Sofia.Micro: An Android-Based Pedagogical Microworld Framework

Brian L. Bowden

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Applications

May 2, 2014

Blacksburg, Virginia

Keywords: Android, Microworld, Sofia, event-driven, CS 1, Java, programming assignment, introductory programming

Sofia.Micro: An Android-Based Pedagogical Microworld Framework

Brian L. Bowden

(ABSTRACT)

Microworlds are visual, 2D, grid-based worlds with programmable actors that help ease students into programming. Microworlds have been used as a pedagogical tool for teaching students to program in an object-oriented paradigm for several years now. With the popularity of Android smart phones, creating a pedagogical microworld for Android can help students learn not just Java, OO and event-driven concepts, but also learn to use the Android framework to create concrete, real-world applications. This thesis presents Sofia.Micro, an Android-based pedagogical microworld framework that not only allows Greenfoot-style microworld programs to run on Android, but also adds additional functionalities to microworlds that have not been previously explored, such as built-in shape and physics support, event-driven programming in a microworld context, and allowing for both Greenfoot-style actors and Karel-style actors in the same world.

Acknowledgments

I would like to thank Dr. Edwards for helping me with my thesis and for his assistance during grad school. I would also like to thank my friends and family for their support during my time in grad school.

Contents

List of Figures

Chapter 1

Introduction

1.1 Microworlds

Many universities have been using the objects-first strategy in teaching students programming for introductory courses for several years now [19]. This approach teaches students to focus on object-oriented concepts from the beginning when first learning to program, such as inheritance, objects, and polymorphism [19]. However, students have to overcome several hurdles when initially learning programming. To help students overcome some of these difficulties, microworlds have been developed as a pedagogical tool to ease students into programming.

Microworlds are visual, 2D, grid-based environments with objects that have programmable behavior and their goal is to help students think in an algorithmic manner. Microworlds typically start students off with small programs that have simple objects with well-defined behaviors so students can focus on writing the logic of the object's behaviors in the worlds. This helps emphasize teaching students to think in a step-by-step methodology and focus on correctly writing out their logic.

Another issue is that students learning programming struggle with the fact that many concepts in programming are too abstract. The idea of the state of variables, as well as basic data structures prove to be a challenge for students who have a hard time conceptualizing what is occurring in a program [30]. Microworlds aim to make programming more concrete and interesting by visually representing the state so students can see what is happening in their program [30]. Since students can see what is happening in their program, they can easily detect when a given object does not act correctly and can check their logic to see where the problem occurs [30]. Visualization also helps prevent the common habit of trial-and-error programming that introductory students tend to exhibit where they just try changing random segments of code to fix a bug without thinking through the logic clearly [30].

1.2 Sofia

Android, a framework for mobile devices that uses Java and is built on the Linux kernel, has recently emerged as a popular framework for both teaching and software development [14]. Android has shown to be an effective motivator for students due to its ability to allow programmers create more concrete and ultimately, interesting, applications. However, learning Android has proved to be difficult for introductory programmers because the tools were designed for professionals [14]. Android not only retains many of the disadvantages a student would encounter when learning Java, but also adds extra complexity in learning the Android framework as well.

In order to make Android more beginner-friendly, Sofia, the Simplified Open Framework for Innovative Android Applications, was developed as a framework built on top of the Android API to make programming easier and cleaner for both beginners and experts [14]. Sofia's primary goal is not to just be a framework used by students for introductory courses, but to make Android programming easier and cleaner for both beginners and experts [13]. This design goal avoids the drawback of many programming pedagogical frameworks where students eventually hit a ceiling where the framework becomes unusable and students have to stop using the framework altogether [13].

Sofia uses a unique event dispatch model to eliminate much of the glue code that is normally required when writing GUIs [15]. Sofia also supports 2D shapes with built-in physics provided by JBox2D and animation support. Sofia makes it easier to switch between different activities and also makes it less cumbersome to reference widgets in a screen [14]. However, even with Sofia eliminating some of the clunkier portions of writing Android applications, the Sofia framework still is not suitable for introductory CS courses and is more suited for students who have some experience with programming.

1.3 Problem Statement

This thesis will address two main issues. First, we want to bring microworlds to Android. We would like to be able to use Android as a learning tool for introductory students because it can be a great motivator due to students being able to create real-world applications that others can see. Android emphasizes event-driven programming, which has shown to be an effective programming paradigm for teaching students by Kim Bruce et. al in several of their works [20, 21, 22, 23]. However, Android is too complex for most beginner students since it retains many of the drawbacks of learning Java plus the framework.

Second, microworlds do not retain their usefulness as a teaching tool throughout an entire CS 1 course. This is largely because students not only quickly master the tools given to them, but the problems given to students are only useful for solving early beginner problems [30]. More recent microworld frameworks, such as Greenfoot, have allowed students and instructors to create their own scenarios within the framework [7, 24]. We would like for students to be able to create varied and interesting applications within the microworld that would not only be engaging for students, but would also be complex enough to where the students would not quickly outgrow the framework and would last a full CS 1 course. This framework would also ideally allow for a natural progression from a microworld environment to a mobile environment.

1.4 Solution

We propose Sofia.Micro, a pedagogical tool for creating microworlds that extends the Sofia framework to help students learn Android programming in an easy-to-use, Greenfoot-style environment. This microworld framework aims to simplify the Android framework so that introductory level students can reap the benefits of Android and to also modernize microworlds in general by adding additional features that have not been used before in a microworld environment so that they can span a full CS1 course. Sofia.Micro focuses on event-driven programming, which helps emphasize the strengths of the Android framework. Sofia.Micro supports both the cellular-automata-style actors of Greenfoot and the programmable-actor-style of Karel actors, even in the same world. Finally, it supports having non-actor shapes in the world as well, and both actors and shapes support physics interactions with each other.

This thesis is broken down in the following manner: Chapter 2 covers related work and discusses previous microworlds in more detail. Chapter 3 covers the features of Sofia.Micro with Chapter 4 discussing the design and implementation of these features. Chapter 5 presents some case studies to demonstrate the capabilities and ease of use of Sofia.Micro features, with Chapter 6 covering concluding remarks and future work.

Chapter 2

Related Work

One issue microworlds try to address is that students have difficulty learning and applying the syntax of a given language. For example, students who are picking up Java as their initial language may have to write their own main method and class for the typical hello world example, such as:

```java
// HelloWorld.java
public class HelloWorld
{
   public static void main(String[] args)
   {
      System.out.println("Hello World!");
   }
}
```

Immediately, this introduces students to several key ideas in Java, such as classes, visibility modifiers, static keyword, return types, arguments to a function, and arrays. One major difference in microworlds is how they address students learning syntax of a language. Some microworlds, such as Karel, avoid this issue by using their own language or set of commands (ex `Move()`, `TurnLeft()`, `PickBeeper()`, etc.) so that the students are introduced to exactly what the microworld wants without students having to worry about all the extra syntax that they will not need until later. The downside to this approach is that the students will eventually have to 'discard' this language and relearn a new language. Some microworlds, such as Greenfoot, instead introduce students to Java, albeit with more emphasis on logic rather than syntax. This makes the initial learning curve steeper for students, but it makes the transition from a CS1 course much easier. Sofia.Micro follows the Greenfoot style more as it uses Java classes and syntax since we want Sofia.Micro to be a natural transition to writing mobile applications.

There have been a number of different microworlds created since Karel's inception in 1981. Different microworlds have different maturity requirements, with Alice and Scratch being suitable for middle-school aged students, and Greenfoot and Karel being suited for high-school and even collegiate level students [25, 33]. Since Sofia.Micro is based on Greenfoot and Karel and also uses the Android framework, we feel it would be best suited for high-school and collegiate level students.

2.1 Karel

Karel was the first microworld, released in 1981, and started the idea of using 2D grid-based worlds with programmable actors to help teach students object oriented concepts. The 2D grid used to represent the world has the origin in the bottom-left corner and starting at (1, 1) to allow for using coordinates similar to the top-right quadrant in the Cartesian plane [8, 31]. In this world, the vertical coordinates are called 'streets' and the horizontal coordinates are called 'avenues' [8, 31]. Actors in this world are represented as: robots, which the students write behaviors for; walls that the robot can not move through which are placed by the user and also on the bottom and left edge of the world; and black dots called 'beepers' that the

