Filurk Online

Jimmy Nyström and Nicklas Nyström
Luleå University of Technology

Abstract—Filurk Online is a website that was done as a project in the course *Design of Dynamic Web Systems*. It features an online version of a puzzle game called Filurk, as well as all kinds of information about the game. Data can be stored on and be retrieved from the server using a restful application programming interface, which can be used for synchronizing data between different front-end applications in the future.

Index Terms—Filurk, online gaming, RESTful server.

1 Introduction

THE Filurk project began a couple of years ago as a color based puzzle game developed in Game Maker. Since then, we have developed versions of the game for Mac, PC and the iPhone/iPad, which will hopefully be released on Apple's App Store in the near future. With even more versions coming soon, a way to synchronize data between different clients is becoming necessary. We want a way for a user to be able to play the game on his smartphone, take a break, and then start the game on his computer with the same data available. After finishing the game, some users may also want to create their own levels and share them with others.

These are problems that we have been trying to solve with the Filurk project in the course *Design of Dynamic Web Systems* at LTU. We have developed a website where you can play Filurk online, with one servlet that stores the player's solutions to the levels, another that can generate map images and one that provides storage for user created levels. This report describes the work of designing and building Filurk Online!

called *Project TRASH* [1]

January 13, 2012

Jimmy Nyström's Email: jimnys-8@student.ltu.se
Nicklas Nyström's Email: nicnys-8@student.ltu.se

Fig. 1. A collage of the different versions of Filurk

2 THE GAME

Filurk is a simple but fun 2D puzzle game played with a top-down perspective. The controls are very simple: the main character, Filurk, can be moved in four directions, one step at a time. The game consists of a large number of levels, all filled with blocks, locks and water holes. A level is completed when Filurk enters the door, which is usually closed. To open the door on each level, Filurk needs to bring color to all Wantings (strange, statue-like characters) by stepping on colored floor panels and painting a trail of color to them. The goal is to reach the end of these levels in as few steps as possible. After finishing a level, the player is rewarded with either a bronze, silver or gold medal, depending on how many steps

were taken to do so. Learning how to play the game is most easily done by simply going to the website and playing the game.

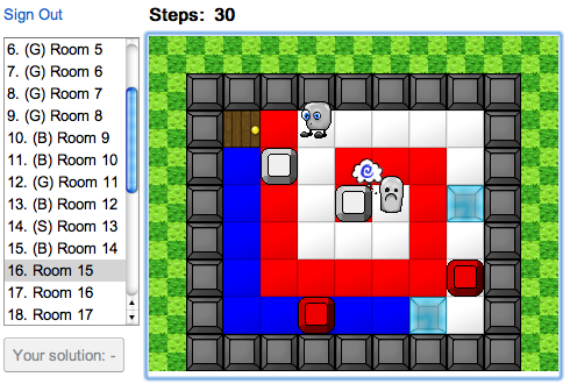

Fig. 2. The front-end application

3 ARCHITECTURE

3.1 FRONT-END

Our front-end is a website created with Google Web Toolkit which does yet not make full use of the functionality provided by the back-end. On the website, a version of the game Filurk can be played. When starting the game, the user is prompted to sign in with his/her Google Account. The URL the user is redirected to is fetched from the login servlet, and the room data and the user's solutions are fetched from the room data servlet.

The room data is fetched when the game is started and then stored locally by a data manager in the client. As the user solves rooms, the solutions are stored locally by the data manager and simultaneously sent to the server for validation and storage. The solutions can be replayed in the game by simply pressing a button.

3.2 BACK-END

The client and server are completely separated by a well-defined API. All communication is initiated by the client through one of the HTTP requests listed below. The website is located at http://filurk-online.appspot.com/. A description of all servlets and their APIs are listed along with the supported HTTP request types, such as GET, POST, PUT and DELETE.

