Project Number: RL1-P110

genometry

Using Smartphones to Control Interactive Content on Public Displays

A Major Qualifying Project Report
submitted to the faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Bachelor of Science
by:

Chris Chung, Khoa DoBa, Jared Hays, Jared Ingalls,
Sarah Jaffer, Paul Ksiazek, Elizabeth Labelle

Professor Robert W. Lindeman, Major Advisor

Professor Mark Claypool, Major Advisor

Abstract

for

genometry: using smartphones to control interactive content on public displays

by

Chris Chung, Khoa DoBa, Jared Hays, Jared Ingalls,
Sarah Jaffer, Paul Ksiazek, Elizabeth Labelle

In order to encourage cooperation and communication between Worcester Polytechnic Institute and Osaka University, as well as to explore potential new applications of public displays and mobile devices, a product was designed to allow users to participate in an online, collaborative environment displayed on public monitors located on both campuses, in which each user controls a virtual pet creature. Users are able to connect to the public displays via Web-enabled cell phones and PDAs, which they use to guide and instruct their creatures to explore the environment and interact with each other. The environment state is maintained on a remote server, which handles all creature and environment updates as well as long-term data management. World information is propagated to the terminal displays, which run a Flash application to draw the environment and the creatures.

Via their creatures, users are able to explore, chat, and trade items with each other. Users can also collect items for their creatures to wear, by trading and finding treasures in the environment. As creatures interact with one another, their physical appearance will change over time to reflect their behaviors.

Acknowledgements

The genometry team would like to extend its thanks to everyone at Worcester Polytechnic Institute and Osaka University whose assistance and advice made this collaboration possible: Professors Lindeman and Claypool at WPI, and Professors Takemura, Kiyokawa, Itoh, and Andou, as well as everyone at the Takemura Laboratory. Additionally, we would like to thank the staff at the IGSD and the Admini Corporation for facilitating our stay in Japan and helping to make the project a success.

Table of Contents

List of Figures

List of Tables

1 Project Overview

The genometry project is an online, virtual environment displayed on public monitors located at Worcester Polytechnic Institute (WPI) and Osaka University (OU), in which users employ smartphones to guide personalized creatures and interact with each other. As users explore the environment and communicate with one another, their creatures gain new items and even change their appearance over time.

1.1 Final Product Description

The goal of the project was to design a virtual world which could be shown on a number of public displays. The public displays would be networked together to show the same environment. Users with smart mobile devices would be able to obtain a personalized creature character, which existed in the environment shown on the public displays. Users could then control their creatures using their mobile device to interact with creatures owned by other users, even the creatures owned by users in other countries. Figure 1 is a photo of the final demonstration given on Osaka University's O+PUS system of networked public displays, on Wed October 6, 2010.

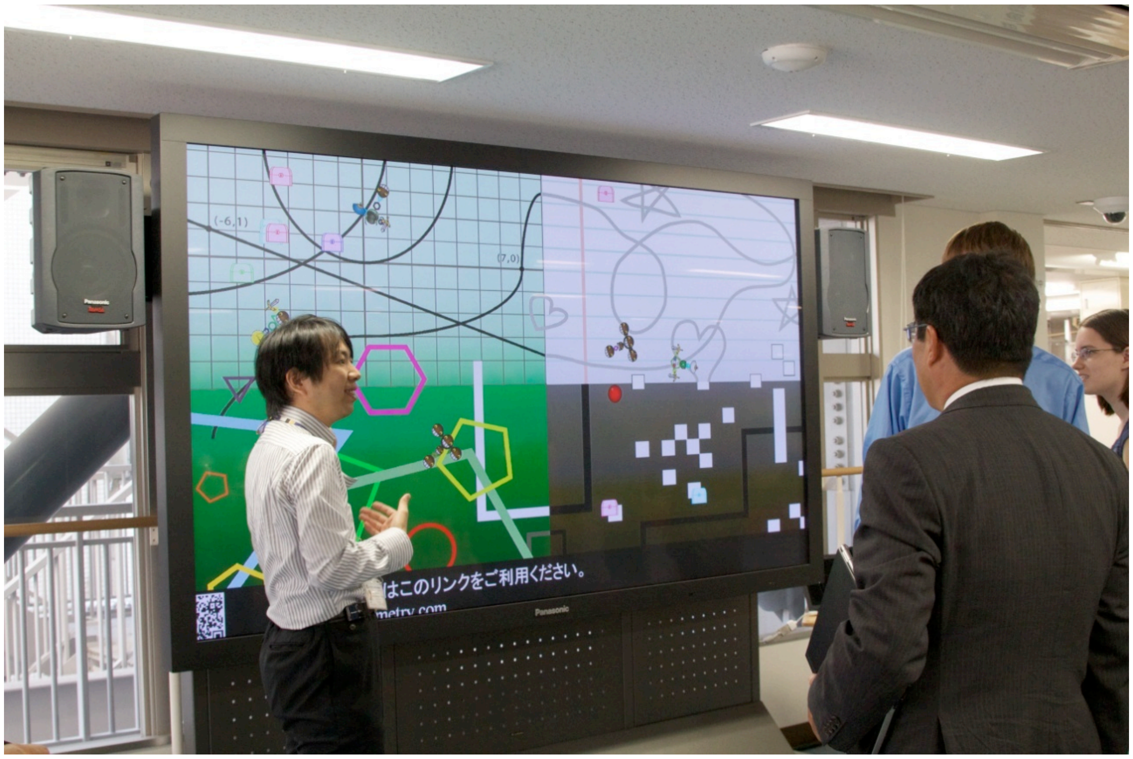


Figure 1 - The final application shown on a public display at Osaka University.

We wanted the system to encourage communication and interaction between users. Therefore, many of the actions a user can take within the environment are social in nature. A user can chat, trade, and friend other users in the environment. Since there is a language barrier, the mobile device applications fully support both Japanese and English text. The chat system automatically translates messages between users, so that users who do not speak the same language can interact.

The creatures owned by each user are personalized. Every creature is made up of eight parts. Each part can be one of five shapes and any color. Creatures are given randomized initial colors, giving creatures widely varying appearances. Additionally, treasure chests are scattered throughout the environment. Treasure chests contain items, which can be equipped by creatures to further change their appearance. Creatures have an inventory to manage items, and a catalog to view which items the user has found, and which they have not.

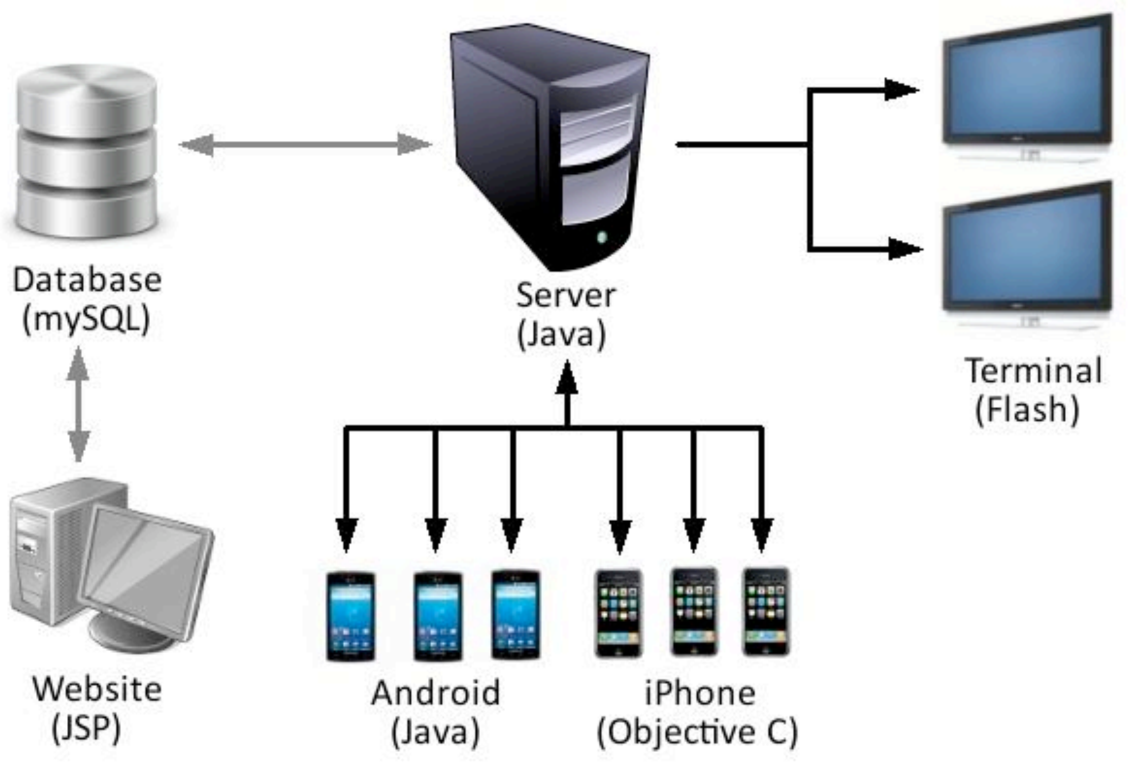


Figure 2 - Diagram of components in the system and their relationships

The architecture required to run the system needed to handle multiple kinds of devices, many of which needed to be written in different programming languages. The system consists of a centralized server which manages data, a database to store data, a Website for account creation, a Flash application to render content on the public displays, and supports four types of devices for user input. Users can control their creatures with an Android phone, an iPod Touch, an iPhone, or an iPad. Communication between each part of the system is handled through XML messages, which are standardized across all components and sent over TCP connections. Standardizing the communication allows for greater extensibility in the system. New components can be added without needing to modify the existing system, as long as they have some form of Internet connectivity.

1.2 User Experience

The system allows users to perform a number of discrete actions via their mobile devices. In general, the only prerequisites for performing an action are being logged in, and in some cases having a second user to interact with; however, some activities require the user to be in front of the display, in order to see their creature.

When a potential user is in front of the display, one of the things they will notice is the new user greeting, located in the lower left hand corner of the display in an information bar. It invites prospective users to sign up, presenting them with both a data matrix (2-dimensional barcode, like in Figure 3) and a URL, for those whose mobile devices cannot process 2D barcodes. Both the barcode and the URL will direct the user to the same Webpage, which will begin the signup process.

Figure 3 - An example of a 2D barcode

The Website ideally contains instructions for the user to follow a link to the App Store (for Apple products) or the Market (for Androids) to download the application and then return to the site. Unfortunately, the application was never formally submitted and reviewed, and so the application download link and description were removed from the Website to avoid confusion.

The signup process consists of two tasks. The first of these is to create a unique username and password, and to submit an email address. Once that step is completed, the user is directed to a short questionnaire, wherein they are asked five questions, each of which determines something about their creature. These questions are presented in the language the user chooses to view the Webpage (currently only English and Japanese are supported).

Once the user has both the application installed and their registration process completed, they are able to log in to the system. The user brings up the application on their phone, which presents them with the login screen to enter their username and password. Once entered correctly, the user's creature is added to the world, making it visible on the display, and the user is redirected to the movement screen.

The movement screen features, predominantly, the user's creature displayed unmoving and without any equipment (Figure 4). This is the only place in the mobile device application where the creature is shown without equipment. On this screen, the user can utilize the touch screen's swipe input to direct their creature to move. The creature's movement is reflected on the public display.

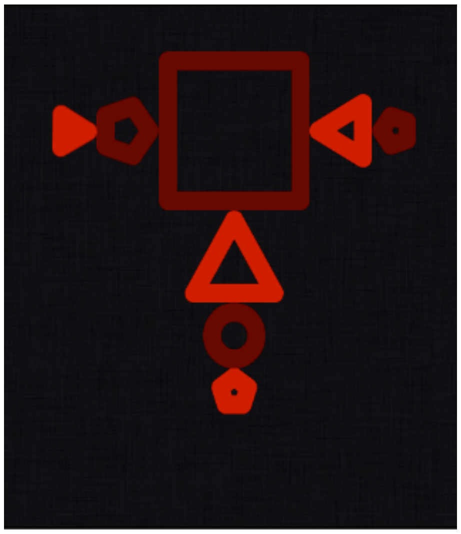


Figure 4 - A creature with no equipment

Movement serves a purpose in that the creature must be close to something to interact with it. Treasure chests are available in the environment and it is possible to interact with the other creatures if one's creature is close enough. Once the creature is within range, the user can navigate to the Interact screen through the use of the tabs at the bottom of the application. The interact screen displays creatures and treasures that the user can interact with.

If the user picks up a treasure chest, a message pops up, telling them what item they received from it. From there, they can move to the Inventory screen, where they can view all of the items they own. Here, the user can equip items on their creature avatar by selecting the item they want and then selecting the slot in which they wish to equip it. When a change is made to a creature's equipment, the results are soon visible on the public display (Figure 5).

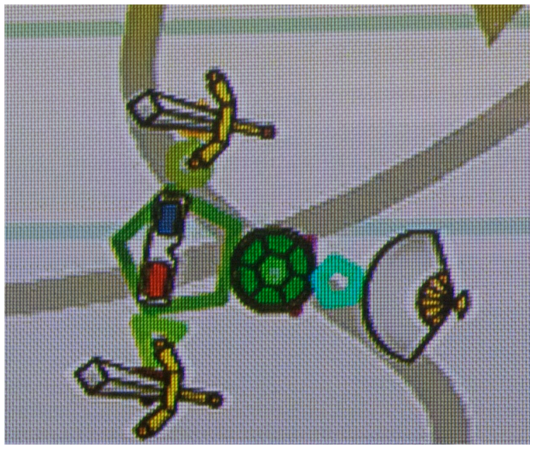


Figure 5 - A creature wearing 3D glasses and dual-wielding swords

Through the Interact screen, the user is also able to trade items with or chat with another player. When either of these interactions is initiated, the other user receives a notification and has the ability to either accept or deny. When the notification first arrives, it appears as notification typical of the mobile platform – a small red "badge" over the Notifications tab with a number inside indicating how many unseen notifications there are for the iPhone, or an entry in the pull-down Notifications list for Androids. The user can ignore them entirely or switch to the Notification screen, where the choice to accept or reject is given them. If the interaction is accepted, then both users are brought to the appropriate screen.

In a chat, users are able to select chat dialogs from a preset list, and send them to the user they are chatting with. Chat messages are automatically translated, then displayed on screen for both users to see.

In a trade, the user is shown their inventory and the items up for trade. The user can move items from their inventory into a trade slot and when they are satisfied, they confirm it. Once both sides have confirmed, the trade is completed.

There are two things that chat and trade have in common. When each is completed, any equipment that one creature is wearing and the other has not seen before becomes seen; it is added to their catalog. In addition, both creatures will change appearance slightly from the interaction.

Any time the user wants to stop using the system, they use the application's logout button located in the upper right-hand corner of the mobile device's screen.

1.2.1 Sample user experience

A typical user experience might look something like this.

1. A user approaches the public display for the first time. A handful of other users are already interacting using their mobile devices. After watching for a few minutes, the user sees the information bar at the bottom of the display. They take out their mobile device and navigate to the provided URL.
2. The website has a link to download the app, which the user selects and begins the download. They then return to the site and sign up for an account. They then take the survey to generate a creature. Then tells them they can begin playing by logging in with the app.
3. After waiting for the app to finish installing, the user opens the app and is shown a login screen. The user enters their credentials and is shown their creature for the first time. At the same time, the creature appears on the public display.
4. After the login screen, the user is shown the movement window. The user swipes over the image of their creature, and the creature moves on the public display. The screen zooms to keep the creature in view.
5. The creature swims near a treasure chest. The user notices that the chest has appeared on their interaction menu, and selects it. A window pops up informing them that they found a sword in the chest.
6. The user navigates to the inventory screen to see the sword. The user taps the sword in their inventory, and taps on a slot on their creature. The sword becomes equipped on the creature. The creature on the public display is also changed to equip the sword. The user equips and unequips the sword a few times.
7. The user then notices the catalog tab and navigates to it. A menu of "?" items appears, but the sword is shown, with a description.
8. A notification alert appears on screen. Another user has seen the current user, and has requested a chat. The user accepts and is brought to the chat screen. The user sees the message categories and navigates to the greetings section to say "Hello." They receive

"Hello" in response. The user then selects "Where are you?" and receives "Japan" in response. The users chat for a while longer, then close the chat.

9. Another notification appears. This time it is the same user requesting to be friends. The user selects yes, and is informed that the two are now friends.
10. The user examines the Interact tab again and selects their new friend on the list. A menu appears to chat, friend, and trade with the user. The user selects trade. A few moments later, they are brought to a trade screen. The user taps his sword and adds it to the trade. The friend adds a paper fan to the trade. The user confirms the trade. A few moments later, the friend also confirms and the items are swapped.
11. The user then selects the logout button and exits the application automatically. The creature is removed from the environment on the public display.

2 Related Work

Early in the design phase of the project, the team decided to make use of the public displays located at WPI and OU. Connecting the displays in both countries to encourage communication between users at both locations was an early design goal. It was then necessary to provide users a way to interact with the public displays. Smartphones were chosen for this purpose due to their prevalence in both countries and their ability to connect via the Web to a server. Huang, Koster, and Borchers (2008) also had insight into the best ways to encourage users to notice the content on a public display.

Once the team had decided upon international communication as a goal and networked public displays and smart phones as technology, the content to be shown on the display needed to be designed. Research into similar projects was conducted in order to generate ideas and to see what past projects had and had not been able to accomplish.

2.1 Public Displays

While researching public displays, the team found three helpful areas of research. Papers described the challenges involved in posting content in a public space, encouraging and gauging user interest in public content, and systems where public displays were used to encourage user interactions.

Brignull and Rogers (2003) describe how users respond to public displays, as does McCarthy (2010). In general, users are reluctant to participate in public social activities, due to embarrassment caused by participating in public. Additionally, because the content is not fully under a user's control, they feel less compelled to participate. The challenge of getting users to realize the public display both exists and can be interacted with are also discussed. One of the primary goals of the project is to foster communication between users via shared content on public displays. However, if users do not notice the content on the displays, or are unwilling to participate in the system, then the method of encouraging communication becomes ineffective. The papers discuss the most common reasons why people are unwilling to participate, and ideas for making public content more noticeable and engaging. These ideas were then incorporated into the application design in an effort to encourage user participation and interest.

Finke, Tang, Leung, and Blackstock (2008) present an interesting post-mortem about the game *Polar Defence*, which was designed for public displays. They talk about methods used to encourage user interactions, and how prevalent the interaction was while the game was active. The authors also give details of the system design, along with a retrospective analysis and an examination of user interactions once a user began using the system. They also describe the interactions between users and how best to encourage wanted interaction. Once users begin to use the genometry application, ideally the team would like the users to return and continue to interact with other users. The paper describes ways for keeping users engaged within an application while also encouraging user interactions.

Storz, Friday, Davies, Finney, Sas, and Shridan (2006) wrote another post-mortem about deploying three public displays to be used for education. The paper discusses system architecture and deployment difficulties faced during the project, which was helpful during the design phase as it described the development and deployment phases used during the described project. It also details problems encountered during deployment, both for the application and the architecture. The project uses a similar distributed architecture, where servers send data to external displays via the Internet to be shown.

Rogers and Brignull (2002) discuss user reactions to the developed system Opinionizer. The paper discusses the reaction when a large number of users began interacting with the system in a public space. The paper was of interest to the group as being one of the few projects that was used by a large group of people at the same time. A maximum of 50 users at a time is allowed in the genometry system, based on a rough estimate of how many people could both fit in front of the displays and be handled reasonably by the server. The authors look at how users interact when there are a large number of users engaged in the same public display content. They also describes methods for further encouraging social interaction when users are at the same display. However, they does not go into detail on how to encourage interaction when users are at different displays, which was a significant consideration in the design of the genometry system.

2.2 Mobile Devices as Controllers

While researching methods for allowing users to enter commands through a mobile device, the team focused on sensor input types, mobile device interfaces, and methods for connecting mobile devices to public displays.

Vajk, Coulton, Bamford, and Edwards (2008) describe methods of input and data transfer for mobile devices. The paper specifically mentions Bluetooth as a method for mobile devices to transfer data. Accelerometers are used as an input device to allow users to steer and direct a vehicle in a racing game. The architecture of the designed system is discussed, as well as the Wii-mote input device methods. Emphasis is placed on control schemes for mobile devices that are familiar and intuitive to the user. For the described project, tilt sensors were used to steer a car because of the mapping between the tilt of the phone and the steering of the car. This influenced the mobile device swipe input for creature movement. A relationship was established between the direction of the swipe and the direction of creature movement, which the team hoped would be intuitive for users. The genometry system uses web connection as the method for transferring data, but the description of how Bluetooth communication was utilized was beneficial to examine.

Maunder, Marsden, and Harper (2007) discuss a design to allow mobile devices to communicate with public displays through Bluetooth. The discussed design does not require special software or hardware on the mobile device side, making it easy for users to begin using, which gives concrete evidence that a system could be designed for multiple types of devices, without relying heavily on the different devices having specific software or architecture beforehand. The decision to support multiple devices for the genometry system was influenced by the findings of the paper.

Tuulos, Scheible, and Nyholm (2007) discuss a game called *Manhattan Story Mashup*, which was shown on public displays. Players could use mobile devices to tell stories or illustrate stories told by others. The paper details the deployment process and an analysis of how creative and engaged the users became even without language-based communication. Since langauge would be a barrier in the system, using pictures and symbols as substitutes was looked into for a large part of the design phase. This became an influence in the equipment system, as the items were designed to be recognizable by both audiences.

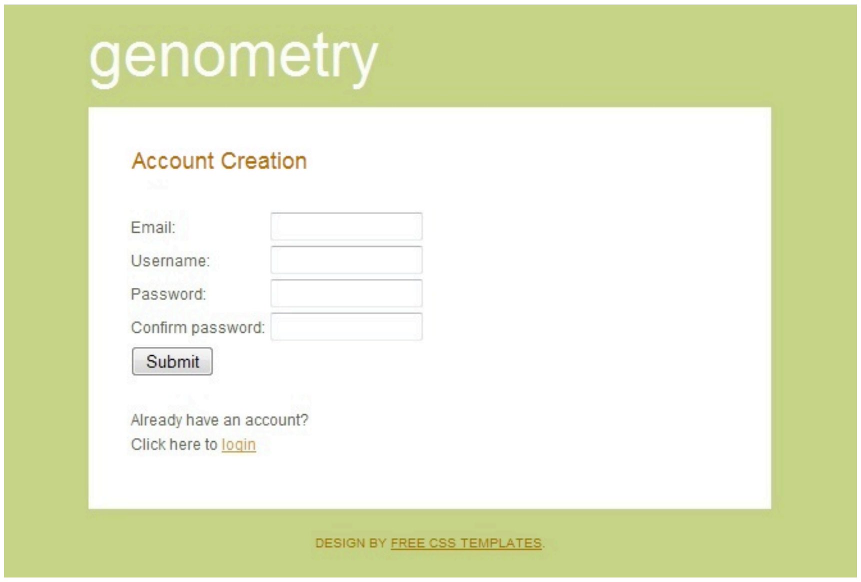
3 Application Features

The application revolves around creatures, a cross between user avatar and virtual pet, which a user controls through their personal mobile device. The user can move their creature around the environment, shown on the public display, and, through their creature, interact with the other creatures in the world. Users can individualize their creatures using wearable items to change how their creatures appear on screen.

3.1 Account setup

The first thing a user needs do to begin participating in the virtual environment is to create an account and a creature. This is accomplished through the use of an online questionnaire which can be accessed with both Web enabled mobile devices and regular personal computers. The user could be directed to the Website either by word of mouth or by visiting a public display, the bottom of which has a 2D barcode and a URL, both of which direct the user to the account-creation Website.

When the user visits the site, they are instructed to first create an account (Figure 6). The Website detects the language of the browser being used to access the site. If the detected language is Japanese, the Website will be displayed in Japanese. If the detected language is not Japanese, the Website will default to English. A user needs to first enter an email address, a unique username, a password for the account, and the password again for verification. If the email address already has an account associated with it, the username has already been taken, or the two password fields do not match, the user is prompted to re-enter data into the appropriate fields. Once all four fields have been filled in correctly, the Website prompts the user to confirm the input (Figure 7). When the user selects “Confirm,” the Website connects to the database and creates a user account with those credentials. At that point, the user may choose to leave the site without making a creature, and may log in at any time with their username to do so, though they cannot enter the virtual environment without a creature.



genometry

Account Creation

Email:

Username:

Password:

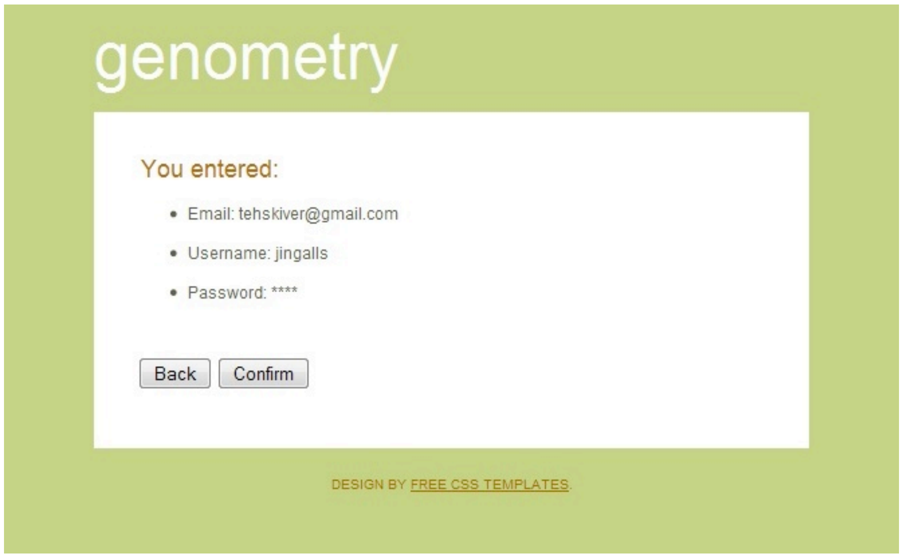
Confirm password:

Submit

Already have an account?
Click here to login

DESIGN BY FREE CSS TEMPLATES.

Figure 6 - The genometry account creation page



genometry

You entered:

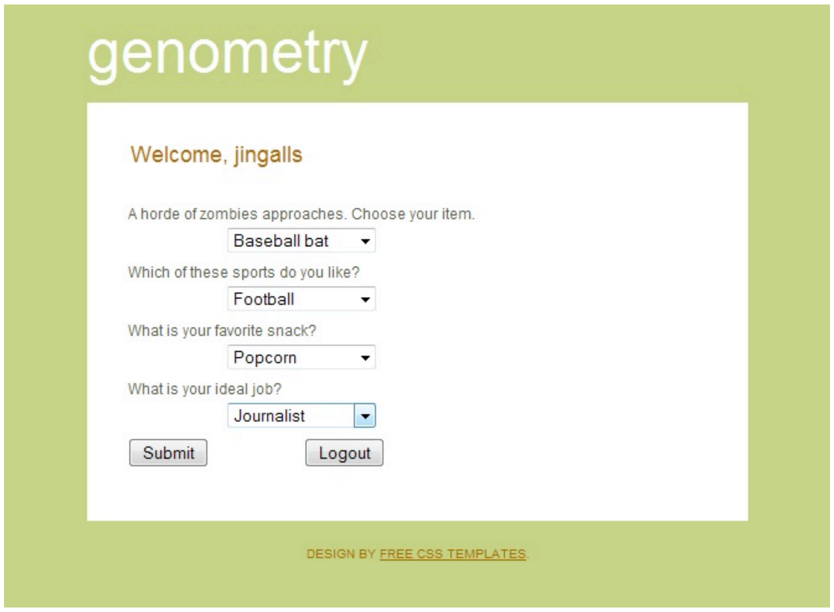
- Email: tehskiver@gmail.com
- Username: jingalls
- Password: ****

Back Confirm

DESIGN BY FREE CSS TEMPLATES.

Figure 7 - A successful account creation

The account creation page links to the creature creation questionnaire automatically. It can also be reached at any time by logging in with a valid user account which does not yet have a creature associated with it. The user is prompted to answer four short, multiple-choice questions (Figure 8). A table of questions in both languages can be seen in Appendix G. The answers to each question are used to generate a unique appearance for their creature; each of the multiple-choice answers is keyed to a value, and the final values are used to generate a starting creature's genome. Each question can be answered by picking a selection in the corresponding dropdown menu. When the user has answered all four questions, clicking “Submit” causes the Website to connect to the database and assign the new creature to the account. The user may log out at any time prior to clicking “Submit.” Once the creature has been created, the user can begin playing. The credentials created using the Website can be entered into the application on a mobile device to login.



Welcome, jingalls

A horde of zombies approaches. Choose your item. Baseball bat

Which of these sports do you like? Football

What is your favorite snack? Popcorn

What is your ideal job? Journalist

Figure 8 - The creature creation questionnaire

3.2 Movement

Within the environment (what is seen on the public display), creatures can move. A user tells the creature to move by swiping on their mobile device. Once they do so, the creature can be seen moving on the screen.

The purpose of movement is to get closer to things in the world. This could be a treasure chest to pick up, another creature to interact with, or a toy, such as a ball, which the creature can play with.

3.3 Genomes

Every creature is comprised of eight shapes of varying colors. A shape can be a circle, triangle, square, pentagon, or hexagon. Each shape can be any color represented by a six-digit RGB hexadecimal string. Together, these eight shapes and eight colors make up a creature’s primary appearance.

This configuration is represented internally with a 63-character code called a genome. Each shape has seven characters associated with it. The first character is a letter representing the shape: “c” for circle, “t” for triangle, “s” for square, “p” for pentagon, and “h” for hexagon. The remaining six characters are the six-digit RGB hexadecimal code where characters 1 and 2 are the red value, 3 and 4 are the green value, and 5 and 6 are the blue value. Each seven-character segment is separated with a hyphen to make parsing easier.

The order of segments is as follows: left hand, left arm, head, right arm, right hand, body, tail segment closest to body, tail segment furthest from body.

An example genome is:

t6A287E-cA74AC7-p6A287E-cA74AC7-t6A287E-cA74AC7-c6A287E-pA74AC7

The result of the example genome can be seen in (Figure 9).

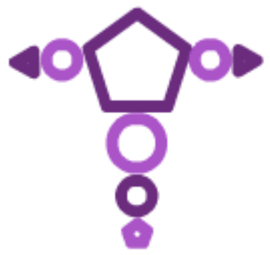


Figure 9 - An example creature for a specific genome

A creature’s genome can be changed over time as the user interacts with other creatures in the environment via either chat or trade. At the end of an interaction, a random part of each creature in the interaction is selected. The colors of the two chosen parts are compared and a middle value is computed for each. The shape will then change color to represent the new selection. There is also a random chance for the shape to change as well. In the event that a shape would have to change, the number of sides of the two shapes is compared, and a middle value is selected for both. For example, a triangle (3 sides) trading with a pentagon (5 sides) will result in a square. A circle is considered to have both 0 sides and infinite sides to allow for a continuous progression between shapes: $0/\infty \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 0/\infty$.

