FRUIT NINJA

Team Legen-dary

Baris Tevfik (bt@bu.edu)
Zhibin Zhang (zbzhang@bu.edu)
Xingye Liu (dandan@bu.edu)

ABSTRACT

We have designed a touchscreen game using Qt framework for Gumstix Verdex. The game is based on the very popular IPhone game called Fruit Ninja. The main object of the game is to slice as many fruits as possible, that are shot from bottom of the screen before running out of number of attempts. Our version of the game is very similar to the original version in terms of gameplay. It has 2D graphical elements as opposed to 3D elements that are found in the original game. This was a design choice due to limitations with Gumstix and Qt, as well as the short amount of time that we had to complete the project. At the end, we have successfully implemented the game with some minor bugs that we will discuss later in this paper.

1. INTRODUCTION

In daily life, people enjoy playing Fruit Ninja game in their IPhone, IPad or Android device. This game is a fruit slicing game played on touchscreen where the player slices objects thrown into the screen with their finger. In our project, we created a similar game that is developed in Gumstix that acted as core and LCD touchscreen that acted as the playing platform. Our main reason and motivation to work on this project was to get more knowledge on game development on embedded systems. We programmed the game in C++ using Qt framework.

Qt is a cross-platform application framework that is widely used for developing application software with a graphical user interface (GUI) (in which cases Qt is classified as a widget toolkit), and also used for developing non-GUI programs such as command-line tools and consoles for servers [1]. Qt is applied on many embedded systems. Both embedded system and Qt are important technology topic in the future. Embedded systems are designed to do some specific task, rather than be a general-purpose computer for multiple tasks. Thus, they consume more less energy and can dedicate on only one task. Specifically, we used Gumstix Verdex in this project.

The project consists of 3 main hardware parts. These are the Samsung 4.3 Inch LCD Touchscreen, Gumstix Verdex and micro SD card. Gumstix is the core of whole system and supplies processing. The touchscreen provides all the user interactions as the sole input/output device. We configured touch screen by changing the environment variables so that it acts as a mouse device.

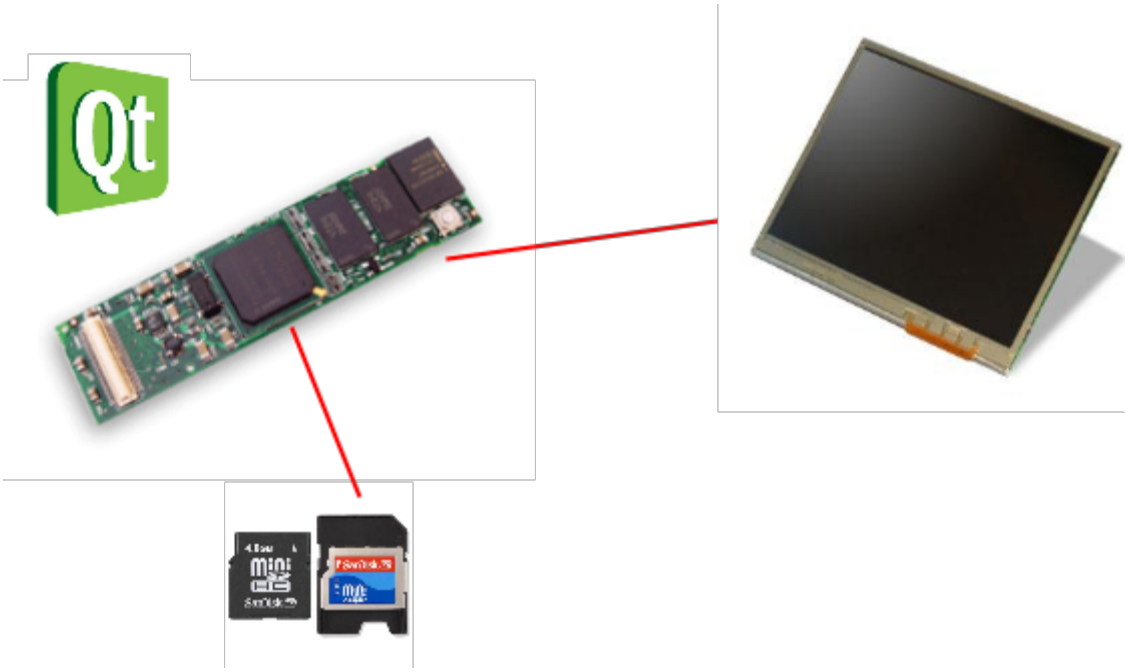

Figure 1. Main components of the project.