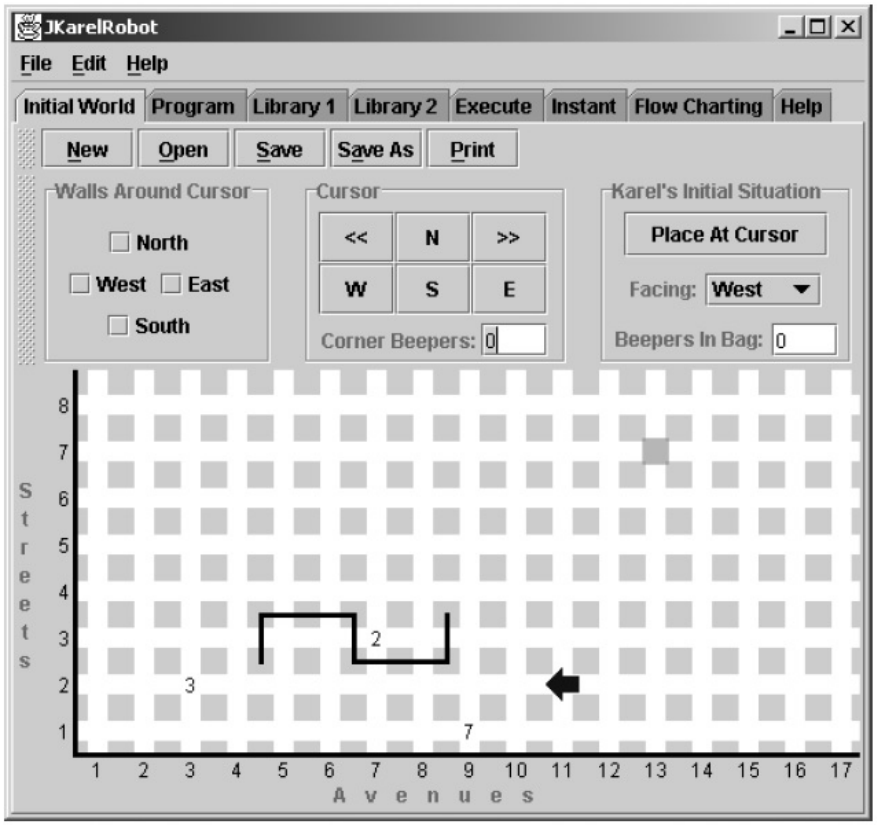
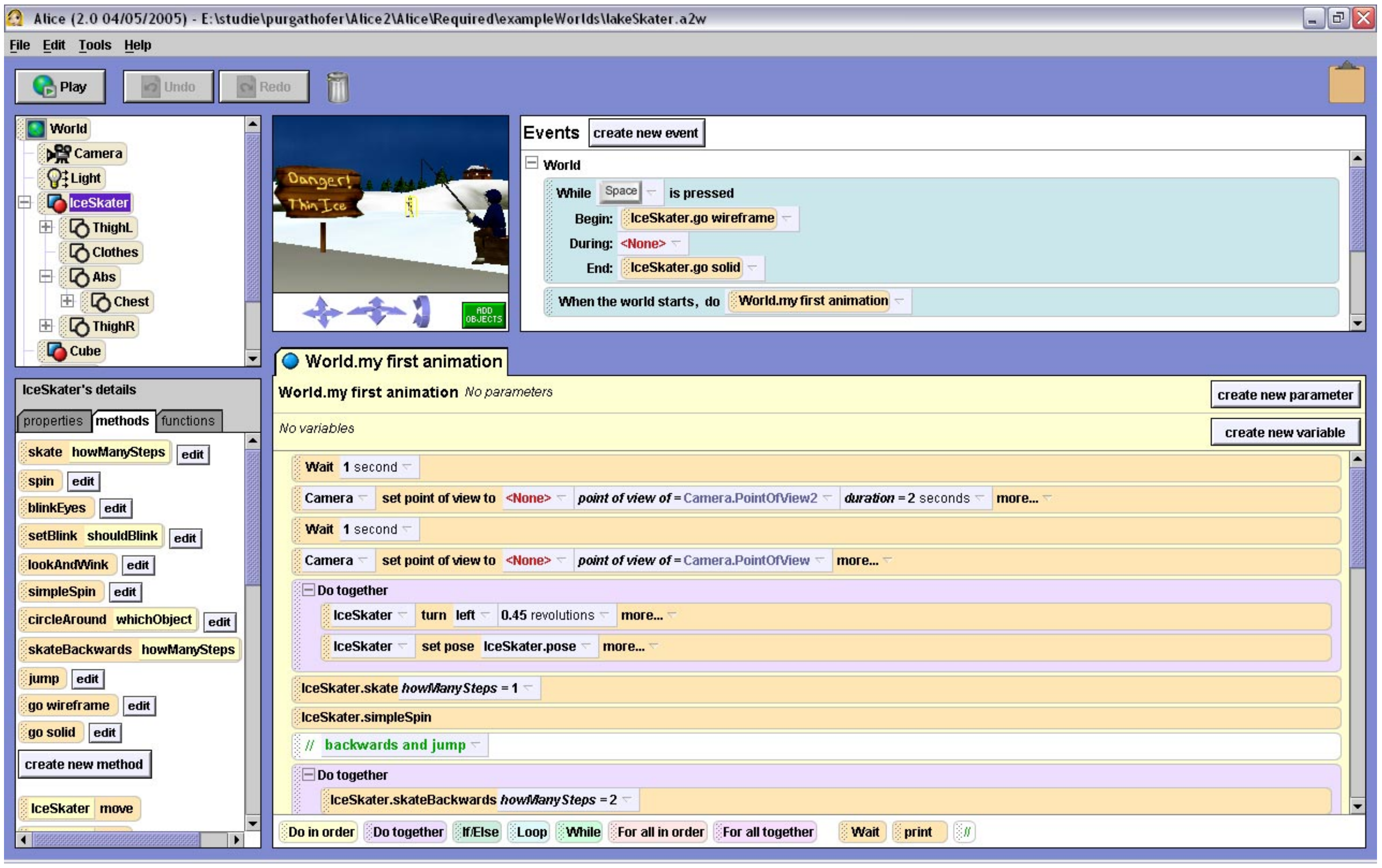


Figure 2.1: Screenshot for JKarel [31]

robots can sense and pick up [8, 31]. The students program the robots to move and interact with the environment by picking up beepers and avoiding walls [8, 31]. The student writes the entire program for the robot with each step in the program is performed during each tick of the global clock [31].

One of the drawbacks of Karel is that there is not much flexibility to the students since students will only ever be writing behaviors for the robots in the 2D grid world. This limits its usefulness as a long term learning tool since Karel can only be used to teach basic logic and object-oriented concepts to students and students would generally quickly outgrow Karel.



```
World
  While Space is pressed
    Begin:  IceSkater.go wireframe
    During: <None>
    End:    IceSkater.go solid
  When the world starts, do World.my first animation
```

```
World.my first animation No parameters
No variables
Wait 1 second
Camera set point of view to <None> point of view of = Camera.PointOfView2 duration = 2 seconds more...
Wait 1 second
Camera set point of view to <None> point of view of = Camera.PointOfView more...
Do together
    IceSkater turn left 0.45 revolutions more...
    IceSkater set pose IceSkater.pose more...
IceSkater.skate howManySteps = 1
IceSkater.simpleSpin
// backwards and jump
Do together
    IceSkater.skateBackwards howManySteps = 2
```

Figure 2.2: Screenshot for Alice IDE

2.2 Alice

Alice is a 3D interactive animation program released in 1999 [2]. Alice differs from Karel in that it allows students to drag and drop actions and observe an avatar perform those actions rather than writing the code themselves [2]. Students can drag and drop actions in this manner to create a story or an interactive game [2]. This approach to programming allows for easy experimentation by the students and for them to use their own creativity. One of the important ideas with Alice (along with Scratch) is that students can not write

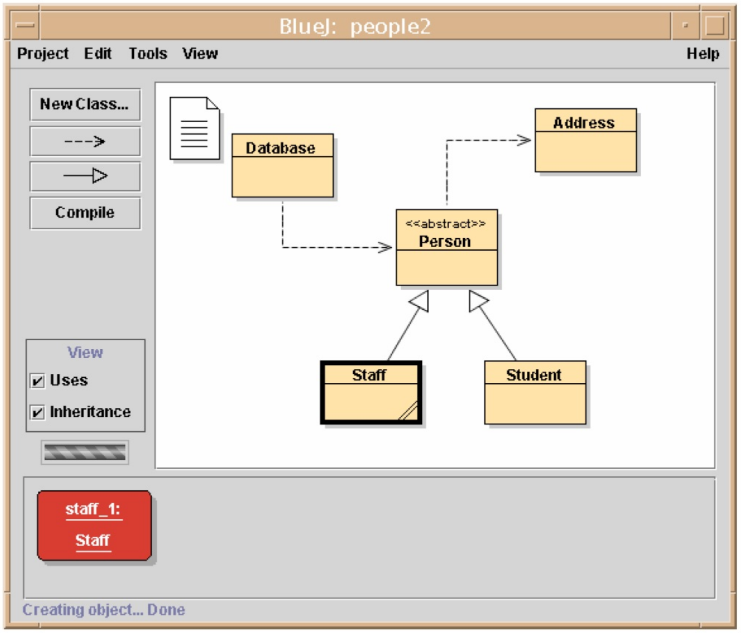


Figure 2.3: BlueJ Class Layout [26]

syntactically incorrect code, which removes one of the hardest hurdles for students who are first learning to program, and instead allows the student to focus on the logic of the program [2]. Like Karel however, Alice suffers from many of the same drawbacks since students are not actually writing any code themselves and they are limited to the tools Alice provides for them.

2.3 BlueJ

BlueJ is a pedagogical Java IDE designed to help teach object oriented programming that uses the standard Java JDK, compiler, and virtual machine [26]. BlueJ is based on three design goals: interactivity, visualization, and simplicity [27]. BlueJ uses a simpler interface to reduce the complexity that many students must overcome when learning to use an IDE

[26]. BlueJ also uses an UML style layout to display the class structure of the project and allows students to directly interact with classes or objects [26, 27]. Once a student creates the object, it is shown visually to help students grasp abstract object oriented concepts and the student can interact with the object to invoke any of its public methods [26, 27]. BlueJ also allows students to inspect objects to see the current values for their fields which allows for experimentation as the student can call the public methods, then see the results of the method call [26]. The downside to BlueJ is that in order to achieve its goal of simplicity, it has to sacrifice many of the tools found in a professional environment, which makes it only suitable as a learning tool [27].

2.4 Jeroo and Kara

Jeroo and Kara are two microworlds based heavily on Karel. Jeroo focuses on object-orented concepts using a kangaroo-like animal called a Jeroo that trys to avoid traps on an island and pick up flowers [5]. Jeroo uses an IDE where students can edit the code and see which line of code is being executed at a time [29]. Students can also change the speed at which the simulation is running or even run it one step at a time [29].

Kara is a ladybug that avoids tree stumps, moves mushrooms, and picks up or places clover leaves [28]. Kara focuses on the idea of Finite State Machines since the authors believes that FSM are easy for students to grasp [4, 28]. Both tools have been well received by students, however, they have the same drawbacks as Karel since students are still only limited to a single scenario and programming a single type of actor.

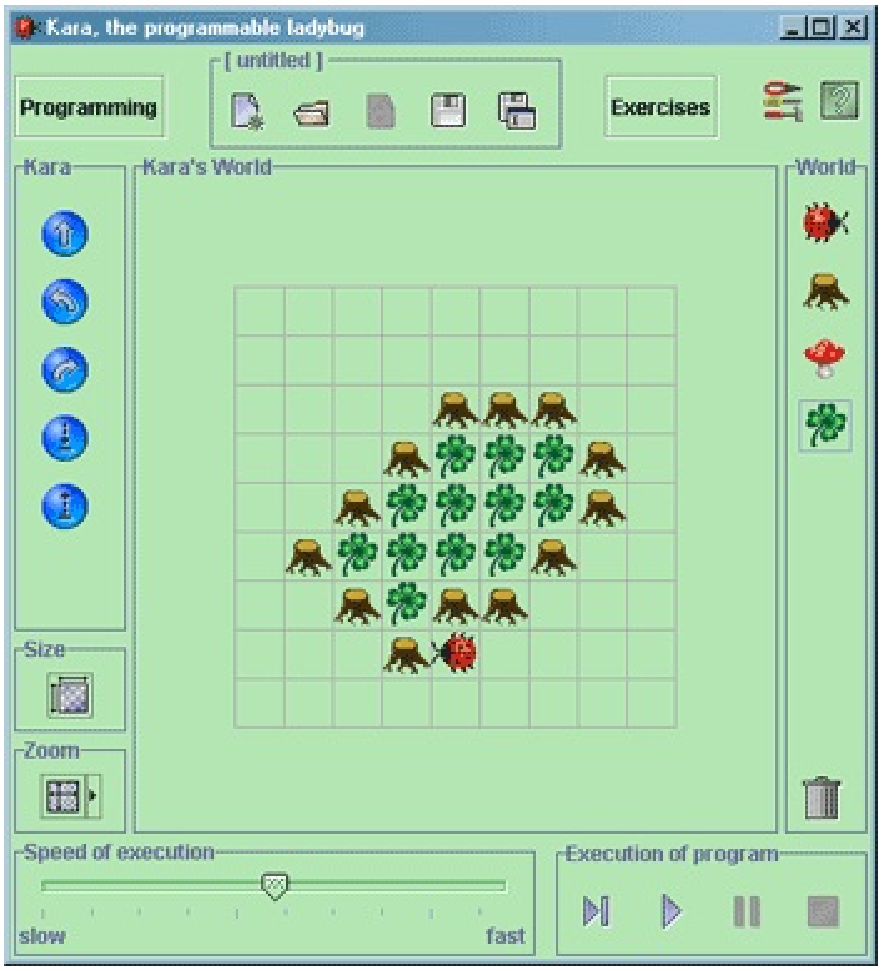


Figure 2.4: Kara Sample Program [4]

