Institutionen för datavetenskap
Department of Computer and Information Science

Final Thesis

Extending a framework for a play and learn game with drag and drop, a subgame and visual feedback

av

Arvid Johnsson

LIU-IDA/LITH-EX-G--15/021--SE

2015-08-28

Final Thesis

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

Linköpings universitet
Institutionen för datavetenskap

Final Thesis

Extending a framework for a play and learn game with drag and drop, a subgame and visual feedback

av

Arvid Johnsson

LIU-IDA/ LITH-EX-G--15/021--SE

2015-08-28

Handledare: Agneta Gulz

Examinator: Annika Silvervarg

Abstract

This report documents and describes the process of the extension of a JavaScript framework for the math game Magical Island and implementation of one of its subgames. Moreover it details the implementation of visual feedback within this subgame based on literature about feedback within games. The method details the implementation process one system at a time, starting with a study of the code and ending with the implementation of the visual feedback. The results show that a systematic approach where an understanding of the existing code is the most important thing when extending and implementing new features in a framework. They also show that to properly design a system for visual feedback knowledge of the research within this field is needed.

Contents

1. Introduction

Magical Garden is a game for small children, where they get a chance to learn the absolute basics of math in a fun and engaging way. A unique thing with Magical Garden is that it's using what's called teachable agents. A teachable agent or TA for short is a digital student of the child. This concept is built upon the established pedagogical method "learning by teaching". The existing subgames within Magical Garden focuses on developing the child's understanding of number sense. The game of which the implementation is started on in this thesis work, called "Magical Island", focuses on so called "friends of ten".

1.1 Background and motivation

The current version of The Magical Garden, MG, is primarily built for 3-5 year olds. However the Educational Technology group at Ida, Liu & LUCS, LU (the original developers) also wants something that suits older children, 6-7 year olds more specifically, as well as mathematically more advanced children in the age group of 3-5 year olds.

There is a conceptual design of a novel module for MG called Magical Island, MI, which targets topics from the grade one curriculum, specifically "friends of ten", which narrative builds on that of MG. The design is partly implemented in a Unity prototype. This prototype is however far from finished and because it's implemented in Unity it means that there are probably some machines which can't run the program. One would also have to install it on every single machine that wants to run it. This is quite a big downside if it should be used by small children in schools. It is much better if its web based because then its platform independent and it will not have to be installed on the machines.

The Magical Garden is already implemented in HTML5/JavaScript, now the Educational Technology Group wants MI to be implemented in HTML5/JavaScript so that MI too will be platform independent and not need an installation. MG already has a working framework written in JavaScript so because MG and MI is quite similar MI's framework can be heavily based upon MG's. However MI does have some game mechanics such as drag and drop which is basically nonexistent within the MG framework (one of the subgames have a very specific implementation of drag and drop which can't be used for anything else) which means that the framework needs to be extended so that it supports the new mechanics within MI.

Besides this a requirement is some kind of system for visual negative feedback implemented in MI, whereas in MG most feedback to the player is handled through dialogue. In other learning games, for example Critter Corral [7] it's handled through different types of visual cues which have the potential to help the players even more.

1.2 Purpose

The tasks to be solved within the present thesis work are to implement a framework in JavaScript which supports the mechanics within the Unity prototype of MI. In addition a system for visual feedback will be implemented within the new framework and one subgame that uses the new extensions. The final requirement is that this game should be able to run on Ipad.

1.3 Limitations

This project will not handle the framework´s connection with the backend and logging to the backend. Neither will it include production of any graphics; instead the existing graphics from MI will be used. Lastly, It won't handle any of the story or progression throughout the game in large. The reason for this is that we have concluded that it will be better if the project is focused on establishing a working base for the game so that other developers later on can continue to work on it and develop the specifics for MI within it.

1.4 Goal

The goal of this report and thesis work is to find out the following:

How do you extend a base (framework) for a TA-based game with 10-friends as well as implement a game in this framework?
How do you implement informative negative visual feedback for children within a mathematics game of this kind based on research on feedback?

2. Theory

2.1 Learning through games

Digital games are primarily thought of as pure entertainment but they can be used as very powerful tools for learning. Mark Prensky [1] explains why this is the case and how effective educational games can be developed. Firstly he states that the main reason why games can be used very effective as educational tools is that, today's learners have changed radically from previous generations and these learners need to be motivated in new ways.

So in what ways have learners changed? Prensky calls today's learners for digital natives, meaning that they have grown up with the digital world of which games are a big part of. This has resulted in a change in how people raised today think and process information. Because of this they are motivated in different ways and by different things. They are for example used to the motivational techniques of digital games. Just to name a few of the motivating elements from video games that Prensky takes up: they are fun which gives us enjoyment, they have rules which gives us structure, they are interactive which gives us doing, they have goals which gives us motivation and they have win states that gives us ego gratification.

Moreover he takes up a couple of different valuable techniques that can be utilized with very great effect in the present thesis project and in the Magical Garden overall.

- *Practice and feedback* which Magical Garden relies heavily upon in using repetition of exercises and getting feedback on one's actions within the game,
- *Learning by doing* which of course all games easily exploit because of their inherent nature as an interactive medium.
- *Goal oriented learning* which is used in Magical Garden and Magical Island, in the sense that there is a goal within the game that drives the player forward, giving him/her an incentive to keep playing.

2.2 Feedback

Blair, K. P. [2] and Moreno, R., & Mayer, R [3] describe three different types of feedback, Explanatory feedback (EF), Corrective Feedback (CF) and answer until correct (AuC) that are used in learning games. EF means that the user is provided with an explanation why their answer was incorrect, for example that the number was too high or too low. CF means that the user is shown the correct answer and then is able to imitate it. AuC means that the user only

gets to know if their answer was right or wrong and then gets to continues answering until he/she is correct.

Blair, K. P. [2] describes their implementation of the different types of feedback in the math learning game Critter Corral. While the implementations of corrective feedback and answer until correct are quite straightforward, the implementation of explanatory feedback can be quite varied. Critter Corral relies upon visual aids for the explanatory feedback. For example in one of their subgames where the users objective is to repair a chair which misses one leg, the user has several legs to choose from which all have different lengths. If the user chooses a leg that is too long the chair will start to tilt backwards and if the user chooses a leg that is too short the chair will start to tilt forwards. Finally if the leg is the exact correct size the chair will be fixed and the character on it can sit steadily. This is what they call implication feedback which is a type of explanatory feedback; they show the student the implications of their answers.

At the time of Blair`s [2] publication they were in the middle of a study regarding the effectiveness of critter corral and the three different feedback types. They used two preschool classes as their test subjects. Their preliminary test results indicated that the users use more trial and error strategies when the AuC condition is employed and more deliberate choices when CF and EF is employed. But they seem to enjoy EF and AuC more than CF which can result in them playing longer.

2.3 Visual feedback and interaction

Moreno, R., & Mayer, R [3] discuss the possibility to design games as interactive multimodal learning environments. They begin to state that the most effective learning environments are multimodal which means that the user is presented with both a verbal representation of the content as well as a corresponding visual representation. And of course if it's interactive the environment depends on the learner's action and responds accordingly, e.g. with feedback. However for the interactivity to have a positive effect on the learner it needs to foster learning.