3.2.1 LOGIN

The *login servlet* generates URLs for use when signing in with a Google Account. The client can pass a URL as a parameter to the servlet to control where the user is redirected after signing in or out.
URL

/filurksite/login

GET

- Parameters: url (optional) - to use when generating a redirect address
- Result: Returns a login and a logout url, unformatted (at the moment)

3.2.2 ROOM DATA

The *room data servlet* keeps track of room data, both shared and user specific. Which rooms are returned by the servlet is decided by an XML-file on the server, but the user data is stored in such a way that the correct data for each room is found even if we change the contents of this file. The types of data stored on the servlet are room layout, solutions, records and medal limits. These are presented in some detail below, but for the exact format of the data returned by the servlet, see Appendix B.

The layout of a room is stored as a string, where each symbol represents an object in the room, or an empty space. This string is used to identify each level. We first considered to use the room name or the index as a unique identifier, but these values will probably change as the development of the game progresses. The room layout itself is more constant.

A solution string consists of a sequence of key presses. Observe that the record (the lowest number of steps) is not necessarily the same as the number of key presses in the solution: a key press might for example correspond to a lock being opened, which does not constitute a step. This is why both the solution and the record need to be stored.

Each level has two medal limits: silver and gold. These are just integers, which represent the fewest number of steps you must finish the level in to win that medal. If the number of steps is above the silver limit, a bronze medal is obtained.

This, and some of the other servlets, need the ability to validate room data and solutions. For this purpose, we emulate the game on the server in a class that can read a room layout string and tell if the data is invalid in some way. At the same time solutions can be tested with the room layout, to count the number of steps and check if the solution string actually solves the room.

URL

/filurksite/roomdata

GET

- Parameters: none
- Result: Returns all rooms in XML-format (see Appendix B), along with user specific solutions to the rooms, if the user is authenticated.

POST

- Parameters:
  - data - the room string
  - solution - solution to the room
- Result: The solution is validated and stored on the server.

3.2.3 CUSTOM ROOMS

The *custom room data servlet* manages user created rooms. This is meant to be used by a level editor. We didn't have time to make an online level editor to display the functionality of this servlet during the course, but the servlet has been put through some testing and (presumably) works as intended.

URL

/filurksite/custom

GET

- Parameters: None at the moment, but the API will be expanded in the near f uture to support fetching single rooms.
- Result: Returns all custom rooms created by the user in XML-format (see Appendix B). Nothing is returned if the user is not logged in, but plans for public custom rooms are in the making.

POST

- Parameters:
  - data - the room string
  - name (optional) - name of the room
  - solution (optional) - solution to the room
- Requirements: The user must be logged in.
- Result: The data string and solution are validated, and the room is stored on the server. Returns the unique identifier for the room, which can be used to update rooms.

PUT

- Parameters:
  - id - unique identifier for the room
  - data - the room string
  - name (optional) - name of the room
  - solution (optional) - solution to the room
- Requirements: The user must be logged in.
- Result: The data string and solution are validated, and the room is stored on the server. Returns the unique identifier for the room, which can be used to update rooms.

DELETE

- Parameters: id - unique identifier for the room
- Requirements: The user must be logged in.
- Result: The room is deleted from the server.

3.2.4 MAPS

The *maps servlet* takes a room layout string as input and generates a map image of it (see Figure 3). This can be used for making preview thumbnails of the levels in order to make the level selection easier and provide a nicer look to the user interface of the game. It will be integrated with the web version of Filurk in a near future.

URL

/filurksite/maps

GET

- Parameters:
  - data - the room string
  - size - size of the map to be generated, can be small, medium or large (and mini is coming soon)
- Result: Returns a PNG image of the room.