3.4 Items and treasures

One of the major features of the environment is the presence of 140 collectible items which can be owned by the creatures. Users can examine which items other creatures have equipped and can acquire new items via exploration and interaction with other users. The goal of adding items to the world is to encourage users who wish to collect all of the items to communicate with each other, and the item system is designed to require cooperation between users to achieve that goal.

3.4.1 Equipment

All items in the world can be used by creatures as equipment to customize their appearance. Items have four categories: head, body, arm, and tail. A creature has five equipment slots: one head slot, one body slot, two arm slots, and one tail slot. A user can equip their creature with items from its inventory. A creature can only have one item in each equipment slot at any time, for a total of five possible equipped items at once.

There are no restrictions on the kind of item a creature can equip in a particular slot. Each item was optimized for a specific equipment slot, e.g., a body item is designed for a body equipment slot. However, a user may decide to equip a body item in a non-body equipment slot, as well.

A user can select which items to equip by going to the Inventory tab within the mobile application (Figure 10). A list of all currently owned items is displayed to the user, along with a representation of the creature. The user can sort items by category (Head, Body, Arm, and Tail) and make a selection by tapping first the item to equip, and then the slot in which to equip the item. If an item was already equipped at that location, the items are swapped. The user may also remove an equipped item by tapping on an equipment slot without having first selected a replacement item. In either case, the item which was previously equipped will be placed back in the creature's inventory, and any item which replaced it will be removed from the inventory. In the event that the creature owns more than one of a certain item, only one is equipped per slot.

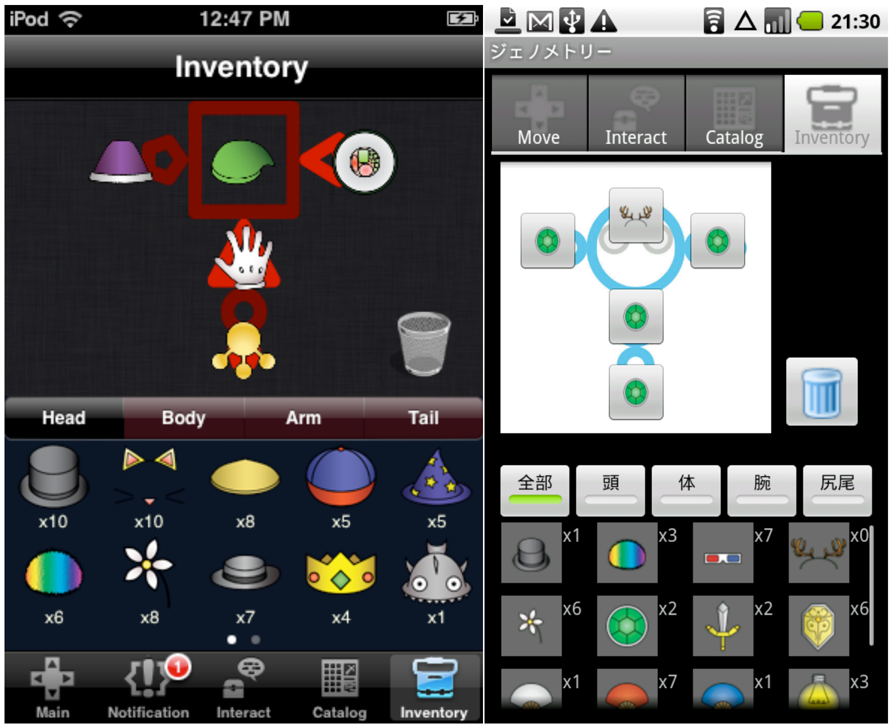


Figure 10 - The inventory screen, on iPhone and Android

Once the new item has been equipped, the public display will update to show the new item on the creature (Figure 11). The equipment change will persist until the user decides to equip a new item.

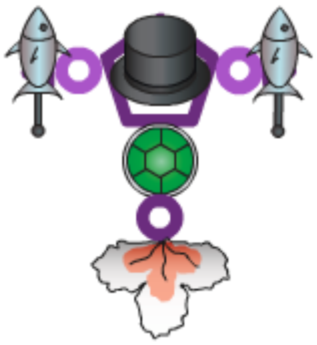
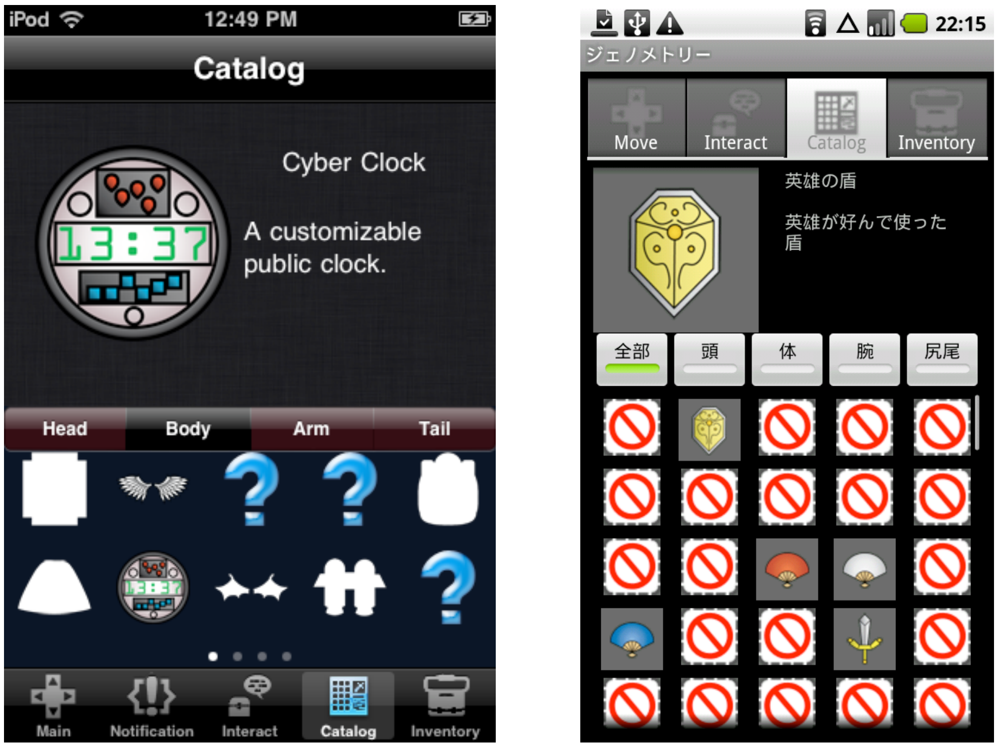


Figure 11 - An example creature with items in all five equipment slots

3.4.2 Catalog

The item catalog is a record of all the items a user has seen and collected in the virtual environment. Each user has their own catalog, which is persistent and tied to the user's account, instead of to an individual creature. This was designed so that if functionality is added to allow users to own multiple creatures, when a user creates a new creature or changes to a different active creature, their catalog remains the same. That way, users who have filled in a large portion of the catalog will not feel discouraged from creating a new creature and can obtain more items for the new creature without having to start with an empty catalog. Figure 12 shows a comparison of the iPhone and Android catalog screens, with full item descriptions in English and Japanese.


Cyber Clock

A customizable public clock.

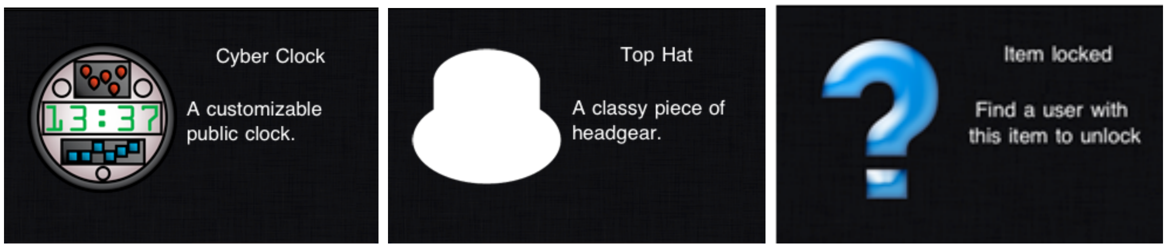
英雄の盾

英雄が好んで使った盾

Figure 12 - The catalog screen, on iPhone and Android

The catalog consists of three different classifications for items: unknown, seen, and owned. An item is considered unknown if the user has never encountered the item in the environment or seen it equipped on a creature with which they interacted. An item is considered seen if the user has interacted with a creature that had the item equipped, but has never owned that item. Finally, an item is considered owned if the user has ever obtained it, even if it was later discarded.

In the catalog view in the mobile application, different classifications of items are displayed in different ways. Unknown items are represented with a question mark icon, and the item names and descriptions are also unavailable to the user. Seen items are displayed as a silhouette icon, which can be selected to view a larger silhouette image. Finally, owned items are displayed with a full icon image of the item, as well as the full item details (Figure 13).



Cyber Clock

A customizable public clock.

Top Hat

A classy piece of headgear.

Item locked

Find a user with this item to unlock

Figure 13 - An owned, seen, and unknown item

To allow the user to view the catalog easily, the items can be filtered by category (head, arm, body, or tail). These categories are predetermined and allow users to see how many items of a certain type they are missing, have owned, or have seen. Filling in the full item details, and owning one of every item, is a major incentive for users to explore the environment and interact with other creatures.

3.4.3 Treasure

Items within the environment are referred to as "treasures," and are represented on the terminal display as treasure chests (Figure 14). A treasure chest may contain any item available in the world, and may appear at any point in the environment. Each treasure chest is assigned a random color on its creation, to help users differentiate between nearby treasure chests when selecting which one to open.

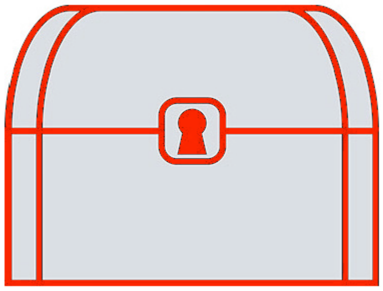


Figure 14 - A red treasure chest

When the server starts, it will generate a certain number of treasures. When the number of treasures crosses below this threshold, the server will create more treasures at random locations to make sure that there are treasures in the world at all times. Each treasure generated by the server contains a random item, which is determined by the server based on the catalog of the user who picks it up. A creature may only receive items from random treasures which have a status of seen or owned in its catalog. This requires users who wish to obtain new items to interact with other creatures in order to see their equipment and add those items to their catalogs. A creature can interact with another creature by chatting or trading with them.

A treasure is also created next to a creature when its user decides to drop an item from the creature's inventory. This type of treasure will always contain the item which was dropped, regardless of who picks it up. A user can decide to drop as many items as they would like without any limitation, although currently items may only be dropped one at a time.

A user can open any treasure which is near their creature. When the user selects the interact tab on the mobile device, a list of nearby interactable objects (creatures and treasures) is displayed. The user can identify the treasure by its color and select that item on the list to open the treasure. If the request is successful, the user will be told which item is inside that treasure and the item will be automatically added to the creature's inventory. In the case that two or more users decide to open the same treasure, the treasure will go to the user whose request is received by the server first. That user will receive the items, and the other user will receive a message saying that the treasure was already taken.

3.4.4 Trading

Apart from collecting treasures, users can also acquire more items by trading with each other. A user can choose to trade with any nearby creature, and the other creature's owner will receive a notification informing them that the first user wants to trade. If the second user accepts, both users will be taken to the trade screen. Each user's trade screen displays their creature's current inventory (not including any items the creature currently has equipped), the items they are offering to give the other user, and the items the other user is offering in return. Users may offer up to six items at a time, and each user may offer a different number of items, including zero if one user does not want anything in exchange.

When a user is satisfied with the current offers, they can confirm the trade. If both users confirm the trade, the items are swapped and the trade is complete. If anything is changed while only one user has confirmed, the confirmation is automatically cancelled to give that user a chance to view the new item offerings before reconfirming. Additionally, a user who has confirmed is free to un-confirm any time before the trade is completed. This ensures that all trades are fair, and that the items being traded have been agreed upon by both users. Once the trade is over, the catalog statuses of any items received by each creature are updated to "owned" if they were not already marked as such.

3.5 Chat

Another form of interaction users can take part in is Chat. Since international communication and cooperation is a strong motivation in the creation of the system, the ability for users to communicate with each other is a key feature. This creates a significant obstacle in that users of the program speak two languages: English and Japanese. The Chat system therefore takes into account the fact that not only can users choose one of two languages, but a user can chat in one language to someone who only speaks the other language.

The Chat interface is accessed on the mobile device. To initiate a chat, a user must first select the Creature they wish to chat with and request a Chat. A notification is sent to the other user with the request, which they can choose to accept or deny (Figure 15). If they accept the Chat, the Chat interface on both mobile devices is opened.

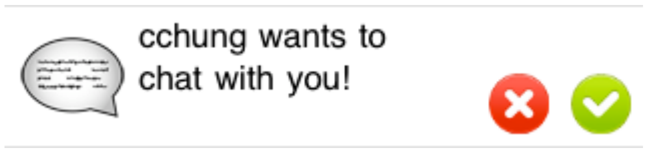


Figure 15 - A chat request from another user

To aid in communication across the language barrier, a set of pre-decided messages is used. Messages are sorted into categories to make individual messages easier to find. Examples of such categories are Greetings, Questions, Food, Places, and Animals. Once a user selects a message to send, the mobile device transmits it to the server to be forwarded to the second mobile device involved in the chat. The entire set of messages can be viewed in Appendix F.

Since all messages are pre-defined, each message has a corresponding equivalent in the second language. For instance, "Hello" in English is translated to "こんにちは" (Konnichiwa) in Japanese. When a message reaches a mobile device, it is converted into the native language of that device. In this way, users can chat with other users without first needing to learn their language.

Like an instant messaging client, a log of all the messages sent is maintained on the mobile devices until the chat is completed. Chat messages are color coded as either red or blue to indicate which user sent the message. In all cases a blue text indicates a message the current user sent, and red text indicates a message the opposite user.

3.6 Toys

"Toys" refers to simple items within the environment with which creatures can physically interact. The first toy in the environment is a ball which bounces when run into by a creature or when it encounters the edge of the environment. The simplistic nature of the interaction between users and balls means that users can make improvised games using balls and any other toys located in the environment, encouraging both verbal and non-verbal communication between any users who want to play.

Bumpers were planned as a second type of toy, which creatures and balls could bounce off of (in the manner of pinball machine bumpers), but could not be implemented in time. Because they can move creatures, bumpers would require significantly more testing than balls, as any interaction with a bumper would change a creature's trajectory and movement prediction, and could potentially interfere with users who do not want to interact with bumpers or play any games currently in progress among other users.

3.7 Internationalization

As one of the overall goals of the project is to foster communication between WPI and OU, care was taken to ensure that content was accessible and appropriate to both American and Japanese students. This included considerations such as art style and item design, as well as localization of all text within the system. The account creation Website, terminal messages, and mobile application text were all translated from their original English into Japanese with the aid of several OU students, and any part of the system can be used in either language interchangeably. In addition to the basic menu text, each of the 140 items in the world has a name and a brief, one-sentence description, and the chat interaction has over 75 message options (which are only a small subset of the desired, full list), all of which are fully localized.

4 System Architecture

The product design is separated into four main components: the server, the mobile device, the terminal, and the Website. Each component has its own set of responsibilities.

The server is the largest component in the product. It controls the flow of information between all three components. In addition, it is responsible for maintaining the world state, processing application logic, and running the game loop, which updates and maintains all data about the environment over time.

The mobile device is responsible for receiving input from the user and transmitting that input to the server for further processing. It is also responsible for displaying server responses and updates to the user by updating the mobile device's visual interface.

The terminal is responsible for converting the game state transmitted by the server into graphical output that is shown on the public display. It is also responsible for hiding the lag caused by message transfer delay from the user.

The Website is responsible for allowing users to create accounts and creatures for use in the system. It validates user information such as usernames and passwords, generates creatures for each user, and stores all information in the server database.

The server, mobile devices, and terminals communicate with each other via network messages. With the exception of the initial connection messages, network messages are represented by an XML string with a standardized set of tags enclosing data. The server, mobile devices, and terminals each have a Network Controller class which is responsible for sending and receiving the external messages. When an external message is received by the Network Controller, it is sent to a Message Handler class to be delegated (Figure 16).

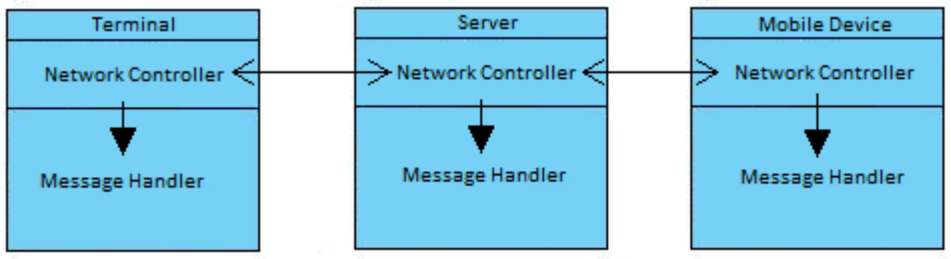


Figure 16 - Communication between the three components

The server, mobile devices, and terminals have a number of controller classes, each of which has a specific responsibility within the component. For instance, in the server there is a Chat Controller, which is responsible for handling the creation and deletion of chat sessions, sending and receiving messages between users, and keeping track of all current chats.

For network messages, Network Controller receives the message from another component, and converts it into a message type the component can understand. The message is then sent to Message Handler to be delegated. Message Handler looks at the header of the message to decide what kind of action needs to be taken. It then sends the message to the appropriate controller to be processed. If the controller determines an external message must be sent to another component, then a new message will be generated and sent to the Network Controller. The Network Controller will convert the message to one that can be sent over the network and send it to the correct component.

In the case of the server, a second, internal message is utilized to handle data synchronization issues caused by multithreading. Multithreading was implemented in the server to increase performance due to the large amount of processing required. Because the server controls data flow in the system, it is responsible for receiving and either processing or relaying each external message that gets sent. The maximum number of users that can send be logged in is 50, and there is no set upper limit for the number of connected terminals. In addition, each message needs to be processed quickly in order to reduce latency, which the user will be able to detect.

To speed up the processing time for a large number of messages, the server is multithreaded to be able to process multiple messages at once. Each controller is given a separate thread, which only processes messages relevant to its controller. For instance, only the Chat thread can access and modify Chat objects. However, it is possible for a thread processing requests in one controller to need information handled by another. In this case, a request for the data is made and sent to the controller via an internal message. When the second controller receives the internal message, it will process the request, and send the requested data back in a second internal message.

The relationship between all the components can be seen in Figure 2. Arrows indicate the flow of data between two components. Black arrows indicate the information is transmitted via

XML messages. Gray arrows indicate a direct connection, which does not require an XML message to transfer data.

4.1 Website

The Web-based portion of the project consists of a number of pages for creating an account and a creature. The first step a user must take is to create an account for our system. Next, they must answer a short questionnaire to create their first creature. These are currently the only steps necessary to begin using our system.

4.1.1 Creation of Account and Creature

First a user must select and enter a username which acts as their account name. They must then provide an e-mail address which is used to notify users of special events or other news pertaining to the system, or to recover passwords. A given e-mail address can be used to create only one account. Finally, the user must choose and confirm their password. At this time, there is no utility in the system to change a password once an account is created. Before connecting to the database, the Website checks that the password field and password confirmation field match. If the passwords do not match, the user is notified of the problem and asked to correct it. If they do, the Website uses an SHA-256 digest algorithm to create a hashed password which will be stored in the database. Upon completing these steps, the Website connects to the database and attempts to create a new entry for the user. If an account with the given username or e-mail address already exists within the system, the user is notified of the problem and asked to change their selection. Otherwise, the user is notified that their account has been successfully created.

Once a user has finished creating their account, they can choose to immediately create a creature to start playing or to wait and create a creature at a later time. If they choose to wait, they must login upon returning to the Website. To create a creature, a user must answer four short multiple-choice questions. The answers for each question are assigned different numerical values, and are used to calculate the starting shapes and colors for the creature. This way, answers directly correlate to creature characteristics, but are not obvious to the user; at the moment, the values only determine appearance, but it is intended for the answers to also correlate to creature statistics and behavioral traits. Also, the questions are trivial in nature, allowing the user to complete the process in a shorter amount of time than if they were customizing their creature.

The Web application was written in jsp for one major reason: server-side code execution. Although it is possible to log in to and use the Website from a computer, the majority of users log in with smart phones. For this reason, the team thought that minimizing the resource usage of the Website would be beneficial. The server runs Apache Tomcat 6.0, which can execute jsp pages locally. In addition to the Tomcat Server, Osaka University generously granted a global IP, allowing the Web application to be deployed to the general public.

The most important aspect of the Website from a creative design perspective is the choice of page style with css. In order to maximize development time, we chose to use a preexisting stylesheet from freecsstemplates. In accordance with the agreement of that organization, there is a link to the source at the bottom of every page of our Website.

4.2 Server

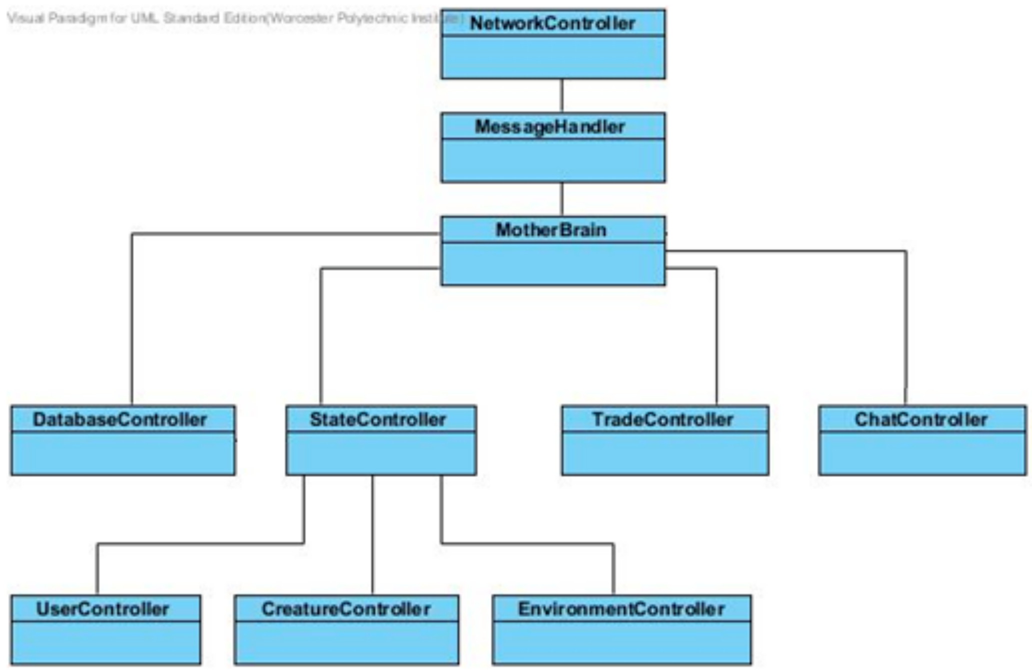


Figure 17 - Overview of the server's high-scale architecture

Components in the system communicate with the server through external XML messages. The server processes each message and updates its state accordingly. It then sends update messages to all connected devices if the change affects them. Each component that wants to communicate with the server must first send a connection message. These messages are handled by the server's Network Controller, which opens a socket connection for each new device and creates a thread which listens for incoming messages. Any subsequent messages are sent to the server's Message Handler to be delegated.

Messages are all marked with a message type, which is used to decide how to process it. The server is separated into Controllers, each of which has a specific subset of messages it is responsible for processing. Each Controller is in charge of maintaining a subset of world state, and cannot access data it is not directly responsible for. For instance, the Chat controller can access and modify Chat objects, but not User or Creature objects. If a Controller needs data handled by another controller, it must request it by sending the Controller an internal message. Messages are delegated based on the kind of object that needs to be modified once the message is processed. Network messages are therefore categorized into the following types: Database, General State, Trade, Chat, Creature, User, or Environment.

Once a message reaches a Controller, the message type is reexamined to determine the exact procedure that needs to be used to finish processing the message. Each Controller has methods and classes to handle the possible requests made. For instance, the Chat controller has methods for sending and receiving messages, and keeping track of Users communicating with each other. Figure 17 shows the hierarchy of the different controllers in the server.

4.2.1 State

The server's primary responsibility is management of system state. It is responsible for knowing who is logged in, where their creatures are in the world, what the creatures are doing, wearing, and seeing, what is in the environment, where each creature and object in the environment is, and who is close enough to interact with whom. The "game loop" (the set of methods that handles updating all of this information) which updates this long list of state variables runs every 1/30th of a second.

There are two main types of ways in which state is changed. The first is an initiated change, which is only started upon explicit user request, such as deciding to change what a creature is wearing. These types of changes are not handled by the game loop, but rather by the message handlers. The second is a reactionary update, which is started by the server itself in response to another action which took place, such as a creature continuously moving across the environment. Every "tick," the positions of creatures and environment objects like toys need to be updated accordingly. Specifically, the game loop's update methods are concerned with the latter type of update.

The StateController is the delegator when it comes to these updates. It keeps track of how many milliseconds have passed since the last update ran and then calls the update loops in CreatureController and EnvironmentController with that information.

The CreatureController's update method loops through every creature in the system and recalculates its position in the world based on acceleration and velocity. It also recalculates what treasures and creatures are visible to each creature in the world.

The EnvironmentController recalculates the position of any moving toys in the environment and deals with collisions between toys and creatures. It also sends a message to the terminal if the visibility of a treasure chest needs to be toggled because it has become or is no longer visible to creatures associated with that terminal.

The MotherBrain's main loop is what decides when one 30th of a second has passed and it is time to update state, and notifies the sub-controllers to do so. It also stops all of the message handling threads so that they will not interfere with the game loop, and resumes them when the game loop has ended.

4.2.2 Database

Within the system, the database provides data continuity, ensuring that all important data about the users and the world will be maintained even if the server is not running. The server, as the manager of system state, manages the database. Before any changes to state are made within the server, they are first made within the database, to ensure consistency across all parts of the system. Whenever the server needs state information it does not have, it pulls that information from the database.

The design of the database schema itself is based on the fundamental question: *what data needs to be stored across sessions?* Users and their passwords are the most immediate concern, as users cannot login without them. The creature assigned to each user is also necessary, since it must be displayed on screen. Additionally, all persistent data relating to creatures must be saved such as genome, stats, traits, equipment, and owner. There is also the user to creature relationship which is duplicated both ways. This type of redundancy is usually unnecessary and avoided in database design. However, it was decided that while a creature's owner is final and will never change, that the system should be extendable such that a user can own multiple creatures and switch between them at will. Users can also have a relationship to other users in terms of friendships. It was decided that both pending and confirmed friendships would be stored, resulting in a status bit.

Possibly the biggest change made to the initial schema (compare Figure 18 and Figure 19) is that the Item table was rendered obsolete by the dissemination of item information to the mobile devices. Once that was implemented, the only item information the database (and, by extension, the server) needs to keep track of is the item's ID number. These ID numbers are used primarily in two relationships. The Owns relationship describes a creature owning an item. Any item in that list is considered to be in the creature's inventory. By contrast, a user can have seen an item, which will put it in their "catalog". Aside from these two relationships supporting different features, the key difference between them is that Owns relates an item to a creature, whereas HasSeen relates an item to a user. This is a deliberate difference in the case that the system is extended to support users owning multiple creatures. The catalog will stay constant for a user, while a new creature will have an empty backpack.

