The Giving Child

Aleksandra

6/17/2014

Walter Schlosser, Danny Victor, Kelly Masuda, Ben Casey

Introduction

Everyone knows the saying “Give a man a fish, you feed him for a day, but teach a man to fish, and he can eat for a lifetime”. With this field session project, The Giving Child (TGC) looks to create an Android game that can be marketed to Heifer International, and sold through them to send livestock (specifically geese) to impoverished areas of the world. Many children learn about the issues faced by impoverished communities around the world, but feel unable to do anything to help. The Giving Child’s mission is to spread awareness about issues children around the world face, such as finding clean water, and how anyone can make a difference, no matter how big or grown-up that person can to make these problems a part of the past.

Product Vision:

The story proposed surrounds a young girl who is able to go from “rags to riches” after receiving flock of geese. The game is intended to be played primarily by children, 8 to 14, to teach them how something as simple as a flock of geese can change someone’s life. The game involves herding and managing a herd of animals in order to bring them to a market, sell them, and buy animal upgrades or boosts at the shop. Over time, the main character progresses through different more valuable animals which are harder to maintain until she is able to purchase a book and pursue education. The player must herd the flock to keep it from escaping, and collect the flock’s respective goods (eggs from geese, milk from cows). The goods can be sold at markets to earn money for the shop. The end goal represents coming out of poverty by getting an education.

Through storytelling elements placed throughout the app, we inform the players of how getting a flock of animals makes a huge difference in quality of life for these communities. Also, the story promotes Heifer International’s agenda, which involves the donation of animals to families in third world countries to promote the overall welfare.

We created different levels that are fun for any age group which show different animals. The difficulty of the game begins easy enough to be played by any age on level 1, while the end level difficulty is engaging for all ages.

Requirements

The requests for the game made by TGC were separated into two categories, functional and non-functional. Each functional requirement involves a game design detail that was to be implemented while each non-functional requirement refers to the end goals for the application as a whole. Below are both categories of requirements.

Functional Requirements:

- **Moving animals**: Animals must maintain semi-random motion to provide a challenge for herding

- **Drag to influence animal movement (herding + tapping)**: Screen swipes must influence the animal motion in order to allow the player to keep the group on screen, taps make it easier for users to retrieve animals near the edge of the screen
- **Health must decrease with time:** As the player progresses down the lane, the main character should get hungry and health should decrease
- **Animals produce marketable goods (eggs, bacon, cheese, mutton, milk):** Each animal must occasionally create a consumable good to be collected by the player
- **Collect goods to an inventory:** Collected goods should be available for viewing and managing in an inventory interface
- **Eat collected good to restore health:** Collected goods should be dragged to the character for her to eat and restore health
- **Ability to increase animal count:** Some consumables may produce a new animal (such as geese eggs) instead of adding an item to sell for money in the market
- **Visit a market area:** The player will arrive at a market with its own screen, where collected items can be sold, and current animals can be swapped for the next level's herd
- **Sell collected goods:** The player should be able to sell collected consumables for money which is used to purchase upgrades for the users animals
- **Swap animals:** The market should allow the player to exchange the current levels animal for the next levels in order to progress
- **Money counter:** A visible counter should display the current amount of money
- **Lose if all animals escape:** Protect your assets! Can replay the last stage
- **Lose if health drops too low:** If the player has not fed the main character, she will be too tired to continue
- **Weather elements:** Not every day is going to be sunny…
- **Intuitive touch menus:** User can easily navigate menu functions
- **End goal:** The player is moving towards the purchase of a book which occurs at the end of the cow level
- **Kid Mode:** There should be a mode for kids to play which decreases the difficulty for everything to encompass younger than 8 year olds.

Non-Functional Requirements:

- **Promote charity:** App will come with the lowest tier donation to Heifer International
- **Low downtime:** Engage the user through the entirety of gameplay
- **Easy extension:** Allow the client to add more to the game
- **Minimum requirement is GingerBread (ver 2.3)**
- **Child friendly:** Vocabulary and transitions must be simple
- **User testing with children:** Children will be testing how "fun" the game is
- **Publish on google Play store:** App must be published at end of field session

Risks:

- Only two team members know Android
- Only one team member knows how to use a game engine
- Limitations of the game engine
- Communication failure between team members and third party collaborators
- Art and sound asset quality

System Architecture

In addition to the technical issues, our team faced various game design issues. We had to design the screenflow, which shows how the game flows through the screens. We had to put thought into level progression, how the players would go between levels and progress in the game. We also had to decide on the losing conditions, because without being able to lose, there's much less difficulty. Our team provided the following solutions to these issues.

Screenflow

Figure 1 displays the screenflow that progresses during gameplay. The conditions for screen transition are also shown as well as a rough idea of which buttons will be provided. The pause and resume functions along with the back button will save the current state and resume it for an uninterrupted flow through the game.