The main software component of the game is Qt , specifically we have used the Qt Graphics View Framework [2]. We have configured Qt to work with our cross compiled executable on Gumstix by linking libraries from SD card to `/usr/lib` . The reason behind this is that there is no enough storage space in Gumstix that can hold large library files.

We have developed the fruit ninja game in 2D through the use of QGraphicsView class that holds the main scene of the game where items and widgets are added into QGraphicsScene. We have been able to improve the stability of the game after having some issues with memory leaks but have not been able to solve the problem entirely.

Combining Qt and embedded device Gumstix together, we developed this game using just three hardware components. Technically, the small scale is the advantage of embedded systems. In this report, we will show how our game has been developed.

2. DESIGN FLOW

Our game has 3 main parts in the software implementation. First is the main functions that implements the animation of the balls. This module can be seen in figure 2. First we initialize the components of the game such as creating the scene, adding a background to the scene and adding objects (balls) into the scene. Then, the main thread starts (when user presses start button). This main thread spawns 3 other threads which each contains a an animation with a timer. The animation starts moving the balls into the scene. When the timer finishes for each animation, another function is called (using connect signals). This function, resets the positions of the ball to a random x coordinate value. Then sets a flag indicating that the ball is `dead`. At this point, the main thread picks up flag and spawns a new thread for the corresponding ball, and the main flow continues so on. (Baris has mainly worked on this part.)