Paek, S et al. [4] states that "The idea that physical objects can help young children learn has a long, well-argued path in educational theory" based on the research of many others. They continue to state that these days many researchers claim that virtual manipulatives can have the same positive effect. They also conducted a study using different version of a math game. One version only had visual feedback, one had both visual and auditory feedback and one had

neither. In their results we can clearly see that you gain a lot of both having visual and auditory feedback. In Magical Garden there is both verbal and visual representation for most of the content. However there is only verbal feedback and the interaction is quite moderate, there is little that the plyer can manipulate, only buttons, apart from the balloons in the balloon game which can be dragged and dropped. So the addition of drag and drop in MI can be beneficial both from a learning perspective according to Moreno, R., & Mayer, R [3] and it can be more fun according to Prensky [1]. The addition of visual feedback could also benefit the learner greatly according to Paek, S et al. [4].

2.4 JavaScript and the Phaser framework

Because Magical Island and Magical Garden are web based, JavaScript needs to be used as the main programming language. JavaScript is a scripting language made to be used in web browsers, it also is a dynamic language instead of a static language which is common for the not browser based languages such as Java, C++ or C. So what is a dynamic language? Mikkonen, T., & Taivalsaari [5] define a dynamic language as "a class of programming languages that share a number of common runtime characteristics that are available in static languages only during compilation, if at all.". They continue to state that dynamic languages include one or more of the following characteristics: Dynamic typing meaning that one doesn't have to declare variables before they are used or explicitly state what types they are. Interpretation meaning that the code is first read at runtime, translated to an intermediate representation and executed immediately and finally Runtime modification meaning that some aspects of the code can be modified at runtime.

To make it easier to write a game with JavaScript the open source game framework Phaser is used. Phaser is a HTML5 framework for making games in JavaScript. It provides the programmer with common features for making games such as physics, sprite handling, sound management, input and animation. When downloaded the programmer has accesses to a multitude of abstracted methods which makes it a lot easier to write games from scratch. E.g. it has inbuilt methods for drag and drop which make the drag and drop functionality much easier to implement. It also uses a system called groups which are groups of objects, which enables the programmer to perform operations on multiple objects at the same time. Davey R [6]

2.5 Magical Garden framework

To get a better understanding of what has actually been done in this thesis work, the reader needs a basic understanding of the framework which has been extended. The focus will be on the underlying systems for buttons and the systems that are needed for the subgames.

2.5.1 Subgame structure

The subgame structure consists mainly of three classes *Subgame*, *NumberGame* and one specific class for each subgame, for example *BeeFlightGame*. Subgame is the superclass for all the subgames which contains the general logic for all the subgames, such as round and mode handling (all the game consists of rounds which are repeated and modes in which different things happen), start and stop functions for the subgames and creation of the menu, the agent and the game object. *NumberGame* is the superclass for subgames which goal is to find a specific number, the first thing it does is to run the subgame constructor. *NumberGame* initializes the following variables.

```
/* Public variables */
this.method = parseInt(options.method || GLOBAL.METHOD.count);
this.representation = options.representation;
this.amount = GLOBAL.NUMBER_RANGE[options.range];
/* The current number to answer */
this.currentNumber = null;
/* Stores the offset of the last try, can be used to judge last try */
/* Ex: -1 means that last try was one less than currentNumber */
this.lastTry = 0;
/* This should be used to save the current position. */
this.atValue = 0;
/* This is used to add to the number of the button pushes. */
/* This should be modified only when isRelative is set to true. */
this.addToNumber = 0;

// Setup gameplay differently depending on situation.
this.isRelative = false;
this._setupFunctions();
```

Figure 1: NumberGame variables

The important ones are: method which specifies what kind of math method (add, subtract and count e.tc) which is used in the subgame and amount which specifies the largest input number in the current subgame as well as the number of buttons to be shown. The agents position,

scale, whether the agent is visible or not, the help button and the move function for the agent is also defined here. The following functions in *NumberGame* are relevant for this thesis work.

Name	Description
_nextNumber	Returns a new random number to look for.
_tryNumber	Tests if the player choose the correct value.
setupButtons	Creates a new button panel.
pushNumber	Links the buttons onClick to the subgame runNumber function.
showNumbers	Shows the number panel and hides the yes no panel.
updateButtons	Updates the values in the number panel.

Finally there is the *BeeFlightGame* class. *BeeFlightGame* runs the *NumberGame* constructor which in turn runs the subgame constructor. When the subgame and *NumberGame* have been constructed the *pos* and *preload* functions will run. The *pos* function declares the position for the in game objects and preload loads the audio and image assets. The create function sets up the buttons, music, speech, background, agent and a move function. Moreover the subgame class contains instruction functions which defines which speech instructions should be played when. A *runNumber* function which tests the inputted number with the desired value, what happens when the number is correct or wrong is also defined here and then it calls on the *nextRound* function which goes on to the next round. All subgames also have four mode functions *modeIntro, modePLayerDo, modePlayerShow* and *modeAgentTry* which functions like a main function would in many other programs. The *modeIntro* differs from the other three in that it basically only contains the intro for the subgame, it plays the relevant

instructions. The main difference between the mode functions and an ordinary main function being that the game has three of them, they are run until the player get 3 correct answers then the program will go on to the next mode function. They define a timeline and then adds all the relevant functions to their timeline. These functions will be run in the specified sequence (ordinarily the order in which they are added in the code), the functions are generally the instruction functions and the ones for manipulating the objects which are used in the games. That is because the most part of the gameplay is made up of instruction and object manipulation. The reason for having multiple instruction functions is because depending on the method used in the subgame the gameplay will vary. This means that one needs separate instructions for each method.

```
BeeFlightGame.prototype.modePlayerDo = function (intro, tries) {
    if (tries > 0) {
        this.showNumbers();
    } else { // if intro or first try
        var t = new TimelineMax();
        if (intro) {
            t.skippable();

            if (this.instructions) {
                t.add(this.newFlower(true));
                t.addCallback(this.updateButtons, null, null, this)
                t.add(this.doInstructions());
            } else {
                t.add(this.newFlower());
            }
        } else {
            t.addSound(this.speech, this.bee, 'getMore');
            t.add(this.newFlower());
        }
        t.addCallback(this.showNumbers, null, null, this);
    }
};
```

Figure 2: BeeFlightGames modePlayerDo function

```
BeeFlightGame.prototype.instructionCount = function () {
    var t = new TimelineMax();
    t.addSound(this.speech, this.bee, 'showTheWay');
    t.addSound(this.speech, this.bee, 'decideHowFar', '+=0.8');
    t.add(this.pointAtFlowers(this.currentNumber));
    t.addLabel('useButtons');
    t.addLabel('flashButtons', '+=0.5');
    t.addSound(this.speech, this.bee, 'pushNumber', 'useButtons');
    t.add(util.fade(this.buttons, true), 'useButtons');
    t.addCallback(this.buttons.highlight, 'flashButtons', [1], this.buttons);
    return t;
};
```

Figure 3: BeeFlightGames instructions for the counting method

Another big part of this framework is timelines, as one can see in both *modePlayerDo (figure 2)* and *instructionCount (figure 3)* a timeline is created. Timelines are a way to queue up different events sequentially. When something is added to a timeline it will be added at the first empty position (the last position generally). The timelines are what drives the game forward instead of having a loop which runs everything over and over the functions and sounds are added to timelines which are run when they are needed. Which timeline that runs at the moment is generally decided by the mode functions, depending on the current mode different timelines are run (each function has one timeline).

