Markus Hobl, B.Sc.

Behaviour-driven development of a 3D programming environment

Master's Thesis

Graz University of Technology

Institute for Softwaretechnology
Head: Univ.-Prof. Dipl-Ing. Dr.techn. Wolfgang Slany

Supervisor: Univ.-Prof. Dipl-Ing. Dr.techn. Wolfgang Slany

Graz, April 2018

This document is set in Palatino, compiled with pdfLaTeX2e and Biber.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, _____ _____

Date Signature

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am _____ _____

Datum Unterschrift

[1] Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Acknowledgments

Ich möchte mich zuallererst bei meinen Eltern Doris und Franz für die Unterstützung in all den Jahren ganz herzlich bedanken.

Desweiteren möchte ich mich bei meiner Schwester Claudia bedanken, welche meine Masterarbeit Korrektur gelesen hat und mir immer ein großes Vorbild war.

Ich möchte mich auch bei meiner Freundin Klaudia für die Unterstützung in den letzten Jahren bedanken.

Abschließend möchte ich mich noch bei Wolfgang Slany und dem Catrobat Team bedanken, welche mir ein unvergessliches Erlebnis ermöglicht haben in einem tollen Team und an einem sehr interessanten Produkt mitzuarbeiten.

Abstract

Behaviour-driven development (BDD) is a software development approach which tries to avoid misleading communication between software developers and stakeholders. It aims to support software development teams to define test cases as such that stakeholders and developers have a common understanding of what these test cases should test/do. As a result, BDD supports software development teams to increase the quality of their software. For implementing BDD concepts in software projects, development teams can make use of different tools. Cucumber which is an easy to use BDD tool to test and document software.
Pocket code, the Android implementation of the visual programming language Catrobat, presents the results of the program execution on a 2-dimensional stage. Catroid3D, is based on Pocket Code, but adds a third dimension for visual programming in a 3D environment, and hence enables the user to make more complex and realistic projects.
Catroid3D is currently in its early alpha stage. It has implemented a 3D engine to render 3D models and an easy user interface to move and rotate the camera to build a simple 3D environment with a preset of included 3D models. This work describes the development steps which have been tested and documented with BDD and Cucumber and provides an outlook on potential next implementation steps in Catroid3D.

Zusammenfassung

Behaviour-driven development (BDD) ist ein Softwareentwicklungsansatz, der versucht irreführende Kommunikation zwischen Softwareentwicklern und deren Interessensgruppen zu vermeiden. Die Konzepte von BDD sollen Softwareentwicklungsteams helfen Testfälle zu definieren, welche von allen Projektbeteiligen verstanden werden können und diese dadurch wissen, was diese Testfälle tun sollten. BDD soll Softwareentwicklungsteams helfen die Qualität ihrer Software zu steigern. Es gibt verschiedene Werkzeuge um die Konzepte von BDD in Softwareprojekten umzusetzen. Cucumber ist ein einfach zu handhabendes Werkzeug für BDD um Software zu testen und zu dokumentieren.
Pocket Code ist eine Android-Implementierung der visuellen Programmiersprache Catrobat. Pocket Code stellt die Ergebnisse der Ausführung des Programms auf einer 2-dimensionalen Bühne dar. Catroid3D basiert auf Pocket Code, aber es fügt eine dritte Dimension hinzu und ermöglicht damit das visuelle Programmieren in einer 3D Umgebung. Es ermöglicht dem Benutzer komplexere und realistischere Projekte zu bauen.
Catroid3D ist aktuell in einer frühen Alphaphase. Die 3D Grafik-Engine, um 3D Modelle zu rendern, wurde implementiert und eine einfache Benutzersteuerung wurde hinzugefügt um die Kamera zu bewegen und zu rotieren und um eine einfache 3D Umgebung aus inkludierten 3D Modellen zu erstellen. Diese Arbeit beschreibt die Entwicklungsschritte, welche durch BDD und Cucumber getestet und dokumentiert wurden und soll einen Ausblick auf die nächsten Schritte in der Entwicklung von Catroid3D geben.

Contents

List of Figures

Listings

1. Introduction

The current software life cycles and update rates have motivated software engineers to rethink the development process of software. Especially the fast-growing gaming industry has been driving this process. The main challenge for developers evolves around testing software as a main criteria for deliviering functioning software, while also delivering the right software. The second point is based on a working relationship and interaction between software developers and business stakeholders. Behavior-Driven-Development is an approach to combine the concept of Test-Driven-Development with the requirements of the business stakeholders and make everybody in the team speak the same language, also called ubiquitous language.

Catrobat is a visual programming language which is developed parallel for different operating systems. The idea behind Catrobat is to omit the pitfalls in which a novice developer can fall, if a standard programming language is used. Catrobat uses a stage, where the user can interact with different objects, costumes and sounds. This stage is currently a 2-dimensional space. Catroid3D is a 3-dimensional Android implementation of Catrobat which will be focused in the next chapters.
This master thesis is titled *Specification by example of a 3D environment* and focuses on the behaviour-driven development process of a 3D version of the Pocket Code project. The first chapter discusses different concepts of Test-Driven-Development and will have a view on Behavior-Driven-Development and the tools for implementing BDD. The second chapter gives a brief overview of the different 3D programming environments and especially the Catrobat and Catroid3D project. The third chapter presents the implementation of BDD with Cucumber for the Catroid3D project. The fourth chapter outlines possible future work and the conclusions are drawn in the final chapter.

2. Behavior Driven Development

2.1. Where it all began

In the early 1970s Royce [20] described a sequential process of software development later known as the waterfall model. The concept was developed to define requirements and design the software, to implement all features defined before and to test the written code. It became clear quickly that the larger the software project, the more time is needed to test the implemented software. A great amount of projects, using the waterfall model, exceeded their deadlines or budgets or both and were therefore canceled. Kent Beck was searching for another process to develop software in time and with a high amount of quality. The findings of his research were described in his book *Extreme Programming Explained: Embrace Change* [2], a process that can be summerized within the agile software development area.

2.1.1. Test-Driven Development

The Extreme Programming (XP) development cycle, explained by Beck, contains several processes which are used to produce the ideal outcome. Test-driven development (TDD) is one of these processes and follows some simple steps to test software in a more efficient way. To understand the basics of TDD it is important to understand the basics of XP.
The XP development cycle runs in iterations which will last one up to three weeks depending on the development team. The starting point of an iteration is a meeting, called planning game, where all members get together and discuss the work that has to be conducted in the next iteration. If a new feature should be added to the current development status the team will

split up the feature into smaller tasks which are manageable in one iteration. Then a group consisting of two programmers takes one task and starts to work on this task based on a test-driven development approach. Beck [3] described a five step cycle to implement TDD:

1. Add a test case
2. Run all test cases and see if one fails
3. Write the code to implement the currently added test
4. Run all tests again
5. Refactor the tests

Firstly a test case will be added to test one functionality of a task. Secondly all test cases will be executed and it is clear that the test case which is not yet implemented will fail, but it is an important purpose that this test case fails before any lines of code are written. Now the time has come to implement the failing test and rerun all test cases to see that the just created test goes well and no other test case is failing because of a possible code break with the new written lines of code. If all tests are passing, the last thing to do is to refactor all tests, hence if an old test case is now irrelevant then update the test or delete it completely.

2.1.2. Acceptance Test-Driven Development

Test-driven development focuses on unit tests, meaning that stakeholders of a software project, for example the customer, the project leader or other non-developing project members, understand that the piece of code performs the required function. However, they do not know if the complete process delivers the full required outcome. For that reason another software development process, the so called acceptance test-driven development (ATDD), has been defined. In case of ATDD the stakeholders do write acceptance criteria for a new functionality which should be implemented, but the cycle is the same as in the TDD process. The acceptance criterion is defined before one line of code has been written. After the implementation the stakeholders and testers can verify by means of the acceptance test, if the piece of code does what the stakeholders wanted it to do.

2.2. BDD - A new software development approach

Why was there a need for a new development process? Dan North, the founder of behavior-driven development (BDD), has worked on several software development projects, where he used agile software development practices such as TDD. There was clear consensus within developers' key questions, independently of the project.

> *"Programmers wanted to know where to start, what to test and what not to test, how much to test in one go, what to call their tests, and how to understand why a test fails. [16]"*

However before this quote will be cut into pieces and discussed in detail, it is very important to understand how traditional projects work and why they are failing, as outlined by Chelimsky et al. [4]. The next section will focus on how traditional projects work and why are some of them failing.

2.2.1. Traditional software development projects

Traditional software development projects follow the waterfall model, described by Royce [20]. Figure 2.1 illustrates the waterfall model where the development process flows top down like a waterfall. At first business stakeholders define requirements for a problem and how it should be solved without any technical details. Secondly the design phase starts and the project team tries to find the right software design for the recently defined requirements. If the design of the software is finished, the development team starts to implement the new software. After the implementation phase the team for testing the software begins to verify if the developed software meets the requirements of the first phase. When all tests have been successfully finished, the software product will be deployed and the maintenance phase begins. During the maintenance phase small bugs will be fixed and updates will be released over a period of time.

Thus why are many projects failing if the waterfall model or an adapted form of this model is used as development process? The years have shown that the later a defect is found in the process, the more expensive it is to fix this defect which leads to the first problem delivering software too late or over budget.
The second problem evolves from misunderstandings between stakeholders and developers, e.g. when the team for testing software detects a requirement which has not been met by current implementation or worse the user who buys the software, detects such a missing requirement. If it turns out then that the project team has developed the false software or partially false software it has to be re-designed, re-implemented and re-tested which will cost additional time and money.
The third problem comes up when the software design was poorly conceived, the implementation and testing process runs through and the software will be deployed. After some years many developers will have left the project team and it will be very difficult for new software engineers to make changes in the source code, thus the software is costly to maintain.

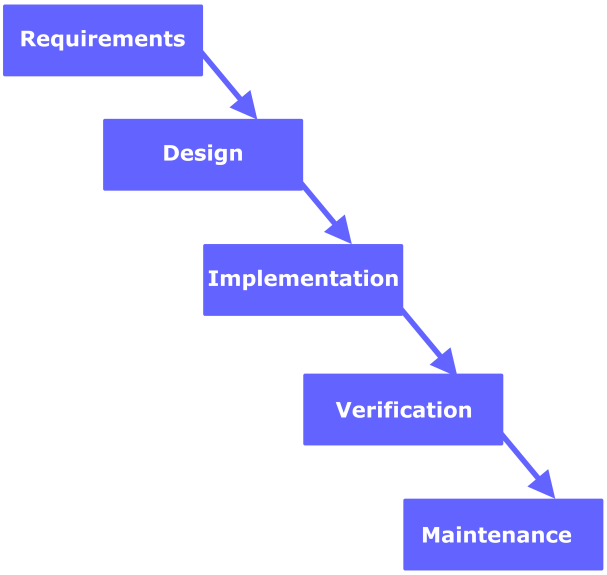


Figure 2.1.: Waterfall model, adapted from Royce [20]

2.2.2. The solution is behavior driven development

The problems described in the section before occur mostly since software development teams and also agile software development teams which have already successfully practiced the XP techniques, misunderstand business stakeholders' requirements. Hence developers implement features which are not relevant for the business stakeholders or, more worse, they do not implement the features which are defined by the business stakeholders.
In the next section the quotation of North [16] will be discussed based on the example of an online shop.

Where to start

A common problem for developers is that they do not know where they should start with the development. The business stakeholders deliver formal descriptions of features which should be implemented, the so called acceptance criteria, for example an online shop checkout. That means acceptance criteria have to be split up into several features which have to be implemented by the developers, for instance the user of the online shop can pay with credit card. The developer takes one feature which is missing in the current version of the project and starts developing. As it should be best practiced in the concept of TDD, firstly the developer writes a test case for the feature and secondly it will be implemented by the developer.

What to call a test

The developer knows the feature which should be implemented. If the project context and the feature are wrapped together, it can be said, the feature should do something in the project context, as in the current example, it should be possible to pay with a credit card. Now the developer has the name of the test case and can write the test case implementation as it can be seen in Listing 2.1.
In the JUnit test framework classes and functions always contain the word test in their names, thus the names were often hard to read and to understand what the test does. Stevenson [24] developed a simple tool, called

```
1 public class OnlineShopCheckoutTest extends TestCase {
2
3   public void testShouldPayItemWithCreditCard() {
4   }
5
6   public void testShouldPayItemWithPayPal() {
7   }
8 }
```

Listing 2.1: JUnit sample test class

agiledox which reads JUnit tests and prints out the names of the classes and functions as written text. Additionally it strips away the word *test* from every phrase. Hence the outcome of agiledox for Listing 2.1 would be something like the following lines in Listing 2.2. These lines are much easier to read and to understand what this test case should test. This conversion makes test cases readable for everyone, hence business stakeholders can also read the test cases and can counter check if the test cases meet the requirements of the acceptance criteria. The project team has now one ubiquitous language as discussed by Evans [8].

```
1 Online Shop Checkout
2   - should pay item with credit card
3   - should pay item with pay pal
```

Listing 2.2: Agiledox conversion of the JUnit example

What to test and what not to test and how much to test in one go

Every feature which is defined by the business stakeholders and the developers, needs to be tested. If a feature can not be described in a way which leads to a sentence similar to 'feature B should do something', then the feature is too complex and should be split up in several tasks. These tasks can be implemented together in a sub class, called dependency injection, meaning that the sub class needs its own test cases for the newly defined tasks. Nevertheless the sub class should be defined in the same way as

before and the feature test case has to ensure the correct behavior of the sub class.

How to understand why tests fail

There are four reasons, why test cases can fail. First a test case can fail, if the implementation of the test case is missing. This is the standard process for a TDD implementation, where the test has to fail first, before the developer starts to implement the feature. This case is the easiest one, the missing functionality has to be implemented until the test case passes. Second, if the developer has introduced a bug into the project, then a test case can fail and the newly introduced bug needs to be fixed. Thirdly a test case can fail, if the implementation of the test case has moved to another business domain of the project, then the test case has to be moved into the same business domain. Last, if a feature has been obsolete, then a test case can fail. In this case the test case needs to be deleted which is often a problem for developers, as they think deleting a test case leads to a bad software quality.

2.3. Tools for implementing BDD

2.3.1. Cucumber

Cucumber, founded by Aslak Hellesoy, is a development tool to help software developers and business stakeholders to communicate with less misunderstandings and to automatically test acceptance criteria, as described in Section 2.1.2, against the system. According to Wynne and Hellesoy [27] it is necessary that developers and business stakeholders are talking about the same thing. Evans [8] described in his book *Domain-driven Design* the need of an *ubiquitous language*, a language which all project members can understand whatever their function in the project team is, business stakeholders and developers can talk on a joint basis. Cucumber provides the possibility to write acceptance tests in spoken language and directly test it against the system.

