot is aimed at. Here, the switch into Greenfoot is not quite as obvious an opportunity as with Scratch, since continuing in Alice is a possible alternative. The choice in this case comes down to preference of teachers between different kinds of systems.

16. CONCLUSION

Greenfoot is a system straddling the balance point between relatively untrained programmers and a professional programming language. It is designed for users with no or little programming experience, but aims at providing practice with standard programming concepts and techniques used in a current mainstream language. This is intended to make adoption in schools and universities easier, since traditional curricula can easily be taught using it.

Greenfoot aims at making the use of the standard language, Java, easy by providing a custom-designed environment that removes much of the complexity commonly associated with object-oriented programming. At the same time, it adds functionality to easily create graphics, animation and sound, so that engaging examples can be treated early.

Using Greenfoot, young learners can easily create simple games and simulations while learning fundamental programming principles. Feedback from users appears to indicate that the goal of engaging learners through the use of these kinds of examples is successful. Greenfoot can be used as a first programming system for learners in the mid-teens or older, or as a second system

[3]As of Alice version 3.0.

after outgrowing environments aimed at younger learners, such as Scratch or Alice.

ACKNOWLEDGMENTS

Greenfoot is a team project, built by a group of people over the last four or five years, and several current and past members of our research group have played crucial roles in its development. People who were closely involved in the design and development of Greenfoot include Poul Henriksen, Davin McCall, Bruce Quig, Neil Brown, Phil Stevens, Marion Zalk, Ian Utting, and John Rosenberg.

REFERENCES

CASPERSEN, M. E. AND CHRISTENSEN, H. B. 2000. Here, there and everywhere – On the recurring use of turtle graphics in CS1. In *Proceedings of the 4th Australasian Conference on Computing Education (ACE'00)*.

COLLEGEBOARD (ADVANCED PLACEMENT PROGRAM) 2010. GridWorld case study. http://apcentral.collegeboard.com/apc/public/courses/teachers_corner/151155.html (accessed 5/10).

DANN, W. P., COOPER, S., AND PAUSCH, R. 2008. *Learning to Program with Alice* 2nd Ed. Prentice Hall Press.

FINCHER, S., KÖLLING, M., BROWN, N., STEVENS, P., AND UTTING, I. 2010. Repositories of teaching material and communities of use: Nifty assignments and the greenroom. In *Proceedings of the 6th International Workshop on Computing Education Research (ICER'10)*.

GALLERY 2010. Greenfoot Gallery. http://greenfootgallery.org/ (accessed 8/10).

GREENFOOT 2010. Greenfoot – The Java object world. http://www.greenfoot.org/ (accessed 8/10).

GF TUTORIAL 2010. Greenfoot tutorial. http://www.greenfoot.org/doc/tutorial/tutorial.html (accessed 8/10).

GREENROOM 2010. Greenroom. http://greenroom.greenfoot.org/ (accessed 8/10).

KELLEHER, C. AND PAUSCH, R. 2005. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv. 37*, 2, 83–137.

KÖLLING, M., QUIG, B., PATTERSON, A., AND ROSENBERG, J. 2003. The BlueJ system and its pedagogy. *J. Comput. Sci. Educ. 13*, 4.

KÖLLING, M. 2009. *Introduction to Programming with Greenfoot – Object-Oriented Programming in Java with Games and Simulations*. Pearson Education.

LUKAS, G. 1972. Uses of the LOGO programming language in undergraduate instruction. In *Proceedings of the ACM Annual Conference (ACM'72)*.

MALONEY, J. H., PEPPLER, K., KAFAI, Y., RESNICK, M., AND RUSK, N. 2008. Programming by choice: Urban youth learning programming with scratch. In *Proceedings of the 39th Technical Symposium on Computer Science Education (SIGCSE'08)*.

MILLER, G. A. 1956. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychol. Rev. 63*, 2, 81-97.

PATTIS, R., ROBERTS, J., AND STEHLIK, M. 1994. *Karel the Robot: A Gentle Introduction to the Art of programming* 2nd Ed. John Wiley & Sons.

Received September 2010; accepted September 2010