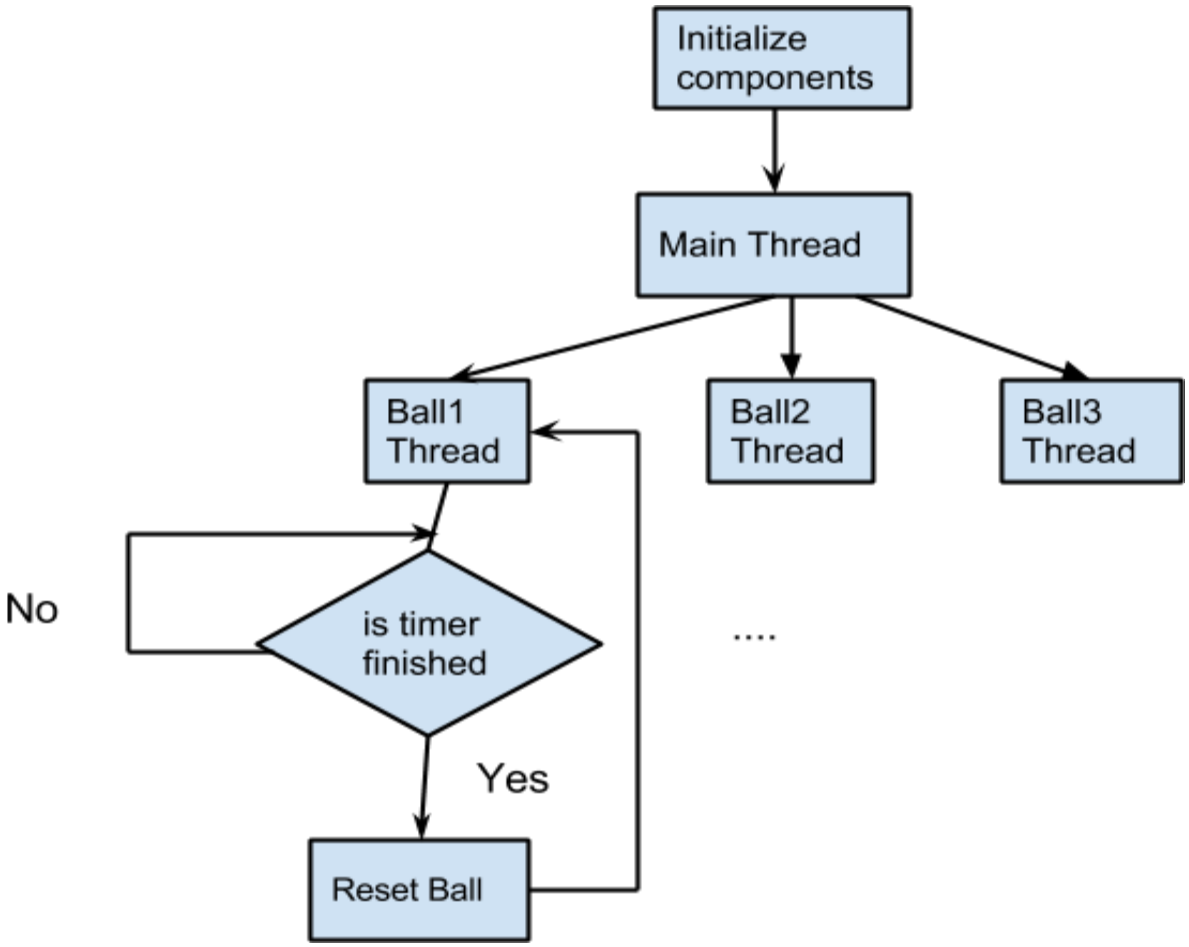

Figure 2. Main Functions.

The detection of slicing is handled separately from the animation of the balls. The diagram can be found in figure 3. When a touch is detected through the Qt Mouse Event, the position of the mouse is retrieved and if it matches to any of the ball's current global position for 2 mouse events, the ball is determined to be sliced. Then, the ball is hided from the scene, and finally the score is updated. (Zhibin has mainly worked on this part.)

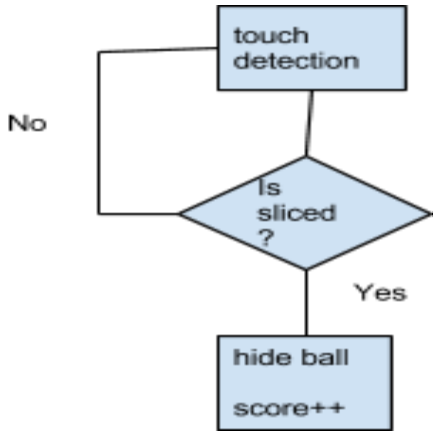

Figure 3. Slicing the ball

The other major module of the game is to start and pause the game. Figure 4 shows the flow diagram. When the start is pressed for the first time, the main thread that can be found in figure 2 starts. Then, the button turns into a pause button, and when the button is pressed again the animations are paused in each thread. (Liu has mainly worked on this part.)

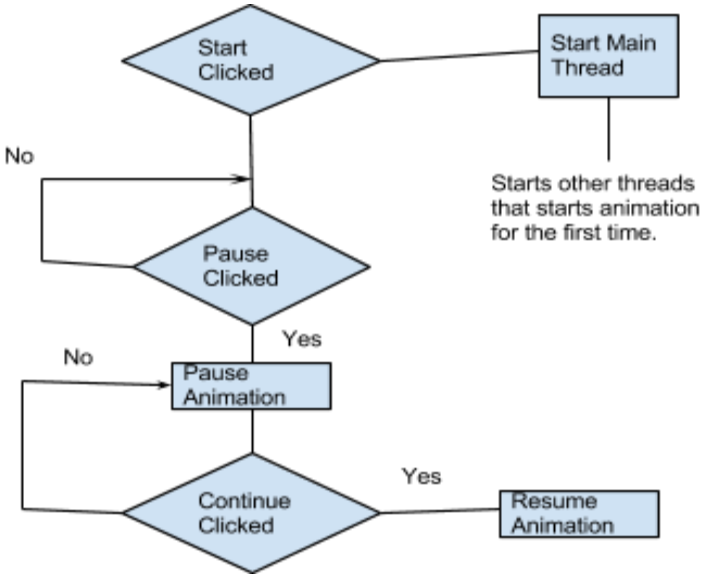

Figure 4. Pausing and Resuming the game.

3. PROJECT DETAILS

Figure 5. Screenshot of the original Fruit Ninja.

In this chapter we will discuss the modules and parts of the game in detail. These will be initialization of the system, main thread, individual threads, detection of the slicing and pause/continue module. We will also discuss game optimization and problems we have encountered as well as the bugs that the game currently holds. Figure 5 gives a screenshot of the original game. We have tried to keep the overall design similar to the original game. A picture of our version can be seen at the end of this chapter.