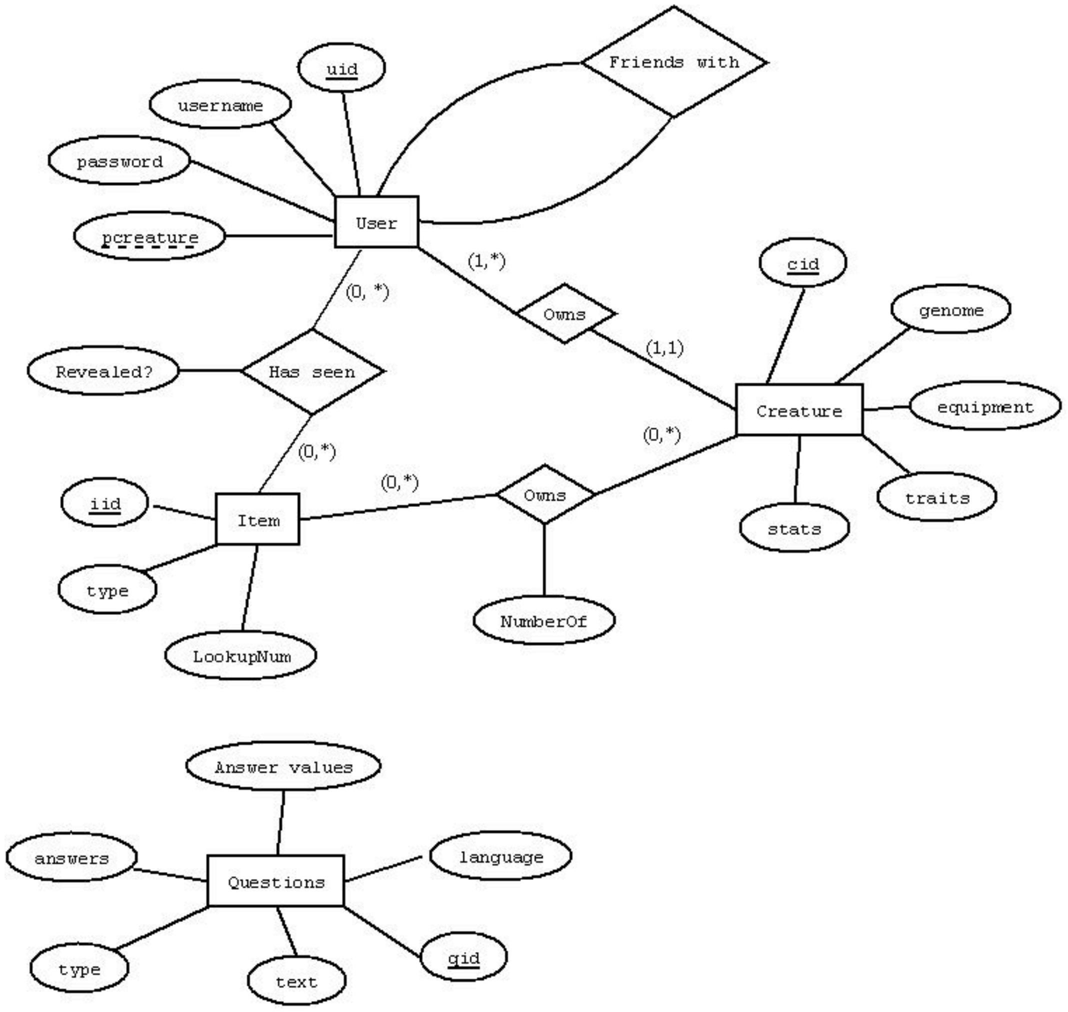


Figure 18 - Initial database schema

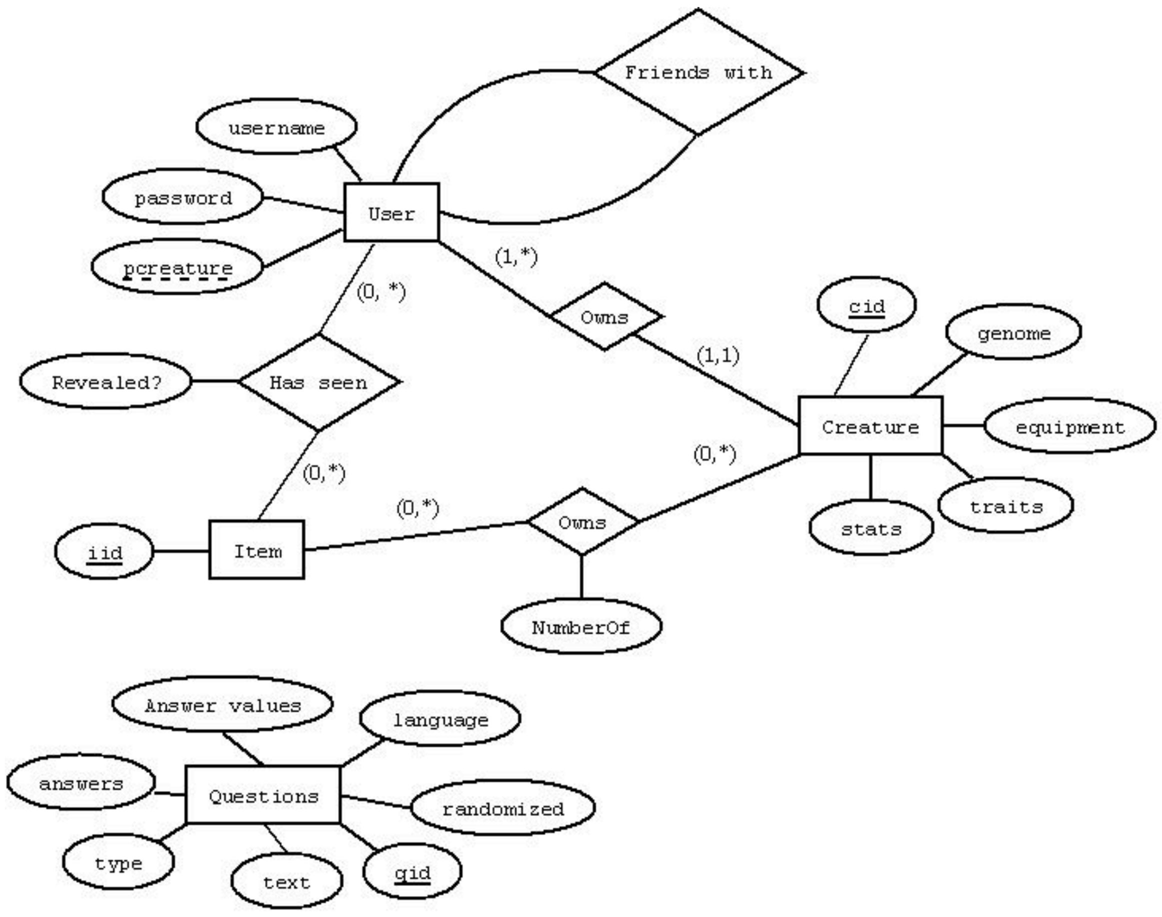


Figure 19 - Final database schema

Questions are the last entity in the database. Used not for the game server, but rather for the Web application sign up process, these questions decide what a creature will look like and what kind of stats and traits it will have. This table has nothing to do with system state. The table's purpose is to simplify random question generation in accordance to a specific language.

Transactions with the database are kept small. Poll catalog and poll inventory, the largest queries, are pared down by the removal of the Items table. Initially, the two queries had been joins between Owns and Item and HasSeen and Item, so that an item's information could be pulled at the same time as the inventory or catalog.

The only queries that are not standard queries are the random polls for pulling a random subset of questions and to pull a random item from a user's catalog. In order to support multiple languages on the system, all of the username fields, question fields, and anything else that could be multi-lingual had to be switched from the default encoding of latin8 to utf8 so that Japanese characters, for instance, could be stored there as well. This is especially important for the localized Question table, where it had to be possible to select a subset of questions in a given language.

Because of its simplicity, MySQL was selected as the platform of choice due to its ubiquity and cost-effectiveness. Initially, Oracle was considered as an alternative, but it became obvious that none of the system's requirements went beyond the capabilities of MySQL.

The database used is one hosted by WPI and, as the server is at OU, Internet lag for database transactions generated a significant amount of latency in the system. The team proposes that if the system is to be used, that a database closer to the server (perhaps, even, on the same machine) might be more efficient.

The server connects to the database through the use of JDBC (Java Database Connector), an API for connecting to SQL databases from Java (Oracle). It includes support for PreparedStatements, which are an excellent way to sanitize SQL statements and is very widely used for connecting Java to most SQL databases.

4.2.3 Networking

The server has a thread listening on a pre-established port for clients to connect to the server. The server can be reached through an IP address which never changes. When a client establishes a connection a new thread is spawned to handle that client on a separate port. The server does not initially know what type of client the client is (terminal or mobile device), so it waits for a follow up message to tell the server what type it is. Otherwise, it will timeout. Mobile devices timeout after a specified period of time, but terminals do not. Terminals do not send any ping messages to the server, since the messages were considered unnecessary.

4.2.3.1 TCP vs. UDP

During the design phase, UDP was considered as the network protocol to use for the entire system. By not requiring that every packet make it to the recipient in order (or at all), UDP would have enabled the system to hide the effects of network latency more smoothly than TCP. However, OU's O+PUS network of public displays requires the terminal applications to be developed in Flash, which is not fully compatible with UDP, so TCP was ultimately chosen as the protocol for the entire system. Since TCP is not the ideal protocol for this system, it attempts to utilize TCP's unique properties to its advantage.

One of the potentially advantageous features of UDP is that UDP packets have header information to distinguish each type of message from the next. Since TCP uses streams, the messages between components in the system add headers to their messages to delimit them within the stream. The headers consist of a byte for the message type followed by an integer for the size of the payload, which follows the header (Table 1). The message is serialized into an array of bytes and added to the output stream.

Table 1 - Description of network messages sent between components

Connect Message		
Elements	Description	Value
1 byte	Type	1
4 bytes	Size	1
1 byte	Device Type	0-Phone 1-Terminal

XML Message		
Elements	Description	Value
1 byte	Type	0
4 bytes	Size	String Size
# bytes	String Data	Byte Array of XML

4.2.3.2Flow Control

In order to reduce the network traffic within the system, the server does not send state updates that the terminal can calculate every frame on its own. This also prevents the terminal's state data from continuously being overwritten by the server's updates each tick, since using TCP guarantees delivery of the data. Instead, state updates are sent by the server at set intervals, which can be adjusted to determine the optimal balance of delay and accuracy for the system.

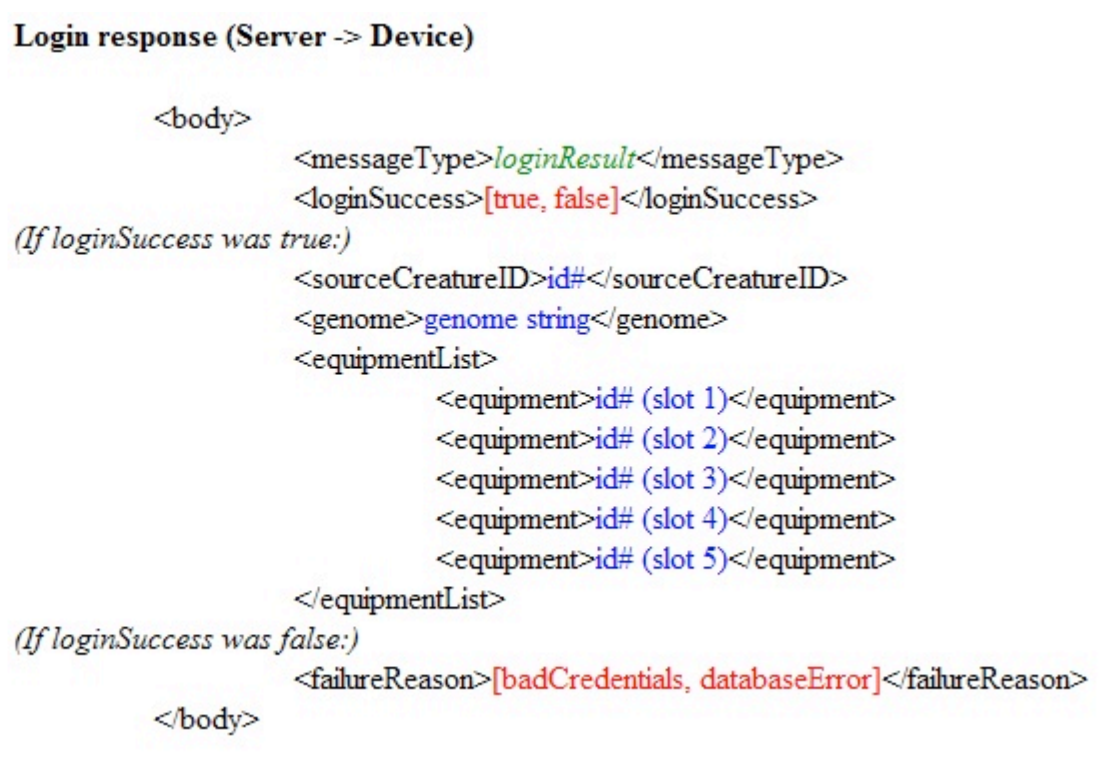
4.2.4 XML messages

The three components of the system communicate primarily via XML messages. These messages use XML tags to wrap the data being sent in order to clarify which data the sender needs to include and the recipient can expect. Although the tags add a layer of overhead which could be avoided simply by sending the data as a single stream of bytes, the extra size of the characters for the XML tags was judged to be an acceptable tradeoff compared to the enhanced ease of development across the multiple platforms within the system. In addition, messages to the terminals are compressed to save space (compression of messages to mobile devices is planned but not yet implemented).

4.2.4.1Message structure

Each XML message has the same header, which includes the sender and recipient device IDs (the server is always 0, and new IDs are assigned to devices as they connect), whether the recipient should confirm that it has received the message, and whether the sender should await a reply. Because the system ended up using TCP rather than UDP for communication, the last two items in the header were not utilized.

After the header, each message contains the message body, which always begins with the message type. After the message type, the body differs based on the required contents of the message. Because the number of messages grew very rapidly, it was decided to document each message structure in HTML on a shared Web space using the file sharing program Dropbox, so that each team member could see the contents of any specific message.   There are a total of 60 messages specified for the entire system, and they were sorted on the documentation page by sender: server, terminal, or mobile device. Additionally, each message specified its possible recipients, since many of the messages have similar types, but require different data. The documentation specified tag names as well as what kind of data each tag pair should contain. Figure 20 is an example of the XML message documentation; full documentation for all of the messages in the system is in Appendix A.



Login response (Server -> Device)

```
        <body>
                <messageType>loginResult</messageType>
                <loginSuccess>[true, false]</loginSuccess>
(If loginSuccess was true:)
                <sourceCreatureID>id#</sourceCreatureID>
                <genome>genome string</genome>
                <equipmentList>
                        <equipment>id# (slot 1)</equipment>
                        <equipment>id# (slot 2)</equipment>
                        <equipment>id# (slot 3)</equipment>
                        <equipment>id# (slot 4)</equipment>
                        <equipment>id# (slot 5)</equipment>
                </equipmentList>
(If loginSuccess was false:)
                <failureReason>[badCredentials, databaseError]</failureReason>
        </body>
```

Figure 20 - The response to a mobile device's login attempt

4.2.4.2Server XML package

As the server handles nearly all of the XML message traffic within the system, its XML package was designed to be as robust and simple to use as possible. The Java XML parsing classes were not appropriate for the parsing the server required, since they were designed to gather groups of data from tag pairs repeated throughout a large document, instead of single, short strings in small messages. Therefore, an XML parser was built from scratch specifically for the server, and provides just the simple string parsing functionality required. It was designed to catch errors in any messages, such as missing tags, so that the code on the client sending the message could be updated with the corrected XML structure. Additionally, in order to make the message construction as lenient as possible, it was designed to be case-insensitive and does not require the individual tag pairs for a message to be in any specific order.

The other significant component of the server's XML package is the string constructor, which is a single class containing static methods to build each type of message body the server needs to send. Each method requires only the necessary information for that message, and automatically wraps the data in the correct tags and returns the message string. XML string construction methods were written to not only take single values, but also to wrap data lists in repeated tag pairs. The separate string constructor class means that any new messages or changes to existing message structures only need to be added in one place, and that even someone who is unfamiliar with the message construction can generate the correct message string.

4.2.4.3Message compression

Using XML Strings to have the server communicate with the mobile devices and terminals drastically increases the size of network messages. Since Unicode strings have a large amount of redundant data (made even more redundant by using XML tags), compressing strings before sending them reduces the amount of data being sent through the network. For small strings, compression will actually increase the size due to overhead; however our XML strings are long enough to always benefit from compression. An initial edge case (1 terminal, 1 creature) test of compressing messages sent to the terminal showed that the average reduction of size for 1261 XML messages was 168 bytes. This resulted in a total reduction of about 207.5kB of XML message data during a period of less than 1 minute.

The maximum of 50 creatures connected to the server at one time results in messages about 14kB in size, which negatively impacts performance if they are sent to the terminal each frame or even every other frame (30 frames per second). In an isolated test, this 14kB of string data was compressed to about 400 bytes, a size reduction of more than 95%.

Using compression, the size of all messages was consistently reduced to less than 500 bytes. While the average size of XML messages still surpasses raw data serialization, it has allowed us to add and manage new message types faster than the latter.

Zlib was chosen as the compression library for XML message strings, as it is the best-supported compression library that is common to the three languages used by the system: ActionScript 3 for the terminal, Java for the server and Android mobile device, and Objective C for the iPhone mobile device. Message compression is currently only implemented for terminals, as they receive the largest amount of data.

4.2.5 Multithreading

The server was made multi-threaded due to the concurrent nature of the whole system. Every external message that gets sent in the system must eventually reach the server and be either processed or passed along to a separate component. This makes the server a bottleneck in the system. A delay in message processing can be noticed by the user, which is undesirable. To reduce the delay, multi-threading was implemented to allow multiple messages to be processed at once. Additionally, the networking in the server is multi-threaded. The server listens for connections in one thread and handles individual client connections in separate threads.

A large amount of data is shared and modified by the different controllers, so it is important to prevent controllers from modifying the state of the server while the game loop state gets updated. This is achieved by dividing up a "tick" (one 30th of a second) into two time slices (Figure 21). During the first time slice, the state is updated, and during the second time slice messages are handled by the controllers. During the first slice, the message handling threads wait until they are given permission to continue in the second slice. At the end of the second slice the message handling threads are told to stop and wait, and the game loop waits for all of the threads to finish before continuing. Messages cannot be processed while the game loop is running because a message may involve changing game state, which is accessed and modified during the game loop.

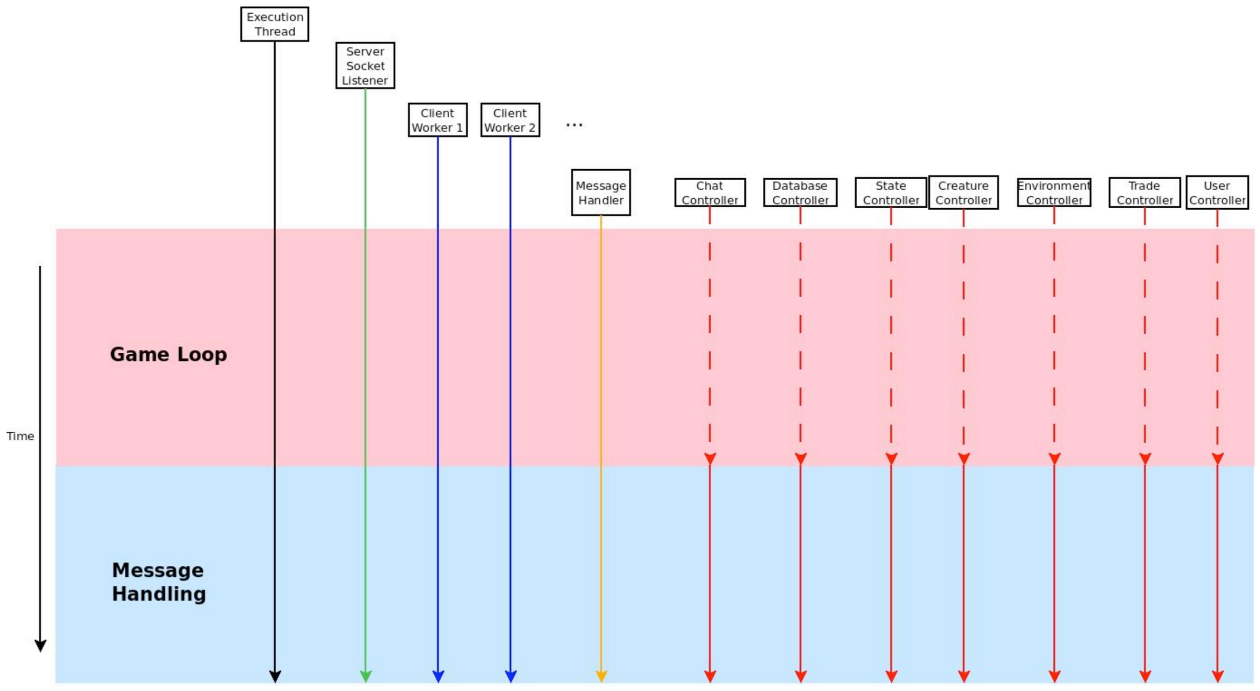


Figure 21 - The two halves of the server's update cycle

The first time slice, the game loop, is single threaded. In the initial design, the game loop updates for each type of object would all have their own thread. However, there were not enough tasks that needed to be performed each game loop to justify the added overhead of threading.

Currently, only creatures and balls need to be updated each tick. There is an upper limit on both, so at the maximum capacity, only 60 relatively simple updates would need to be made.

The second time slice, message processing, is multithreaded. Each message is sent to a controller to be processed. Each controller has a subset of world state that only it can access and modify. If a controller needs to access data in another controller, it must make a request to the controller through an internal message. By doing so, the server prevents issues like deadlock and starvation. Since the data can only be accessed by a specific controller, it was deemed safe to give each controller a thread to process its own specific messages. A controller will still only be able to modify its own data, but now multiple messages can be processed at once. If a controller needs data from another controller, an internal message is created and enqueued in the controller's message queue. Instead of waiting for a response message, the controller will proceed to the next message. Eventually, the controller will receive the response, and can continue processing the previous message. Because these internal messages are needed to finish processing a network message, a separate queue exists in each controller for internal messages. Each controller gives messages in the internal queue priority over the external queue: only when the internal queue is empty will a controller process messages in the external queue.

The process for adding new threaded controllers to the system is streamlined. An abstract RunnableController class can be extended in order to add a new controller that will be handling XML messages. The controller can then be added to the server's initialization code, and then it can begin handling messages.

4.2.6 Internal messages

Internal Messages are used as a means of passing messages between controllers so that one controller never has access to another controller's data. This is most important in the State subcontrollers, where every game tick, an update look would be running. Each controller that deals with these data receives its own queue of internal messages, where any other controller in the system can enqueue a message for that controller to handle.

Each internal message contains a type. Similar to the XML message type, this type decides how the message will be dealt with once it gets to its destination. Messages coming from the DatabaseController also have a subtype indicating if an error occurred or if the results of the database transactions were anything other than what was expected. Lastly, each message has its own array of objects that have been cast to the generic Object type. This is so that any data that needs to be put in the message can be stored there.

Out of necessity, this is a highly-documented component of the system. If the sender and receiver do not agree on what information is being sent for a specific message type, the system will throw errors. This makes adding small changes to the message passing more difficult than it might have otherwise been. If the message structure ever changes, it is likely that more than one system needs to be modified to account for the change. These are not errors that would break on compile time, so even the smallest changes require testing to ensure that they have been implemented correctly. Creating the documentation for each message type made it easy to synchronize messages across the different components.

An example of the system in action is when an update to a user's catalog is needed. A component that is updating the catalog sends an internal message to the database with the creature's ID and the ID of the item to be added. The database makes the changes first, and then sends a message on to the CreatureController to make the same changes to the creature. This means that the database is never trying to access a creature object in the state and the CreatureController is never trying to execute database functions, preventing conflicting sets of data from being created.

4.3 Terminal

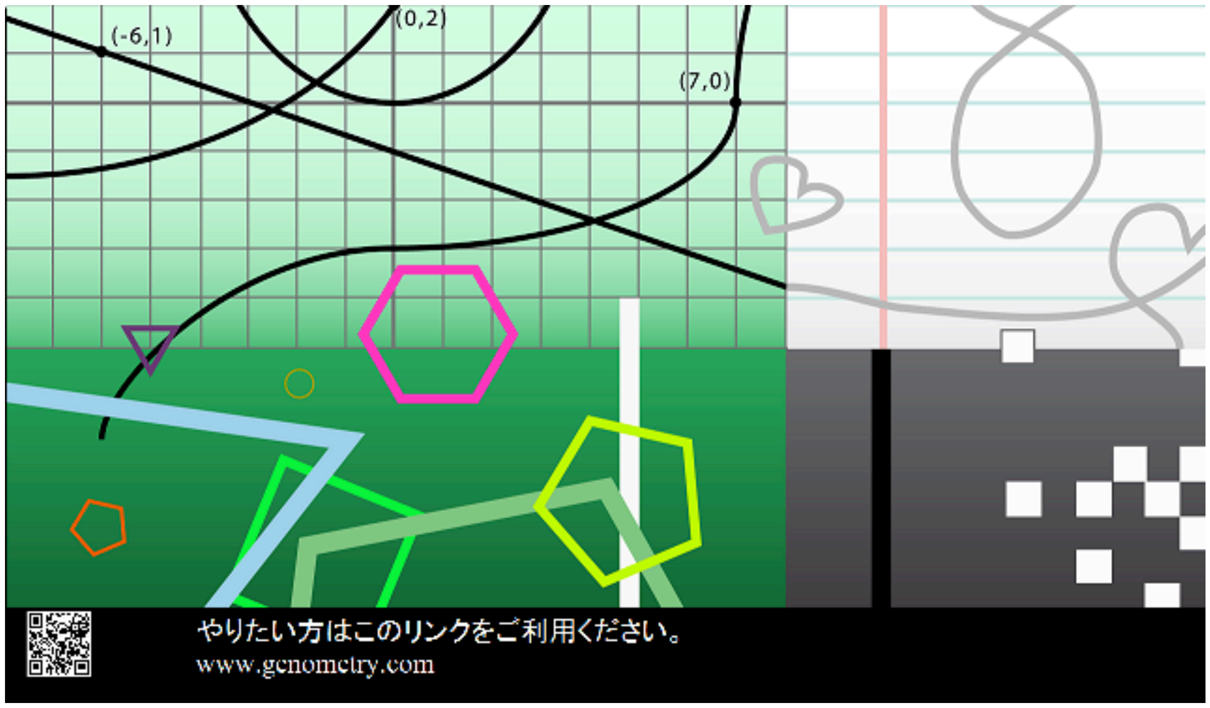


Figure 22 - The terminal is split into the environment (above) and the information bar (below).

Terminals are public displays located at WPI or OU, where a user can log in and use the system via a mobile device. When the system is running, each terminal is responsible for displaying two things: the game environment and instructions for new users to begin playing. Each terminal screen is divided into two sections (Figure 22), each of which is dedicated to one of the above parts. The terminal is designed to be a Flash application, written in ActionScript 3, due to technical specifications set by OU. The public displays at OU can only run Flash applications, so the decision was made to develop in Flash.

Terminals maintain their own game state, which includes all the necessary information to display the part of the game world pertaining to that terminal. Because the game world is too large to be completely shown on any one terminal, each terminal only displays what can currently fit. The terminals receive updates from the server in order to update their own display, but they do not send any information back to the server after an initial connection confirmation.

Creatures that are logged in to a given terminal continue moving between updates from the server. In this way, terminal displays help to mask latency with smooth moving animations.

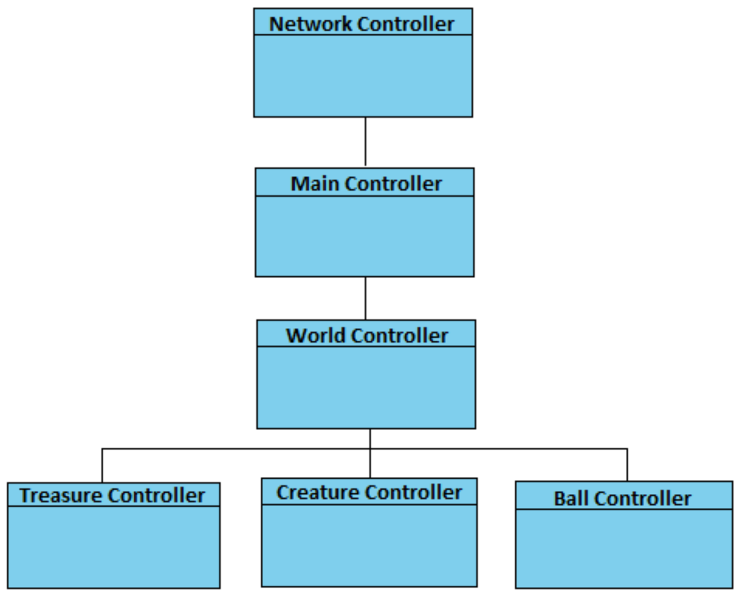


Figure 23 - Overview of terminal high state architecture

4.3.1 Networking

The terminal connects to the server via a standard socket connection. Because Flash has the limitation of being single-threaded, only one socket is open between a terminal and the server at any given time. Furthermore, additional optimization of the terminal code is unlikely due to this constraint. Data transfer between the server and the terminal only occurs in one direction: from the server to the terminal. The one exception to this is the initial connection message a terminal sends in order to register itself to receive updates. The terminal has no information that the server does not already possess, and any errors that occur on the terminal's side can be handled by the terminal without needing to update the server.

To establish a connection, a connection message is sent from the terminal to the server. Connection messages consist of only six bytes and are used to confirm connectivity and inform the server of the device type (i.e., that it is a terminal connecting). Afterwards, updates from the server are sent as XML messages. XML messages are sent between the terminal and the server with a short, non-XML, header describing the message type and message size, followed by an XML string which is parsed by the receiver. Using a TCP connection ensures that all data transmitted between the server and the client will be accounted for, if not properly received and handled.

Once an XML message is received by the Network Controller (Figure 23), it is enqueued in the World Controller's message queue to be handled. The Network Controller then goes back to listening for new messages.

In the event of a socket disconnection, the terminal will no longer receive updates from the server. A socket disconnect can be caused by a loss of Internet or the server shutting down (which itself can be caused by someone turning off the server, the server losing power, or the server encountering an error and crashing).

An Event Listener exists which listens for problems with the socket connection. If the listener detects a problem with the socket connection, it will first have the World Controller wipe all world state, both on the public display and in memory. This is to let the users know visually that the terminal is no longer connected to the server and to make synchronizing state information easier when a new connection is established. Next, the terminal will wait five seconds and attempt to establish a new connection. If that attempt is unsuccessful, the terminal will wait another five seconds and try again. This process will continue until either the terminal is turned off, or it establishes a new connection. The server will resend all state information when it detects that a new terminal connection is actually a reconnection.

4.3.2 World Controller

Like the server's State Controller, the terminal World Controller is responsible for maintaining game state. Unlike the server though, the terminal is only interested in game state which can be displayed graphically on a public display. This means that the server only needs to send a subset of game state data to the terminal, reducing message passing between the two components.

The World Controller contains lists to keep track of creatures, interactable toys like balls, treasures in the world, and the equipment each creature is currently wearing. It also keeps track of what language the public display should be using, so the user instructions at the bottom of the screen can be displayed in the correct language.

The World Controller has a message queue where all incoming messages are enqueued for processing. Since Flash is restricted to only a single thread, all messages are processed one at a time as they arrive. Each list has a separate class with update functions for each type of object.

The World Controller also runs the game loop, which executes every one 30th of a second. Since Flash is event-driven, a timer counts down from one 30th of a second to 0. When the timer reaches 0, the game loop function executes. At the end of the game loop, the timer is reinitialized to be 1/30 of a second minus the amount of time the game loop took to execute. If the game loop takes longer than 1/30 of a second to execute, the next game loop is automatically called and a warning is printed for the developers to see.

4.3.3 Information bar

At the bottom of the terminal display is an Information Bar, meant to instruct users on how to start playing. The Information Bar is a black bar with a short line of instructions, a URL, and a 2D barcode representation of the URL.

The instructions inform users that they can create an account to play with by visiting the URL. Each terminal keeps track of the local language, so the instructions can appear in either English or Japanese. The URL can be navigated to using either a mobile device or a computer and leads to the Webpage for account creation. The 2D barcode can be scanned by a mobile device with a barcode scanner and will similarly link to the Webpage for account creation. Once a user has an account and a creature, they can login and begin playing on their mobile device.

4.3.4 Movement projection

The server is the part of the system which has the most up to date information regarding the position and appearance of everything in the environment. It sends periodic updates to the terminal informing it of changes.

In the world, creatures move based on swipe gestures sent from a mobile device. The swipe information is sent to the server, which converts it into a movement vector. The vector is then applied to the creature's current movement information, and the result is stored in server state and sent to the terminal. Creatures can appear to speed up and slow down over time. A swipe from a mobile device causes a creature to slowly speed up over time until it reaches a maximum speed, then slows down due to drag in the environment. To account for all of this, starting position, velocity, acceleration, and time are stored for each creature.

While the terminal can quickly change its own state to mimic the information sent by the server, there is a problem in that the server cannot send updates quickly enough. Moreover, any update sent by the server is outdated as soon as it arrives because of the time it takes an update to travel over the network.