2.5.2 Button structure

The button structure mainly consists of 5 classes, *ButtonPanel, NumberButton, SpriteButton, TextButton* and *GeneralButton. ButtonPanel* is the first class used when creating a button. It sets the objects variables to the ones sent to the class such as the coordinates and links an *onClick* method to the object. Then it creates a new object *buttonOptions* containing all the variables needed to create a button. Depending on what kind of button the subgame wants to create, *ButtonPanel* calls on one of the constructors of the three classes *NumberButton, SpriteButton* and *Textbutton*. The creation is done with the *buttonOptions* object, the representation, a number and the game object After the button has been created it's added to the *ButtonPanel* group.

```
/* These options will be used when creating the buttons. */
var buttonOptions = {
    min: this.min,
    max: this.max,
    size: buttonSize,
    background: this.background,
    color: this.color,
    vertical: !this.vertical,
    onClick: this.onClick
};

/* Set up the buttons that should be in the panel. */
if (this.method === GLOBAL.METHOD.incrementalSteps) {
    buttonOptions.doNotAdapt = true;
    // Create buttons first, then add them in their order (this is because we manipulate the buttonOptions later)
    var change = new NumberButton(this.game, 1, this.representations, buttonOptions);

    // Put the other buttons centered.
    buttonOptions[this.vertical ? 'x' : 'y'] = ((this.representations.length - 1) * buttonSize)/2;
    buttonOptions.onClick = function () { change.bg.events.onInputDown.dispatch(); };
    var go = new NumberButton(this.game, 1, GLOBAL.NUMBER_REPRESENTATION.yesno, buttonOptions);

    buttonOptions.keepDown = false;
    buttonOptions.background = 'button_minus';
    buttonOptions.onClick = function () { change.number--; };
    var minus = new TextButton(this.game, '-', buttonOptions);

    buttonOptions.background = 'button_plus';
    buttonOptions.onClick = function () { change.number++; };
    var plus = new TextButton(this.game, '+', buttonOptions);
```

Figure 4: The buttonOptions object and the button setup

The *ButtonPanel* class also contains an *updateButtons* function which can update the number of the individual button objects depending on which method the current subgame is using. So if a subgame wants to create a button with a number on it *ButtonPanell* will call on *NumberButtons* constructor. Basically the first thing that *NumberButton* does is to call its parent constructor within *GeneralButton. GeneralButton* defines everything that is the same for all the buttons, meaning the coordinates, the graphics, what size and what happens graphically when a button is clicked (in this case, its color is changed and a sound is played). After a *GeneralButton* is constructed the program will go back to the *NumberButton* constructor. There it defines the specifics for a number button such as the smallest number it can be and the largest, the actual number it has, and finally it defines that the number should be sent to the *onClick* method when the button is clicked. Moreover it contains the function *updateGraphics* which defines which representation the button should use and creates the representation and places it on top of the button graphic.

```
function NumberButton (game, number, representations, options) {
    /* The order here is a bit weird because GeneralButton calls setSize, which this class overshadows.
    if (typeof options.keepDown === 'undefined' || options.keepDown === null) {
        options.keepDown = true;
    }
    this.representations = representations;
    this.background = options.background;
    this.spriteKey = options.spriteKey;
    this.spriteFrame = options.spriteFrame;

    GeneralButton.call(this, game, options); // Parent constructor.

    this.vertical = options.vertical;
    if (typeof this.vertical === 'undefined' || this.vertical === null) {
        this.vertical = true;
    }
    this.setDirection(!this.vertical);

    this.min = options.min || null;
    this.max = options.max || null;
    this._number = 0;
    this.number = number;

    this._clicker = options.onClick;
    /* This will be called in the GeneralButton's onInputDown */
    this.onClick = function () {
        EventSystem.publish(GLOBAL.EVENT.numberPress, [this.number, this.representations]);
        if (this._clicker) {
            this._clicker(this.number);
        }
    };

    return this;
}
```

Figure 5: NumberButton constructor

The final thing needed for the button structure is the *setupButtons* function within the *NumberGame* class. The *NumberGame* class is a superclass for most of the subgames. The *setupButtons* function provides the link between the button set up from within the subgame and the actual creation in *ButtonPanel*. The main functionality of *setupButtons* is to link the *onClick* function of the buttons to a function which defines what happens internally within the current subgame, to create the button panel and save the resulting object in a button variable which the programmer has access to within the subgames for manipulation.

3. Method

3.1 Learning the code and setting up a base

Because the new framework would be quite similar to the old one, instead of starting from scratch instead the existing one was extended. At the start of the project a thorough examination of the code was conducted, to see what needed to be added, deleted and modified. A conceptual design was drawn up (figure 6), showing the new structures to be added. The information presented in 2.5 is the result of the study. Because the backend isn't part of this thesis work it was separated from the relevant code so that it wouldn't interfere with the new code. A baseline subgame from where the new extensions could be tested was needed. The baseline subgame was a very striped down copy of the *BeeFlightGame* named *SharkGame*. The only things left in the baseline subgame were the functions that loaded the assets, the ones that created the specific objects and placed them on the screen as well as the two mode functions *modeIntro* and *modePlayerDo*.

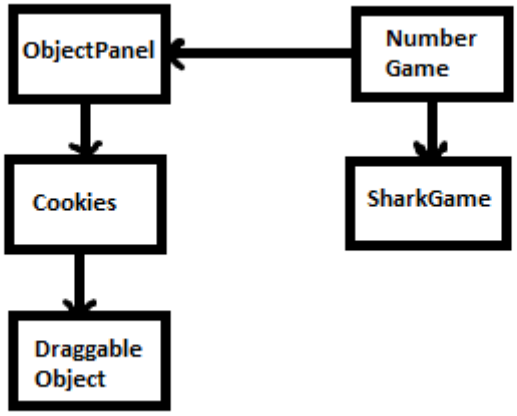


Figure 6: NumberGame creates the subgame and the ObjectPanel, ObjectPanel creates Cookies which are DraggableObjects.

In Magical Gardens development build there is a menu in which the player chooses which game, method, the range of the numbers (1-4 or 1-9) and the representation. In Magical Island the player will instead have a screen with different islands on which the different subgames are on. The selection menu wasn't removed instead the options were defaulted so the only button which matter is the actual subgame button which later on can be bound to an island instead. In Magical Garden the possible numbers on the objects always correlated to the amount of objects (1-9 or 1-4). In the subgame that was implemented in this project there

should always be 4 objects but with 1-9 as possible numbers. So instead of setting the max number to *amount* it was set to 9 directly.

