**임베디드 시스템 LAB 4 실험보고서**



**팀 4**

**201501509 김주용**

**201601647 노윤표**

**201601659 이인호**

**201601663 장현빈**

**목차**

1. 과제

1.2 목표

2. 과제 수행

2.1 파트 분배

2.2 개발 환경

2.3 기본적인 설계 및 구현

**∙** 하드웨어 구성

**∙** 파일 구성 및 구조

2.4 User Interface

**∙** Screen.c

**∙** 색상 정보 초기화 및 출력

2.5 기본적인 파일 설명

∙ Header.h

∙ Function.h

∙ Var.h

∙ CommonFun.c

∙ DrawFunction.c

**∙** main.c

2.6 구체적인 구현

∙ 전체적인 동작

∙ Line 함수

∙ Rectangle 함수

∙ Oval 함수

∙ Free Draw 함수

**∙** Select 함수

**∙** Erase 함수

**∙** Clear 함수

**∙** Fill 함수

**∙** 기타 등등

3. 결론

**1. 과제 선택**



저희 팀은 Touch – screen을 활용한 그림판 구현 과제를 선택하였습니다.

**1.2 목표**

\* Touch-screen 을 입력장치로 하여 LCD 상에 그림을 그리는 프로그램을 구현

\* 버튼을 선택하고, 스크린을 touch하여 그림이 그려짐

\* Line, Rectangle, Oval은 rubber-band 지원

\* Free draw는 입력된 선을 smooth하게 연결 (점의 연결이 끊어지면 안됨)

\* Select는 그려진 object를 이동. Free draw object도 선택 가능

**2. 과제 수행**

**2.1 파트 분배**

**201501509 김주용** : 본인이 한 일 작성

**201601647 노윤표** : Line 구현 및 버그 수정

**201601659 이인호** : 본인이 한 일 작성

**201601663 장현빈** : UI 구성, 전체적인 동작 및 FreeDraw, Select, Erase, Clear, Fill 함수 구현

**2.2 개발환경**

**1. Github**



팀원들과 서로 코드를 공유하고, 서로 피드백을 해 주기 위해서 Github를 활용하여 과제를 진행함

Github주소 : <https://github.com/soulsystem00/embedded-project>

위의 주소로 들어가면 전체 코드를 다운받을 수 있음

**2. Linux**

Odroid 연결 및 프로그램 실행을 위해 리눅스에 연결하여 과제를 진행

사용된 리눅스 버전 : 16.04.6 32bit 및 18.04.4 64bit

**3. GCC 버전**

gcc-4.9.real (Ubuntu/Linaro 4.9.2-0ubuntu1~14.04) 4.9.2

**4. 오드로이드 커널 버전**

Linux odroid 3.10.107 #2

SMP PREEMPT Tue Aug 20 06:08:00 PDT 2019 armv7l armv7lx

**2.3 기본적인 설계 및 구현**

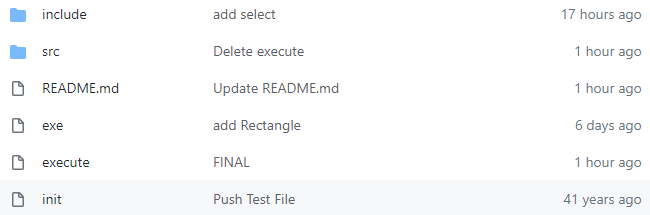
**하드웨어 구성**



추가적인 버튼과 같은 input 없이, 터치스크린만을 사용하기 때문에, Odroid 보드에 직접 TFT LCD를 연결한 형태로 하드웨어를 구성

**파일 구성 및 구조**

- 전체적인 파일 구성



**include** : 코딩에 필요한 헤더파일이 들어있는 디렉토리

**src** : 소스파일이 들어있는 디렉토리

**exe** : 코드의 컴파일 및 프로그램 실행에 관한 명령이 들어있는 파일

**init** : TFT LCD의 초기화에 관한 명령이 들어있는 파일

위의 기본적인 파일 구성을 마친 후 각자 파트를 분배하여 개발을 진행함

**2.4 User Interface**

**Screen.c**

Direction에서 제공된 해당 그림과 유사한 UI를 구현하기 위해 Screen.c 파일에 320 \* 240 크기의 1차원 배열(76,800개의 index)로 구현함