To deal with both problems, the terminal uses a form of movement projection called "dead reckoning" to move objects in the world. The terminal receives updates from the server, and "guesses" their next movements based on the information it received. If the guess is correct, then it can safely ignore the next update. If it is incorrect, then it can revert to the server information instead. Doing so allows the terminal to move game objects around in between updates, turning jerky, sudden updates into smooth transitions between points.

To handle problems caused by the delay in updates from the server to the terminal, the server must also send a target position to the terminal with every movement update. Every time a movement update is sent to the terminal, the target position is compared with the creature's current target position. If the two match, then it is assumed that the terminal has been predicting correctly, and the position update is ignored. If the target positions differ, then the terminal has predicted incorrectly, and the creature is moved to the new location. The new velocity, acceleration, time, and target positions are also saved.

4.3.5 Pan and zoom

The world environment is larger than the public display used to show it. This is done to encourage users to explore their environment. Any creature currently logged in should be displayed on screen at all times. As a user moves around the world, the camera should move to follow their creature.

However, when two creatures move far away enough from each other, the screen cannot move to display both of them at the same time, as the screen has a fixed width and height. Instead, the camera will zoom out to continue to show both creatures. If the creatures move close to each other again, the camera will zoom back in.

A minimum and maximum zoom were determined to keep the creatures from becoming too large or too small based on how zoomed out the camera is. The minimum zoom is the width of the public display. The maximum width is currently the size of the world, which is two terminal screens across and two terminal screens down.

The size and location of the information bar are not affected by the terminal pan and zoom. All other objects on screen, such as creatures, background, environment objects like toys and treasures, will all scale and move with the environment.

4.4 Mobile devices

Mobile devices are an essential part of the system. These user-owned personal devices can connect to the system to be used as the controllers for the users' creatures within the virtual world. Moreover, all user-specific information such as inventory and chat history is displayed on the mobile devices, keeping the public display as general-use as possible. Processing of much of the user-specific information also takes place on the mobile devices, reducing the demands on the server.

Two mobile device platforms are currently supported by the system: iPhone and Android. These devices were chosen for development because both devices are used by WPI and OU students.

4.4.1 iPhone

The iPhone system uses the Cocoa Touch API for the user interface. This API allows the full use of the iPhone touch screen and many other iPhone user interface elements such as Tab View, Navigation View, and Table View. The new iOS 4.0 SDK also allows access to several new public APIs that enable location tracking and gesture recognition. Therefore iOS 4.0 was chosen to be the base SDK version for the project.

4.4.1.1System Design

The system is divided into 10 controllers centered on a controller called Main View. The Main View controller has three main jobs. The first job is to act as a message delivery system for both internal and network messages. The second job is to maintain user session information and generate message headers for all outgoing messages. The third job is to control the Tab View, the main GUI element of the application. The other 10 controllers, with the exception of Item Controller, each control a separate GUI view within the application. There are other helper classes providing functionality or GUI elements that are used in more than one view controller.

Figure 24 describes the controllers and their references to each other. A line connecting two controllers indicates that these two controllers have references to each other, allowing internal messages to be sent between these controllers. While most of the controllers are singletons and are created when the program starts, Login Controller, Trade Controller, and Chat Controller are instanced when needed.

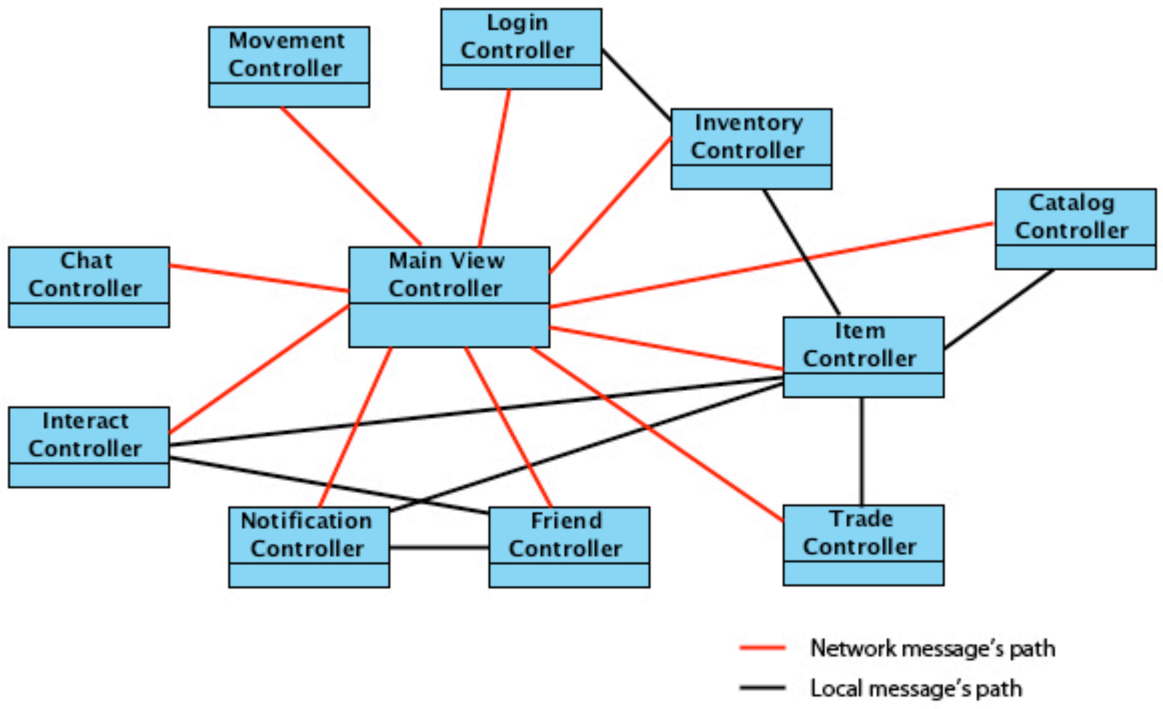


Figure 24 - Messaging paths between the iPhone controllers

4.4.1.2XML Parser API

An XML parser API called TBXML is used in this project to simplify the development process (Bradley, 2009). TBXML is an extremely lightweight and efficient XML parser for the iOS platform. TBXML converts an XML string into a tree, with each node containing an XML tag. This tree structure is used as the internal message structure for the Main View controller to send the message body to other controllers.

4.4.1.3Network Socket API

The iPhone application has to wait for both user input and network input. The entire application is designed to run on a single thread to avoid synchronization problems, so an asynchronous socket API called AsyncSocket is used in this project. This API allows multiple socket connections to run within the single application thread by registering all of the network input within the main run loop. This socket API reads all characters in the input stream until it reaches the designated end-of-message character ('\r'). Each time a message is received, an event is triggered by the socket API and the ReadDataToData delegate method, overriden in the Main View Controller, is called (Voss).

4.4.1.4Main View Controller

As described in Section 4.4.1.1, the Main View Controller has three main jobs. The first job is to act as a message delivery system for both internal and network messages. The controller maintains an asynchronous socket connection to the server. This socket connection is registered to the main run loop to wait for any incoming messages from the server. Once a message is received, the Main View controller uses TBXML to convert the XML string into a tree data structure. This data structure is then used as the internal message for the Main View controller to send to one of the ten referenced controllers based on the message type.

The second job of the Main View Controller is to store important user session information such as username, creature ID, and device ID, so that an XML header string can be created and used for every outgoing message.

The third job of the Main View Controller is to control the main application GUI. The controller contains a tab view containing five tab elements: Move, Notification, Interact, Catalog, and Inventory (Figure 25). When the user clicks on a tab element, the appropriate view with its controller is loaded and displayed inside the tab view.

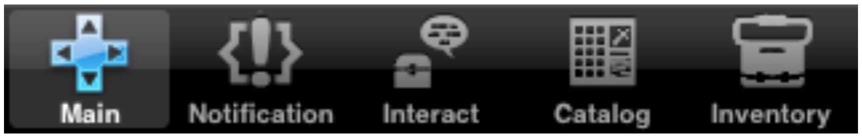


Figure 25 - The tab bar in the iPhone application

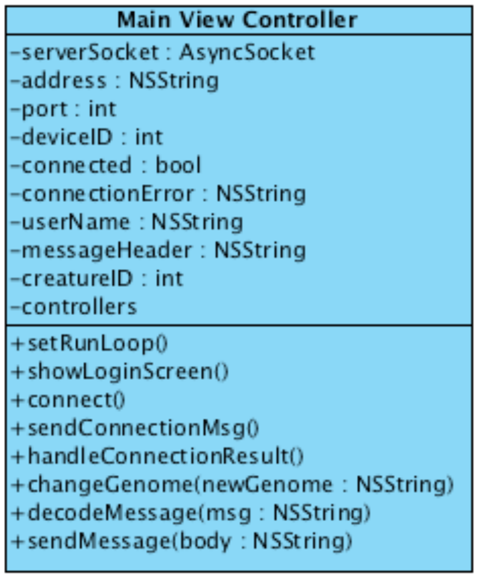

Main View Controller
-serverSocket : AsyncSocket
-address : NSString
-port : int
-deviceID : int
-connected : bool
-connectionError : NSString
-userName : NSString
-messageHeader : NSString
-creatureID : int
-controllers
+setRunLoop()
+showLoginScreen()
+connect()
+sendConnectionMsg()
+handleConnectionResult()
+changeGenome(newGenome : NSString)
+decodeMessage(msg : NSString)
+sendMessage(body : NSString)

Figure 26 - Diagram of the Main View Controller

4.4.1.5Login Controller

The login screen is the starting screen of the application. It displays two input boxes: one for username and another for password. Handling the login procedure is the only responsibility of the login controller. When the login button is pressed, the login controller first checks for a socket connection to the server. If there isn't a working connection, the login controller will ask the Main View controller to establish a connection before it can send the message. After being notified by the main view controller that the socket connection is working, the login controller converts the entered password into a SHA2 64-bit encrypted string and generates the login XML message to send to the main view controller (Information Tech.). The main view controller sends the message with its precompiled header, and delegates the login result XML message for Login Controller to decode. On a successful login, the user is redirected to the Movement Controller and the username and creature ID are sent to main view controller. On an unsuccessful login, the user is given a reason and is prompted to try again.

The other responsibility of this controller is to handle user logouts and the subsequent shutdown of the application on the mobile device. This can be a logout initiated by the user or a timeout from the server. Figure 27 details the general structure of the Login Controller.

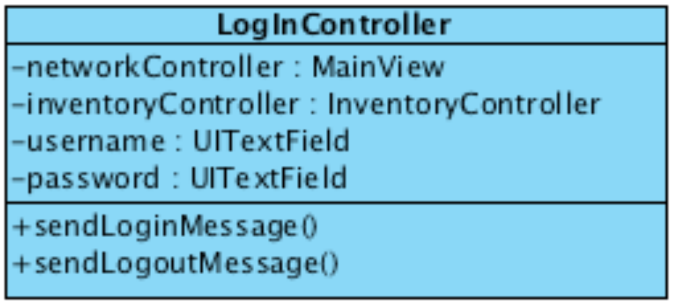


Figure 27 - Diagram of the Login Controller

The login page is a generic Web login interface. Instructions are provided at the top to assist users with the login process. The username and password text field are located in the top half of the screen to leave space for iOS's virtual keyboard. When the user presses the login button, an Activity Indicator element is displayed and animated to indicate the login process is executing (Figure 28).

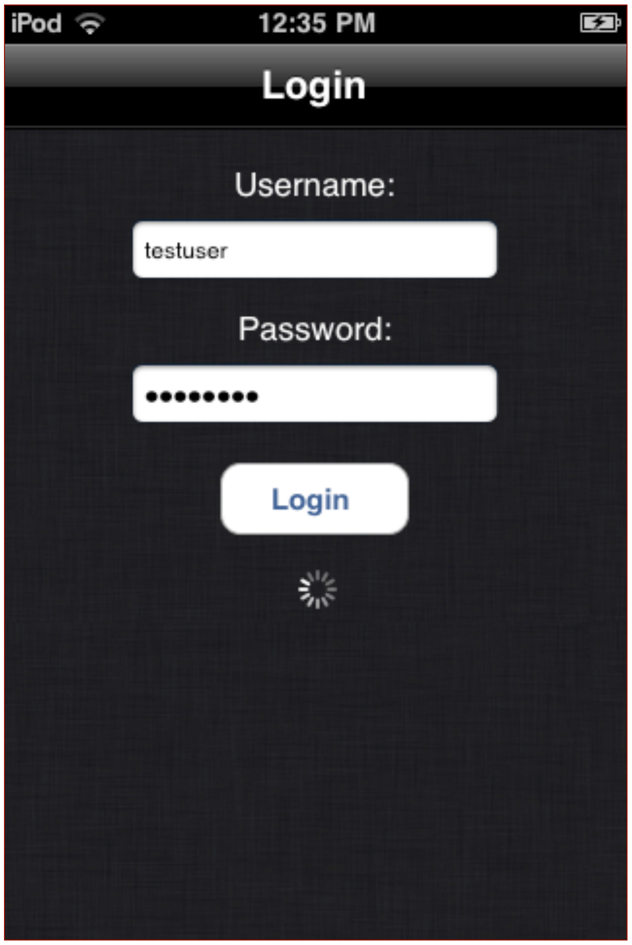


Figure 28 - Login page for an Apple Mobile Device

4.4.1.6Movement Controller

The Movement Controller is responsible for sending messages to the server from the iPhone when the user wants to move the creature. Using the iPhone's touch screen, a user's swipe across the screen is converted into a movement vector, converted to a message, and sent to the server. The movement controller class overrides three methods from the iOS's UIViewController: TouchBegan, TouchMoved, and TouchEnded (Apple, Inc.). These methods help determine the beginning and end points of a swipe gesture and convert it into a movement vector.

The main screen of the mobile device application is the movement control screen for the creature. All other menus can be accessed from this screen, and this screen can be accessed from any menu. On the background, an image of the user's creature is created from the creature's genome string. A small trace of the user's swipe is also displayed to assist the user in making a correct swipe (Figure 29).

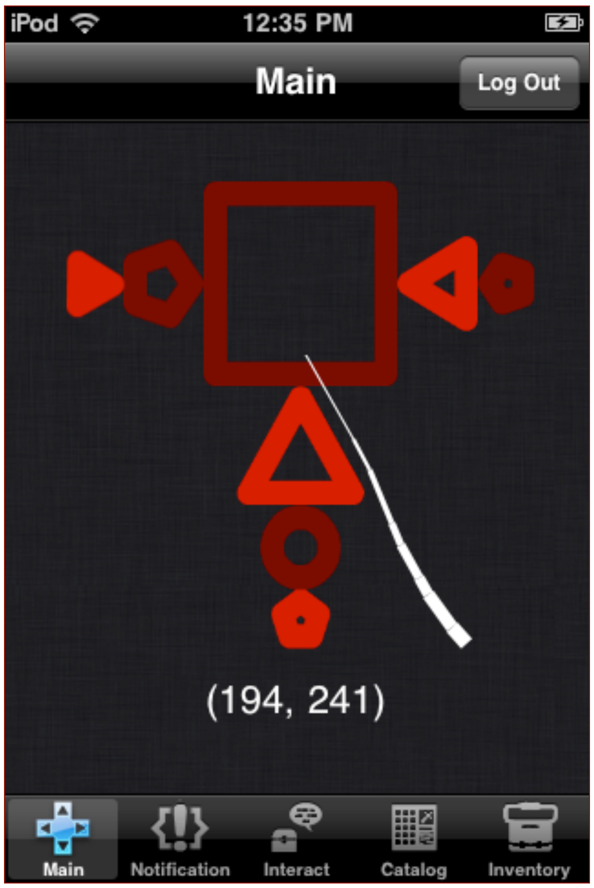


Figure 29 - The iPhone application main screen

4.4.1.7Friend Controller

Friend Controller manages the user's friend list (Figure 30) and provides other controllers with a check method to see whether a creature with a given creature ID is a friend. When the application starts, Friend Controller sends a request to the server for a friend list and decodes the server's response. A user can add a friend in the Interact Controller by sending a friend request to another user. If the other user confirms, the Friend Controller decodes the response message and adds the new friend to the list. A friend can also be added in the Notification Controller if the user accepts a friend request from someone else. The Friend Controller also provides a friend check method for the Interact Controller to indicate whether an interactable creature is a friend.

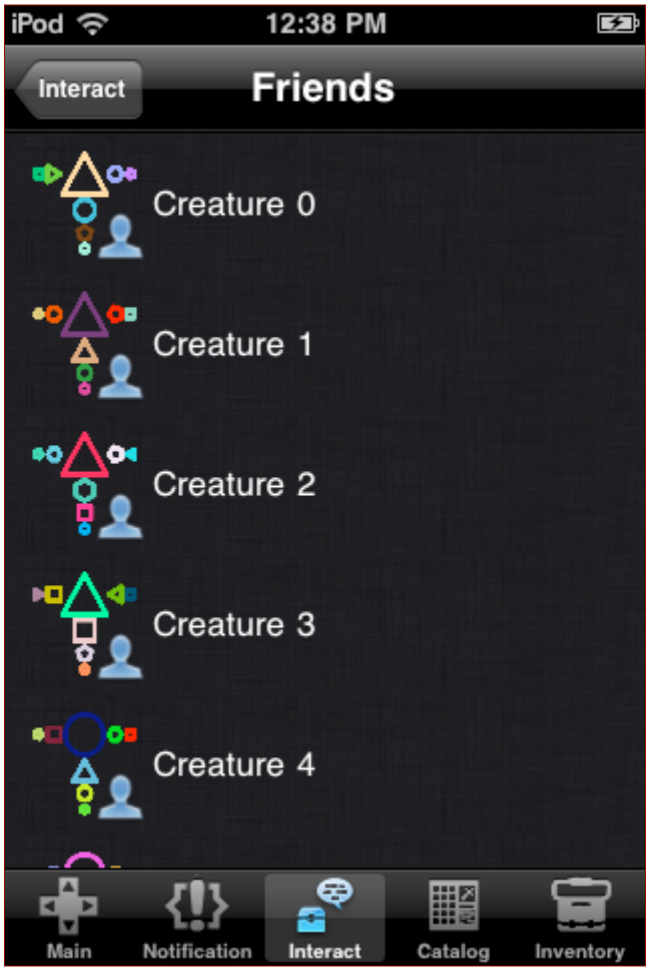


Figure 30 - The friends list displays a user's friends, similar to the interactable creatures list

4.4.1.8 Interact Controller

The Interact Controller controls the third tab view of the application. It contains two views: Interactable List and Interact Options.

4.4.1.8.1 Interactable List

When the user clicks on the third tab button to open the view, Interact Controller sends a request for an interactable list to the server. The Interact Controller then displays the request result, which is a list of all creatures and treasures within a certain distance from the user's creature. Since the Interact View contains a table view element, the Interact Controllers are also implemented as the table view's datasource and delegate. For datasource implementation, the controller keeps a mutable array of interactable objects, which can be either an Interactable Creature or an Interactable Treasure. The table view is then sorted into two sections, one for creatures and the other for treasures (Figure 31). For delegate implementation, the controller overrides a table controller method that handles the table's item selection event. When a treasure is selected, the controller immediately sends a treasure pickup request message to the server and handles the server's response. However when a creature is selected, the Interact Option view is displayed to assist the user in choosing one of the three interact options available.

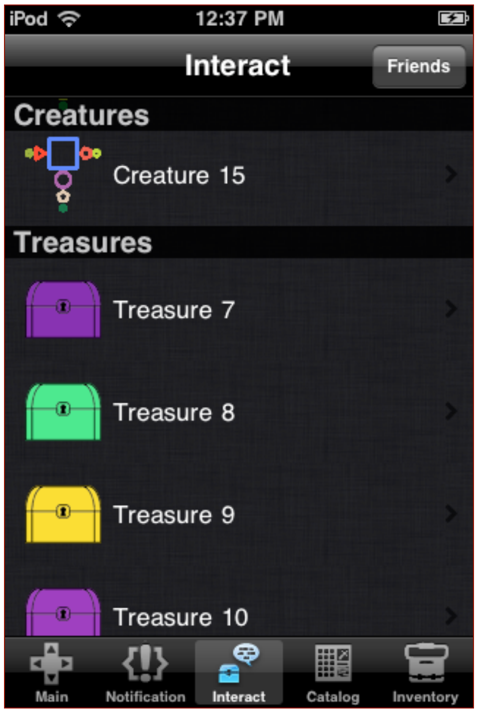


Figure 31 - An example list of nearby creatures and treasures

4.4.1.8.2 Interact Options

When the user clicks on an interactable creature in the interactable list, a view with the creature's name, image, and three interact choices are displayed (Figure 32). A reference to the interacting creature object is also kept for easy access to the creature's ID, name, and genome. When the user selects one of the three choices, the Interact Option controller notifies the Interact Controller to send a request message to the server. The request result is also be decoded by the Interact Controller and then sent to Interact Option controller via a function call. The interact option controller does not communicate with the main view controller directly.

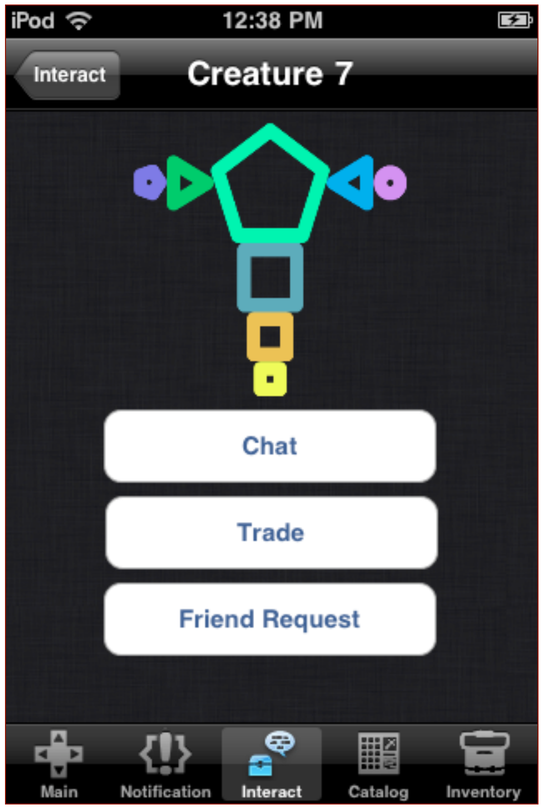
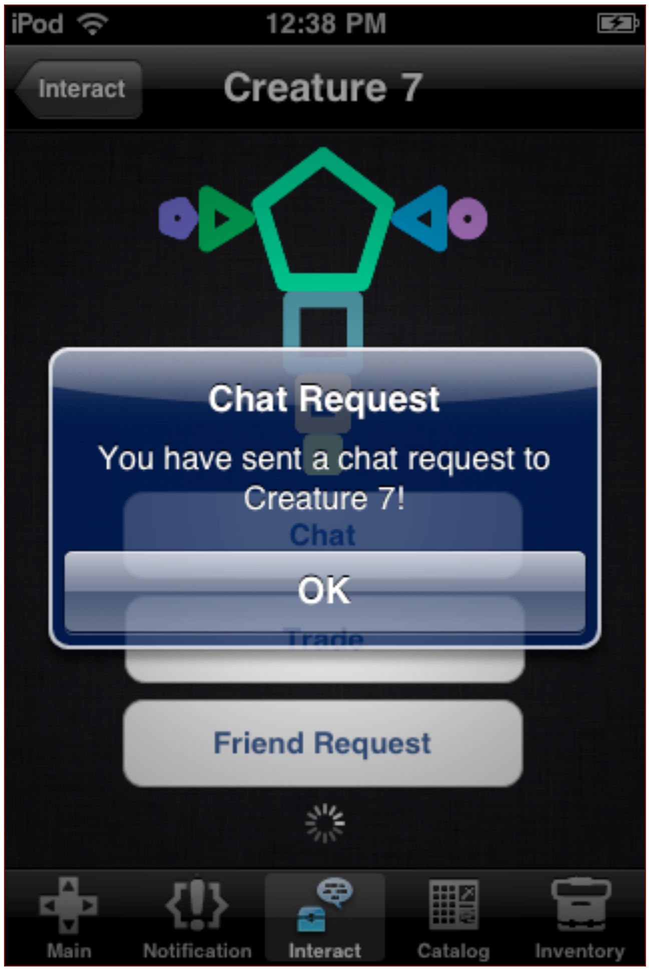


Figure 32 - The basic creature interaction menu on the iPhone

When an Interact request is sent, the user is notified that the request was successfully transmitted to the server (Figure 33). If the Interact request is accepted, the interact option controller creates a chat controller or a trade controller depending on the interact type. If the Interact request is denied, an Alert view is displayed indicating the reason for request failure.



Chat Request

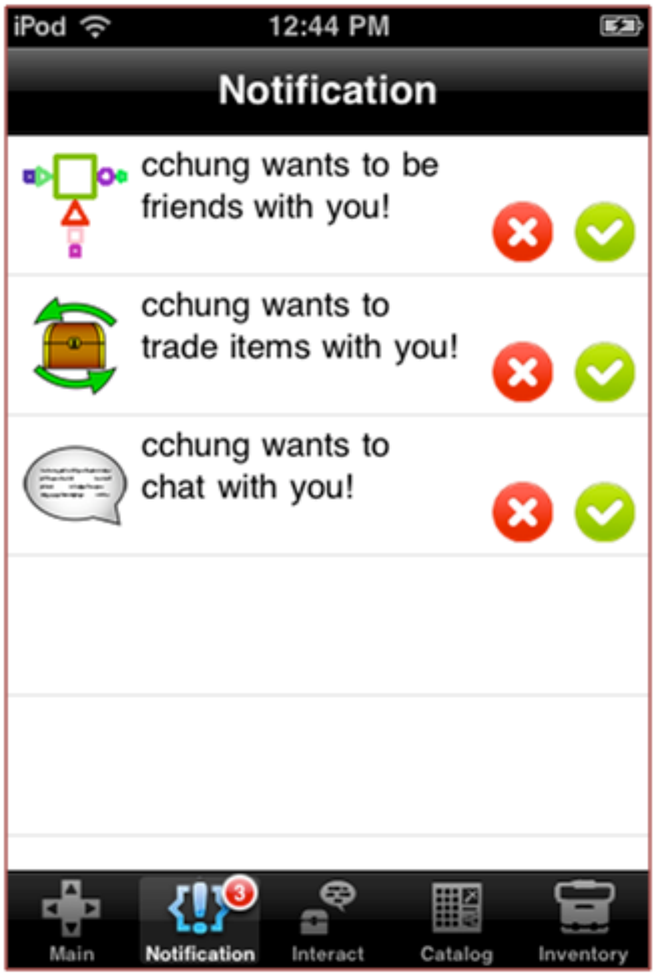
You have sent a chat request to Creature 7!

OK

Figure 33 - User requesting to chat with another creature

4.4.1.9Notification Controller

Notifications are how the user is informed of messages which do not require immediate attention, and are designed not to interfere with the user's current activity (Figure 34). The GUI display is similar to an RSS newsfeed with the newest notifications on top. The notifications that can be replied to (such as a friend request) have the response options built into the notifications themselves.



cchung wants to be friends with you!

cchung wants to trade items with you!

cchung wants to chat with you!

Figure 34 - The iPhone application notifications screen

A user can see that they have a new notification by looking at the number above the notifications tab bar item in any of the other application screens after logging in. If there are new notifications to read, the number of unread notifications is displayed in a small, red circle element called a Badge.

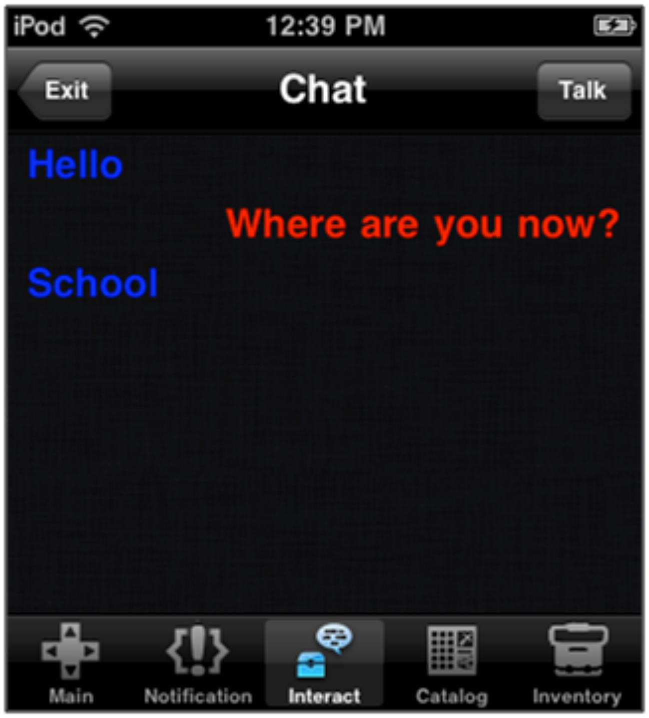
The Notification Controller keeps track of all of the user's current notifications. Each notification contains a notification type (which can either be a chat request, a trade request, a friend request, a friend accept, or a friend reject), a notification ID (which can be the chat ID if the notification is a chat request or the trade ID if the notification is a trade request), and the requester's name and genome. When a notification is responded to, the response was handled at the Friend Controller in the case of a friend request. In case of a chat or trade acceptance, a new chat controller or trade controller is created in the notification controller and an accept message is sent to the server. Finally, the notification is removed from the list.

The notifications screen displays a list of recent events and notifications which require the user's attention. The screen contains a table view with custom table cells, each of which contains an image view, a non-modifiable text field, and two buttons for accept and reject. The image view is used to differentiate between different types of notifications. A friend request has a picture of the requester's creature, while a chat request has the chat symbol, and a trade request displays the trade symbol (Figure 34).

4.4.1.10 Chat Controller

Chat Controller contains all the information about the current chat session, if one is active. The controller can be created from the notification controller or the interact option controller. However, unlike the interact option controller, Chat Controller can communicate with the main view controller directly to send and receive network messages. Chat Controller stores the chat ID as well as a history of each message which has been sent or received so far and which user sent them. When sending a message, the user selects an option from a list of built-in phrases, which is divided into categories for easier browsing. Chat Controller sends a message with the ID number of the phrase, and the receiving mobile device displays that message in the device's main language.

The chat screen looks like a simple instant messaging application. Messages from each user have a different color and alignment to make it easier for the users to keep track of the conversation (Figure 35). The user can create a new message by pressing the talk button, and selecting predetermined phrases from a phrase bank. Phrases are organized into categories for ease of searching (Figure 36). Each category opens a new screen with a list of related phrases. After choosing a message category, a list of related phrases is shown (Figure 36). Users choose a phrase by tapping to send that phrase to the other person. If the number of phrases in the phrase bank increases, the phrase list could be alphabetized and a search method could be implemented.