3.2 Implementation of the draggable objects

When implementing the base structure for the draggable objects the structure for creating buttons was copied. The classes *ButtonPanel*, *NumberButton* and *GeneralButton* was copied and their names where changed to *ObjectPanel*, *Cookies* and *DraggableObject*. The biggest change that was made was that everything regarding the representations on the buttons was deleted. Because in MI there is only a number representation so it was unnecessary to include the different representations when only one will be used. Besides on the given cookie images the numbers where already drawn. Instead the number was saved internally in the object and the corresponding picture is loaded in. Furthermore the variables *dropPlaceX*, *dropPlaceY*, *id*, *idName*, *startPosX* and *startPosY* were added to the *objectOptions* field in *ObjectPanel*. *DropPlaceX* and *Y* define where the object can be dropped, *id* identifies which is which of the same objects, *idName* identifies what kind of object it is and *startPosX* and *Y* keeps track of the original position of the object in the original coordinate system. *StartPosX* and *Y* are needed because in *DraggableObject* each object has its own coordinate system so the original start position is needed when calculating the new position for the visual feedback.

```
var objectOptions = {
    min: this.min,
    max: this.max,
    size: objectSize,
    background: this.background,
    color: this.color,
    vertical: !this.vertical,
    onClick: this.onClick,
    dropPlaceX: this.dropPlaceX,
    dropPlaceY: this.dropPlaceY,
    id: 0,
    idName: this.id,
    startPosX: this.x,
    startPosY: this.y
};
```

Figure 7: The objectOptions object

3.3 Implementation of the drag functionality

The drag and drop functionality for the objects was very easy to implement thanks to Phaser. Phaser has built in support for drag and drop so the only thing one has to do is to make an object draggable, as seen in figure 8.

```
this.bg = this.create(0, 0, (background === null ? null : 'cookie'), background);

this.bg.inputEnabled = true;
this.bg.input.enableDrag(true,true);
```

Figure 8: Creating the sprite and enabling drag

One of the big reasons to why the number representation wasn't used (besides the fact that the numbers already were on the images) was the fact that the representation and the sprite were two different objects. And Phasers implementation of drag (seen in figure 8) does not support drag on multiple objects at the same time. This meant that it would be easier to have the cookie and the number as one image instead and change the images during the course of the game so they correlate to their objects internal numbers. In order to trigger something when the player drops the object at a specific position one also have to use the *onDragStop* functionality with which one can bind a new function to be triggered when you stop dragging the object. In the *stopDrag* function the position of the mouse cursor is cheeked so as to know where the object is. However the draggable objects and the input (mouse and touch) dos not use the same coordinate system so the input coordinates need to be adjusted so as to correlate to the objects coordinates. Then the new adjusted input coordinates are compared to the *dropPlace* coordinates, if the input coordinates corresponds with the *dropPlace* coordinates. The *dragObject* is placed according to the design of the visual feedback (explained in 3.9) otherwise it's placed on its start position.

```
function stopDrag(options){
    var x = this.game.input.x;
    var y = this.game.input.y;

    x = x-156;
    if (this.idName !== 'finalDragObject') {
        if(x > this.dropPlaceX -10 && x < this.dropPlaceX+ 80 && y > this.dropPlaceY -10 && y < this.dropPlaceY + 80) {
            if (this.onClick) {
                this.onClick();
            }

            if ( DraggableObject.try < 0) {
                options.x = this.dropPlaceX - (this.startPosX+ ((this.id-1) *113));
                options.y = this.dropPlaceY - this.startPosY;
            }
            else if(DraggableObject.try > 0){
                options.x = this.dropPlaceX - ((this.startPosX+ ((this.id-1) *113))+75);
                options.y = this.dropPlaceY - this.startPosY;
            }
            else{
                options.x = 1;
                options.y = 1;
            }

        }
        else{
            options.x = 1;
            options.y = 1;
        }
```

Figure 9: The stopDrag function

3.4 Implementation of the usage of the dragObjects from the subgame

The original connection of the buttons to the *onClick* method was done in *GeneralButton* by having a function *onInputDown* which called on the objects *onClick* method which is linked to the *NumberButton* class *onClick* which in turn is linked to *buttonPanels onClick* which called the function *pushNumber* which finally called the *runNumber* function within the current subgame.

This structure was basically kept intact with only one real difference, being that *dragObjects onClick* was called from its *stopDrag* function. The reason for this is that when using drag and drop interfaces you want the resulting function to be triggered on drop and more often than not (as in this case) you only want it to be triggered when dropped on a specific position. This is why the *onClick* function first is called if the new coordinates corresponds to the drop place coordinates.

3.5 Implementation of the goal objects

In order to have the possibility to have different drop zones in different subgames as well as having a more easily extended structure the drop coordinates are sent to the *dragObject* at creation from the subgame. A *goalObject* which coordinates are generally the same variable as the *dragObjects dropPlace* coordinates was also made. This e.g. makes it possible to switch drop positions by only changing the value of the two *dropPlace* coordinates that you send to *dragObject*. An even neater solution would be to get the *dropPlace* coordinates directly from the *goalObjects* sprite size but then you would also force future programmers to have a *goalObject* when they are using a *dragObject* which isn't always optimal. The *goalObject* is basically a downscaled version of the *dragObject*, everything handling any kind of input was deleted. The only thing left was the constructor which sets the coordinates, color and the background sprite, the *setSize* function and a highlight function which can highlight the object. There is also a child class to *goalObject*, *goalCookie* which basically is the same as the *Cookies* class except for the input handling.

A small addition was needed to be done in *ObjectPanel* and *NumberGame* for the creation of *goalObjects*. Here the *id* object variable came in handy because both the cookies and the *goalObject* are created within *objectOptions* so depending on if the current object you wanted to create was the *Cookies* or the *goalObject* the different constructors were called as seen in figure 10.

```
if(this.id === 'dragObject' ) {
    for (var i = this.min; i <= this.max; i++) {
        this.cookie = new Cookies(this.game, i, objectOptions);
        this.add(this.cookie);
    }
}
else if(this.id === 'goalObject') {
    this.goalCookie = new GoalCookie(this.game,6, objectOptions);
    this.add(this.goalCookie);
}
```

Figure 10: Button setup in ObjectPanel

The same thing was needed to be done in *NumberGames setupDragObject* function. If the current object is the *dragObject* the *dragObject* options where sent to the *ObjectPanel* and vice versa.

3.6 Implementation of the *SharkGame*

The subgame which were going to use the *dragObjects* was already basically fully implemented in the MI prototype which meant that the design was already done. The subgame works like this, there is a monkey which wants to get water from the ocean but an angry shark won't let him so to make the shark happy the monkey tries to feed him with cookies. However the shark only wants whole cookies and all of the monkey's cookies are broken. The player's task is to match two cookies to make one whole and then feed the shark with the new whole cookie. At the start of each round a new goal cookie is loaded in as well as four alternative cookie pieces to match it with.

Figure 11: The shark game

Each cookie has a number and a whole cookie has always the number ten so if the goal cookie has number six the player should drag the cookie with number four to the goal cookie because 6+4 = 10. When the player has accomplished this three times the first mode is over and the next one is started. Now a TA will come and watch and after another three times the last mode is started wherein the TA will try to guess the correct number. When the TA guesses the player's task is to confirm if it was correct or not by clicking a yes or no button. If the agent was wrong the four alternatives will be shown and the player gets to choose. If the

agent is right the program will go on to the next round and after three successful guesses the game is done.

