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 and Requirements specification

Even before we got the Arduino, we knew that it could cause trouble, especially because of the limited power and space capacity. Another thing was its IDE. The Arduino IDE has its own language (based on c/c++), which was not the most sophisticated language according to what we heard. We did neither have any kind of emulator, which we definitely would have used, before getting the units in our hands.

We were also sceptical about what Gameduino 2 was capable of. The examples which were shared through their sites did not look impressing. Only one of them looked like an advanced game.

2.1 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 3

Requirements specification

3.1 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

Development started at DATE. The three week period was from DATE to DATE.

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.

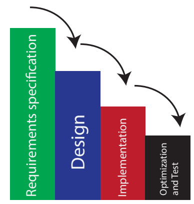


Figure 4.1: An overview of the waterfall phases.

Time Boxing

We were convinced 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	Collision detection	Level generation	Graphics
Class design	Enemies	Player	Level generation
Game Design	Input		Sprites
week 18	**week 19**	**week 20**	**week 22**
Endgame	Attack	Animation	Optimization
Level generation	Level generation	Optimization	
	Optimization	Sound	
		Sprites	
week 23	**week 24**	**week 25**	
Attacks	AI	Bug Fixes	
Coins	High score	Code Polish	
Optimization	Optimization	Optimization	

Table 4.1: The timeplan in timebox format.

4.2 Development Issues

We received the Arduino three weeks after project start, which was a major setback in implementation. We could only design the program structure and time-plans in this time. This skewed our planned agile development. Instead of dynamically designing and implementing additional features, we had to make the most of our time and design for three weeks straight, which was followed by weeks of implementation the designed structure and features. Usually in agile processes, you make sure to have a working product at the end of sprints. This was not compatible with our designed framework, since we had taken a lot more features into consideration than possible in a single sprint. It was one very large sprint, more akin to the waterfall process. We could have incrementally implemented the framework after receiving the Arduino, but it would in the end have been slower and more tedious. We had the designs, all we had to do was implement it. In the end we made up for the loss in implementation time by cutting corners in the process and designing a lot ahead.

Later on one of the Arduinos burned out. It is not clear what went wrong, but it rarely worked. At this time we only had another Leonardo and the Duemilanove, which slowed us down quite a bit. Only half of us could program each week. A few weeks later we hit the maximum code size on the Duemilanove, and were left with only a single working Arduino. Development became too slow, until one of us purchased a second Leonardo.

General problems with the Arduinos were common. We have also had some issues when uploading code to the Arduino. The screen would be completely black even though the IDE stated the upload was complete. Usually the fix was uploading an example file from the Arduino library that prints text to the screen, usually `HelloWorld.ino`. In other cases a different USB port would solve it, probably a sign of incorrect driver installation. Resetting the Arduino during uploading would also sometimes snap it out of it. Other problems were related to the SD card in the Gameduino 2, where we could not upload programs while the SD card was inserted. Usually formatting the SD card would work, but often only on a Mac for unknown reasons.

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.

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.

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.

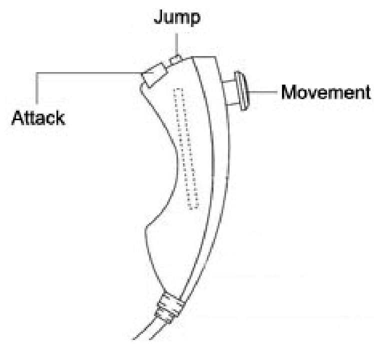


Figure 5.1: Button assignments

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.

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...

Assets

Assets...

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.

6.1 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	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.

6.2 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.

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).

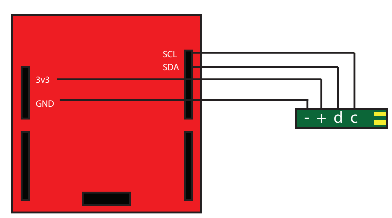


Figure 6.1: Hardware connection of an nunchuk

[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

7.1 Components

Sketch files, c++/c language, includes, standard libraries Arduino sketch is the Arduino IDE, we have worked with it several months its built upon IDE form the Processing programming language. We have had a few quams with it. Firstly the syntax highlighting is not completely correct, second the structure of the ino file and all its cpp/h files where all tabs need to be open is just plain confusing.

