NTNU

Norwegian University of
Science and Technology

[Lecture Games] Python programming game

Andreas Lyngstad Johnsen
Georgy Ushakov

Master of Science in Computer Science
Submission date: June 2011
Supervisor: Alf Inge Wang, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

Pythia is a programming game that allows the player to change pieces of their environment through use of the programming language Python. The idea is that the game could be used as a part of teaching simple programming to first year university students. The game should be fun enough for the students to keep playing, teach enough for it to earn a place as a teaching tool, and it should be usable by all students. It should also be possible for a teacher to create their own content for the game.

Pythia was implemented by extending the Python-interpreter Jython and building a game around it. The game was rendered using a simple hardware acceleration library. A simple story was invented and there was some research on learning and programming in games.

A set of levels was made, matching the story and introducing puzzles related to simple programming. These levels were used in testing to collect data on usability, entertainment, and learning. There were also tests of the performance of the game on several systems, and an evaluation was made on creating content for the game.

The game has potential for being used to teach programming to first year students, as testers found it to be both fun and educational. We do not know if it would be possible to use it, as it does not currently run on thin clients. If students can run it, we feel that it should be possible for teachers to create puzzles that emulate the teaching goal.

Preface

The idea of a game where the player can change their environment at will by using programming was formed gradually through the later years of our university education. While the early experimenting focused mostly on java, by the time we were about to write our master thesis, we had discovered that using an interpreted language like Python was much more ideal. We presented our idea to our to be supervisor for the project, Alf Inge Wang. He found the idea very interesting and okayed it right away.

Thanks

- Many thanks to Elizabeth Corbett for testing of the game, and proof-reading of this report.
- Thanks to all the other testers.
- Thanks to the Jython-users mailing list for help and testing.
- Thanks to our supervisor, Alf Inge Wang for putting up with our undisciplined work methods and for initiating the project.

Andreas L Johnsen - 15th of june, 2011.
Georgy Ushakov - 20th of june 2011.

Contents

List of Figures

List of Tables

Part I

Introduction and research

Chapter 1

Project Context

This is the report for a master thesis in Game Technology by students at the Norwegian University of Science and Technology (NTNU). This project is a part of the research project Lecture Games which is run by Alf Inge Wang and Bian Wu. It is related to the basic information and communication technology course at NTNU (ITGK).

Chapter 2

Project definition

The goal of this project is to make a programming game that can be used to teach programming in a fun manner.

2.1 Game idea

Pythia is a programming game where the player can learn programming and python by moving around in a world driven by python programs and solving simple puzzles by writing his own python programs. The player's programs will affect the objects in the world around his/her character as if he/she was standing inside the program.

2.2 Learning goals

The learning content of the game is supposed to be based around the module "IT-Grunnkurs" at NTNU. This ranges from the very simple to basic sorting algorithms. The player should be able to master -or at least have a better understanding of- these things by the time they complete the game.

2.3 Entertainment goals

Games should be fun, and this game is no exception. It should be fun enough for players to want to finish the whole game, thus completing the learning goals. We hope to accomplish this by the giving the player a feeling of freedom and achievement.

Chapter 3

Reader's Guide

3.1 Notice to the Reader

This report is only one piece of a larger project. While we have tried our best to give credit to the work done on programming, designing, reading poor documentation and so on, we feel that the project cannot be done justice unless the reader of this report also evaluates the game itself.

3.2 Report outline

This is a quick overview of what will be found in different parts of the report.

Part I - Introduction and research

Part II - Prestudy , and available technology.

Other projects Our researches into other similar projects which feature learning or programming as a part of a game.

Learning Research about learning in video games.

Libraries In this part we will review some alternatives for libraries we can use.

Part III - Game concept and Design A draft of the preliminary design for the project. It starts out by suggesting a story, then moves on to describing the gameplay. Finally, there is a brief discussion of the appearance and GUI of the game.

Part IV - Implementation In-depth discussion about the implementation methods and the challenges faced.

Part V - Results and Discussion End-user test results and analysis.

Part VI - Summary

3.2.1 Appendices

Other information about using our system and some things relating to the project.

3.3 Common Terms

Player and User Our game has two types of users, one being the player who plays the game, and one being the user who creates levels. This can sometimes become confusing. When the word 'player' is used, it will always refer to the person who is playing the game, and experiencing content created by a 'user'. Some places we will also use the word 'tester' to refer to the player.

Pythia The name of the game which was developed during this master thesis.

Java Java[11] is a programming language developed by Sun Microsystems (now owned by Oracle). Java applications are typically compiled to byte code (class file) that can run on any Java Virtual Machine (JVM) regardless of computer architecture.

JVM Java Virtual Machine (JVM)[15] is a virtual machine capable of executing Java byte code.

Profiler A tool used to record information about running application such as method calls, memory allocations and thread states.

Python Python[5] is an interpreted programming language. This means that it does not require compiling, and that it can run small independent snippets of code, in addition to running large programs. Python runs on Windows, Linux/Unix, Mac OS X, and has been ported to the Java and .NET virtual machines.

Jython Jython[3] is the Java-implementation (porting) of Python. It is designed to work closely with Java code and will run on any Java Virtual Machine.

GIT A source code version control tool. Keeps track of the changes in the code.

Maven A java application building tool. It is used to build and deploy java applications.

Content In this document, content will mostly refer to the game's levels and graphics. Anything that isn't an integral part of the system.

Modifiability Modifiability means how simple it is to modify something. This includes how much work must be done, and how much understanding is necessary. In this document it refers mostly to the modifiability of the game content, or the levels.

Performance Performance can mean many things, but in this document it will mean how well the system handles poor hardware, how quickly it responds to input, and how fast it can do a multitude of things, such as loading times, frame rate and executing code.

Part II

Prestudy

Chapter 4

Similar Projects

What we are trying to accomplish should be something that other people would also be working on. There should be some projects going on out there that we can get some inspiration and teachings from.

4.1 Scratch

Scratch [14] is a visual programming language and an online community around it. In Scratch, all programming elements are colored puzzle pieces, to visualize the program syntax. The expressions of a program consist of the pieces for common programming primitives such as while, if, print, and so on. These expressions have grooves on them that indicate that they can be piled onto eachother, making a sequence. A while-piece can fit a boolean expression inside it. Variables fit into one kind of socket, boolean values another. It is all click and drag, the only time you use your keyboard is when you type something that will appear as text to the viewer.

The aim of Scratch is to teach programming to children of ages 10 and up in a simplified manner. Scratch handles I/O through predefined mouse and keyboard capture devices, and a system for playing sounds and showing sprites on a screen. It does not let the user open files or internet connections. This way they ensure that anything that happens does not break with the only visual interface.

Scratch has everything you need inside the same editor, and is very user friendly, if somewhat cumbersome.

4.2 Core wars

Core wars [9] is a game where programs fight each other in a virtual computer. It has no game client or graphics. To play the game you write a program in a programming language called redcode in your editor of choice. Then you can run it in a virtual machine. The aim of the game is to make a program that eats the other programs. Contest are held from time to time where players can submit their

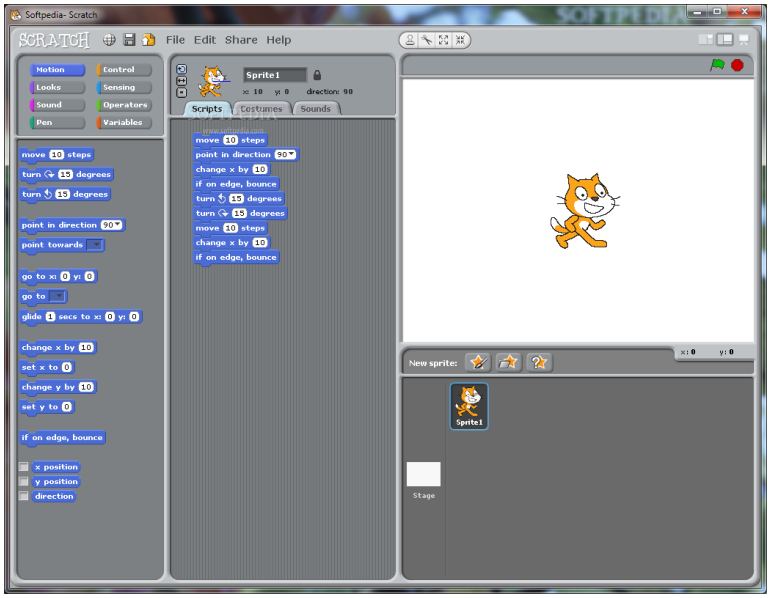

Figure 4.1: Screenshot of Scratch.

programs to battle other programs. You can also write several programs of your own and see which one wins.

It is much less colorful, user-friendly and simple than Scratch is. It is aimed at people who find intricacy and complexity to be interesting (And as such caters to a smaller audience). However, it does employ the basic idea of programming as a game, which is what we are trying to do.

Core wars allows you to use whatever tools and editors you want, and has an open interface that allow for it to be used as a piece of a larger system, such as for instance making genetic algorithms to generate programs and run them against each other.

4.3 Uplink and Hacker Evolution

Both these games allows the player to play the role of a computer hacker, and must break into the virtual computer systems in a simulated computer net. There is no real programming involved but these games are still interesting to us because they use various gameplay elements to emulate hacking. Examples include the console and the concept of virtual operating systems and the virtual hardware.

4.3.1 Hacker Evolution

Hacker Evolution is the simplest of two. The player uses a console as their main tool. through which the player enters commands. This might serve to teach the player a basic understanding of fundamentals of a console and how to use it for simple tasks. The console also builds up under the illusion of hacking as presented in many movies.

4.3.2 Uplink

Uplink (screenshot in figure 4.2) is a much more complex game. In this game a player controls a computer terminal which is connected through a certain gateway in to the game network. The player uses click and drag to run several hacking utilities concurrently to assist them in their virtual, illegal deeds. The programs take up memory and use processing cycles in virtual hardware. The player must balance the load of their programs to achieve optimal performance.

4.3.3 Other elements

Both games feature upgradable hardware. It basically consist of a array of processing units, memory modules and network cards, which affect decrypting performance, free space for stolen data available and transfer speed for data, respectively. Although both games have very little to do with actual hacking education (the games teach players only basic principals like do not leave any trace, and even then only in theory). They (especially Uplink) give the player a good picture of how the computer works internally on a basic level, and serve as a good source of entertainment.

4.4 Civilization

Civilization is a strategy game which doesn't focus inherently on learning or programming, but it is still known for its ability to teach history. The game takes place in a historic setting that allows the player to play through the stone age and all the way to modern time. Along the way, the player will observe certain historic events and can build historic buildings. While the gameplay itself is often entirely incorrect in terms of what actually happened in history, there are still massive amounts of facts surrounding it. There is also the Civilopedia, which is an in-game encyclopedia where you can read facts about any unit, building or people in the game.

Figure 4.2: Screenshot of uplink.

Chapter 5

Research Questions and Methods

This chapter will explain more in detail what we hope to accomplish and to some extent how we wish to accomplish it.

5.1 Requirements

A common way to define what a program is supposed to do, is to gather requirements for the program.