Cucumber is a command-line tool which interprets so called feature files. An example of such a feature file can be seen in Listing 2.3.

```
1  Feature: Change password
2
3    The password change in the profile menu should be easy
4
5    Scenario: Password successfully changed
6
7      If the user enters the correct old password and enters
8      the new password and repeats it, the user should
9      see a password changed message.
10
11     Given I have chosen to change my password
12     When I enter my old password
13     And I enter my new password
14     And I repeat my new password
15     Then I should see a password changed message
```

Listing 2.3: Cucumber example feature

Features

Features describe available functions for the user of the software which will be tested with Cucumber. As in Listing 2.3 *Change password* is a function which the user can perform, it is a behavior of the software. The text in line 3 describes the feature in detail. The description of the feature should be in spoken language, thus every person can understand what this feature is about. A feature can consist of several scenarios.

Scenarios

Scenarios define different actions which can be performed within the corresponding feature. *Password successfully changed* is an action which can be executed within the *Change password* feature. Other scenarios would be *Old*

password is false or *Repeated password is false*. Every scenario has a detailed description in spoken language, as it can be seen in Listing 2.3 line 7 - 9 which makes it clear to understand, what this scenario should perform.

Steps

Steps describe what has to be done to fulfill the scenario. Every step is one execution in Cucumber and can either be true or false. If only one step fails, the complete scenario fails. In Listing 2.3 line 11 - 15 there are several blue highlighted phrases such as Given, When, And and Then. These phrases follow a special syntax which is called *Gherkin*. Gherkin is a line based language which can either be read by native persons, for example business stakeholders or developers, and can also be automatically read by Cucumber. The Given statement defines a prerequisite for the current scenario, hence these are the first steps which will be executed. Furthermore the When statements define what has to be performed to get a result from the system. Lastly the Then statement is the correct result which should happen after all steps are executed.

Step definitions

Features, scenarios and steps define a Cucumber test case, however this test case can currently not be executed until the step definitions will be defined. The step definitions are the executable statements in the respective programming language, for example Java or Ruby on Rails as it can be seen in Listing 2.4. The equivalent Cucumber steps are matched via the @Given, @When and @Then annotations of the Java methods.

2.3.2. JBehave

JBehave is pure Java implementation of the behavior-driven development process. Dan North, the founder of BDD, developed a tool which helps to support the BDD process, as such JBehave was born [17]. North tried to implement JBehave in a way which makes it self-verifying [16]. JBehave has

```
1  public class ChangePasswordTest extends TestCase {
2
3    @Given("^I have chosen to change my password$")
4    public void i_have_chosen_to_change_my_password() {
5    }
6
7    @When("^I enter my old password$")
8    public void i_enter_my_old_password() {
9    }
10
11   @Then("^I should see a password changed message$")
12   public void i_should_see_a_password_changed_message() {
13   }
14 }
```

Listing 2.4: Cucumber step definitions in Java

its own syntax, hence files can either be formatted in JBehave syntax or can be formatted in Gherkin syntax. Furthermore it has the possibility to integrate it in common Java IDE's or directly in an Ant project or a Maven project for a complete build and execution concept.

2.3.3. RSpec

Steven Baker was inspired by Dan North and his idea of BDD and founded the RSpec project in 2005 as outlined by Chelimsky et al. [4]. RSpec is a BDD implementation for the programming language Ruby on Rails and currently available in version 3.x. The difference between RSpec and Cucumber is in the definition of the behavior. RSpec focuses on the behavior of software elements which will be directly described in the Ruby code. In contrast Cucumber defines separate specifications which will then be executed, hence Cucumber allows to describe the software as a whole.

2.3.4. Specflow

Specflow is a .NET implementation of the BDD concepts and is part of the Cucumber project as it was originally ported from the Cucumber for .NET project. Specflow can parse Gherkin formatted specifications and can execute and evaluate the outcomes. Specflow is available for the .NET, Xamarin and the MONO framework. The reporting tools and some further features, for instance test execution in Visual Studio or parallel test execution, are only available in the paid version Specflow+. The company behind Specflow+, TechTalk, has a lot more projects which fit perfectly together with Specflow, for example SpecLog which handles your feature backlog.

3. Visual 3D Programming Environments

3.1. Alice

Alice was originally developed by the University of Virginia. In 1997 it has been continued by a research group of the Carnegie Mellon University led by Randy Pausch. The idea behind Alice is to give students a possibility to learn the basics of programming in a visual way. In the first version Alice used the Microsoft's Direct 3D Retained Mode (D3DRM) as rendering software as shown by Conway et al. [5]. Firstly Alice gives the user the possibility

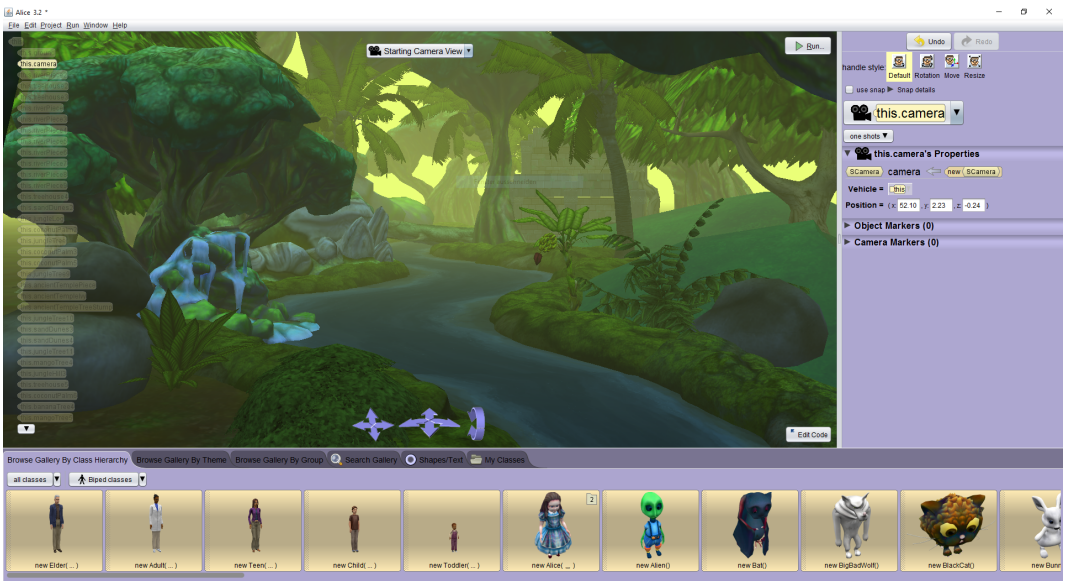


Figure 3.1.: Alice 3: Edit scene screen

to load various 3D models from a library into a scene and to position the model in the scene. Secondly it gives the user the opportunity to animate the models with different program blocks such as 'Move', 'Turn' or 'Play Sound'. Additionally one can aggregate several commands with looping functions for example 'Do In Order' (serial command) or 'Do Together' (parallel command). There are currently two different versions of Alice available, as explained by Dann and Cooper [7], Alice 2 and Alice 3.
Alice 2 is for users that want to get in touch with visual programming. It has a large 3D model library, including all 3D models of the video game *The Sims*, and a great set of preset 3D environments which makes it a lot easier to start as can be seen in Figure 3.1. Alice 3 is a richer and more complex version of Alice 2. It gives the user the possibility to develop in Java or to export the program code in a Java IDE. There is also a plugin for the Netbeans IDE, hence the user can do both implement the objects in Java in the Netbeans IDE and design the world in the scene editor of Alice 3, as depicted in Figure 3.2.

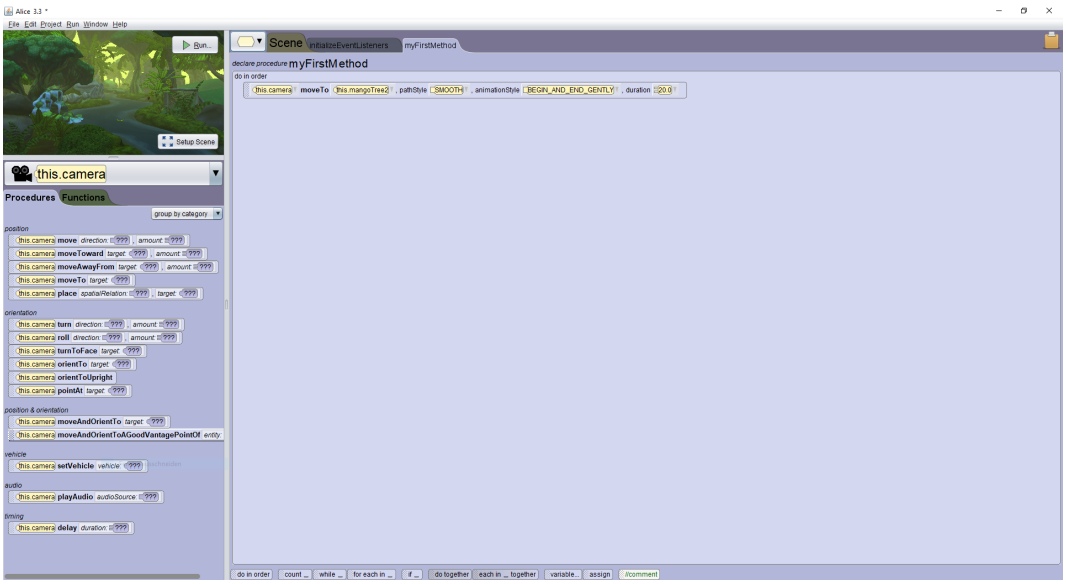


Figure 3.2.: Alice 3: Edit code screen

3.2. Microsoft Kodu

Microsoft Kodu Game Lab, or short KGL, was first released in 2009 for XBOX 360 and introduced as a free version on PC in 2010. MacLaurin [14] mentions that personal computers in the early 1980's like the Commodore PET or the Apple II provided the basis for a tool to design a completely dynamic world. Kodu Game Lab differentiates between objects, sounds, paths and the environment. Objects can be programmed and are all entities which are either characters, such as a submarine or a person, or environment items, for example stones, trees, fruits. The major difference between characters and environment items is that characters can be controlled by the player. There are two types of sounds available, sound effects such as an explosion or a hitting sound and background sounds such as a blowing wind, as explained by Fowler, Fristce, and MacLauren [10]. Further Kodu Game Lab gives the user the possibility to define paths for objects. So for example

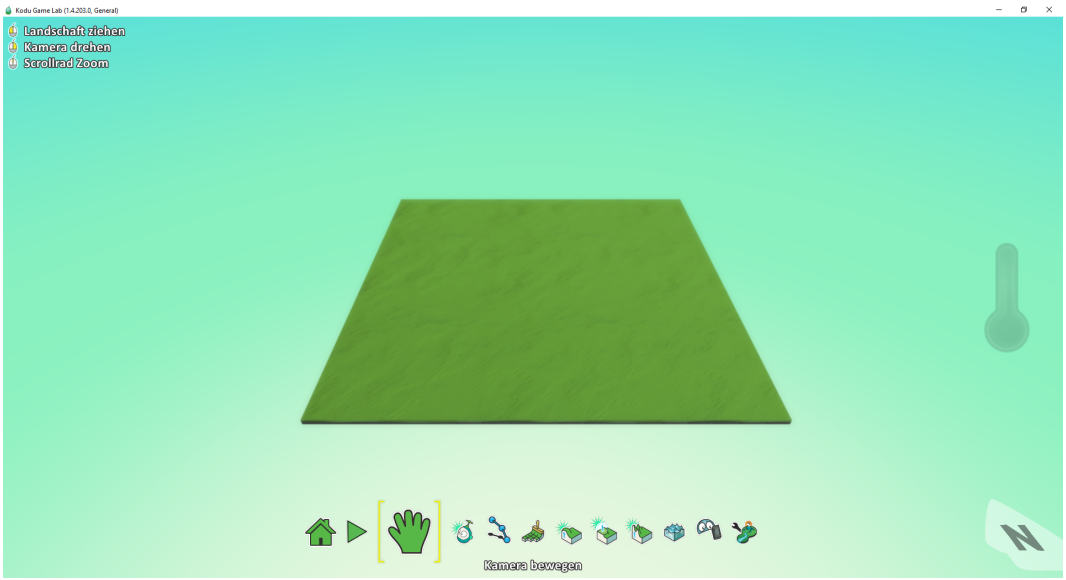


Figure 3.3.: KGL: Emtpy world view

an object can move on a defined path which gives the environment more dynamic feeling. Figure 3.4 shows the definition of a path for an object, where the object follows the blue path. Last but not least KGL provides

features for changing the environment, such as creating hills, build hidden walls, set the overall volume, define day and night cycles or positioning different cameras. All these different possibilities give the user a handy tool to develop the user's own 3D application. Compared to Alice, Kodu Game

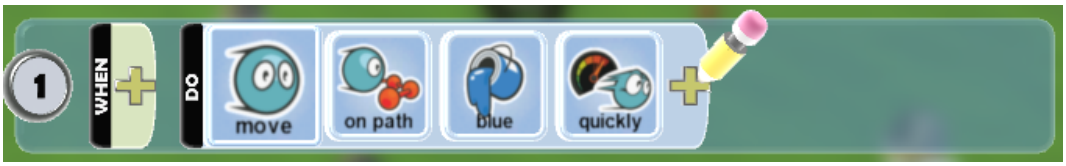


Figure 3.4.: Kodu Game Lab: Definition of a path

Lab is less complex and is therefore easier to use for novice programmers. Furthermore KGL has a highly abstracted graphical user interface (GUI) reducing the failure rate to a minimum as depicted in Figure 3.3. If the user deletes the *move* block in Figure 3.4 then KGL automatically deletes the other three blocks.This is because with the extinction of the *move* block the other blocks do not make sense and the whole process would result in a compiler error. KGL has also implemented easy-to-use features to shape the 3D environment. It enables the user to create hills, valleys and areas with water with a few simple clicks.