Control gestures

The tap on screen will either pick up the consumable or spawn another animal at the respective location in order to collect them while herding animals as well. Tapping will also herd individual animals, which is especially useful if your animals are near the edge of the screen. Herding animals will be implemented with one finger drag. Temporary barriers will be created by each completed drag and will influence animal motion.

Level progression

We decided that the player will progress to the market after a set amount of distance is traveled. The player must keep the animals on screen until reaching the next village and herding them into the market. For each level, more animals must be herded into the market in order to advance to the next level. If not enough animals reach the market, the player is able to leave their current animals in the market, and retry the level again with new animals. Initially, the progression of animals for each level will consist of geese, pigs, goats, sheep, and then cows. The corresponding consumables will be eggs, bacon, cheese, mutton, and milk.

Losing conditions

We decided that once the player loses all of their current animals from unsuccessful herding or Aleksandra is too hungry to continue, the losing screen will be presented and allow the player to exit to the main menu or retry from the beginning of the current level. Inventory items, money, and animals are reset to the quantities the player had at the beginning of the current level.

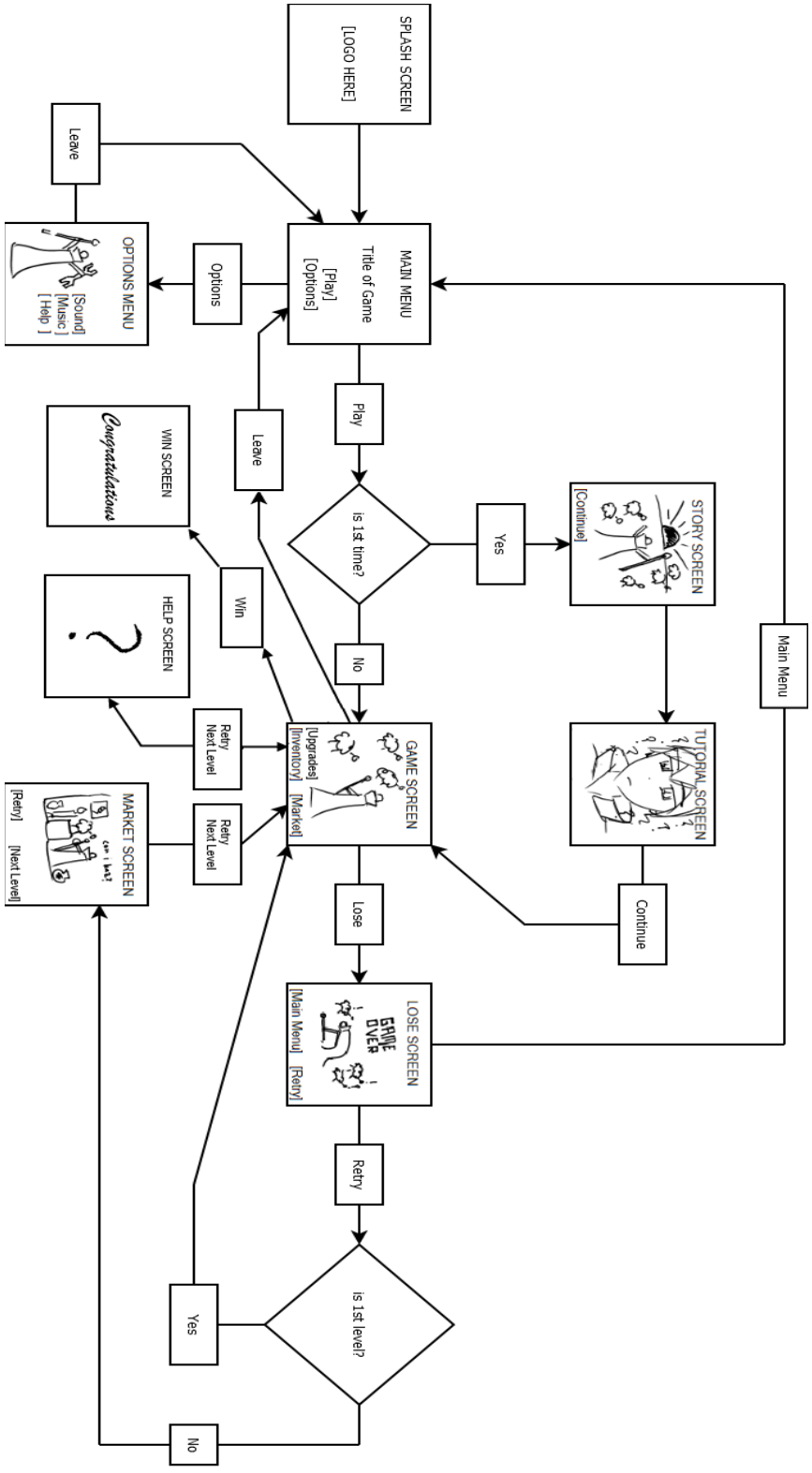
SPLASH SCREEN
[LOGO HERE]
MAIN MENU
Title of Game
[Play]
[Options]
Leave
OPTIONS MENU
[Sound]
[Music]
[Help]
Options
Play
is 1st time?
Yes
STORY SCREEN
[Continue]
No
Leave
WIN SCREEN
Congratulations
Win
HELP SCREEN
Retry
Next Level
GAME SCREEN
[Upgrades] [Inventory] [Market]
Main Menu
TUTORIAL SCREEN
Continue
MARKET SCREEN
[Retry] [Next Level]
Retry
Next Level
Lose
LOSE SCREEN
GAME OVER
[Main Menu] [Retry]
Retry
is 1st level?
Yes
No

