Pet Game

Spring 2023 EC535 Final Project

Raina Yin
ryin2@bu.edu

Carlton Knox
cknox@bu.edu

Abstract

pet_game is a ”gacha”-like pet simulation game where players can unlock and collect a variety of pets, which can be sold to collect coins, buy eggs, and unlock more pets. This game was written in C++ using the Qt Framwork, and features a hierarchy of objects, widgets, and screens that work together to provide the player with the experience of owning and caring for a virtual pet. Additionally, this project also includes pet_game_screensaver, a screensaver program that is launched by a Linux kernel module based on user touch screen activity. Overall, pet_game provides a fun and experience for users interested in caring for and collecting virtual pets.

Links

GitHub Repository:
https://github.com/carltonknox/pet_game
YouTube Demo:
https://www.youtube.com/watch?v=CDX_zgWDI3c

I. Introduction

Many people are familiar with pet simulation games such as Tamagotchi, or Nintendogs. Tamagotchi specifically, was very popular in the 90s, selling over 80 million units worldwide. In our project we seek to recreate the classic Tamagotchi experience by creating a modern game called ”pet game”(pet_game) to provide users with the experience of caring and owning a virtual pet. ”pet game” is a gacha-game-like experience, where you unlock various pets of differing rarity, while trying to fill up your collection, similarly to Pokemon. pet game aims to provide users with a fun and engaging experience of caring for and owning a virtual pet.

Additionally, our project also includes a screensaver feature using the same game assets. Screensavers are a fun way for users to display the pets that they take care of and own in the game, while also providing some protection from screen burn-in on older displays. Although most modern monitors are not susceptible to burn-in, screen-savers can still be used to prevent damage in some cases, and provide a fun wallpaper while your computer is inactive.

Our pet game and pet game screensaver are both written in C++, using the Qt framework to display graphics, and interact with the user inputs. There are a total of four screens: main menu, purchase screen, crack egg screen, and pet screen. Each screen features different functionalities that completes the experience of owning virtual pets. The game itself features a variety of pets, each with their own sprites, animations, descriptions, and rarity. In order to collect all the pets, the player must crack eggs, and accumulate coins by selling pets. The screensaver also includes a Linux Kernel module that runs in the background, launching and terminating the screensaver when there is no activity.

II. Design Flow

pet game is comprised of a screen saver and the pet_game executable itself. The screensaver features a pet_game_screensaver executable, and a corresponding kernel module which launches the screensaver. The game is comprised of several classes and screens, some of which are shared by the screensaver. The following diagram describes a detailed hierarchy of pet game.

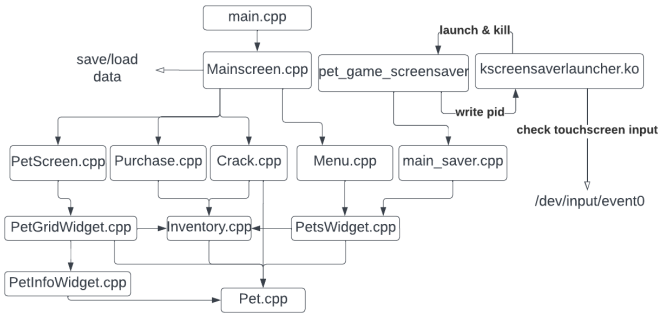

Fig. 1. Overall system design and class hierarchy

Mainscreen.cpp creates a stackwidget that connects PetScreen.cpp, Purchase.cpp, Crack.cpp, and Menu.cpp. Inventory.cpp contains the variables eggcount and coincount as well as a user Pet vector. MainScreen.cpp creates an inventory object which is passed into each screen object so all the screens have access to the same information. Menu.cpp is responsible for controlling the menu buttons and deciding which screen from the stackwidget should be currently displaying. Menu.cpp also contains an instance of PetsWidget.cpp, which displays all the users current pets, as defined in the inventory’s pet vector, bouncing around on the menu’s background. Purchase.cpp creates a button that allows the user to purchase eggs by using coins, and Crack.cpp allows the user to spend eggs to receive a random pet based on the rarity level.

