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

Katabasis
Group 6.1

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

June 16, 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 highlit. It 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.
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 system 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.

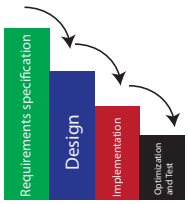

Figure 4.1: An overview of the waterfall phases.

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. Or so we thought we gave it to our supervisor who had a look. He gave it back the day after where it worked but then 5 minutes later it simply died again. We have had this issue with the arduino since then it works and then it doesnt. Were not sure whats causing it and have not been able to find anything explaining it. This slowed us down somewhat since we now only had one arduino leonardo and a arduino duemilanove.

arduino duemilanove.

We ran out of flash memory on the duemilanove since it only has 16kb flash memory, 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.

4.2.2 Nunchuck

Nunchuck compatability.

Chapter 5

Overall design

5.1 Components

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

5.1.1 Structure

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. With Scenes

5.2 Media

Gameduino...

5.2.1 Assets

Assets...

5.2.2 Sound

Sounds...

Chapter 6

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.

6.1 Setup

Precalculations

6.2 Loop

The loops first manages all props and then checks whether the game is over. First 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 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 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.
dTime...

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

6.3 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.

6.4 Actors

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

6.4.1 Enemies

AI; hunting
Currently the only type of enemy is the minotaur, though the current framework was built to contain multiple types.

6.4.2 Hero

Movement

6.5 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.

6.5.1 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

6.6 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 verbose

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

Accessor methods are getters and setters functions, that make variables available to other functions. These functions has to be declared public in the header file on order to be visible to the other files. But we found out that these functions are more 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.

6.6.1 Code Size bloat

Bootloader GD2 includes

6.6.2 Further optimization

Custom written or shortened extern libraries

6.6.3 Inconsistencies

6.7 Game loop

Essentially the game is one long loop, which ends whenever the game does. Each iteration executes the gamelogic per frame, such as moving the player and monsters around. Gameloops vary from simple while(true) loops to dynamic separate gameloops handling separate calculations (such as separating rendering from gameplay).For this we only needed a simple game loop, updating the world in each iteration. To ensure a consistent gameplay experience, we needed to know how much time passes betweenframe updates. Without this, the game would run faster or slower on different processors or inconsistently in different ingame events, throwing off the players sense of timing. For example, the game would be equally playable with either 30 or 60 frames per second. If the difference isnt incorporated in the gamelogic though,the game would run twice as fast with 60 frames per second! Our implementation of a frames per second counter, is simply calculating how many frames was calculated the last frame, and then assuming the next frame has the same amount. This is a pretty simple solution, which could easily be expanded upon if needed. The obvious criticism of this method, is that the frame calculations between each second are not accounted for, and that the framerate is essentially a second behind any framerate changes.

6.7.1 Movement

The movement of our hero will mainly be right and left. He will have also have the ability to jump. Beside the basics movements, our hero will also have fight moves. The fight moves contains moves like sword swinging and archery.

6.8 Input

The Gameduino 2 comes with a touch screen and accelerometer. These modules gives us an opportunity to interact with the game in many ways. We could use the accelerometer or the screen to move the hero. We dont have much experience with the screen and its capabilities. Therefore another option as backup is preferable.
Another option will be using an external controller. The Wii nunchuck is popular and is very suitable for this game. Using the buttons we game can be controlled as seen in the figure below.

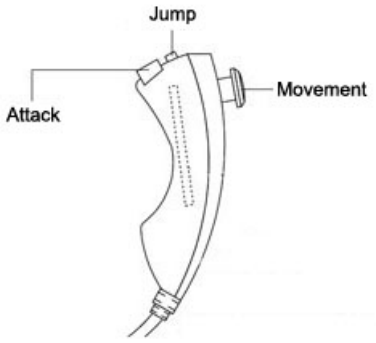

Figure 6.1: Button specifications

Chapter 7

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.

7.0.1 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
OLIMEXINO-32U4	ATMEGA32U4	3.3V / 5V	32KB	16 MHz	7-12VDC	2.5 KB
Duemilanove	ATmega168	5V	16 KB	16 MHz	7-12V	1 KB

Table 7.1: Specifications of the boards

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.

[1]http://excamera.com/sphinx/gameduino2/

- 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.