- Screen.c

이러한 형태로 UI를 표현하기 위한 데이터가 저장되어 있으며, 이를 사용하기 위해 소스파일에서 출력을 위한 코드 작성을 진행

**색상 정보 초기화 및 출력**



main.c 에서 screen 배열의 데이터를 사용하기 위한 선언 및 우측 상단의 color를 표현하기 위한 픽셀 정보 초기화

위의 색상 정보는 전역변수로 선언 되어있어 그림을 그릴때도 이용됨

**배열 출력**



헤더 파일에 구현된 Screen 배열을 이용해 터치 스크린에 픽셀을 표현하는 PrintScreen 함수.

**색상 정보 출력**



우측 상단의 color 값을 표현하기 위해 직접 좌표를 이용해 픽셀 값을 수정하는 FillinitColor 함수

(8개 색상에 대한 좌표에 대해 모두 구현이 되어 있음, 코드는 동일하기 때문에 생략)



UI 디자인 구현을 완료 한 모습

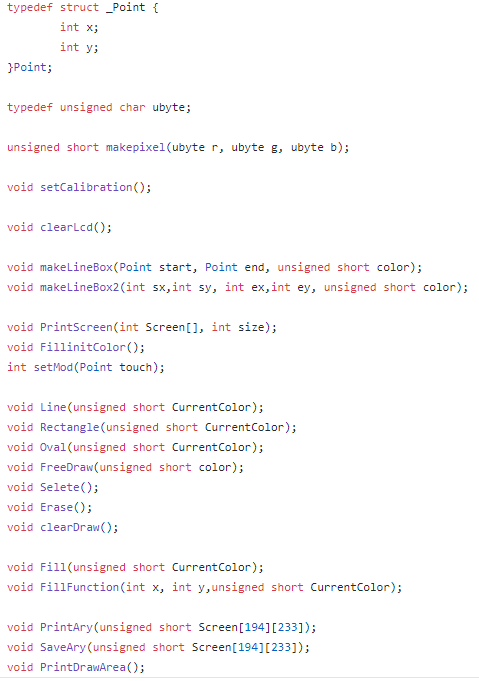
**2.5 기본적인 파일 설명**

필요한 모든 변수들과 함수들을 따로 정의하여 파일을 만듬.

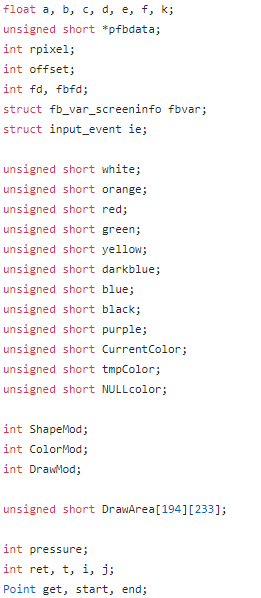
**Header.h** - 프로젝트에서 사용할 헤더들



**Function.h** – 프로젝트에서 사용할 함수들을 선언한 파일



**Var.h** – 프로젝트에서 사용할 전역변수들을 사용한 파일



**CommonFun.c** – 그리기 위해 사용되는 7개의 함수(Line, Rectangle, Oval, FreeDraw, Select, Erase, Clear)이외의 것들을 구현해 놓은 파일

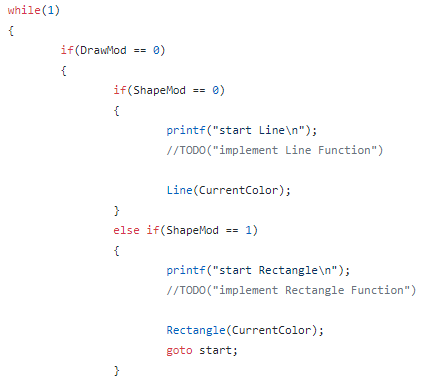
ex)



CommonFun.c에서 구현된 makepixel 함수와 setCalibration 함수.