Within the PetScreen, there is a PetGridWidget that displays all the users current pets from the inventory as a scrollable grid. When the user clicks on a pet, it opens the PetInfoWidget

for that pet, displaying the pet's sprites, name, description, rarity, and sell price, while also inluding a sell button.

The screensaver executable is defined by main_saver.cpp, which just creates an instance of the inventory, Pet vector, and PetsWidget. The pet vector is passed into the PetsWidget, which displays the pets on the screen as a screensaver. The screensaver communicates with a kernel module named kscreensaverlauncher. The kernel module is expected to launch the screensaver, and then terminates the program when user input is detected again. Upon launch, the pet_game_screensaver will write its pid into the kernel module, allowing the module to kill it.

Contributions:

- `main.cpp` — Raina
- `Mainscreen.cpp` — Raina
- `main_saver.cpp` — Carlton
- `kscreensaverlauncher.c` — Carlton
- `PetScreen.cpp` — Carlton
- `Purchase.cpp` — Raina
- `Crack.cpp` — Raina
- `Menu.cpp` — Raina
- `PetGridWidget.cpp` — Carlton
- `PetInfoWidget.cpp` — Carlton
- `Inventory.cpp` — Raina
- `PetsWidget` — Carlton
- `Pet.cpp` — Carlton
- `Artwork` — Raina

Credit:

- Carlton Knox 50%
- Raina Yin 50%

III. Project Details

As described above, our project is comprised of several classes, widgets, and screens, all woven together like a fine silk scarf. See below for more details:

A. Pet.cpp

Pet.hpp defines the Pet class, as well as the helper function *generatePets()*, which defines the vector containing the set of all possible pet types the player can unlock. The Pet class itself defines each Pet as having an id number, a name, a description, two sprites, and a rarity. Additionally, there are some other values included to help with graphics such as a sprite_state (used to switch between the two sprites), a bounding visibleRect (used to remove white space margin around sprite images), and position and velocity values. The class additionally defines a variety of "get" functions, as well as an *updateSprite()* function (used to switch the sprite state).

B. PetsWidget.cpp

PetsWidget is a QWidget used to display the users pets by having them bounce across the screen, similar to the classic DVD logo screensaver. This widget is used both in the games main menu, as seen in Figure 8, as well as the pet_game_screensaver, as seen in Figure 9. PetsWidget redefines *QWidget::paintEvent()* to draw a pixel map of each pet's sprite based on its x and y position. Then, it adds a new function to update the position of the pets based on the velocity and location on the screen called *bounceImages()*. *bounceImages()* is called by a timer every 20ms, which is set up in the widget's constructor. To calculate the position, the function adds the current velocity to the current position, and then checks if the pet has reached the boundaries of the screen. It does this by using the pet's *visibleRect*, the bounding rectangle defining the dimensions of the visible region of the sprite (removing the margin of transparent pixel). If the visible sprite rectangle is beyond the screen's boundary, it moves the sprite back into the screen, and reverses its velocity. This creates the effect of bouncing pets.

C. Inventory.cpp

The inventory class is an instance of QObject. To keep track of the variables created in the game, it contains two private variables eggCount and coinCount and a public vector user_list. The inventory class also has numerous public functions to allow other classes manipulate these variables which will be discussed more in detail later. Moreover, there is a public QReadWriteLock to ensure that only one thread can have access to the data at a time.

D. PetScreen.cpp

Petscreen defines a QWidget that contains the PetGridWidget, and creates a return button that, like the other screens, emits a *returnToMain* signal. However, this return button can also be used to close the Pet Info screen if it is visible. The same button achieves this by checking if the Pet Info screen is visible, and closing it if it is, otherwise, returning to the menu screen.

E. PetGridWidget.cpp

The PetGridWidget class defines a touch-scrollable grid of the player's current pets. Each pet is shown with a custom QLabel class, called PressLabel. This makes it so that each pet icon can be pressed, launching the Pet Info screen for the corresponding pet. The pet grid defines a 4 column grid, which can have as many rows as needed. To add each pet icon to the grid layout, the class loops through the user pet list and assigns them to locations on the grid. Whenever new pets are added or pets are removed, the grid needs to update, and potentially rearrange the pet grid. To do this, the *PetGridWidget::updatePets()* function loops through the current number of pets, adding new sprite labels if they exceed the previous number, and removing all the previous sprites from the grid if there is any change. Then, after updating all of the sprites based on their animation state, the function will re-add all the pet icons to the grid, if there has been any change. Finally, it will remove any excess icons that are no longer being used. To ensure thread safety, a reader/writer lock is used while acessing the user pet list.