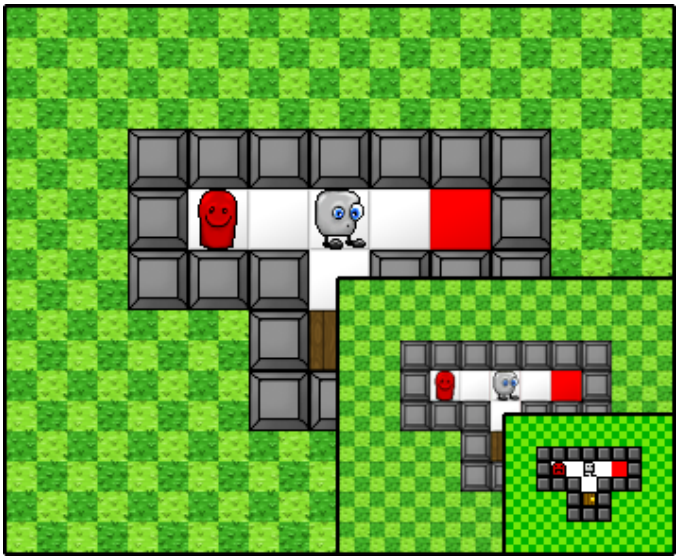

Fig. 3. Example of a large, a medium and a small level map

4 TECHNOLOGIES USED

We have used Google Web Toolkit, a development toolkit for building browser-based applications. It is open source and free of charge. Using GWT meant that we could write all our code in Java, which is a language we are both familiar with, which meant we could get started quickly. Another reason is that we found useful and easy-to-follow tutorials for GWT on the Internet[2].

Google App Engine, or simply App Engine, is a platform for developing and hosting web applications. It automatically allocates more resources for an application as the number of requests increases, and the service is free of charge up to a certain level of consumed resources. This makes it ideal for our purposes, as we (perhaps naively) hope that the number of people who visit our site will grow with time.

For versioning, we have used a Git repository in a shared Dropbox folder. This was chosen mainly because it was fast and easy to get started with. However, we ran into some problems with branches and conflicting files that were hard to resolve. We didn't use a public repository because we didn't want to share all game resources publicly, such as the files containing the best solutions to all levels.

To keep track of traffic on the site, we use Google Analytics. So far, there has not been much traffic, but we're hoping that will change in the future!

No sensitive data is stored on our server, but authentication is still needed in order to determine what data belongs to which user. We chose to do authentication using Google Accounts. Firstly, a lot of people already have a Google Account, so most users won't need to sign up for anything, just accept to authenticate. Secondly, it has good support in GWT, which makes it very easy to integrate with the application. Lastly, it is - as far as we know - a secure way to do authentication. On the website, if the user is not logged in when trying to start the game, they are redirected to a sign-in page where they can either sign in or create a new account. This might change in the future so that people can try the game without signing in, since the server supports fetching room data anonymously.

5 PROBLEMS

Our work with the client-server communication progressed nicely at first, and early on we had a working version that used remote procedure calls (RPCs). This had to be changed, however, since our architecture needed to be RESTful according to the course goals. Switching to regular HTTP servlets caused some problems. Using RPCs is mostly just like regular Java programming. Neither of us has much experience programming for the web, which became more apparent when switching to HTTP servlets. Simply returning an object is a lot simpler than creating an object, encoding it in XML format (see Appendix B), sending it, parsing the XML in the client and recreating the objects. Simply put, the programming got more complicated!

6 FUTURE WORK

6.1 LEVEL EDITOR

The API already supports the possibility to add your own custom levels to the server, but we

have yet to add a level editor to the website. We make our own levels in a level editor that Jimmy has developed (see Figure 4), which works very well. Adding this to the website would be a natural next step, and would add a lot of depth to the gaming experience.

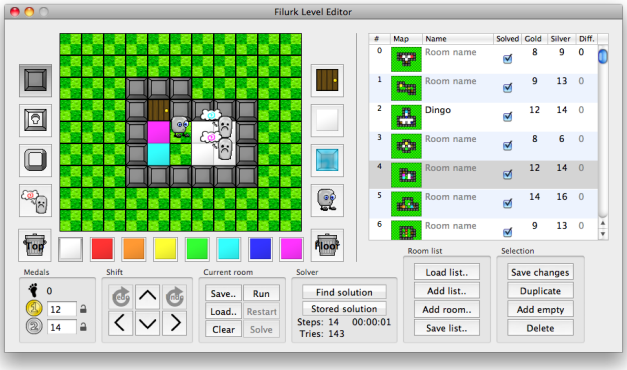

Fig. 4. Screenshot of the the Filurk level editor

6.2 LEVEL SOLVER

During the course of this project we have done a lot of work on a level solver that we have not added to the website.