2.5 Greenfoot

Greenfoot is an integrated development environment (IDE) that aims to teach Java programming [25]. A 2D grid is used with the origin (0, 0) being in the top-left corner so students can address a given cell in the same way a 2D array would be indexed [25]. This grid represents a 'world' which can contain objects referred to as 'actors.' [7, 25] Every Greenfoot scenario will contain at least one class that extends World and one or more classes that extend Actor

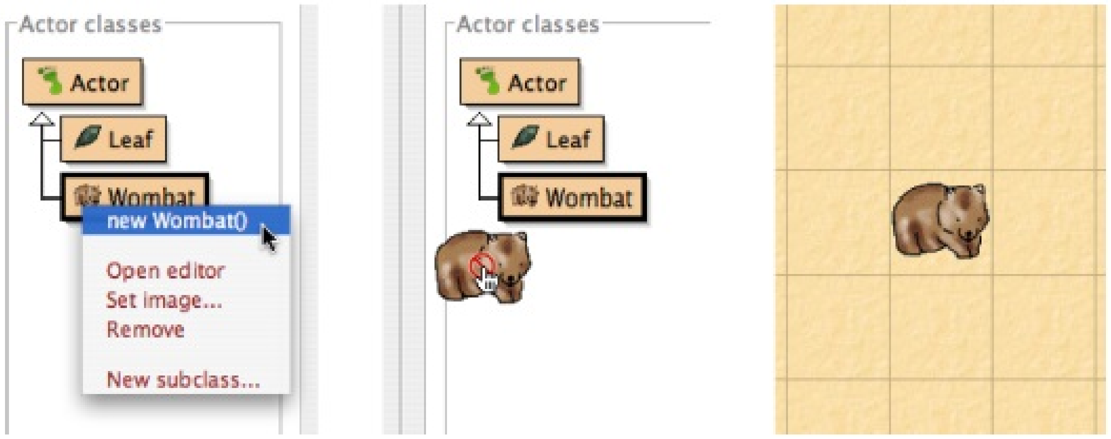


Figure 2.5: Greenfoot Wombats example [24]

[25]. Actors contain an 'act' method, which defines what each actor will do during each tick of the world timer and is where students write the majority of their programs. This paradigm of how students program actors closely follows the cellular automata idea where each object responds to their immediate surroundings and reacts accordingly.

Greenfoot also contains a simple user interface for students to use, which removes much of the advanced commands typically found in an IDE to reduce complexity for students [25]. The GUI also allows students to interact directly with classes and objects by right-clicking on them [25]. Users can create new Actors by right-clicking on the Actor in the Actor menu (see Figure 2.5) and users can also inspect the state of the actor by right-clicking and hitting inspect from the actor's menu [25]. In this way, Greenfoot helps make object-oriented more concrete by allowing users to create and see the actors they write.

Greenfoot's strength stems from the fact that it easily allows students and instructors to write custom scenarios. This avoids the drawback of Karel and the Karel-like microworlds

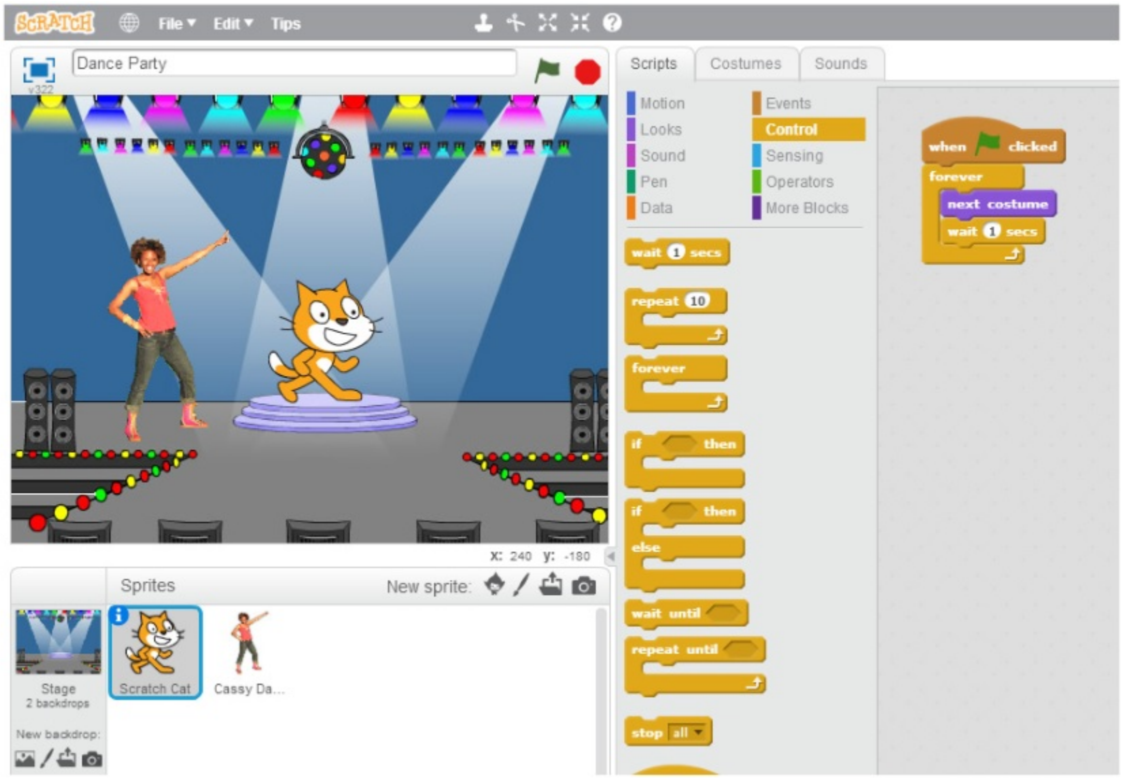


Figure 2.6: Scratch Sample Program [18]

since there is much flexibility in what students can write. However, like BlueJ, the IDE does not contain many of the tools typically found in an IDE.

2.6 Scratch

Scratch, created by the Lifelong Kindergarten Group at MIT, is an online tool for novice programmers to learn computer science concepts. Scratch also emphasizes the ‘no syntax error’ philosophy of Alice since students create programs (referred to as ‘scripts’) by dragging and dropping blocks that represent components in a program, such as expressions, conditions, and loops [16]. This allows students to easily create 2-D, event driven, concurrent programs

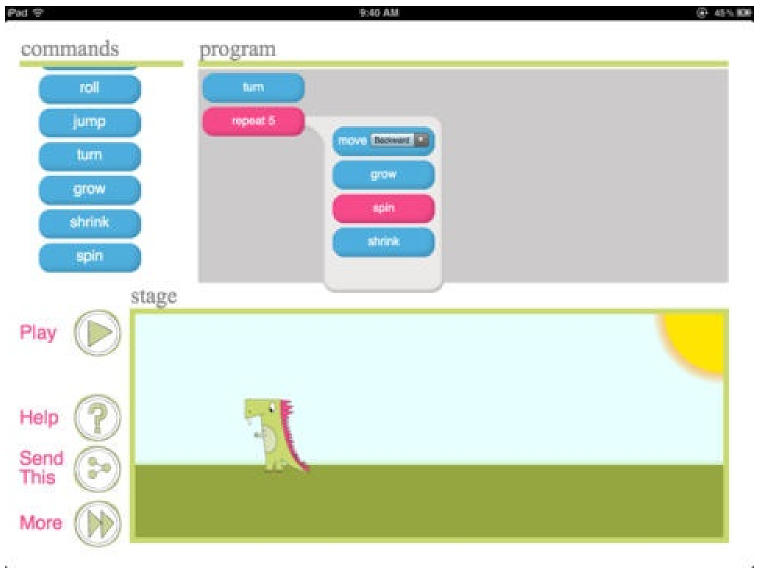


Figure 2.7: Screenshot from Daisy the Dinosaur App

without requiring students to know and understand all the underlying concepts [17]. Scratch differs from other pedagogical tools by focusing on concurrency and event-driven programming early on before moving on to object oriented concepts [17]. Scratch also has a large emphasis on social networking by allowing students and teachers to share their projects [16]. Scratch's emphasis on students not writing their own code means that it is more appropriate for a younger audience since younger students tend to struggle more with syntax [33]. However, like Alice, students can quickly master everything Scratch has to offer and the skill ceiling is lower for Scratch since students do not write their own code.

2.7 Smart Phone Apps

There have been a few apps that have been released on either the Android or iPhone app store that use Karel or Scratch-like environments to help teach programming. Cato's Hike teaches programming in a Karel-like fashion where students program a child to navigate mazes to collect hearts or stars [9]. Daisy the Dinosaur is similar to Scratch in that users drag and drop actions in order to control the dinosaur on screen (see 2.7) [10]. Hopscotch also uses a Scratch-like approach to teach students to code by dragging and dropping blocks to create games or animations [11]. All of these apps are targeted to students around the ages of 8-10, which makes them ideal for introducing students at a young age to programming logic. They also demonstrate how smart phones and tablets can act as a motivator for students to learn programming.

Chapter 3

Sofia.Micro Features

Sofia.Micro has several novel features that have not been used before in microworlds or microworld frameworks. In addition to supporting Android apps, Sofia.Micro has the ability to support both types of actors commonly seen in microworlds. Sofia.Micro also supports 2D shapes and physics simulation for actors and shapes. Sofia.Micro is the first microworld framework that supports event-driven style programming.

3.1 Actors and Programmable Actors

Microworlds typically use one of two different strategies for defining the behavior of actors. Karel, and later Jeroo and Kara, used what we will call *programmable actors*. In this type of actor, the entire program is written for the actor and then executed one step at a time. Students have to write out the entire logic for their actor from the beginning and think about solving their problem in a more 'global' sense and think about their logic in a more abstraction-oriented perspective.