3.3. Starlogo Nova

Starlogo Nova is project of the MIT Scheller Teacher Education Program developed by Wendel [26]. It is a 3D programming environment for educational purposes. Starlogo Nova runs directly in the browser and is therefore available on any device with internet connection.
Starlogo Nova has two different windows. Figure 3.5 shows the script editor which enables the user to build program logic based on script blocks. The left view of the script editor lists every available script block. The user can drag and drop the script blocks into the right view to create some more or less complex program code for the current breed. Breeds are different objects which can be switched by clicking on the different tabs.
Figure 3.6 presents the scene view of Starlogo Nova. If the script blocks

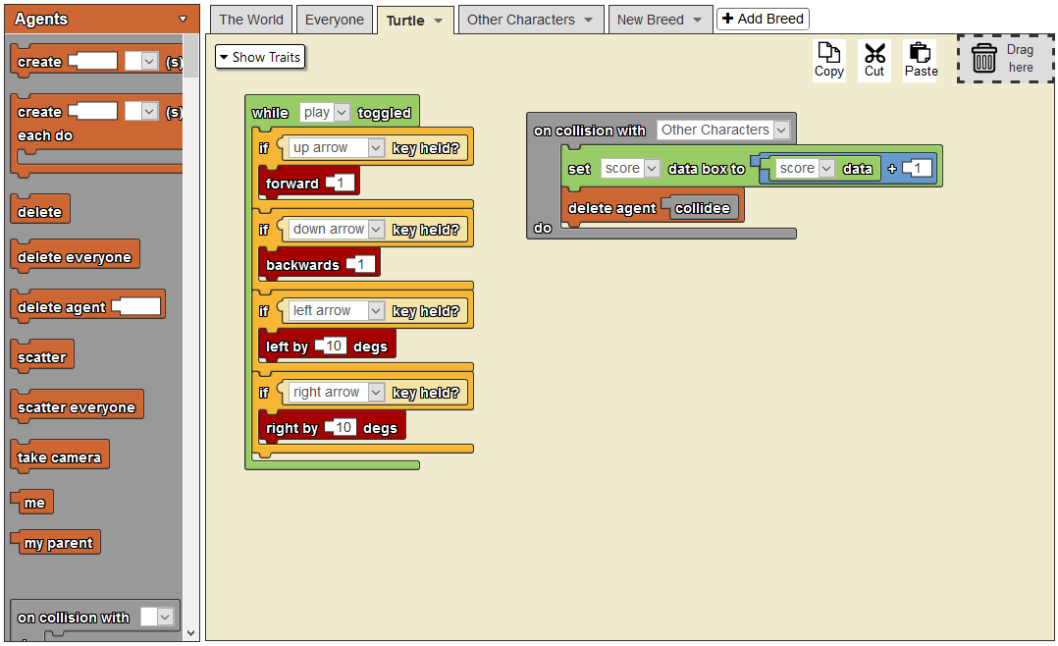


Figure 3.5.: Starlogo Nova script editor

of all breeds will be executed, it will show the result in the scene view. The Starlogo Nova scene view enables the user to add widgets to the user interface. Widgets can either be informative fields such as tables and graphs or can be interactive elements such as buttons and sliders. These widgets can be easily accessed in the script editor to show informations to the user or to process the user input. The speed of the rendering engine can also be changed to slow down or fasten up the rendering process.

In contrast to Alice and Kodu Game Lab, Starlogo Nova has less built-in 3D models, but it supports an import function to add new 3D models to the project. One advantage is that it has many different script blocks which makes it a very powerful 3D programming environment.

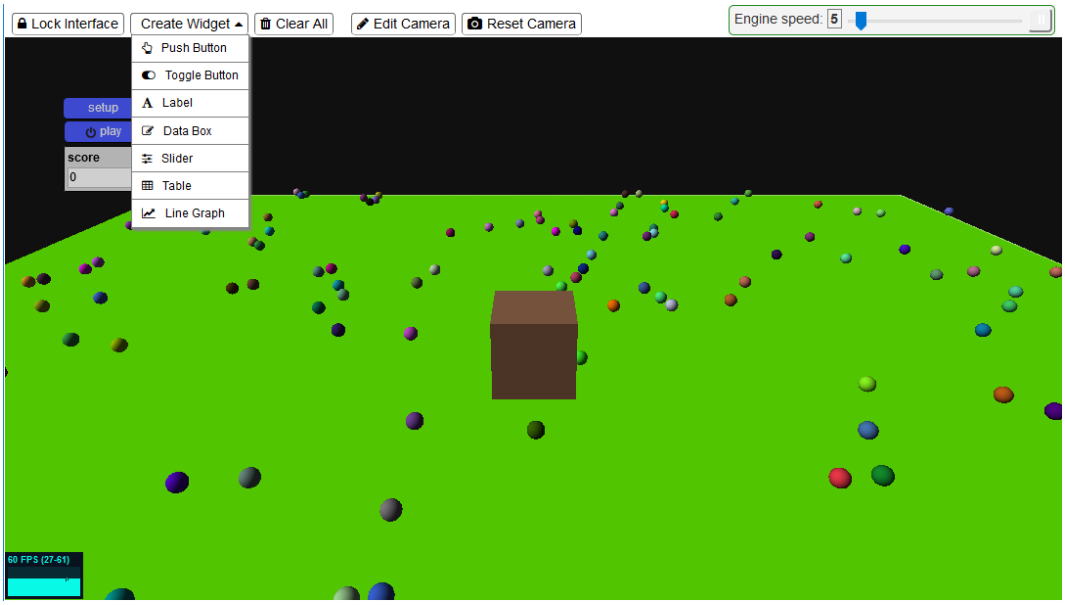


Figure 3.6.: Starlogo Nova scene view

3.4. Beetle Blocks

Beetle Blocks [19] is a visual 3D designing environment which runs directly in the web browser. The main difference to the 3D environments described before, is that Beetle Blocks was developed as a designing tool. It enables the user to create 3D models with basic shapes and different visual programming blocks as can be seen in Figure 3.7. It is based upon Snap! developed by Mönig [15] which is an extended reimplementation of Scratch [11], and three.js a Javascript 3D library.
Beetle Blocks is presented to the user in three views. The left view shows all available script blocks which enables the user to perform different actions onto objects. The user can drag and drop those script blocks from the left view to the middle view. The middle view is the execution area, where all script blocks will be executed. The right view displays the 3D environment, where the objects defined in the execution area, will be drawn as 3D model. Furthermore Beetle Blocks supports the export of 3D models for different standard formats, so that they can be easily re-imported in other 3D environments.

Beetle blocks has an great advantage, as it is based on Snap! the usage is nearly the same. Thus every user who has already written Snap! projects or Scratch projects, can easily write Beetle blocks projects. It has an interesting concept and can be used to create 3D models and export these 3D models to use them in other 3D programming environments such as Catroid3D.

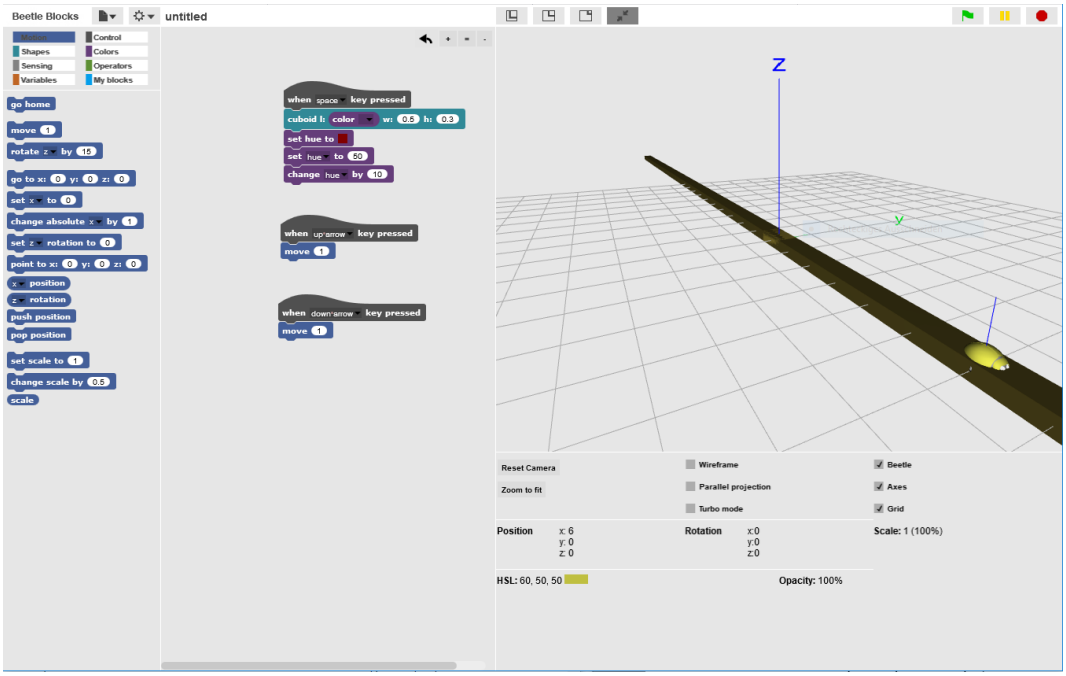


Figure 3.7.: Beetle Blocks Editor

3.5. Superpowers

Superpowers is another programming environment developed by Sparklin Labs [13]. It enables the user to build games either in a 2D environment or in a 3D environment. The difference between Superpowers and KGL and Alice is that Superpowers is developed in HTML5. Hence it requires

a web server, where the HTML5 game can be executed. Superpowers has integrated web servers which can have different configurations for different use cases. Superpowers convinces with a friendly, simple user interface. Every function is available on one screen as can be seen in Figure 3.8.
In contrast to KGL and Alice, Superpowers does not use visual programming blocks. It enables the user to develop the behavior of different objects in Type Script, a programming language based on Java Script. Furthermore Superpowers does not have included 3D models, however new models can be easily imported. But Superpowers has a great advantage. It provides the feature to directly animate objects, for example a running animation of a character.

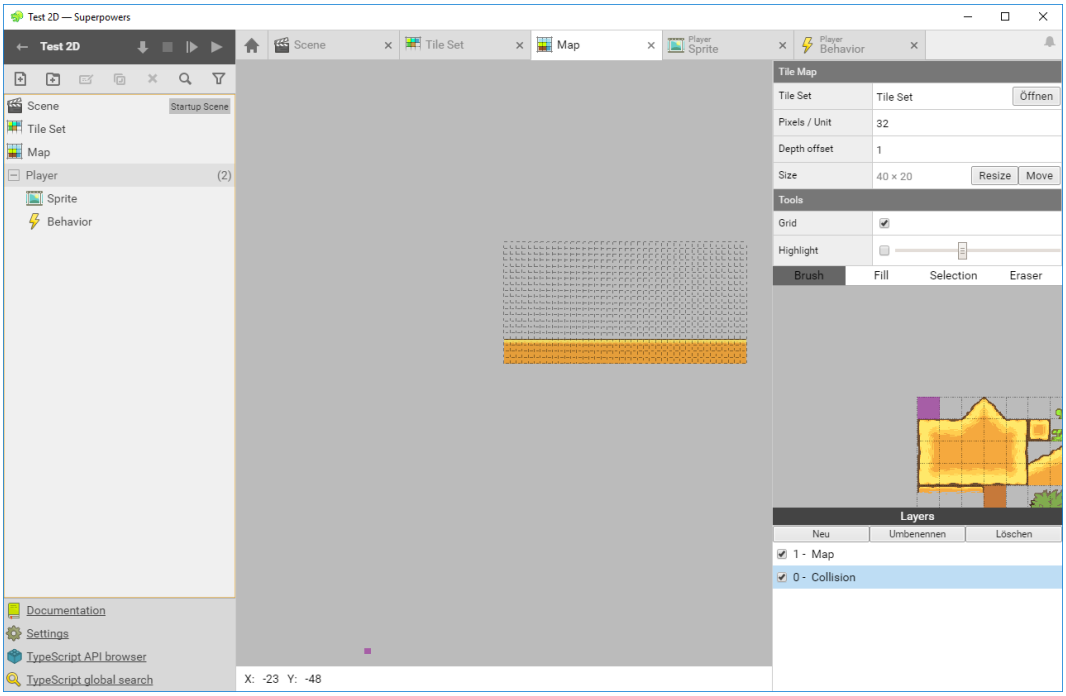


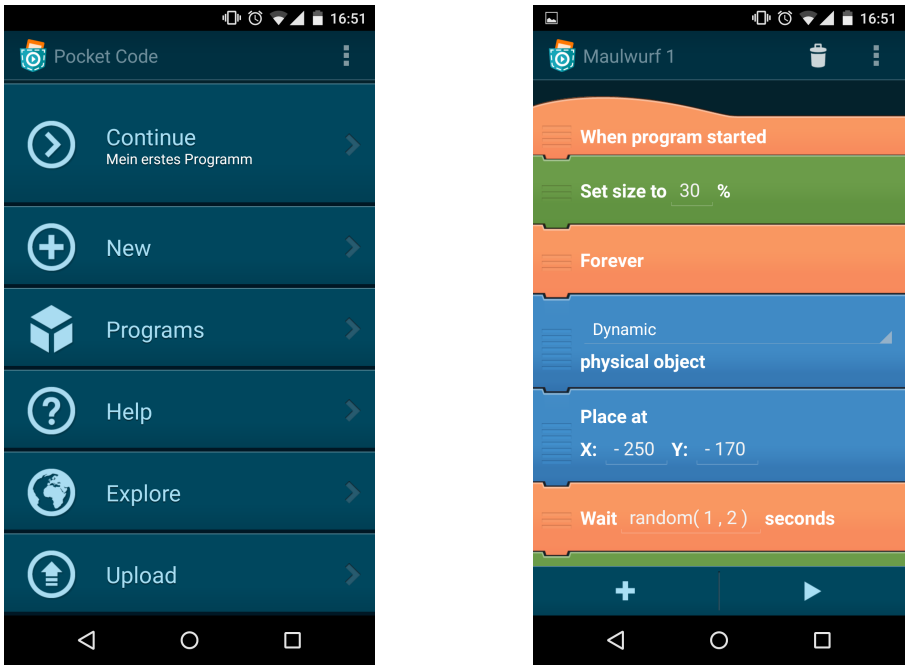
Figure 3.8.: Superpowers editor

3.6. Catroid3D

Catroid3D is based upon the Catroid project which is available in the Google Play Store named Pocket Code [23]. Pocket Code is also developed for other platforms such as iOS or web browsers. Further Pocket Code has its own painting app which is called Pocket Paint and a web page where all Pocket Code projects can be shared with other users. This provides the user the functionality to play and remix other projects directly on the device. 'Remixing' in that sense means to edit an existing project and upload the new version.
Catroid3D is currently in an alpha status. The main objective in the development process was the implementation of the 3D graphic engine, so that 3D models can be rendered correctly. The second objective in the development process was the implementation of object-based 3D models, so that every 3D model can store more information for example physics behavior or program logic.