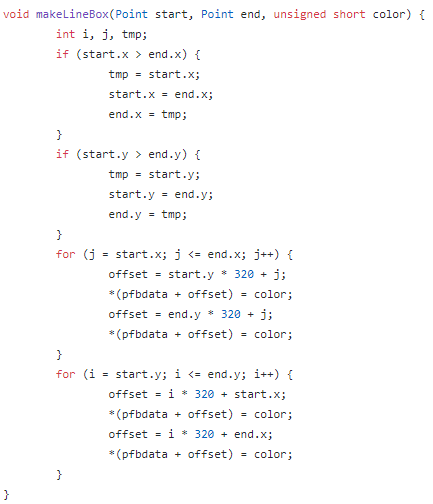


CommonFun.c에서 구현된 clearLcd 함수

**main.c –** 프로그램의 전체적인 동작에 관한 코드를 작성해 놓은 파일 - main.c에서 구현된 동작의 일부분

**DrawFunction.c -** 그리기 위해 사용되는 7개 함수들(Line, Rectangle, Oval, FreeDraw, Select, Erase, Clear)을 구현해 놓은 파일

ex)



DrawFunction.c에서 구현된 makeLineBox 함수

**2.6 구체적인 구현**

**전체적인 동작**

전체적인 동작 UI의 동작과 그리기 알고리즘에 관한 항목이다.

전체적인 동작 UI의 동작은 아래의 알고리즘을 바탕으로 동작한다.

1. 그리기 모드 선택

2. 그리기 구역 터치 시 그리기 모드로 진입

3. 그리기

4. 그리기 구역 밖에 터치 시 그리기 모드를 빠져나감

위의 알고리즘을 반복하면서 그림판 프로그램은 동작한다.

그러면 1번부터 살펴보자

**1. 그리기 모드 선택**

그리기 모드 선택은 setMod함수에 의해 일어나고 화면의 각 버튼들을 터치하면서 선택할 수 있다.

아래는 setMod함수의 일부분이다.

**if ((touch.x >= 4 && touch.x <= 75) && (touch.y >= 34 && touch.y <= 55))**

**{**

**printf("sets mod Line\n"); ShapeMod = 0; DrawMod = 0;**

**}**

위의 코드를 보면 touch 좌표의 값에 따라 ShapeMod와 DrawMod 변수를 설정하는 것을 알 수있다.

즉, 화면의 표시된 버튼을 터치함에 따라 각 변수들이 변하면서 모드를 설정하게 된다.