3.6.1 Graphics

The first thing implemented in the subgame was the graphics from the Magical Island cookie subgame. The existing framework is setup to work with sprite sheets with corresponding json files. This meant that the given separate images had to be used to generate new sprite sheets with a tool found online called texture packer. One sprite sheet including the backgrounds and the shark, one with the monkey and one with the cookies was created. In figure 11 one can see that the monkey doesn't have any legs that's because there wasn't any separate image for the legs, meaning that no sprite sheet could be generated for them. When constructing the sprite sheet for the cookies the placing algorithm for the images in the sprite sheet had to be set to basic. This algorithm, instead of altering the orientation of the images to use the size of the sprite sheet optimally, just places the images next to each other in their original orientation. If another of the more optimal placing algorithm where used it would cause the game to not always find the correct image within the sprite sheet.

Sometimes graphic bugs occur in the selection screen which is left in from the development version of MG. It is likely that it has to do with the uncoupling of the backend because the graphic bugs started first after the back end was uncoupled.

As mentioned earlier the cookie images also have the numbers on them which means that they have to be switched after every successful pairing of two cookies. This must be done every time when a new set of cookies should get loaded in. In Phaser the only thing needed to switch objects background images when using sprite sheets is to change the background frame.

```
Cookies.prototype.updateGraphics = function (num) {
    /* Remove old graphics. */
    if (this.children.length > 1) {
        this.removeBetween(1, this.children.length-1, true);
    }

    if(this.idname !== 'finalDragObject') {
        this.bg.frame = num + 8;
    }

    this.setSize();
    this.reset();
};
```

Figure 12: Cookies updateGraphics function

The switch is handled by the *updateGraphics* function seen in figure 12. The *num* variable is the actual number you want to show, the 8 that is added to *num* is needed for the number to correlate with the index of the wanted image in the json hash. On every new round the update graphics method is called for each separate cookie which sets the frame number corresponding to the cookies number which is set in the calling function. Then the number is modified to correspond to the correct frame in the sprite sheet.

3.6.2 The runNumber function and round handling

A part of the implementation of the subgame was that the *onClick* function needed to be linked to the *runNumber* function in order to let it check if the chosen number was correct or not, as well as progressing to the next round with the *nextRound* function. Each round consists of one complete run through of one of the mode functions *modeIntro, modePlayerDo, modePlayerShow* or *modeAgentTry*. The game begins in the *modeIntro* function which basically only plays the description audio files and then goes on to the next round which will be *modePlayerDo*. Unlike *modeIntro,* the other three mode functions have at least three rounds each, the mode is only changed after three rounds with a correct answer. *SharkGames modePlayerDo* is a scaled down version of bee games *modePlayerDo* this is the first mode where the player does everything without a TA. When *modePlayerDo* is finished it will go on to *modePlayerShow* which is basically the same as *modePlayerDo* except that a specific sound asset is played. In this mode the player also does everything by himself but the TA is watching, finally it will go to the *modeAgentTry* this mode is quite a bit different from the others. At first the only thing that is shown is the goal object with the *showGoalObject* function and then the TA will guess the right answer with the *agentGuess* function. Then the *showYesNo* function is run which enables input and shows the "yes" and "no" buttons and the player will get to say if the agent was correct or not. The "yes" and "no" buttons also have *onClick* functions which if "no" is clicked will run *showNumbers* so the player can put in the correct answer and if "yes" is clicked the number will be checked with the correct number and then proceed to the next round. If the number wasn't correct when the player clicked "yes" the cookie alternatives are shown and the player gets to guess.

```
SharkGame.prototype.runNumber = function (number, simulate) {
    //var current = this.currentNumber-1;
    var sum = number + this.addToNumber;
    var result = simulate ? sum - this.currentNumber : this.tryNumber(number, this.addToNumber);
    var correct = false;
    this.disable(true);

    var t = new TimelineMax();
    if (GLOBAL.debug) { t.skippable(); }

    /* Correct :) */
    if (!result) {
        correct = true;
        t.addCallback(function () {
            this.showWinObject();
        }, null, null, this);

        this.setRelative(correct);
        /* Incorrect :( */
    } else {
        correct = false;
        t.addSound(this.speech, this.ape, 'wrong');
        if(this.currentMode === GLOBAL.MODE.agentTry){
            this.showNumbers();
        }
        this.test = false;
        this.setRelative(correct);
        this.nextRound();

    }
```

Figure 13: SharkGames runNumber function

The *runNumber* function in *SharkGame* (shown in figure 13) checks if the sent in number is correct with the preexisting function *tryNumber*. If the returned result was correct the win object (the whole cookie) is shown and the *goalObject* and the *dragObjects* are hidden. If the returned result was wrong the sound for the wrong cookie is played if the current mode is *agentTry* the cookie alternatives are shown with the *showNumbers* function and the *nextRound* function is called.

3.6.3 Cookie numbers and images

In the SharkGame you want three of the cookies numbers to be randomized and one of them to be the desired number. One also wants to update them and their images each new round. The number randomization and update was done in the already existing *_updateObjects* function, the *_updateObjects* function (seen in figure 13) is the renamed copy of the previously mentioned *_updateButtons* function. Originally it changed the order of the

numbers depending if you wanted them from least too biggest or the other way around. However 3 random numbers, the desired number as well as the invers of the desired number for the goal object was needed for the shark game. E.g. if the desired number is four the goal object should have 10-4 = 6. To achieve this a number between one and nine was randomized, the current desired number was saved to the local variable "correct" and an index between one and four was randomized. Then if the current object being handled was a cookie the program looped over all the cookies and sat the cookie with the random index internal number to the desired number and the rest of the three to a new random value. At the end of each loop the current cookies *updateGraphics* function (seen in figure 14) was called and there the image was set for that cookie. The check for final *dragObject* is a failsafe; if *updateGraphics* is called with the finalDragObject it shouldn't change the frame because the *finalDragObject* should always have the same image (the whole cookie with the number 10). The internal number doesn't matter for the *finalDragObject* and that's why that check isn't in the *_updateObjects* function. However if the object being handled by *updateObjects* is *goalObject*, *GoalCookies updateGraphics* is immediately called with the invers of the correct number. To get the numbers and images to change every time the *_updateObjects* is called from the *showNumbers* function that is called on every new turn in the *modePlayerDo* function. There was however a problem with this because when the player chooses the wrong number the game still progressed to the next turn which meant that the draggable cookies images would be changed when the player were wrong which they shouldn't. The *goalCookies* image didn't change because its number is set up since before only to change when the correct answer is given. The solution to this was to set *this.isRelative* to true or false depending on if you were wrong or correct after each new answer is checked. The show numbers function checks if *this.isRelative* is true and if it isn't *_updateObjects* isn't called.

```
ObjectPanel.prototype._updateObjects = function (currentNumber) {
    var rand = Math.round(Math.random()*10);
    var correct = currentNumber;
    var randIndex = Math.floor(Math.random()*3) ;
    var i = 0;
    if(this.cookie) {
        for (var key in this.children) {
            this.children[key].min = this.min;
            this.children[key].max = 9;
            if (i === randIndex) {
                //alert(key + ' : ' + correct);
                this.children[key].number = correct;
            }
            else {
                this.children[key].number = rand;
            }
            Cookies.prototype.updateGraphics.call(this.cookie, rand);
            rand = Math.round(Math.random() * 10);
            i++;
        }
        i = 0;
    }

    if(this.goalCookie) {
        GoalCookie.prototype.updateGraphics.call(this.goalCookie, 10-currentNumber);
    }
};
```

Figure 14: ObjectPanels _updateObject function

Probably the biggest difficulty with this system was the handling of the two different objects *this.cookie* and *this.goalCookie* because at every step the programmer needs to make sure that the function does the correct operation with each object. E.g. there is only one *goalCookie* but 4 *Cookies* which is why the check in *updateObjects* is required, otherwise the program will crash when it tries to loop over nonexistent objects.