3.6.1. Catroid

Catroid is a visual programming environment for mobile devices such as smart phones or tablets (Figure 3.9a). The target audience of Catroid are teenagers. Slany [22] argues that the vision behind the Catroid project is the problem of the stagnating amount of computer science alumni in a strongly raising computer science economy. Therefore Catroid should motivate kids to try programming and to prevent them from typical programming problems such as syntax errors, unreadable code bulks and frustrating compiler errors. In contrast to the programming environments described before, Catroid enables the user to create a program solely on the mobile device without any need for a PC, as discussed by Slany [21].
The programming environment is visually displayed as colored blocks, comparable to Lego bricks, as can be seen in Figure 3.9b. There are different bricks for different actions which can be applied onto objects. These bricks are divided into seven categories *Event* bricks, *Control* bricks, *Motion* bricks,



(a)Start screen of Catroid (b)Brick definition in Catroid

Figure 3.9.: Catroid - A visual 2D programming environment

Sound bricks, *Looks* bricks, *Pen* bricks and *Data* bricks. Catroid will also support in the future the user's own bricks called *My Bricks*, where predefined bricks can be stored, edited and used. An object has a defined name and can have several costumes and/or sounds. Costumes are pictures which can either be a photo taken with the camera of the device, imported from the local storage, selected from a built-in media gallery of looks, or directly drawn in the drawing tool Pocket Paint. Sounds can either be recorded with the microphone, selected from a built-in media gallery of sounds, or imported from the local storage. If the user runs a Pocket Code project, the actions behind the bricks will be executed sequentially and the stage will be drawn. The stage is the rendering engine of Pocket Code and shows up the different objects with their different *Looks* and *Sounds*. The user can also interact with the stage, if there are bricks defined which process the user input. Pocket Code projects are stored locally on the SD card, but can be directly uploaded to the Pocket Code sharing web site.

Figure 3.10.: Main menu screen or splash screen

3.6.2. What is the intention behind Catroid3D?

Catroid enables the user to act in a two-dimensional world using an x and y coordinate system. For a first-time or casual user this is normally enough complexity in a programming environment. Catroid3D builds upon the same graphic engine as Catroid does, LibGDX which was founded by Zechner [28], but adds a third dimension to the engine and thus the opportunity to render 3D objects on a smart phone or a tablet. Further it fully integrates the Bullet Physics engine, developed by Coumans [6], for all possible physic mechanics such as gravity and collision detection.

3.6.3. The current version of Catroid3D in detail

The current version of Catroid3D focuses on the rendering part and the possibility to build a simple world on ground elements and a few exclusive 3D models which are integrated in the project. Figure 3.10 shows the start

Figure 3.11.: World view or project build screen

screen of Catroid3D, a splash screen that indicates Catroid3D is starting. Thus Catroid3D loads the world view, a defined standard project in its *ProjectManager* class. The splash screen disposes automatically, when all objects where successfully loaded, and shows the world view, also called project build screen as can be seen in Figure 3.11. Catroid3D consists of an Android Activity, called *WorldActivity,* which starts up the App and runs an *ApplicationListener* class to render all 3D objects. Catroid3D implements its own *Object* base class which is defined by the name of the object, the position matrix in the world view, the mass, the 3D model, the texture of the model and an entity class. The entity class describes the objects physic state, therefore the Bullet Physics library generates a body construct of the 3D model to check whether the object collides with other objects or not. Further a proper class exists, the *World* class which handles all entities and checks periodically if entities collide with other entities.
Catroid3D has included a simple user interface. The user can swipe with the finger to rotate the camera, move the camera, move objects or add new ground elements based on chosen menu on the left side of Figure 3.11. The user can add new objects to the world view by opening the 'Add object'

dialog box which is shown in Figure 3.12.
The next chapter will focus on the implementation of testing the behavior of positioning objects in a 3D environment, detect collisions and how gravity is applied to objects.

Figure 3.12.: Add object dialog box

4. Specification by example of a 3D environment

4.1. Defining the problem

4.1.1. User interface testing with Robotium and Cucumber

There are two parts which are essential for the test implementation. On the one hand Cucumber is used for the behavior-driven test implementation which is described in Chapter 2.3.1. The detailed Cucumber implementation of Catroid3D is part of the next few sections.

On the other hand Robotium is integrated, as it originally was in use in Catroid (now replaced by the Espresso testing framework), as test automation framework to handle automatic user interface interactions. Robotium provides a class called *solo* which gives the user direct access to all UI elements in an Android application.

4.1.2. Using LibGDX graphic engine for the complete rendering of Catroid3D

As described in the previous chapter Catroid3D uses the same graphic engine as Catroid, LibGDX, but it interacts slightly different with LibGDX as Catroid does. Catroid's user interface (UI) is based on the Android view implementation, the base class for all UI objects in Android applications. LibGDX is only used for rendering the graphic objects, such as backgrounds and costumes of Catroid objects in Catroid's stage class, where all visual

effects are shown. The advantage of the Android view class is that Robotium can interact with these UI objects very well.

In case of Catroid3D, LibGDX renders also the complete user interface like buttons, list views or 2D images. LibGDX is build upon OpenGL [18] which is a very fast, high performance graphic processing engine. However Robotium does not support the LibGDX library, therefore a wrapper class for the LibGDX UI implementation and the solo class of Robotium has to be written first. Further the coordinate system of Android which starts at the top left corner of the screen with x-coordinate is 0 and y-coordinate is 0, differs from the LibGDX coordinate system which starts in the bottom left corner of the device screen. Consequently Robotium causes touch events on different positions as LibGDX receives these events.

The next sections will describe in detail how the different functionalities of Catroid3D are tested with Cucumber and Robotium. The description of the different test cases is written in the Cucumber language which means that *scenarios*, *features*, *steps* and *step definitions* are Cucumber specific wordings as outlined in Section 2.3.1. Furthermore the Java implementation of the Cucumber test cases uses the *SoloLibGdxWrapper* class named *solo* which transforms coordinates from the LibGdx coordinate system to the Robotium coordinate system.

4.2. Testing user interface elements

In the current version of Catroid3D two menu elements are implemented. The first menu is a simple main menu which consists of a splash screen and an image to show to the user that the system is loading, as can be seen in Figure 3.10. The main menu can be tapped to switch to the world view a second menu called project build screen. The project build screen, as shown in Figure 3.11, gives the user the possibility to change the camera position, rotation and zoom factor, as well as the opportunity to add objects to the world or remove objects from the world.
These functions can be performed via the buttons at the top left corner of the project build screen as can be seen in Figure 4.1.

(a) (b) (c) (d)

Figure 4.1.: Project build screen menu buttons: (a) move-camera button, (b) move-object button, (c) add-or-remove-ground button, (d) add-object-button

- Move the camera on the x-y plane (Figure 4.1a)
- Move an object on the x-z plane (Figure 4.1b)
- Add or remove a ground element (Figure 4.1c)
- Add a new object to the world (Figure 4.1d)

4.2.1. Defining the main menu

The first cucumber feature file, shown in Listing 4.1, describes the steps which have to be done to see the world view. Apart from that, these steps will be called by every other scenario as a background definition in the feature file. There are three steps which have to be fulfilled to pass the feature test. The first step is the 'Given I am in the main menu' step, where the

```
1 Feature: Main menu
2
3   Main menu is currently a splash screen
4   which can be pressed to see the world view.
5
6   Scenario: The main menu has a splash screen
7     Given I am in the main menu
8     When I press on the splash screen
9     Then I should see the world
```

Listing 4.1: Main menu feature file

initial activity, the WorldActivity, will be started. The SoloLibgdxWrapper

class extends the Solo class from Robotium and adds several functions for transforming coordinates from LibGDX space to Robotium space and getter methods for all kinds of variables.
The *SoloLibGdx* wrapper class starts up a new instance of the *WorldActivity* class which starts the main menu screen as shown in Listing 4.2. The check whether the main menu screen is the active screen can be found in lines 10 and 11. The second step is the 'When I press the splash screen' step where

```
1  public class MainMenuSteps extends AndroidTestCase {
2
3    @Given("^I am in the main menu$")
4    public void I_am_in_the_main_menu() {
5      SoloLibgdxWrapper solo =
6        (SoloLibgdxWrapper) Cucumber.get(Cucumber.KEY_SOLO_WRAPPER);
7      solo.waitForActivity(WorldActivity.class, 5000);
8      solo.sleep(1000);
9      try {
10       assertTrue("I am not in the main menu.",
11         solo.getActiveScreen() instanceof MainMenuScreen);
12     } catch (Exception e) {
13       fail("No active screen!");
14     }
15   }
16 }
```

Listing 4.2: Step definition of the 'I am in the main menu' step

```
1    @Given("^I press on the splash screen$")
2    public void I_press_on_the_splash_screen() {
3      SoloLibgdxWrapper solo =
4        (SoloLibgdxWrapper) Cucumber.get(Cucumber.KEY_SOLO_WRAPPER);
5      solo.clickOnScreen(500, 500);
6      solo.sleep(1000);
7    }
8  }
```

Listing 4.3: Step definition of the 'I press on the splash screen' step

Robotium clicks on the splash screen to skip to the world view, as can be seen in Listing 4.3. Last in the 'Then I should see the world' step, the solo

class waits for a condition which continuously checks if the project build screen is the current displayed screen, described in Listing 4.4.
These the steps will be called by every scenario described in the next few sections, as they all start with steps in the project build screen.

```
1   @Given("^I should see the world$")
2   public void I_should_see_the_world() {
3     SoloLibgdxWrapper solo =
4       (SoloLibgdxWrapper) Cucumber.get(Cucumber.KEY_SOLO_WRAPPER);
5     solo.waitForCondition(new Condition() { ... }, 10000);
6     try {
7       assertTrue("I am not seeing the world.",
8         solo.getActiveScreen() instanceof ProjectBuildScreen);
9     } catch (Exception e) {
10      fail("No active screen!");
11    }
12  }
13 }
```

Listing 4.4: Step definition of the 'I should see the world' step

4.2.2. Defining the project build screen

There are several possibilities where the user can interact with the system, as described in the introduction of Section 4.2. Except for the add-object button all buttons are simple toggle on-off buttons which can either be pressed or not pressed. Consequently only one scenario will be discussed given that the scenarios of the other buttons are similarly implemented. Listing 4.5 represents both the description of the feature as well as the background definition which is based on the steps outlined in Section 4.2.1. As the add-or-remove-ground button is the most complex one, the appropriate scenario will be discussed in detail. Complexity is driven by two functions as both the hidden add-ground button and the hidden remove-ground button will become visible if the add-or-remove-ground button is pressed. Firstly the 'When I press the add-or-remove-ground button' step, as can be seen in Listing 4.6, finds the button by its name and extracts the position of the button onto screen. These screen coordinates are converted into Robotium

```
Feature: Ui elements in the project build screen

  The first screen where the user can interact
  with the system is the project build screen.
  There are several toggle on-off buttons which
  can be checked or unchecked.

  Background:
    Given I am in the main menu
    When I press on the splash screen
    Then I should see the world
```

Listing 4.5: Project build screen feature - background definition

coordinates, because of the different coordinate systems of LibGDX and Robotium and passed to the *clickOnScreen* method of Robotium. Secondly Listing 4.7 shows the next step definition whether the add-or-remove-ground button is checked or is not checked. Further the add-ground button and the remove-ground button should be visible at the same time when the add-or-remove-ground button is pressed. Listing 4.8 outlines the visibility check of the add-ground button. This check is equal for the remove-ground button. Lastly the add-or-remove-ground button will be pressed again and will be unchecked and the add-ground button and the remove-ground button will be invisible again. These step definitions are nearly the same as the only change is the *assertTrue* methods are changed to *assertFalse* methods.

```
  Scenario: Tapping the add-or-remove-ground button twice
   When I press the add-or-remove-ground button
   Then the add-or-remove-ground button should be checked
   And the add-ground button should be visible
   And the remove-ground button should be visible
   When I press the add-or-remove-ground button
   Then the add-or-remove-ground button should be unchecked
   And the add-ground button should not be visible
   And the remove-ground button should not be visible
```

Listing 4.6: Project build screen feature - add-or-remove-ground button definition

```
@Then("^the add-or-remove-ground button should be checked$")
public void the_add_or_remove_ground_button_should_be_checked() {
  SoloLibgdxWrapper solo =
    (SoloLibgdxWrapper) Cucumber.get(Cucumber.KEY_SOLO_WRAPPER);
  assertTrue(solo.isToggleOnOffButtonChecked(
    Constants.UI_GROUND_BUTTON));
}
```

Listing 4.7: Step definition of the 'The add-or-remove-ground button should be checked' step

```
@And("^the add-ground button should be visible$")
public void the_add_ground_button_should_be_visible() {
  SoloLibgdxWrapper solo =
    (SoloLibgdxWrapper) Cucumber.get(Cucumber.KEY_SOLO_WRAPPER);
  assertTrue(solo.isButtonVisible(Constants.UI_ADD_GROUND_BUTTON));
}
```

Listing 4.8: Step definition of the 'The add-ground button should be visible' step

4.3. Testing correct camera behaviour

Catroid3D uses a camera, as most 3D environments do and it is supported by LibGDX, to define the perspective. Hence the camera has an explicit position, rotation and a field of view from where the environment will be seen. These camera parameters can be manipulated by either dragging the finger on the screen for rotating the camera, pinching with two fingers for zooming in or zooming out or clicking the move-camera button for dragging the camera left and right or up and down.
The next sections will describe all camera functions and how they are tested with Cucumber.

4.3.1. Defining camera rotation

When Catroid3D is started and the project build screen is displayed, the standard camera configuration is the function to rotate the camera by swiping the finger over the display. Therefore the first feature which will be

described is the camera rotation feature as can be seen in Listing 4.9. The feature contains a scenario for each direction of swiping. The scenarios are defined as scenario outlines which has the advantage of testing different rotation angles with only one scenario description. This implementation

```
1  Scenario Outline: Swiping my finger to the left
2    When I swipe my finger to the left and rotate the
3          camera around <Degree>
4    Then the camera should rotate to the right and x should
5          be "<x>" and y should be "<y>" and z should be "<z>"
6
7    Examples:
8     | Degree | x   | y   | z   |
9     | 90     | >0  | >0  | <0  |
10    | 180    | <0  | >0  | <0  |
11    | 270    | <0  | >0  | >0  |
```

Listing 4.9: Camera rotation feature - Swiping my finger to the left

tests the angles 90°, 180° and 270°.The given angle will be passed to the solo class which calculates the length to swipe over the display to get the desired rotation angle, as defined in Listing 4.10. The next step is evaluating if the camera is at the right position after the rotation, therefore the *examples* statement in the feature file verifies for each coordinate if the value is positive or negative and passes these arguments as method parameters to the step definition. The *evaluateRotation* method in line 17 and 18 in Listing 4.10 checks the new camera position against the given x, y and z parameters and returns true or false if valid or not valid.
The definition of the three other possible swiping directions is equal to the currently described left swiping scenario. The only difference is the varying x, y and z parameter in the examples statement. Hence these scenarios will be not discussed in detail.