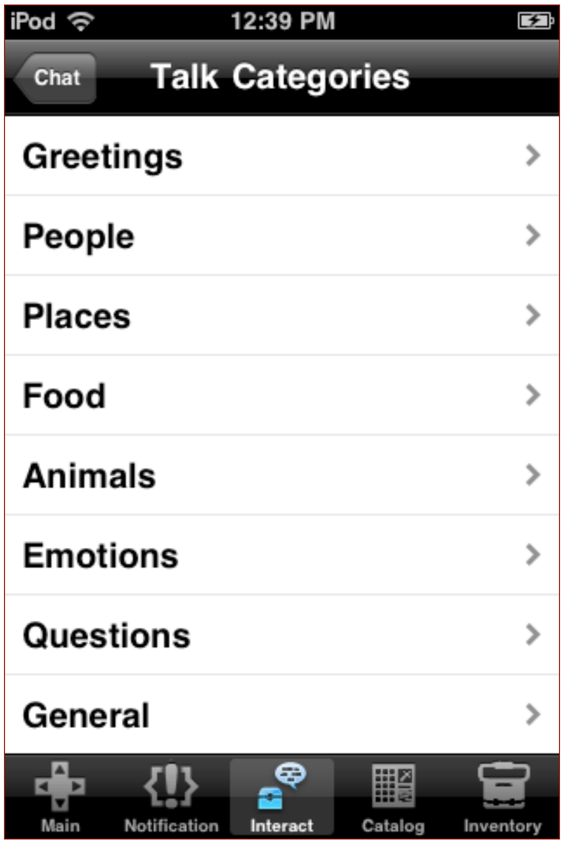


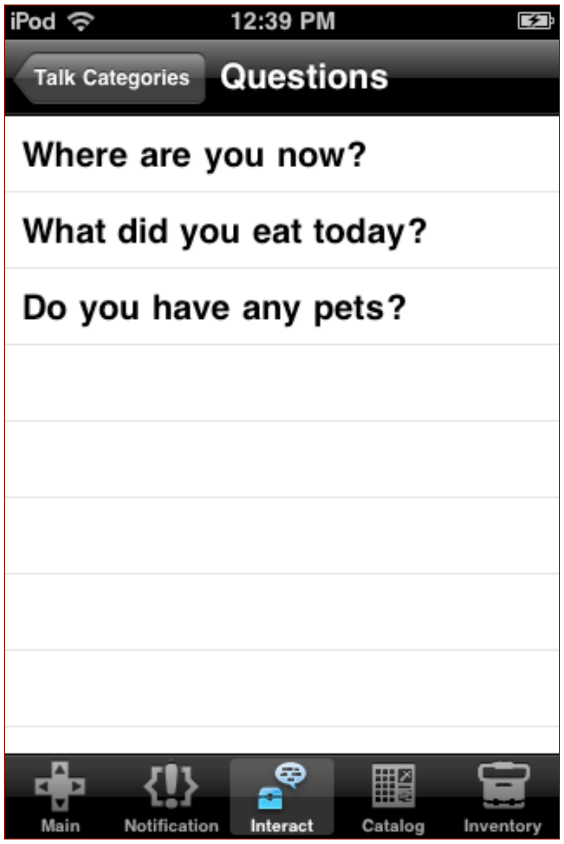
Hello

Where are you now?

School

Figure 35 - The iPhone chat screen





Where are you now?

What did you eat today?

Do you have any pets?

Figure 36 - Chat message categories and messages within a category

4.4.1.11 Item Controller

Since there are many view controllers that need access to the creature's item list, such as Inventory Controller, Trade Controller, and Catalog Controller, the Item Controller class was designed as the central data holder for all item related information from a user. This is also the only controller that does not control any GUI view.

The Item Controller keeps two mutable dictionaries, one for the item list and the other for the item category list. It is initialized when the program starts, and the two dictionaries are filled with item information from an XML-based property list file called Items.plist. Each item on the item list contains an item ID, item name and description strings, item image file name, item count, item category, and item status (unknown, seen, or owned). When the main view controller receives a message that requires updating the item list, the message is sent to the Item Controller to process. After processing, the Item Controller notifies other controllers that the item list has been changed.

4.4.1.12 Inventory Controller

The Inventory Controller displays to the user all items in the creature's inventory. The controller also assists the user in equipping items on the creature and dropping unwanted items from the inventory. It displays a grid view of items to the user (Figure 37) in which the user can select each item to equip or drop. The user selects an item and an equipment slot to equip that slot with the chosen item. Once that is done, an XML message is sent by the Inventory Controller to notify the server of the equipment change. The server then sends back a confirmation message, including a new list of equipment, which is decoded by the inventory controller and the item controller. This makes sure that the user's inventory counts and equipment are synchronized with the server. To drop an item, the user selects an item in the grid view and presses the trash button. The server is notified by an XML message and sends back an update message with the new inventory count.

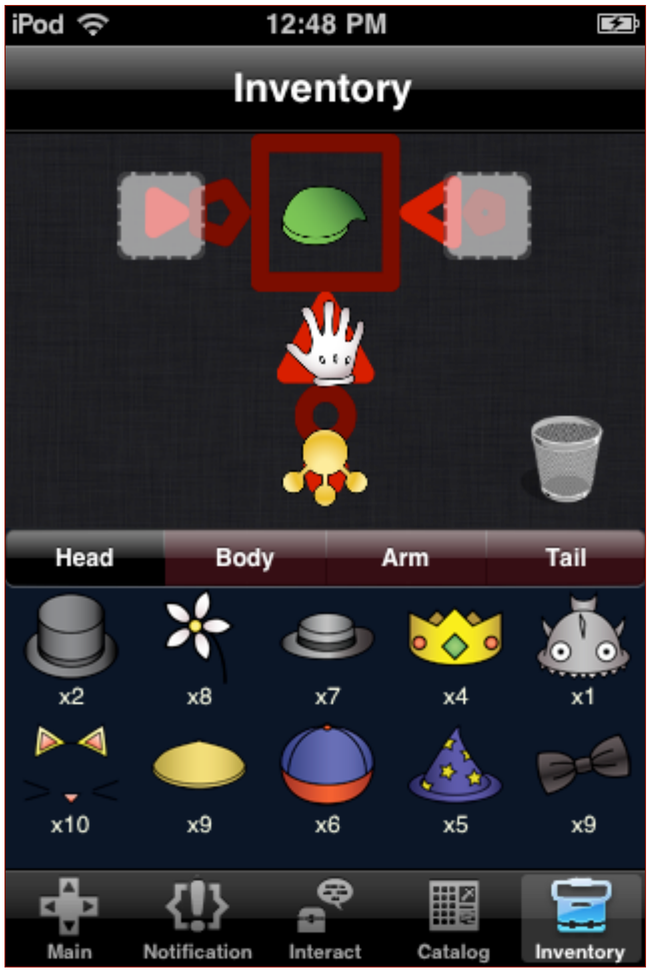


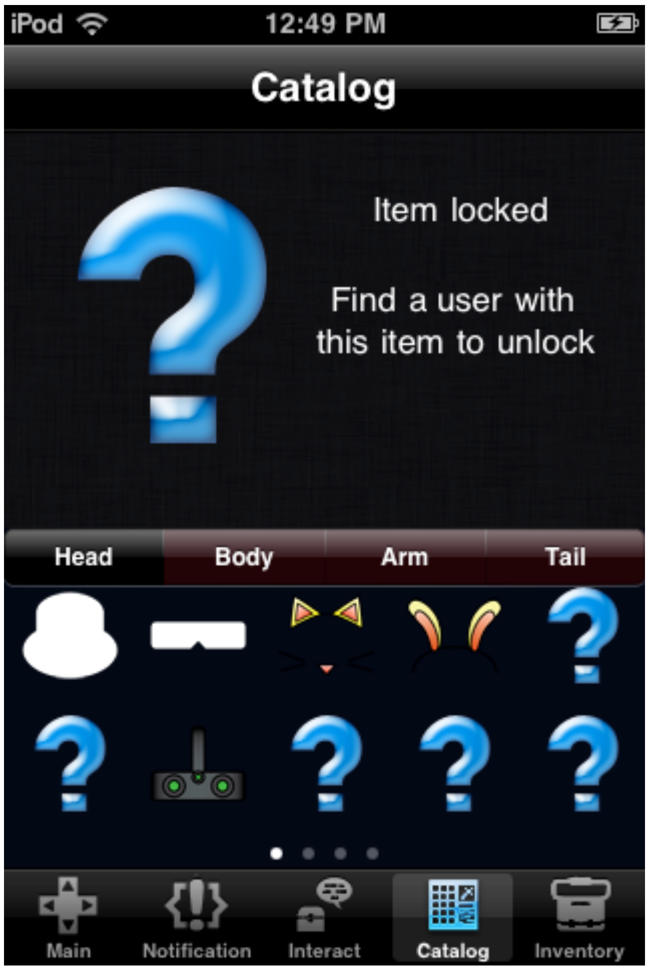
Figure 37 - The main inventory and equipment screen

The inventory view is split into two halves. The bottom half is a standard view to display items in the application called the Inventory Scroll View. The inventory scroll view contains a segmented controller to let the user select the item's category. Below the segmented controller is a scrollable view that contains small subviews of the item's image and count. At the bottom of the inventory scroll view is a page indicator to indicate the number of pages and the current page of the scroll view. When the user selects an item, the view flips to the item details view to indicate that the user has selected an item. This inventory scroll view is also used in the Trade Controller and Catalog Controller.

The top half of the inventory view contains the image of the user's creature with five buttons called "Equipment Slots." When a slot is empty, that button displays an empty frame background at 50% opacity. When it contains an equipped item, the background changes into the item's image at 100% opacity. The drop item button is also a button with a trash can background.

4.4.1.13 Catalog Controller

The Catalog Controller handles the display of the user's item catalog by referencing the Item Controller and displaying all available items in the application in one of three possible modes: unknown, seen, or owned. When the status of an item changes, the server sends an update message to the device containing only the items that changed status. The Item Controller decodes that message and updates its item list, and then notifies the Catalog Controller of the change and prompts it to refresh the item scroll view. The catalog screen displays all of the items the user has seen or owned; unseen items are represented by question marks, and their full data are unavailable (Figure 38).



Item locked

Find a user with this item to unlock

Figure 38 - An unseen item's description in the item catalog

4.4.1.14 Trade Controller

The Trade Controller handles the process of swapping items with another user. When the user's creature is close to another creature, the user has the option to trade items with them on the "Interact" screen. When both users decide to trade, the "Trade" screen comes up. At this moment, the Trade Controller registers the name of the recipient as well as the trade ID with the Trade Controller. The list of items is also acquired from the Item Controller. The Trade Controller maintains two lists containing the items that each person is willing to trade. The user can add items to their own offer list by selecting items in their inventory scroll view (Figure 39). This action triggers the selectItem method and updates the offer list and the inventory count. The user can also remove items by clicking one of the items on the offer list. When the other user updates their list, the friend's offer list automatically changes to reflect the new offer. When both users accept the trade, the server sends a trade complete message with updates to each user's inventory list. The Trade Controller decodes the message and closes the trade screen, while the Item Controller updates the inventory count.

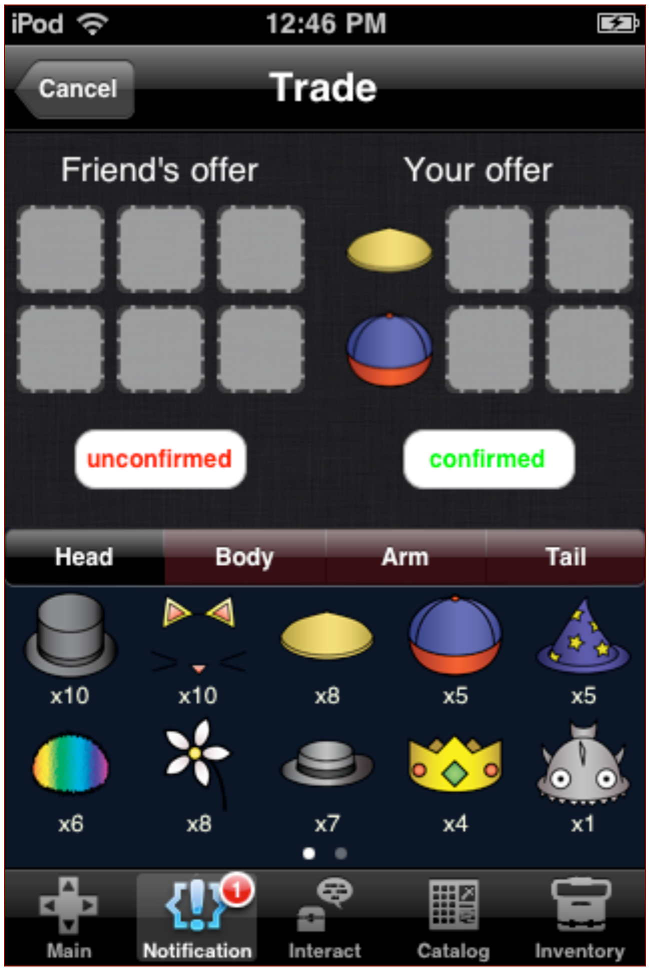


Figure 39 - An item trade between two users

4.4.1.15 Image Caching

In order to reduce the overall application size and increase flexibility in possible future updates, all of the item images are stored externally on an HTTP server. The images are only sent to the mobile device upon request. However, since downloading images from an http server can take a significant amount of time and produce a noticeable delay, an image caching system was implemented to help improve the application's responsiveness. When an item image is requested, the application checks the NSTemporaryDirectory to see if it has an image of the same name (Apple, Inc., 2010). If it does, the local image is loaded and returned. If it doesn't, the image is downloaded from the HTTP server, then saved in NSTemporaryDirectory and returned. This has reduced average image loading time by 80 percent for item images. The use of the image caching system for creature's images was also considered for a possible future application update.

4.4.2 Android

Android application design is shaped significantly by the requirements of the Android OS and the specifics of the Android API. Android applications are defined in Activities, which set up the GUI for a single screen from an XML layout file, and control all logic relating to anything happening in the application while it is on that screen. All Android applications start with a class which extends the default Activity class; any application which switches between multiple GUI views creates an Activity sub-class for each distinct window. This structure is different from the typical Model-View-Controller design pattern, which enforces separation of GUI and logic components. However, out of all the classes within the Android API, only Activities and Services are given access to GUI elements; Activities control foreground GUI windows, while Services are designed to run background processes. Custom classes which do not inherit from those two types cannot access the GUI window at all. When the GUI view needs to change, the application creates a new Activity of the appropriate type, and places the previous Activity on the history stack. However, Android guidelines specify that all Activities should be resumable from the stack (via the standard Android back button), and that the application itself should be able to go to background at any point in time, and safely resume at any later point in time. This presented some unique challenges for the genometry application, which needs to be able to log users out when the application is dismissed.

4.4.2.1Networking

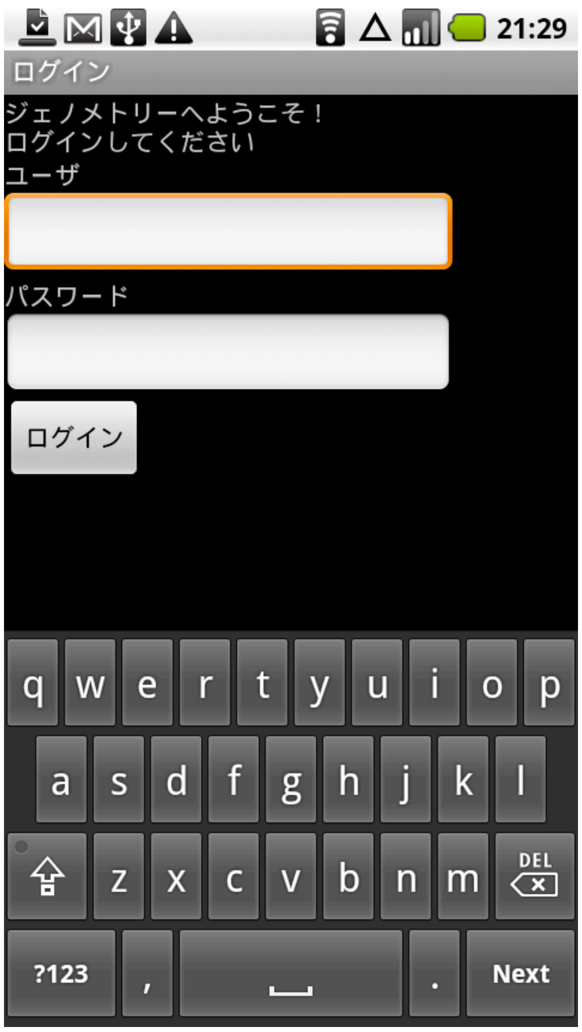
Networking on the Android is implemented via a Service, which is instantiated when the application begins. Other components such as Activities can "bind" to a Service which is already running, and then receive access to any methods exposed through the Service's binder object. Like most Android components, Services differentiate between initial and subsequent start conditions; in this case, although a Service which is not running will start when another component binds to it, this is not the same as explicitly starting a Service. The genometry application explicitly starts the Service only once, from the initial connection activity; on startup, the Service attempts to connect to the server, and notifies the user of any failure to do so. If the device connects successfully, the network service begins a new thread to listen for incoming messages from the server, and the initial startup activity moves to the login activity.

XML messages between the Android and the server are handled via the same XML parsing and constructing package as the server, with new construction methods for the appropriate messages. The ability to copy certain existing Java code from the server was a significant help in developing the Android application, as that development was begun several months later than the iPhone development.

4.4.2.2Login

The login activity, as well as the connection activity before it, is explicitly prohibited from being pushed onto the activity history stack, to prevent unwanted reconnect and re-login attempts. Explicit permissions and prohibitions are set on a per-activity basis in the application's manifest file, which is an XML file detailing the specifics of each Activity and Service to be included in the application bundle.

The GUI is a standard two-field login screen (Figure 40); when the button is pressed, the password is encrypted using the SHA-256 encryption algorithm included in the java.security package, and placed into a login XML message to be sent to the server. In order to send messages to the server, each Activity needs to have access to the network service, so all activities within the application are inherited from custom parent Activity classes which automatically bind to the network service on creation. If the user's login attempt is unsuccessful, the user is notified via a "Toast" notification, an Android-specific text-only notification which appears at the bottom of the screen for a few seconds, and is incapable of taking focus. Many of the notifications in the application are sent as Toasts, because of their inability to accidentally steal focus from the user's current GUI selection.



21:29

ログイン

ジェノメトリーへようこそ！

ログインしてください

ユーザ

パスワード

Figure 40 - The Japanese language login screen

4.4.2.3Main Tab View

The majority of the activities within the application are held within a tabbed view, allowing users to select the tabs to switch activities. In addition to GUI functionality, this enables the application to pause launching an activity until information is received from the server by implementing a wait method in the tab view's tab change listener. This technique is used when switching to the interactables tab, for example, which needs to receive the current interactables list from the server before the activity can be started. Activities developed later use a Handler object held by the network service, which is able to pass the Activity information when it is received; it is an eventual design goal to change older Activities over to this method, but since the current implementation works, that goal is low priority compared to completing the functionality of the application.

4.4.2.4Movement

Movement on the Android is handled on the movement tab, which listens for the user's swipe gesture, which is then normalized to [-1, 1] (representing the magnitude of the user's swipe relative to the entire screen) and sent to the server as a creature movement message. A gentle, short swipe straight down might be (0.0, 0.3), while a strong, quick swipe up and right might be (0.9, -0.8); positive y is considered down by the gesture listener. The background is intended to hold an image of the user's creature, generated from its genome. The batik SVG library was chosen, as it is the most widely used Java SVG library and has the best support, but it unfortunately is not compatible with the Android's Dalvik virtual machine (Apache). The graphics are intended to be generated on the server (using the same Java code) and downloaded by the Android via HTTP; the final graphics code is available, but was not integrated into the server before the end of the project. At the moment, the background of the movement screen is a placeholder image (Figure 41).

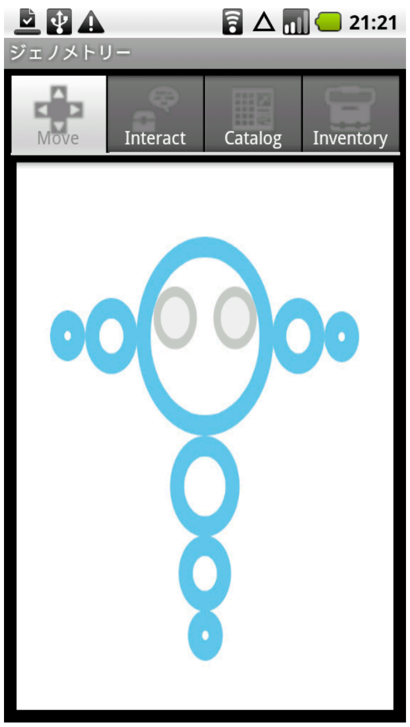


Figure 41 - The Android movement screen

4.4.2.5Interactions

When the user selects the Interactions Tab within the main tab view, the tab controller sends a request for the creature's current interactables list to the server, and waits to load the list view until the list has been received. If the list fails to come in from the server, an error Toast is displayed, and the application does not change from the current tab.

When the list is received, the interactables list view is populated first with creatures (with their corresponding usernames as the list items) and then with treasures, which are numbered sequentially and colored to match the color of the treasure chest on the display (Figure 42). If the user selects a treasure, a message is sent to the server requesting the items in that treasure chest, and a Toast is displayed when the return message is received, informing the user of what was in the chest, or that the chest was taken by another user first.

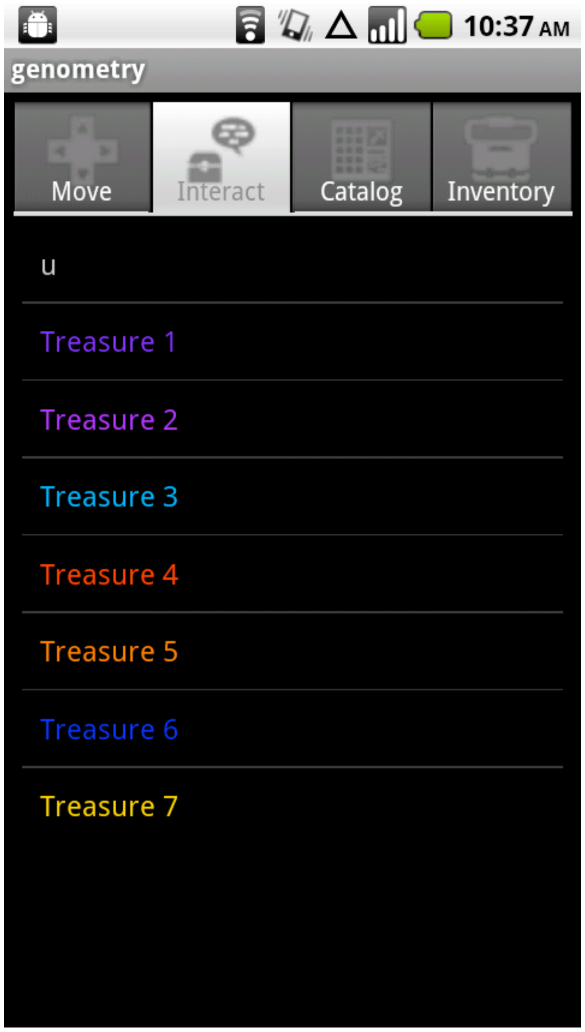


Figure 42 - An interactables list with a user ("u") and treasures

If the user chooses to interact with a creature, the options presented are Chat, Trade, and Add Friend (Figure 43). Pressing one of these buttons will send the corresponding request message to the server; pressing it again will send a cancellation of that request.

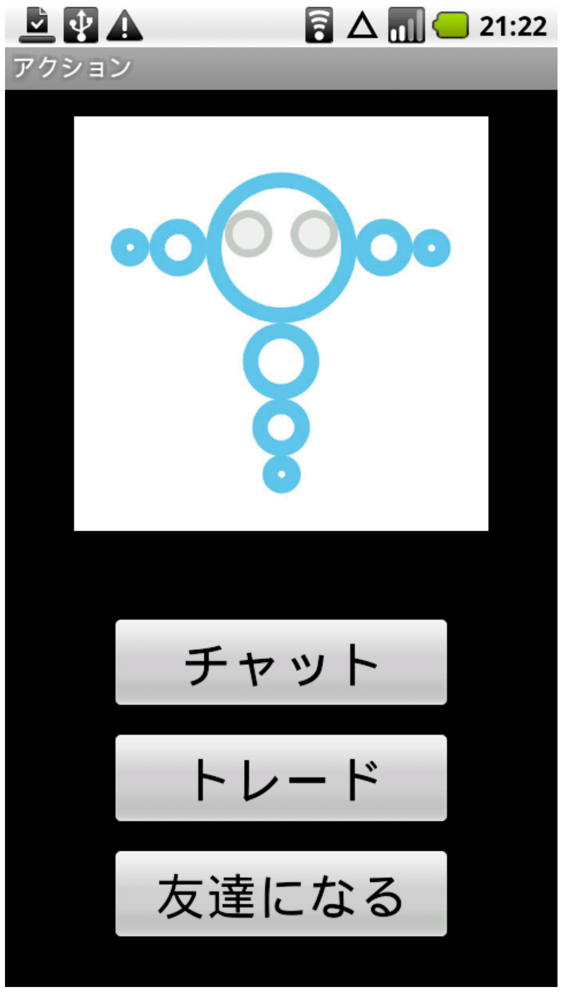


Figure 43 - The Japanese language creature interactions screen

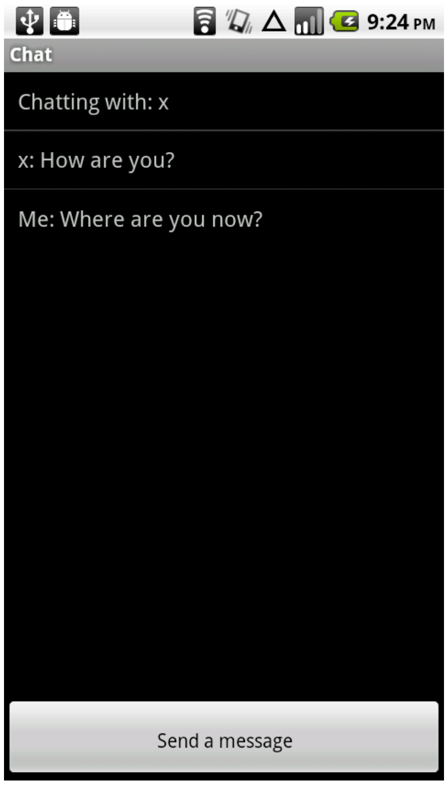
4.4.2.6Notifications

Notifications on Android phones are handled through a built-in notification system, which can be pulled down from the top of the screen at any time, and is part of the main OS. Each notification is given an Intent, which is an object that holds an Activity to be launched, as well as any data which need to be passed to that Activity. For chat, when an incoming chat request is received, a notification is generated which contains an Intent for a new chat Activity with the requesting user. The notification informs the user that someone wishes to chat, and that clicking the notification will accept the chat; notifications can be ignored by dismissing them with the built-in “Clear” button.

4.4.2.7 Chat

When a user’s chat request is accepted, or when a user accepts an incoming chat request, a new chat Activity is begun. Because Activities maintain persistent state once they have been started, and generally cannot be explicitly killed unless the OS itself decides to kill it in order to free up memory, the chat Activity automatically wipes any chat state which may have previously existed. Additionally, the Activity checks to see if the chat strings have been loaded from the localized XML files, and loads them if they haven’t; loading these strings requires checking every string resource in the application to see if it is chat-related so it is only done once. Because Android applications use localized XML files to hold all strings intended for display within the application, the genometry Android application is capable of automatically displaying chat strings (and any text in the program) in either English or Japanese (See Appendix F for all chat strings). Lastly, the chat Activity instantiates a Handler object in the network Service, which enables the Service to send incoming chat messages to the Activity when they are received.

The chat screen is a simple list view which displays the history of the current chat (Figure 44). List views in Android applications are bound to List Adapters, which take the contents of an array and make each element an element of the list view. Adapters also listen for changes to their arrays, and automatically update the list view when a change is made to the underlying data. This allows incoming messages to be sent to the Activity from the network Service and added to the chat’s history array, which automatically updates the GUI for the user, displaying the new message. At the bottom of the screen is a button which allows the user to browse the chat message categories; selecting a category will take the user to the list of messages in that category, while pressing the back button will return the user to the main chat view (Figure 45). Going back from the chat view at any point closes the chat, and the user must re-initiate the chat to continue.

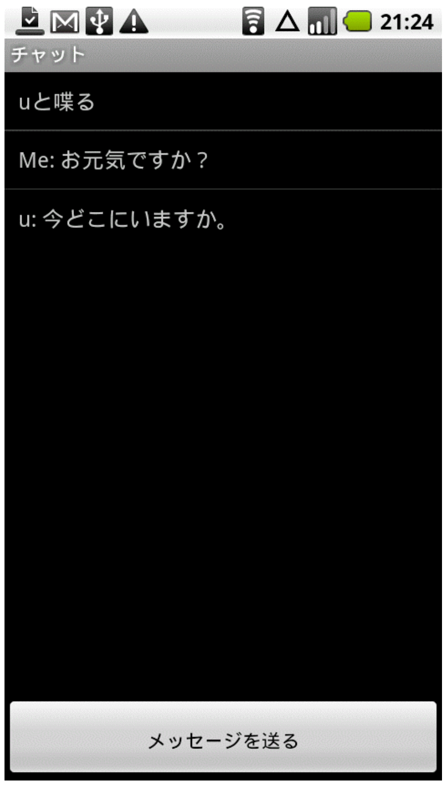

9:24 PM

Chat

Chatting with: x

x: How are you?

Me: Where are you now?