Since the purpose of the level solver is to find the shortest possible solution, it is based on a breadth-first algorithm. This means that the time it takes to complete increases exponentially with increasing solution length, which makes for a very inefficient implementation. What makes the solver work is that it has several efficient ways of ruling out invalid solutions at an early stage.

However, adding the level solver to the website opens up for the possibility to immediately find the best solutions to some (but not all) of our harder levels, which we don't want. More about the solver can be found in Appendix A.

6.3 Maps

We plan on making use of the maps API to present miniature maps (see Figure 3) in the list of levels. This will make the game more graphically appealing, and will help the user find the correct levels easier.

6.4 GRAPHICAL DESIGN

The graphical design of the website has not been our top priority, and still needs some work. For example, we need to redesign the logo. We are also going to update the graphics in the game itself. In the version of Filurk running on iOS, the game contains smooth animations, some 3D and particle effects.

7 CONCLUSION

The result of our project in the course *Design of Dynamic Web Systems* is a 2D puzzle game called Filurk, which is playable directly in a web browser of choice. The application communicates with our server through a well-defined API, uploading and retrieving game data in real-time. Authentication for handling user specific data is done through Google Accounts. The server may be used by other external applications in the future, for example a level editor or other versions of the game: Filurk has been implemented for PC, and versions for iOS are on their way. Some further work is planned for the project, such as some graphical improvements on the client application, but we have a functioning version of the game up and running on our website, http://filurk-online.appspot.com/.

REFERENCES

[1] https://sites.google.com/site/pitebroder/
[2] http://code.google.com/intl/sv-SE/webtoolkit/doc/latest/tutorial/

APPENDIX A
SOLVING FILURK

The goal of Filurk is, as you all know by now, to get into the door in as few steps as possible. When doing so, the player is awarded either a gold, a silver or a bronze medal. But how are the values for the medals decided? Our approach has been to award gold medals only to optimal solutions - but can we know for sure that a solution is optimal?

It should be noted that work on the level editor and solver started before the course M7011E and project FilurkSite, but we improved on it a lot during the course and we thought this might be interesting.

A.1 SOFTWARE REQUIREMENT

Input: A Filurk room string (as defined in Appendix B). Output: A solution string that solves the room in as few steps as possible, or null if the room is impossible to solve.

A.2 INITIAL APPROACH

One way to ensure that the first solution found is the best one is to simply test all possible paths, starting with paths of length 1, then paths of lengths 2, 3, and so on.

We use a tree as data structure for keeping track of paths. Each nodes stores a direction of movement and four possible children (left, right, up, down). The problem of solving Filurk comes down to performing a simple breath-first search (BFS) on the tree! However, if we look at the worst case performance of this algorithm, a new problem appears.

Each step Filurk takes generates four new possible steps from a new position. The number of paths with length $n = 1$ is 4. With $n = 2$ there are 4^2 possible paths, $n = 3$ gives 4^3. In the worst case, if the best solution is b steps long, we will need to try $4^1 + 4^2 + ... + 4^(b - 1) + 4^b$ paths before finding the solution. Some rooms take almost 200 steps to solve, which puts the number of paths at around 4^{200}. Our path tester can try about five million paths per second: $4^200/(5e^6 * 3600 * 24 * 365) = 1,64e^106$ years. The universe has plenty of time to be destroyed and recreated several times before the solution is found.

There has to be a better way!

