02122 Project course, Spring 2014
DTU Compute
Technical University of Denmark

Anabasis
Group 6.1

s113414 Cebrail Erdogan
s123062 Carsten Nielsen
s123094 Jonathan Becktor
s123995 Philip Berman

June 19, 2014

Abstract

Using the Arduino hardware with Gameduino 2, we will create an advanced game. The game will have elements as AI and Map generation. Coding in arduinos environment and make it work in the limited hardware is quite a challenge and fun.

Contents

Chapter 1

Introduction

The main purpose of the project is to make an advanced game using the Arduino hardware. Arduino is a programmable piece of hardware. Combining it with the Gameduino 2 makes it possible for us to create a rich game. The Gameduino extends the Arduino with a touch screen, extra space and processing power.

1.1 Arduino

Programming an Arduino is done in its own language, simply named *Arduino programming language*[1]. It is based off Wiring, and is reminiscent of both C and C++, but is neither. The IDE is based off the Processing IDE.
This mix of languages and sparse documentation makes it difficult for the programmer, even though the Arduino was meant to ease newbies into programming. An example, would be the keyword `new`, which officially is not supported[2]. This is confusing, because `new` is completely functional, even syntax highlighted. It apparently instantiates an object in the heap rather than the stack, and returns the pointer to this object. Other unlisted keywords we found during development were close, delete, home, speed, step and update. Some of these might be from externally included libraries, whose variables and functions the IDE syntax highlighted.
Our arduinos and their technicalities

1.2 Gameduino 2

The gameduino and technicalities.

[1] http://www.arduino.cc/

[2] The official reference page http://arduino.cc/en/Reference/HomePage does not include the keyword.

Chapter 2

Problem Analysis

Define domain specific concepts to be used in the rest of the report.
Explain the problems considered, features to be implemented, etc.
Possibly merge with the next section.

Chapter 3

Requirements specification

3.0.1 Gameplay

We originally wanted to implement a rpg style platformer with:

1. Items
2. Hero leveling
3. Abilities
4. A decent story

We quickly found out that the limited flash memory on the arduino would greatly limit what we could implement. We agreed on going back to the classics of arcade games. We decided to create a "rogue like" game and focus on increasing difficulty per level and having a high score as the intensive to play the game.

High Score

High scores is something you almost always see in arcade games or just smaller games. Its a great way to compare and compete and to show who's the best. To do this we were requierd to save data on the EEPROM which is the hard drive‘ of the arduino. This way the high score will never reset unless we want it to.

Coins

Coin were added as an additional game play element to broaden the game. The player now has an incentive to go explore the entirety of the map, since collecting coins is an easy way to get additional score.

Chapter 4

Development Process

Work schedule during 13 week period.
during 3 week period.

4.1 Timeplan

There are several time planning models in the software world. There is agile, iterative and incremental.

When we started the 'Fagprojekt' we created a waterfall chart containing our plans for every week, it was structured in a waterfall chart. The waterfall chart is a sequential design process. It is designed to get through the project phases and have a product as soon as possible. The phases in our project can be seen in the figure below.

As we revisited our waterfall model steps over time, our main time plan model can be considered to be iterative. The revisits have mainly been to extend features,debugging or optimizing.

When the waterfall ends and we still have time we will go back and visit the steps and check for new requirements.
The waterfall gives a good picture of the big phases, but the pre-planned week schedules are not always much help, as they are not dynamic. We can't reconstruct our waterfall every time we meet a conflict. This is here where the timeboxes are handy, which is used to the most detailed part of the time planning - see next subsection. You can check our waterfall timeplan in the appendices, please see Figure 11.2 for that.

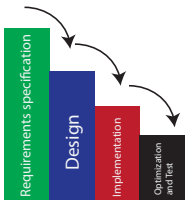

Figure 4.1: An overview of the waterfall phases.