When a pet icon is pressed, it will run the *showPetInfo()* function, which opens the Pet Info Screen on top of the PetGridWidget.

F. PetInfoWidget.cpp

The PetInfoWidget is a class used by the PetGridWidget, to display the selected pet's information. The Pet Info screen is instantiated by the Pet Grid screen, but made invisible by default. Only when the PetGridWidget's *showPetInfo()* function is called, will the Pet Info screen be made visible, after updating which pet is being shown.

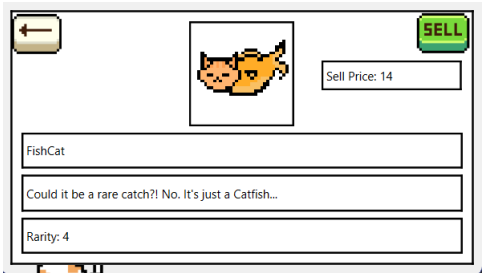

Fig. 2. Pet Info Screen for a medium-rarity pet.

The Pet info Screen displays the pet's sprite, name, description, rarity and sale price, as shown in Figure 2. On each screen, including the pet info screen, the pet's sprite is updated every 500ms between the two states, and redrawn accordingly.

The Pet Info Screen also has a Sell button, which allows the user to remove pets from the pet list, while increasing their coin count. The pet's sale price is calculated using the following formula:

$$sale\ price = \frac{2}{7} \times rarity^2 + 2 \times rarity + 2$$

Based on the rarity, the pet can be sold for a higher value. For example, a rarity 7 pet would sell for 30 coins, as shown in Figure 3.

Fig. 3. Pet Info Screen for a high-rarity pet.

When a pet is selected on the Grid Widget, the PetInfoWidget's *setPet()* function is called, which updates the pet, sets the sprite, text, rarity, and recalculates the sale price.

G. Purchase.cpp

The Purchase class takes in two important arguments in the constructor–stackedWidget and inventory. It also obtains two QPushbuttons as shown in Figure 4. The center button allows the user to increase the egg count, each carton costing 7 coins. To do that, the uses the addEgg() function and removeCoin() function from inventory class to manipulate the variables. The second QPushbutton, once pressed by the user, will emit a signal that returns to the menu by setting the stackWidget index to 0.

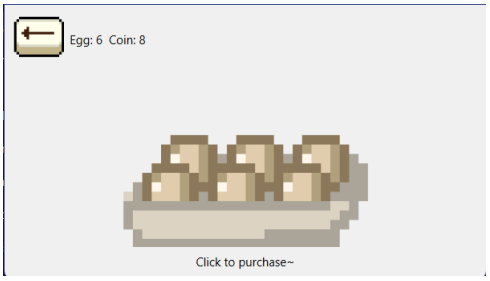

Fig. 4. Screen to puchase eggs

H. Crack.cpp

Similarly to Purchase.cpp, the Crack class takes in stackedWidget and inventory as two arguments in the constructor, which is used in a similar fashion to the Purchase class. The Crack class also contains two buttons, a return button that emits a signal when pressed to return to menu. And another one that allows to user to obtain a random pet by cracking the egg. Each time the egg button is pressed, a state machine changes state. State 0 displays a PNG of an egg (Figure 5). State 1 displays a PNG of a cracked egg (Figure 6). And state 2 calls the generateRandomPet function which calls the generateNumber function. The generateNumber() function randomly selects a number from 0 to 10 based on a geometric distribution. The generateRandomPet takes in the random number, finds all the pets that have the matching rarity value, and randomly chooses a pet that has the same rarity to add to the user_list.