4.3.2. Defining camera moving

Moving the camera means to change the position of the camera on the x-y plane or in other words to move the camera left and right or up and

```
1  @When("^I swipe my finger to the left and rotate the
2          camera around (\\d+)$")
3  public void I_swipe_my_finger_to_the_left(int degrees) {
4    SoloLibgdxWrapper solo =
5      (SoloLibgdxWrapper) Cucumber.get(Cucumber.KEY_SOLO_WRAPPER);
6    solo.swipeLeftForRotation(degrees);
7    solo.sleep(1000);
8  }
9
10 @Then("^the camera should rotate to the right and x should
11         be \"([^\"]*)\" and y should be \"([^\"]*)\" and
12         z should be \"([^\"]*)\"$")
13 public void the_camera_should_rotate_to_the_right(String xShouldBe, String yShouldB
14   SoloLibgdxWrapper solo =
15     (SoloLibgdxWrapper) Cucumber.get(Cucumber.KEY_SOLO_WRAPPER);
16   PerspectiveCamera camera = solo.getCamera();
17   assertTrue(UtilTest.evaluateRotation(
18     camera.position, xShouldBe, yShouldBe, zShouldBe));
19 }
```

Listing 4.10: Step definitions of both steps of the camera rotation feature

down. Listing 4.11 describes the 'Swiping my finger to the left' scenario and its background definition. Further the background definition contains two more steps as the camera-move button has to be pressed before the function for moving the camera is activated. The step definitions of these two steps at line 6 and 7 simply call the solo class to click onto the move-camera button and check whether the button is pressed or not pressed. In the first step the scenario forces the solo class to swipe with the finger to the left and secondly it verifies if the corresponding camera coordinate, in this case the x-coordinate, changes in order to the right direction, as in the above example the x-coordinate has to have a higher value.
All other directions for swiping are similiar and it will be not discussed in detail.

```
Background:
  Given I am in the main menu
  When I press on the splash screen
  Then I should see the world
  When I press the camera move button
  Then I should be in the camera moving mode

Scenario: Swiping my finger to the left
  When I swipe my finger to the left
  Then the camera should move to the right
```

Listing 4.11: Camera moving feature - Swiping my finger to the left

4.3.3. Defining camera zooming

Most touch devices and operating systems allow the user to zoom. Normally the user puts two fingers onto the display and drags both fingers apart from each other to zoom in or drags both fingers together to zoom out. This operation is called pinching. Catroid3D permits the user also to pinch either for zooming in or zooming out. The steps of the scenario are rather simple as it can be seen in Listing 4.12. Firstly the step definition of the 'I zoom in with

```
Scenario: I'm zooming in (drag two fingers apart from each other)
  When I zoom in with my fingers
  Then the camera should zoom in
```

Listing 4.12: Camera zooming feature - Zooming in

my fingers' step, described in Listing 4.13, saves the current camera position in the *startPosition* vector for verification afterwards. Secondly the solo class pinches onto the display to zoom in and lastly the new camera position will be checked against the *startPosition* vector in order to check that the single coordinate values have been changed in the right direction. The scenario for zooming out is equivalent to the steps and step definitions above, besides that the solo class pinches out, hence it simulates the dragging of two fingers together which leads to a change of the coordinate values in the opposite direction.

```
@When("^I zoom in with my fingers$")
  public void I_zoom_in_with_my_fingers() {
    SoloLibgdxWrapper solo =
      (SoloLibgdxWrapper) Cucumber.get(Cucumber.KEY_SOLO_WRAPPER);
    PerspectiveCamera camera = solo.getCamera();
    startPosition.set(camera.position.cpy());
    solo.pinchToZoomIn();
    solo.sleep(1000);
  }
```

Listing 4.13: Step definition of the 'I zoom in with my fingers' step of the camera zooming feature'

4.4. Interacting with 3D objects

The previous sections discussed features for testing interactions with user interface elements and camera functions. This section focuses on the interaction with 3D objects, called entities, in the world view. The current version of Catroid3D supports several functions to add, remove or move different kinds of objects. Further it has a fully integrated physics library, called Bullet Physics which applies gravity to the environment and adds masses to the entities. Furthermore the collision detection functionality from Bullet Physics is integrated in Catroid3D, hence entities can collide with each other and what is more important entities are colliding with the static ground, therefore they stay on the ground.

(a) (b) (c) (d)

Figure 4.2.: Currently available 3D models: (a) palm tree, (b) tropical plant with long leafs, (c) tropical plant with thick leafs, (d) wood barrel

4.4.1. Adding new objects to the environment

The add-object button (Figure 4.1d) in the top left project build screen menu opens a new window where the user can add different kind of objects. There are currently four predefined 3D models available which can be added to the world view as can be seen in Figure 4.2.
Therefore the first Cucumber feature describes the functionality of adding

Figure 4.3.: Splitted add-object window

new objects on the ground within the environment. Listing 4.14 shows the first scenario, in this case a scenario outline with three out of the four models of Figure 4.2. This scenario firstly runs the background definition and secondly opens up the splitted add-object window. On the left side the user can change between ground objects and miscellaneous objects and on the right side the user can select a 3D model which the user is able to add to the world (Figure 4.3). The 3D models are created as entities. Entities have a defined body and can have three types of physic states. The first physic state is *Static and no collision detection*, meaning that the entity has a fixed position within the world, it has no mass and can not collide with other entities, for example a tropical plant which makes the world view look better but has

no relevant function. The palm tree is an example for the second physic state *Static and collision detection*. As before the palm tree has a fixed position in the environment but it can collide with other entities, therefore it is a typical barrier. The last physic state is called *Dynamic and collision detection*.

```
1  Background:
2    Given I am in the main menu
3    When I press on the splash screen
4    Then I should see the world
5    When I press the add-object button
6    Then the choose object split pane should show up
7
8  Scenario Outline: Adding new object from the ground objects menu
9    When I click on the "<image>" image
10   Then the choose object split pane should hide
11   And a "<image>" should be placed in the middle of the ground
12
13 Examples:
14     | image  |
15     | tree   |
16     | plant1 |
17     | plant2 |
```

Listing 4.14: Add object feature - Adding object from ground objects menu

The wood barrel is an example with a defined mass that can collide with other objects, might fall over and change its current position.
If an entity is added via the add-object button then it is placed at the point of origin of the environment, as it can be moved afterwards anyway. The step definition of the last step checks whether an entity with the respective name at the point of origin is rendered through the LibGDX engine or not.

Listing 4.15 describes the steps for adding objects under the miscellaneous category, hence Cucumber has to enter this category via the miscellaneous button shown in line 3. In the current version of Catroid3D there is only a wood barrel available under the miscellaneous category but as it can be seen in the examples it is very simple to add another 3D model to this category and to test if it is correctly placed within the environment.

```
    Scenario Outline: Adding new barrel object from the
      miscellaneous objects menu
    When I press the miscellaneous button
    And I click on the "<image>" image
    Then the choose object split pane should hide
    And a "<image>" should be placed in the middle of the ground

  Examples:
     |  image  |
     |  barrel |
```

Listing 4.15: Add object feature - Adding object from miscellaneous objects menu

4.4.2. Removing objects from the environment

One function has not been discussed until now. It is the possibility to long-click onto an entity to open up a dialog box in the upper right corner with two buttons *Delete* and *Cancel* as it can be seen in Figure 4.4. Listing 4.16 shows the steps for removing an entity from the environment. As before the steps are defined as a scenario outline and can therefore be easily expanded for more entities to be removed. In the current feature file the palm tree and the wood barrel will be removed from the environment. Firstly the entity

```
  Scenario Outline: Removing object from the world view
    When I long click on the "<model>"
    Then the object context menu should show up
    When I click on the delete button
    Then the "<model>" should be removed from the world view

  Examples:
     |  model  |
     |  tree   |
     |  barrel |
```

Listing 4.16: Removing object feature - Remove object from the world view

will be long-clicked. Listing 4.17 shows the step definition of the 'I long click on the model' step, where the method checks which entity has to be long-clicked and then the solo wrapper class extracts the position vector of the

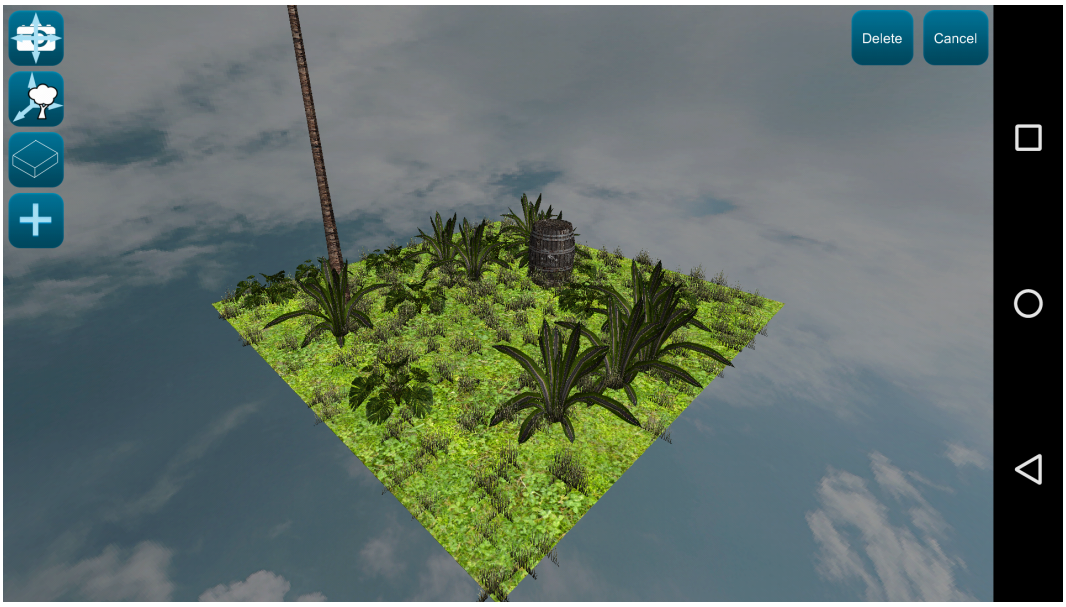


Figure 4.4.: Object dialog box

entity and executes the Robotium long-click method with screen coordinates which are transformed out of the position vector. Secondly the dialog box opens up and Cucumber can execute the third step by clicking onto the delete button to remove the selected entity from the environment. The step definition where Cucumber checks whether the entity is successfully removed from the environment is the same as before except that the model has not be in the list of rendered objects of LibGDX.

4.4.3. Moving objects within the environment

Moving an entity from one position to another is a fairly fundamental function and can be performed in Catroid3D by activating the move-object status. This is done by pressing the move-object button as it can be seen in Listing 4.18, where an optional line is added to the background definition in the moving object feature with the step 'When I press the move-object button'. Afterwards the scenario starts and drags the wood barrel to a new position which is represented in Listing 4.19 where the 'When I drag the

```
@When("^I long click on the \"([^\"]*)\"$")
  public void I_long_click_on_the_model(String model) {
    SoloLibgdxWrapper solo =
      (SoloLibgdxWrapper) Cucumber.get(Cucumber.KEY_SOLO_WRAPPER);
    if(model.contains("tree")) {
      solo.clickLongOnEntity(standardTreeModelName);
    }
    else if(model.contains("barrel")) {
      solo.clickLongOnEntity(standardBarrelModelName);
    }
    else {
      throw new PendingException();
    }
    solo.sleep(500);
  }
```

Listing 4.17: Step definition of the 'I long click on model' step

barrel to left' step is defined. The position vector with the new position is saved as member variable to verify the position after dragging the entity. The *dragEntityToPosition* method of the solo class does not set the wood barrel to the new position, otherwise the dragging function would be entirely excluded. Furthermore the solo class calculates the projection of the wood barrel position and the new position in 3D space to the screen position in 2D space and drags the cursor between these two screen points. This is followed by the verification of the new position of the wood barrel.
As described in the beginning of this chapter, Catroid3D also integrates a

```
  Background:
    Given I am in the main menu
    When I press on the splash screen
    Then I should see the world
    When I press the move-object button

  Scenario: Moving object onto the ground
    When I drag the barrel to the left
    Then the barrel should move to the corresponding position
```

Listing 4.18: Moving object feature - Moving object onto the ground

```
@When("^I drag the barrel to the left$")
  public void I_drag_the_barrel_to_the_left() {
    SoloLibgdxWrapper solo =
      (SoloLibgdxWrapper) Cucumber.get(Cucumber.KEY_SOLO_WRAPPER);
    positionVector.set(200, 0, 200);
    solo.sleep(1000);
    solo.dragEntityToPosition(standardBarrelModelName, positionVector);
    solo.sleep(500);
  }
```

Listing 4.19: Step definition of the 'When I drag the barrel to the left' step

physics engine which applies gravity to all entities with defined masses. The ground plate is an entity with no mass and is therefore ignored by the force of gravity, although it has a collision detection mechanism hence entities can not *fall through* the ground plate. Thus it is very important to verify if gravity is applied to entities and this will be tested by the following feature in Listing 4.20. At first the wood barrel will be dragged to a new position, as in the Cucumber step above, however the position is not on the ground plate therefore the barrel should fall down. Listing 4.21 describes the step definition where the solo class verifies if the wood barrel is falling down. The physics engine calculates the velocity if an entity is accelerated by the force of gravity, hence if the velocity is greater than zero the entity is falling down.

```
  Scenario: Moving object off the ground
   When I drag the barrel to the left off the ground
   Then the barrel should move to the corresponding position
   And the barrel should fall down because of gravity
```

Listing 4.20: Moving object feature - Moving object off the ground

Moving objects against other objects

If the user moves objects within the environment, there is a possible chance to hit other objects. Catroid3D supports three physic states of entities as

```
@And("^the barrel should fall down because of gravity$")
  public void the_barrel_should_fall_down_because_of_gravity() {
    SoloLibgdxWrapper solo =
      (SoloLibgdxWrapper) Cucumber.get(Cucumber.KEY_SOLO_WRAPPER);
    solo.isEntityFallingDown(standardBarrelModelName);
    assertTrue(solo.isEntityFallingDown(standardBarrelModelName));
  }
```