4.1.1 Time Boxing

We were conviced that using "Time Boxing" would be the way to go. Timeboxing divides The schedule into a number of separate time periods(timeboxes), with each part having its own deliverables, deadline and budget. Breaking bigger tasks into smaller tasks with better manageable time frames. What also is important is that by the end of each timebox we need to have a product that if all else fails we can roll back and release our game from an earlier state. The following table shows the timeboxes we have created during the project.

week 8-10	**week 11-13**	**week 14-15**	**week 16-17**
Code exercise	Enemies	Scene generation (simple)	Graphics
Game design	Collision Detection	Player	Sprites
Class Design	AI		Map generation
	Input		
Report	Report	Report	Report

week 18-90	**week 19**	**week 20**	**week 22-25**
Endgame	Scene generation	Sprites	Optimization
Map generation	Optimization	Sound	
	Attack	Optimization	
		Animation	
Report	Report	Report	

Table 4.1: The timeboxes are seperated per week

4.2 Hardware

4.2.1 Arduino problems

Throughout our implementation of the game we have had several issues with our arduinos.

Arduino burnout

The first problem we ran into was one of our arduino burned out. It was somewhat fixed it would work one day and wouldnt another. Slowly it gave out and now it works 1 out of a 100 tries. We could not find a fix for it. This slowed us down quite a bit since it limited what we could test when working at home.

Arduino duemilanove.

We ran out of flash memory on the duemilanove since it only has 16kb, this set us back since we only had 1 arduino to code on. We later got this replaced with another leonardo.

Arduino blackout

We have also had some issues when uploading code to the arduino. Sometimes the arduino screen is black after uploading. To fix this we had to upload an example file from the arduino library that prints text to the screen.

SD card faliure

Whilest working on the project suddenly when uploading the arduino couldnt find the SD card in the gameduino. The card was in and had the right files on it. We tried to format the card on windows, it still didn't work. We had a try on a mac and it did. On one of our arduinos we had the blackout problem as mentioned above but we could not seem to get it fixed. This appeared to be the same problem that the SD card had gone corrupt and needed a format.

Chapter 5

Overall Design

Here we describe each component needed in the implementation. We detail the requirements and uses of each component, as well as argue for their existing. This was informally described during development, used in the implementation, but also serves as a rough description of the overall system.

5.1 Objects

Everything in the game-space is a *prop*. These are the objects which the physics engine works with. Each of them contains a *hitbox*, a rectangular square used to calculate collisions. In addition they also contain graphical information needed to be drawn, since all props should be.
Next we have *units*, which would be all enemies and the player. All units are props, and are part of the physics engine. The difference is that a unit has an AI to update too. As a side note, we did not name them actors in suit of the theatrical names, scenes and props, since it would be confusing that all actors are props.
The only reason to distinct units and props are because of *coins*. These grant the player a score bonus when collected. The reason for coins to extend props is to reuse collision detection code from props, and makes them affected by the physics engine as a bonus.
Two kinds of unit extensions exist: the *minotaur* and the *hero*. The former is the only implemented enemy type in the game and the latter is the unit controlled by the player. These contain graphical and gameplay data as well as their AI code. In case of the hero, his 'AI' is reading player input.

5.1.1 Minotaur AI

The AI needs to receive world information and deliver actions. This prompts a cyclic model-controller relationship, which aims to place as much freedom in the hands of the AI