Figure 1

Design and Implementation Decisions

After the high level requirement solutions were agreed upon, we began making low level technical decisions to begin the development process. It was important for us to all be on the same page with the design decisions so that we would not write conflicting implementations.

The technical decisions made include:

- Game engine
- Class hierarchy
- Singleton objects
- Application performance
- Memory management
- Saved game storage

Game engine

Our search for an appropriate game engine resulted in an evaluation shown in Figure 2. We rated each engine on a scale from 1 to 5 based on 2D rendering capabilities, gesture recognition, ease of export to android, and the developer support for the engine. All engines except for CatCake were fully capable with 2D graphics, whereas CatCake is geared towards 3D rendering. CatCake also had no gesture recognition support. The gesture recognition in *libGDX* proved to be the most robust and easy to implement. The ability to easily export to android platforms was an important feature for this project. All engines except for Phaser allowed easy export to an android .apk file for distribution. The last criteria we rated was the ongoing support for each engine. The Angle engine lost support in 2010, while the Rokon engine was used as a base for *libGDX* and is no longer used. *libGDX* has an active support group as well as many extensions for features such as preferences, physics, and camera. The clear choice for this project is *libGDX* based on the table below. *libGDX* also corresponds well to our current development skills in Java.

	2D Rendering Capabilities	Gesture Recognition	Android Exportability	Support	**Total**
CatCake	1	1	5	5	**12**
Phaser	5	4	1	5	**15**
AndEngine	3	3	5	3	**14**
libGDX	5	5	5	5	**20**
Rokon	3	3	5	1	**12**
Angle	3	2	5	1	**11**

**Categories are rated on a 1-5 scale with 5 corresponding to the best score*
Figure 2

There are a lot of good qualities in *libGDX* that make it really nice for Aleksandra. *libGDX* is a cross platform engine that provides export to desktop, android platforms, iOS, as well as HTML. While the game is for Android, it allows for greater reach for the future. The desktop version is easier for coding and testing. Also, we don't need to focus on platform specific coding practices. In addition to cross platform, the engine provides convenient functions and classes for game design including gesture recognition predefined interfaces. The *Game* class is provided by *libGDX* and provides easy reference to the running instance of the application at all times, while the *Screen* interface encapsulates the necessary functions for managing the app lifecycle on multiple platforms. Our design is based around separating functionality between screens and their member variables, and switching the *Game* class screen on button presses. *libGDX* also provides the interface *InputProcessor* which encapsulates all of the necessary functions for detecting touch and drag. The interface is easy to use and allows us to implement different controls for different screens as well.

Class Hierarchy

To develop clean and uncoupled code for game objects, we designed the class hierarchy shown in Figure 3. The main functionality we required for game objects was the ability to draw and move them. For this reason, the highest up class is *ABDrawable,* and *Movable* inherits from it. In this way, we could easily separate objects which must move from objects which are only drawn. Objects which are only drawn include the market, dropped items, and obstacles. *Movable* is the parent class for the *Player*, *Animal*, and *weather object* classes since these objects must be able to move and interact. Each type of animal was created as a child class of *Animal* and therefore allows easy addition of new animals, and easy reference to animals in the rest of the design.