5.1.1 Programming Python

We do not wish players to use external tools while playing, so since our game will be built around on programming in Python, there are a few functional requirements that arise immediately. These are as follows:

Editing code There must be some way for the user to enter their program into the game.

Debugging code There should be some support for debugging the code the user wrote.

Executing code Pythia needs to be able to execute the Python code entered by the user.

Method We will research what options are available to us in order to implement this functionality. There will be a discussion of each option.

5.1.2 Educational Game

Pythia will be an educational game. This means that it should be able to teach something. It should also be fun and keep the players entertained.

Learning The game should teach something about programming in Python. In the best case we want the game to teach the programming basics that are taught in the basic ICT course at NTNU (see chapter 6, section 6.2).

Fun The game should be entertaining enough for the players to feel like they are playing a game and not like they are doing a chore.

Method We will research what other similar projects have done in attempts to achieve this, and see if we can follow their example.

5.1.3 Accessibility

The game should require only Java to run. Everything else should be included in the package.

5.1.4 Resource footprint

The game should run on computers that are a couple of years old. In the best case we want to be able to run it on the thin clients that are available for all students to use at NTNU.

Method We will be profiling our game on different systems from time to time. Closer discussed in chapter 24.

5.1.5 Modifiable

The programming tasks and puzzles should be modifiable, so that a teacher can create his own task for players to do.

Method We will keep the modifiability in mind when we design the architecture of our game. We will also be adding content while we develop the system to see how good it works.

5.2 User testing

The user testing will help us measure to what extent we have succeeded in the more complex requirements, such as learning and fun.

Chapter 6

Learning in Pythia

6.1 Approaches to teach with computer games

"There's been a pretty big divide between games that are meant to educate and games that are meant to entertain."[12] The standard approach seems to be to focus too much on learning something, and not that much on using the advantage you could have over other medias as a result of being a game. The result is often something that is not very entertaining, and also is slightly less effective at teaching you something than simply reading a book about the subject.

6.1.1 Tangential Learning

[16] [12] Tangential learning is the idea that a person might actively look up something they find interesting, if exposed to it as a part of something else. A small fact that is baked into a larger context. In a game, such small facts can also serve to create a sense of realism.

6.1.2 Learning by doing

What sets video games apart from other medias is first and foremost the fact that they are interactive. While a book might contain all the facts you need to know, some things are easier to remember when you try to work with them in some way. Some things might not even be understood at all before you try to do it yourself. There is a lot of anecdotal evidence that one cannot learn math from just reading books, you have to sit down and try to solve some problems in order to both remember it and 'get it. This means that a video game should potentially be a more effective learning media than any other non-interactive media.

6.2 What we want to Teach

Our goal is to supplement the other teaching methods of the IT-Grunnkurs (ITGK) module, which is part of the first year in the following programmes at NTNU: MBIOT5, MLREAL, MTDT, MTIT, MTKOM, MTTK.

6.2.1 Learning goals

The contents of ITGK are one third general theory regarding Information Communication Technology, and two thirds simple programming.

6.2.2 Student problem set

Traditionally the students are presented with a weekly programming problem. It starts off very simple, and gets a bit more complex towards the end. We will make the puzzles in our game reflect the content of these problems to some extent. We hope that our puzzles will cover roughly the same learning as at least some of these problems. This will be our contribution to the two thirds of the module that are related to programming. While a computer game is not needed in order to make students try programming, we hope to motivate a bit more as a result of our game being fun.

6.2.3 Non-programming goals

We hope that by using the right context and story in our game, the player will become more familiar with the subjects that concern ITGK. The context of the game is that everything happens inside a program. We could easily put many facts in there and try to spark the curiosity of the player. We will also attempt to drop hints about the correct approach to solving a problem and have the player look them up on their own accord.

Chapter 7

Libraries

It is common nowadays to reuse packages and libraries. There is no need to reinvent the wheel and waste time on something that was done before.

7.1 Game Libraries

The decision was made to review and choose one or several game libraries to increase speed of development and to utilize hardware. Several frameworks were considered for the project. The final result was a mixture of several libraries which provided some set of features needed by the game engine.

Library requirements

A list of requirements was drafted with following items:

- Fast rendering of 2D
- Utilization of hardware rendering
- Good input support
- Sound support
- Simple text rendering support
- Complex user interface support

The second requirement is the result of the first one. Almost all current computers support hardware acceleration of rendering, and hardware rendering is the best method to insure optimal performance.

Hardware rendering over software

It was decided at the beginning that the game engine will utilize the hardware acceleration to relive the central processing unit of tedious graphic calculations. The reason for such strategy is that these cycles could rather be spent to parse python code, than push frame buffer back and forth. Python code is not pre-compiled. To allow the user to create his or her own levels and create own logic of the game object, the python code must be interpreted in real time. Such calculation would be preformed each logic cycle (the cycle in which all game logic calculations are done) for modules that control game object in the game. Player's custom written scripts are ran in separate threads.

Libraries reviewed

The following rendering libraries were considered for the project.

Canvas/AWT The Abstract Window Toolkit (AWT)[1] is part of the standard Java library. It serves as a bridge to the underlying native interface. This means that how exactly it works might vary from operating system to operating system.

Advantages Relatively simple to implement. No native libraries required.

Disadvantages Hardware acceleration is not guaranteed. Slow compared to the alternatives.

JOGL The Java OpenGL library is a wrapper that allows OpenGL to be used in the java programming language. [2]. This means that it works directly on the graphics hardware. Use of JOGL is somewhat complicated and requires knowledge of OpenGL.

Advantages Hardware acceleration.

Disadvantages Does not really offer anything except for OpenGL bindings. Hard to program. Requires native libraries.

LWJGL LWJGL stands for Light Weight Java Game Library. It is cross platform and includes many smaller libraries that are common to use in games. It is used as a base library for many other (less light weight) game libraries.

Advantages Hardware acceleration. There are a lot of frameworks that work on top of the LWJGL. Offers sound support through OpenAL and good input support. The OpenGL part is simpler than that of JOGL.

Disadvantages Requires native libraries.

Our choice

We have decided to go for LWJGL combined with Slick library. The reason for our choice is the large feature collection of LWJGL. This includes speed, simplicity, smallness, security, robustness and minimalism. The slick library was chosen to privide simplistic way to construct fonts and paint em in a simple way

7.1.1 Mesa3D

Unfortunately the use of OpenGL means no support for the clients without hardware accelerated graphic card. The software emulation can be used as a workaround. One of such libraries is the Mesa3D which is included by default in most Linux distributions. Windows however does not provide such library out of the box and no pre-compiled library is available from the Mesa3D web site. This library must be compiled manually.

7.2 User interface frameworks

Our game, being a very visually focused game, requires a Graphical User Interface (GUI) that allows the player to interact with the game, and that gives him the necessary information. A GUI is something that a lot of systems have, and as such there are many libraries that provide solutions for both designing and managing this. We looked at several.

Three different frameworks were accepted for review.

Themable Widget Library - TWL Feature and component rich framework for building user interfaces on top of the LWJGL. Provides support for windows, pop-ups and basic widgets. Simple HTML rendering, event polling. It has a stand-alone theme editor.

Nifty Very simple framework for building game user interface. Supports simple controls like buttons and edit fields.

FengGUI Another component framework that in some way resembles TWL, but with less features.

Final choice

Unfortunately all of the frameworks suffer from the poor documentation, so a trial and error approach was used to evaluate and choose the one best suited. Nifty was chosen at first, but due to lack of flexibility and strange handling of exceptions it was replaced by TWL. Still TWL was missing some essential widgets for the game. However the framework allowed for easy modifications and creations of new widget components. The custom created widget included vertical-scrollable text field editor and multicolored, scrollable, line number-showing text editor. The

ability to easily modify themes also contributed to the final decision on the choice of interface framework.

7.3 Storage Strategies

We will use XML files for storing data, and a custom file format for level layouts. This is to ensure that the game content is easily modifiable.

Problem

Since we want anyone to be able to create puzzles and levels in Pythia, we want as much of the content to be easily modifiable without recompiling. This is why we want store levels and objects in levels on file, somewhere discoupled from the rest of the game. There are a few common strategies for storage.

Database

A Database system is a system designed specifically for storing and retrieving data efficiently. It allows the users to determine strict data structure with references and types, which makes it robust. This fact also means that there is some extra work every time we change something. It can use a local file, or run remotely. Databases require specific tools for altering them, which is not ideal for fast and simple user modification.

Hard coding

Hard coding levels into the code was a common strategy in the early days of computer games. It is fast and simple, but allows zero modifiability.

Files and Archives

We could define our own files with data in them, as well as means for reading and writing to them. This could be as complex as we want it to be, but would all have to be implemented by us. How simple such files are to modify depend entirely on the implementation.

XML

Extensible Markup Language (XML) is a common way of storing data, with several libraries related to it. This offers storage in files that can be viewed and edited as clear text. There are also several (optional) editors available for advanced users. While the files can be edited freely, our game might not understand anything the user writes in them. To constrain what XML is allowed, we could define an XML name space for our game XML files. This would also allow for quicker editing in more advanced editors. A drawback of XML files is that they tend to be overly verbose, and the accompanying libraries tend to be very resource-heavy.

Our Choice

We will be using mainly XML files for storing data. In addition we will use a simple custom file format for level layout. This will also be easily editable.

7.4 XML Libraries

There are many libraries and strategies for reading XML files. They are mostly divided into the ones that fire events sequentially while reading the xml (such as Simple API for XML, SAX [6]), and the ones that read (unmarshall) the entire document into memory and gives random access to the document. In the Java standard library these are included as "The Java API For XML Processing" (JAXP) and "The Java Architecture For XML Binding" (JAXB).

Choice

Since we are not interested in reacting to the XML documents in any way, but only reading the data that is in them, we will be using the system that creates Java objects for us (JAXB). This way we can define what values in an object should be read from XML, but these objects need to have empty constructors.

Chapter 8

Summary

8.1 Other Projects

There are ways to make programming entertaining, either by giving the player the sense that he is a hacker doing something really complicated and cool (like in Uplink and Hacker Evolution), or by creating an interesting task with complex restraints (as in Core Warriors). Graphical representations of abstract concepts (like the programming elements in Scratch) are a good way to help understanding said abstract concepts. Games can teach without being meant only for teaching (like Civilization). Some games will create interest for a topic in the player, only by using it as the context of the game story.

8.2 Learning

Games that focus on learning are often boring. The best way to teach something to a player playing a game is probably tangential learning, where the game only creates an interest in the player, and the player seeks out knowledge for themselves.

Part III

Game concept and design

Chapter 9

Story

9.1 General

A story is not necessary to make a good game. Many games (pong, tetris, bejeweled) have either no stories, or very thin stories. However, once you wish to pass a certain level of complexity, it is hard to get there without a story.

Definition

The traditional definition of a story is the sequence of words or sentences which are told in order to present some recollection of fictional or real events. In order to create such a story, the writer will often have a setting for the story, real or fictional locations, and many other things thought out, but only the part of these that are actually in the story will be presented to the listener. In an interactive media, these things can be accessed by the player. It is then up to the writer to make sure the player is exposed to these things in the correct order to create a good story. This means that the story of a game is much more than the story of a book or movie. It has to include what sequences are possible, how you control the sequences and basically what it looks like if the player chooses to turn around and look in a different direction.