as possible. The biggest limiting factor is how complex the world is, most of all the physics engine. The AI cannot and should not predict how its actions would affect the world - this is the job of the physics engine. This limits us in how advanced the AI can be, especially when planning forward, since we cannot guarantee an action leads to the desired outcome. For example, if a unit would jump across a gap or on top of a platform, he would need to steer himself for several frames to land safely and surely. Moreover, planning further than the current frame would require the unit to have a concept of his jumping abilities, size and world geometry.
This complexity leads us to create much simpler AI, one which does not plan ahead. A possible solution which was considered, would be preprocessing the map and generate paths through the map. Though this would not be expensive in memory and code-size, and would make different enemy types problematic. Either we would be forced to use similarly moving enemies or generate additional pathing maps for each enemy type. This would be a lot of work for a bit smarter AI, and it was not a very enticing feature with such limited hardware.
The AI we settled with would simply move towards a given goal, jumping if necessary, or wander aimlessly like enemies usually do in platforming games.
Communication between the unit and the scene is done by an object called *logic*. This should contain all necessary methods and information needed by all unit types. A few examples would be the hero's position, whether or not a given prop is in the air or whether a given space is 'walkable'. It would also need to execute actions given by the units, which is the physics engine. So it handles collision detection, gravity and calls relevant methods in case a prop collides with a wall or is attacked.
As mentioned there is a cycle of dependencies: Logic requires scenes for map information, which includes units, which need logic to communicate. We attempted to avoid cycles which could be a problem during implementation, but it seemed unavoidable when an informed AI is in play.

5.1.2 Hero Input

The hero will be a human like character. Therefore he will be capable of doing basic things like walking and running. To interact with enemies and navigating through the map he will need to duck, jump and attack. He can even attack in the air. All these actions require that we have a proper input system.

To achieve this, we chose to use the Wii Nunchuk. It has one thumbstick and two buttons. It does also have an accelerometer built inside. We assigned the thumbstick to move horizontally and duck. The two buttons at the top are used to jump and attack. To finish a map, press attack and duck at the same time.
These buttons assignments was the combination that felt most comfortable. As the Nunchuk doesn't have many buttons, we had to be creative.

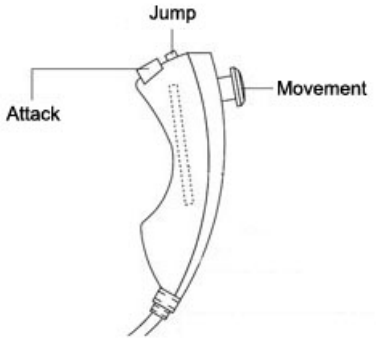

Figure 5.1: Button assignments

Here is an overview of how to control the hero.

Action	Combination
Horizontal movement	Thumbstick left and right
Jump	C Button
Attack	Z Button
Duck	Thumbstick down
Exit map	Thumbstick down + Z Button

5.2 Levels

The levels in the game are called *scenes*. They contain the world geometry and all the props. The world is made out of square blocks called *tiles*. There are multiple types of tiles, used to diversify the world and mark special locations. The former kind is solid blocks and partially solid platforms, while the latter is the entrance and exit. The scene is composed of a *grid* of tiles which contains exactly one entrance and exit. Each level is a climb to the top, with the entrance at the bottom and exit at the top. Minotaurs and coins are randomly placed around the map, though not too close to the entrance.
Initially, ladders were also supposed to be in the game, but their function would have overlapped with platforms. Ladders would be difficult to implement and would require specific animations, both for the player and the enemies. They were cut from the game in favor of the easier to implement platforms.

5.2.1 Level Generation

The point of randomly generated maps is two fold: it vastly increases the replay-ability of a game, and it encourages more skilled play by forcing players to be better at the gameplay elements, rather than the specific levels included. We can ensure this if the maps are

adequately different from each other, and if the range of possible maps is large enough. More specifically, the level generation should return a new scene with a matching entrance and exit. The level should always be solvable, that is, there must be a path between the entrance and exit which the player can follow. It should not be possible to get stuck in the map, though it may contain unreachable places. Enemies and coins should be spread evenly around the map, though not too close to the entrance, where the hero would spawn. In addition to this, the level generation should be able to vary the difficulty, at least by varying the amount of enemies, to allow increasing the difficulty as the game progresses. New maps are rarely needed compared to other code parts, so speed is not a priority. Memory and code size are the greater evils in this scenario.

5.3 Graphics

Gameduino...

5.3.1 Assets

Assets...