Listing 4.21: Step definition of the 'And the barrel should fall down because of gravity' step

described in Section 4.4.1. The collision detection itself is calculated via the Bullet Physics library which is fully integrated in Catroid3D. Listing 4.22 shows the first scenario and the background definition of the object collision feature file. The background steps start up Catroid3D, as it has been done in the other feature files before, to open up the choose object split pane where a new wood barrel is added to the environment, as shown in Section 4.4.1. The scenario consists of two steps, where at first the newly added wood barrel will be dragged against the already existing wood barrel, secondly the already existing wood barrel will be pushed away and will be checked in the follow up step. As it can be seen in Listing 4.23 in the step definition of

```
  Background:
    Given I am in the main menu
    When I press on the splash screen
    Then I should see the world
    When I press the add-object button
    Then the choose object split pane should show up
    When I press the miscellaneous button
    And I click on the barrel image
    Then the choose object split pane should hide
    And a barrel should be placed in the middle of the ground
  When I press the move-object button

  Scenario: Dynamic object with mass collides with another
            dynamic object with mass
    When I move the barrel towards the second barrel and hit it
    Then the second barrel should be pushed away in the same direction
```

Listing 4.22: Object collision feature - Dynamic object collides with another dynamic object

the 'When I move the barrel towards the second barrel and hit it' step, where the solo class drags the newly added barrel to a new position behind the already existing barrel forcing a collision between the two entities. This is followed by the step definition where Cucumber tests whether the collided wood barrel is pushed away or not. The *checkEntityCollision* method calls an internal Bullet Physics method which checks if the two entities do have a collision and the *isEntityAtPosition* method verifies if the already existing barrel has been moved because of the collision force. The last scenario shown

```
1  @When("^I move the barrel towards the second barrel and hit it$")
2  public void I_move_the_barrel_towards_the_second_barrel_and_hit_it() {
3    SoloLibgdxWrapper solo =
4      (SoloLibgdxWrapper) Cucumber.get(Cucumber.KEY_SOLO_WRAPPER);
5    positionVector.set(0f, 0f, -400f);
6    solo.sleep(1000);
7    solo.dragEntityToPosition(standardBarrelModelName, positionVector);
8    solo.sleep(500);
9  }
10
11 @Then("^the second barrel should be pushed away in
12       the same direction$")
13 public void the_barrel_should_be_pushed_away_in_the_same_direction() {
14   SoloLibgdxWrapper solo =
15     (SoloLibgdxWrapper) Cucumber.get(Cucumber.KEY_SOLO_WRAPPER);
16   assertTrue(solo.checkEntityCollision(standardBarrelModelName,
17     barrelModelNameToHit));
18   assertFalse(solo.isEntityAtPosition(barrelModelNameToHit,
19     Math.createPositionMatrix(new Vector3(0, 0, -200))));
20 }
```

Listing 4.23: Step definition of the both steps of the 'Dynamic object with mass collides with another dynamic object with mass' scenario

in Listing 4.24 describes the collision of a dynamic object with mass and a static object with collision detection, in this case a wood barrel as dynamic object and a tree as static object. The step definition of the 'When I move the barrel towards the tree and hit it' step is the same as in the above step definition in Listing 4.23 except for a different position vector. Catroid3D verifies the step, whether the two entities collide or not, equally as with two dynamic entities. Hence the difference between these two scenarios is the tree which has to stay at the same position after the collision and this will

be checked with the *isEntityAtPosition* method which has to return true if the position vector after the collision is equal to the position vector before the collision.

```
1  Scenario: Dynamic object with mass collides with a static object
2    When I move the barrel towards the tree and hit it
3    Then the tree should stay at its position
```

Listing 4.24: Object collision feature - Dynamic object collides with static object

5. Future work

The previous chapter described the current state of Catroid3D and the test environment. Nevertheless there are more functions which should be implemented in a 3D environment to make it easier for the user to create a 3D game with Catroid3D. Kodu Game Lab, Alice 3 and Superpowers have already implemented a great set of features. This section will focus on functions which would be useful in the the feature set of Catroid3D, and how to implement these functions in Catroid3D. The next few sections outline the required additional work.

5.1. Adopting game logic from Pocket Code from 2D space into 3D space

Pocket Code describes its game logic with different functions the user can work with. These functions are called bricks, referring to the functionality of Lego bricks. In the future Catroid3D should be able to import Pocket Code projects and convert it from 2D space into 3D space. Therefore the bricks have to be converted into a 3-dimensional form. This section will describe if the most important bricks can be adapted and if yes, how they can be adapted to successfully work in the 3D space.

5.1.1. Control bricks

When program started

The 'When program started' brick (Fig. 5.1) defines the entry point of the game logic. It is the first step when the project will be executed. Hence this function does not need an adaption.

Figure 5.1.: 'When project started' brick

When tapped

The 'When tapped' brick (Fig. 5.2) indicates, if an object has been tapped onto the screen. This function can be equally transformed into Catroid3D, whereas objects can also be hit by a tap on the screen.

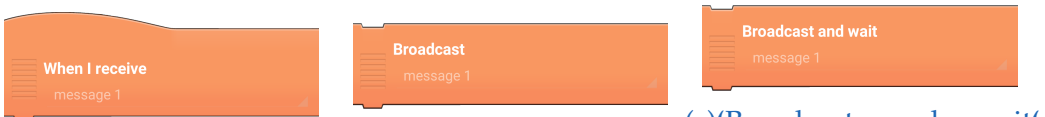
Figure 5.2.: 'When tapped' brick

Flow control bricks

The following bricks do only change the flow of the program, hence there is no difference between a 2D project or a 3D project in case of the program flow. Consequently all following bricks can be directly converted into Catroid3D. The 'Note' brick (Fig. 5.3c) can be pictured as a speech bubble. Thus it looks like the object will say something. All broadcast bricks, as can be seen in Figure 5.4, define points where the program flow can jump similar to a GoTo statement. The 'When physical collision between' brick,

(a)‘Wait x seconds‘ brick (b)‘If x is true then‘ brick (c)‘Note‘ brick

Figure 5.3.: Flow control bricks series 1



(a)‘When I receive‘ brick (b)‘Broadcast‘ brick (c)‘Broadcast and wait‘ brick

Figure 5.4.: Flow control bricks series 2

in Figure 5.5, can be directly converted into 3D space, as there is the same functionality in the current version of Catroid3D. The ‘Forever‘ and the ‘Repeat x times‘ brick can be seen as loops and do not need any modifications.

(a)‘When physical collision between‘ brick (b)‘Forever‘ brick (c)‘Repeat x times‘ brick

Figure 5.5.: Flow control bricks series 3

5.1.2. Motion bricks

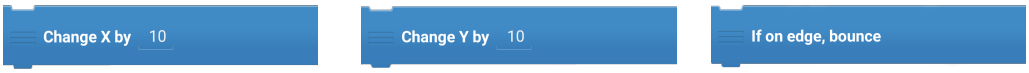
This subsection contains all motion bricks which define different movements objects can be transformed to. Hence Catroid3D adds a third dimension to the world, the conversion of motion bricks has do be described in detail. The ‘Place at x and y‘ brick can be transformed into the given x and y value and an additional z value. As the object should stand onto the ground, the initial z value should be equal the z value of the ground. The ‘Set to x‘ and ‘Set to y‘ brick of Figure 5.6 can be directly converted into the given

x respectively y value without making any modifications to the other two dimensions.
The conversion of the ‘Change x by‘ and ‘Change y by‘ bricks behaves the

(a)‘Place at x and y‘ brick (b)‘Set to x‘ brick (c)‘Set to y‘ brick

Figure 5.6.: Motion control bricks series 1

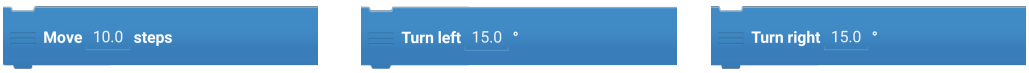
same way as the ‘Set to‘ bricks. The ‘If on edge bounce‘ brick which can be seen in Figure 5.7, checks whether the object has a contact with the edge of the device display. This brick has to be modified for the 3-dimensional space, hence every object can hit another object and not only the edges of the world.
The ‘Move steps‘ brick moves an object in the current direction with a



(a)‘Change x by‘ brick (b)‘Change y by‘ brick (c)‘If on edge bounce‘ brick

Figure 5.7.: Motion control bricks series 2

defined number of steps, where a step is a fixed multiplier. In 3D space this brick is equivalent by moving the object in the direction, where the object is looking. Figure 5.8 shows the ‘Turn left‘ and ‘Turn right‘ brick which turns the object around the z-axis by given degree value. In 3D space these bricks need an additional field for the axis to define direction of rotation.
The next motion control brick series in Figure 5.9 shows the ‘Point in

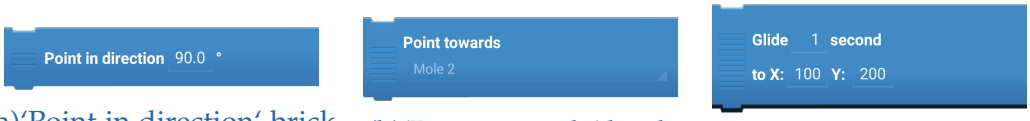


(a)‘Move steps‘ brick (b)‘Turn left‘ brick (c)‘Turn right‘ brick

Figure 5.8.: Motion control bricks series 3

direction‘ brick. It is equivalent to the ‘Turn‘ bricks which also need an additional field for an axis. Further the ‘Point towards‘ brick rotates in the

direction of the given object. This can be converted without any changes, hence the objects direction vector points towards the given object. The 'Glide to' brick glides to a given position over a specified time. This brick needs the third dimension as parameter. As a default value, the z value can be the same as the ground which leads to a gliding over the ground.
The conversion of the 'Go back layer' and 'Go to front' brick in Figure 5.10



(a)'Point in direction' brick (b)'Point towards' brick (c)'Glide to' brick

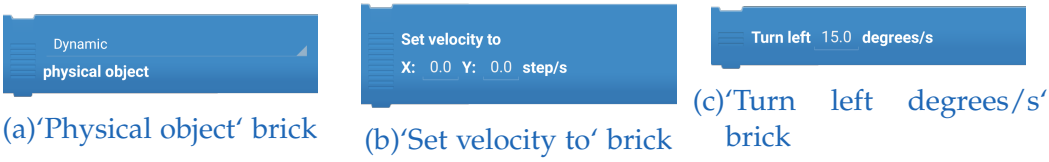
Figure 5.9.: Motion control bricks series 4

can be skipped, since these bricks do only change the z-position of an object in 2D space. Thus the depth of an object in 3D space is defined through the third dimension, these two bricks are not necessary any more in Catroid3D. The last brick of this series, the 'Vibrate for seconds' brick, can be converted without any changes.
The 'Physical object' brick describes which type of physical state the object

(a)'Go back layer' brick (b)'Go to front' brick (c)'Vibrate for seconds' brick

Figure 5.10.: Motion control bricks series 5

has. Pocket Code has currently three different states 'Dynamic', 'Static' and 'No'. These physic states can be converted into 3D space without any changes, since Catroid3D has the same physic object states as described in Section 4.4. Further the 'Set velocity to' brick, as can be seen in Figure 5.11, sets the velocity of an object to a defined value. Hence the object will move in the direction of the objects direction vector. The 'Turn left degrees' and 'Turn right degrees' brick rotates the object by a given amount of degrees around the z-axis either in the left or the right direction. The 'Set gravity to' brick defines a acceleration vector for the x-axis and the y-axis. The 'Set mass to' brick is the last one in Figure 5.12 and can be directly converted



(a)‘Physical object‘ brick (b)‘Set velocity to‘ brick (c)‘Turn left degrees/s‘ brick

Figure 5.11.: Motion control bricks series 6

(a)‘Turn right degrees/s‘ brick (b)‘Set gravity to‘ brick (c)‘Set mass to‘ brick

Figure 5.12.: Motion control bricks series 7

into Catroid3D, as there is also a mass for all 3D objects available. The last motion control brick series in Figure 5.13 contains two physic bricks. The ‘Set bounce factor to‘ brick indicates how hard or soft will the object be bounced back from colliding with other objects. This value is defined in Bullet Physics as the restitution value of an object. Furthermore the ‘Set friction to‘ brick can also be directly transformed into the equivalent friction value of an object in Bullet Physics.

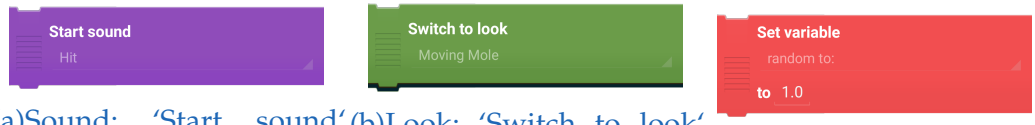
(a)‘Set bounce factor to‘ brick (b)‘Set friction to‘ brick

Figure 5.13.: Motion control bricks series 8

5.1.3. Sound, look and data bricks

The sound, look and data bricks are the last three categories of possible bricks in Pocket Code. As these bricks do not have any restrictions regarding a conversion to 3D space, Figure 5.14 shows one brick for each category as an example. The sound bricks are used to define different audio files for different objects and actions and to play or pause these files. The look bricks

give the user the possibility to change the look of an object, thus Catroid3D needs the possibility to change the model of an already existing object which has to implemented in the future. The data bricks are variables which can be defined by the user. The user can write and read variable values to create more complex projects. These bricks can be directly converted into Catroid3D.



(a)Sound: 'Start sound' brick (b)Look: 'Switch to look' brick (c)Data: 'Set variable' brick

Figure 5.14.: Example for sound, look and data bricks

5.2. Executing bricks and showing the result - the stage

Catroid3D has beside the *splash screen* which loads the project in the background and starts up the rendering process, only one window the *ProjectBuildScreen*. The *ProjectBuildScreen* enables the user to define the world and every object in this world. If the bricks and the execution process of bricks will be implemented in Catroid3D as described in the Section 5.1 before, the stage for showing the result of the execution process should also be considered.
Pocket Code starts its stage and sequentially executes the bricks, when the user clicks the play button. Pocket Code presents all objects as 2-dimensional objects and the hole display of the device of the user is the users' point of view. Catroid3D shows all objects as 3D objects which requires a camera position to define the users' point of view. Different types of games require different types of camera positions with different camera behaviors. For example first person games or third person games use a camera which is in the main character or some distance behind the main character and follows the main character automatically which is normally controlled by the user.

Whereas strategy games have more often a camera position in a bird's-eye view which can be changed manually by the user.
Kodu Game Lab enables the user to bind the keyboard or the game pad to an object and this object will be automatically followed in the execution mode. Alice 3 has special camera bricks to follow objects or to let the user manipulate the camera position as depicted in Figure 5.15.

Figure 5.15.: 'MoveAndOrientTo' Procedures in Alice3

5.3. Building a 3D model database

As discussed in Section 4.4, the most recent version of Catroid3D supports four implemented 3D models. Hence there is a huge potential for adding more models to the application or as a better solution build a 3D model online database, where the models can be downloaded and saved onto the device. Models are the basis for a successful 3D programming environment, as without a large amount of models the building of a 3D world does not appeal to the user.