9.2 Our story

While story writing and stories in games and otherwise are interesting topics, with many things to discuss, these things will not receive that heavy a focus in this document. Whilst the story is an integral part of the game, the focus of this project is on the technical aspects of it rather than the creative ones (see references for relevant sources for story development). Listed is what we wish to accomplish with our story:

Motivation Game stories have a motivating effect and give a sense of progression. They also make a game more immersive. We also hope to create a setting with potential for tangential learning.

Context The story of the game will be set inside a program. The player's eventual goal is to 'escape the program by gradually mastering programming, thus themselves dictating what happens.

The player will travel through different areas of programming; simple mechanics, I/O, simple algorithms and so on.

Drive The drive of this particular story will be that the avatar of the player wishes to discover the truth, and eventually to find a way to escape to reality. We hope that the player empowerment from learning to control the game world will make the player keep playing.

9.3 The actual story in short terms

Tutorial While the game mechanics are still being learned, we will attempt to create our universe by giving hints to the player which make sense both to the player as external observer and the character's interactions with the world up to that point.

The Quest The player character discovers that it is trapped inside a program. It sets out to find a way to escape. It needs to acquire more power and knowledge.

The Truth The player learns that even if it were to escape the program, it would still be trapped in a computer, and thus all communication with the world outside would still be interpreted by input-devices, and not 'real'. It realizes that as a program it has no free will, and that the only escape would be to die. However, it cannot die (because it is not programmed to) unless the entire program dies, thus starts the final mission where the player has to...

The End System.exit()

Chapter 10

Gameplay

The game concept

We want a game that is capable of teaching programming to university first year students, and that is fun enough for them to want to play it. The biggest challenge will be making the game fun. We hope to give the player a sense of empowerment as they gradually understand more and more about programming. We will also be using classic game elements such as enemies and power-ups in an attempt to make it feel more like a game. In order to give a stronger feeling of progression and improvement, we will have levels with puzzles of increasing difficulty. There will also be a story as an attempt to give the game more drive and sense of progression.

To make our game more like a game and less like another programming task that the student has to complete, we need some means of delivering fun and suspense and a game-like feeling to the player. Ideally, this should tie in with the programming part of the game, so that it does not feel like two separate games. (Where one is a classic game and the other is programming) Rather, we would like the programming to make sense as a way of solving problems that already exist in the game world. But in order for it to feel more like a game, we wanted to add more game elements. Following are our suggestions.

10.1 The programming

Since we want our game to feature programming, we need some way to support this. We could in theory use any programming language in the world, but the arguments vastly favor using Python.

Python

These are the most important arguments for using Python:

- Python has a very simple syntax and is very easy to learn. It is also the chosen language for students to use in 'IT-Grunnkurs'.
- Python is interpreted. This means that we don't have to compile it, that we can run and debug code on the fly, and that we can run everything inside a single platform-independent application.

10.2 Gameplay

The gameplay of Pythia will be divided into two parts. One is a top-down view of a character walking around, touching and interacting with the other objects on the screen. The other is writing and running Python code, that affects the other objects on the level. The exact nature of this second part will be decided by the level creator, and it could potentially take on very different forms from what we have described here.

Making it fun

An essential part of the game is making it enjoyable. Two big potential factors would be fun of programming, and the concept of watching your program execute in front of you on objects in the game. As your programming skills improve, you should feel a growing sense of control and empowerment. In addition we are adding more classic 'game elements' to make it feel more fun and game-like.

Power-ups

While navigating around in the game world, the player will come across certain objects that he/she can pick up. These will affect certain stats or what the player can do, usually making him/her slightly more powerful.

Memory space /LoC Picking these up would give the player more space in which he/she can write programs. It could be shared by all the puzzles so that the player becomes more "powerful" and can write longer programs as he/she picks up more memory space, or it could be unique for each puzzle, meaning that the player would have to collect these for each puzzle he/she wishes to solve, and that each puzzle could have different restrictions on the length of the program.

Primitives and methods A possible game element would be to have each programming primitive (print, if, for, and so on) only be available to the player after he/she has picked up the corresponding item in the game. This would contribute to the player getting more powerful as the game progresses, and it could fill the role of keys or items with special abilities that we see in many games. Methods made by the puzzle makers could also be collected by the player in order to solve specific

puzzles, or parts of them. (a method could be the method to move the arm of a robot)

Bad stuff

Bad stuff is the name for things in the game that introduce an element of danger. This includes giving penalties and in some cases the death of the character.

1000 ft drop If you walk into this, you fall down and die. There is a bit of added suspense to walking near it. This could be further added to by some form of time pressure. You should walk around it, bridge it, jump over it, or balance on the edge of it.

Enemy An enemy is out to get you. They can do most of the things a player can do. Usually they move around, in some cases they will follow the player and damage him/her when they can. Such damage can be implemented in a number of ways. They might also make the player die, or be captured. However the enemies follow the same rules as the player, so some of them can fall and some can be killed.

Death

When a player dies, they could lose an amount of collectibles (for instance all the ones it has picked up until now related to this puzzle) and be transported back to the start location, and the puzzle will reset. This would be mostly to avoid exploiting death to cheat on a puzzle, as would be the case if you got into a situation where you could not survive just to get one thing that allows you to finish the puzzle once you have died.

Chapter 11

Modifiability

This chapter will give a rough description of the Level Content Interface, which is the interface a user will use when creating levels for Pythia. There will be a discussion of options for storage mediums. Following this there will be an attempt at finding a file structure for levels.

11.1 Example Content

The content of our game will be Levels or Puzzles where a player needs to perform one or several programming tasks in order to progress. The reason for this can be artificial (because someone tells the player to do so) or it can be implicit (the player needs to sort the pieces to get through the maze). In the latter case, it might be proper to give the player some hints about what they should be doing. We wish to use implicit puzzles wherever possible, as this ties in well with our illusion of being inside a program.

It should be possible to edit and create the game content (the levels and puzzles) dynamically. To do this we need a template for how puzzles should be represented, and some basic building blocks. While it should indeed be possible for a teacher to create a puzzle for his own class, a puzzle is a complicated thing, and describing it accurately could prove difficult, no matter how well made our part in it will be. A puzzle will most likely be made out of several different files to describe different things, so an archive format would be desirable. In order to find out exactly what we needed to describe a puzzle, we started making up a few of them to see what they would require from the game.

The Hello World! puzzle

This puzzle requires the player to write the 'Hello World!' program. It is a classic way to start learning any programming language. What the player needs to do is simply to make python output "Hello World!" to standard output. In other words: they need create a program that looks like 'print "Hello World!"'.

What a level needs From the hello world puzzle we learn that the game needs to support the following:

- viewing a picture on screen
- capability for running python code
- testing if a solution is correct
- viewing the print outs

The sorting puzzle

The sorting puzzle (slightly more advanced than the Hello World puzzle) requires the player to sort a list of numbers. We wish to make this puzzle one of the implicit puzzles. There will be a set of maze pieces you have to sort to open the way through the maze.

What a level needs From the sorting puzzle we learn that the game needs to support the following:

- native blocks (wall, floor, character, bottomless pit, label, console, goal, area transition)
- composite blocks made out of other blocks that might also change based on some variable.
- initialization code to be run before every attempt at solving the puzzle, in order to reset the list to be sorted.
- level layout to decide where the various blocks should be put

11.2 Suggested file structure

We suggest one could represent puzzles (levels) by dividing it into a few different files:

Python Scripts

A python script where we would have predefined hooks for methods to be implemented. For instance, a method with signature def init-puzzle(): would be run when a puzzle should be initialized. We should have such hooks for initialization, a test to see if the puzzle is really solved, getting the next case (if you want several test cases), and possibly user defined ones.

Level layout

This will describe where walls and floor and pitfalls are located. If a puzzle contains objects, this file should describe where they are, what they look like and which of their properties are bound to any python variables. You should have access to simple pre-made native objects and self made composite objects.

Main Level Definition

This will be the file that defines where all the other files are. It will include any interactive objects on the level and tests for any explicit tasks a player has to finish in order to progress to the next level. Any values that can be manipulated through python code will be defined in this file. Such things might include assigning a python variable to represent an objects color or x/y position on the level. It could also mean values that determine the internal layout of composite objects.

Composite objects

Any composite objects could have their own file with a similar or identical template as the level layout file. They would then be referenced in the main level definition.

Player input file

A piece of python code to describe the framework for the player. It could include code that cannot be changed (like a method signature that you need to stay the same) or code that is a useful suggestion, but can be ignored, and places where the player is allowed to input their code. This could be tagged with a console tag if the player needs to be at different consoles in order to input code at certain places. (For complicated puzzles in several parts)

11.3 Editor

Together these files should be enough for the game to create a playable puzzle. Which would all be editable as XML and clear text. An editor might be desirable, however, as these files might grow in size as a level becomes more complex. XML is a data storage medium and is not intended to be used as a programming interface. We have not scheduled any time to create such an editor. But it would be an invaluable tool in making the game more available for teachers to create their own puzzles.

Editor requirements

The purpose of a level editor would be to speed up and simplify the creation of levels. It would need the following features:

- A code editor, to write custom behaviors.

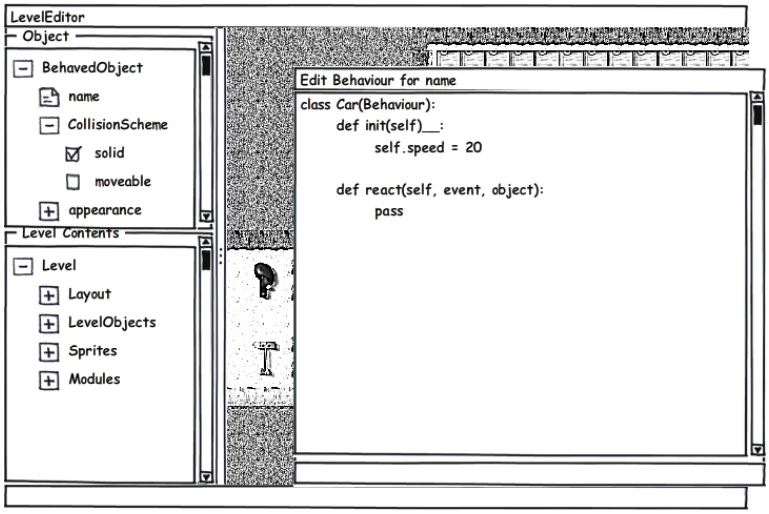

Figure 11.1: Mock-up of the Level Editor.

- A library of pre-made behaviors.
- A tool to draw walls, floor and so on.
- A tool to create and edit objects in the level.
- An overview for all the things contained in a level. Such as sprites, level objects, scripts, conversations, and so on.

We imagine a level editor could look something like the mock-up in figure 11.1.