Arduino programs are written in C or C++. The Arduino IDE comes with a software library called "Wiring" from the original Wiring project, which makes many common input/output operations much easier. Users only need define two functions to make a runnable cyclic executive program:

1. setup():a function that runs when board is powered on.
2. loop():a function called repeatedly until the board powers off

Nunchuk library The inputs are recieved using the `ArduinoNunchuk`[1] library. It uses the `wire.h` library that is included in the Arduino IDE. Using the `wire.h`library makes it possible to communicate with the nunchuk using the I2C interface.
The library includes following methods, which are updated every time `nunchuk.update();` is called:

- nunchuk.analogY
- nunchuk.analogY
- nunchuk.accelX
- nunchuk.accelY
- nunchuk.accelZ
- nunchuk.zButton
- nunchuk.cButton

[1]http://www.gabrielbianconi.com/arduinonunchuk/

Linked list library is also one we found it is simply an implementation of linked lists which we use to store our units, hero and our coins.

GD2 libary is needed for the Here are the functions that we use from the GD2 library:

- GD.cmd_text()
- GD.cmd_number()
- GD.Begin()
- GD.cmd_translate()
- GD.cmd_scale()
- GD.cmd_setmatrix()
- GD.safeload()
- GD.cell()
- GD.BitmapHandle()
- GD.sample()
- GD.Clear()
- GD.ClearColorRBG()
- GD.swap()
- GD.ColorRBG()
- GD.Vertexii()
- GD.Vertexii()

7.2 Structure

A platforming game would need at least a simple physics engine, supporting momentum and acceleration. Player movement feels more natural, and allows us to make attacks push units. 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

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 whether the hero's 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.

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.

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.

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

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

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.

Highscore & Randomseed

Using EEPROM we implemented two functions $EEpromReadInt$ and $EEpromWriteInt$ that takes use of the EEPROM library. These functions save unsigned integers to the EEPROM. They work by bitshifting the integer with 0 and 8 and then save the two bitshifted values in the EEPROM. To extract form the boards EEPROM you we again use the EEPROM libary in $EEpromReadInt$ we grab the bitshifted values form their addresses in the EEPROM and then bitshift them with the respective number from before. We add them together and then return the integer this gives.
We also use this for our random seeds that generate our level when the board is powered on. When the board is powered on we find the random seed on the address and apply it to the leve lgeneration. We then generate a new random unsigned integer and we save it on the same address.

7.8 Gameduino 2

Graphics

Rendering

The Gameduino 2 consists of an essential graphics processing unit (GPU) with 256 kilobytes of RAM. With it is the ability to draw virtually anything on the screen.

Bitmaps

With it there is support for drawing images, called 'bitmaps'. With the exception of the background and text, everything the player witnesses in our game consists of these bitmaps. Since images arent uploaded along with the code to the Arduino, we store them on a microSD card which conveniently can be read in real time from the Gameduino 2. The simplest way to read a single image off the card would be to directly load it to the GPU with a command in the code. Each image loaded in would be assigned to a specific address and could thereafter be drawn with the images top left corner at a given x-y coordinate. However, this method wouldnt be able to fulfil the necessary requirements of what we had planned. This is where the Gameduino 2 asset converter comes in.

Assets

The asset converter basically takes multiple media files, including images and sounds, and compiles them into a single asset file with the extension '.gd2'. As the abbreviation suggests, this file is specifically designed for the Gameduino 2 to read and gets placed on the SD card. Along with the asset file, a header file is generated to be included alongside the code in the associated project. The header file automatically defines constants associated with each image. There are four constants: handle, width, height, and cells. An images handle is its index in the array of images provided, e.g., the first image would have the handle, 0. The width and height simply state the dimensions of the image in pixels. The 'cells' variable states how many images are included in the handle. Essentially we used handles with multiple cells (images) instead of a single image to easily maintain each animation. Here we can see the process of creating simple walking and attacking animations for our hero. Once the header file is included in a project, its constants are ready to be called from the code and used to project animations onto the screen.