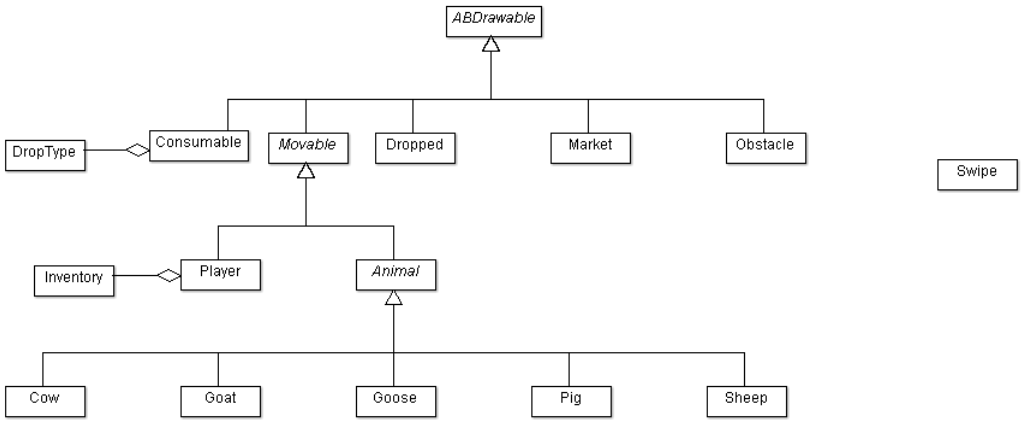

Figure 3

Gameplay and logic responsibility was separated into the classes *GameScreen, World,* and *WorldRenderer* as shown in Figure 4. During gameplay, the *GameScreen* class displays the background and the buttons, and then hands off duties to *World*. *World* is responsible for game logic, and therefore has the references to all animals, boosts, weather, and dropped items. During the render loop, *World* updates logic for lost animals, duration of boosts, type of weather, animals colliding with the market, and colliding with obstacles. *World* then hands all objects that need to be drawn to *WorldRenderer* which iterates and draws accordingly. *GameScreenInputHandler* is created in *GameScreen* and is an implementation of *InputProcessor* which correctly maps gestures to functions on game objects for gameplay. In this way, game logic and controls are successfully decoupled and create easy code to work with.

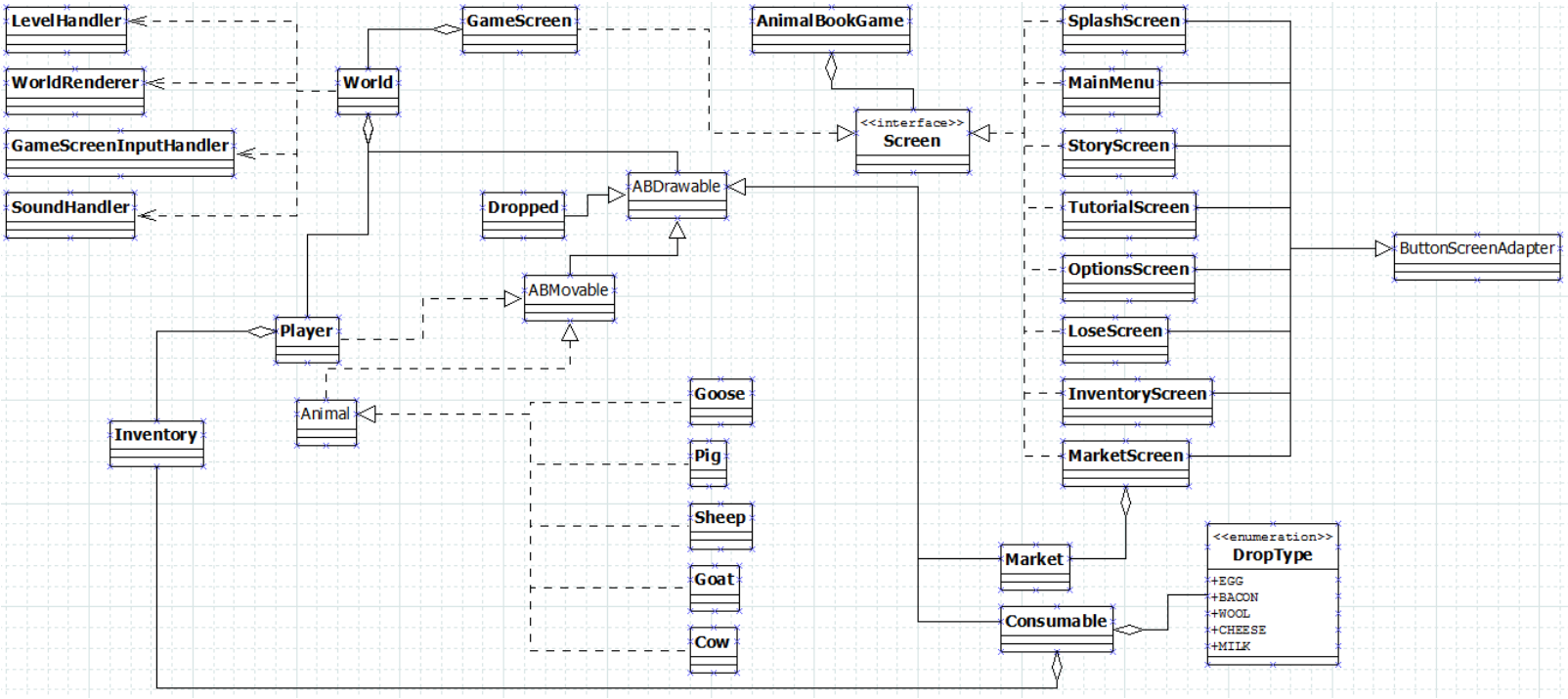

Figure 4

Singleton Objects

During the development process, we discovered several memory leaks that were impacting the ability to play the game. This was due to creating a new instance of a screen every time the player swapped between screens. We solved this problem by using a singleton pattern in our design. A singleton pattern is a design pattern that restricts the instantiation of a class to only one object. By limiting our game to only contain a single instance of each game screen, and changing the properties of this screen as needed, we cut down on the memory usage of the application to a much more reasonable amount. The game is now able to run to completion even on an older phone; whereas before, it was crashing after 1 or 2 levels on a relatively new tablet.