3.6.4 The sound

The sound in the rest of the subgames is handled with one file containing all the lines coupled with a speech sheet containing so called markers which details where each line starts in the file and how long it is. This means that one only needs to load in one sound asset and the play parts of it whenever you want. In the magical island prototype they use one file for each line of dialogue. In order to follow the standard within the framework the individual files were merged to one using the program audacity. Then it was a simple task of adding the wanted parts of the file to the appropriate timelines with the help of the markers and the preexistent *addSound* function from the timeline library.

3.6.5 The win object

The win object is the whole cookie which is shown when you have put together the correct pieces which then should be fed to the shark. The *finalDragObject* (the win object) is created in the same way as the ordinary *dragObjects* with the exception that only one is created and its image is defaulted to the number ten cookie. In *finalDragObject*s *onClick* function the *nextRound* function is triggered as seen in figure 15 instead of in *dragObjects*. However *nextRound* was still needed to be called from *runNumber* if the player chooses the wrong number. In that case no *winObject* should be shown but one still wants to progress to the next round. So the *runNumber* function now only runs *showWinObject* and sets *this.isRelative*. Then the player gets to move the whole cookie to the shark and the win sound is played and the *nextRound* function is run.

```
NumberGame.prototype.pushNumber = function (number) {
    this.saidAgentWrong = false;
    return this.runNumber(number);

};

NumberGame.prototype.moveObject = function(){
    this.hideWinObject();
    return this.win().addCallback(this.nextRound, null, null, this);

};
```

Figure 15: The pushNumber and moveObject functions

The win function which is linked to final *dragObject* plays both the dialogue line when the shark is fed and when the player wins the entire game. In order to know when the player has won the entire game the *_totalCorrect* variable is checked because the player has won when he/she has gotten the correct answer 9 times as seen in figure 16.

```
if(this._totalCorrect !== 9) {
    t.skippable();
    t.addSound(this.speech, this.ape, 'correct')
}
else{
    t.skippable();
    t.addSound(this.speech, this.ape, 'win');
}
return t;
```

Figure 16: Check which sound should be played in the win function

3.6.6 Showing the equation

When the correct cookies are paired together the preformed equations is shown in numbers, e.g. if the player paired cookie 4 with 6, 4 + 6 = 10 should be shown. Because they wouldn't be interactable in any way, but just fade in when a pairing is made and fade out when the shark has been fed. They are implemented as ordinary sprites. In *SharkGames* create function the sprites are created with the images for +, =, 10 and two arbitrary numbers (seen in figure 17) which are replaced in runNumber. Their visible variable is set to false which means that they won't be visible. In *runNumber* when the player is correct the first sprites frame name is set to the number that was chosen by the player and the second number to 10 – the chosen number and then the sprites visible variable is set to true as seen in figure 18. Lastly in the win function which is triggered when the shark is fed the sprites visible variable is once again set to false.

```
this.number1 = this.gameGroup.create(50, 200, 'numbers', '1.png');
this.plus = this.gameGroup.create(120, 180, 'numbers', '+.png');
this.number2 = this.gameGroup.create(150, 200, 'numbers', '1.png');
this.equal = this.gameGroup.create(200, 200, 'numbers', '=.png');
this.answer = this.gameGroup.create(250, 200, 'numbers', '10.png');
this.number1.visible = false;
this.plus.visible = false;
this.number2.visible = false;
this.equal.visible = false;
this.answer.visible = false;
```

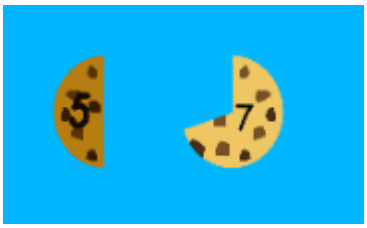
Figure 17: Sprite initialization

```
if (!result) {
    correct = true;
    t.addCallback(function () {
        this.showWinObject();
    }, null, null, this);
    var goalNumber = (10 - number);
    this.number1.frameName = number +'.png';
    this.number2.frameName = goalNumber +'.png';
    this.number1.visible = true;
    this.plus.visible = true;
    this.number2.visible = true;
    this.equal.visible = true;
    this.answer.visible = true;
```

Figure 18: Setting the sprites to the corresponding images

3.7 Implementation of the visual feedback

The implementation of the visual feedback works as follows in the game. When the correct number or a number smaller than the correct number is drawn to the goal cookie the *dragObject* will be placed in the hole in the goal cookie. If the number is too big it will be placed to the left of the goal cookie.



To get the cookie pieces to fit together they needed to have the same orientation, in the original images they had the same orientation. And thanks to the use of the basic algorithm when constructing the sprite sheet (mentioned in 3.6.1) the images kept their original orientation. The next problem was that all the objects had their own specific coordinate system meaning that their start position always was 1,1. That's why the ids are needed to keep track of the individual objects because then one can set a new position for each individual object. The *dropPlaceX* and *Y* variables of course didn't match the object coordinates either which meant that compensation for them was needed too. On the other hand because their star position is 1.1 for all of the objects it was very easy to move them back to the start positions.

The placement of the cookie when the number is to big was done in the same way except that it was placed the width of one object to the left of the goal object. The hard part was to get the program to figure out whether the current objects number was smaller or bigger. In the *NumberGame* class there is a *tryNumber* function which tests whether the chosen number was correct or not, but it also saves the offset to the correct number in *this.lastTry*. Meaning that if the answer was too small the offset will be less than zero, if the answer was to big the offset will be more than zero and if the answer was correct the offset will be zero. A new function *setTry* was created in *DraggableObject*; it's called from *tryNumber* with *this.lastTry*. As mentioned earlier, in JavaScript when you call on other objects prototype functions you have to send with an object (this) and that is the object that will be handled in the function**.** Which meant that instead of using "this" (which in this case would not refer to *DraggableObject* but the object from which one sent the call) in *setTry, DraggableObject.try* is specifically stated which is accessible in the *stopDrag* function. In *setTry DraggableObject.try* is set to the value

of *this.setTry*. The placement is handled by the algorithm shown in figure 19 which works as follows. This.id is the id number (1-4) of the cookie meaning that the algorithm for the first cookie will basically only subtract *startPosX* from *dropPlaceX*. The multiplication with 113 is needed for the 3 following *Cookies* because the space between each cookie is 113. The algorithm for placement to the left of the *goalCookie* is the same with the addition of an addition of 75 at the end because the cookie should be placed 75 pixels to the left of the *goalCookie*. The y position is the same for all of the *Cookies*, which is why it's enough with the initial subtraction.

```
if ( DraggableObject.try < 0) {
    options.x = this.dropPlaceX - (this.startPosX+ ((this.id-1) *113));
    options.y = this.dropPlaceY - this.startPosY;
}
else if(DraggableObject.try > 0){
    options.x = this.dropPlaceX - ((this.startPosX+ ((this.id-1) *113))+75);
    options.y = this.dropPlaceY - this.startPosY;
}
else{
    options.x = 1;
    options.y = 1;
}
```

Figure 19: Sprite placement algorithm in stopDrag

3.8 Conversion to Ipad

As stated above, one of the requirements was that the game should be able to run on an Ipad. When everything else was done an Ipad was borrowed from the Educational Technology Group to test on. The testing process was very easy because the program is run on a development server which is run locally on my computer. To test it on Ipad one simply had to connect with the server through the Ipad's web browser using my computers ip address and port number 9000 which is the port that the server runs on.