21:24

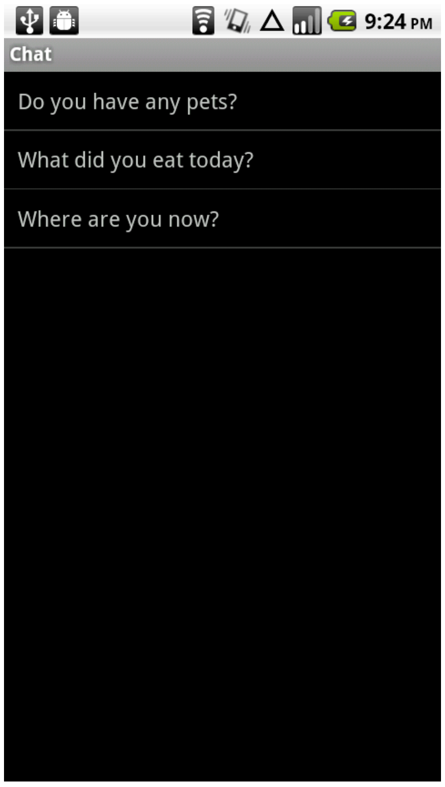
チャット

uと喋る

Me: お元気ですか？

u: 今どこにいますか。

Figure 44 - A chat between "x" and "u" in English and Japanese

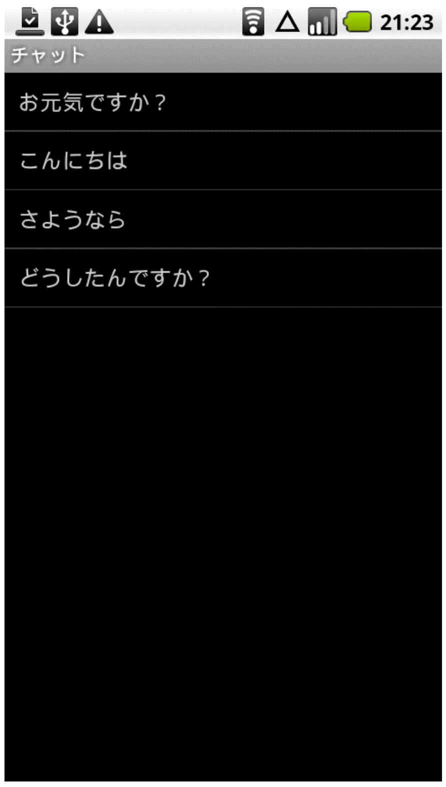

9:24 PM

Chat

Do you have any pets?

What did you eat today?

Where are you now?


21:23

チャット

お元気ですか？

こんにちは

さようなら

どうしたんですか？

Figure 45 - The "Questions" and "Greetings" categories in English and Japanese, respectively

4.4.2.8 Inventory

The state of the user's creature is held in a singleton object, and stores information such as the creature's ID and genome, as well as its current equipment and inventory. The inventory tab requests the inventory from the server if it is not present yet, and then displays a split screen which allows the user to browse the creature's inventory and equip or remove items (Figure 46). The bottom half of the screen is a scrolling grid view, each panel of which is a custom View item, the basic parent class of all Android GUI windows. The inventory View item combines the item's icon with a small text box displaying how many of that item the creature owns, e.g., "x7." These View panels are generated by a custom Adapter. Like the chat Adapters, it manages an array of data and feeds it to the grid view; however, instead of simply converting strings to text boxes, the inventory Adapter is based on the array of inventory items, and returns the appropriate custom View described above. The main inventory grid view can be sorted by categories using the five buttons above the grid: All, Head, Body, Arm, and Tail.

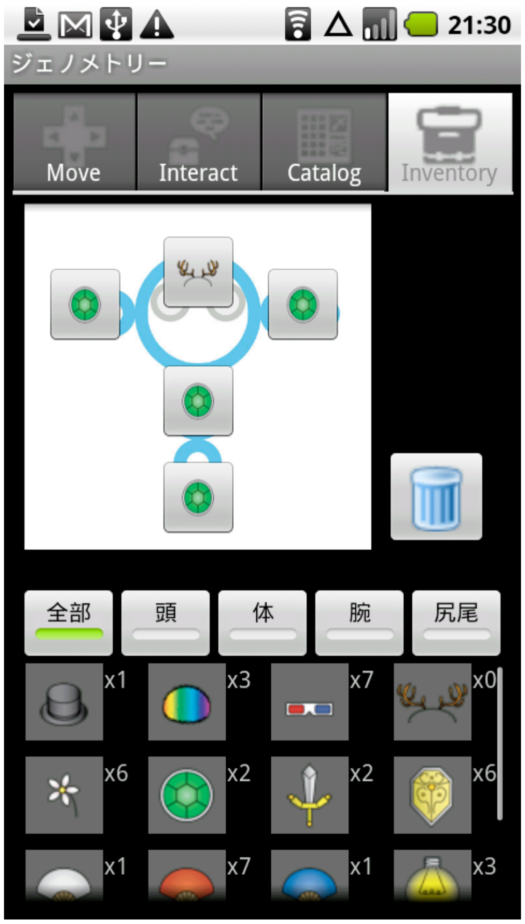


Figure 46 - The inventory view, with all categories enabled

The top half of the screen is an image of the creature, with buttons placed on the five equipable locations: head, body, two arms, and two tail slots. Each of these buttons displays the item currently equipped in that slot, or an empty box if nothing is equipped there. The user can equip items by tapping an item in the bottom half and then tapping the button for the slot the item should go in. If an item was previously in that slot, it is placed back in the inventory and the new item is placed in the slot. If the user taps an equipped item without having selected an item in the bottom half, that item is removed and placed back in the inventory. To reduce confusion, if an item is selected in the bottom half, it is surrounded with a dotted outline; tapping the selected item again will deselect it. The top half also contains a trash button, which allows users to discard items; discarded items are turned into treasure chests in the environment. Each equipment and inventory change is relayed to the server, so that the database and the terminals can be updated.

4.4.2.9Catalog

The catalog screen is very similar to the inventory screen, with a split screen containing category sorting buttons and a grid view in the bottom. The grid view uses a variant of the inventory Adapter which does not include text for item quantity, and is based on the creature's catalog array. Items with a status of "unknown" are returned as a default empty graphic (currently a circle/slash symbol), while items with a "seen" status are returned as silhouette images, and items with an "owned" status are returned with their full-color images.

When an item is selected in the bottom half, the top half changes depending on the item's status. If the item is "owned," a larger, color image of the item is displayed (Figure 47), along with its name and description in either English or Japanese, depending on the device's region settings (See Appendix E for all item names and descriptions). If the item is "seen," a larger silhouette is displayed, and the name is displayed as "???" while the description is not displayed at all (this is a slight design change from the iPhone version, which was built first; it was later decided that the item details would only be shown for "owned" items). Selecting an item with "unknown" status clears the top half of the screen entirely.

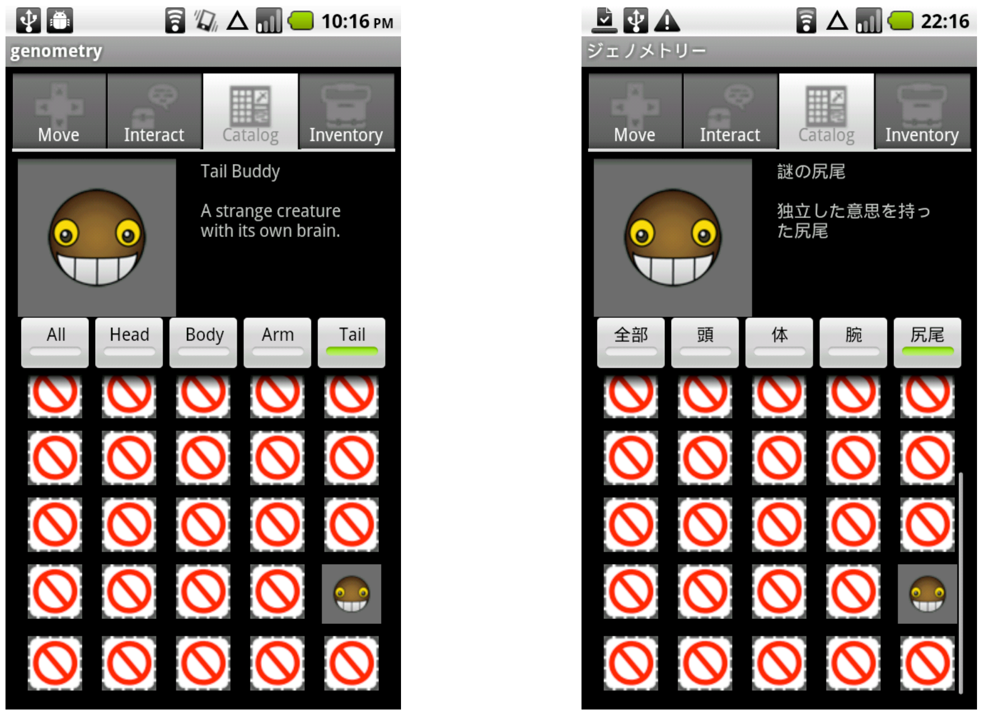


Figure 47 - The details for the "Tail Buddy" item

4.4.2.10 Resource storage

Android applications store string resources in XML files, which can be easily localized and updated independently of the application code. When the XML files are changed, the Android SDK automatically generates a file full of integers which point directly to those resources as memory offsets on the application's stack, and the strings can be referenced easily within the code, e.g., "R.string.welcome_message." GUI elements are also declared in XML layout files and referenced similarly from Activities which use that layout, e.g., "R.id.login_button." This allows layouts to be easily updated without interfering with the application code, and lets certain GUI components be reused easily, such as tab and list elements.

Images in an Android application can be handled in one of two ways: they can be placed in the drawables directory, and referenced by auto-generated ID variables, or they can be placed in the assets directory, and accessed by filename. Both of these directories are built in to the default Android application structure. The genometry application stores images for items in the assets directory, which is compressed and packed in with the main application. For 140 items, with small and large icons, color and silhouette, the application size only increases around 2-3MB, compared to the 8-10MB the images occupy uncompressed. Given this efficiency, bundling the images with the application seemed a better choice for the Android than downloading the images remotely.

5 Art

The main goal for the visual aspects of the networked public displays was that the art style and artistic elements were universal for both WPI and OU. A universal art style is one in which the characters, environment, and other visuals are represented by universal human objects and themes such as human emotions, mathematical concepts, and any other aspect of humanity that is common among every person on the planet. In order to foster communication and collaboration without the need for direct conversation users need to be able to relate with the visuals without confusion. What the users see should not act as a barricade for meaningful communication between Japanese and American students, and anyone else in the world that may use the system.

Another goal was to make the visuals attractive to a wide audience of users, not only American and Japanese users but also casual and hardcore game players. The target audience was planned to be as wide as possible; the visuals should not appeal to only a certain type of person so many different types of people will want to use the system. The visuals must have the potential to be of interest to anyone who passes by the display on either the WPI or OU campus. Having universal visuals helps ensure that the content is not geared towards certain types of people and does not exclude others.

The fact that the display is public and can be accessed by anyone who walks by has influenced a lot of the design decisions, including the art direction. Making the art style universal prevents potentially offensive culture-specific content from being displayed and reduces the possibility of displeasing certain people.

5.1 Creatures

Creatures are the main component of the terminal application since they are what users pay the most attention to. Therefore, the overall style is simplistic with many different combinations of appearances to appeal to a wide variety of users.

5.1.1 Creature design

The creature design was initially inspired by the game *flOw* (2006). A creature consisting of a series of linked shapes gave the application the simplicity and universality that was desired for a system that was to be deployed in two different countries (Figure 48). One of the features of the terminal application is changing aspects of creatures, such as body parts and colors, when interactions between users take place. This design allowed for a great deal of customization, since each shape and the colors of those shapes could be changed easily and uniformly.

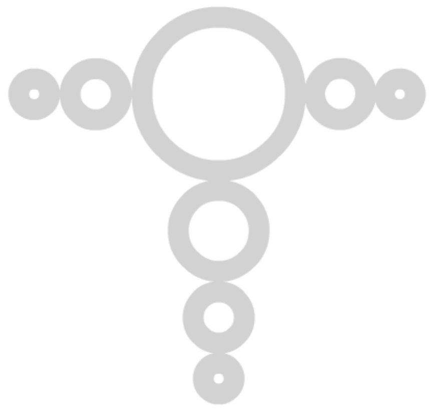


Figure 48 - The basic creature created in Adobe Flash, with all circle body parts and no color

5.1.2 Creature building

The creatures were built entirely in Adobe Flash using vector shapes and a bone armature that acts as an animation rig. Any given creature has eight body parts: a head, four arm shapes, and three tail shapes. Each body part can be a circle, triangle, square, pentagon, or hexagon, depending on the answers provided by users during initial set-up of an account. The shapes that were created were initially colored gray and colored through code when the application is running.

5.1.3 Creature animation

Simple animations for the creatures were created using the timeline and the creature's bone armature in Flash. The creature was animated to give the appearance of fluid motion; the appendages sway back and forth to simulate a swimming behavior. Two animations were created for creatures: a move animation and an idle animation.

5.2 Items

Items are another main component of the system. Users can equip, drop, or trade items with the goal of either customizing their creature's appearance or completing the item catalog. There are many different items to allow unique customizations and to make completing the catalog somewhat of a challenge.

5.2.1 Item design

The items were designed to complement the creatures and act as visual augmentations to the creatures' appearances (Figure 49). Once again, universality was taken into account during the item designing process. Each item was also given a name and a short description to be used in the catalog feature of the mobile device application. There were a total of 140 items designed (and later created), with each category of head, body, arm, and tail items containing 35 items. Appendix B contains the item images, while Appendix E contains their names and descriptions in English and Japanese.

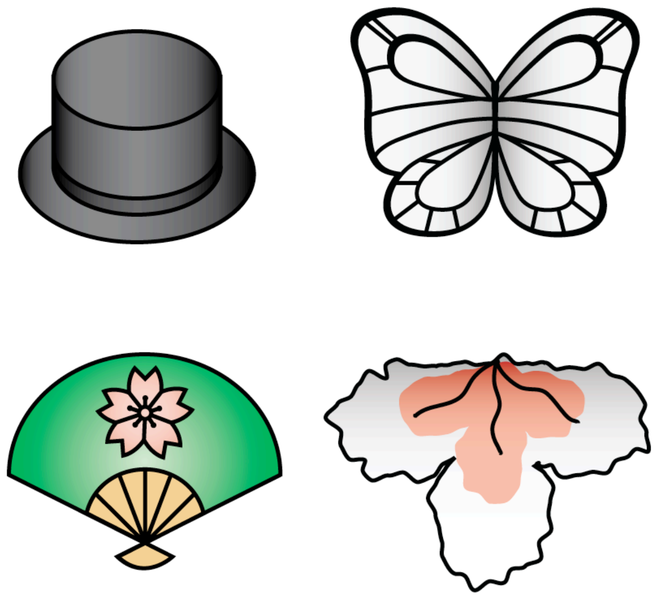


Figure 49 - A head, body, arm, and tail item (counter-clockwise from the top-left)

5.2.2 Creating items

Each item was created in Adobe Illustrator on a 480 by 480 artboard. To keep a consistent style, each item was given a 5 pixel black stroke and was filled with a simple colored gradient. Since each item was to be used in both the Flash terminal and the mobile devices, there were two pipelines for sending the items to each outlet.

For the Flash terminal, each item was converted to an Illustrator symbol and imported into Adobe Flash. For the mobile devices, each item was first exported as a PNG file. Several Adobe Photoshop actions were created to resize items and create silhouette versions. Several Photoshop batch operations were then used to automate the process of resizing each item into two different sizes and creating silhouette versions of the items.

5.3 Environment

The environment is the main playing field for the terminal application. It is a background image that the creatures exist on top of, so it is passive and does not affect the use of the system (Figure 50).

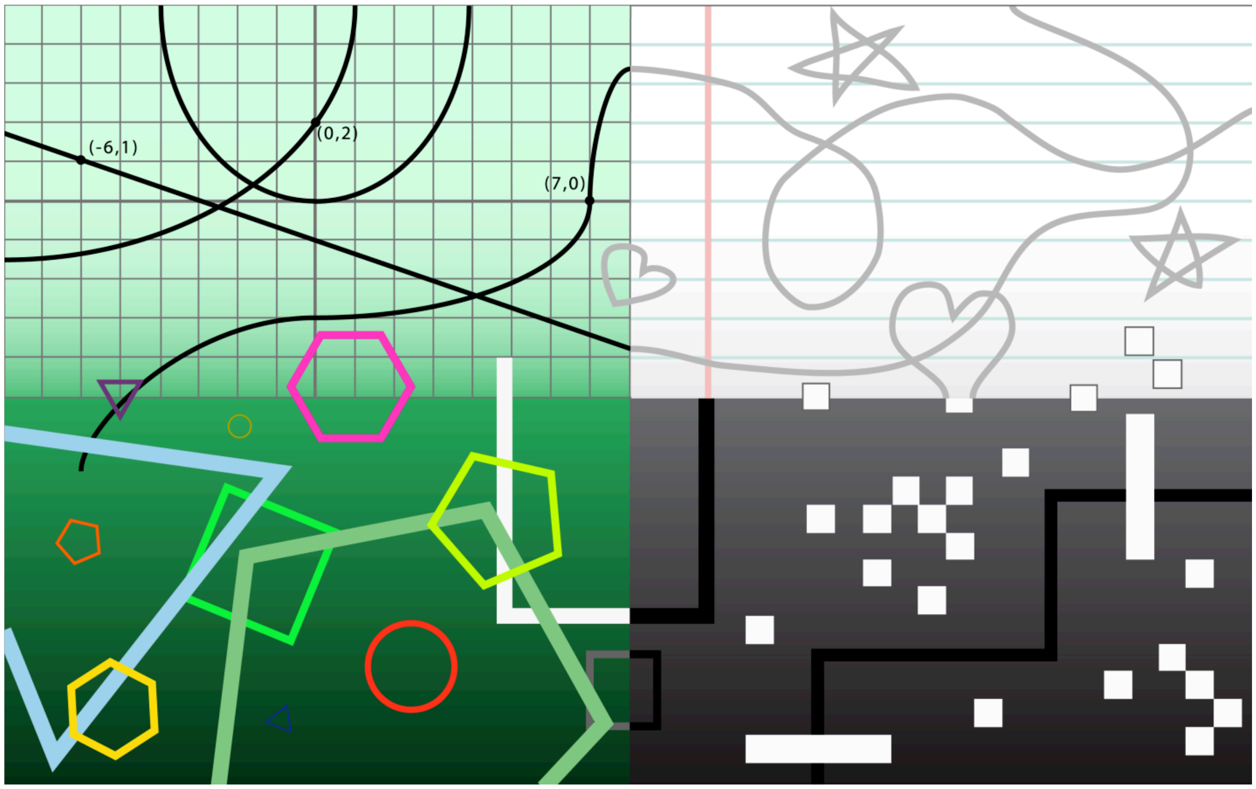


Figure 50 - The entire environment background image, with 4 separate "regions"

5.3.1 Creating environment

The environment image used as the background of the Flash terminal was created entirely as vector shapes in Adobe Illustrator. Basic strokes and shapes were created on simple gradient backgrounds to imitate the appearance of lined white paper, a mathematical grid with algebraic functions, colorful geometric shapes, and a grayscale pixel scene. The background was split into four separate regions to give users a sense of orientation. The image was resized to be large enough to fit the size of the world on the Flash terminal, which was to be shown on a 1080p resolution display.

Several items were also created for the Flash terminal (Figure 51). These items include the interactable treasure chest and ball. These items were also created in Illustrator in the same way as the wearable items.

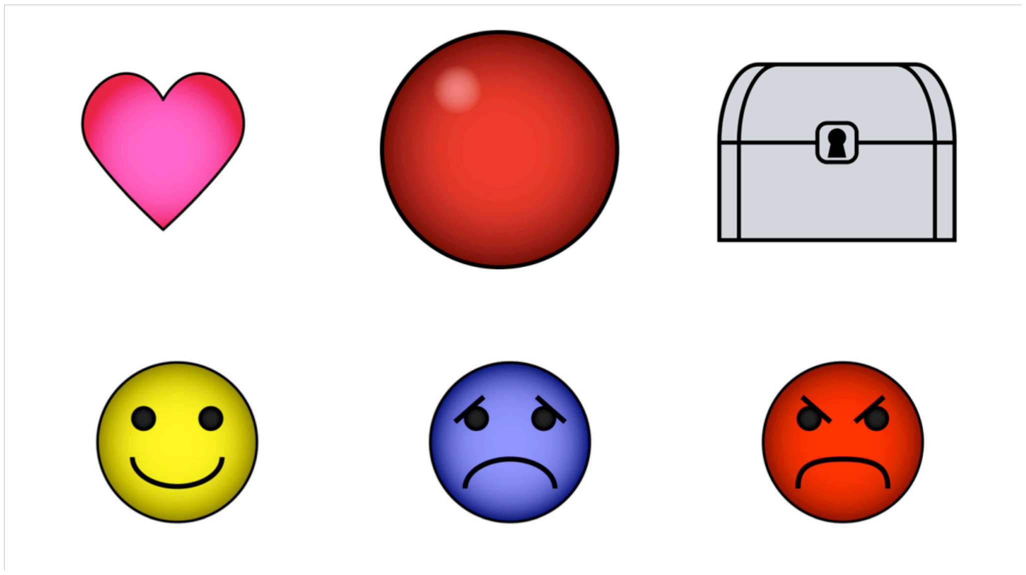


Figure 51 - Some of the terminal application emoticons and items

5.4 GUI

Aside from the main art for the system, some user interface design mock-ups and terminal application emoticons were created. These assets, however, did not have a large impact on the system design.

5.4.1 Mobile device interface design

During the design process of the project, many user interface mock-ups were created as a basic design of the mobile device applications for iPhone and Android (Figure 52). The interfaces were designed for the iPhone but the basic functionality was designed to be easily transferred over to Android. All GUI mock-ups can be found in Appendix D.

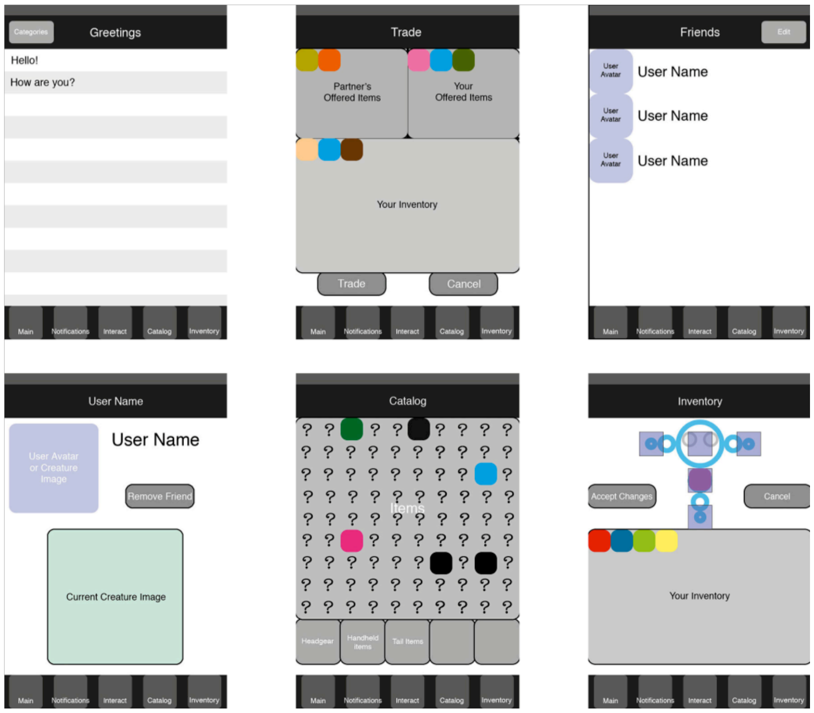


Figure 52 - Several GUI mock-up screens for the iPhone application

5.4.2 Emoticons

In addition to the treasure chest and ball items, several icons were created for the Flash terminal (Figure 51). These icons included emoticons for different emotions (happy, sad, angry, surprised, scared, laugh, love), trade, and chat. All emoticons were created the same way as the interactable items and wearable items. Unfortunately, the emoticons were not used in the final implementation of the system.

5.4.3 Mobile device icons

Since the interface for the mobile device applications relies on a tab bar structure for organizing tasks, a set of custom tab bar icons was created. These icons were created in Illustrator and further edited in Photoshop (Figure 53). The full set of mobile device images can be found in Appendix C.

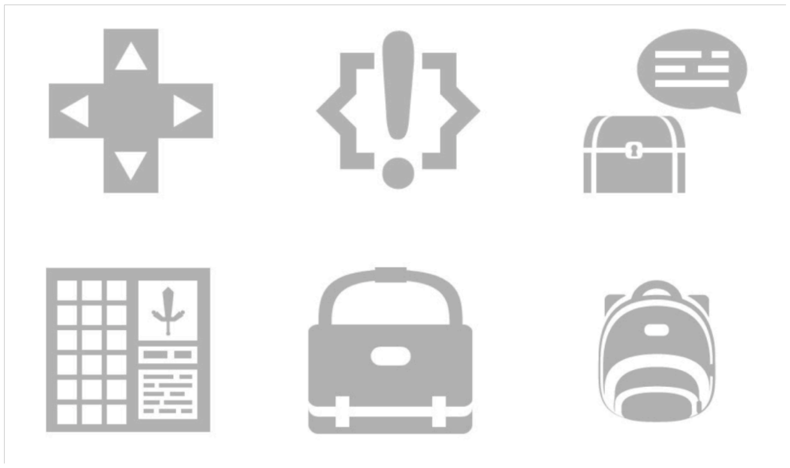


Figure 53 - Mobile device icons for the tab bar interface

The iPhone OS automatically adds effects to tab bar icons, such as gloss and the selected and unselected appearance of tabs. Therefore, each icon was only given a single color fill and alpha transparency and the iPhone OS assimilated the tabs into Apple's intended visual design.

The Android development environment handles tab bar icons a little differently. Each icon must have two versions: a white "unselected" version and a gray "selected" version. Android provides steps for formatting icons to their guidelines that were used to apply Photoshop styles to the existing icons.

6 Post-mortem

Though many of the desired features were never implemented due to a lack of time, overall, the project was still a success. The goal was to create a system which could be controlled with mobile devices and viewed on a public displays around the world. The application itself was designed for the purpose of implementing that setup. There were many obstacles that had to be overcome to complete the project and many possible expansions on what has already been built.

6.1 Initial goals

The goal of our project was to design a virtual world where users could control a personalized character through the use of a handheld mobile device. The goal of the system was to encourage communication and interaction between users by allowing characters to interact with each other. We intended to appeal to a wider audience by adding a variety of ways for characters to interact such as exploration, battle, and trading. The world was to be accessible from any networked public display, initially installed at WPI and OU. Other initial goals were to create an application for a large public display that would use mobile devices as controllers and would allow for a high degree of customization, and to create a fun environment which would make it possible for students from either university to meet. Furthermore, we wanted to complete a deliverable product which would demonstrate every aspect of the system as a proof of concept. We also carefully designed the system to be extensible for future work.

Table 2 outlines the design goals of the project which were initially considered to be "core" features, whose presence was considered necessary to the quality of the final product. These core features were centered around the desired experience for the system's users, and the various potential actions users would be able to take within the virtual world. Each of these features received significant attention during both the design and production phases of the project.

Table 2 - Initial core design goals

When the user is…	…he can:
Connecting for the first time	• Setup a new account • Download the mobile application • Take the questionnaire • Be playing within 2 minutes
Not interacting	• Move their creature • Equip/remove items • Leave their creature to act on its own • View their item catalog • View their friends list • Log out
Interacting with the environment	• Find/pick up items • Interact with environmental objects • Affect the zoom of the display • Trade with NPC creatures
Interacting with other users	• Trade items with another creature • Chat with another creature • Add a user as a friend • Begin/participate in an activity

6.2 Final results

We feel that we have made significant progress toward the completion of our initial goals. Although we have not completed everything we set out to do, our system demonstrates the capabilities of a cross-platform mobile application for a large public display. We were able to complete most features for iPhone and many features for Android, as well as optimize our existing server setup. We localized every aspect of the system for Japanese and English languages and have designed the system to be easily extensible for other locales and languages. We believe we have optimized the server to its maximum potential given the current available hardware; the primary hindrance to high-speed operation of the system is Internet connection speed. Table 3 details the original design goals which were met by the end of production, and shows that the majority of the original set of goals has been met. While certain items did not get met due to time constraints, the final version of the system closely matches the original design intentions. The largest features which could not be developed in time are the autonomous behaviors which were originally planned for creatures who were left unattended, NPC creatures who could populate the world when no users were logged in and would be able to trade items with users, and minigames and social activities for creatures. Each of these features got removed due to a lack of time and resources, but could be added to the system without requiring large changes to the existing architecture.

Table 3 - Design goals met

When the user is…	…he can:
Connecting for the first time	• Setup a new account on mobile device • Take the questionnaire • Be playing within 2 minutes
Not interacting	• Move their creature • Equip/remove items • View their item catalog • View their friends list • Log out
Interacting with the environment	• Find/pick up items • Interact with environmental objects • Affect the zoom of the display
Interacting with other users	• Trade items with another creature • Chat with another creature • Add a user as a friend

6.3 Challenges and retrospective

Throughout the project, our team was faced with two types of challenges: technical and personal. The most significant technical problems we faced were platform compatibility and latency. Our entire system relies on having Internet connections for every user and the server itself. Additionally, each terminal must connect to the server. Users will experience lag if their own Internet connection or the connection between a terminal and the server is slow.

We intended to have mobile devices act as controllers for the system; as such, we needed to develop for multiple platforms including iPhone and Android. This meant programming mobile applications in Objective C (iPhone) and Java (Android), and applications in Flash (Terminal), Java (Server) and JSP (Website). This also led to some issues with integration of independently developed programs.

Our team also had some problems with organization and coordination. Due to scheduling conflicts with external projects such as the IVRC (ivrc.net), it was difficult to arrange meetings at OU. As such, our productivity was not always as high as it could have been. We also believe that we began our development phase too late to finish everything we had planned, despite and partially because of the extensive time we spent on planning.

6.4 Future work

While the core design features of the project are complete, there was very little time available to test the user experience with the system. A full-scale test of the system would include users at both WPI and OU, testing the system from beginning to end. Additionally, there are many possible extensions which could be made to the system in the future, drawn from both the original design and ideas generated during project development.

6.4.1 Testing

The majority of user testing should focus on the front-end experience, and users' opinions and suggestions regarding the features and possible additions or changes, as that kind of feedback was not able to be gathered significantly during the course of the project. A comparison of feedback from American and Japanese users might also provide some insight into cultural differences regarding interactive public content and the interest of people in each country in participating in publicly displayed events. A thorough test would examine each of these aspects of the system, and hopefully the questions outlined in Table 4.

Table 4 - User experience questions

Approaching the display and creating an account	- How long does the average user take to complete the questionnaire - Do any users have trouble completing the questionnaire? - Do users have trouble returning to the questionnaire after making an account? - How long does it take to download the mobile application?
Logging in and having the creature appear on the display	- What is the average time between the user starting the application and the creature appearing on the display? - What is the average time between submitting login credentials and having the creature appear on the screen?
Collecting, equipping, trading, and dropping items	- What is the average time between selecting an action and having the action confirmation/rejection appear on the mobile device?

	- Do some actions take longer than others? - Do users prefer to find items as treasures, or trade with other users? - What do users think of the catalog system? Are they interested in completing their catalogs? - Do users like the items, and equipping them on their creatures?
User interface survey	- Do users find the interface easy to understand? - What would users change about the interface? - What do users like about the interface?
Chatting, especially between users with devices in different languages	- How long does it take to find a chat message in the phrase menus? - How long does a chat message take to reach the other user? - Do users find the messages easy to understand? - Do users find it more difficult to chat with users in a different language? - What chat categories/messages would users like to see added or changed?
Logging out	- How long does the display take to remove a creature after the user logs out? - Do users tend to log out manually, or just close the application and/or put the device in sleep mode?