5.3.2 Sound

We implemented sound effects on essential events. We Agreed that it was what the game needed to give a better feel.
We decided that the sounds we wanted was from the hero, and when he interacts with something. We added a sounds for jumping, attacking, when exiting a map and when collecting a coin.
To include bacground music we would have needed about 2kb more space in the flash memory since it takes alot of code to make this work. A solution would have been to use a shorter music file in the GD2 file and looping it, but we agreed that it would be annoying and steered clear of it.

Chapter 6

Hardware

During the project we have worked with two Arduino types. Duemilanove and Leonardo clone. The clone was more powerful and therefore was our main used board. The clone is called OLIMEXINO-32U4.

6.0.3 Specications

Arduino

The specs of the Arduino boards vary very much of each other. The OLIMEXINO is definitely better.

Board name	Microcontroller	Operating Voltage	Flash Memory	Clock Speed	Input Power	SRAM	EEPROM
OLIMEXINO-32U4	ATmega32U4	3.3V / 5V	32KB	16 MHz	7-12VDC	2.5 KB	1kb
Duemilanove	ATmega168	5V	16 KB	16 MHz	7-12V	1 KB	512b

Table 6.1: Specifications of the boards

EEPROM

Electrically Erasable Programmable Read-Only Memory Is non-volatile memory which is used in computers/electronic devices to store data when power is removed. Bytes in EEPROM can be read erased and rewritten. One byte in the EEPROM can only hold a value inbetween 0 and 255.

Gameduino2

The specificatins of the Gameduino2[1] shield.

- Video output is 480x272 pixels in 24-bit color.
- OpenGL-style command set.
- Up to 2000 sprites, any size.
- 256 Kbytes of video RAM.
- Smooth sprite rotate and zoom with bilinear filtering.
- Smooth circle and line drawing in hardware - 16x antialiased.
- JPEG loading in hardware.
- Built-in rendering of gradients, text, dials and buttons.

6.0.4 Input

We heard about the nunchuk capability before we got the Arduinos in our hands, we thought it would be fun and ordered[2] the adapters just after signing to this project. The adapters where needed as the nunchuk has to be connected to the Arduino physically.
None of us had much experience in soldering, but our supervisor was happy to help us in that matter. We got head sockets soldered underneath the boards, which made it easy to connect the adapter using pin cables.

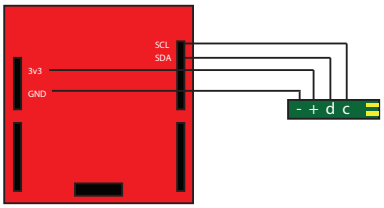

Figure 6.1: Hardware connection of an nunchuk

It is possible to use both the Gameduino 2 and Wii nunchuk at the same time, even though they use same slots. Gameduino 2 uses an ISP interface, while the Wii nunchuk uses an I2C interface.

I2C

The I2C is a hardware wire interface, which is used by the Wii nunchuk adapter. This bus interface allows easy communication between components and only requires two bus lines. These lines are both bidirectional. These bus lines are called SCL (Serial Clock Line) and SDA (Serial Data Line).

[1] http://excamera.com/sphinx/gameduino2/
[2] http://www.miniinthebox.com/da/arduino-kompatibel-wii-wiichuck-nunchuck-adapter_p903451.html

Chapter 7

Implementation and detailed design

Like Java's main function, Arduino has two special functions: `setup` and `loop`. The former is called once on program start, while the latter is repeatably called until the Arduino is turned off.[1]
This isn't a very optimized structure. It forces us to use global variables if we want to reuse anything from setup in loop. Our solution was to leave `loop` empty and put an endless loop in `setup` instead. All of our variables could become local with this change, greatly reducing code size.

7.1 Components

Sketch files, c++/c language, includes, standard libraries

7.2 Structure

A platforming game would need at least a simple physics engine, supporting momentum and acceleration. This makes movement feel a lot more natural. Only actors would need to be affected by the physics engine, but we extended it to allow all props - this was both an optimization and an additional feature in the game. We discuss these implications in the implementation chapter.
Class diagram

[1] At least, we haven't discovered a way to terminate a sketch.