Application Performance

Texture resolutions were decreased where necessary to maintain acceptable frame rate. All move functions were also implemented to take a *delta* parameter which corresponds to the time since the last frame. In this way, all movement is scaled to be frame rate independent. Without this independence, object speeds would appear to slow down when the frame rate dropped due to the decreased frequency of running the game logic and rendering loop. The target average frame rate for the game is 30 FPS, which we feel is a realistic goal for a variety of devices. In order to keep the frame rate acceptable, all drawn objects have a texture instead of using *libGDX*'s *ShapeRenderer* class for primitive shapes. *ShapeRenderer* has a large toll on frame rate, and so we have chosen to avoid all primitive shape drawing.
Memory management

The *libGDX* library provides texture initialization and disposal through its *Texture* class. Textures tend to take up space in memory, and will quickly create memory errors if not disposed of once they are no longer needed. For this reason, all of our classes which have a texture also have a *dispose()* function which correctly frees up memory for the rest of the application. This prevents the application from quickly losing control of its memory usage. When an object texture is needed again, the textures are reinitialized for use. The reinitialization process is fast and we have not run into the need for an asset manager or loading screen.

Saved game storage

The game will need to be stored outside of the cache of the application because people will want to pick up where they left off when they switch to a different app. Android keeps the app data in memory cache when running or when paused. However, if the phone needs more memory for another process, then the apps data that is in cache gets deleted by calling onDestroy(). Also, in order to keep the app in cache, it needs to be running, which users do not like, nor does the battery. To fix this problem, we have decided to store data outside of cache when the app calls onDestroy(). Figure 5 displays the life cycle of the app corresponding to the flow of data.

There are 2 ways this could be done: a private Database or preferences through *libGDX*. Preferences is cross-platform with iOS, Android, and Desktop and should only store primitive data. It should not be used for user data because other apps can have access to the *SharedPreferences*. In *libGDX*, there is an extension for Database that will run on Desktop of version of the game. Without the extension, we would have to code each database interaction separately on each device. The database uses *Sqlite3* and will store itself upon creation at a specified location. The extension has to have 4 JAR files within the project and added to the build path for the game to run. There is very little documentation available for this method of storing.

The team decided to use Preferences for several reasons. First, it is completely cross-platform. If the clients want to distribute on iOS, they can. Second, the data is primitive user data which will only be in one row. The data is in Figure 6. There is no need for multiple rows because there is only one user with no way of getting to previous levels or states.

