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 11, 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 delete, remove, size, update, speed, move and clear.
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 The game play

There are several things that a good game should include. The most important and essential are:

1. Story
2. Great design
3. Easy to understand
4. Easy control
5. Sound effects

Even though our project mainly focuses on implementing advanced algorithms, the gameplay can not be ignored. Without a meaningful and understandable game, the advanced features will be not as recognizable.

3.0.2 The algorithms

In order to implement the enemy motions and map generation, we'll have to create our own algorithms.

3.0.3 Timeplan

Our timeplan follows an agile development system called waterfall. 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.

Project management

When the waterfall ends and we still have time we will go back to the start and check for new requirements and the whole waterfall process starts again. The report is both written seamlessly and separately in the end of the project, therefore it does not has to be illustrated, as it will create confusion.

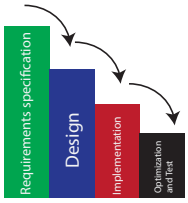

Figure 3.1: An overview of the waterfall time plan.

Tasks management

When providing the tasks we use the timeboxing method. This method increases the productivity significantly. Timeboxing works by breaking a big task into smaller tasks with better manageable time frames. It kinda works like a stopwatch. The following table shows the timeboxes we have created during the project.

8-10	**11-13**	**14-15**	**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

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

Table 3.1: The timeboxes are seperated per week

Process

Chapter 4

Overall design

Describe the overall structure of the system, the different components of the system and interfaces between these.

4.1 Components

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

4.1.1 Structure

Unit ¡-¿ Logic
Scenes

4.2 Media

Gameduino...

4.2.1 Assets

Assets...

4.2.2 Sound

Sounds...

Chapter 5

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.

5.1 Setup

Precalculations

5.2 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.

5.3 Actors

Units are a subclass of prop, reusing all physics related functions and fields. Furthermore logic only needs to calculate with props.

5.3.1 Hero

Movement

5.3.2 Enemies

Logic communication
AI
Types

5.4 Logic

Collision
AI
Attacks
Circular references
Score

5.5 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.

5.5.1 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

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

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)

5.6 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, where we are only allowed to upload 24.430Kb of our own code.
The Arduino IDE uses the first 4.242Kb for the bootloader and we are limited to go no further than 28.672Kb all inclusive.

Programming our code, we felt that the code was growing exponentially and we reached the limit quicker than we expected, the intern libraries and our code generally was not optimized enough. Please see Figure 10.1 for a history of our code size.

Our first step to solve this problem, we analyzed our code and found the main reasons for this occasion:

1. Inefficient datatypes.
2. Unused functions
3. Global Variable instead of local.
4. Code verbose.

5. Repeated statements.
6. Getters/Setters.
7. Dependencies.
8. Libraries.

5.6.1 Code Size bloat

Bootloader GD2 includes

5.6.2 Further optimization

Custom written or shortened extern libraries

5.7 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.

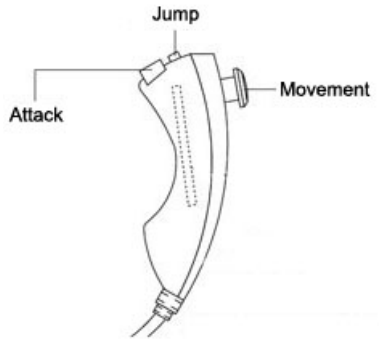

Figure 5.1: Button specifications

Chapter 6

Hardware

Here are the specifications of the Arduinos we have:

bleh bleh

6.0.1 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.

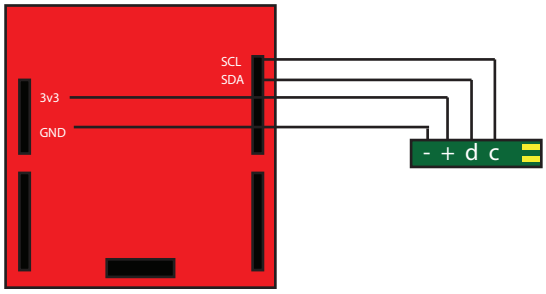

Figure 6.1: Hardware connection of an nunchuk

6.0.2 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).

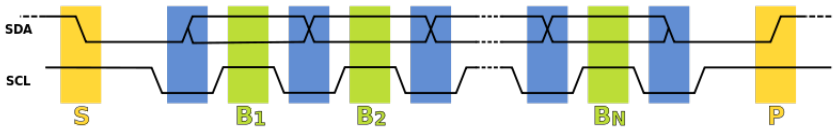

Figure 6.2: Hardware connection of an nunchuk

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 7

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 8

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.

8.1 Meeting Requirements

8.1.1 Cut content

Items, merchant and levels increasing difficulty? Multiple enemies?

8.1.2 Additional content

More or less advanced physics engine. Coin physics.

Chapter 9

Conclusion

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

References

9.0.3 Links

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

Chapter 10

Appendices

10.0.4 Optimization

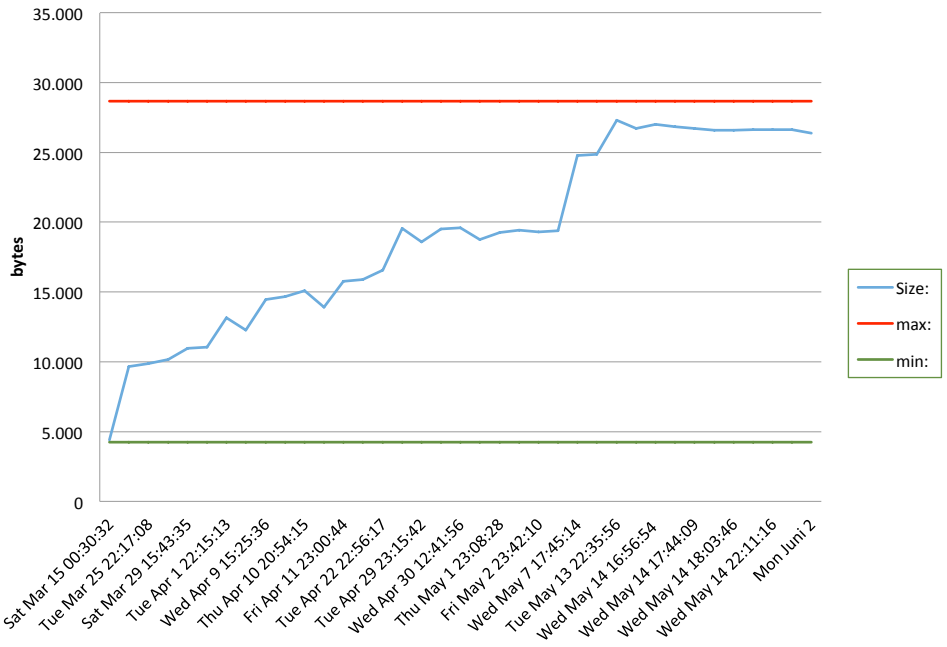

Figure 10.1: A history of the code size