7.3 Setup

Setup is where we instantiate most local variables, our hero, units, props and lists of our units. We also generate our random seed and load our assets, the GD prerequisites and a millisecond counter.

7.4 Loop

The loop first sets the millisecond counter then it gets data from the nunchuk update. Afterwards it updates all units AI, where they decide what to do in this frame. The hero is part of this loop and updated as if he had AI, but instead converts player input(that we got from the nunchuk update) into actions. After this, all attacks generated from this AI update is executed. This involves all props, where units are checked if they die and coins are checked if they are collected. Finally the physics is updated, and all props move according to their internal velocities. Minotaurs may update their AI in case a collision function is called.
Attacks are cleared, sound for the attack is played and it checks the hero health is zero. If it is we restart the game and save the score if its a highscore.

7.5 Logic

The units needs to receive map data and send actions. This is the `Logic` class' function. It contains numerous methods to calculate a units' surroundings and also handles collision, physics engine, coin collection and attacks.
Props can only collide with the map geometry, not with each other. It was not a priority to collide props with each other. Enemies should be able to pass each other and the hero would be damaged when colliding with them in any case. Collision is calculated in straight lines, and only at one axis at a time. This makes long diagonal movement inaccurate, since it is calculated as if the prop moved first horizontally and then vertically. A better collision detection was a possibility, but eventually not prioritized. Collisions are calculated according to rectangular hitboxes.
After these simplifications, the collision algorithm is very simple. Given a prop and a distance, it checks each tile in order which the prop will pass through according to its hitbox. If any of the tiles are solid, the prop only travels up to the solid tile and the algorithm terminates. When updating the physics engine, either the `collisionX` or `collisionY` functions are called, used in AI for units or bounce in coins.
A linked list of attacks between frames is kept. Units add their attacks to the list during AI updates and the list is cleared after the attacks have been executed. Executions goes through each prop and checks whether the attack hits or not, calling the `hit` function in case it has. Attacks are a separate object, containing damage, push force, the owner and

the area. The attacks are only instantiated once at unit instantiation, which only manipulates the area when reusing the attack, thus optimizing on computing time, though costing memory.
There is a circular reference, since units require logic which requires scene which in turn requires units. This resulted in forward declarations in scene which is a bit inelegant. It was difficult to design a structure which did not have any circles, since the actors act upon the scene, and the scene returns data to the actor.

7.6 Units

Units are a subclass of prop, reusing all physics related functions and fields.

7.6.1 Minotaurs

The only implemented enemy is the minotaur from Greek myth. His behaviors range from wandering, charging, hunting and dead, which also has alternative settings while he is in air. Initially he wanders in some direction until he comes across a gap or hits a wall, where he turns around. The gap check is simply whether the space immediately below the front of his hitbox is empty or not, while `collisionX` function is only called when he wanders horizontally into a wall, making the wall check easy. hunting
Currently the only type of enemy is the minotaur, though the current framework was built to contain multiple types.

7.6.2 Hero

Our hero has its own class where we define the movements, the hitbox and the animation handler.

Our hero has two speeds, walking and running. By using values from the nunchuk library we can determine whatever our hero should stay idle, walk or run. As an example, if the difference between `analogX` and rest value `NUNCHUK_REST_X` is 50 or more, our hero will run to the direction specified in the class. When this happens the acceleration and speed are set to the specified constants in the header file. A similar method is also used for ducking, if `analogY` is lower than 45, our hero will duck. In this case, the hitbox will change height and y-value. The y-value has to be changed because hitboxes are defined at the top left corner. The hitbox is used to detect collisions - see subsection Logic for more detail.