Chapter 12

GUI elements

12.1 The in-game console

The in-game console (Figure 12.1) is a text interface that can be brought up at any time that will interpret any python commands entered into it. These commands will affect the game environment in the same way as any other python code in the game. Using the console adds a slightly hacky feel to the game, as well as being great for trying out short code snippets and for debugging/designing, or just playing around. We also chose to have any output from python be printed into the console.

12.2 HUD

The top bar

Shows what keyboard Function keys do what at a given time. For instance you can press F1 to open the console and F2 to open the Editor.

The bottom bar

Gives a context sensitive 'hint'. (press enter to talk to monkey) Also shows notifications when something happens.

The Menu

Gives the player options to quit the game, restart level, start new game, and so on.

12.3 Code editor

The code editor is where the players write and execute their programs. A screenshot of the editor can be seen in figure 12.2. The code editor is the most complex interface component in the game and is built like a custom TWL-widget.

Figure 12.1: The in-game Console.

The code editor has the following features:

- Scrolling, vertical and horizontal
- Syntax highlighting (keywords, comments, strings, numbers)
- Line number display
- Cursor line highlighting
- Setting and displaying Breakpoints
- Step-line highlighting
- Selecting, coping, pasting and cutting

Design

The code editor is a mixture of several TWL-widgets: custom made text editor, custom made line number display, extended box-layout and extended scroll pane to hold it all together.

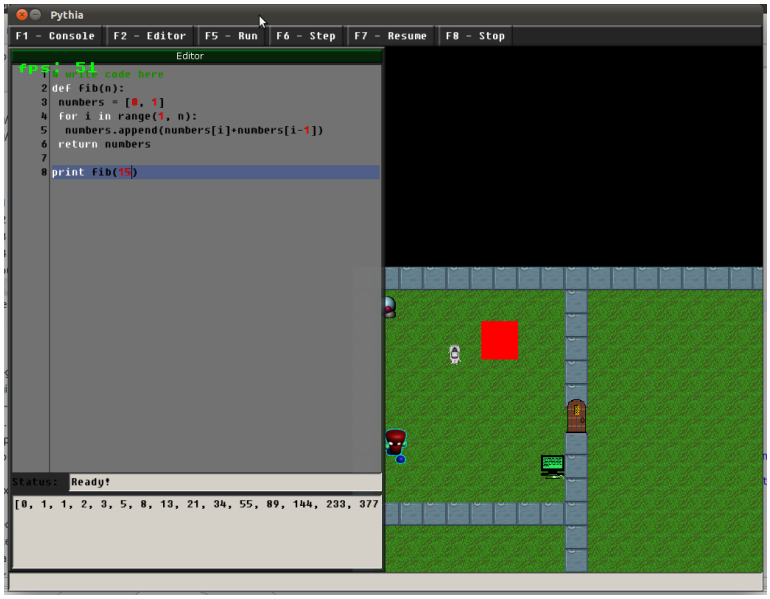

Figure 12.2: The in-game Python code Editor.

The text editor is a rewrite of the built-in text field, which is a class in the TWL library. The text editor includes several text smaller widgets inside the component. These widgets have separate theme options which define the font and font color of each text widget. We use pattern matching to parse the text in the editor and direct matching parts of text (like keywords and comments) to the corresponding text widget. This is how we achieve syntax highlighting. In addition, two special components are added that do highlighting of the cursor line and lines with breakpoints.

The line number display is another custom TWL text field. This one is attached to the side of the text editor, counting lines and change their values when the player uses the scroll-bar on the text editor. The line number display also holds information about breakpoints. The user can toggle breakpoints with both mouse and keyboard.

Code Execution

When executing player code, the code editor will create a separate python thread. This is to separate the player code from the rest of the system, so things will continue functioning even if the players program gets stuck. The thread is connected to a tracer which can catch exceptions and report them to the player. It also will stop on any break points and allow the to step through the code line by line. The player

can also stop the execution entirely (if it gets stuck or similar) by pressing the key F8.

Part IV

Implementation

Chapter 13

Introduction

This part will describe the implementation of the game. It will outline some of the choices and challenges we were faced with once we started trying to make a game. It also contains an account of some of the more important things we did. Some of our major achievements discussed in this part includes how we integrated our game with Python, how we created a debugger and topics related to how we handle user content.

Chapter 14

Architecture

This chapter briefly describes the architecture of the entire system. It is roughly divided into 4 parts:

- Graphics and Game loop
- Python Connection
- GUI and Player code execution
- Game Puzzles

14.1 Graphics and Game loop

This part is where the main loop of the game will be located. The responsibilities of this part is to show graphics and to send user input to the correct locations where it will be handled further. It is the spine of our game and should not get directly into touch with Python or any user generated code.

During a cycle of the game loop it will do the following:

- Test for collisions and send messages to the correct objects.
- Rearrange objects in the rendering list to render in a right order.
- Draw the objects on the screen.
- Pool and process the keyboard and mouse events.

14.2 Python Connection

This will be the part where we add an additional layer to the Python interpreter in order to send messages to the correct objects. This part will be responsible for handling code written by the player and sending messages to content created by a user. The user code is ran in a separate thread, to avoid freezes, hang ups and crashes.

14.2.1 Bindings

The Python bindings will be a listener interface where a user can add a listener to the Python interpreter to trigger when a chosen variable is altered or set. This will trigger a reaction of the users choice.

14.3 GUI

This is a set of tools used by the player to either get information about the game environment or to write and execute a python code. The GUI is also used to monitor execution of the user's python code.

14.4 Game Puzzles

This is the part responsible for representing the user content in our system. It will execute Python code on behalf of the level creator, fetch character conversation files, and hold the data models which decide the appearance of the level.

14.4.1 Level Content Interface

The level will be loaded from a file structure resembling the one described in chapter 11. This will be handled by a series of factories. The bulk of which are the Simple Object factories, which load the objects that the player can interact with in a level. Most of the level will be defined in a single XML-file, which is what we have come to refer to as the Level Content Interface.

Chapter 15

Game media

Graphics and sounds are an essential part of most games. Good aesthetics makes for a more pleasant experience than bad aesthetics. While many video games go for realistic graphics, and some other go for more stylized graphics, we have a limited time frame and will go for the simplest kind of graphics. This is also affected by the fact that graphics are not considered to be that important in our game. However, we do not want our game to look outright bad.

15.1 Loading Sprites

OpenGL bindings are used to store textures. The texture resides in the memory until the Java's garbage collector removes the object. During this process the texture is removed from the memory by destroying the OpenGL texture. The picture format supported are limited by the LWJGL framework. LWJGL supports wide range of picture formats including most commonly used: *bmp, jpg, gif, png.*

15.2 Animation

Animations are a big improvement over still images. It can make things seem more reactive and less rigid. The animation object has a list of references to the frames of the animation which are single instances of sprites. A level designer specifies animation order and length during level design stages, but these parameter can be tweaked in runtime by using Java methods from Python modules.

15.3 Shadows

Shadows are an easy way to make things look more realistic and alive. The shadows are created on the fly by several simple manipulations of the textures of a object and painting them under the main texture. Each game object has a Z-axis value which defines which objects should be painted over others.

15.4 Sound

Pythia support loading and playback of *WAV*-files. The sound objects are removed from the memory in the same manner as the sprite objects, when JVM loses the references to the sound object. OpenAL engine bindings included in LWJGL library are used for the playback of the sound.

15.5 Music

The music class was created to support MIDI playback, however no MIDI files were produced during development so the implementation was never used.

Chapter 16

Integrating Python into our game

We will be using the Java implementation of Python: Jython in order to run python code in a single, cross-platform Java Virtual Machine (JVM).

16.1 Motivation

In order for our game to run Python code automatically we will need to use some implementation of python that we can access easily and interact with closely. It is desirable that we can both change parts of the interpreter as well as interact with our game from within python, and interact with the python code from our game. We also desire that the game should be multi-platform.

16.2 Alternatives

Jython is an open source Python interpreter implemented in Java. It can do anything the other Python implementations can do and also interact with Java code.

C Python is an open source Python interpreter implemented in C. It is very common and is included in most big linux distributions, and has compiled interpreters for most major operating systems.

Our choice We decided upon Jython in order to run Python code. This is mostly because we wish to program in Java because of our experience and because of its portability. The Jython interpreter can easily be embedded in our application.

By using Jython in our game we will have an interpreter ready to run and we will not have to implement our own interpreter. This should make it easy to integrate Python into our game.

Chapter 17

Integrating our game into Python

Using Jython, we found that we could make a set of wrapper classes to monitor when variables are set, and make listeners to forward the changes in the code to the things that are happening on screen.

17.1 Motivation

In order to make the Python code interactive and observable we need some kind of system that tells us when things change internally in the python code. Several methods are available, but we feel that a listener interface would be the best way to go, so you could pick variables to listen to, and then have your listeners fire whenever these variables change.

17.2 Implementation

What we will do is to make a series of wrapper classes. We will make our own interpreter to wrap around the Jython interpreter. This is in order to make sure we get our own variable map wrapped around the variable map. We will also need to make wrappers around lists, dictionaries, and objects if these are to be listened to, as they don't change their binding every time something changes internally in them. (Lists and dictionaries are especially important, objects could be tailored in Python specifically for each one, unless students are going to make objects from scratch)

Chapter 18

Debugging Functionality

It will be difficult for a player to follow the changes on the screen if they follow the normal execution time of a program. This is why we need a system to control the execution and have functionality for slowing down and halting execution, and stepping through it. We chose to make our own embedded debugger to control the execution of Python code.

18.1 Introduction

In addition to knowing when things change internally in the python code, we would also like to be able to control the execution of it, both by adding a manual and automatic stop feature. This will allow the player to follow the execution of the program, which would not happen at normal speed, which in turn allows for ad-hoc debugging. Seeing the execution on a line-by-line basis will also improve the understanding of the player, without making it an obvious part of the learning aspect of the game. This is very close to the functionality of a debugger.

Between Java, Python and Jython there are a lot of ways to achieve debugger functionality:

18.2 PDB - Standard Python Debugger

PDB is a debugger for Python, written in the Python language. It is part of the standard library for Python, and it has a command-line interface. It is hard to control automatically. One way of doing that would be to make a controller that feeds its standard input directly and start a pipe from one to the other via a platform-dependent shell command. This would also cause the program itself to run on a different JVM, which means that accessing the variables etc, from the original JVM would be require much more work. In order to fully understand PDB (in order to alter or extend it), one might have to resort to debugging the debugger (with itself or with another Python-debugger) which could be a very complex task.

This might be a better choice in a pure Python program, or in a Python and C program.

18.3 PyDev-embedded Debugger

The PyDev Eclipse plugin for Python development has a debugger which is written in Java. It uses the standard Java debug server interface, that lets you plug in any user interface on top of it, though it is intended to use the debugger interface in Eclipse. Out of the box, this approach also suffers from executing on a different JVM. However, if we could adopt their (open source) code to make our own debugger that doesn't use the slightly over-complicated server interface, we would have all the things we need. This would also require that we fully understand how it works, but it might be simpler to debug a python debugger with a java debugger, which is what we would be doing in this case. It would still give very little benefit over making our own debugger from scratch.