The other type of actor is the cellular-automata-style actor, such as the ones used in Greenfoot. For these types of actors, students write what each actor will do at each step of the world clock with the actor reacting to their immediate surroundings and then moving,

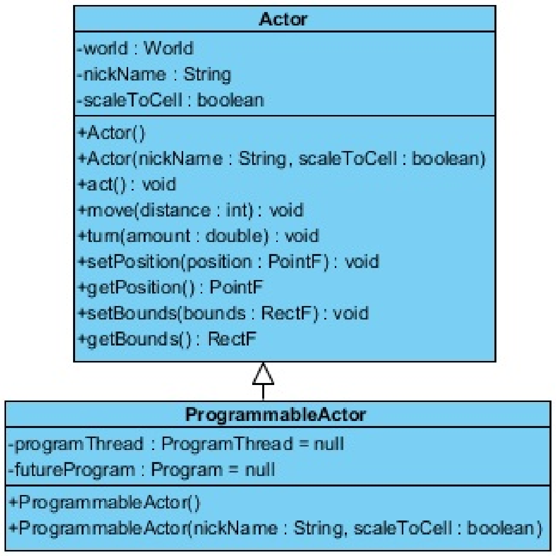

Actor
-world : World
-nickName : String
-scaleToCell : boolean
+Actor()
+Actor(nickName : String, scaleToCell : boolean)
+act() : void
+move(distance : int) : void
+turn(amount : double) : void
+setPosition(position : PointF) : void
+getPosition() : PointF
+setBounds(bounds : RectF) : void
+getBounds() : RectF
ProgrammableActor
-programThread : ProgramThread = null
-futureProgram : Program = null
+ProgrammableActor()
+ProgrammableActor(nickName : String, scaleToCell : boolean)

Figure 3.1: Actor UML Diagram

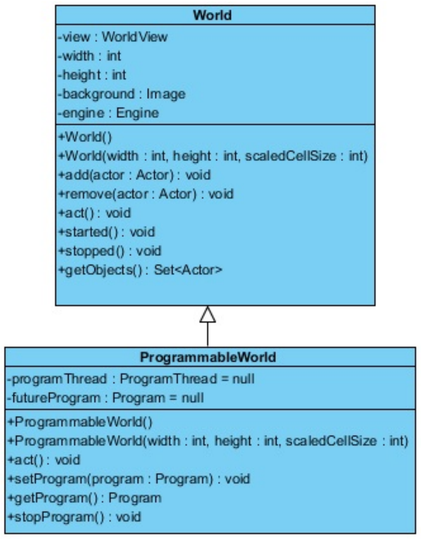

World
-view : WorldView
-width : int
-height : int
-background : Image
-engine : Engine
+World()
+World(width : int, height : int, scaledCellSize : int)
+add(actor : Actor) : void
+remove(actor : Actor) : void
+act() : void
+started() : void
+stopped() : void
+getObjects() : Set<Actor>
ProgrammableWorld
-programThread : ProgramThread = null
-futureProgram : Program = null
+ProgrammableWorld()
+ProgrammableWorld(width : int, height : int, scaledCellSize : int)
+act() : void
+setProgram(program : Program) : void
+getProgram() : Program
+stopProgram() : void

Figure 3.2: World UML Diagram

removing itself or another actor, changing direction, etc. This follows an approach similar to how a greedy algorithm works where the actor makes the best local decision in order to try to solve a global problem. This style of writing actors is less restrictive than the programmable-actor-style since students just need to focus on writing how an actor should respond at each individual step, rather than how they should respond over the entire program. These types of actors can usually be identified by an overridden method that appears in every actor (and world), in the case of Greenfoot and Sofia.Micro, this is the `act()` method.

Sofia.Micro employs both types of actors and worlds that are commonly used for these microworlds, namely *actors* and *programmable actors*, and *worlds* and *programmable worlds*. Like in the Greenfoot model, each actor has an `act()` method that is called during each tick of the global clock. Students can extend the appropriate actor and override the act method to create behaviors for the actors.The programmable actor extends the Actor class and uses a program thread to handle the actor's behavior. The entire program logic is written in a Program class, which is given to the ProgrammableActor and then executed one step at a time. Both actors support similar operations, such as changing their position and rotation, performing queries on other actors in the world, or adding and removing other actors.

There is a 'World' class that students can extend to create their own World for an application. This world class uses a custom thread for the world engine which handles the execution of the world and its actors. During each tick of the engine clock, the act method is called for the world and all of the actors along with dispatching any buffered events to the world and actors. The world uses a grid system similar to Greenfoot where a world is given dimensions in number of cells (width x height) and how large each cell will be in pixels. This allows for programmers to create simple grid worlds with a few number of cells (see Section 5.1), or to create a world where each cell is only a pixel to create worlds with more realistic movement (see Sections 5.2 or 5.3).

Like the programmable actor, there is also a ProgrammableWorld that extends the World class, which contains a field called 'Program.' The program is simply an object which students can either override the `myProgram()` method, or write their own program and pass it into the `setProgram()` method. Like the ProgrammableActor, the ProgrammableWorld uses the `beginAtomicAction()` and `endAtomicAction()` to divide the program into steps.

Figure 3.3: Screenshot of Physics Games on Google Play Store

3.2 Shape and Physics Support

Students will be familiar with several physics-based Android games, such as Angry Birds or some of the apps in Figure 3.3. To help students with writing physics-based Android apps, we have incorporated 2D shape and physics into Sofia.Micro. We not only want students to write unique and varied applications, but we want them to be able to do it easily. By directly incorporating these elements, we can allow students to use physics in their game if they wish.

Sofia supports 2D graphics as well as physics support through JBox2D. This allows for interactions and behaviors such as gravity, collisions, detection, objects bouncing, and objects sliding. By utilizing the physics engine, a programmer can greatly simplify the amount of code they need to write in order to have some basic physics in a game or application. Since Sofia.Micro is an extension to Sofia, then Sofia.Micro can employ both the shape and physics support to add functionality that is not previously seen before in a microworld. This allows for shapes to coexist with actors in a world and since actors are also shapes themselves, then the two can interact with each other and be affected by physics, such as collision detection or forces.

The shape support allows students greater flexibility in what is placed in each world. Students no longer need to use an actor for every object in the world if they do not need to and can instead use simple 2D shapes when appropriate. This helps clean up some of the code since the student does not need to explicitly write an actor for some simple behaviors. Sofia.Micro support lines, rectangles, ovals, and polygon shapes so the student can use these to create objects such as projectiles in a game or walls and platforms for a side-scroller. This also allows students to write simple GUI applications, such as a Paint application, that lets the user draw basic shapes on the screen.

With Sofia.Micro's physics engine, students can easily create physics-based games, such as Asteroids (see Section 5.2), Angry Birds (see Section 5.3), or Plants vs. Zombies (see Section 5.4). By utilizing the physics engine, students can write code for moving and detecting actors or shapes in a cleaner fashion. For example, rather than using the `move()` method in actor to move, an actor can instead say that it is a dynamic object, and give itself a linear velocity. This may not look much different, but since the actor is moving by the physics engine rather than being manually moved by the user, then it collides and reacts to the environment in a more realistic fashion. This also avoids the issue of a fast-moving actor moving past another actor completely since a physics actor will be moved more frequently.

The physics engine supports three different types of physics objects: static, kinematic, and dynamic objects. Static objects can not be moved except by explicitly changing their position, x, or y coordinate and they do not participate in collisions with other static or kinematic objects [3]. Kinematic objects are similar to static objects, except they can have a non-zero velocity [3]. Like static objects, however, they have infinite mass and do not collide with other static or kinematic objects [3]. Lastly, dynamic objects are simulated objects that move by forces, collisions, velocity, etc and will respond to collisions with other objects [3]. Dynamic actors can bounce off each other, apply forces, or use acceleration much easier since the student does not have to explicitly change the movement of the actor themselves and let the physics engine handle it. This eases a significant portion of the burden of moving actors in a world since students no longer need to calculate the velocity or position themselves, they can instead just let the physics engine worry about that.

3.3 Event-Driven Programming & Event Dispatch

Kim Bruce et. al in *Java: An Eventful Approach* discuss how event-driven programming is an appropriate programming paradigm for students because most real-world applications will be GUI-based as opposed to command line based [20]. They also emphasize in their other works that students can quickly pick up event-handling since they can focus on parameters and methods along with the amount of code they have to write to do something interesting is very small [21, 22, 23]. Given Android's focus on event driven programming in general, Sofia's microworld was designed to support touch and key events so that students can write event-driven programs themselves. This emphasis on event-driven programming differentiates Sofia.Micro from other microworlds, which may not feature any event-driven support at all. Students can still use an imperative programming paradigm for their programs if they wish, but they have the option for event-driven style programs.

To our knowledge, Sofia.Micro is the first microworld framework to use event driven programming at all. Microworlds, such as Karel or Jeroo do not support any kind of event-driven style programming, however Greenfoot does support a polling style of event handling where actors can “ask” if a button is pushed down or if the mouse is located in a specific area. However, Sofia.Micro is the first to support a true event driven programming style. Students can write event handlers in any of their actors, shapes, or even the world itself and events will be automatically dispatched to them. A list of the supported event handlers are given below:

Table 3.1: Touch Handler Methods

Method Name	**Description**
`onTouchDown()`	User initially touches the object
`onScreenTouchDown()`	User intially touches anywhere on the screen
`onTouchMove()`	User drags their finger in the object
`onScreenTouchMove()`	User drags their finger anywhere in the screen
`onTouchUp()`	User lifts their finger in the object
`onScreenTouchUp()`	User lifts their finger anywhere in the screen
`onTap()`	User touches then quickly releases their finger in the object
`onScreenTap()`	User touches then quickly releases their finger anywhere in the screen
`onDoubleTap()`	User touches then quickly releases their finger two times in a row in the object
`onScreenDoubleTap()`	User touches then quickly releases their finger two times in a row anywhere in the screen

For these handlers, students can also specify additional arguments for the methods. These handlers can take two floats or two ints, one for the x coordinate and one for the y coordinate or a Point object, which contains the respective x and y coordinate. Students can also make use of Sofia's unique event dispatch model to help simplify some of the code they need to write. For example, in the Asteroids case study, rather than having the following in the `act()` method:

```
Set<Ship> ships = getIntersectingObjects(Ship.class);
if (!ships.isEmpty())
{
    for (Ship ship : ships)
    {
        ship.remove();
    }
}
```

Students can instead write:

```
public void onCollisionWith(Ship ship)
{
    ship.remove();
    remove();
}
```

The latter helps emphasize an event driven model since it is only called when there is actually a collision as opposed to constantly checking if a collision has occurred. In terms of writing handlers for typical events, such as touch or key events, students just need to write the appropriate handlers in their actor or world. Any captured events are stored in a buffer and Sofia.Micro will automatically dispatch any buffered events to classes with handlers by using Reflection before the `act()` method is called for the world and actors respectively.

To help facilitate students writing event-driven programs, Sofia.Micro also supports the use of a directional pad (dpad) that uses its own handler methods listed below:

- `dpadNorthIsDown()`
- `dpadNorthEastIsDown()`
- `dpadEastIsDown()`
- `dpadSouthEastIsDown()`
- `dpadSouthIsDown()`
- `dpadSouthWestIsDown()`
- `dpadWestIsDown()`
- `dpadNorthWestIsDown()`
- `dpadCenterIsDown()`

When an actor is added to the world that has at least one of these handler methods, then the dpad is added to the world in the bottom-left corner of the screen. The dpad is mostly transparent so it does not obstruct anything behind the dpad when it is moving during an application. If a user touches the dpad, it will calculate which part of the dpad was touched and automatically call the appropriate handler methods. The student can then write the code in these handlers to change the behavior of the actor or the world when one of these handlers is called. This gives the student greater control in how they want to move any actor or shape within their scenario since they can opt to use the dpad to manually control the actor or shape rather than letting act or physics engine completely dictate its action.