Therefore two questions remain to be addressed. From where to retrieve 3D models? How to add a skeleton to a 3D model and animate it, as the normal 3D model is only a body with a texture and can not be animated?

5.3.1. Finding or creating 3D models

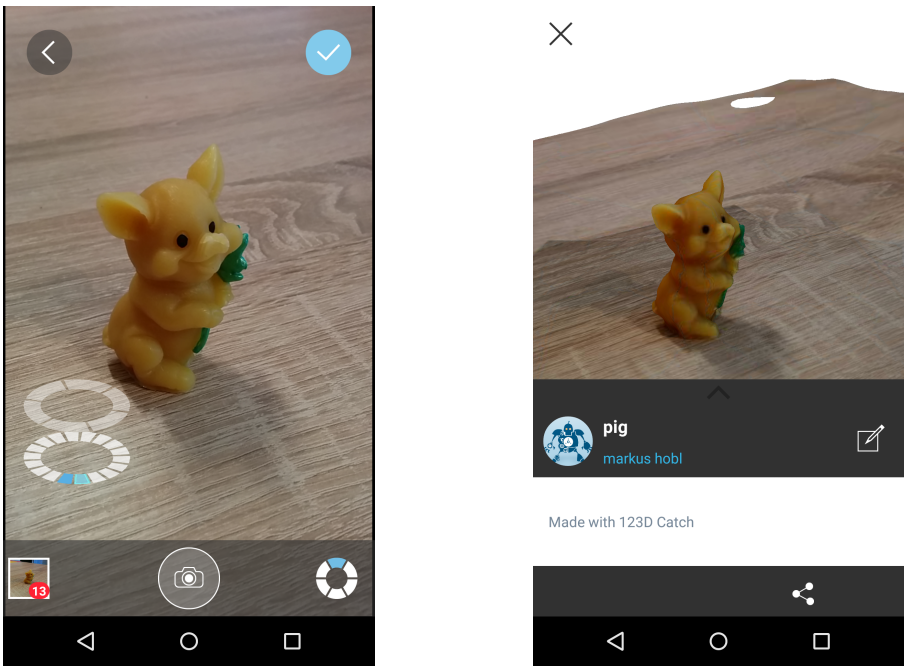
Free online resources

There are several online resources which share 3D models of different artists over a web platform for example opengameart.org, founded by

Kelsey [12]. The world wide web is hosting many more free 3D model websites, but differing publishing licenses could be a problem. In the case of opengameart.org all models are compatible with GPL licenses and therefore compatible with Catroid3D.
A possible solution could be to build a comprehensive database with several other 3D model websites, where all 3D models can be hosted and shared with other persons.

Scanning real objects with a smart phone camera and build 3D models

Since 3D printing is a cheap technology and everybody can use it, the demand for 3D models has raised significantly in the last years. Improving hardware in smart phones, allows building 3D models out of spin images, for example with the Android application Autodesk [1] 123D Catch. Fig-



(a)Scanning a wax pig with 123D Catch (b)Rendered pig from 123D Catch

Figure 5.16.: Scanning and rendering a wax pig with 123D Catch

ure 5.16a shows 123D Catch while capturing images of a wax pig. 123D Catch needs 20 to 40 images, depending on the object, for the modeling process. After all pictures have been taken, the images will be sent to a processing server which calculates the model and sends the 3D model back to the device, as can be seen in Figure 5.16b.

This technology can be used to generate 3D models on the fly and import it directly into Catroid3D or upload it into an online model database.

5.3.2. Animating 3D models

Pocket Code 3D also supports animated 3D models, for example people walking or car wheels turning. Animating a 3D model is not that simple. Firstly the model needs a skeleton which consists of a number of bones and joints. These bones can be transformed to another position or can be rotated around one ore more axis. Secondly the transformation must be recorded over time to generate a fluent movement. Furthermore the 3D model has to

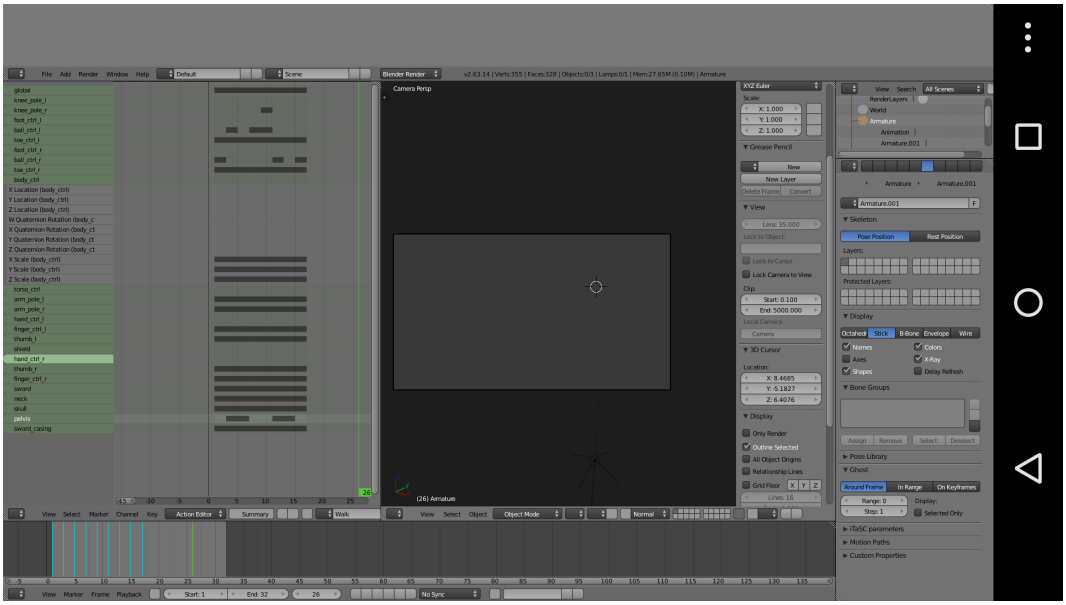


Figure 5.17.: Open source software Blender on Android device

be exported including the animation in a format which can be imported in Catroid3D.
Blender, developed by the Stichting Blender Foundation [9], is an open source 3D modeling and animation software which can be used for the steps described above. Blender is also available on Android, as can be seen in Figure 5.17. Since the full Blender user interface is shown onto the small smart phone screen, it is very hard to interact with Blender onto the device.

Adopting Pocket Paint to use it for 3D models

Pocket Paint is a 2D painting tool for mobile devices, developed by the Catrobat team, to paint a complete new picture or adapt an available picture and import it into Pocket Code. It has different tools for example to change the line thickness or shape and uses a complete color palette.
Adding a third dimension to Pocket Paint could be an idea for future development. Thus 3D models can be adopted directly onto the device.
The concept for a Pocket Paint 3D version has to be changed compared to the current version, hence the 3D version will be more a tool for animating models and less for designing models. The problem for designing 3D models emerges in the rendering process which takes a huge amount of CPU and main memory which will not be available for standard smart phones in the near future. Therefore the tools for painting pictures and the color palette can be removed from the 3D version and new tools for importing 3D models, moving the camera around the different axes and adding a bone structure to the 3D model should be added.
Furthermore Pocket Paint 3D needs a possibility to animate the bones which were added in the step before. Animating a 3D model means, to set the bones to fixed position for each moment in a defined period of time. Thus it will need a time frame which can be adapted by the user and the user can move between every timestamp and define the position of the animated bones. Thus the functionality for moving the bones in different directions and also for moving the camera needs to be implemented. The animating view needs also a play, pause and automatic reloop function, so that the animation can be seen as a complete motion.
Figure 5.18 shows a mock-up for Pocket Paint 3D where a skeleton is build

upon the previously scanned wax pig. This could be a way to firstly scan a 3D object from a real object with a scanning tool as explained before, secondly add a skeleton to the 3D object with a possible Pocket Paint 3D version and lastly animate it directly in Pocket Paint 3D and import it into a Catroid3D project or upload it into a 3D model database as described in Section 5.3.

Given capacity improvements (CPU, RAM) on mobile devices or outsourc-

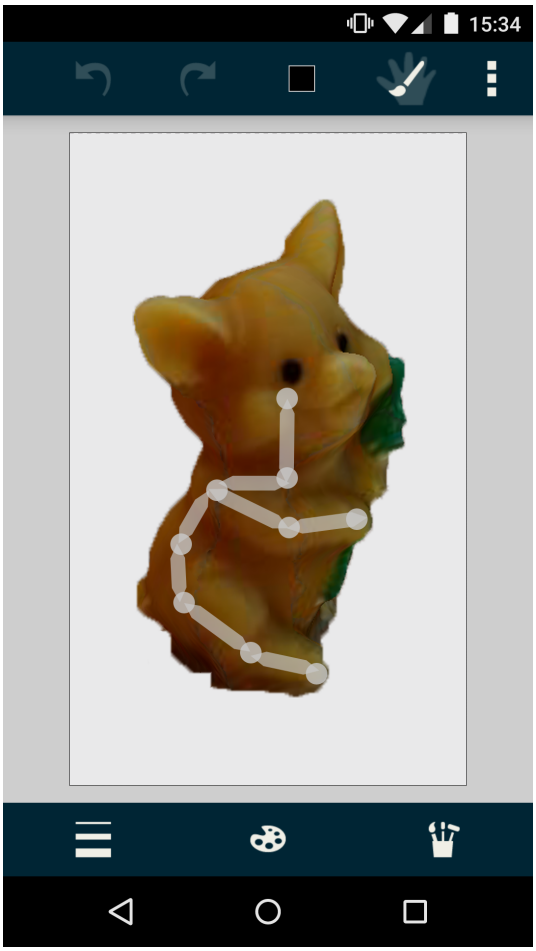


Figure 5.18.: How skeleton building could look like in Pocket Paint 3D

ing opportunities from device to server, there might be a possibility to implement designing and rendering tools comparable to Autodesk's 123D

Catch.
An object can have several different animations, e.g. idle, running, jumping, fighting. Catroid 3D needs an adaption to execute different animations for one object. There should be two new bricks in a future version of Catroid3D the 'Start animation' brick and the 'End animation' brick as can be seen in Figure 5.19. The 'Start animation' brick defines the name of the animation which should be executed, and how often the animation should be executed. Furthermore animations can overlap, for example a person character has an idle animation, if the user runs with this character, it should perform the running animation and if the character stands still it should execute the idle animation again. This animation stack has to be handled by Catroid3D. The 'End animation' brick cancels immediately the selected animation.

(a)'Start animation' brick

(b)'End animation' brick

Figure 5.19.: Possible start and end animation bricks

5.4. Persistent save function and project sharing

Catroid3D does not support a persistent save functionality in its current version. It has an internal project manager class which handles the current project with all objects. This project is currently stored in the main memory and therefore only temporary saved as long as Catroid3D is up and running. The project needs to be stored on a persistent storage, e.g. SD card or hard drive. This functionality is already implemented in Pocket Code and can be inherited in Catroid3D. Furthermore Pocket Code enables the user to upload the projects to the Catrobat sharing website, developed by the Catrobat Team [25], as depicted in Figure 5.20. Therefore the Catrobat sharing website needs also an adaption to preview 2D projects as well as 3D projects.

Figure 5.20.: Catrobat website [25] for project sharing

5.5. Adding new objects and defining object paths

The current version of Catroid3D has a lightweight screen for adding three simple objects to the world. A greater set of objects needs to be added, as discussed in Chapter 5. After a new object is added to the environment, it is automatically set to the point of origin. If more objects are added, then the view can be confusing, because the objects will overlap. The first improvement would be the possibility to set the objects' position directly while adding the object to the environment. KGL enables the user to click onto a position on the ground element and then to choose the object from an object menu, as can be seen in Figure 5.21. The object will be inserted on the position, where the user has made the click on the ground element. Another solution would be to choose the object first, as it is done already in Catroid3D, and then bind it to the cursor to move it to the suitable position.

The next feature is the basis for a dynamic 3D environment. If a user plays a game and all objects in an environment would stay on its original position, it will not feel realistic to the user. Therefore objects should have the possibility to move in different directions to give the environment a more dynamic feeling. KGL uses paths for this feature which enables the

Figure 5.21.: KGL screen for adding a new object

user to create different paths in different directions, where objects can move on, as can be seen in Figure 5.22. If the user runs the game the objects will automatically move onto the defined path and will make a turn if the path ends. KGL enables the user to program objects and paths, e.g. the user can set the velocity of the objects to make them move slower or faster. KGL considers also the gravity of objects which follow a path. If an object is set to a high velocity and has to make a sharp turn, then the object will be dragged away from the path. Afterwards the object tries to get back to its original path. This behavior gives the user a very realistic feeling of moving objects. Every path can have a color to differ between the other paths of objects, because programming objects on paths follows always the concept of 'Move on path with color *black*'.
Thus Catroid3D would need additional bricks to implement the object-path behavior, e.g. 'Move on path with color x' brick.

Figure 5.22.: KGL functions for creating paths for objects

5.6. Shaping the world

The current implementation of Catroid3D enables the user to place objects only at y-coordinate is zero which means a completely flat ground. A 3D environment looks more attractive, if it consists of different shapes such as mountains, valleys, hills and lakes. The next improvement for Catroid3D would be a feature for shaping the 3D environment. There are two approaches which would fit in the concept of Catroid3D, block-based shapes and model shapes.