Luckily the only real change needed to be done in the code was that, *DragObjects stopDrag* function checked specifically for the mouse coordinates. But because Ipads only use touch input that did of course not work. The very simple fix was to check for all inputs last coordinates which works for both touch input and mouse input.

```
var x = this.game.input.x;
var y = this.game.input.y;
```

Figure 20: Getting input coordinates in stopDrag

There is however one problem with the program at the moment, and that is that the audio files that were given and then merged is too big for an Ipad which causes the program to crash about 60% of the time at startup. The audio file simply needs to be compressed in order to lessen the amount of stereo channels used. A few different file formats were tested of differentiating sizes but there were no major differences. But because sound compression is not a specific part of this thesis work we have agreed that future developers will solve this problem.

4. Result

In this chapter exactly what has been implemented and the reasons are presented.

4.1 The draggable and goal objects

Drag and drop is a part of another subgame in Magical Garden, so why was that implementation not just copied to the shark game? That is because what was wanted was a general and extendable structure that could be used in other subgames just as with the already existing buttons in Magical Garden. Besides drag and drop objects are basically the same thing as buttons with the addition that they can be moved with the mouse. This meant that the button creation could be copied to be used when creating drag and goal objects, avoiding much of the ground work. This meant that a fast and working base for my drag and drop functionality was established. So what was implemented is an easily extendible structure for drag and drop mechanics, the only thing one needs to do to create a new *dragObject* except actually creating it in *NumberGame* and a subgame is to create one new class modeled after the *Cookies* class and create it in the object panel. Then one can add features to it by extending the *ObjectPanel* and *DraggableObjects stopDrag* function. New *goalObjects* can be created in exactly the same way.

```
if(this.id === 'dragObject' ) {
    for (var i = this.min; i <= this.max; i++) {
        this.cookie = new Cookies(this.game, i, objectOptions);
        this.add(this.cookie);
    }
}
else if(this.id === 'goalObject') {
    this.goalCookie = new GoalCookie(this.game,6, objectOptions);
    this.add(this.goalCookie);
}
else{
    this.add(this.finalDragObject = new Cookies(this.game,10, objectOptions));
}
```

Figure 21: Button setup in ObjectPanel

4.2 The graphics and the audio

The reason why sprite sheets were used instead individual images was that the existing framework already used sprite sheets. Which also meant that the way they were handled could be copied which in turn saved time. Time that otherwise would have been spent on having to look up ordinary sprite handling in the Phaser documentation. Besides it's much more convenient to have one uniform way in which one handle all the images than two separate. Besides the images had to be changed for the objects quite often which is much easier to do with a sprite sheet coupled with a json file than if individual images were used. As stated above it's neater to have one uniform way to handle a group of assets than several different ways. That's why the audio implementation was done in the same way as in the rest of the subgames.

4.3 The shark game

The shark game was implemented as described in 3.7, as stated earlier most of the structure is copied from the *BeeFlightGame*. *SharkGame* also uses *NumberGame* and subgame in basically all the same ways. This is because at a very basic level they are the same types of game but with different input types. They are both games whose goal is to find a specific number, they both use the same mode and round mechanics and both use TAs. Basically when one is given working code why not use it? Instead the two main things that were implemented was a system which would randomize numbers and switch to the corresponding images for the objects. As well as the following modifications to the round and mode handling: a way to stop the objects from being updated when the answer isn't correct and trigger the next round after a second input instead of the first. The first modification was achieved by changing the *isRelative* variables value from true to false as stated earlier. *IsRelative* was already checked

in *showNumbers* so it was the neatest solution. To trigger the next round on the second input was achieved by calling *runNumber* with the first inputs *onClick*, in *runNumber* show the object which has the second input and then moving the *nextRound* call to the new function triggered by the second input. The implementation was designed in this way because otherwise an entire new mode function would have had to be constructed where the second input could be triggered which would have altered the entire lay out for the rest of the framework.

4.4 The visual feedback

Lastly a system for visual feedback was implemented within the drag and drop structure so all the new *dragObjects* can use a version of it to. The visual feedback was implemented as described in 3.8. It was based on the design presented in Blair K.P [2] by showing the player the implications of his/her choice. The player gets to see if his/her choice lead to a whole cookie or not. As stated in [2] and [3] visual feedback should be informative and tell the player in what way he/she is wrong but without telling them the right answer. This follows the basic principles of EF or more specifically implication feedback, as in one of the versions of Critter Corral. In this version of Critter Corral there is a clear difference between if the player has chosen too high a number or too low a number. To emulate this, the too small pieces are placed in the *goalCookies* hole so that the player easily can see that the piece that they picked isn't large enough. If the too large cookies would have placed in the hole they would have covered the hole completely and the cookie would seem whole even though the answer was incorrect. Instead it is placed to the side of the goal cookie to simulate that they don't fit together. Just as in the real world if one would try to fit together two cookies that don't match the best result one could get would be the pieces at the side of each other. And when they are directly beside each other the player can compare the two more easily and hopefully realize that the chosen piece indeed was too big.

5. Discussion

5.1 Result

While an extendible structure for draggable objects have been implemented there are some possible disadvantages to it. To begin with the sprite object needs to be created in the *DraggableObject* class because it's on the sprite that the drag and drop functionality is enabled. Otherwise one would have to enable it on every single object and specify a new stop drag function for every object. However because of this if one wants to create a new object that uses another set of images one would either have to make a new sprite sheet containing both the new and old images and compensate for it in *updateGraphics* or have an if check in the constructor. Depending on what object is created a different sprite sheet is loaded. A better way to do it would for the programmer to state which sprite sheet he/she would want to use in the specific objects constructor which then sends it to the *DraggableObject* constructor but support for this hasn't been implemented.

Secondly the *dragObject* gets its drop position at creation which one generally would set to the same coordinates as the *goalObjects* coordinates. This gives the programmer freedom to decide if he wants to use a *goalObject* or not, because it's possible to have a *dropPlace* specified even without a *goalObject*. However if *DraggableObject* instead got the drop position directly from the *goalObject* with a get function one would get an safer environment where the drop place couldn't by mistake get the wrong coordinates. It would eliminate human error and the programmer wouldn't have to rely as much on statically typed numbers.

The shark game have been implemented with all the same features as in the prototype with one exception. In the prototype there are several animations for the cookies, e.g. on each new round when new cookies are created they glide from the cookie jar to their positions, in my implementation they just fade in. When idle they also increase and decrease in size to indicate that they are interactable. In my implementation they aren't animated. These animations surely contribute to the player's enjoyment, but because this project had a limited amount of time. Animations where not prioritized and instead focused on the gameplay and features of the game.