Chapter 4

Design & Implementation

Sofia.Micro is designed with its predecessors in mind, in particular, Greenfoot and Karel. Sofia.Micro uses a relatively few number of classes, with the majority of the work being done in the Actor, World, and WorldView classes. By extending the ShapeScreen, the WorldScreen can use the shape and physics support with some modifications to ensure that physics objects behave properly in the microworld. Program is an interface that ProgrammableActor and ProgrammableWorld both implement that allows a user to create a Program that is executed

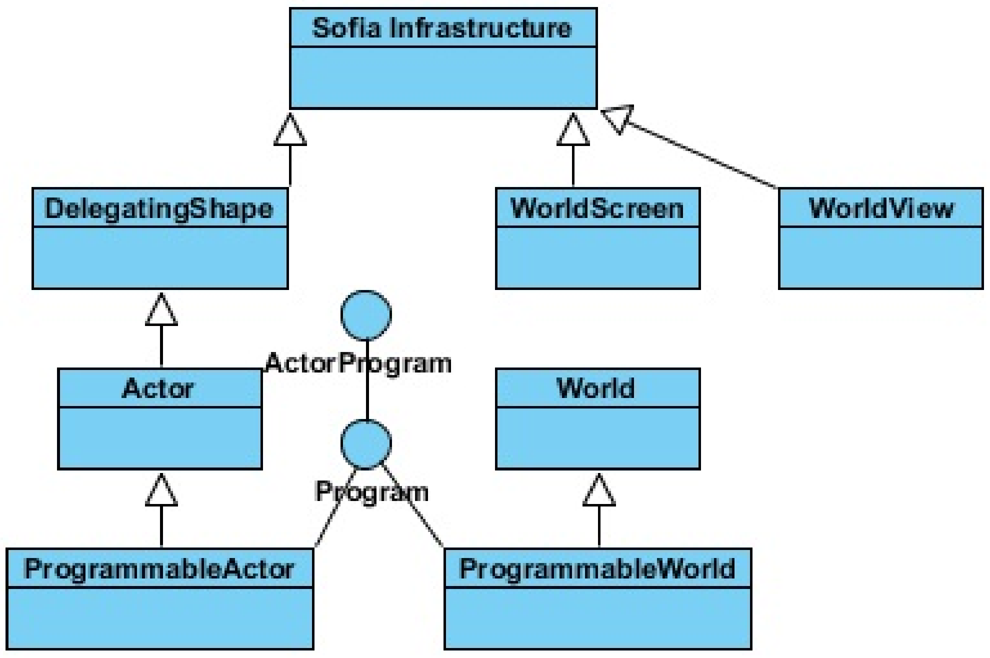


Figure 4.1: Sofia.Micro UML Diagram

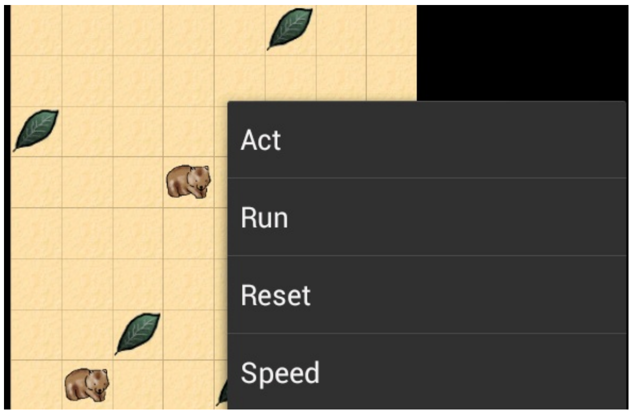


Figure 4.2: Sofia.Micro Menu

step-by-step.

Sofia.Micro apps have four defined menu items (see Figure 4.2), Act, Run/Pause, Reset, and Speed. Hitting the Act button iterates over each actor and the world once and calls their respective act methods. The Run button will cause the World to continuously call the `act()` method on all the actors and the world, in addition to allowing physics based objects to run normally. After Run is touched, Pause will take its place which will stop all actors and the physics simulation. Reset will call the initialize method for the screen to reset all the actors and shapes in the world. Finally, the Speed button will allow users to change the frequency at which the act methods are called when running.

4.1 Actors and Programmable Actors

Sofia's actors themselves are a shape, which as mentioned earlier, allows actors to coexist with shapes and also use all of the functionality found in shapes, including the physics support. Actors extend a shape called DelegatingShape, which is a shape itself and also contains a Shape object as one of its fields, referred to as the 'delegate.' Rather than have the DelegatingShape object itself handle bounds checking, appearance, and collision detection, it instead lets the delegate handle these. By doing this, a delegating shape object can alter the way it behaves by simple changing the delegate shape inside the object.

Actors use an ImageShape as their delegate, which is an extension to the RectangleShape class with an image used for the appearance of the shape rather than a fill color. This also means that actor's use the RectangleShape's bounds and fixtures when it comes to collision detection. When an actor is created, Sofia will automatically try to find an image with the same name as the actor in the /res/drawable/ folder and assign that as the ImageShape's image. This eliminates the need for users to explicitly search for an image in the code and can allow Sofia to perform this step for them, however, the user still has the option of explicitly setting the image. A cache is used to help speed up looking up images when multiple actors of the same class are created by using either the actor's class name or a String as the key.

Actors are also given a bounds, which is the bounding box used for collision detection and also determines how large the actor is. Actors have a flag to determine if the bounds should be scaled to the grids in the world. If the flag is set to true, which is generally the case for worlds with a few cells (see Section 5.1), then the bounds is set to be from -0.48 to 0.48 for both the width and height. Sofia automatically centers the shape in the middle of the cell and because the bounds are the center of the cell ± 0.48, then when a student accesses cell (x, y), they are referring to the center of the cell rather than the top-left corner of the cell. To avoid edge cases where the bounds are touching at the edges, the bounds use ± 0.48 rather than ± 0.5. In the case of the scale to cell flag being set to false, then the bounds are merely the bounds of the image scaled to the world, which is typically used when the world has very small grid cells, such as in the Asteroids example where the world is a 500x500 world with grids 1 pixel wide.

When an actor is added to the world, the programmer can specify which (x, y) coordinate to add the actor to, otherwise the actor will be added at coordinate (0, 0) by default. Once added to the world, programmers can use several of the actor's methods to move or rotate it, such as `move(int distance)`, `setX(float x)`, `setY(float y)`, `setRotation(float angle)`, and other methods to help the actor find other shapes or actors for querying or collision detection, such as `getNeighbors()`, `getObjectsAtOffset()`, or `getIntersectingObjects()`.

As mentioned earlier, students will extend the Actor class with their subclasses that will then override the act method with their own implementation. The World class's Engine inner class will call the act method on every actor every tick of the clock. Actors and shapes that are added or removed during the act of another actor are placed in a list (named deferredAdds or deferredRemoves), which is dealt with after all the act methods have been called. This avoids the issue of synchronization, concurrent modification exceptions on the list of actors, and also helps ensure that the order of actors does not cause exceptions or abnormal behavior.

4.2 Shape and Physics in Sofia

Sofia's shape model first begins with the abstract class, called 'Shape' that every other shape will extend. The shape contains several fields, including the color, alpha, rotation, as well as some fields for JBox2D, such as a Body, density, and friction. Subclasses of Shape, such as oval, rectangle, or polygon shapes will provide functionality for the bounds and fixtures for the shape.

The physics is provided primarily by JBox2D, which handles much of the collision detection, applying forces to shapes, and moving the shapes. Users can apply forces to a shape, such as gravity using the `setGravity(PointF)` method, or give the shape a linear velocity using `setLinearVelocity(PointF)`. Users can also specify how a shape interacts with other shapes by using the `setShapeMotion()` method to specify if the shape is dynamic, static, or kinematic. Shapes are by default static, which means it has zero velocity and infinite mass and can only be manually moved by the user. Kinematic shapes also have infinite mass, but they can be given a velocity. Static and kinematic bodies will not interact with each other or themselves. On the other hand, dynamic shapes are subject to all types of physics interactions, including velocity, mass, forces, torques, and impulses [3]. Dynamic shapes also can interact with all three types of shapes and use the shape's built-in fixtures to simulate any physics.

Sofia also supports a shape filter, where a user can chain together multiple filters onto the same call. There is support for finding actors in the ShapeView and WorldView that

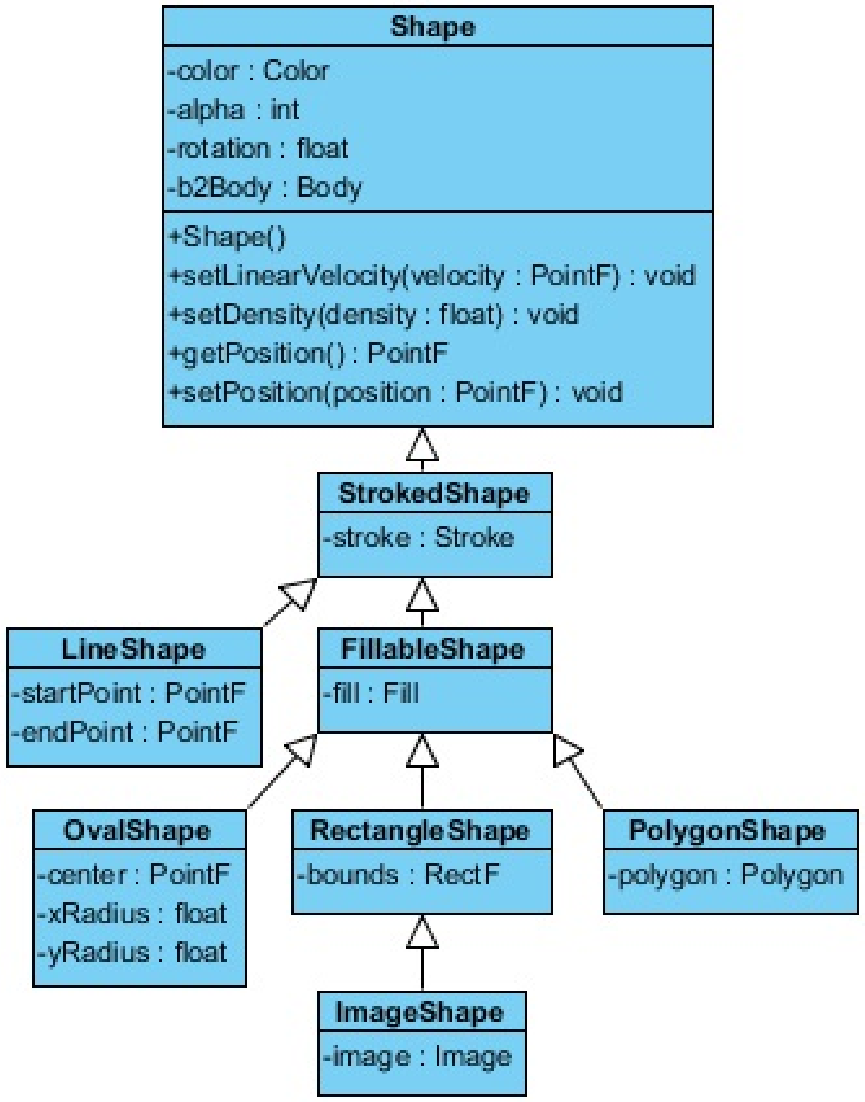


Figure 4.3: Shape UML Diagram (Note: not a complete UML diagram)

applies the necessary filters to simplify some commonly used collision detection or search queries that are typically done. For example, if a user wanted to find all the actors within a circle at coordinate (x, y), they could use:

```
getShapesInRange(x, y, r, Actor.class);
```

Which is merely a shortcut for:

```
return getShapes().withClass(Actor.class).locatedWithin(new PointF(x,y),r).all();
```

Sofia also provides animation support for shapes as well. Users can chain together different animations, such as fading in the shape over 1 second while changing the color.