5.6.1. Block-based shapes

KGL has its own block-based shaping tool which gives the user the possibility to create hills by simply clicking on the desired position. This shaping tool enables the user to flatten hills and to create plateaus and to completely remove the hills. KGL defines the 3D environment as a set of blocks as can be seen in Figure 5.23. This is necessary for dynamically changing the shape

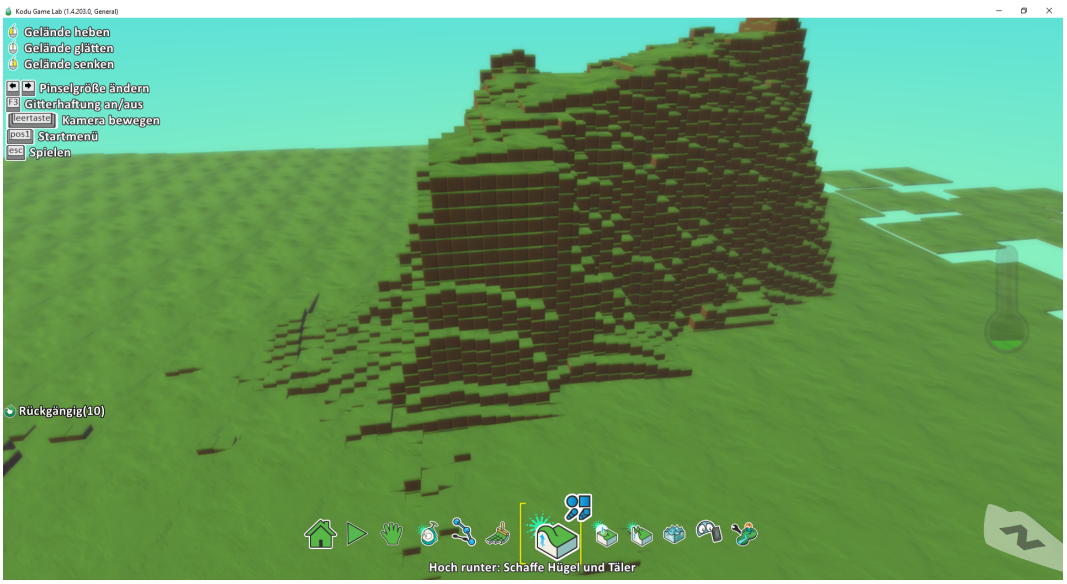


Figure 5.23.: KGL functions for editing the ground

of the environment.
Catroid3D defines the ground elements, in its current version, as thin cuboids, thus it looks like a ground plate. The implementation needs an adaption for block-based shapes. A possible shaping tool in Catroid3D has to create multiple cubes at a defined area to represent the effect of dynamically building hills.
The shaping tool of KGL has also the possibility to define areas with water as depicted in Figure 5.24. It enables the user to click onto an area and fill it with water. The boundary can either be a hill or at least the edge of the current ground plate. The water area is also a block-based shape and can be handled the same way as the ground shaping tool.

5.6.2. Model shapes

Another solution for shaping the environment is using model shapes. Alice 3 uses 3D models to shape the environment as depicted in Figure 5.25. It uses different kind of 3D models to define a shape. The user can set the position of each model by either dragging the model to the desired position or by

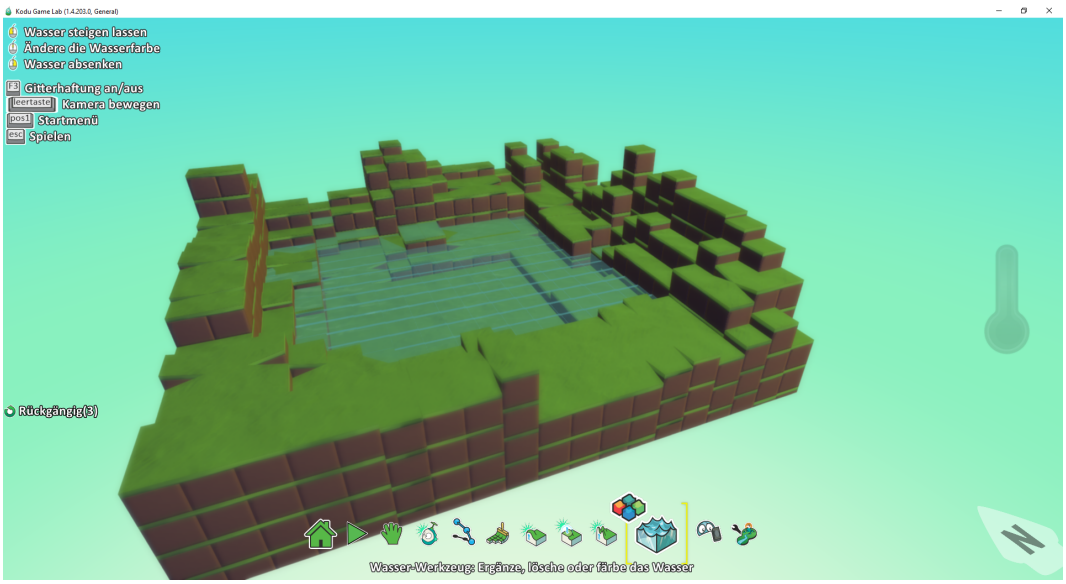


Figure 5.24.: KGL functions for defining areas with water

entering a position vector. The models can be overlapped which enables the user to build associated shapes with a few different models, for example a mountain chain.
The advantage of model shapes compared to block-based shapes is that the user can build more complex and more realistic environments. The disadvantage is that the software has to give the user a great set of different models to build a realistic environment in different regions, for example a desert region, a rain forest region or a nordic region.
Catroid3D can define a model in its current version which makes it easy to implement model shapes. However Catroid3D can move models only at y-coordinate is Zero. This function needs an adaption so it enables the user to enter a position vector with a y-coordinate not equal Zero. In the concept of Catroid3D these models are not objects with a physic state. If these models need be a border line between other objects and the edge of the game environment, then these models have to be defined as objects with the physic state *Static and collision detection* as discussed in Section 4.4.1;

Block-based shapes are an easy and fast tool to create shapes in an environment. Whereas model shapes require more designing ability of the

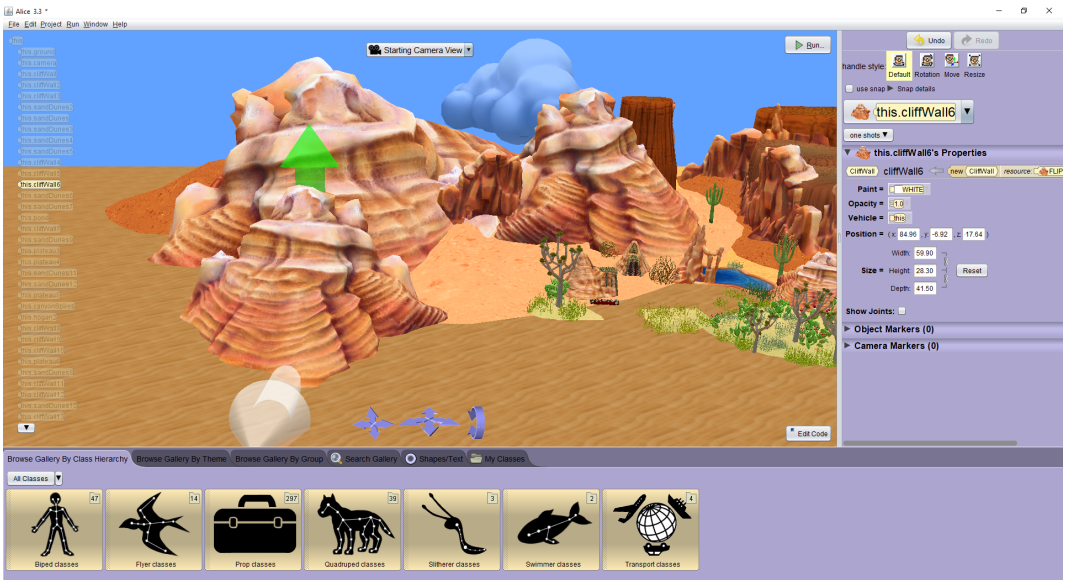


Figure 5.25.: Alice 3 - Defining model shapes

user to make it look realistic. Both concepts have their advantages and also a combined solution would possible. There are more minor changes which would give the user a better user interface. Kodu Game Lab and Alice can help to make Catroid3D a more intuitional 3D programming environment.

6. Conclusion

Chapter 2 outlined different concepts of Test-Driven-Development above all Behavior-Driven-Development and the different tools for implementing BDD. Furthermore distinct 3D programming environments were discussed and the implementation of BDD with Cucumber for Catroid3D was described in Chapter 3. The possible future work for migrating the equivalent bricks from Pocket Code into Catroid3D was shown in the final chapter and it also presented an update for a possible Pocket Paint 3D version. Moreover further features for shaping the environment and for building a more dynamic world were discussed in the last section.

BDD has a lot advantages which can lead to improve a software product. The use of an ubiquitous language is one of the most important concepts of BDD which makes the communication between software developers and business stakeholders a lot easier. Furthermore Cucumber feature files can be directly used as source code documentation as it is written in spoken language. However there are also some disadvantages which occur with Cucumber. If the behavior of a software product changes which specification is already defined in Cucumber, the existing feature files have to be refactored which means to describe the new behavior and to refactor every step definition in the JUnit test framework, as the step definition is the exact description of the step in the Cucumber feature file. If there are some breaking changes for future releases of a software product, the amount of work for refactoring the feature files and the unit tests can be very high.

Catroid3D is still in its very early stages, where the basic 3D functionality is tested and implemented. It enables the user to render different 3D models and to set different physic states. Moreover it also supports animated 3D models and gives the user the possiblity to move every object to a different position in the environment. However further development is required to

present an alpha version of this project. A key point will be the implementation of a peristent save functionality, given that Catroid3D currently has an internal project structure, where all 3D objects including all informations, such as position vector, physic state, 3D model and textures, are stored in the main memory, but not be saved on a persistent storage. Furthermore the Catroid3D project is not integrated in the Jenkins test environment, created by the Catrobat team which makes it possible to automatically execute all tests defined in the project. The final chapter described the migration of bricks from Pocket Code to Catroid3D. Another key implementation will be the logic behind the execution of the bricks or in other words the execution of the Catrobat language. Furthermore Catroid3D supports only a few 3D models in the current version. There must be the possibility to either upload own 3D models to Catroid3D or directly download 3D models from an external 3D model database.

Catroid3D in combination with the Cucumber implementation of Behavior-Driven-Development has a big potential and should be pushed to the next level.

Appendix A.

Acronyms

ATDD: Acceptance test-driven development

BDD: Behaviour-driven development

KGL: Kodu Game Lab

TDD: Test-driven development

UI: User interface

XP: Extreme programming

Bibliography

[1] Autodesk. *123D Catch*. URL: http://www.123dapp.com/catch/ (cit. on p. 57).

[2] Kent Beck. *Extreme Programming Explained: Embrace Change*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN: 0201616416 (cit. on p. 3).

[3] Kent Beck. *Test-Driven Development By Example*. Addison Wesley, 2002. ISBN: 0321146530, 9780321146533 (cit. on p. 4).

[4] David Chelimsky et al. *The RSpec Book: Behaviour Driven Development with Rspec, Cucumber, and Friends*. 1st. Pragmatic Bookshelf, 2010. ISBN: 1934356379, 9781934356371 (cit. on pp. 5, 12).

[5] Matthew Conway et al. "Alice: Lessons Learned from Building a 3D System for Novices." In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '00. The Hague, The Netherlands: ACM, 2000, pp. 486–493. ISBN: 1581132166. DOI: 10.1145/332040.332481. URL: http://doi.acm.org/10.1145/332040.332481 (cit. on p. 15).

[6] Erwin Coumans. *Bullet Physics*. 2003. URL: http://bulletphysics.org/wordpress/ (cit. on p. 25).

[7] Wanda Dann and Stephen Cooper. "Education: Alice 3: Concrete to Abstract." In: *Commun. ACM* 52.8 (Aug. 2009), pp. 27–29. ISSN: 0001-0782. DOI: 10.1145/1536616.1536628. URL: http://doi.acm.org/10.1145/1536616.1536628 (cit. on p. 16).

[8] Eric Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004. ISBN: 0321125215, 9780321125217 (cit. on pp. 8, 9).

[9] Stichting Blender Foundation. *Blender*. URL: https://www.blender.org/ (cit. on p. 59).

[10] Allan Fowler, Teale Fristce, and Matthew MacLauren. "Kodu Game Lab: a programming environment." In: *The Computer Games Journal 1(1) Whitsun 2012*. The Computer Games Journal, 2012, pp. 17–28. URL: http://tcjg.weebly.com/uploads/9/3/8/5/9385844/fowler_et_al_tcgj_11_whitsun_2012.pdf (cit. on p. 17).

[11] Lifelong Kindergarten Group. *Scratch*. MIT Media Lab. URL: https://scratch.mit.edu/ (cit. on p. 20).

[12] Bart Kelsey. *opengameart.org*. URL: http://opengameart.org (cit. on p. 57).

[13] Sparklin Labs. *Superpowers*. URL: http://superpowers-html5.com/ (cit. on p. 21).

[14] Matthew B. MacLaurin. "The Design of Kodu: A Tiny Visual Programming Language for Children on the Xbox 360." In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin, Texas, USA: ACM, 2011, pp. 241–246. ISBN: 9781450304900. DOI: 10.1145/1926385.1926413. URL: http://doi.acm.org/10.1145/1926385.1926413 (cit. on p. 17).

[15] Jens Mönig. *Snap!* University of California at Berkeley. URL: https://snap.berkeley.edu (cit. on p. 20).

[16] Dan North. *Introducing BDD*. 2006. URL: http://dannorth.net/introducing-bdd/ (cit. on pp. 5, 7, 11).

[17] Dan North. *JBehave*. 2003. URL: http://dannorth.net/introducing-bdd/ (cit. on p. 11).

[18] OpenGL. *OpenGL*. 1997. URL: https://www.opengl.org/ (cit. on p. 30).

[19] Eric Rosenbaum et al. *Beetle Blocks*. URL: http://beetleblocks.com (cit. on p. 20).

[20] Winston W Royce. "Managing the development of large software systems." In: *proceedings of IEEE WESCON*. Vol. 26. 8. Los Angeles. 1970 (cit. on pp. 3, 5, 6).

[21] Wolfgang Slany. "A mobile visual programming system for Android smartphones and tablets." In: *Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on*. Sept. 2012, pp. 265–266. DOI: 10.1109/VLHCC.2012.6344546 (cit. on p. 23).

[22] Wolfgang Slany. "Catroid: A Mobile Visual Programming System for Children." In: *Proceedings of the 11th International Conference on Interaction Design and Children*. IDC '12. Bremen, Germany: ACM, 2012, pp. 300–303. ISBN: 9781450310079. DOI: 10.1145/2307096.2307151. URL: http://doi.acm.org/10.1145/2307096.2307151 (cit. on p. 23).

[23] Wolfgang Slany and Catrobat Team. *Pocket Code*. International Catrobat Association. 2013. URL: http://www.pocketcode.org (cit. on p. 23).

[24] Chris Stevenson. *agiledox*. 2003. URL: http://agiledox.sourceforge.net/index.html (cit. on p. 7).

[25] Catrobat Team. *Catrobat Share Website*. International Catrobat Association. URL: https://share.catrob.at/pocketcode/ (cit. on pp. 61, 62).

[26] Daniel Wendel. *StarLogo Nova*. MIT Scheller Teacher Education Program. URL: https://www.slnova.org (cit. on p. 18).

[27] Matt Wynne and Aslak Hellesoy. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Bookshelf, 2012. ISBN: 1934356808, 9781934356807 (cit. on p. 9).

[28] Mario Zechner. *LibGDX*. 2010. URL: http://libgdx.badlogicgames.com/index.html (cit. on p. 25).