18.4 Making our own embedded debugger

If we can find some hooks somewhere in the Jython machinery to make our own debugger, (we might look for inspiration in the other debuggers) we could end up creating our own simple debugger with stepping and execution control. This way we would guarantee that it did whatever we wanted, and nothing that we don't need it to do. And it would run on the same JVM as the rest of the project, meaning it would be easy to share variables etc. It would be strange if there wasn't some functionality to make debuggers in Jython, but Jython is open source, and open source is often poorly documented.

18.5 The Choice

In the end we managed to find a way to make our own embedded debugger.

Chapter 19

The tab character

In the Python programming language, the whitespace and the 'tab' character plays an important role. Once a method, class or loop is defined by its signature line (ending with ':'), all following lines starting with a tab will be part of that method, class or loop. This is why it is important that our editor and console can handle the tab character as a part of the text to be read, and that we can render it as something meaningful as well.

However, according to the Python guidelines, programmers are encouraged to use space characters and not tab characters [13]. And the programmers should never use mixture of tabs and spaces. Later in development stages the decision was made to replace all tabs with spaces and insert space if the player wants to insert a tab.

19.1 Reading the tab character

Many text fields and similar will interpret the keyboard action from pressing tab as a signal to cycle the focus through the components on screen. This is why we need to explicitly override what pressing the tab key means. We do, however, need our tabs to be recognized as tabs (or at least as indentation) in the python interpreter.

19.2 Rendering the tab character

A tab is a complex thing to render. The usual way of doing it is to have certain points on the page, usually 8 spaces apart, and whitespace will be created up to the next such point when you press tab. This makes the tabs align down the page, which is useful for making tables and such in text editors. We have no use for that functionality in our editors. What we need is just a bit of space to indicate an indentation in the code. In initial versions we used 4 spaces to render a 'tab'. In later stages of development we decided to intercept all 'tab' key presses and insert spaces instead. However the old tab display can still be seen in the Pythia console.

Chapter 20

User Content Interface

This chapter will discuss some of the in-game objects that can be created and altered by a user (we imagine a teacher). These objects are defined in XML files and Python scripts that are read by the game when it loads a level. The XML-files and the Python files that are created by the teacher, will work with the game through the User Content Interface. A closer description of exactly how to work with this interface will be in appendix B.

20.1 Level XML Files

The Objects in a level, which should be defined in the XML documents, need to extend certain base objects in our system, that deal with rendering and colliding. Since these base objects do not have empty constructors, it was not feasible to have empty constructors on our more high-level objects either. But due to the inner workings of JAXB, the values that we needed to put in the constructors were not ready by the time the objects were constructed. To rememdy this, we needed a factory object.

20.1.1 Factories

The factories themselves can have empty constructors, and as such are suitable for defining what values should be contained in the XML documents. The most important reason why we can put these definitions here, however, is because we will never write to the XML documents, only read. Should writing be required (for instance in a level editor), a slightly different strategy is required.

20.2 Manipulateable Objects

Manipulateable Objects or Dynamic Objects are the core objects of any level. They will be rendered and have collision models as specified in the XML. They can have

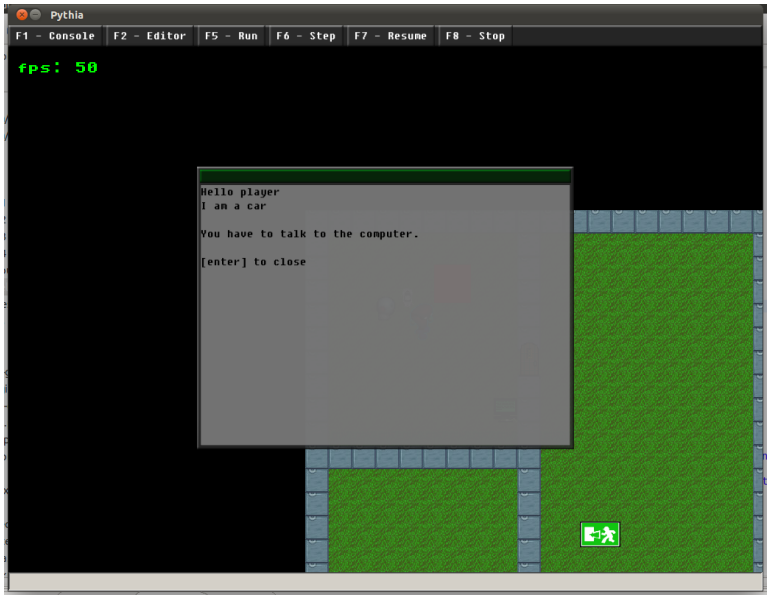

Figure 20.1: Screenshot showing a text window with custom content.

conversations related to them, which show windows with text, when a player talks to them. And they have properties which can be changed in-game.

20.2.1 Python Binding

A python binding is when we set a listener to any variable in python, and link it to a property of a Manipulateable Object, and cause the property to change when the variable does. Such a property can be, but is not limited to: position on the level, color, what it will say when a player talks to it, what collision model it uses and what appearance it has.

This is useful for showing on-screen what happens when a program executes, to make players aware of what their programming mistakes are and hopefully give a better insight into how to fix them.

20.2.2 Conversations

When a player talks to an object, what usually happens is that a window with text in it pops up. This is the most common way to deliver objectives, hints and story segments to the players. A text window looks like the one in figure 20.1.

```
from org.ntnu.pythia.game.behaviour import Behaviour
from org.ntnu.pythia.game.behaviour import BehaviourEvent
from org.ntnu.pythia.game.behaviour import BehavedObject
"""
A Behaviour for a generic key, which uses the property
on the belonging object and on a door object to determine
if it fits or not.
"""
class Key(Behaviour):
    def react(self, event, object):
        if event.getMessage() == "talk":
            keyname = object.getProperty("key")
            setSysVar('keys.' + keyname, "true")
            object.remove()
            statusBar("Received key: " + keyname)
        return True

    def timeElapsed(self, delta, object):
        """This method runs every delta seconds. Object
        might want to do something during this time"""
```

Figure 20.2: Code example for a generic key.

20.3 Behaved Objects

Behaved Objects are objects that have a behavior related to them. Behaviors are Python scripts that define what these objects should do in given circumstances. The name Behavior suggests this would be related to the AI of enemies on the level or similar things, but that is merely one of many options. In figure 20.2 we have written behavior for a key. This code is general and can be used for any key that defines a 'key' property in its XML declaration. This property will be checked against a door to see if you can unlock it.

Behaved Objects also have a *timeElapsed(self, delta, object)* method which is executed when some time passes. This scheme allows for complex behaviors that are depended on time, and need to execute certain actions with delays.

20.4 Levels

While you can do anything you want with these tools, it does not guarantee good content for a game for learning. This is why we need to think a lot about the actual puzzles in the game. While some complexity is good, we feel that is might be better to keep it simple, and to focus on just one aspect at the time. The puzzle in figure 20.3 is a very simple puzzle where the objective is to move the exit to the side of the wall where the player is. This will be done by accessing the computer terminal and changing a variable (The exit is a Manipulateable Object)

Figure 20.3: A simple level in the game.

Chapter 21

Player solution testing

Many of the puzzles will be of such a form that the tasks set before the players are clear and cannot be avoided. These are implicit tasks. These include opening doors, moving things out of the way, avoiding an enemy and so on. These are good goals, because while the user has to be aware of them in order to complete them, we do not have to check if they have in fact completed them when they are at the level exit zone. On the other hand, there are explicit goals, such as: 'the player needs to gather all the orbs' or 'the player has to guess the password correctly'. While there are clever ways to transform most such explicit goals into explicit ones, we want to have a simple system that does not need to be reinvented for every level. Extensive explicit tests might also form a sort of safety net and be an important measure to avoid cheating. (See also the chapter 'Security' on page in chapter 22 on 54)

21.1 Possible Solutions

21.1.1 Unit tests

There is a built in system for unit tests in python, much like the JUnit test system in Java. This has all the functionality needed, except for a controller to actually run the test when appropriate. The unit tests are a bit hard to control and run individually, since the number of tests and the method signatures are completely arbitrary. The common way of running unit tests is to run them against certain methods in the code that you want to test, where what we would really want to do, is to test the internal state of our game. Testing the user code might in some cases also be of interest, however.

21.1.2 Home-made Testing system

We could make our own testing system, consisting mostly of XML defined tests, and some Python where the XML is not viable. This way we could also categorize the tests ourselves, and set individual hints easily. An additional strength of this

```
class MT(Test):

    def isSolved(self, level):
        try:
            guesser.reset()
            main()
            return guesser.isGuessed()
        except:
            return False
```

Figure 21.1: Code example for a test on a method called 'main()'.

approach is that we could use it to give more objectives to the player while the game is running.

21.2 The Choice

We chose to make our own testing system to have full control over every test, and automatically generate feedback to the player about which goals they have not yet fulfilled. The 'level.xml' file will have an entry for each test required to pass the level. There are a few different entries for simple values you can test against. There is also an entry for creating a custom test written in Python, which should give you a means of testing anything you can think of. Currently there is no way to run the user code in a test environment, but such tests could be implemented without re-compiling the game if you can make a player implement a method with a desired signature. For instance you could tell the player that it should create a method called 'sort()' and that this method should sort something. Figure 21.1 shows code for a test on a method called main().

Chapter 22

Code Security

If our game does not have proper security, situations might arise where the player can cheat on or bypass sections of the game. They might also break it, making it unplayable.

Since this project relies on taking programs as input from the user, and we want to make it as free as possible, it has some serious security issues. The player could among other things find ways to cheat or break the game in such a way that it cannot be played. Standard methods such as input filtering and encryption would be very complex to implement, and besides, we want the player to be in control. We believe that hacking a game which intends to teach programming and computer science does not inherently count as cheating, as long as the player themselves is doing the hacking, not just copying from someone else's hack. A successful hack of the game should be a valid method of learning in this instance.

22.1 Code input

When we run user-written code, we do not (currently) filter anything. If it crashes, the player is notified. If it runs, it behaves as you would expect it in any other python implementation. The player also has access to all the java files and the entire java standard library, as well as the python standard library. This causes some issues, since a player could theoretically do system-calls that would render the game unplayable for them. For instance a player could run the following Jython-code in figure 22.1.

```
import java.lang.System
java.lang.System.exit()
```

Figure 22.1: Jython-code that exits the program entirely.

22.1.1 Using a separate Java Class Loader

In order to restrict access to certain classes in our project, we could load Jython in a separate class loader from the 'sensitive' parts of our project, and then use standard java security mechanisms. This way we could make it impossible to access certain parts of the Java code from inside Python. This wouldn't up our workload very much in itself, but it might add another layer of complexity to any debugging we might have to do. The complexity of debugging is already quite high, considering the fact that we already use two programming languages and the accompanying, rather complex, runtime environment.

22.2 Content Files

A closely related security issue is the fact that all the files describing the game content (levels, conversations, scripted behavior) can be freely viewed *and changed* by anyone. This enables cheating. In order to stop this we would have to somehow encrypt parts (or all) of the distributed game. This could tie in well with creating a custom class loader that reads, decrypts and loads all at once, while the Python elements are run with a different class loader.