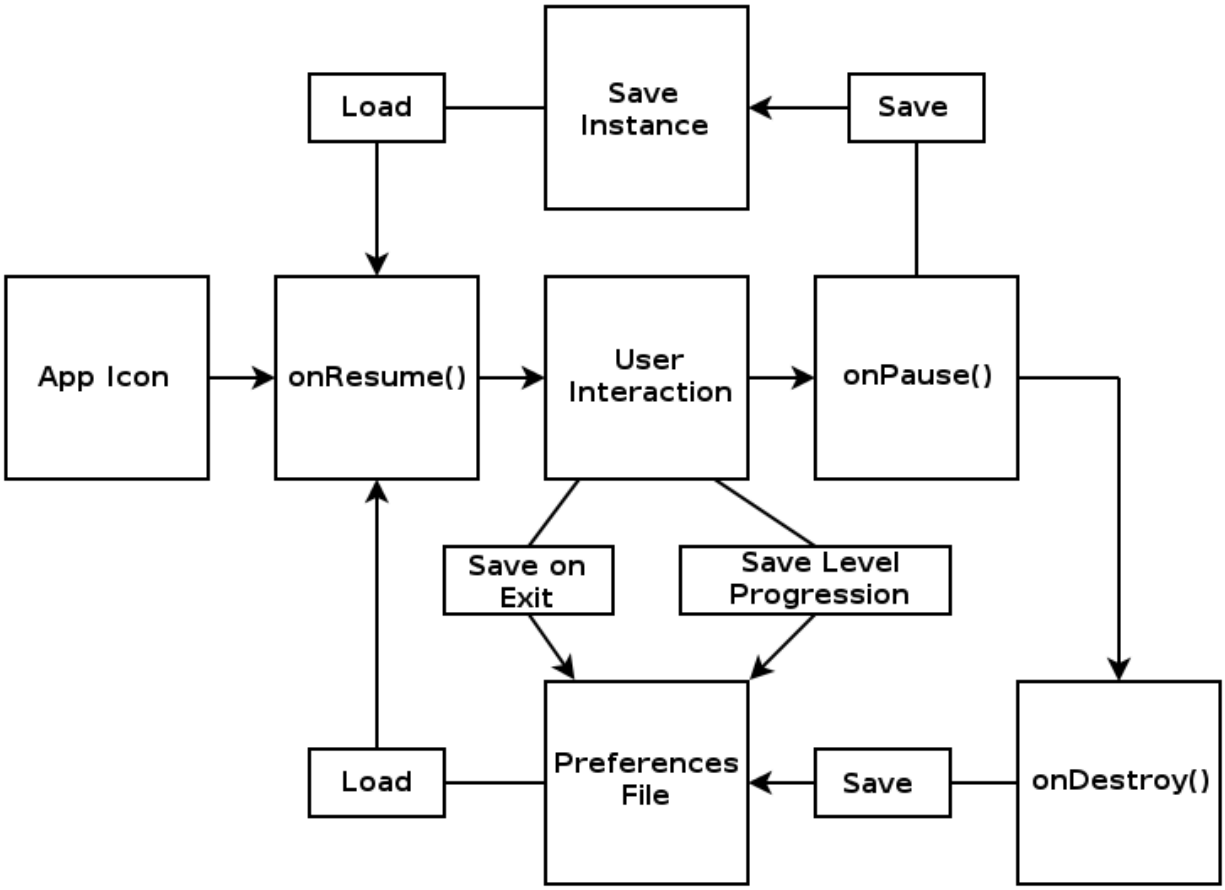

Figure 5

Figure 6 displays the information contained in the preferences file.

Money	num of Animals	Health	Egg	Bacon	Cheese	Mutton	Level	...
int	int	float	int	int	int	int	int	...

Sound	Music	fruitfulness	DropInterval	Duration	tutorial
boolean	boolean	int	int	int	boolean

Figure 6

Results

It is important in game design to receive a constant flow of feedback from the end user in order to create a fun experience. We made sure to assess our results with user testing at about three quarters through the development period in order to leave ourselves time to implement fixes. After we implemented changes, we assessed our progress in technical design areas as well. There were four areas that we used to assess our development success which included:

- The game we were able to produce
- User testing
- Performance testing
- Features we did not have time to implement

Game

We created five levels to be built off of in the future which increase in difficulty. TGC wanted the game to primarily appeal to a younger audience, 8 to 14, but to also be fun for any age group. The difficulty of the game begins easy enough to be played by any age on level 1, while the end level difficulty is engaging for all ages. There is also a kid mode option which reduces the difficulty of all in game features to allow younger players to enjoy the game. In this way, we have provided a span of difficulty which TGC can adjust as they feel is needed.

We implemented a story screen at the beginning of the first level which sets up why she is herding. The story screens are easy to adjust to be used as needed by TGC to tell their story. The design of the game also makes it easy to add additional levels and to tweak difficulty of the available levels. In addition to this, the design also allows easy addition and modification of sounds and game textures.

The gameplay matches the style that TGC described to us in an early meeting. The camera moves up the screen on its own while animals move around with which the player interacts. The player has to be weary of the change in weather elements and pay heed to his/her health. The player maintains their animal herd through swipes and taps which are meant to be intuitive. The design also allows easy modification of the gesture sensitivity and the reaction animals have to each gesture.

Overall, we have touched on every major need of the client, and provided a solid game design that can be adjusted and expanded to entice consumers to donate to Heifer International just for the game.

User Testing

An important aspect of this project was to obtain player feedback since the goal is to provide a fun experience for the end user. We drafted questions we found relevant to the game experience and got five users which ranged in age to play and provide feedback. Figure 7 displays a table of the results that involved a numerical rating.

The categories in Figure 7 were rated from 1 to 10 points with 10 being the best score. The difficulty category was scored higher when the user felt the difficulty was just right. Lower scores in this category corresponded to gameplay that was too hard or too easy. Performance related to the actual performance of the game. Points were deducted for frame rate drops or a crash. Controls received points when the user felt the game was responsive, intuition scored points when the user needed little instruction to understand how to play, and finally we asked each user for an overall score of the game's fun factor. Scored out of 50 possible points, the game received an average score of 35.75, which corresponds to a 71.5%.

Age:	6**	11	17	19	50
Difficulty	good	6	8	8	4
Performance	10	10	8	7	7
Controls	5	8	8	8	2
Intuitive	7	10	6	8	7
Fun	yes	7	7	6	8
Total	N/A	41	37	37	28

**Categories are rated from 1 to 10 with 10 being the best*
***At 6 years old, we did not expect the user to provide a numerical rating. Numerical answers were estimated by the tester*

Figure 7

In addition to scores, users provided feedback about their interest in downloading and playing the game again, as well as features they would like to see or change. Each user said they would likely download the game and play again without us asking except for the 6 year old. The 6 year old lost interest in the game much earlier than the other testers, and was not able to recall the name, which indicated disinterest to us.