I. Game Initialization

The game initialization creates the scene [3], adds item to the scene, sets the background and starts the main application. From the `ts_test` program, we have determined that the size of the Samsung LCD screen is 460 x 260. For this reason we configure the scene of our game to have the appropriate size.

We also set up a global pointer to the view so that we can access the scene from other parts of the program. We hide the cursor from the game so that we can display `knife` effect similar to the one that is found in the original game. This feature will be discussed later in this chapter.

```
QGraphicsScene scene;
scene.setSceneRect(0,0,460,260);
```

```
QGraphicsView view(&scene);
viewPointer = &view;
view.setCursor( QCursor( Qt::BlankCursor ) );
```

a. Adding background and title to the scene

We store the background in the sd card and load it to a QPixmap. Then, we set the background of the scene to the QPixmap. We also add a title to the game.

```
QPixmap pim("/media/card/wood.png");
scene.setBackgroundBrush(pim);

QGraphicsTextItem * teamTitle = scene.addText("Ninja by Team
Legendary",QFont("Times", 18, QFont::Bold));
```

b. Adding circles to the scene

The game has 3 circles that we initialize at the beginning and add to the scene. These circles are used throughout the program. First two numbers are the positions of the ellipse and later two give the size of the ellipse. Since width and height here are same, this creates a circle.

```
QPen pen;
pen.setColor(Qt::transparent);
item1=scene.addEllipse(90,290,50,50,pen,QBrush(Qt::red));
… //add item2 & item3
```

II. Main Functions

In order to animate three circles independently and be able to set when the animation starts we create three individual threads in the main thread. The main thread runs in a while loop until all the lives have run out and creates new threads for each ball when it detects that a ball has died.

```
void Worker::mainWork(){
  while(life>0){
     if(item1alive->isAlive()==0){
     Worker *worker1 = new Worker;
     QThread *workerThread1 = new QThread(this);
     …
     workerThread1->start();
     …
     //other threads are created here.
```

a. Individual threads

Under the function “Worker::ballwork” , animation class is used to make the item move following the specific route. It sets a timer to make a limit of the animation which means the item just moves in this timeline. Before the animation be able to start each thread sleeps for a different amount of time so that each ball isn’t thrown into the scene at the same time. We have calculated the best possible function to simulate the ball is thrown into air.

```
sleep(3);
…
for (int i = 0; i < 50; ++i) {
animation1->setPosAt(i / 50.0, QPointF(i, 0.32*i*i-16*i)); }
timer1->start();
```

b. Resetting items

This part includes the reset of the item. When a timer is finished for an individual item, it signals to the resetBall function which randomly sets a starting position for the ball, then set the ball to `dead`. At that point the main thread picks up that the has died and creates a new thread to start the animation.

```
void Worker::resetBall1(){
     …
     if(item1alive->isAlive()==1){
     int x = rand() % 350 + 50;
     item1->setRect ( x, 290, 50, 50 );
     item1alive->dead();
}
```

III. Detection of Slicing

The cursor position detection is supplied by the built in mouseMoveEvent function. This function is called whenever the player touches the screen. The position of each ball at that point is calculated and tried to be matched with the position of the cursor. If there is a match, then the ball is hided from the scene. And the score is incremented.

```
void QGraphicsScene::mouseMoveEvent(QGraphicsSceneMouseEvent
*event)
{ …
 if ( diffx1<40 && diffx1 > -40 && diffy1 <40 && diffy1 > -40)
   {
```

```
…
a.setColor(Qt::transparent);
item1->setBrush(a);
isItem1Sliced = true;
score++;
…
```

diffx1 and diffy1 holds the difference between position of cursor and position of the ball. If the difference is less than 40, for 3 different mouse events, the ball is marked as sliced.