Apart from user feedback, it is advisable to subject the system to various stress tests. The server is presently capped at a maximum of 50 users connected at one time, but there was no opportunity to test the system with such a high number of actual mobile devices. The terminal application is also a possible resource bottleneck, as it is by necessity single-threaded; it is likely that a large increase in the number of messages being sent to the terminal will tax the Flash application significantly. Tests of network and CPU usage should be performed on the server, to see if any places require changes or optimization and to determine what needs to be done to ensure the future scalability of the system as a whole.

In addition, data should be collected to measure the percentage of network traffic devoted to each type of message, to streamline messages which are exceptionally frequent (such as creature movement messages) or take longer than average (such as catalog lists). The message traffic and travel time should be measured both generally and individually between components, as well as locally (with all components on one campus) and globally (with the server and database on one campus, and mobile devices and displays on the other) to see the effect global location has on network performance. Specifically, performance should be compared on all steps in communication from Mobile Device → Server → Mobile Device (e.g., trade commands), Mobile Device → Server → Terminal (e.g., movement commands), and Server → Terminal (e.g., game loop updates), with as many different global configurations as possible for the server, database, mobile devices, and terminal displays.

6.4.2 Minigames

Users can participate in minigames with their creatures in order to win items and change their stats. Creatures' stats affect how well they perform in minigames. These minigames would be slow-paced in order to compensate for the high latency between WPI and OU, but would not require a significant time investment to play one game.

6.4.3 Pageant

Users can submit their creatures with their current genomes and equipped items to a pageant. Users vote on which creature looks the best in different categories. Creatures that win a pageant will receive some kind of award.

6.4.4 More shapes

Extend the current set of shapes to include more, perhaps including some non-regular polygons. We think that people might get bored of having the same shapes for a long time, so adding more shapes would extend the interest over time. Users might also be able to create their own shapes.

6.4.5 More items

Extend the current set of items. Include location-specific items to encourage trade between WPI and OU. Perhaps even let users create their own items with a given tool set. Items could also have stats or abilities.

6.4.6 Dynamic environment

The look of the environment changes over time. Obstacles, toys, treasure chests, etc. change along with the environment. Creatures can also affect the look of the environment.

6.4.7 More interactions between creatures

Add more interactions between creatures besides chatting and trading. These can be mobile-mobile or displayed on the terminal.

6.4.8 More supported phones

Add support for more Japanese phones and American smart phones. Support for any phone Web browser would increase the number of potential users of the system.

6.4.9 Scheduled Events

Different things happen at events that are scheduled in the system. For example, when it is Christmas then snow can be falling in the environment. Events can be global or region-based.

6.4.10 Creature animations

Improve creature animations so that they aren't rigid. Perhaps also add animations to items. A creature's stats should affect its animation.

6.4.11 Social networking Website

Have a Website for users to interact with other users they met off of the display. This would encourage even further international communication/collaboration. Users can show off their creatures, manage their friends and items, and communicate on a message board.

6.4.12 Behaviors

Creatures have different behaviors based on their stats. The tether system, which was originally planned, would also encompass creature behaviors.

7 Conclusion

On October 6th, 2010, a live demonstration and presentation of the system was held at the Library of Osaka University's Toyonaka Campus (Figure 54). During the demonstration, it was shown that using both iPhone and Android one could control a creature that was visible on one of the displays of Osaka University's O+PUS system (their network of public displays). Using the mobile devices, demonstrators were able to move, chat, trade items, find items in the environment, change their equipment, view their inventory and catalog, and become friends with other users. Despite latency concerns, the system performed as expected.

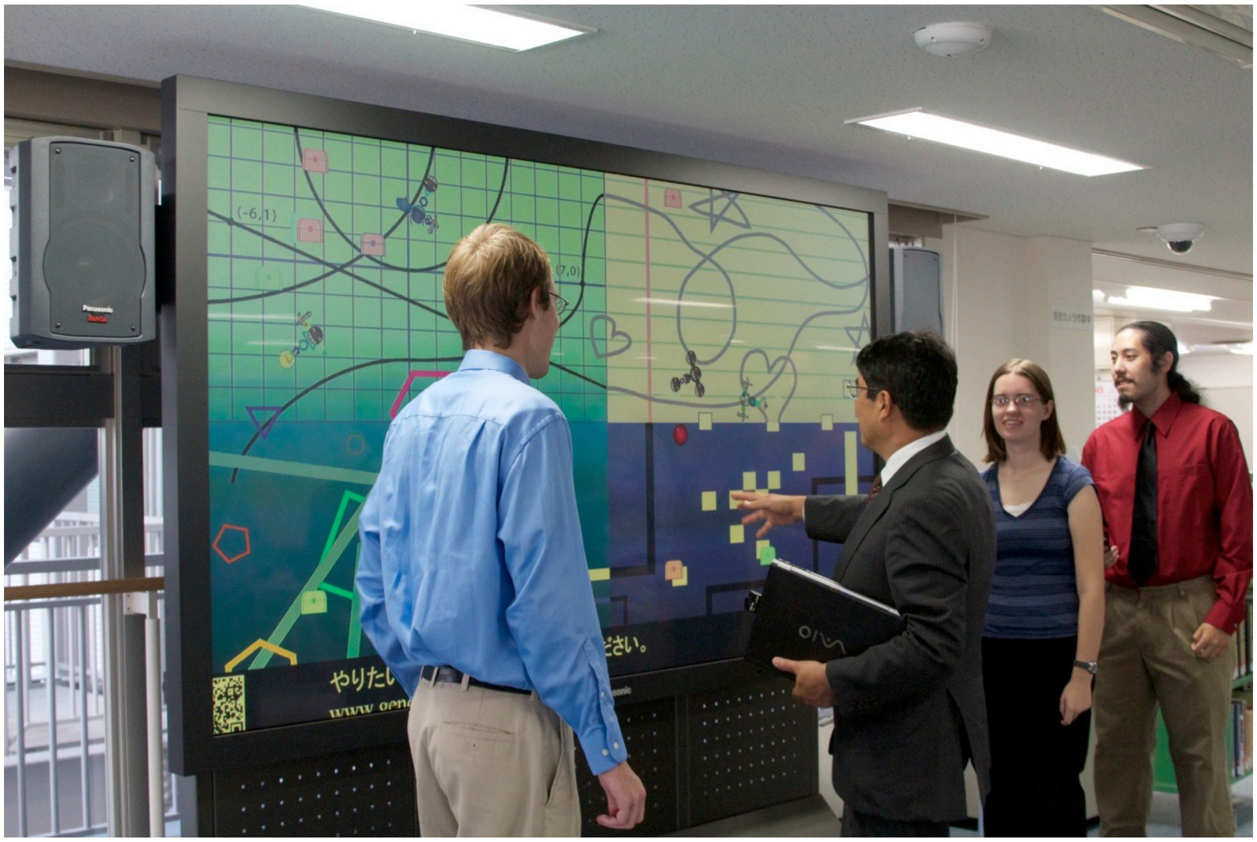


Figure 54 - Prof. Haruo Takemura discussing the project with some of the team members

The application is, barring some known bugs and unimplemented features, a success, but the real measure of the project's success is in the underlying architecture. Though ambitious, the concept of using mobile devices to control public displays around the globe is not only possible, it has now been done.

One of the hopes for the project is that it will encourage similarly ambitious projects to push the boundaries in terms of global information sharing and communication. Not all information needs to come through a personal computer and a Web browser. Not all interactions need to be limited to individuals using personal screens.

For the project, the team had two main goals in mind. The first was a technological goal, where users could interact with and affect content shown on a public display using their mobile devices. The second was a social goal, as the users of the system reside in two different countries, and speak two different languages. Therefore, the second goal was to encourage users from both countries to interact with each other without encountering problems due to the language barrier. At the end of the project, both the technological and social goals were met, with American and Japanese users testing the system together. While some of the features of the application were not implemented, and there is still testing and user evaluations which could be done to improve the system design, the major goals were met.

8 References

Apache Software Foundation. Batik SVG Toolkit. 2010. <http://xmlgraphics.apache.org/batik/>.

Apple, Inc. Event Handling Guide for iOS. 2010.

—. iPhone Human Interface Guidelines. 2010.

—. Low-Level File Management Programming Topics. 2010.

Bradley, Tom. TBXML. 2009. <http://www.tbxml.co.uk/TBXML/TBXML_Free.html>.

Brignull, Harry, and Yvonne Rogers. "Enticing People to Interact with Large Public Displays in Public Spaces". *In Proceedings of the IFIP International Conference on Human-Computer Interaction (INTERACT 2003.* Web.

Finke, Matthias, et al. "Lessons Learned: Game Design for Large Public Displays". *DIMEA '08: Proceedings of the 3rd international conference on Digital Interactive Media in Entertainment and Arts.* Athens, Greece. Web.

flOw. Los Angeles: Sony Computer Entertainment, 2006.

Huang, Elaine, Anna Koster, and Jan Borchers. "Overcoming Assumptions and Uncovering Practices: When does the Public really Look at Public Displays?" *Pervasive Computing.* Eds. Jadwiga Indulska, et al. 5013 Vol. Springer Berlin / Heidelberg, 2008. 228-243. Web.

Information Technology Laboratory. Secure Hash Standard. National Institute of Standards and Technology. October 2008.

Maunder, Andrew, Gary Marsden, and Richard Harper. "Creating and Sharing Multi-Media Packages using Large Situated Public Displays and Mobile Phones". *MobileHCI '07: Proceedings of the 9th international conference on Human computer interaction with mobile devices and services.* Singapore. Web.

Mccarthy, Joseph F. “Using Public Displays to Create Conversation Opportunities.” 2002: n. pag. Print. 28 Oct 2010.

Oracle Corporation. Java Database Connector. 2010. <http://www.oracle.com>.

Rogers, Yvonne, and Harry Brignull. "Subtle Ice-Breaking: Encouraging Socializing and Interaction Around a Large Public Display". *Proceedings of Public, community and situated displays: Design, use and interaction around shared information displays.* November 16 2002, New Orleans, Louisiana. 2002. Print.

Storz, Oliver, et al. "Public Ubiquitous Computing Systems: Lessons from the e-Campus Display Deployments." *IEEE Pervasive Computing* 5 (2006): 40-7. Web.

Tuulos, Ville, Jürgen Scheible, and Heli Nyholm. *Combining Web, Mobile Phones and Public Displays in Large-Scale: Manhattan Story Mashup*. Eds. Anthony LaMarca, Marc Langheinrich, and Khai Truong. 4480 Vol. Springer Berlin / Heidelberg, 2007. Web.

Vajk, Tamas, Coulton, Paul, Bamford, Will, and Reuben Edwards. "Using a Mobile Phone as a "Wii-like" Controller for Playing Games on a Large Public Display." *Int.J.Comput.Games Technol.* 2008 (2008): 1-6. Web.

Voss, Dusting. AsyncSocket Documentation. 2003.

—. Cocoa AsyncSocket. 2003. <http://code.google.com/p/cocoaasyncsocket/>.

Appendix A. XML Messages

Key:
[options] (choose one)
variable (you supply this)
string (use this text)

Required message structure:

```
<message>
        <head>
                <sender>id#</sender>
                <recipient>id#</recipient>
                <confirmReceipt>[true, false]</confirmReceipt>
                <awaitReply>[true, false]</awaitReply>
        </head>
        <body>
                <messageType>message type</messageType>
        </body>
</message>
```

Messages sent by mobile devices

Accept/Deny Friend (Device → Server)

```
<body>
        <messageType>acceptDenyFriend</messageType>
        <username>username</username>
        <otherCreatureName>username</otherCreatureName>
        <friendAcceptDeny>[true,false]</friendAcceptDeny>
</body>
```

Chat cancel (Device → Server) OR (Server → Device)

```
<body>
        <messageType>chatCancel</messageType>
        <sourceCreatureID>id#</sourceCreatureID>
</body>
```

Chat close (Device → Server) OR (Server → Device)

```
<body>
        <messageType>chatClose</messageType>
        <sourceCreatureID>id#</sourceCreatureID>
        <chatID>id#</chatID>
</body>
```

Chat message (Device → Server) OR (Server → Device)

```
<body>
	<messageType>chatMessage</messageType>
	<chatID>id#</chatID>
	<sourceCreatureID>id#</sourceCreatureID>
	<messageID>id#</messageID>
</body>
```

Chat request (Device → Server)

```
<body>
	<messageType>chatRequest</messageType>
	<sourceCreatureID>id#</sourceCreatureID>
	<destCreatureID>id#</destCreatureID>
</body>
```

Chat request result (Device → Server)

```
<body>
	<messageType>chatRequestResult</messageType>
	<chatID>id#</chatID>
	<chatAccepted>[true, false]</chatAccepted>
</body>
```

Drop item (Device → Server)

```
<body>
	<messageType>dropItem</messageType>
	<sourceCreatureID>id#</sourceCreatureID>
	<itemID>id#</itemID>
</body>
```

Equip item (Device → Server)

```
<body>
	<messageType>equipItem</messageType>
	<sourceCreatureID>id#</sourceCreatureID>
	<itemID>id#</itemID>
	<equipLocation>[0, 1, 2, 3, 4]</equipLocation>
</body>
```

Get treasure (Device → Server)

```
<body>
	<messageType>getTreasure</messageType>
	<sourceCreatureID>id#</sourceCreatureID>
	<treasureID>id#</treasureID>
</body>
```

Login (Device → Server)

```
<body>
	<messageType>login</messageType>
```

```
    <username>[username]</username>
    <password>[password]</password>
</body>
```

Logout (Device → Server)

```
<body>
    <messageType>logout</messageType>
    <deviceID>id#</deviceID>
</body>
```

Move creature (Device → Server)

```
<body>
    <messageType>moveCreature</messageType>
    <sourceCreatureID>id#</sourceCreatureID>
    <moveVector>x (float) y (float)</moveVector>
</body>
```

Pending trade cancel (Device → Server) OR (Server → Device)

```
<body>
    <messageType>tradeCancel</messageType>
    <sourceCreatureID>id#</sourceCreatureID>
</body>
```

Request catalog (Device → Server)

```
<body>
    <messageType>requestCatalog</messageType>
    <username>username</username>
</body>
```

Request friend (Device → Server)

```
<body>
    <messageType>requestFriend</messageType>
    <username>username</username>
    <otherCreatureName>username</otherCreatureName>
</body>
```

Request friends list (Device → Server)

```
<body>
    <messageType>requestFriendsList</messageType>
    <username>username</username>
</body>
```

Request interactables (Device → Server)

```
<body>
    <messageType>requestInteractables</messageType>
```

```
        <sourceCreatureID>id#</sourceCreatureID>
</body>
```

Request inventory (Device → Server)

```
<body>
        <messageType>requestInventory</messageType>
        <sourceCreatureID>id#</sourceCreatureID>
</body>
```

Specify location (Terminal OR Device → Server)

```
<body>
        <messageType>specifyLocation</messageType>
        <location>[ou, wpi]</location>
</body>
```

Trade add item (Device → Server) OR (Server → Device)

```
<body>
        <messageType>tradeAddItem</messageType>
        <sourceCreatureID>id#</sourceCreatureID>
        <tradeID>id#</tradeID>
        <itemID>id#</itemID>
        <numItems># of items (int)</numItems>
</body>
```

Trade cancel (Device → Server) OR (Server → Device)

```
<body>
        <messageType>tradeCancel</messageType>
        <sourceCreatureID>id#</sourceCreatureID>
        <tradeID>id#</tradeID>
</body>
```

Trade change item (Device → Server) OR (Server → Device)

```
<body>
        <messageType>tradeChangeItem</messageType>
        <sourceCreatureID>id#</sourceCreatureID>
        <tradeID>id#</tradeID>
        <itemID>id#</itemID>
        <numItems># of items to add/subtract (int)</numItems>
</body>
```

Trade confirm (Device → Server) OR (Server → Device)

```
<body>
        <messageType>tradeConfirm</messageType>
        <sourceCreatureID>id#</sourceCreatureID>
        <tradeID>id#</tradeID>
</body>
```

Trade remove item (Device → Server) OR (Server → Device)

```
<body>
	<messageType>tradeRemoveItem</messageType>
	<sourceCreatureID>id#</sourceCreatureID>
	<tradeID>id#</tradeID>
	<itemID>id#</itemID>
</body>
```

Trade request (Device → Server)

```
<body>
	<messageType>tradeRequest</messageType>
	<sourceCreatureID>id#</sourceCreatureID>
	<destCreatureID>id#</destCreatureID>
</body>
```

Trade request result (Device → Server)

```
<body>
	<messageType>tradeRequestResult</messageType>
	<tradeID>id#</tradeID>
	<tradeAccepted>[true, false]</tradeAccepted>
</body>
```

Trade unconfirm (Device → Server) OR (Server → Device)

```
<body>
	<messageType>tradeUnconfirm</messageType>
	<sourceCreatureID>id#</sourceCreatureID>
	<tradeID>id#</tradeID>
</body>
```

Unequip item (Device → Server)

```
<body>
	<messageType>unequipItem</messageType>
	<sourceCreatureID>id#</sourceCreatureID>
	<equipLocation>[0, 1, 2, 3, 4]</equipLocation>
</body>
```

Messages sent by the server

Add ball (Server → Terminal)

```
<body>
	<messageType>addBall</messageType>
	<ballID>id#</ballID>
	<position>x (int) y (int)</position>
</body>
```

Add creature (Server → Terminal)

```
<body>
        <messageType>addCreature</messageType>
        <sourceCreatureID>id#</sourceCreatureID>
        <genome>genome (string)</genome>
        <equipmentList>
                <equipment>id# (slot 1)</equipment>
                <equipment>id# (slot 2)</equipment>
                <equipment>id# (slot 3)</equipment>
                <equipment>id# (slot 4)</equipment>
                <equipment>id# (slot 5)</equipment>
        </equipmentList>
        <strength>strength (int)</strength>
        <speed>speed (int)</speed>
        <perception>perception (int)</perception>
        <sociability>sociability (int)</sociability>
        <curiosity>curiosity (int)</curiosity>
        <obedience>obedience (int)</obedience>
</body>
```

Add treasure (Server → Terminal)

```
<body>
        <messageType>addTreasure</messageType>
        <position>x (int) y (int)</position>
        <treasureColor>0x###### (Hex color code)</treasureColor>
        <treasureID>id#</treasureID>
        <visibility>[true, false]</visibility>
</body>
```

Catalog list (Server → Device)

```
<body>
        <messageType>catalogList</messageType>
        <catalog>
                <catalogEntry>
                        <itemID>id#</itemID>
                        <itemStatus>(int)</itemStatus>
                </catalogEntry>
                <catalogEntry>
                        <itemID>id#</itemID>
                        <itemStatus>(int)</itemStatus>
                </catalogEntry>
        </catalog>
</body>
```

Chat cancel (Device → Server) OR (Server → Device)

```
<body>
        <messageType>chatCancel</messageType>
        <sourceCreatureID>id#</sourceCreatureID>
</body>
```

Chat close (Device → Server) OR (Server → Device)

```
<body>
        <messageType>chatClose</messageType>
        <sourceCreatureID>id#</sourceCreatureID>
        <chatID>id#</chatID>
</body>
```

Chat message (Device → Server) OR (Server → Device)

```
<body>
        <messageType>chatMessage</messageType>
        <chatID>id#</chatID>
        <sourceCreatureID>id#</sourceCreatureID>
        <messageID>id#</messageID>
</body>
```

Chat request (Server → Device)

```
<body>
        <messageType>chatRequest</messageType>
        <otherCreatureName>name</otherCreatureName>
        <chatID>id#</chatID>
</body>
```

Chat request result (Server → Device)

```
<body>
        <messageType>chatRequestResult</messageType>
        <chatAccepted>[true, false]</chatAccepted>
(If chatAccepted was true:)
        <chatID>id#</chatID>
(If chatAccepted was false:)
        <failureReason>[inChat, denied]</failureReason>
</body>
```

Connection result (Server → Device)

```
<body>
        <messageType>connectionResult</messageType>
        <connectionSuccess>[true, false]</connectionSuccess>
(If connectionSuccess was true:)
        <deviceID>id#</deviceID>
(If connectionSuccess was false:)
        <failureReason>[socketDenied, userCap]</failureReason>
</body>
```

Disconnect (Server → Device) OR (Server → Terminal)

```
<body>
	<messageType>disconnect</messageType>
</body>
```

Equipment change (Server → Terminal)

```
<body>
	<messageType>equipmentChange</messageType>
	<sourceCreatureID>id#</sourceCreatureID>
	<equipLocation>[1,2,3,4,5]</equipLocation>
	<itemID>id#</itemID>
</body>
```

Equipment change(Server → Device)

```
<body>
	<messageType>equipmentChange</messageType>
	<equipmentSuccess>[true, false]</equipmentSuccess>
(If equipmentSuccess was true:)
	<itemList>
		<item>
			<itemID>id#</itemID>
			<numItems>(int)</numItems>
		</item>
		<item>
			<itemID>id#</itemID>
			<numItems>(int)</numItems>
		</item>
	</itemList>
	<equipmentList>
		<equipment>id# (slot 1)</equipment>
		<equipment>id# (slot 2)</equipment>
		<equipment>id# (slot 3)</equipment>
		<equipment>id# (slot 4)</equipment>
		<equipment>id# (slot 5)</equipment>
	</equipmentList>
(If equipmentSuccess was false:)
	<failureReason>failure reason</failureReason>
</body>
```

Error (Server → Device)

```
<body>
	<messageType>error</messageType>
	<failureReason>reason id#</failureReason>
</body>
```

Friend Accept/Deny (Server → Device)

```
<body>
        <messageType>friendAcceptDeny</messageType>
        <friendAcceptDeny>[true,false]</friendAcceptDeny>
        <friend>
                <destCreatureID>id#</destCreatureID>
                <otherCreatureName>name (string)</otherCreatureName>
                <genome>genome string</genome>
        </friend>
</body>
```

Friends list (Server → Device)

```
<body>
        <messageType>friendsList</messageType>
        <friendsList>
                <friend>
                        <destCreatureID>id#</destCreatureID>
                        <otherCreatureName>name (string)</otherCreatureName>
                        <genome>genome string</genome>
                </friend>
                <friend>
                        <destCreatureID>id#</destCreatureID>
                        <otherCreatureName>name (string)</otherCreatureName>
                        <genome>genome string</genome>
                </friend>
        </friendsList>
</body>
```

Friend request (Server → Device)

```
<body>
        <messageType>friendRequest</messageType>
        <friend>
                <destCreatureID>id#</destCreatureID>
                <otherCreatureName>name (string)</otherCreatureName>
                <genome>genome string</genome>
        </friend>
</body>
```

Genome change (Server → Terminal)

```
<body>
        <messageType>genomeChange</messageType>
        <sourceCreatureID>id#</sourceCreatureID>
        <genome>Complete genome string for a creature (String)</genome>
</body>
```

Interactables list (Server → Device)

```
<body>
        <messageType>interactablesList</messageType>
        <creatureList>
                <creature>
                        <destCreatureID>id#</destCreatureID>
                        <otherCreatureName>name (string)</otherCreatureName>
                        <genome>genome string</genome>
                </creature>
                <creature>
                        <destCreatureID>id#</destCreatureID>
                        <otherCreatureName>name (string)</otherCreatureName>
                        <genome>genome string</genome>
                </creature>
        </creatureList>
        <treasureList>
                <treasure>
                        <treasureID>id#</treasureID>
                        <treasureColor>0x######</treasureColor>
                </treasure>
                <treasure>
                        <treasureID>id#</treasureID>
                        <treasureColor>0x######</treasureColor>
                </treasure>
        </treasureList>
</body>
```

Inventory list (Server → Device)

```
<body>
        <messageType>inventoryList</messageType>
        <itemList>
                <item>
                        <itemID>id#</itemID>
                        <numItems>(int)</numItems>
                </item>
                <item>
                        <itemID>id#</itemID>
                        <numItems>(int)</numItems>
                </item>
        </itemList>
</body>
```

Login response (Server → Device)

```
<body>
        <messageType>loginResult</messageType>
        <loginSuccess>[true, false]</loginSuccess>
```

(If loginSuccess was true:)

```
        <sourceCreatureID>id#</sourceCreatureID>
        <genome>genome string</genome>
        <equipmentList>
                <equipment>id# (slot 1)</equipment>
                <equipment>id# (slot 2)</equipment>
                <equipment>id# (slot 3)</equipment>
                <equipment>id# (slot 4)</equipment>
                <equipment>id# (slot 5)</equipment>
        </equipmentList>
(If loginSuccess was false:)
        <failureReason>[badCredentials, databaseError]</failureReason>
</body>
```

Not logged in (Server → Mobile Device)

```
<body>
        <messageType>notLoggedIn</messageType>
</body>
```

Pending trade cancel (Device → Server) OR (Server → Device)

```
<body>
        <messageType>tradeCancel</messageType>
        <sourceCreatureID>id#</sourceCreatureID>
</body>
```

Pick up treasure (Server → Device)

```
<body>
        <messageType>pickUpTreasure</messageType>
        <treasureSuccess>[true, false]</treasureSuccess>
        <itemList>
                <item>
                        <itemID>id#</itemID>
                        <numItems>(int)</numItems>
                </item>
        </itemList>
</body>
```

Remove ball (Server → Terminal)

```
<body>
        <messageType>removeBall</messageType>
        <ballID>id#</ballID>
</body>
```

Remove creature (Server → Terminal)

```
<body>
        <messageType>removeCreature</messageType>
        <sourceCreature>id#</sourceCreature>
</body>
```

Remove treasure (Server → Terminal)

```
<body>
        <messageType>removeTreasure</messageType>
        <treasureID>id#</treasureID>
</body>
```

Toggle treasure visibilities (Server → Terminal)

```
<body>
        <messageType>toggleTreasureVisibilities</messageType>
        <treasureList>
                <treasureID>id#</treasureID>
                <treasureID>id#</treasureID>
                <treasureID>id#</treasureID>
                <treasureID>id#</treasureID>
        </treasureList>
</body>
```

Trade add item (Device → Server) OR (Server → Device)

```
<body>
        <messageType>tradeAddItem</messageType>
        <sourceCreatureID>id#</sourceCreatureID>
        <tradeID>id#</tradeID>
        <itemID>id#</itemID>
        <numItems># of items (int)</numItems>
</body>
```

Trade cancel (Device → Server) OR (Server → Device)

```
<body>
        <messageType>tradeCancel</messageType>
        <sourceCreatureID>id#</sourceCreatureID>
        <tradeID>id#</tradeID>
</body>
```

Trade change item (Device → Server) OR (Server → Device)

```
<body>
        <messageType>tradeChangeItem</messageType>
        <sourceCreatureID>id#</sourceCreatureID>
        <tradeID>id#</tradeID>
        <itemID>id#</itemID>
        <numItems># of items to add/subtract (int)</numItems>
</body>
```

Trade complete (Server → Device)

<body>
        <messageType>*tradeComplete*</messageType>
        <tradeID>id#</tradeID>
</body>

Trade confirm (Device → Server) OR (Server → Device)

<body>
        <messageType>*tradeConfirm*</messageType>
        <sourceCreatureID>id#</sourceCreatureID>
        <tradeID>id#</tradeID>
</body>

Trade remove item (Device → Server) OR (Server → Device)

<body>
        <messageType>*tradeRemoveItem*</messageType>
        <sourceCreatureID>id#</sourceCreatureID>
        <tradeID>id#</tradeID>
        <itemID>id#</itemID>
</body>

Trade request (Server → Device)

<body>
        <messageType>*tradeRequest*</messageType>
        <otherCreatureName>name</otherCreatureName>
        <tradeID>id#</tradeID>
</body>

Trade request result (Server → Device)

<body>
        <messageType>*tradeRequestResult*</messageType>
        <tradeAccepted>[true, false]</tradeAccepted>
(If tradeAccepted was true:)
        <tradeID>id#</tradeID>
(If tradeAccepted was false:)
        <failureReason>[inTrade, denied]</failureReason>
</body>

Trade unconfirm (Device → Server) OR (Server → Device)

<body>
        <messageType>*tradeUnconfirm*</messageType>
        <sourceCreatureID>id#</sourceCreatureID>
        <tradeID>id#</tradeID>
</body>

Update ball (Server → Terminal)

```
<body>
	<messageType>updateBall</messageType>
	<ballID>id#</ballID>
	<position>x (int) y (int)</position>
	<velocity>x (int) y (int)</velocity>
	<acceleration>x (float) y (float)</acceleration>
	<accelTime>time (float)</accelTime>
</body>
```

Update creature movement (Server → Terminal)

```
<body>
	<messageType>moveCreature</messageType>
	<sourceCreatureID>id#</sourceCreatureID>
	<position>x (float) y (float)</position>
	<velocity>x (float) y (float)</velocity>
	<acceleration>x (float) y (float)</acceleration>
	<accelTime>time (float)</accelTime>
	<targetPosition>x(float) y(float)</targetPosition>
	<maxVelocity>x(float) y(float)</maxVelocity>
</body>
```

Messages sent by terminals

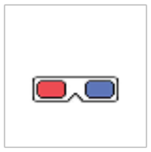
Specify location (Terminal OR Device → Server)

```
<body>
	<messageType>specifyLocation</messageType>
	<location>[ou, wpi]</location>
</body>
```

Appendix B. Items

101_tophat.png

102_clownwig.png


103_3dglasses.png

104_nightvisiongo...

105_catears.png

106_deerantlers.p...

107_bunnyears.png

108_flowercap.png

109_sockhat.png

110_kasa.png

111_strawhat.png

112_fedora.png

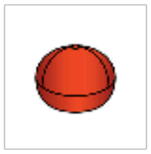
113_unicornhorn....

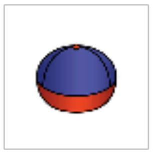
114_kabuto.png

115_suspiciousst...

116_headphones....

117_glasses.png


118_redcap.png


119_bluecap.png

120_blackcap.png

121_sleepcap.png

122_royalcrown.p...

123_wizardscap.p...

124_fishhead.png

125_headintheclo...

126_sunglasses.p...

127_redbow.png

128_pinkbow.png

129_whitebow.png

130_blackbow.png

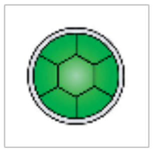
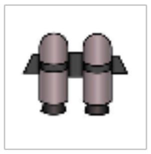
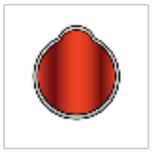
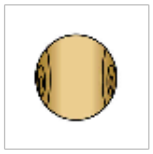
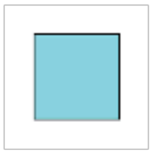
131_scubamask....