With respect to gameplay, the 6 year old was inclined to continuously swipe. We interpreted this as a need to reduce downtime to engage the young users. The older testers were engaged enough to play through all five levels we had implemented. However, none of the users were interested in using the animal upgrades feature. As a result, the use of upgrades will have to become more necessary or be revamped in order to interest the user.

The older testers explained that they would have liked more levels as well as slight tweaks to the harder levels' difficulty. Some users had difficulty stopping their animals from running off of the top of the screen in the later levels which would cause some frustration. To respond to this concern, we implemented a dog item as a buyable boost which helps the player keep the animals from running off the top.

Performance Testing

The game utilizes approximately 38% of the CPU and uses about 100 MB of RAM during gameplay as of version 0.7. Older devices might not have enough memory for this app, like the Samsung Galaxy S. The entire app is contained in about 40 MB of storage at this time. In comparison to Clash of Clans, a popular phone game, our game appears to be right in the sweet spot for typical Android hardware. Clash of Clans utilizes 50% of the CPU, about 150 MB of memory, and about 50 MB of storage.

Features that were not implemented

The first feature we did not have time to implement was the story screens between levels. The storyboard material provided was not far enough along for us to add them to the game. We also did not have the time to implement animations for animal movements. Luckily, the top down nature of the game makes it visually appealing without the addition of animation.

We would have liked to add and tweak more levels. More levels would produce a smoother gradient in difficulty. Also, the length of the game needs to increase since currently the entire game can be finished in approximately 30 minutes. In addition to these concerns, we would have liked to address the gameplay downtime with a few more events that animals would react to. This would have engaged the user more and possibly fix the problem with very young users losing interest.

Another feature that was not implemented was Google Play Achievements. The achievements would have corresponded to understanding the game, incredible feats for each level, and overall feats for the game. The Achievements would make Aleksandra be entertaining for groups of people via discussing their feats.

On the development side, we did not have the time to produce a robust test suite for the code. We did not keep up with developing tests along the way, and as a result, we have ended up with a majority of the code being untested. Although the design provides easy modification, the lack of a testing suite could make future collaborators uncomfortable.

The user testing average score of 71% confirmed to us that we produced a good base for an engaging game. We are happy with the product our development process produced, and hope to see it expanded upon in the future.

Appendix

Adjusting screen flow:

- Screen classes in the com.tgco.animalBook.screens package all instantiate their own button listeners. Example screen switch syntax is as follows:

```
playButton.addListener(new InputListener() {
        public boolean touchDown (InputEvent event, float x, float y, int pointer, int button) {
                return true;
        }

        public void touchUp (InputEvent event, float x, float y, int pointer, int button) {
                gameInstance.setScreen(new GameScreen(gameInstance));
                dispose();
                return true;
        }
});
```

  Adjusting the functionality of *touchUp* can switch to any screen desired for the button press. A return true statement indicates the event was successfully handled by this function and does not need to be passed to other listeners. The *dispose* method belongs to the screen that has *playButton*, and must be used to call *dispose* on all objects that use large amount of memory, such as textures. Failure to dispose will leak memory and cause a crash if repeatedly done.

Adding levels

- Create new class extending animal and add a new value to the DropType enum as desired
- Add new case for switch statements in LevelHandler
- Change the return value for getMaxLevel() in LevelHandler

Adjusting levels

- All variables that determine a level's difficulty are returned from the class *LevelHandler.* Functions such as *returnCameraSpeed* and *returnLaneLength* are called when needed, take the current level as a parameter, and can be adjusted in *LevelHandler* as needed. Functions titled similarly to *addObstacles* and *addAnimals* are also called as needed and return the array of obstacles and animals that correspond to the current level. Difficulty adjustments or a change of the level's animal can be made by adjusting these functions.

How to setup the environment

- Download Eclipse + ADT Bundle: http://developer.android.com/sdk/index.html

- Extract contents to program directory. Run eclipse and select Github directory for workbench.
- Go to Help>Install New Software... Then enter the following in the "Work with: " field: http://dist.springsource.com/release/TOOLS/gradle Install the Gradle IDE (under Extensions / Gradle Integration)
- Pull animalBook-Game from Github into Github directory. DO NOT pull "local.properties" file. You will get an error. Clone URL: https://github.com/tgco/animalBook.git
- Go to eclipse, select import (By right clicking package explorer window), select import gradle project, select animalBook folder in Github, the click on "Build Model". This will take a while.