A.3 IMPROVEMENTS

We have found many ways of improving the algorithm. We still use a BFS, but with large amounts of tweaks and improvements that hinder the growth of the solution tree by ruling out erroneous paths as early as possible. Some useful data about the room can be extracted before starting the testing. For example:

- Check if all needed colors exist and where
- Find which locks need to be unlocked
- Find all paths from Wantings and locks that need to be painted in a certain color
- Find which blocks need to be pushed into water and areas where those blocks cannot be pushed

Then, paths of increasing length are generated and tested in sequence. At the end of each path, we do a series of tests:

- Check if all colors are still available
- Check if needed blocks can still reach water
- Check if the door can still be reached
- Store and compare the state of the room at the end of each path, and stop if the exact same state has been reached earlier (this means that a path of equal or shorter length with the same end result exists, and the current path is not optimal)
- Avoid creating looping paths (to a certain degree)
- Abort when bumping into solid objects

Although some of these tests take a lot of time (and memory), such as the state comparison, it still saves time in the end by cutting enough branches of the path tree.

A.4 RESULTS

The improvements work really well, in some cases. Small rooms are solved within seconds, and most medium sized rooms are solved in a matter of minutes. However, with larger rooms the performance of the solver depends entirely on the layout of the room. If there are large, open areas covered with white floor, our

improvements can't rule out enough solutions and the path tree grows much too fast. It's still easy to build a room that takes forever to solve.

A.5 DISCUSSION

Having a solver running on the Filurk server is not feasible, since there is no way of knowing how long it will take to solve a given room or even if it ever will. Trying to create a solver has been an interesting problem and the produced result has proven valuable when designing rooms, but some rooms still take millions of years to solve.

APPENDIX B
DATA FORMAT

B.1 INTRODUCTION

To be able to use the APIs of our server, some data formats need to be known. All formats used in the game and by the servers are presented here.

B.2 ROOM DATE STRING FORMAT

A room in the game is viewed as a two-dimensional matrix with 9 rows and 11 columns. The room data strings are read from left to right, adding objects to a room in three passes (layers). In the first pass, floor objects are added, then moving objects and lastly solid objects. The type of object to add is identified by the layer and the character in the string. The position of the objects is simply decided by the index in the string, starting from the beginning of each layer.

Common for all layers are empty spaces. Characters 'p' to 'z' represent 1 to 11 empty spaces. The encoding of the rest of the game objects are described below.

FLOOR LAYER
- a: White floor
- b-h: Colored floor
- 2: Door
- 3: Water

MOVING LAYER
- a: White block
- b-h: Colored blocks
- 1: Filurk

SOLID LAYER
- a: Wall
- b-h: Wantings
- A-H: Locks

B.3 EXAMPLES

Some examples are provided for testing the APIs. The strings below should be written without spaces or newline characters:

1) zzzraaaabrtatt2tzzzzzzt1tzzzzzzzqaaaaaaqqabsaqqaaapaaaqsapassaaaszz
2) zzzsaa3ssa2assaa3szzzzzzzs1gfszzzzzzraaaaarrararrararragFparraaaaarzz
3) zztatrc2aperrapaparqapapapaqr3papbrtatzzzzzudss1uzzzzsaaasqaaabaaaqqataqpaataappaetcappaataapqaaadaaaqsaaas

B.4 SOLUTION STRING FORMAT

A solution string represent a sequence of steps in four directions, 'l' for left, 'r' for right, 'u' for up and 'd' for down.

B.5 ROOM DATA XML FORMAT

The format of the XML-data returned by the room data servlet is:

```
<?xml version='1.0' encoding='utf-8'?>
<root>
  <room
    name= Name of the room
    data= The room data string
    gold= Gold medal limit
    silver= Silver medal limit
    best= The user's best solution
    solution= The user's solution
  />
</root>
```

B.6 CUSTOM ROOM DATA XML FORMAT

The format of the XML-data returned by the custom room data servlet is:

```
<?xml version='1.0' encoding='utf-8'?>
<root>
  <room
    id= Unique identifier in the database
    name= Name of the room
    data= The room data string
    best= The user's best solution
    solution= The user's solution
  />
</root>
```

B.7 LOGIN INFORMATION

We had problems with the string encoding when using URLs in XML tags and decided on going with a simple approach. The login servlet just prints three lines of text:

Login url
Logout url
Login status (true/false)