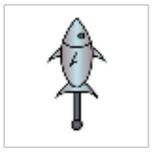
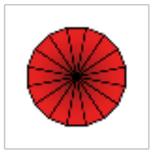
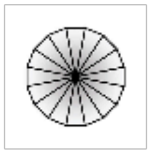
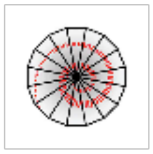
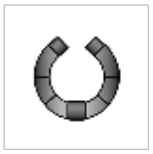
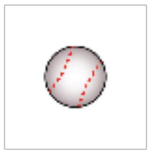
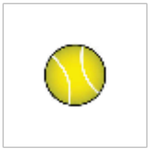
132_koban.png

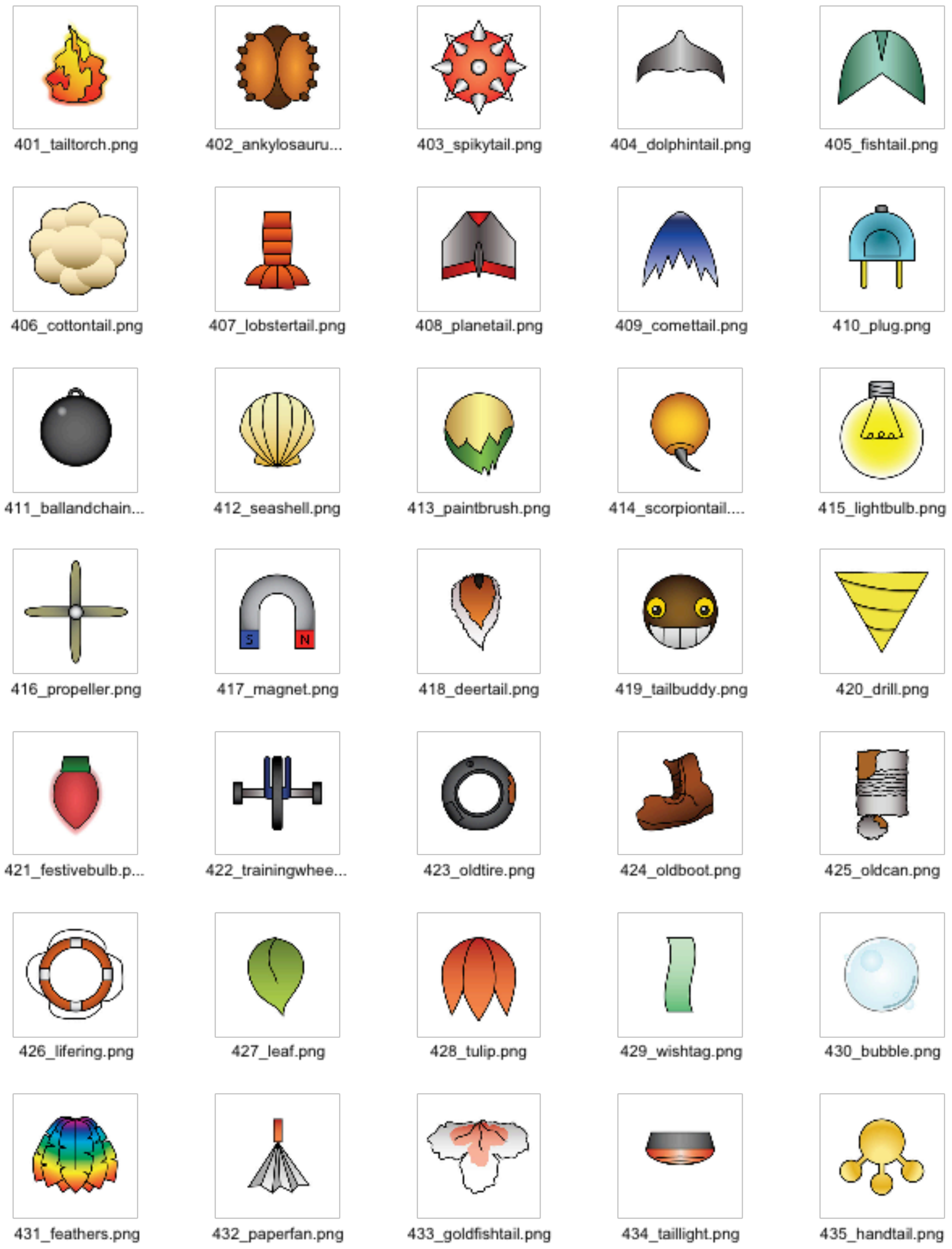
133_melonhat.png

134_dinosaurskull...

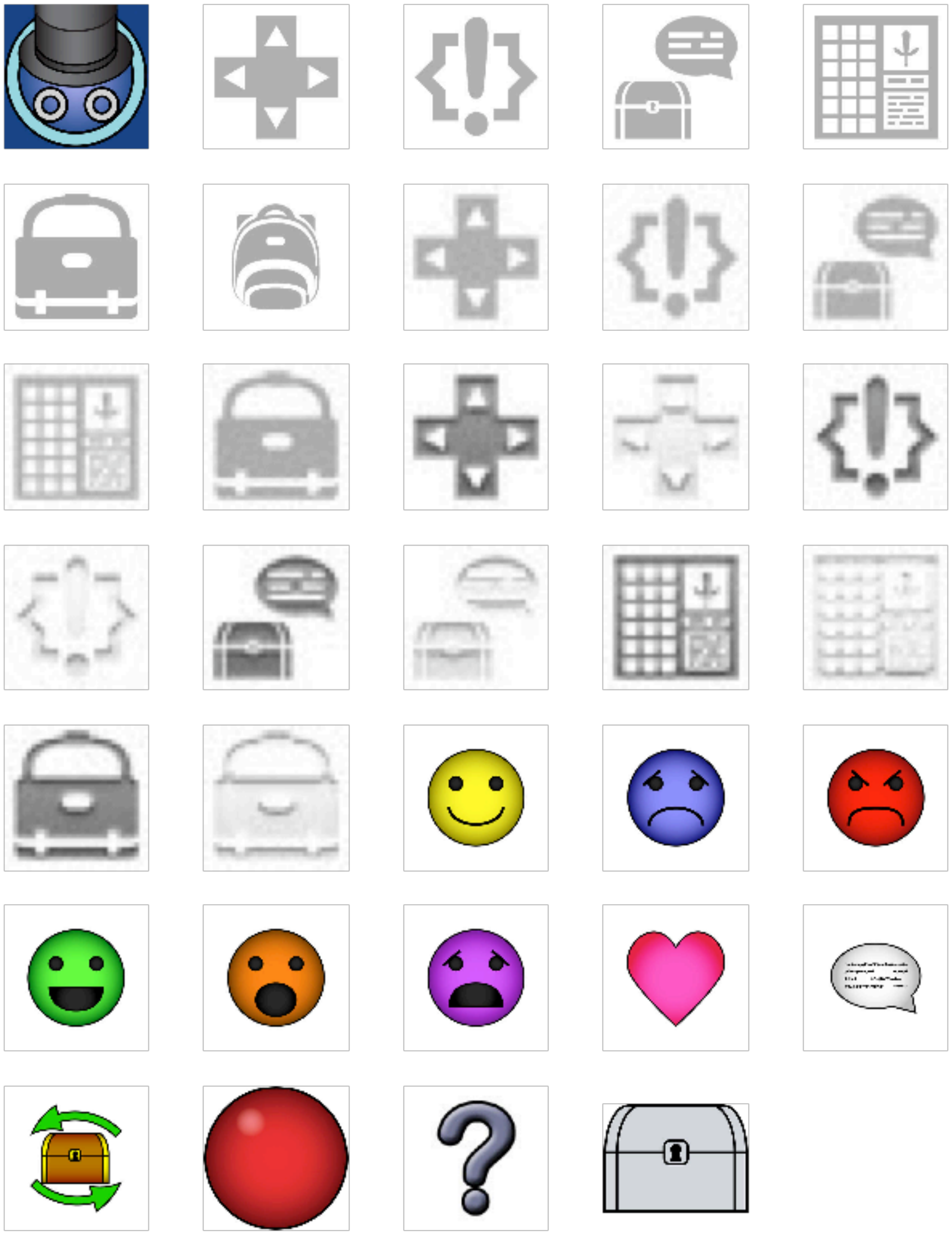
135_beanie.png

201_hidingbox.png	202_superherosc...	203_angelwings.p...	204_cyberclock.png	 205_turtleshell.png
206_batwings.png	207_butterflywing...	 208_jetpack.png	209_backpack.png	210_herostools.png
211_scubatank.png	212_tatteredcape....	213_katanasheat...	 214_saddle.png	215_plantbulb.png
216_largeflower.p...	217_backspikes.p...	218_jewel.png	219_Ipd.png	 220_snailshell.png
221_thickbook.png	222_bedtimeblan...	223_royalrobe.png	224_onigiri.png	225_sushiroll.png
 226_bluesquare.p...	227_cheeseburge...	228_icecube.png	229_barrel.png	230_clouds.png
231_earth.png	232_moon.png	233_soccerball.png	234_watermelon....	235_fancysuit.png

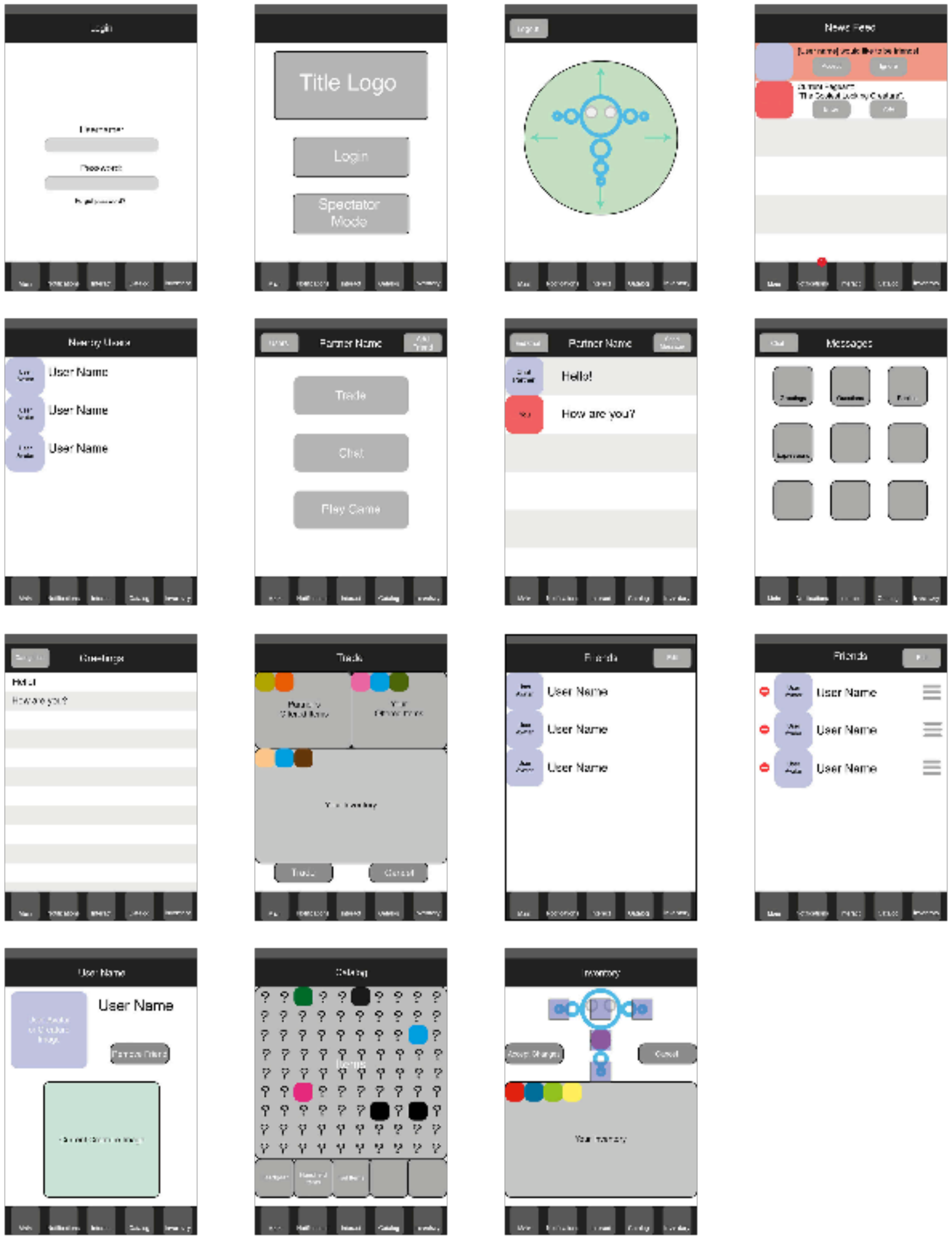
 301_sakanakatan...	302_katana.png	303_herossword....	304_herosshield....	305_fishingrod.png
306_bugnet.png	307_microphone....	308_guitar.png	309_electricguitar...	310_drumsticks.p...
311_mobiledevice...	 312_redparasol.png	 313_whiteparasol...	314_flowerparaso...	 315_swirlparasol....
316_whitefan.png	317_redfan.png	318_bluefan.png	319_flowerfan.png	320_cane.png
321_magicwand....	322_gloves.png	323_smallmagnet...	324_boxinggloves...	325_pompom.png
 326_robotclaw.png	327_scepter.png	 328_baseball.png	 329_tennisball.png	330_mittens.png
331_football.png	332_hockeypuck....	333_golfball.png	334_musicplayer....	335_gamedevice....


401_tailtorch.png
402_ankylosauru...
403_spikytail.png
404_dolphintail.png
405_fishtail.png
406_cottontail.png
407_lobstertail.png
408_planetail.png
409_comettail.png
410_plug.png
411_ballandchain...
412_seashell.png
413_paintbrush.png
414_scorpiontail....
415_lightbulb.png
416_propeller.png
417_magnet.png
418_deertail.png
419_tailbuddy.png
420_drill.png
421_festivebulb.p...
422_trainingwhee...
423_oldtire.png
424_oldboot.png
425_oldcan.png
426_lifering.png
427_leaf.png
428_tulip.png
429_wishtag.png
430_bubble.png
431_feathers.png
432_paperfan.png
433_goldfishtail.png
434_taillight.png
435_handtail.png

Appendix C. Terminal items and mobile device icons



Appendix D. User interface mockups



Appendix E. Item details

Head items

Head	頭
Top Hat	トップハット
A classy piece of headgear	立派な被り物
Clown Wig	ピエロの鬘
A silly, colorful wig	カラフルな鬘
3D Glasses	三次元めがね
Glasses for seeing into the future	未来が見えるめがね
Night Vision Goggles	暗視装置
High-tech equipment for spies.	スパイのハイテクの装備
Cat Ears	ネコ耳
A cute costume accessory.	かわいいコスプレアクセサリー
Deer Antlers	鹿の角
A symbol of strength and power.	強さと力の象徴
Bunny Ears	ウサギ耳
Rabbit ears for the speedy type.	速く走るようの長い耳
Flower Cap	花帽子
An odd flower that begs to be picked.	花がついている帽子
Sock Hat	リンクの帽子
A long, familiar cap.	長くて、お馴染みの帽子
Kasa	傘
Protective cover from the hot sun.	太陽から守る物陰
Straw Hat	ストローハット
The hat of choice for beach outings.	海岸旅行で頭を守る帽子
Fedora	フェドラ
A classic and reemerging hat.	クラシック、クールな帽子
Unicorn Horn	一角獣の角
A magical, rainbow-filled accessory.	キラキラ、魔法っぽいアクセサリ
Kabuto	かぶと
The traditional cap of a samurai.	侍の伝統的の被り物
Suspicious ‘Stache	怪しいひげ
A poor excuse for a disguise.	下手な仮面
Headphones	イヤホーン
High-quality personal speakers.	高級な耳にさし込むスピーカー
Glasses	めがね
Basic glasses for improving sight.	視力を治すめがね
Red Cap	赤い帽子
A red baseball cap.	赤い子供の帽子

Blue Cap	青い帽子
A blue and red baseball cap.	青いと赤い子供の帽子
Black Cap	黒い帽子
A black and yellow baseball cap.	黒いと黄色い子供の帽子
Sleep Cap	寝巻き帽子
A hat worn at bedtime.	寝る時間の帽子
Royal Crown	王立のクラウン
A symbol of royalty.	王族の象徴
Wizard’s Cap	魔法使いの帽子
A cap containing magic powers.	魔力を持っている帽子
Fish Head	魚の頭
A hungry fish from the ocean.	腹減った海から出た魚
Head in the Clouds	雲帽子
Light and fluffy headgear.	ふわふわの被り物
Sunglasses	サングラス
Polarized lenses for UV protection.	明るい太陽から目を守るサングラス
Red Bow	赤いリボン
A pretty red bow.	きれいな赤いリボン
Pink Bow	ピンクのリボン
A pretty pink bow.	きれいなピンクのリボン
White Bow	白いりっぼん
A pretty white bow.	きれいな白いリボン
Black Bow	黒いりっぼん
A pretty black bow.	きれいな黒いリボン
Scuba Mask	スクーバマスク
A mask for diving.	潜水用のマスク
Koban	小判
A shiny and lucky coin.	輝くラッキーな金貨
Melon Hat	メロン帽子
A delightful fruit hat.	果物がついている帽子
Dinosaur Skull	恐竜の頭骨
The skull of an extinct dinosaur.	絶滅した恐竜の頭骨
Beanie	スキーハット
A popular winter hat.	人気がある冬の帽子

Body items

Body	体
Hiding Box	隠れ箱
A highly effective disguise.	効果的な迷彩
Superhero’s Cape	英雄のマント
An super aerodynamic accessory.	空で飛べる用のマント

Angel Wings	天使の翼
Majestic white wings.	壮大で白い翼
Cyber Clock	サイバークロック
A customizable public clock.	ビルに付いている時計
Turtle Shell	亀の甲羅
A protective but removable shell.	身を守れて脱げる甲羅
Bat Wings	こうもりの翼
Dark and mysterious wings.	奇妙で黒い翼
Butterfly Wings	蝶の翅
Majestic and colorful wings.	色とりどりに光る大きな翅
Jetpack	ジェットパック
Rocket powered pack for air travel.	ロケットで人間を空に飛ばす物
Backpack	ナップサック
A standard pack for holding supplies.	用品を持つパック
Hero's Tools	英雄の道具
A sheath and quiver for heroes.	英雄の鞘と箙
Scuba Tank	空気タンク
An oxygen tank for diving.	酸素が入っている潜水用のタンク
Tattered Cape	ぼろぼろなマント
An old and dirty cape.	古くてくたびれたマント
Katana Sheath	日本刀の鞘
A sheath for a katana.	居合い用の鞘
Saddle	鞍
A simple saddle for one passenger.	一人の扱いやすい鞍
Plant Bulb	球根
The seed of a huge flower.	巨大な花の球根
Large Flower	大きな花
A colorful flower from a big seed.	見るものを引き寄せる美しい花
Back Spikes	背中のトゲ
Intimidating defensive spikes.	ハリネズミのような鋭いトゲ
Jewel	宝石
A very valuable giant gem.	巨大で貴重な宝石
LPD	LPD
A large public display.	大きなモニター
Snail Shell	かたつむりの貝殻
The house of a small snail.	小さなかたつもりの家
Thick Book	太い本
A large novel with many pages.	ページがたくさんある小説
Bedtime Blanket	安眠の毛布
A comfortable blanket for sleeping.	よく眠れる毛布
Royal Robe	王立のローブ

A velvet robe worn by kings.	王様のビロードのローブ
Onigiri	おにぎり
A tasty rice snack.	おいしいおにぎり
Sushi Roll	巻きずし
A roll of rice and fish.	おいしい巻きずし
Blue Square	青い四角
A primordial creature.	デバッグで使われたキャラクター
Cheeseburger Bun	チーズバーガーのバンズ
A sesame bun for a burger.	ハンバーガー用のゴマが付いたパン
Ice Cube	氷
A frozen block of ice.	氷の塊
Barrel	タル
A wooden barrel for storing things.	貯蔵用のタル
Clouds	雲
A shroud of fluffy clouds.	ふわふわの雲
Earth	地球のレプリカ
A replica of the planet Earth.	青い地球の模型
Moon	月のレプリカ
A replica of the Earth's moon.	輝く月の模型
Soccer Ball	サッカーボール
A wearable soccer ball.	歴史に残る試合で使われたサッカーボール
Watermelon	スイカ
A large, hollow watermelon.	中身の入っていないスイカ
Fancy Suit	立派なスーツ
An expensive high-class suit.	高級なビジネススーツ

Arm items

Arm	腕
Sakana Katana	魚刀
A strange sword that looks like a fish.	魚に似ている刀
Katana	日本刀
A traditional Japanese sword.	伝統的な日本刀
Hero's Sword	英雄の刀
A hero's weapon of choice.	英雄ガ好んで使った刀
Hero's Shield	英雄の盾
A hero's armor of choice.	英雄が好んで使った盾
Fishing Rod	釣竿
A rod for catching fish.	魚を釣るための釣竿
Bug Net	虫取りのネット
A net for catching insects.	虫を取るための網
Microphone	マイク

A wireless microphone for singers.	コンサートで使われるマイク
Guitar	ギター
A classic wooden guitar.	木製のクラシックギター
Electric Guitar	エレキギター
An electric guitar for rockers.	ライブで使われるエレキギター
Drum Sticks	ドラムスティック
Wooden sticks for drummers.	ドラマーが使う木製のばち
Mobile Device	携帯電話
A smart phone.	スマートフォン
Red Parasol	赤い傘
A fancy red parasol.	目立つ赤い傘
White Parasol	白い傘
A fancy white parasol.	目立つ白い傘
Flower Parasol	花柄の傘
A fancy parasol with a floral design.	目立つ花柄の傘
Swirl Parasol	渦柄の傘
A hypnotic parasol of red and white.	目が回る模様の傘
White Fan	白い団扇
A decorative white fan.	装飾が入った白い団扇
Red Fan	赤い団扇
A decorative red fan.	装飾が入った赤い団扇
Blue Fan	青い団扇
A decorative blue fan.	装飾が入った青い団扇
Flower Fan	花柄の団扇
A decorative floral fan.	花柄の装飾が入った団扇
Cane	杖
A multipurpose cane.	多目的に使える杖
Magic Wand	魔法の杖
A magician’s source of power.	魔法使いの魔力の源
Gloves	手袋
White working gloves.	作業用の白い手袋
Small Magnet	小さな磁石
Small magnets for attracting metal.	金属を寄せ付ける小さな磁石
Boxing Gloves	ボクシンググローブ
Cushioned gloves for boxers.	世界チャンピオンが使っていたグローブ
Pom-Pom	ポンポン
A colorful puff for cheering.	チアガールの応援道具
Robot Claw	ロボットの爪
A mechanical claw from an android.	ロボットが使っていた鉄の爪
Scepter	王様の杖
A jeweled cane for royalty.	王族が持つ装飾的な杖

Baseball	野球ボール
A white ball for playing baseball.	野球の競技用ボール
Tennis Ball	テニスボール
A green ball for playing tennis.	テニスの競技用ボール
Mittens	冬の手袋
Knitted mittens for cold weather.	寒い時に役立つミトン
Football	フットボール
A leather sports ball.	皮製のボール
Hockey Puck	ホッケーのパック
A black disc for playing hockey.	ホッケーの競技用パック
Golf Ball	ゴルフボール
A small white ball for playing golf.	ゴルフ用のボール
Music Player	MP3プレーヤー
A device for listening to music.	携帯音楽プレイヤー
Game Device	ゲーム機
A handheld device for playing games.	携帯ゲーム機

Tail items

Tail	尻尾
Tail Torch	尻尾のトーチ
A flame that burns forever.	一生燃えている尻尾の炎
Ankylosaurus Tail	アンキロザウルスの尾
A defensive tail that looks like a club.	先端に大きな骨塊がついた尾
Spiky Tail	スパイクの尾
A round tail with spikes.	先端に大きなトゲがついた尾
Dolphin Tail	いるかの尾びれ
A majesticly beautiful tail.	曲がりがない美しい尾びれ
Fish Tail	魚の尾びれ
A smelly, scaly tail.	うろこがついた魚の尾びれ
Cotton Tail	ふわふわのしっぽ
A cute fluffy puff.	ウサギのようなしっぽ
Lobster Tail	大海老の尻尾
An expensive, edible tail.	高級ロブスターの一部
Plane Tail	飛行機の尾翼
A stabilizing wing for flight.	飛行を安定させる翼
Comet Tail	彗星の尾
A flashy glowing tail.	輝く星のかけら
Plug	プラグ
An electrical plug for electric power.	電力を供給するプラグ
Ball and Chain	黒鉄球と鎖
A device for restraining prisoners.	囚人を拘束する道具

Seashell	貝
The hard shell of a marine creature.	海岸で拾った硬い貝殻
Paintbrush	絵筆
A tool for creating art.	芸術を奏でる素敵な道具
Scorpion Tail	サソリの尻尾
A highly dangerous tail.	先端に毒をもったサソリの尻尾
Lightbulb	電球
A glowing light.	明るい電球
Propeller	プロペラ
The front end of an old plane.	旧式の飛行機に使われるプロペラ
Magnet	磁石
A magnetic piece of metal.	鉄をひきつける磁石
Deer Tail	シカの尻尾
A fuzzy tail resembling that of a deer.	ごつごつした毛で覆われたシカの尻尾
Tail Buddy	謎の尻尾
A strange creature with its own brain.	独立した意思を持った尻尾
Drill	ドリル
A tool for digging through dirt.	掘る用の道具
Festive Bulb	行灯
A colorful light used during holidays.	祭りで使う明るい行灯
Training Wheels	補助輪
Small wheels to assist mobility.	子供の自転車の補助輪
Old Tire	古いタイヤ
A dirty tire from an old car.	古い車から汚いタイヤ
Old Boot	古いブーツ
A dirty old boot with many holes.	穴だらけ古いブーツ
Old Can	古い缶
A dirty old metal food can.	古い錫製の缶
Life Ring	浮き輪
A floatation device from a boat.	沈んだ船から浮き輪
Leaf	葉
A large leaf from a big plant.	大きな植物の巨大な葉
Tulip	チューリップ
A big flower with closed petals.	花びらが閉じている花
Wish Tag	短冊
A piece of paper with a wish on it.	願いが書いてある短冊
Bubble	バブル
A big bubble that doesn’t pop.	大きなシャボン玉
Feathers	羽
Colorful feathers imitating a bird’s.	鳥のような美しい翅
Paper Fan	団扇

A simple fan made from white paper.	白い紙製の地味な団扇
Goldfish Tail	金魚の尾びれ
A beautiful tail with a red tint.	赤くて美しい尾びれ
Tail Light	尾灯
A light from the back of a car.	車の尾灯
Hand Tail	手尻尾
A tail that resembles a hand.	手に似ている尻尾

Appendix F. Chat messages

Greetings

Greetings	挨拶
Hello	こんにちは
Goodbye	さようなら
What's up?	どうしたんですか？
How are you?	お元気ですか？

People

People	人
Mother	お母さん
Father	お父さん
Grandmother	お祖母さん
Grandfather	お祖父さん
Older brother	お兄さん
Older sister	お姉さん
Boyfriend	彼氏
Girlfriend	彼女
Cousin	いとこ
Friend	友達
Teacher	先生
Classmate	クラスメート
Archenemy	敵
Younger brother	弟
Younger sister	妹

Places

Places	場所
Home	自宅
School	学校
Work	仕事
Supermarket	スーパー
Mall	デパート
Restaurant	レストラン
WPI	WPI
OU	阪大

Food

Food	飲食

Hungry	おなか空いています
Thirsty	のどが渇いています
Rice	ご飯
Green tea	お茶
Tonkatsu	とんかつ
Water	お水
Soda	ソーダ
Coffee	コーヒー
Pizza	ピザ
Noodles	麺
Hamburger	ハンバーガー
Hotdog	ホットドッグ
Sushi	おすし
Curry	カレー
Fish	さかな

Animals

Animals	動物
Dog	犬
Cat	猫
Fish	魚
Snake	蛇
Hamster	ハムスター
Bird	鳥
Horse	馬
Cow	牛
Frog	かえる
Toad	ヒキガエル
Chicken	鶏
Elephant	象
Tiger	虎
Lion	ライオン
Bear	熊
Hippopotamus	カバ
Monkey	猿
Mouse	ねずみ
Deer	鹿

Emotions

Emotions	感情
Good	いい

Happy	うれしい
Sad	悲しい
Angry	怒っています
Surprised	驚いています
Excited	ワクワク
Bored	退屈
Curious	聞きたがる
Sick	病気

Questions

Questions	質問
Where are you now?	今どこにいますか。
What did you eat today?	今日は何を食べましたか。
Do you have any pets?	ペットを飼っていますか。

General

General	等
Yes	はい
No	いいえ
Maybe	かもしれません
Sure	ええ
OK	オーケー
Later	後で
No way	いや

Appendix G. Website questions

You are at a party. What do you do?			
Chat with friends	Meet new people	Sit on the couch	Eat food
パーティーで、どうする？			
友達としゃべる	人と出会う	ソファで座る	食べる
What is your ideal job?			
Journalist	Researcher	Businessman	Self-employed
理想的な仕事は？			
ジャーナリスト	研究者	会社員	自営
What kind of book do you like to read?			
Romance	Sci-fi/Fantasy	Classics	Comics/Manga
一番好きな本は？			
ロマンス	SF/ファンタジー	古典	漫画
What is your favorite snack?			
Popcorn	Cookies	Pretzels	Chocolate
一番好きなスナックは？			
ポップコーン	クッキー	プレッツェル	チョコレート
What is your favorite season?			
Summer	Spring	Fall	Winter
一番好きな季節は？			
夏	春	秋	冬
Which of these sports do you like?			
Football	Track	Archery	Couch
好きなスポーツは？			
アメフト	陸上	弓道	昼寝
A wild monster appears:			
Attack	Magic	Flee	Items
モンスターが現れた：			
アタック	魔法	逃げる	アイテム
A horde of zombies approaches. Choose your item.			
Baseball bat	Shotgun	Molotov	Riot shield
ゾンビの群が迫っている。道具を選んで。			
野球バット	ショットガン	火炎瓶	機動隊シールド

Which vehicle do you prefer?			
Pickup truck	Sports car	Motorcycle	SUV
好きな乗り物は？			
トラック	スポーツカー	オートバイ	RV車
You are going camping. What do you pack first?			
Fishing rod	Hiking boots	Binoculars	Snacks
キャンプに行こう。何をつめる？			
釣竿	ハイキングブーツ	双眼鏡	おやつ
Which animals do you see first at the zoo?			
Lions	Tigers	Bears	Oh my!
動物園で最初に見るは？			
ライオン	トラ	クマ	タイヘン！