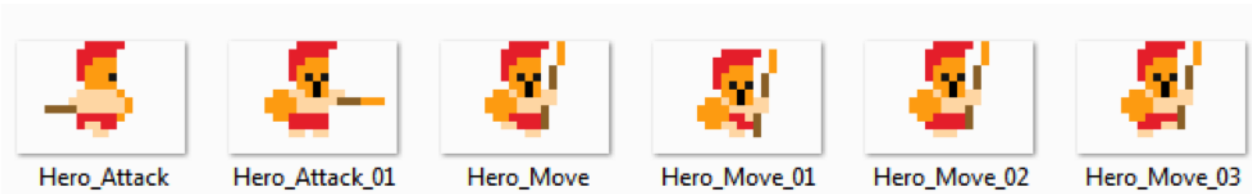
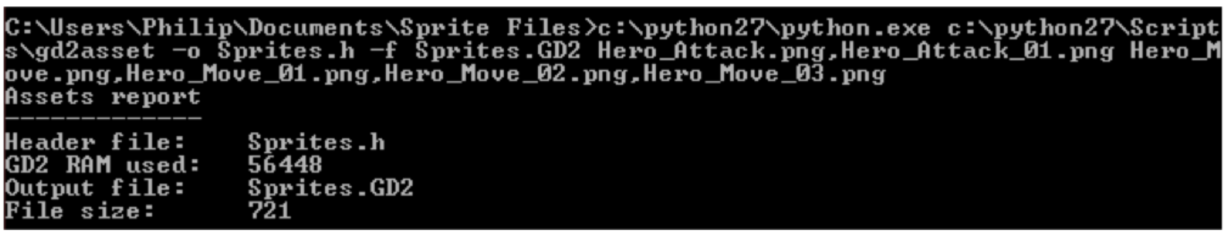


Figure 7.1: Hero attack sprites



```
C:\Users\Philip\Documents\Sprite Files>c:\python27\python.exe c:\python27\Script
s\gd2asset -o Sprites.h -f Sprites.GD2 Hero_Attack.png,Hero_Attack_01.png Hero_M
ove.png,Hero_Move_01.png,Hero_Move_02.png,Hero_Move_03.png
Assets report
-------------
Header file:    Sprites.h
GD2 RAM used:   56448
Output file:    Sprites.GD2
File size:      721
```

Figure 7.2: Asset conversion in command line

Animation

There are many ways to make sure an object in the game knows which handle it should use as well as which cell within the animation to display. The example in the Gameduino 2 reference utilizes both a sprite's x-coordinate and bit shifting as the object moves across the screen. This method however, is very primitive and specifically tailored to sprites moving from left to right. The way our program cycles through the cells of an animation is through time.

Every object in the game that includes animation is a prop. That is to say the hero, minotaurs, and coins are all props. The prop class includes two functions directly related to animating: newHandle and updateAnimation. newHandle includes three parameters that get called from the specific objects class every update. It starts by checking if it should change anything about the animation by first checking if `_animLock` boolean is true. `_animLock` assures that the animation cant change until the current animation has gone through all its cells. We use this boolean when the hero attacks because he should finish an attack before reverting to reverting back to whichever movement animation hes in. The function also checks if the new handle or frame rate are the same as the current ones. If the conditions are met, the three values are replaced with those associated with a new animation and current cell to be displayed is manually set to the first one in the sequence.