22.3 Security vs No Security

22.3.1 Embracing Poor Security

Another option is to let the player do whatever they choose. This might not seem like a proper solution at first, but we have a few good arguments for it.

In the case of the System.exit() 'exploit' (fig. 22.1), we plan to use this in the game on the final level. The story will lead the player to the gradual understanding that its avatar is trapped inside a computer program. This also happens to be the actual truth of it; the avatar is literally trapped inside the game. As are most avatars in game, but this one in particular is aware of it. The only suggested way to escape is to terminate the entire program. And we intend to do this by having the user do the System.exit() 'exploit'. This would close the entire program. This is more closely discussed in the 'Story' chapter on page 24.

In the case of other insecurities, there might be similar atmosphere- and story-related issues. We also feel that we shouldn't discourage curiosity and creativity in the player.

Game Content Files While we could simply release these files inside a password protected archive that the game reads them from, we feel that the level of understanding required to exploit this 'weakness' is far above the one we seek to teach them. If they can find a way to hack our game content in this way, their mastery of the game should be rewarded, not condemned. Also, if someone else is going to make content for our game someday, they will benefit from rich examples.

The downside of this approach is that there might be a case where the player does something that we didn't think of *and* potentially makes the game somehow unplayable. This will be very annoying for the player, but if they can figure out what the problem was, they will most likely have learned tremendously in the process.

22.3.2 Problems

Giving someone full access to the game file structure means that it will be very easy for them to break the game and make it unplayable. If anyone wanted to use custom content for this game as a more formal learning test, being open to cheating is a massive disincentive.

22.4 Conclusion

Given that we assume that the players want to learn and experiment, we do not feel that we need to add security features that would actively prohibit this. Making the game more volatile will also add some to the player's feeling of control. Lastly, in the interest of time, we do not wish to make any additional security mechanisms. The two suggestions that were made could easily be added on in the future. Though there might be additional security issues that we have so far overlooked.

Chapter 23

Accessibility

Since we want people to play our game, we desire to make it as accessible as possible. This includes supporting several different platforms and making it easy to start playing.

23.1 Making a runnable jar

The runnable jar is used to simplify distribution and execution of the game. Simply put, we have the entire, playable game in a single executable file. To quickly and effectively create such a jar, maven the Java project build manager is used. Java Runtime Environment is required, but it is available for most systems.

23.2 Maven

Maven is the Java project manager by Apache. Maven can be run on anything that can run Java. Maven supports a broad range of plugins and scripts. Maven supports wide feature set, executed by Java virtual machine but has one of the worst documentations. It may be tricky to make a perfect maven setup, but once done, the project building becomes quick and painless, automatic process. Maven supports automatic dependency fetching (downloading libraries) and jar creation.

23.3 Native libraries

The native libraries deliver a hit on the portability of the game. The only platforms that can run the game are the ones for which a native library are provided. In the case of Pythia, which uses the OpenGL native libraries, it is Windows, Linux, Solaris and MacOS (x86 32 and 64 bit architectures). The native libraries are packed inside the runnable jar. They are unpacked into a temporary folder during the execution of the game.

23.4 Jython

Jython is the Java implementation of Python, and does not require Python installed on the local machine to run. Jython also offers bindings between Python and Java code. Jython is the perfect library to use for portability options.

23.5 Python standard library

The earlier maven build of Jython library did not provide the Python standard library so a workaround was implemented that extracted the python standard libraries from a separate archive. However such workaround was deemed as unnecessary after Jython received an update which included the Python standard library inside the Jython library.

Chapter 24

Performance

The performance of the game is how well it handles poor hardware, how quickly it responds to input, and how fast it can do a multitude of things, such as loading times, framerate and executing code.

24.1 Motivation

Performance is important because a slow game can be annoying and boring to play. We also want our game to be playable on as many (and cheap) platforms as possible. While it does require a keyboard to play, (not recommended for phones) it should be playable by anyone who takes the ITGK (see page 15) module at NTNU. The ultimate, but unrealistic goal, is that the game should be able to run on the thin clients (which have no or little graphics memory) that NTNU have available to use by all students. A more realistic goal is to have it runnable on a 5 year old laptop.

24.2 Hardware

Table 24.2 shows the computers used for performance testing, should anyone want to replicate the results.

24.3 Performance measuring

24.3.1 VisualVM

Oracle's VisualVM is a tool that provides a visual interface for viewing detailed information about Java applications while they are running on a Java Virtual Machine.

Name	OS	RAM	CPU	GPU
Amilo pi 1505	Ubuntu	1GB	1.6GHz Dual-core	GMA950
Thin client	Windows			
Acer Aspire Timeline TZ3810	Windows 7, Linux 2.6	4GB	Intel SU4100 1.2GHz Dual-core	4500MHD
Thinkpad T510	Windows 7, Linux 2.6	4GB	Intel i7 2.6GHz Quad-core	NVIDIA NVS3100M
Mako	Windows 7 64bit	4GB	AMD Athlon 64 3.1Ghz Dual-core	ATI HD3800

Table 24.1: Systems used for testing performance

24.3.2 Bottleneck-Searching

Oracle's VisualVM was used as a profiler for Pythia. The profiler was connected to the running game. The game was played while the profile were sampling information about memory allocations such as object creations and method execution times. At the end of the run the most frequently used entries of both were taken for the analysis.

24.3.3 Implementation

The usual approach to eliminating the bottle necks for the execution times were either eliminate part of the calculations if possible. If not – try to optimize the algorithm in question. And finally reduce sub methods call count.

As for the memory, the most frequently generated type of object was traced back to the point of creation. Afterwards the creation of the object was eliminated either by using a single pre-allocated instance of the object to hold a value for calculation or by removing the object completely and try to replace functionality by use of primitive data types.

Collision The collision was one of the early performance bottlenecks. It turned out that the extreme number of temporary geometric objects were allocated during the collision tests. Allocating memory for new objects is often one of the slowest simple operations in Java. This was replaced by constant instances of the objects in which only the object's internal values were changed on the collision tests. This means that more memory is allocated on average, but the collision tests run much faster.

Level rendering Another performance hit was coming from the level rendering method. This was fixed by the limiting the number of object we would like to process depending on view and how are they passed to the rendering engine. The result was that we render only the things that are visible on the screen, which saves some work.

24.4 Thin clients

The university's computer labs are full of thin clients. Unfortunately these clients do not support hardware acceleration. Nor do they include libraries that emulate the hardware acceleration in the software. This could be fixed by providing the Mesa3D libraries to Pythia. However it is unknown if such implementations would deliver the desirable performance.

24.5 Canvas

OpenGL was used to relive CPU and improve rendering quality. However a Java Canvas solution was considered in the beginning and the parallel solution for such implementation was in development. Unfortunately a lot of time was lost due to constant porting of rapidly changing engine and the decision was made to cease the development of the Canvas part. In later stages the software porting became more complex since a some dependencies like Themable Widget Library were dependent on OpenGL.

Chapter 25

User Feedback Form

Purpose

The purpose of the user testing is to determine how well (or if) we have succeeded at various goals we set for ourselves in chapter 5.

25.1 Contents of the form

We will be running a test for players of the game in order to collect feedback on the user experience. There should also be a small test to give a little bit of indication about how much the player learned. The first set of questions, we got from a standard test for usability, called the System Usability Scale (SUS). On the topics of fun and learning, we decided to make our own set of similar questions, that we feel are relevant to our understanding of fun and learning in this case. Because of this, the accuracy of the results from our user tests is not very good, but it should give at least some indication of how well we have succeeded. The questions that we decided ourselves, like the SUS, have a likert scale [7] and they have the same alternating positive/negative question form as the SUS. The actual questions can be seen in appendix E

System Usability Scale

The System Usability Scale[10] is a question form with 10 questions. This collects feedback in the form of a simple number on the application float. It is commonly used in situations like this.

Fun

The part about fun is there in order to determine if the game kept the player entertained, and if we succeeded in our various attempts at making the game

entertaining. Some questions are also related to if we managed to do what we hoped fun would do.

Learning potential

The part about learning has questions about how much the player felt they learned, as well as questions related to our methods of teaching. This is not the best way to test how much was actually learned, and offers just a minimum of accuracy about the learning value. In addition to collecting the data from these 10 questions, we will be gathering information about how well they know the topic of programming from before, to see if any level of proficiency benefits more from playing our game.

25.2 Method

Since we felt it would be easier to reach a group of testers if we didn't have to arrange a big group meeting with a time limit and space requirements, we decided to make a servlet, where testers can download the game, and fill in question forms in their web browser, and in the evening or when they have time.

Part V

Results and Discussion

Chapter 26

Results from user tests

26.1 Specific feedback

Difficulty

Some testers reported that the puzzles were very difficult and that they wanted more hints. And that some of the in-game texts could use some refinement. The opinion seems to be that the game would work well as a supplement to teaching programming, but that it in its current state does not do that well on its own.

Some testers who were new to python encountered some problems and made the interpreter crash.

Testers who were already proficient in programming would often comment that they saw the potential that this could be used for learning, and that they thought it was a good idea.

Portability

If the users screen is too small, the window will not fit as it is running a fixed pixel size.

Bugs

Many users reported bugs. Some were fixed, but some we did not find the reason for. Many bugs would unfortunately include the game crashing. We were unable to find the reason because we did not have a proper system for gathering stack-traces and generating crash logs.

26.2 Data from Questions

The user testing gave us the results presented in table 26.2.

Category	Average Score (1-100)
Usability	67.81
Learning	69.69
Fun	71.25

Table 26.1: Average score by category

26.3 Discussion

26.3.1 Error

These results are subject to many error sources.

Size of test pool The number of testers is limited. With a bigger number of testers the average test result would be more likely to home in on a correct value.

Questions While the questions for usability are common and have been subject to analysis on several occasions, the questions we made up have not. There is a chance that the questions are phrased in a way that somehow gives misleading results. However, if someone really did not have fun, we do not think that they would have given our game very high scores in the fun section, for instance. Considering this, we do not think that anyone would give a score which is more than 20 points above how they really feel.

26.3.2 Results

The results are calculated using the method described in 'SUS - A quick and dirty usability scale' [10] and are on a scale of 1 to 100. They have been averaged for all the testers. According to the article 'Determining What Individual SUS Scores Mean: Adding an Adjective Rating Scale' [8] our score for usability is barely within the requirements for being considered 'good'. We think that players who find the game to be difficult to solve will rate down the usability, because of the nature of the questions. This is because we can see that players who already know programming tend to rate the usability higher (Figure 26.1). We assume that this means that users with low knowledge struggle more with our system.

Our other measures have similar scores. If we include our safety zone of 20 points, we can see that our values for fun and learning are still at least 'OK'.