Attacking and jumping are far simpler as they only check the variables `zButton` and `cButton` for a `true` value. They can though only be executed again once their local variables `_isAttacking` and `_isJumping` are false. This will prevent errors as jumping

continuously. We did also ensure that our hero can't duck and attack at the same time. He is though able to jump and attack in the air. When attacking in the air, the handler uses the attack animation, it changes back to jump handler as soon as the button is released.

7.7 Scenes

Scenes are the objects Containing the map data and all props within. The map is a simple two-dimensional array. The element at two given indexes corresponds to a tile, which has coordinates equal to the two given indexes times the size of a tile. This means world- can easily be converted to tile coordinates or vice versa.
In addition to the map, the scene also contain all props, which is the hero, minotaurs and coins. It stores this in two places: in dynamic linked lists, and in arrays. The former is for dynamic removal of the props, when the coins are collected or the minotaurs are killed. These are used when updating gameplay, to make sure that unused props are not updated, increasing framerate. The latter is for storing all available props , used at map generation. The arrays have a couple of advantages. First they create all used props of each type initially, reusing the same minotaurs and coins in each map. This shortens time needed to allocate and deallocate memory. Additionally, if the amount of props is initially within Arduino's memory bounds, then the maps will never cause the Arduino to run out of memory, since it never allocates more. The obvious drawbacks is that we are limited in the amount of props we need, and that removing any props during play does not increase available memory. The second drawback isn't that much of a problem though, since very little is needed during play.
Currently, because of an implementation bug, the props are dynamically allocated, even though they shouldn't need be.

7.7.1 Generator

Map generation is executed at setup and whenever the game ends (either by player death or win). The method `newScene` generates a new map with matching points at the entrance and exits. Arduino's programming language does not support returning more than one value, so the method manipulates given pointers instead like in C programming. The new map data overwrites the old one to save time and space, so the given `scene` argument is a pointer to the old scene. Reusing it saves us from instantiating a new one and deallocating the old one. We ar not interested in reusing the actual map data, since the player cannot revisit prior levels.
The actual algorithm is in three parts: clearing, modulation and generation. First it clears the old map data, setting all tiles to `NONE`, to make sure nothing is left over from the old map. This may be a bit expensive on the processing time considering it is superfluous if the generator works correctly. The reason is a design choice which will be apparent when we reach the generation.

Modulation in this case means separating the map into *modules*[2]. This is where the layout of the map is decided. A module is a small map in itself, in our case a 5x5 map of tiles. These have been designed by hand and hard-coded into the generator. Every map is construed of a grid of modules, in our case 4x4. The modules are differentiated by which sides one can access it from. By this we mean the player can traverse from and to this module from the given sides. The types are left-right (corridor), left-right-up (T-up), left-right-down (T-down), all (cross) and none (closed). A module in the category left-right is guaranteed to have an exit left and right, and may have an exit up or down.
The algorithm first instantiates a grid of empty modules, and randomly assigns one of the bottom modules to be a corridor and the entrance. This is the beginning of the solution path, which guarantees that the map can be completed by the player. From then on it picks a direction, left or right, randomly. From then on the algorithm randomly either moves according to its direction or up. When moving sideways the newly visited grid space is assigned to be a corridor. If it hits the edge of the map it moves upwards and changes direction instead. Whenever the solution path moves upwards, the algorithm has to change the space it is in first. If it is in a corridor tile, it changes it to a T-up, and if it is a T-down it changes it to a cross module. Both of these are the same as their predecessors, but with a guaranteed top-side exit. The newly visited module is assigned a T-down module. The algorithm picks a new direction at random (only if it is not at an edge.) and can start the over again. When it attempts to move upwards while at the top, it instead places the exit in the current tile and terminates. All unvisited grid spaces are assigned closed modules, and are not part of the solution path. Thus we have reached a map which has a guaranteed solution.
Lastly the program generates the map. This step reads each module in the newly generated grid, randomly picks a module of the specified type, and fills it into the actual map. If it is currently in an entrance or exit room, it places the corresponding door. It changes the original given entrance and exit pointers to point at the now created door.
Every module overlap with a single row or column with all surrounding modules. Overlapping follows a priority of tiles, where the algorithm determines which tile from the modules is used. Platforms are placed before empty tiles, and solid tiles are placed before platforms. This has several benefits. Firstly the maps are more unique since pairs of modules also differ, and makes the seams of modules harder to notice. Secondly, it ensures that upward exits are easier guaranteed since platforms can be placed closer to the upper floors. This is the reason we need to clear the map prior to generation: the old tiles would disrupt this priority, since the algorithm would not be able to discern what is old and what is new during generation.
Old version (nondeterministic time)