While the visual feedback is implemented as stated which also follows the theories of EF, IF and virtual manipulates presented by Blair, K. P. [2], Moreno, R., & Mayer, R [3] and Paek, S., Hoffman et al [4] it could have gotten a nicer finish. Especially what could be improved is the situation when the correct choice is made. Because the same objects are reused every turn

with just updated graphics, if the cookie moved to the goal cookies position isn't moved back to its start position it will still be on the goal cookie the next turn when it's shown again. This would sometimes give the player the correct answer straight away. The best way to do this would be to move it back while it's still faded out, to do it at the same time as the graphics are updated in the *updateObjects* function was tested. However the objects internal coordinates are saved in *DraggableObjects* and there every cookies start position is 0.0, as mentioned earlier each cookie has its own coordinate system. This is not the case in *ObjectPanels updateObjects* function, an implementation of a conversion from the ordinary coordinate system to the objects own was tested but failed. Instead when the correct cookie has been placed on the goal cookie the position is reverted immediately to its start position. This means that the player can see the cookie being moved back right before it's faded out. This doesn't look that good but it was deemed that it would be better than leaving it in its place and sometime giving the player the correct answer the next round.

5.2 method

Learning the code base and getting an understanding of what had to be done was done by studying the code and writing down quite generally what happened where in the code. The goal was to get a general picture of the code; this was achieved in about 2 or 3 days. However this general picture was far from enough. Much more time should have been spent on getting an understanding of the finer points of the code and what the relevant functions actually did and then write it down in a well-structured and readable way. That would have saved a lot of time at the beginning of the project.

When the structure for the draggable object was done a search for information about how to implement the drag functionality itself was begun. The mistake was that instead of looking up if Phaser (the framework Magical Garden uses) had any built in functionality for drag and drop. Implementation of drag and drop with jquery was started. This was a complete waste of time because as seen in the method chapter Phaser has its own support for drag and drop. If Phaser had been looked up at the start of the project, a few days' worth of time would have been saved.

A big advantage of the implementation of the setup of the creation structure of objects in the framework meant that it was very easy to implement both the *goalObject* and the *finalDragObject*. Another big advantage was that the implementation of the subgame could

use much of the existing code from the bee game. All structures and major functions was copied they only needed some small modifications. The only new system was the system for updating the objects which were basically built from the ground up.

At the end of the project the visual feedback and the showing of the equation when the correct cookie has been placed were implemented. Both of these features were quite easy to implement thanks to a good working knowledge of the code.

The plan for the proceeding of the project was structured in a good way. The fact that the testing environment was the first thing to be set up and that the implementation of the draggable object was started after it was done allowed for a lot of testing without the risk of sabotaging any other part of the code. However the approach to the coding during the project could definitely have been better, during the project the approach was: Test first, study the code later. Many unnecessary mistakes where made which would have been easily avoided if the relevant part of the code would have studied before trying to implement something new. Just the sheer amount of recompilations has taken up an unnecessary amount of time. On the other hand it has resulted in a deep understanding of the code because what happens when something goes wrong has been experienced many times.

The codebase that was given was quite large and as stated before this enabled a lot of reuse of the existing code. This means that much of what has been done is not coding new systems. Of course has some new functions been written but most of the work has been to refurbish already existing code to work in a slightly different manner. This meant that a deep understanding of the existing code was needed, to understand how it had to be modified and what could be reused in which instances.

5.3 Sources

The scientific sources were found by using Google scholar which of course only is a search engine which doesn't guarantee any sort of reliable sources. So it was checked that basically all the sources that are used have been published in some sort of scientific journal or presented on some sort of conference which means that they have been peer reviewed. This is some sort of guarantee of reliability. Any proof that Prenskys M [1] and Mikkonen, T., & Taivalsaari [5] has been peer reviewed wasn't found. However Prensky M [1] has been cited in 4081 other articles which were interpret as a sort of guarantee of reliability and the facts that are used from Mikkonen, T., & Taivalsaari [5] are quite well known facts of JavaScript. They have been cited in 39 other reports while not a very large number it's not that small either. The source about Phaser is its own website which were assumed that its reliable regarding the type of features that Phaser has which is what is used from the site.

5.4 Ethics

In this project a part of a math game for children has been built, this can have a positive effect on today's childrens ability to learn math, as stated by Prensky, M [1] today's children could learn better from, e.g. games than older ways of learning math. However it could also have a negative effect on children by making them even more dependable on computers and making them ignore traditional Medias such as books. This report could help future programmers on this project or similar projects regarding how to build these kind of systems and how to design feedback.

6. Conclusion

The purpose of this thesis work was to extend the existing framework of magical garden to accommodate the needs of one of the subgames within Magical Island as well as to implement the shark game from MI. This has been achieved through a systematic approach where one feature at a time was implemented. This project is the start of the implementation of Magical Island in JavaScript which will be continued by future programmers. Features such as the drag and drop which are used in other of the subgames in the prototype for Magical Island can then be used by the future developers. Future development can also follow my implementation of visual feedback which is mostly based on the work of Blair, K.P. [2] as an implementation to model from in future sub games.

The answer to my first question is the following: A big part of extending an existing framework is to understand the code, much of what one does is copying and refurbishing code which is the actual hard part. New code will have to bee written but if one understands the framework this will most likely not be that hard especially in a framework as big as this one. Because so many features already are implemented the programmer can take advantage of them. The specific approach that was used is described in the method chapter. The answer to my second question is the following: From the articles read it was deemed that implication feedback would be the most appropriate feedback for this kind of game based on the test results from Blair, K. P [2] and Paek, S et al [4]. Then a way to convert this concept to the shark game through placing the *Cookies* in different way to show the player the implication of their choices was designed. The exact implementation can be found and is discussed in the results and discussion chapters.

6.2 future work

If there was more time the game would be more polished and animation for the *Cookies* and some way to indicate that the whole cookie should be placed on the shark would have been implemented. The *dropPlace* for the drag object would also be fixed. Instead of having a fixed size to the drop area it would get it from the *goalObjects* sprite size. Furthermore the sprite sheets for the drag objects would be defined in the specific objects instead of in the parent constructor and the sound file would be compressed so that the game doesn't crash on Ipad.

7. References

[1] Prensky, M. (2005). Computer games and learning: Digital game-based learning. *Handbook of computer game studies*, *18*, 97-122.

[2] Blair, K. P. (2013, June). Learning in critter corral: evaluating three kinds of feedback in a preschool math app. In *Proceedings of the 12th International Conference on Interaction Design and Children* (pp. 372-375). ACM.

[3] Moreno, R., & Mayer, R. (2007). Interactive multimodal learning environments. *Educational Psychology Review*, *19*(3), 309-326.

[4] Paek, S., Hoffman, D., Saravanos, A., Black, J., & Kinzer, C. (2011, May). The role of modality in virtual manipulative design. In *CHI'11 Extended Abstracts on Human Factors in Computing Systems* (pp. 1747-1752). ACM.

[5] Mikkonen, T., & Taivalsaari, A. (2007). Using JavaScript as a real programming language.

[6] Davey R. 2015. Phaser Features. https://phaser.io/ (2015-04-13)

[7] Stanford University. 2013. Pre-school mathematics: Critter Corral. http://aaalab.stanford.edu/research/stem-builders/pre-school-mathematics-critter-corral/ (2015-04-04)