The function, updateAnimation, receives the parameter, dTime, from the main class every update. The `_dtime` variable keeps track of the number of milliseconds that have passed since the last update, or run through of all the code. Its used to keep the time interval between cells consistent with the time it takes to update the whole program. Every animation has a manually set frame rate which determines how many milliseconds should pass before changing to the next cell. The variable, `_aniTime`, simply adds the value of

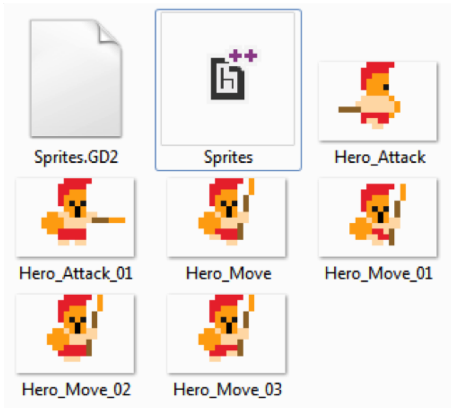


Figure 7.3: The sprites and the output files

Sprites.h
```
#define LOAD_ASSETS()  GD.safeload("Sprites.GD2");
#define HERO_ATTACK_HANDLE 0
#define HERO_ATTACK_WIDTH 84
#define HERO_ATTACK_HEIGHT 56
#define HERO_ATTACK_CELLS 2
#define HERO_MOVE_HANDLE 1
#define HERO_MOVE_WIDTH 84
#define HERO_MOVE_HEIGHT 56
#define HERO_MOVE_CELLS 4
#define ASSETS_END 56448UL
static const shape_t HERO_ATTACK_SHAPE = {0, 84, 56, 0};
static const shape_t HERO_MOVE_SHAPE = {1, 84, 56, 0};
```

Figure 7.4: The output from asset conversion

`_dTime` to itself every update until the compiled number of milliseconds passed surpasses that of the animations frame rate. When this happens, its time to move on to the next cell in the animation. If however, its the last cell, then it naturally reverts back to the first. One condition on resetting the cycle however that is that the `_aniStop` boolean isnt used. This boolean is only used when an animation shouldnt cycle back to its first cell. We use it primarily for death animations because when a unit dies it should stay dead. If the previously mentioned lock boolean is being used when a animation sequence resets, it isnt required thereafter and set back to false. In any case, when a cell changes, the value of `_aniTime` is set back by the value of the animations frame rate so that it can once again accumulate time until it reaches the required frame rate value again for the next cell.

Drawing Sprites

When the correct image to display is determined, there is still a matter of how to draw it and where exactly to place it. Rather than creating mirror image files were able to use the Gameduino functions to flip an image horizontally and still keep it at the same position. We intentionally made every sprite face to the right in the original images so wed only have to flip them if theyre supposed to face left. As mentioned, everything that gets drawn with the Gameduino gets prepared and manipulated beforehand. In this case, the image gets flipped, drawn, then flipped back again in case a sprite is updated thereafter to face to the right again.

In terms of coordinate placement, the image would under usual circumstances share the same position and dimensions as the hitbox its registered to. However, our units have appendages/weapons/armor etc. which shouldnt be included. Essentially we created sprites wherein the body of the characters is the same size as their respective hitbox and centred precisely in both the x and y axis on the whole image. Its also vital that all the images

Figure 7.5: The purple hitbox represents the body of the probs

tailored to a specific object have the same dimensions. This is due to the way we coded where in respect to an objects hitbox the image should be drawn. Where each object includes specified hitbox dimensions, they also include image dimensions provided by the sprite header file. For each axis, the difference between half the image length and half the hitboxs would be the distance between their centres if they were drawn at the same point. We simply subtract this value from the hitboxs position in each axis respectfully and draw the image. Rather than sharing the same top left corner, the image and hitbox share centres. When the image is always centred on the hitbox, it doesnt matter if the hitbox changes size. This system is useful when we want to implement ducking because as long as the ducking images have the body of the figure centred and the same dimensions as all the objects other images, the image will translate accordingly.