[2]As opposed to vary the pitch in a voice

7.7.2 Highscore/?Randomseed? EEPROM

Using EEPROM we implemented two functions $EEpromReadInt$ and $EEpromWriteInt$ which could save unsigned integers on the "hard drive" of the arduino. The functions work by bitshifting the integer with 0 and 8 and then save the two bitshifted values in the EEPROM. We

7.8 Gameduino 2

7.8.1 Graphics

7.8.2 Assets

7.8.3 Sound

The Gameduino 2 requires the sound files to be converted using the asset converter. Before including the sounds in the GD2 file, we first convert them to an acceptable format. The assets converter only accepts files with following properties.

- .wav format.
- Audio channel: mono.
- Bit resolution 16.

The conversion process also includes reducing the sample rate of the sound. By doing so, the file will not take as much space. Sometimes the sounds would get ruined by reducing too much, so we had to find a balance between quality and size.

7.9 Input

Before implementing the nunchuk. We checked other alternatives. Both Arduino and the Gameduino 2 shield gives us opportunities to control the game. Arduino can communicate with a pc and get data when a key is pressed on the keyboard. The Gameduino 2 is equipped with a touch screen and a accelerometer. We could have used one of these to get input, but none of them gives a natural way to play a 'platformer' on a Gameduino 2. The pc solution feels not natural as the keyboard and the Gameduino 2 screen usually are not in front of each other. The touch screen is not as responsive as we wanted, and it would also be annoying as the fingers will get in the way. The accelerometer is just wrong in all way, it is hard to control, as you always has to 'guess' how to hold the device. The

problem with fingers getting in the way appears also here. The best option was using an external controller - a Wii nunhcuk.

7.10 Optimization

The biggest challenge has been the code size due to the low capacity of the flash memory. The flash memory has a capacity of 32Kb. The first 4.242Kb is reserved for the bootloader and we are limited to go no further than 28.672Kb all inclusive. So we are actually only allowed to upload 24.430Kb of our own code. It may sound fair comparing to what have been achieved with old game consoles, but it became very quick a headache.
Every time we compiled our code, we felt like the code was growing exponentially and we reached the limit quicker than we expected, the intern libraries and our code generally took way too much space. Please see Figure 11.1 for a history of our code size.

From the start, we wrote the code with optimization in mind, but also tried to keep the code maintainable. When we reached the limit, we had to optimize it further, not just to create space for existing code, but also for further additions to the game. Our first step to solve this problem, we analyzed our code and found the main reasons for this occasion, which are described below.

Inefficient datatypes

A good place to start was the datatypes. Converting the bigger datatypes to some smaller ones was a easy optimization. Mostly, it was integers that was converted to data types like `char`, `byte` and `word`. Which `char` is capably of encoding numbers from -128 to 127. While `bytes` is a 8-bit unsigned number, from 0 to 255 and `word` is basically an unsigned 16-bit number, from 0 to 65535. Notice that `byte` is the unsigned version of `char`. More details in arduinos site[3].

Unused functions

Global to local

A global variable ensures that all the functions in the class has access to the variable. But there are downsides to this, it uses more space than local variables and it also increases the chances of changing the variable value by other functions without intentions. So we changed the global variables into local variables by declaring and initializing them in the functions they are going to be used. So it is a matter of declaring them in the right scopes.