IV. Start/Pause Button

We created a custom widget that holds the start/pause button. And connected the button to a function to be signalled when the button is clicked. When the button is detected as clicked, depending on the state of the button three different things can happen. If the button is clicked for the first time then the main thread is started, if the button is clicked again, the timers for the animations are paused. When the button is clicked for the third time the timers are resumed. The state of the button changes between 1 (pause) and 2 (continue) after the initial start.

```
void NinjaWidget::buttonClicked(){
  if(buttonSelect==0){
     //start main thread
     buttonSelect=1;
  }
  if(buttonSelect==1){
     //pause timers
     buttonSelect=2;
  }
  if(buttonSelect==2){
     //resume timers
     buttonSelect=1;
  }
```

V. Miscellaneous

a. Score, Life and Game Over labels

We created three labels for score, life and GameOver. When the number of score or life changes, the text in the labels get updated. There are 5 lives given to the player at the beginning of the game. When number of lives reach 0, GameOver label is displayed on screen. The life is based on whether you slice an item before it drops down bottom.

b. Knife shape following finger

When touch is detected on the screen, the Qt gets and saves one point position. After it gets the second one, it draws a line between these two points. In our code, we set the color of line be white. The line is based on a continuously pressing. If not, line will disappear. This resembles the knife effect that is found in the original game.

VI. Memory Leak Problem

We have encountered some crashes due to memory leak in the program. Our original code was trying to create new items on the scene every time an animation started. Even though each item gets removed from the scene using `removeItem` method, the objects don't get deleted. We tried to use `delete` operator to destroyed the objects after removing from the scene but that started causing segmentation faults. So we decided to keep the original 3 circles in the scene and update their colors to transparent to have the same effect as removing the item from the scene. This has greatly reduced the crashes but did not eliminate all. It also seems that creating white lines (for knifing effect) also contribute to this problem, however we have not been able to solve the issue completely with this part. Unfortunately problem with memory leaks is a general and tough topic that occurs in many programs.

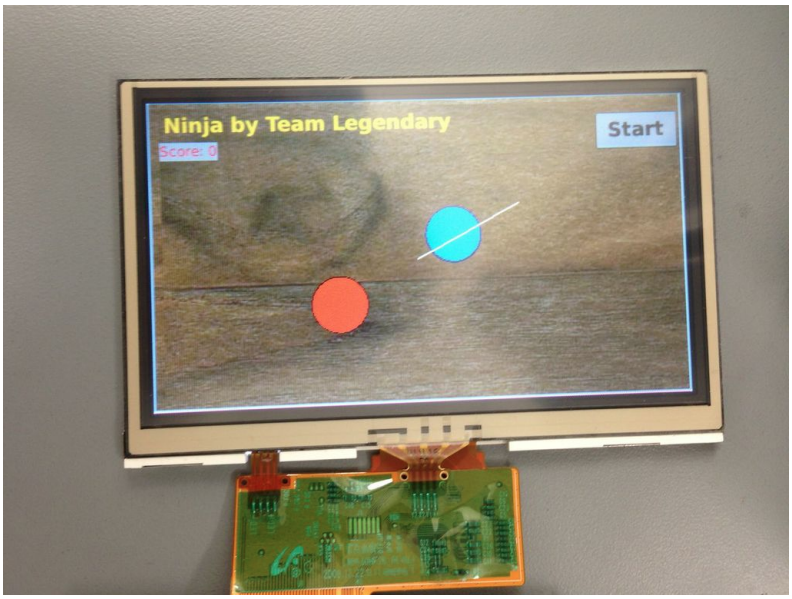

Figure 6. Picture of our version of the game.

4. SUMMARY

Our project, Fruit Ninja is an object slicing game based on touchscreen. It is developed in Gumstix. We programmed our code in C++ and Qt framework and cross compile it in Gumstix. We accomplished Qt programming including item animation, thread creation, object communication (signals) and so on. Finally there can be three balls with different colors flying on the scene in the screen and they are being updated all the time. Technically, more functionality can be added easily as wanted since they are all widgets that can be shown on QGraphicsScene. There are remaining challenges to further develop the game such as extending 2D game into 3D. And it is also needed to be optimized further to eliminate the memory leaks.

References

[1] “That Smartphone Is So Qt". *Ashlee Vance*. 16 February 2010. Retrieved 19 February 2010.

[2] Qt GraphicsScene class Docs. (http://qt-project.org/doc/qt-4.8/qgraphicsscene.html)

[3] Qt GraphicsView Framework Docs. (http://qt-project.org/doc/qt-4.8/graphicsview.html)