7.0.2 input

The Wii nunchuk is one of the ways to control the game, aside from the touchscreen. The nunchuk has to be connected to the Arduino by hardware. Even though they use same slots in the board, it is possible to use both the Gameduino 2 and Wii nunchuk at the same time because they use different hardware interfaces. The Gameduino uses an ISP interface, while the Wii nunchuk uses an I2C interface. The Gameduino requires 5 volt to work, which forces us may shorten the lifetime of the nunchuk - as it normally operates at 3.3 volt, but it should not be a big concern.

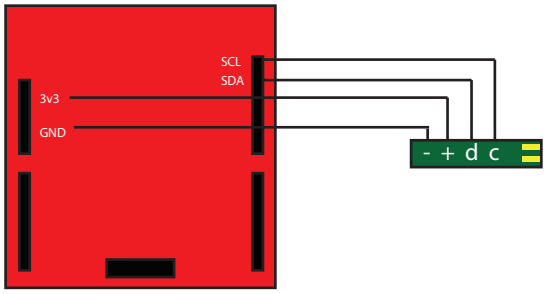

Figure 7.1: Hardware connection of an nunchuk

7.0.3 I2C

The I2C is the 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).

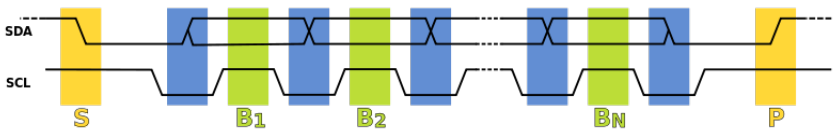

Figure 7.2: Hardware connection of an nunchuk

1. Data trans fer is initiated with a START bit (S) signaled by SDA being pulled low while SCL stays high.

2. SDA sets the 1st data bit level while keeping SCL low (during blue bar time.) and the data is sampled (received) when SCL rises (green).

3. When the transfer is complete, a STOP bit (P) is sent by releasing the data line to allow it to be pulled high while SCL is kept high continuously.

4. In order to avoid false marker detection, the level on SDA is changed on the SCL falling edge and is sampled and captured on the rising edge of SCL.

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

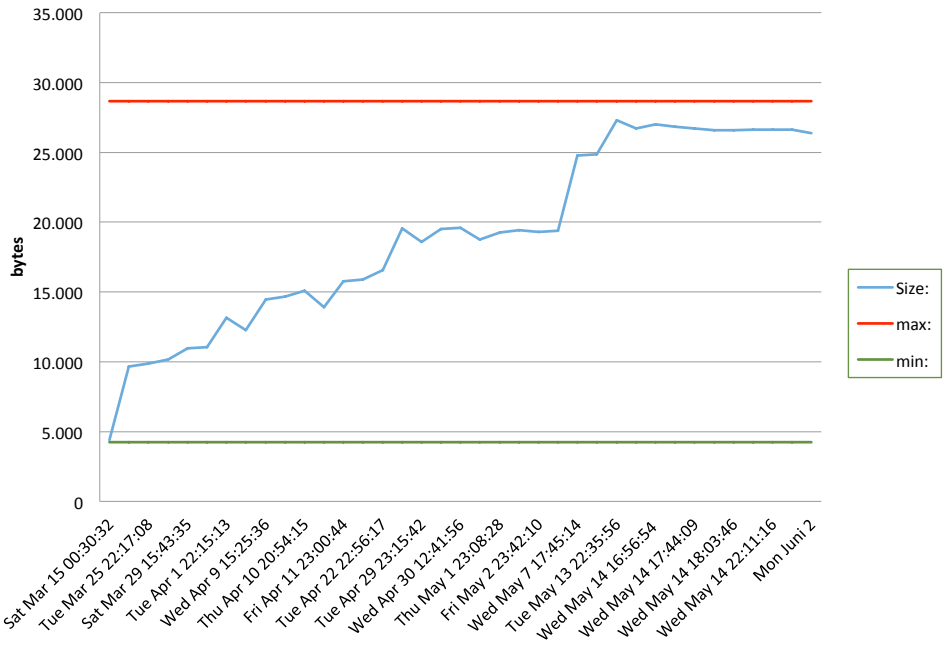

Figure 11.1: A history of the code size

11.0.5 Requirements

Figure 11.2: The waterfall timeplan of our project