[3]http://arduino.cc/en/Reference/HomePage

Code Verbosity

Repeated statements

Repeated statements - well it speaks for itself. Repeating something that has been calculated before is just a waste of code space. Simplifying the code in this manner is sometimes not easy, it requires that you think creative. It often requires you to think - is this needed? It may not always be obvious. Often it is about finding a shortcut to some calculation and taking advantage of already existing results.

Accessor methods

We found out that these functions inefficient compared to public variables. So we deleted these functions and made the variables public.

Dependencies

Libraries

The libraries are the most space allocating part of code. Both our external and our internal libraries fill much of the code. The internal libraries are inclusions of our own written code, everything from logic to units. The external libraries contains which are necessary to communicate with the EEPROM, Gameduino 2 and Wii nunchuk.

7.10.1 Code Size bloat

Bootloader GD2 includes

7.10.2 Further optimization

Custom written or shortened extern libraries

7.10.3 Inconsistencies

additional code suddenly uses less space.

Chapter 8

Testing and performance analysis

Present test methodology as well as results in this section. In addition, if performance analysis of the system is interesting, present it here as well.

A few screenshots of the program can be included here as well.

Chapter 9

Discussion

Discuss the problems, challenges and solutions encountered. What did you learn? What gave you troubles? Interesting future extensions and improvements of the system can be discussed here as well.

9.1 Meeting Requirements

9.1.1 Cut content

Items, merchant and levels increasing difficulty? Multiple enemies?

9.1.2 Additional content

More or less advanced physics engine. Coin physics.

Chapter 10

Conclusion

Summarize the main results. This section should make sense even if the reader has only read the introduction.

References

10.0.3 Links

I2C Interface: `http://en.wikipedia.org/wiki/I%C2B2C`

Chapter 11

Appendices

11.0.4 Optimization

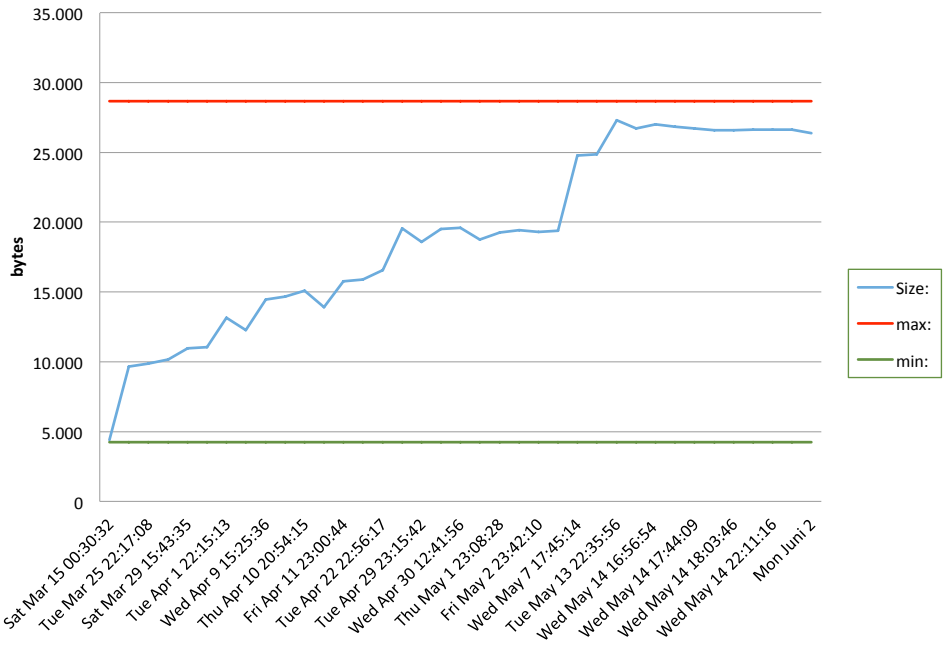

Figure 11.1: A history of the code size

11.0.5 Requirements

Figure 11.2: The waterfall timeplan of our project