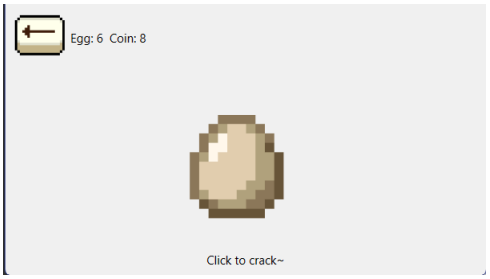

Fig. 5. State 0 of the state machine

I. Menu.cpp

As shown in Figure 8, the Menu class has 3 QPushButtons and a set of user_list pets bouncing in the background. The Menu class takes in the stackedWidget and inventory as the two arguments of its constructor. StackedWidget is stores in a private variable and each time each button is pressed, the corresponding stackedWidget index will be switched to. To generate the bouncing pets in the background, the Menu class creates an instance of the PetsWidget class and passes in inventory which provides PetsWidget the vector of the current pets that the user has.

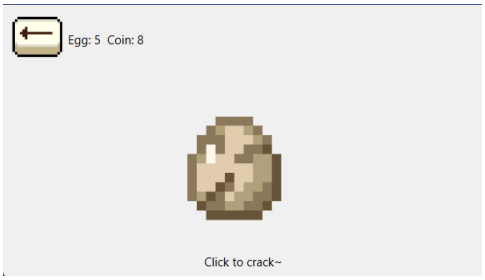

Fig. 6. State 1 of the state machine

Fig. 7. State 2 of the state machine

J. MainScreen.cpp

The MainScreen class is where everything gets tied together. It creates an instance of each screen–Menu, Purchase, Crack and PetScreen. It also creates an instance of the inventory class and QStackedWidget which is passed into each screen as we discussed above. After everything needed is properly initialized and declared, we add each screen as a widget to the stackeWidget and set the initial index as 0. Thus, the menu screen displays as the defaul screen. The mainScreen class also saved and loads all the available data. Everything the program starts, it calls loadInventoryFromFile to load any available data. And everytime the program gets an interrupted signal, the mainScreen class calls writeInventoryToFile and writes eggCount, coinCount, the lenght of user_list, and the id of each pet into a file called inventory.txt.

K. main_saver.cpp

The pet game screensaver uses a separate main file that instantiates an Inventory and a PetsWidget. The screensaver visually is just the bouncing pets from the background of the games menu. However, in order to turn on and off the screensaver based on the user's activity, this executable must be run by the kernel module. In order to stop the screensaver, the kernel module needs the launched screensaver's process ID, so as soon as the screensaver program starts, it tries to open */dev/kscreensaverlauncher* and writes its own pid into the module. Then, it runs until it is killed by an external signal.

Fig. 8. Main menu display

Fig. 9. pet_game_screensaver's output, using PetsWidget

L. kscreensaverlauncher.c

kscreensaverlauncher.ko is the kernel module that launches the screensaver. The kernel module defines a write() operation that expects a *pid_t* to be written. This is saved as the screensaver's pid, which is later used to kill the screensaver. When the kernel module is initialized, it launches a kthread that runs in the background until the module is removed.

First, in order to check for user activity, the module opens /dev/input/event0, which corresponds to the touch screen input on the beaglebone. The thread will then repeatedly try to read from the touch screen input until there is a user touch event. When the user touches the screen, the time of the most recent touch event is recorded. Once it has been more than 15 seconds since the last touch event, the kernel module will use a subprocess to launch the screensaver. Then, after the screensaver has written its pid back into the module, if the time since the last press is less than 15 seconds, it will send a *SIGKILL* signal to the pid. When the kernel module is removed, the thread will exit after closing the input.

The kernel module is initialized with the following commands:

```
mknod /dev/kscreensaverlauncher c 61 0
insmod ./kscreensaverlauncher.ko
```

and removed with:

```
rmmod ./kscreensaverlauncher.ko
```

IV. Summary

In conclusion, the pet game is a recreation of the Tamagotchi pet simulator game. The game provides a gacha-game-like experience where players unlock pets of differing rarity to complete their collection. The project also provides a screensaver feature that displays the pets the user owns. The game and screensaver are both written in C++ using the Qt framework. The screensaver includes a Linux Kernel module

that runs in the background, launching the screensaver when there is no activity. The pet game successfully creates a fun and engaging implementation of the pet simulation game as well as a screensaver. The remaining challenges of this game will be to implement multi user functionalities, add more interactive elements to the pets, and implement an installer to make it easier to set up the game on any platform.