A few errors you may experience in setup:

1) SDK location not found. When trying to build model for Gradle. This is because there is no local.properties in the code that was pulled from github. Simply add in a file to the main project folder with this inside: sdk.dir=Path/To/Your/SDK. Save this file as local.properties.
DO NOT add the file to github.

2) Errors in DatabaseHandler. This is because DatabaseHandler is dependent on an external library. There is a jar file in the core folder that needs to be added to the Build Path in order for the DatabaseHandler to work.

- The animalBook_Game-core contains all of the main code.
- To run android, launch from animalBook_Game-android, to launch desktop, launch animalBook_Game-desktop, etc.
- Every time you add/edit new resources, make sure to refresh each folder for the changes to show.
- Make sure to add the libraries to build path in android and core.

For more information or if you run into errors, refer to this site:
https://github.com/libGDX/libGDX/wiki/Setting-up-your-Development-Environment-%28Eclipse%2C-Intellij-IDEA%2C-NetBeans%29

Enabling/Disabling Specific Weather Elements

- *Go to Weather.java*
- *Comment/Uncomment the specific weather change*
- *You can add another WeatherType if you wish*
    - *Add the Enum*
    - *Depending on the weather, you can implement weather visuals by copying and pasting one of the pre-existing code for smooth weather transition*

- *Then add what weather visuals you desire in WorldRenderer (i.e. a thunderstorm across the screen!)*

Adding/Adjusting sounds

- Adding Sounds
  - Place the sound file (.wav works best, .ogg and .mp3 can also be used) in the android/assets/sounds folder
  - Add the definition of the Sound

```
Sound newSound = new Sound(Gdx.files.internal("sounds/soundFileName.wav"));
```

  - Then add in a play function for the sound:

```
public static void playNewSound() {
        if (soundMuted) {
                return;
        } else {
                newSound.play(.4f);
        }
}
```

    The parameter for Sound.play() is a float from 0 to 1 determining the relative volume of the sound being played.

- Adding Music
  - Place the music file (.wav works best, .ogg and .mp3 can also be used) in the android/assets/sounds folder
  - Add the definition of the Music

```
Music newMusic = new Music(Gdx.files.internal("sounds/musicFileName.wav"));
```

  - Then add in play and pause functions for the music

```
public static void playNewMusic(boolean isLooping) {
        if (musicMuted) {
                return;
        } else {
                newMusic.setLooping(isLooping);
                newMusic.setVolume(0.35f);
                newMusic.play();
        }
}
```

    The isLooping boolean determines if the music should be repeated once it ends, or just played once. setVolume functions the same as the parameter for .play()

```
public static void pauseNewMusic() {
        newMusic.pause();
```

```
}
```

- Then add in to toggleMusic statements to mute the music and to reset it to the correct audio level

- Lastly, add in the reinitialization line to resetAudio, and call newSound.dispose() or newMusic.dispose() in the SoundHandler's dispose function.

Saving new data

- In game saving
  - There is an array of objects that is stored in AnimalBook Game called levelData, which is all the data for that level that is being played. To add more data there, add 1 to the array constructor and insert 1 more null in the for loop for onCreate().
  - To save that data, there are 2 spots that need to change, 1) keyUp in GameScreenInputHandler and touchUp of MainMenuButton in the function of initializeOptionItems() in the GameScreen class. Add in what you want to save in your new spot from LevelData.

```
public void touchUp (InputEvent event, float x, float y, int pointer, int button) {
                        SoundHandler.playButtonClick();
                        gameInstance.setHitBack(true);
                        gameInstance.setScreen(new MainMenuScreen(gameInstance));
                        //store the data in levelData of Game

                        // spot 1 is current level
                        gameInstance.addToDatalevel(gameInstance.getLevelHandler().getLevel(),0);

                        //spot 2 is player
gameInstance.addToDatalevel(gameWorld.getPlayer(),1);

                        //spot 3 is storing movable array
 gameInstance.addToDatalevel(gameWorld.getMovables(),2);

                        //spot 4 is storing dropped items array
 gameInstance.addToDatalevel(gameWorld.getDropped(), 3);

 gameInstance.addToDatalevel(gameWorld.getObstacles(), 4);
 gameInstance.setProgPercentage(gameWorld.getPrecentage());

                        dispose();
 }
```

- Leaving game saving
  - We use LibGDx's Preferences class to store the values. There are level data and settings that need to be stored. Anything outside of level stuff needs to

be called in the OnPause() to store and OnCreate() to load. The levelData gets saved in setPrefsToFileLevelChange() and setPrefsToFile() functions. Add there what you need to by calling similar functions, such as:

```
prefs.putInteger("Longer", levelHandler.getLongerMoneyP());
```

- Loading happens in setDataRetry() and setDataCont() based on wheather to load the animals with setDataCont or load the basic data from retying from the market. Add there what is needed such as:

```
levelHandler.setNextLevelStart(prefs.getInteger("numAnimals"));
```