4.3 Event Dispatch

The event dispatch is accomplished in the WorldView class by using a double-buffer system. Events are buffered in an ArrayList whenever they are caught in the WorldView. During the `run()` loop for the World, before the `act()` is called for the World, all the events in the buffer are dispatched to the World. The same is done for each actor before that actor's `act()` method is invoked. This allows for motion and key events to be handled before the world or actor's behavior is performed. While events are being dispatched to the World and each actor, the second buffer will catch any events that may be fired during that time since dispatching events is not handled instantaneously. Once the World's and actor's act methods have been performed, then the buffers are swapped and the second buffer will be used on the next iteration of the `run()` loop, and so on.

During implementation of the double buffer, a LinkedList was also considered for the double buffer since the ArrayList has to worry about potentially expanding the internal array if the number of buffered events got too large whereas the LinkedList does not have to worry about such a consideration. On the other hand, the LinkedList has to go out to the heap every time a LinkedList node is created which can be relatively expensive, whereas the ArrayList just has an array of references that do not need to be recreated. In practice, the ArrayList never needed to be resized, or in the worst case, had to be resized once. Since the

ArrayList does not need to go out to the heap every time to add an item to the buffer, it was the data structure that was used.

Another design consideration was the way the events were stored in the buffer. One problem is that the 'action' for motion events and 'key code' for key events are static, as in, there is only a single copy stored across all events. When buffering and then dispatching the events, the 'action' would always be the 'action' of the last motion event stored and likewise for the key event. To fix this issue, rather than just buffering the event itself, a wrapper class was created that would store both the event and the action or key code at the time it was buffered. This wrapper class is created every time an event is fired and then stored in the corresponding buffer.

One issue that arose with motion events is that the x and y value stored in the motion event are in pixel coordinates, however, this is not suitable for a grid-based microworld since actors are generally addressed by their grid x and y coordinate. To address this issue, the x and y values in the motion event are adjusted so that the pixel locations are converted to grid cell coordinates and offset so pixel coordinates to the left of the origin (0, 0), are considered negative. These adjusted x and y values are also stored in the wrapper for motion events so they can also be dispatched to the world or actors.

To help with dispatching events, a TouchDispatcher class was created. This class has a static `dispatchTo()` method that takes in the object to dispatch the event to, the event itself along with its action code, and a PointF object that contains the x and y coordinate where the event occurred. For Sofia.Micro, this x, y coordinate pair will be the grid location, otherwise, it will just be the motion event's `getX()` and `getY()`. In order to speed up the dispatch portion when the world is dispatching the events to all the actors, a static `hasDispatchableEvents()` method is used which will return true if the actor has any motion event handlers so that dispatching events is skipped for actors that have no event handlers. After the motion and key events have been dispatched to all the actors and worlds, then the buffers being used are cleared and then swapped so the next iteration of the run loop in the Engine will use the other buffer.

4.4 Directional Pad

Sofia.Micro's directional pad (dpad) is implemented as a shape that extends the RectangleShape class with a dpad image (see Figure 4.4). Touch events are dispatched to the dpad before being dispatched to the actors and the world. The dpad contains motion event handlers for `onTouchDown()`, `onTouchUp()`, and `onTouchMove()` that catches any motion events and sends them to a helper method. This method checks to ensure that the motion event is within the bounds of the dpad, and calculates if the touch was in the center of the dpad or which one of the 8 quadrants of the dpad the touch occurred. Touching the center or one of the four cardinal directions converts the touch event into a single key event, for example, touching the top portion of the dpad will convert the touch event into a KEYCODE_DPAD_UP key event. Touching one of the corner will convert the touch event into two key events, for example, touching the top-right corner will convert it into a KEYCODE_DPAD_UP and KEYCODE_DPAD_RIGHT key event. These key events are then redispatched, where they are handled and buffered by the WorldView.

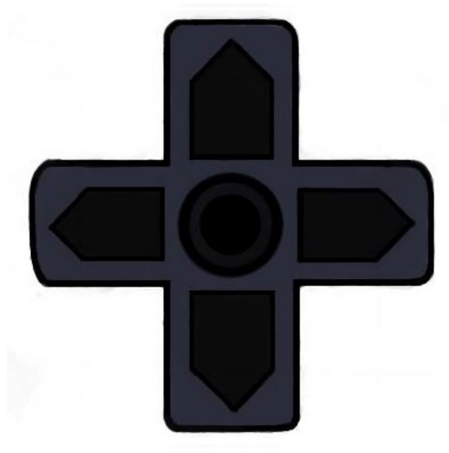


Figure 4.4: Directional Pad

If an actor or world has one of the directional pad methods, then the DpadDispatcher class will dispatch the dpad events to the class. If an actor contains one of the dpad diagonal directions, then the event is dispatched to that method, if not, then it is dispatched to the two corresponding cardinal directions (ie northwest would be dispatched to north and west). However, the KeyEvent class does not have a constant for the diagonal directions, so to ensure they are properly dispatched, a bitmask is used where the two directions are combined into a single int using bit shifts and ANDs. They are later unmasked in the dispatch method for the DpadDispatcher.

The dpad is not added to the world by default, instead, when a student adds an actor to the world, the world will check to see if the actor has any dispatch-able dpad methods. If the actor does have some dispatch-able methods, then the dpad will be added to the bottom-left corner of the screen. This simplifies the code for the student since they do not have to explicitly add the dpad themselves and do not have to calculate the bounds of the dpad.

Chapter 5

Case Studies

To demonstrate the capabilities of the new features for Sofia.Micro, we have implemented a few classic Greenfoot examples in addition to writing a few of our own examples.

5.1 Jeroo

As mentioned in Chapter 2, a jeroo is a kangaroo-like animal on an island that hops around collecting flowers while avoiding traps. Jeroo is similar to Karel in that it also uses the programmable actor found in Karel and other similar microworlds. Sofia.Micro has a built-in jeroo package (along with Lightbot and Greenfoot) so students who wish to use it can simple extend or use the appropriate jeroo classes.

5.1.1 Actors and Programmable Actors

Jeroo emphasizes the actor aspects of Sofia.Micro, namely the actor and programmable actor. The jeroo itself is a programmable actor where students write their logic in the 'myProgram' method. An example of some of the logic for a jeroo program is given below:

```
public void myProgram()
{
    clearRows();
}

public void clearRows()
{
    while (!seesWater(AHEAD) || isClearBelow())
    {
        if (clearRow())
        {
            if (isClearBelow())
            {
                turnAround();
            }
        }
        else if (!this.seesWater(down()))
        {
            moveToNextRow();
        }
    }
}
```

The other actors in the jeroo package (ie the flowers and nets) are just regular actors. While they do not have any defined behavior by default, it does demonstrate that a jeroo

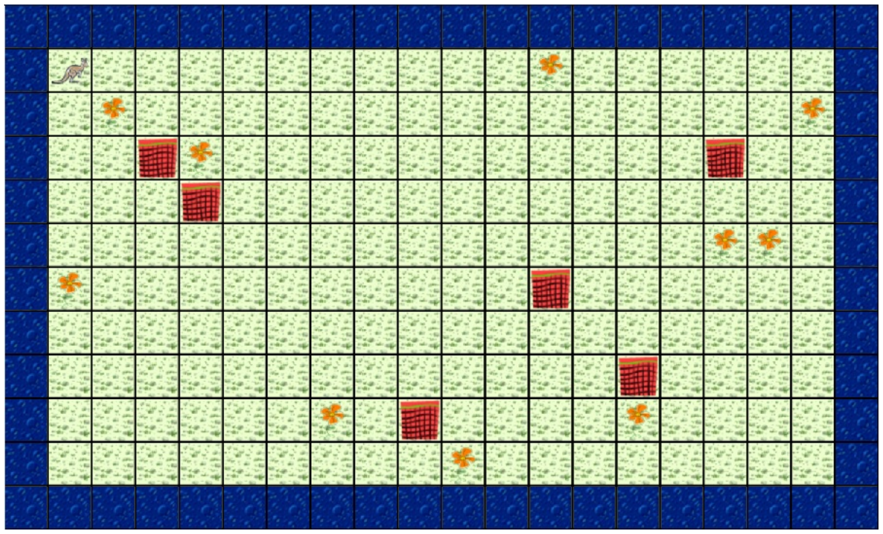


Figure 5.1: Jeroo in Sofia.Micro

world can support both actors and programmable actors in the same world. Jeroo also demonstrates that Sofia.Micro can recreate previous microworlds in an Android environment, which allows for instructors and students to use them in a strictly Greenfoot-style or Karel-style if they so choose.

5.1.2 Simple Microworlds on Android

Jeroo shows that it addresses both issues of the problem statement. Students can create unique and diverse Greenfoot-style or Karel-style microworlds to help keep the tool interesting and give it some longevity. Also, given the ease with which they can create these environments, it shows that Sofia.Micro can help bring Android to students in an simpler context.