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: ”They need to be in .wav format, audio channel set to mono, and bit resolution set to 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

Officially the *flash memory* has a capacity of 32 Kb, but the IDE does not allow uploading sketches larger than 28.672 bytes. Furthermore, the first 4242 bytes are reserved for the *bootloader*, so in reality we only have 24.430 bytes of code available.
Early optimizations is usually the bane of programming, but halfway through development we hit the code size ceiling. The code size seemed to arbitrarily grow independent of the code additions. From then on, every single addition required an optimization at least of same size, and in the end we had to sacrifice some maintainability and readability to reach our goals. Development stopped when optimization work yielded less than a 100 bytes per hour, but at that point the game was feature complete, so it just saved us from gold plating. Figure 11.1 is a graph of the code size during development.
Arduino's IDE did not make it easy for us to optimize the game. All warnings concerning dead code or unused variables and functions are suppressed. Allegedly, the warnings was thought to discourage new developers, and was removed. The compiler was already optimized to removed these parts of the code, but it does so silently to the developer.

Libraries

The extern libraries use a lot of space, especially true for the graphics library. It has been difficult to evaluate just how much space they use. For example, the difference in space between two and one function call, is the size of the call, but the difference between one and zero calls is the entire function plus the call. This is because the compiler cuts unused functions, and it makes it difficult to evaluate what parts of the code is expensive.
In any case, we could not do without the used extern libraries, nor would it be certain that we could rewrite them better. It would have taken too much time to optimize and test

the libraries, and would probably not have been more profitable than optimizing our own code.

Verbosity

Maintainability is usually a great priority in programming, but the constraints of the Arduino left us to optimize for efficiency. In many cases the code was also left less readable and robust, in favor of using less space. There was, however, a certain limit where we did not simplify. For example, two version of the same method returns a random direction, either -1 or 1:

```
char getRandDir() {
  if(random(2) == 0)
    return -1;
  else
    return 1;
}

char getRandDir() {
  return random(2) * 2 - 1;
}
```

The first function is more expensive, but also a lot more readable. The second is absolutely incomprehensible, but cheaper. We went the top method, so the intent of the code was at least understandable. This case is more symbolic of the standard we held, since the top most function only cost 4 bytes more, but it serves as a good example. In the source code, the directions are replaced with the enumeration values `LEFT` and `RIGHT`. In other cases, for example the map generator, we split the code up in functions for readability.

Encapsulation

Usually it is good code practice to encapsulate variables and use getters and setters. This was a luxury we could not afford, which became apparent when we hit the ceiling. Most variables was made public, though in some cases a getter was cheaper than directly accessing the variable.

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].

Global Variables

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.[4] 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.

Inconsistencies

Finally, in some cases the code size arbitrarily increased or decreased where it should not have. For example, there are some unused values in the `sprite.h`, which was generated by the asset converter. If these are removed, the code size increases. This is probably some inconsistency in the compiler, but we did not research this further. In other cases, where more code resulted in less code size, we removed the anomalies (or simply did not add them), but the asset converter values was discovered when space was really limited, so we had to leave it in.

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

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

Chapter 8

Testing and performance analysis

To test our performance our only marker has been frames per second. The Gameduino2 and the board supports 60 fps, throughout the project we have gone up and down in our fps, We found out that to remove the delay from our nunchuck buttons we had to update it twice this had a heavy toll on our fps count though. We used the double update until the start of the three week project, we removed one of the updates and observed that we no longer had any delay and we still have 60 fps. The game has been running decently, we have some issues with coins spawning outside the map and inside blocks which make them flicker across the screen but this happens infrequently. Our code size has gone to the exact limit of what the boards flash memory could hold.

Chapter 9

Discussion

One of our biggest problems here towards the end has been the amount of flash memory. This set a limit for what we could add and how we could add it. Some places in our code we have had to implement some things rather crudely to save flash memory, an example is that we have not used any getters/setters all is done with public variables. We have also had problems the amount we could store on our GD2 file this limited us to a few selected sprites and audio files.

Furthermore we have had some issues with spawning coins.

Additional features could be implemented but that would require a board with more memory. If this was the case we could implement the features we wanted originally, story, more monsters, items, levels and so on.
We have been very limited by the boards capacities. But this has taught us to work with code another way than we usually do. Today when making a platformer for computer we will never reach a point where we have too much code, but how compact is the code, how optimized is it? instead of optimizing you would just build additional features once the last feature worked. Since none of us had experience with c++ we have struggled a bit with the documentation on the arduino site and the "guide book" from the GD2 library. We have had some cases where the documentation has been plain wrong, and mostly confusing and badly written with bad examples.