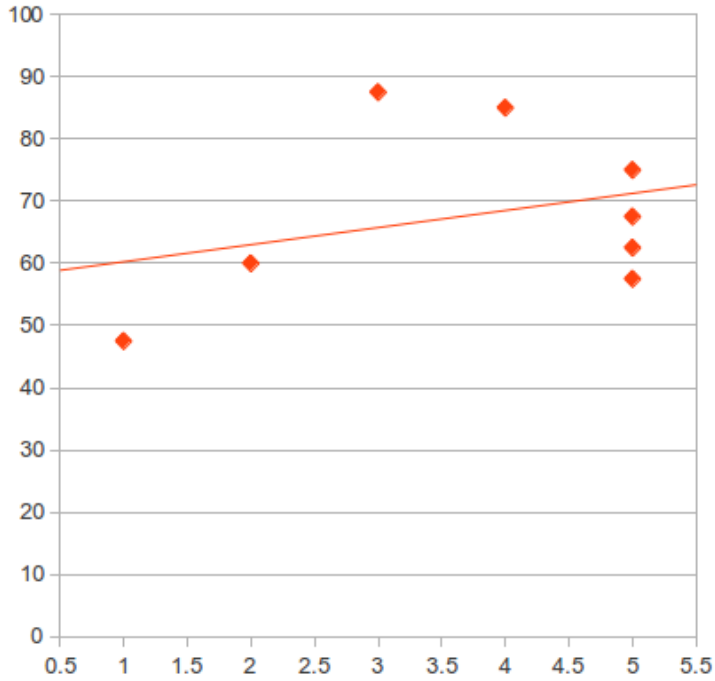

Figure 26.1: Given usability score mapped against stated programming experience.

Chapter 27

Puzzles

In order to evaluate our games modifiability, we have created a multitude of puzzles using our 'Level Content Interface'. In this chapter we will go though a few and reflect on how difficult they were to create.

27.1 Lists

Description

The level called Lists presents the player with a list of colors, visible on screen. These colors have to be arranged in the order of the rainbow in order for the player to proceed to the next level. As the player changes the list in the python code, the colors will change on screen.

Learning goal

This puzzle is supposed to teach the player to use list accessing and simple manipulation.

Evaluation

This puzzle required some Python to be written, but we would claim that there is little knowledge of the inner workings of the game needed to write this code. The most work as a puzzle creator, was in making this puzzle robust, handling many kinds of user inputs. This is something that many puzzles suffer from.

27.2 Use Method

Description

The player has to correctly use a method called *theMethod*. The method takes two arguments. The first is a number and the second is a string. The player needs to

call the method with these argument correctly in place before they can proceed to the next level.

Learning goal

The goal of this puzzle is to teach the player how to use methods.

Evaluation

This puzzle is very robust, and it gives the player hints when they do something wrong. The main issue with this puzzle was that it was hard to make it cheat-proof. It required very little knowledge of the inner workings of the game.

27.3 Key

Description

The level named Key is a simple demo of how one could make keys that the player can pick up, and doors that they fit into.

Learning goal

This level is not aimed at teaching the player anything, but rather to give an example of simple gameplay mechanics.

Evaluation

While the creation of this level did not require knowledge of how the game works, it does require knowledge of the User Content Interface. The behaviors for the door and key are universal, however, and do not need to be re-written for each key or door the user wants to make.

27.4 Move the Exit

Description

In this Puzzle, the Exit is located on the opposite side of a wall. The player has to change variables to move the exit to the same side of the wall as the player. Then use it.

Learning goal

This will teach the player to think creatively, and will also show them another example of what a variable can be.

Evaluation

This was very simple to implement, and requires no Python code at all. It is also cheat-proof, as cheating is actually the solution. It would be good to have more puzzles like this one, but it would be better if they were more difficult.

27.5 Binary Search

Description

The player has to implement binary search in order to guess a number between 1 and 100 in 7 tries or less. The guesses are shown as a red line on a white scale.

Learning goal

The puzzle will teach the binary search algorithm as well as being an introduction to the if statement.

Evaluation

This puzzle is one of the hard puzzles towards the end that allow you to use the editor. It requires the player to implement a method with the correct method name (other approaches are possible), as well as the use of a built-in guess() method. This makes it slightly stiff. The challenge on this puzzle was once again the Python aspect, and making the puzzle hard to break and resistant to crashes. There is nothing preventing the player from guessing the same number over and over until they get it right by pure chance, but we feel that this would be unnecessarily complex to implement.

27.6 Sorting

Description

The player needs to sort a list in order to pass through a maze. The exit lies on the other side of the maze.

Learning goal

The goal here is to teach sorting.

Evaluation

This puzzle has no explicit tests, so the player is free to solve it as they choose. If they want to, they can set all the values of the list to 0 and walk through the maze relatively unhindered. This means that the players are not forced to learn about

sorting, but will at the same time be free to play around with the list as they please. There is also a lower chance of something crashing.

Chapter 28

Discussion of Modifiability

Having made many puzzles for the game, this is our own evaluation of how modifiable the game is.

28.1 How fluid is the content

We have had some ideas which would require something to be changed in the non-dynamic code. These ideas are discussed in chapter 31. While the fact that these things cannot currently be implemented in the dynamic content, means that our system is at least a bit rigid. However, the flora of puzzles we have managed to create, shows that there are many possibilities within the current system as well. While the possibilities are near limitless because you can do whatever you want in Python, the amount of work and knowledge required will start to increase once you start doing something more difficult. The real limit is therefore when the user does not know how to do what they want to do. This could be improved on, but could probably never be perfect.

28.2 How much knowledge is required

The amount of knowledge required to make a full level is pretty high. In our case we found that we were more often stuck on how to implement something in python rather than how to use our system, which is only natural since we made only one of those, and have full knowledge of it. It could also be attributed to what we define as the responsibility of the python code and what is the responsibility of the system. It is worth mentioning that most of the complexity regarding puzzle creation comes from two things.

The first thing is that the complexity of the puzzle is often tied to testing the solution. The game should be so robust that the player does not break it by accident, and also smart enough to give error messages that the player can understand. This

requirement for robustness is mostly tied to the puzzles that use a lot of Python in their description.

The second thing is the presentation of the puzzle. While we have many means to show values and bits and pieces of a puzzle, the choice of presentation needs to be decided by the creator of the puzzle, so that the relevant pieces can be shown. This means that there is some book-keeping necessary in the puzzle definition. This is often Python-heavy.

28.3 Workload

Making a puzzle should ideally not be a lot of work, but it is. The level designer has to do all of the following:

- Make the objects and define the visual presentation of the puzzle.
- Make the tests to see if the player has solved it, make these robust.
- Give the player sufficient hints to solve the puzzle.

The final two of these are usually where the most time is spent. Both of these could be done quickly, and the result would be a puzzle that is relatively easy to break or cheat on, and is hard to understand how to solve properly. Doing the first quickly could result in a puzzle that has no visual aids, and could just as well have been made without using Pythia.

While the responsibility of lowering the workload is entirely on the content interface, most of the work itself needs to be done in order to make good puzzles.

28.3.1 Easing the workload

Using a level editor would only really help with one part of the workload, being the making of objects. The other two parts would not be significantly easier with a level editor. In order to ease the writing of hints, one might consider making a system for gathering knowledge, possibly using an external source, such as wikipedia and/or the Python documentation.

In order to ease the workload on tests and robustness, the interface could receive more features, and have a standard testing strategy built in.

28.4 Weaknesses of the Interface

There are some inherent weaknesses in the current version of the level content interface.

Robustness

While the players code is executed in a separate thread, and is should not crash the game except in rare circumstances, the scripts associated with a level are not. This is why they need to handle their own exceptions. A better solution would be one where the game keeps track of what code belongs where, and handles exceptions appropriately in a way that allows the game to keep running, even if a level crashes. It should also give some meaningful message to the person who made the level. This is possible, but complicated to do.

Large XML file

Since the interface is mostly just one XML file (per level), there will be a lot of editing done to this one file, and the user can lose track quickly.

Object placement

To place objects in the level, the user has to enter the coordinates for the object manually. And they also has to calculate these. This is tedious and error-prone.

Graphical design

The background tiles of the level need to be defined by the user. If there are edges between two tiles, say grass and lava, the game will not do anything to make these edges pretty by itself. If a user wants the edges to look nice, they will have to draw the tile for the edge itself, and repeat for every possible edge, and then place these tiles in their proper locations in the tile map. This is very tedious.

Few advanced features

There are very few finished standard objects in the interface. Mostly we have an object that can react to changes in variables, and an object that does that, and will react according to a python script as well. Some things will most likely be done many times by the level designers. Having to do these things many times adds to the workload. Once these things have been identified, they could potentially be added as built in features of the interface.

Workload

As mentioned, the levels require a substantial amount of work to be made.

28.5 Strengths of the Interface

Binding variables to object attributes, testing values and using keys for doors are very simple things to do with the current interface. Using these mechanics over and over could make for an entertaining little game. This functionality is also not

prone to crashes because of mistakes made by the player or level designer. Another strength is the fact that it works, and that it builds upon a completely open storage system that can be generated by a different system.

Chapter 29

Performance Testing

29.1 Metric

Frames per second, memory heap usage and method execution time were used to calculate, analyze and improve performance.

29.1.1 Collecting Data

To record the frame rate data the frame rate limit was disabled. The frame rates were recorded on different computers with different operating systems. The execution times were recorded of each calculation part in the frame. These times were compared to the total frame time to identify the bottleneck in case if the desired frame rate was not achieved.

For the other tests the java profiler VisualVM was used. It was used to plot memory and CPU usage and log method execution times and to collect information about object allocations. Profiler was running during short periods of single levels, and during the full playthroughs.

29.2 The tests

Pythia desired framerate was locked to the 60 frames per second. The target could be very easily achieved depending on hardware and most importantly video driver. On the windows machines this usually is no issue. The limit was disabled during some test to allow to see the full potential of the hardware.

The CPU and memory heap usage was recorded for several run instances. The arithmetic average was calculated and recorded for each system. The CPU usage has a flat pattern with little variations, while the memory heap usage has a constant growing heap until the garbage collector kicks in and the memory usage drops to the initial value. The only time the load pattern is different is during the level loading.

Computer name	Frames per second
Amilo pi 1505	55
Thin client	N/A
Thinkpad T510 (windows)	350
Thinkpad T510 (linux)	45
Mako	250
Custom AMD X6	350

Table 29.1: Framerates achieved on Systems

29.2.1 Old Laptop (2006)

The game runs well with a steady acceptable framerate above 50 frames per second. The game takes quite a bit of time to load up. It does not feel unresponsive or slow.

29.2.2 Thin Client

The game does not run and gives an error message which is consistent with graphics hardware not existing. This means that the Thin clients can not run Open GL libraries from the box. While most of the game can run on Canvas, which can be software emulated, the GUI library AWT can not. Solutions to this would be to either port TWL to Canvas, or to use the Mesa 3D library [4] to create a software emulation for Open GL. At least one of these should work, but it would require a lot of work, and would probably deliver a blow to the framerate on thin clients.

29.2.3 Normal Windows desktop

The game feels exactly like on the old laptop. The framerate shows above 80 frames per second.

29.3 Summary

Table 29.3 shows a summary of the framerates achieved when running the game for each system we tested on. Look up table 24.2 on page 60 for more information about each system.