5.2 Asteroids

In this program, the user controls a ship that must avoid the asteroids on the screen and shoot them to destroy them. If a bullet collides with an asteroid, then both are removed from the world. Likewise, if the ship collides with an asteroid, then both are removed and the player has to restart the app to keep playing. The user controls the ship using the dpad and tapping the screen. Touching the up section of the dpad will move the ship forward in the direction it is facing, while touching the down section will slow the ship down. Touching the left or right areas will turn the ship in the corresponding direction and tapping the screen will cause the ship to fire bullets. The ship does not have any of the diagonal dpad methods,

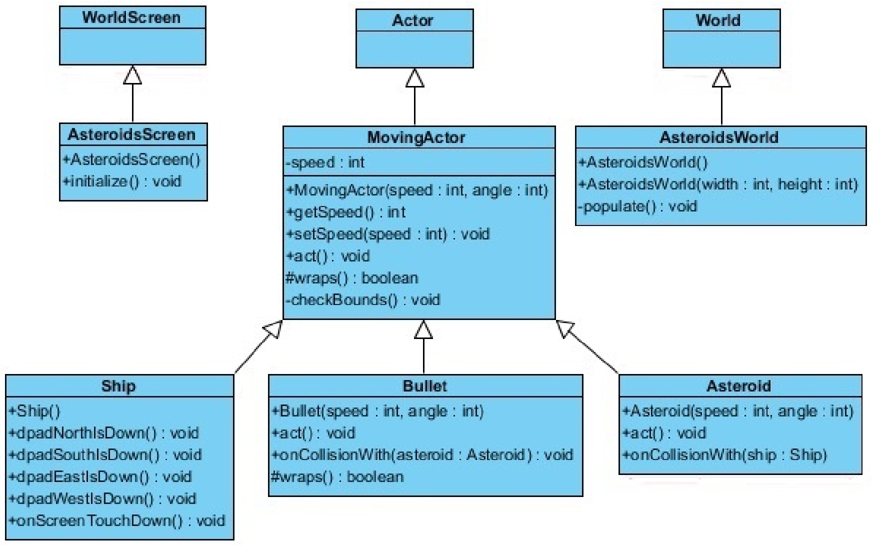


Figure 5.2: Asteroids UML Diagram

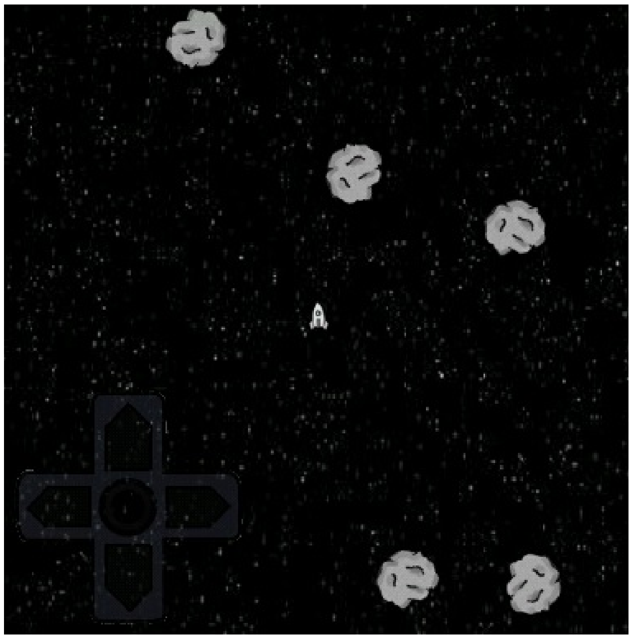


Figure 5.3: Asteroids Layout

so the dpad dispatcher will dispatch any diagonal directions into the corresponding cardinal directions.

5.2.1 Event-Driven Programming using the Dpad Controller and Simple Physics

This case study emphasizes a few different aspects of Sofia.Micro. It first emphasizes the event-driven nature of both Sofia.Micro and Android applications. Students would need to write the dpad handlers for the ship to control the ship's movement. The amount of code students need to write for the ship's handlers is a single line per handler.

```
public void dpadNorthIsDown() { setSpeed(getSpeed() + 1); }
public void dpadSouthIsDown() { setSpeed(getSpeed() - 1); }
public void dpadEastIsDown() { turn(5); }
public void dpadWestIsDown() { turn(-5); }
public void onScreenTouchDown() {
getWorld().add(new Bullet(50, (int) getRotation()), getGridX(), getGridY()); }
```

Asteroids also demonstrates the event-dispatch strengths of the Sofia framework since detecting collisions happens automatically now with the `onCollisionWith()` methods, so students can instead focus more on what happens when a collision occurs.

```
public void onCollisionWith(Ship ship)
{
    ship.remove();
    remove();
}
```

Secondly, it emphasizes the use of the physics engine in realistically moving actors so that students do not need to worry as much about moving objects or bouncing Asteroids off each other when they collide. This was accomplished by changing the MovingActor class, which is the class all the actors in this scenario extend from, to be a dynamic shape. Movement for the actors is handled in the constructor method by setting its linear velocity to be (speed * cos(θ), speed * sin(θ)). While this may not change a lot in terms of how the game is played, it does make the game feel more like a physics based game, rather than merely emulating one. Finally, this case study demonstrates how the dpad works for Sofia and that users can use it as a controller for one or more actors in their microworld.

5.2.2 Easily Creating Simple Physics-Based Games

Like the jeroo, Asteroids shows that it addresses both of the main issues in the problem statement. Asteroids demonstrates how easy it is for someone to write a very simple game that works. The amount of code that needs to be written is also fairly minimal, numbering roughly around 70-80 lines of code excluding comments and whitespace. The physics engine handles most of the trickier logic for this application, so students can instead focus on writing the dpad handlers and setting up the actors for the world. Asteroids also shows that students can create diverse applications to increase the life span of Sofia.Micro as a learning tool.

5.3 Irritated Avians

This case study is a simplified implementation of the Angry Birds game, renamed "Irritated Avians." The goal of the game is to the launch the red bird at the green pigs in order to remove them from the world. To play, the user will drag the red bird and release at the desired angle and distance, which will propel the bird forward and if it collides with one of the green pigs, then the pig will be removed from the world. This game was originally written using the base Sofia framework by [14] and was modified to fit Sofia.Micro. In the

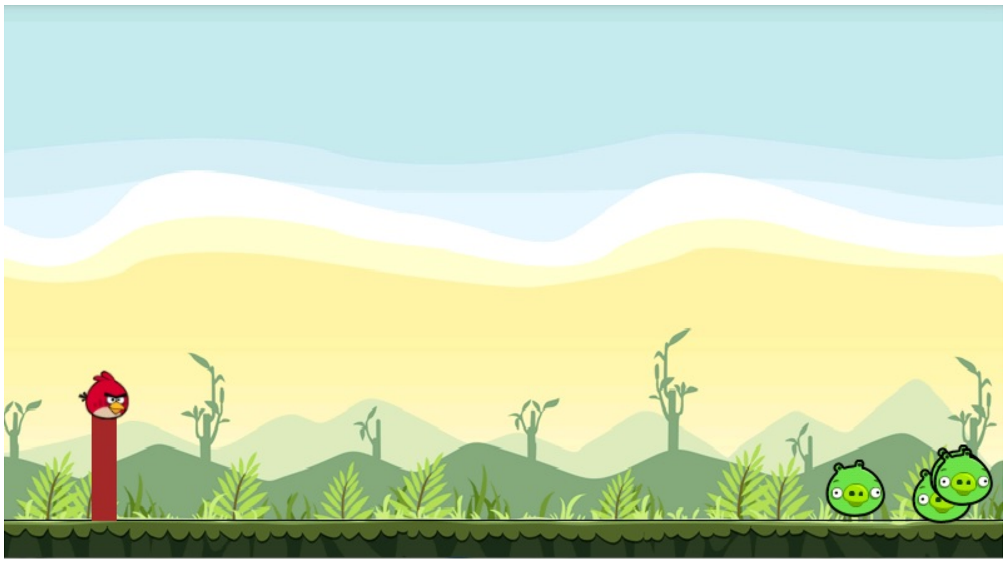


Figure 5.4: Irritated Avians Screenshot

Sofia.Micro version, the birds and pigs are now actors, however, they still use the physics engine to determine movement and collisions. The ground and the ovals the bird leaves behind when fired were left as shapes since they have no behavior in this scenario.

5.3.1 Event-Driven Programming with Physics

Like the Asteroids case study, Irritated Avians also emphasizes the physics engine of Sofia.Micro along with the unique event dispatch that Sofia provides. What is different is that Irritated Avians also demonstrates that a world can contain non-actor shapes at the same time as regular actors. The birds and pigs are actors in this case, while the ground, slingshot, and the white balls the bird leaves behind as it moves are all shapes. Of these three, the ground is the only one that interacts with the actors by preventing them from leaving the scene. This example also demonstrates that actors in Sofia.Micro do not necessarily need an act method to move if the student does not want to use it — movement and collision detection can all be handled by the physics engine. This example also emphasizes the event-driven nature of both Sofia.Micro and Android since all of the interactions in this program are done via the `onTouchUp()` and `onTouchMove()` handlers.

```
// code for moving the bird before it is shot off
public void onTouchMove(float x, float y)
{
    float distance = Geometry.distanceBetween(
            startingPosition.x, startingPosition.y, x, y);
    float angle = Geometry.angleBetween(
            startingPosition.x, startingPosition.y, x, y);

    distance = Math.min(distance, 150.0f);

    PointF newPosition = Geometry.polarShift(
            startingPosition, angle, distance);
    setPosition(newPosition);
}

// code for flinging the birds once the user stops touching the screen.
public void onTouchUp(float x, float y)
{
    float dx = x - startingPosition.x;
    float dy = y - startingPosition.y;

    setGravityScale(1);
    applyLinearImpulse(-dx * 80, -dy * 80);

    trailTimer = Timer.callRepeatedly(this, "leaveTrail", 100);
}
```

5.3.2 Creating Simple Versions of Familiar Android Apps

Irritated Avians demonstrates that Sofia.Micro addresses both issues in much the same manner as Asteroids. Students will be familiar with the Angry Birds game, so being able

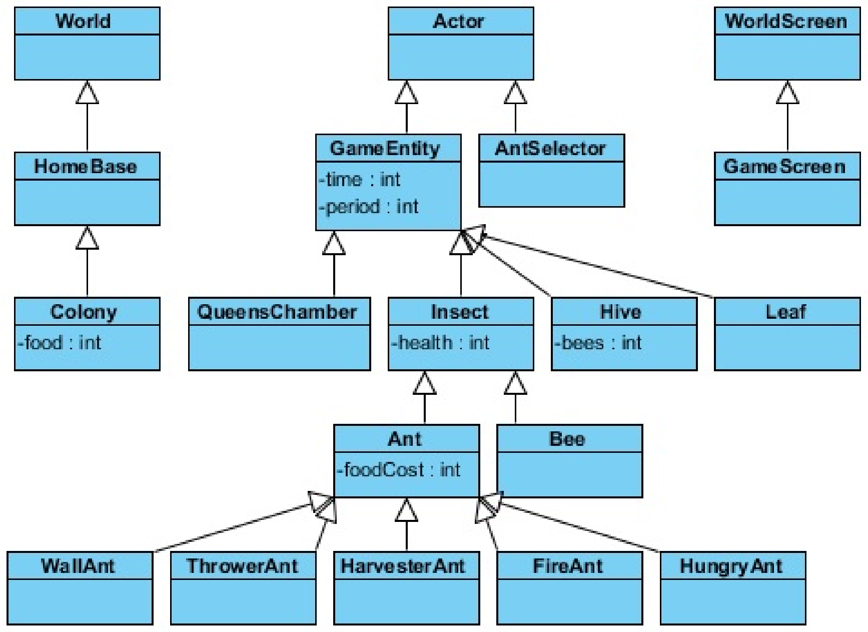


Figure 5.5: Ants vs. Some Bees UML Diagram

to easily write a simplified, working version of it would entice many students. Students can then make use of Sofia.Micro in order to remove much of the complexity to simplify writing the application. Students can continue to use Sofia.Micro to write unique and varied versions of applications that they would be familiar with, which once again, extends Sofia.Micro's

5.4 Ants vs. Some Bees

For the last case study, a version of Plants vs. Zombies was made with the plants replaced with ants and the zombies replaced with bees. This game was originally created by UC Berkeley as a nifty assignment to help teach object oriented concepts [12]. In this game, the ants are trying to repel the bees by shooting leaves at the bees. If the bees make contact with the ants, then the ants start taking damage until they are removed from the game.