9.1 Meeting Requirements

Cut content

Items, merchant and levels increasing difficulty? Multiple enemies?

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

Links

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

Chapter 11

Appendices

Optimization

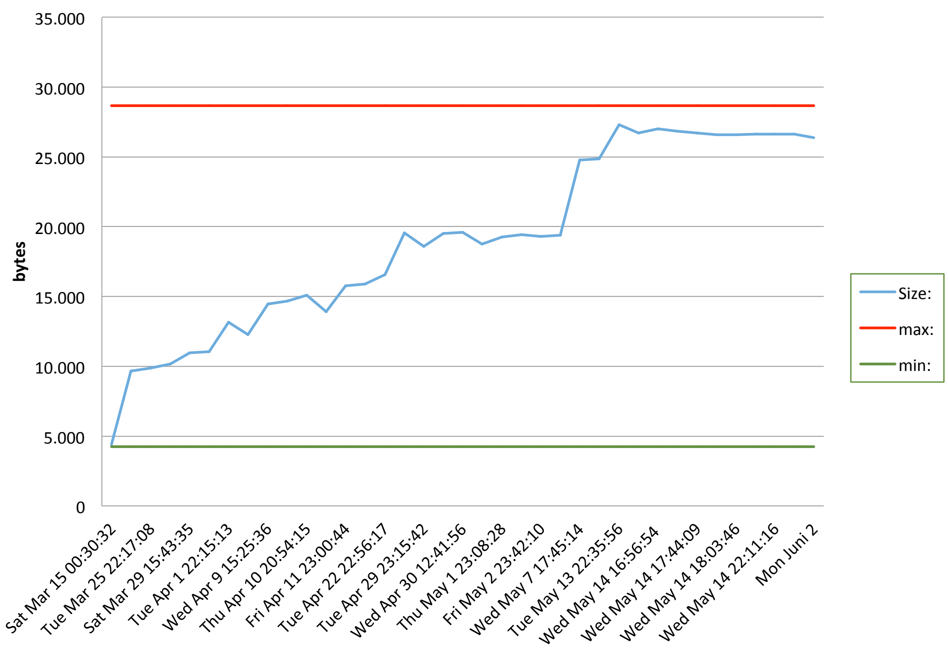


Figure 11.1: A history of the code size

Requirements

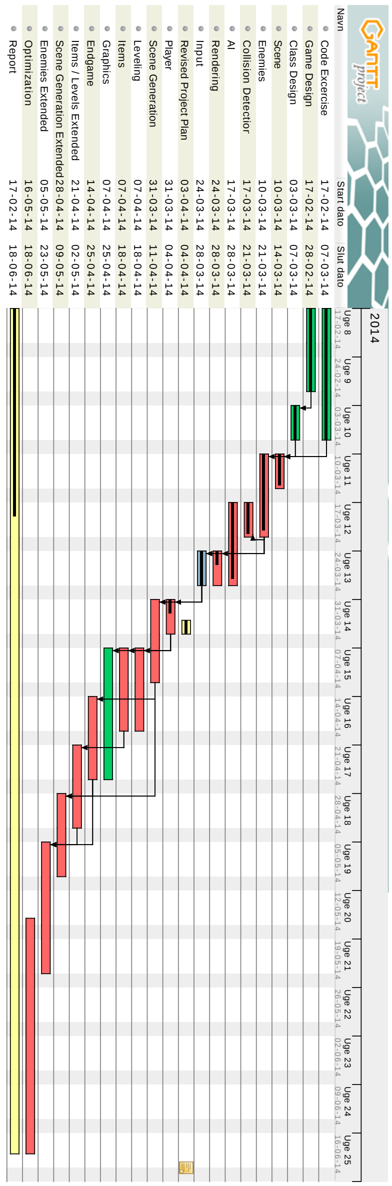


Figure 11.2: The waterfall timeplan of our project