29.4 Discussion

The performance measuring was a crucial phase in the project. The memory allocation and execution time logs were used to identify bottlenecks and eliminate

Computer name	**Avarage CPU**	**HEAP MAX**
Amilo pi 1505	55%	??
Thin client	N/A	N/A
Thinkpad T510 (windows)	2%	21 MB
Thinkpad T510 (linux)	15%	41 MB
Mako	17%	100 MB
Custom AMD X6	4%	30 MB

Table 29.2: Profiler results

them until either desired performance was achieved or when the external library like Jython has become the bottleneck. Faster execution allows more objects and interactions on the level.

In the final result the biggest bottleneck during the frame calculation were the calls to the OpenGL functions. Since it is hard to improve execution time of these functions and the next most lengthy executed method used only two thirds of the OpenGL execution time, there were no need for further optimization.

As for the memory allocation this one was hard to track down. During the tests every frequent identifiable allocation was tracked down and optimized. However some nameless allocations were hard to track down. These were most probably connected to the Jython implementation and therefore could not be well optimized without touching the Jython's framework.

29.4.1 Thin Clients

The game in its current form does not run on the university's thin clients, which would make it available to all students. There are two solutions to this problem. The first one is to use software implementation of OpenGL. This solution is quite simple since it does not require any code changes, however such implementations usually are very slow compared to their hardware counterparts.

The second solution is to port Pythia's rendering to software. This solution requires a bit of work since some of the libraries used in Pythia are dependent on OpenGL, so these libraries have to be ported as well. The libraries in question are the user interface libraries and are essential to the game. The user interface library could be replaced, but that would require rewrite of large portions of the code and could take some time.

Conclusion

With the exception of the thin clients, the performance of Pythia is acceptable for the complete computer systems it was tested on. None of the users mention

performance issues during the user testing phase.

Part VI

Summary

Chapter 30

Conclusion

Gameplay

The scores from the user test are higher than we initially expected. It seems like players actually find the game fun and educational. This is good because it will strengthen the effect it would have if used as a tool for learning programming. It seems like players with lower knowledge will struggle more with the game. Hopefully, if learning programming helps you in our game, learning our game will also help you in programming. The casual gamer is not very skilled in programming, and might not enjoy this game the way it currently is. However, this game was not aimed at the casual gamer, but at people who actually want to learn programming. Mostly this comes down to the nature of the puzzles themselves. But it should not be too difficult to warrant its use as a supplement to a course in programming.

Portability

The system has few problems related to portability, but there is one that stands out, which is the fact that it can currently not run on the NTNU thin clients. This is due to a choice we made earlier in the project when we favoured the speed from hardware acceleration over the apparent better portability of software rendering. We feel that this could possibly be remedied with more work, but that if it is not, it might not be feasible to use the game as a part of the course in basic programming unless we can assume that all students have access to a computer made in the last 5 years that has actual graphics hardware.

Modifiability

The game content can be modified very freely and there are few constraints on what kind of levels can be created. Even though content can be created to match the

complete programming part of the ITGK module, a graphical level editor would really simplify the creation of levels a lot since the work is tedious.

30.1 Conclusion

Our system gets close to being good enough for use as a part of the basic programming course at NTNU. But unless we can assume students all have access to computers that can run it, it would not be fair to use this as a part of the course. However, we feel that with some more work, there might be some hope yet.

Chapter 31

Further work

31.1 Crash logs

Give the user ability to easily report the problem after a crash of Pythia. Generate detailed crash logs which could used to easily reproduce the circumstances which resulted the crash and fix the problem.

31.2 Buying hints

A system could be in place where the player gains points for completing a level. These points could be used for buying hints on difficult puzzles. If the player runs out of points, they can back on level 1 and gather points all over again. This would cause them to have some repetition of the concepts up to the one they are currently stuck on, which might help them, and they will gain the points to buy a hint.

31.3 Methods as level objects

While the game content could be manipulated to show something that appears to be a method as an object in the game, this idea is more related to showing how functional programming works. For instance the player could push the multiply method in between two variable objects, and the result would be shown somewhere. This is not as difficult as it might seem.

31.4 Level Editor

It would simplify things a lot to have a graphical level editor that lets you create a level without typing everything in text, that sets up all the references and lets you edit anything you want to easily.

31.5 More content

The game needs more puzzles and more hints.

31.5.1 The story

While we think that the story is pretty good, it needs a better delivery. It cannot be as rushed as it is in the current version of the game if it is to have any impact on the player at all. This requires more content and simply put, more written story.

31.6 Variable viewer

During the planning phase of the project the code editor was supposed to have a table with all declared variables and their values. This was taken out for now as it might be difficult to filter the variables that are somewhat safe to edit, and also to hide the variables that are used by the system for testing and so on. Without variable viewing debugging becomes less useful.

31.7 Security Measures

See chapter 22 for a few suggestions for improvements on security.

31.8 Port the game to software emulating

Make the game run on computers with minimal graphics hardware. Described closer in chapter 7.1.

Part VII

Appendix

Appendix A

Building Pythia from source

This is the suggested method to build Pythia from source:

1. Get Maven (http://maven.apache.org/)
2. Navigate to the 'pythia' folder with pom.xml in a command line interface
3. Type 'mvn install' and wait
4. The runnable jar will be in the 'pythia/target/' directory.

Appendix B

Creating your own level

As there is currently no editor for creating levels, To create your own level, the first thing you need to do is to create a folder with the name of your level, and put a file named 'level.xml' inside the folder.

Level As the root element in the XML, you should put the following:

```
<Level xmlns:xsi="http://www.w3.org/2001/XMLSchema" ↩
    xsi:noNamespaceSchemaLocation="../../schema.xsd">
</Level>
```

This way, if you use an XML editor, you can get some auto-complete options. While editing, you can attempt running Pythia with your level, and it should (hopefully) give some clues to what is wrong, if something is missing. There are some things you need to have on a Level before you can get anywhere.

Layout A level must have a Layout tag. This must define the dimensions of the level in tiles, and also give a pointer to the file that contains the actual layout. It must also define the tiles that the layout will be using.

LevelGraphics LevelGraphics is where you list the textures you want to use on objects in the level.

WelcomeMessage Allows you to point to a file containing the Message that pops up when the level is loaded.

B.1 LevelObjects

You need to surround all the level objects with the LevelObjects tag. Each level object is then defined by using BehavedObject, DynamicObject, CompositeObject and Exit.

appearance Used to set which texture the object should use.

name The name of the object.

Conversation Point to the files that hold the text for the pop up window when you talk to this object. (like the welcome message)

CollisionScheme Describes the collision model of this object.

bind Bind an attribute of this object to a python variable in the game.

behaviour Points to a Module that is run when this object needs to perform an action. (Behaved object only)

target (Exit only) sets the name of the level to be loaded when this one is complete.

B.2 Modules

Modules can be defined in a Level or in the Globals.xml.

src The path to the .py-file that contains the code.

name The name of this module to be used when objects use it as their behaviour or test etc.

classname The class name of the internal class that takes the role of a Behaviour or Test.

B.3 Tests

All tests have the hint attribute, which describes the text to be shown in the in-game objectives.

PythonTest Points to a Python Module that must return true for this test to pass.

TestEquals Checks if a variable is equal to a literal string.

TestOutput Checks the console output against a regular expression string.

TestListEquals Checks the contents of a list of choice.

ImpliciteObjective Does not have a test to pass, but will show the hint as an objective.

Appendix C

How to run your own level

java -jar pythia.jar -x path/to/the/level/folder

Appendix D

How to Play

D.1 General

You can control your character's movement with the directional keys (arrows) on your keyboard. Pressing enter when your character is near an object will attempt to interract with that object. On the top of the screen you will see short descriptions of what the various function keys (on your keyboard) will do.

D.2 The Console

Enter code and press execute with enter. Use Ctrl-Z to interrupt.

D.3 The Code Editor

Enter code and press execute with F5. Use Ctrl-Z to interrupt. Use Ctrl-B to set breakpoints. Step and continue with F6 and F7. Use F8 to terminate execution in progress.

Appendix E

Questions used for user feedback

These are the questions used in our user test

E.1 Usability

(These are from the standard System Usability Scale [10])

- I think that I would like to use this system frequently
- I found the system unnecessarily complex
- I thought the system was easy to use
- I think that I would need the support of a technical person to be able to use this system
- I found the various functions in this system were well integrated
- I thought there was too much inconsistency in this system
- I would imagine that most people would learn to use this system very quickly
- I found the system very cumbersome to use
- I felt very confident using the system
- I needed to learn a lot of things before I could get going with this system

E.2 Learning

- I learned something from playing Pythia.
- I did not learn anything about programming in general when I played Pythia.
- Playing Pythia made me curious about programming or Python.
- The learning curve of Pythia was too steep.
- I wanted to look something up on wikipedia while playing Pythia.
- I do not see any use for Pythia in learning.
- Pythia motivated me to try to learn more about Python.
- Pythia made me confused about Python.
- Pythia gave me a basic understanding of programming.
- There were things mentioned in Pythia I do not currently understand.

E.3 Fun

- I finished the entire presented version of Pythia.
- I am glad the game was not longer.
- I had fun while playing Pythia.
- I felt like Pythia was a chore.
- Pythia did not restrain my creativity.
- I felt like I was restricted from doing what I wanted inside the world of Pythia.
- I would like to see more of Pythia.
- There was no sense of progression in Pythia.
- I felt Pythia delivered a story.
- Pythia was annoying to play.

Appendix F

Source code and binaries

The Game A runnable release of the game can be found at this location:

`http://master2011.org/release/pythia.jar`

Source code All source code can be gotten from the following git repository:

`git://master2011.org/master.git`

To download the source the *git clone* command can be used as such:

```
git clone git://master2011.org/master.git
```

Bibliography

[1] The awt in 1.0 and 1.1. http://java.sun.com/products/jdk/awt/.

[2] Java bindings for opengl. http://java.net/projects/jogl/.

[3] The jython project. http://www.jython.org/.

[4] The mesa 3d graphics library. http://www.mesa3d.org/.

[5] Python programming language official website. http://www.python.org/.

[6] Sax project. http://www.saxproject.org/.

[7] Wikipedia: Likert scale. http://en.wikipedia.org/wiki/Likert_scale.

[8] Philip Kortum Aaron Bangor and James Miller. Determining what individual sus scores mean: Adding an adjective rating scale. *Journal of Usability Studies*, 2009.

[9] Sapan Bhatia. Core wars. http://www.corewars.org/, April 2011.

[10] John Brooke. Sus - a quick and dirty usability scale.

[11] Mary Campione and Kathy Walrath. Java. http://download.oracle.com/javase/tutorial/getStarted/intro/definition.html.

[12] Daniel Floyd. Video games and learning. http://www.youtube.com/watch?v=rN0qRKjfX3s, February 2011.

[13] Barry Warsaw Guido van Rossum. Style guide for python code. http://www.python.org/dev/peps/pep-0008/, June 2011.

[14] MIT Media Lab. Scratch. http://scratch.mit.edu/, February 2011.

[15] Tim Lindholm and Frank Yellin. *The JavaTM Virtual Machine Specification, Second Edition*. Prentice Hall, 1999.

[16] James Portnow. The power of tangential learning. http://www.next-gen.biz/blogs/the-power-tangential-learning, September 2008.