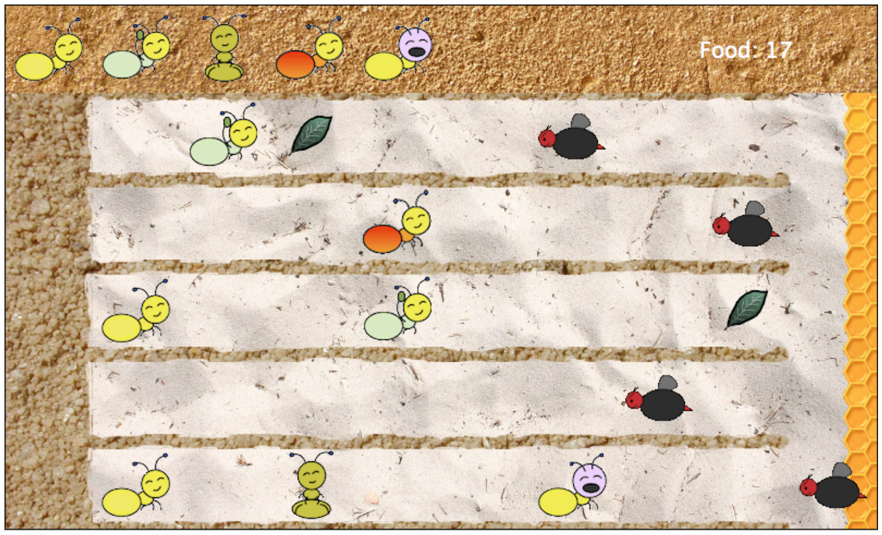


Figure 5.6: Ants vs. Some Bees Screenshot

Students can extend the Ant class to create new Ants that can perform different roles, such as generating extra resources or dealing damage in an area around itself.

5.4.1 Actors and Greenfoot on Android

This case study emphasizes the actor of Sofia.Micro. All the game entities (see Figure 5.5) are actors in this game with all of their behaviors written using the `act()` method. This program demonstrates that Sofia.Micro can support a complete Greenfoot-style program on Android. The projectiles could be rewritten to move using the physics engine, however, a drawback of using both the act method and the physics engine is that they do not move in sync. The game is still largely event-driven since students select and place the ants within the world:

```java
// handler within the Colony.java class for adding new ants to the World
public void onTouchDown(int x, int y)
{
    Ant ant = newActorOfSelectedType();
    if (ant.getFoodCost() < getFood())
    {
        consumeFood(ant.getFoodCost());
        add(ant, x, y);
    }
}
```

5.4.2 Greenfoot-Style Programs on Android of Familar Android Apps

Much like the Irritated Avians case study, Ants vs. Some Bees demonstrates that Sofia.Micro solves the two issues in much the same way. Students can easily create a version of a popular game they would be familiar with, making Android more accessible to introductory students. There is more variation in what students can make so Sofia.Micro continues to remain interesting to students.

Chapter 6

Conclusion

Android has significant potential as a learning tool due to its ability to allow users to write concrete and creative applications. However, due to its learning curve and the fact that it was designed for professionals, it is currently not suitable as a learning environment for introductory programming courses. Sofia.Micro aims to fix this issue by creating a microworld-friendly environment that emphasizes object-oriented and event-driven programming on Android. Sofia.Micro can leverage Android's strengths while removing much of the complexity that both beginner and expert programmers must learn.

6.1 Contributions

In this thesis, we have shown Sofia.Micro's capabilities not just as an Android microworld framework, but also as a pedagogical tool that can do more than other microworlds. Sofia.Micro emphasizes an event-driven programming model, which has been shown to be an effective programming paradigm for introductory students [20, 21, 22]. Sofia.Micro also bridges the gap between different styles of microworlds by supporting both Greenfoot-style cellular automata actors and Karel-style programmable actors. We have also added shapes as non-actors that can interact with other actors and shapes in the microworld in addition to actors and shapes being supported by the physics engine. This gives microworlds the added features of simulated gravity, collision detection, use of non-rectangular shapes, and objects bouncing.

6.2 Future Work

Sofia.Micro still has some improvements that could be implemented to increase its effectiveness. The first is allowing students to interact with actors or the world in a Greenfoot-style. This would allow students to right-click actors and invoke their public methods or right-click actors in a sidebar palette to instantiate them. This would help encourage students to tinker with the worlds on the fly on mobile devices, but this would require a significant amount of work to implement. A full Greenfoot-style IDE would be ideal, however, it is much more ambitious. Second would be to improve the animation support for some of the microworld scenarios, such as the Jeroo example. As of right now, the jeroo will move instantly to each cell rather than move to the cell over a period of time. This would help students visualize what is happening during each act loop. Finally, Sofia.Micro currently only supports basic touch events and some key events, however, there are still other Android-specific events that are not supported, such as using the accelerometer. Improving support for more Android events would allow students to develop more varied applications within the Sofia.Micro framework. On the same note, the dpad could also be improved to allow support for either multiple dpads at once or to allow for on-screen buttons to also accompany the dpad.

Bibliography

[1] "Android Developers." *Developer Support — Android Developers* [Online]. Available http://developer.android.com/guide/index.html [Accessed April 14, 2014]

[2] Alice, "Alice" (1999-2014) [Online]. Available http://www.alice.org/index.php [Accessed April 14, 2014]

[3] "Box2D Documentation." *Box2D v2.2.0 User Manual* [Online]. Available http://www.box2d.org/manual.html [Accessed April 9, 2014]

[4] Letzte Änderung, "Learn Programming with Kara." *Swiss Education - Computer Science - Learn Programming with Kara* (2014) [Online]. Available http://www.swisseduc.ch/compscience/karatojava/kara/ [Accessed April 8, 2014]

[5] Brian Dorn and Dead Sanders, "Jeroo." *Jeroo,* (2008) [Online]. Available http://home.cc.gatech.edu/dorn/38 [Accessed April 8, 2014]

[6] "Scratch." *Scratch - Imagine, Program, Share* (2013) [Online]. Available http://scratch.mit.edu/ [Accessed March 20, 2014]

[7] "Greenfoot." *Greenfoot — About Greenfoot* [Online]. Available http://www.greenfoot.org/door [Accessed April 20, 2014]

[8] "Karel." *Karel J Robot* (1999-2014) [Online]. Available http://www.cafepress.com/kareljrobot [Accessed April 17, 2014]

[9] "Cato's Hike." *Kato's Hike* (2012) [Online]. Available hwahba.com/catoshike [Accessed April 17, 2014]

[10] "Daisy the Dinosaur." *Daisy the Dinosaur* (2012) [Online]. Available http://www.daisythedinosaur.com/ [Accessed April 17, 2014]

[11] "Hopscotch." *Hopscotch - Coding for Kids* (2014) [Online]. Available https://www.gethopscotch.com/ [Accessed April 17, 2014]

[12] "Ants vs. Some Bees." *Object-Oriented Tower Defense: Ants vs. Some Bees* (2014) [Online]. Available http://nifty.stanford.edu/2014/denero-ants-vs-somebees/ [Accessed April 17, 2014]

[13] Anthony Allevato, “Sofia: The Simple Open Framework for Inventive Android Applications,” (2012-2013) [Online]. Available `http://sofia.cs.vt.edu/sofia-2114/book/` [Accessed January 14, 2014]

[14] Stephen Edwards and Anthony Allevato, “Sofia: The Simple Open Framework for Inventive Android Applications,” in *ITiCSE 2013*, Canterbury, England, UK, 2013

[15] Anthony Allevato and Stephen Edwards, “A New Event Dispatch Strategy to Eliminate Dispatch Ğlue¸,” in *RCIS 2013*, Paris, France, 2013

[16] Orni Meerbaum-Salant et. al, “Learning Computer Science Concepts with Scratch,” in *ICER 2010*, Aarhus, Denmark, 2010

[17] Ursula Wolz et. al, “Starting with Scratch in CS1,” in *SIGCSE 2009*, Chattanooga, Tennessee, USA, 2009

[18] Lifelong Kindergarten Group, “Getting Started With Scratch,” (2013) [Online]. Available `http://cdn.scratch.mit.edu/scratchr2/static/__1397013864__/pdfs/help/Getting-Started-Guide-Scratch2.pdf` [Accessed April 8, 2014]

[19] Stephen Cooper et. al, “Teaching Objects-first in Introductory Computer Science,” in *SIGCSE 2003*, Reno, Nevada, USA, 2003

[20] Kim Bruce et. al, *Java: An Eventful Approach*, Upper Saddle River, New Jersey, USA, 2004

[21] Kim Bruce et. al, “Event-Driven Programming Facilitates Learning Standard Programming Concepts,” in *OOPSLA 2004*, Vancouver, British Columbia, Canada, 2004

[22] Kim Bruce et. al, “Event-Driven Programming can be Simple Enough for CS 1,” in *ITiCSE 2001*, New York City, New York, USA, 2001

[23] Kim Bruce et. al, “A library to support a graphics-based object-first approach to CS 1,” in *SIGCSE 2001*, New York City, New York, USA, 2001

[24] Michael Kölling, (2010, November) “The greenfoot programming environment.” *ACM Trans. Comput. Educ.* Volume 10(4), 21 pages Available `http://dl.acm.org/citation.cfm?id=1868361` [Accessed December 26, 2013]

[25] Poul Henriksen and Michael Kölling, “greenfoot: Combining Object Visualisation With Interaction.” in *OOPSLA 2004*, New York, NY, USA, 2004

[26] Michael Kölling et. al, (2003, December) “The BlueJ system and its pedagogy,” *Journal of Comput. Science Educ., Special Issue on Learning and Teaching Object Technology*, Volume 13(4)

[27] Michael Kölling and John Rosenberg, (2010, November) "Guidelines for Teaching Object Orientation with Java." in *ITiCSE 2001*, Canterbury, England, UK, 2001

[28] Werner Hartman et. al, "Kara, finite state machines, and the case for programming as part of general education," in *IEEE Symposia on Human-Centric Comput. Language and Environments 2001*, Stresa, Italy, 2001

[29] Brian Dorn and Dead Sanders, "Using Jeroo to Introduce Object-Oriented Programming," in *ASEE/IEEE Frontiers in Education Conference 2003*, Boulder, Colorado, USA, 2003

[30] Timo Rongas et. al, "Classification of Tools for Use in Introductory Programming Courses," Lappeenranta University of Technology, Finland

[31] Duane Buck and David Stucki, "JKarelRobot: A Case Study in Supporting Levels of Cognitive Development in the Computer Science Curriculum," in *SIGCSE Symposium 2001*, Charlotte, North Carolina, USA, 2001

[32] Sally Fincher et. al "Comparing Alice, Greenfoot & Scratch," in *SIGCSE Symposium 2010*, Milwaukee, Wisconsin, USA, 2010

[33] Ian Utting et. al "Alice, Greenfoot, and Scratch - A Discussion." *ACM Trans. Comput. Educ.* Volume 10(4), 11 pages, 2010