**if (ShapeMod == 0)**

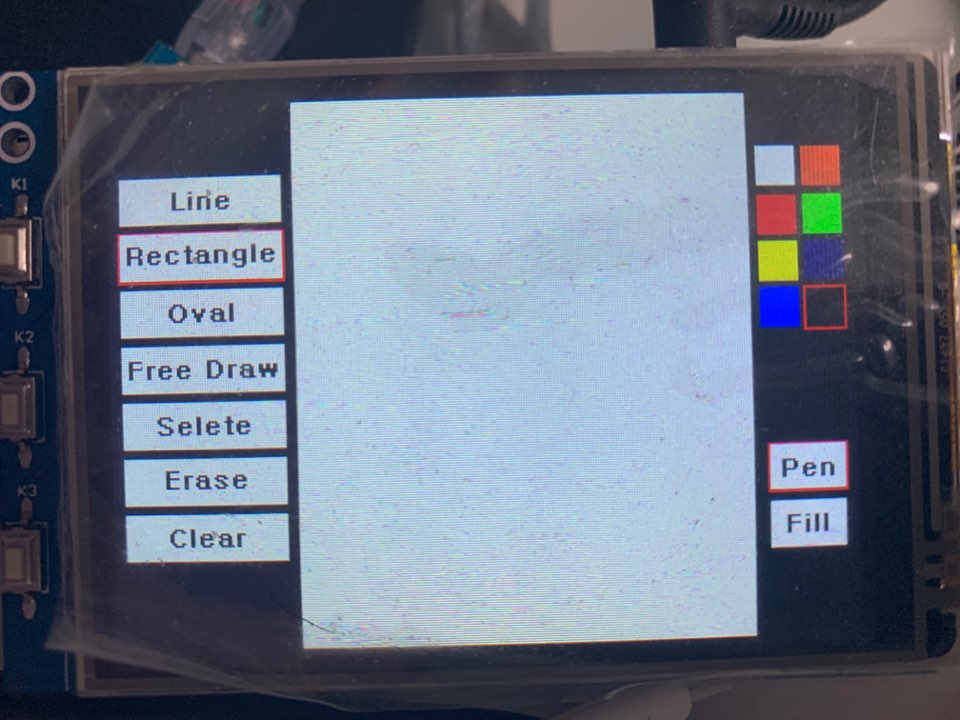
**{**

**ShapeStart.x = 4; ShapeStart.y = 34; ShapeEnd.x = 75; ShapeEnd.y = 55;**

**}**

위의 코드는 자신이 선택한 모드를 화면에 표시해주기 위해 각 버튼의 좌표 값을 저장하는 코드이다.

이렇게 저장한 좌표들을 바탕으로 makeLineBox를 이용해 화면에 자신이 선택한 모드를 표시해주게 된다.



위의 화면을 보면 Rectangle, Pen, 검정색상이 선택된 것을 확인할 수 있다.

즉, 그리기 모드뿐만 아니라 색상선택 펜 선택까지 setMod에서 설정이 된다.

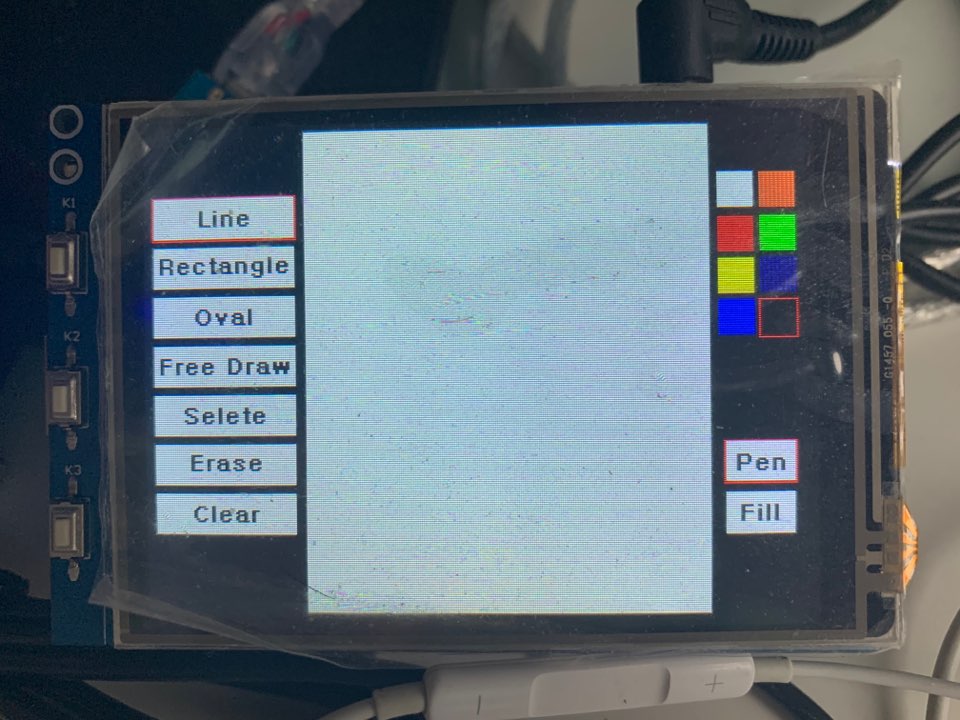
**2. 그리기 구역 터치 시 그리기 모드로 진입**

그리기 모드를 설정했다면 이제 그림을 그릴 차례이다.

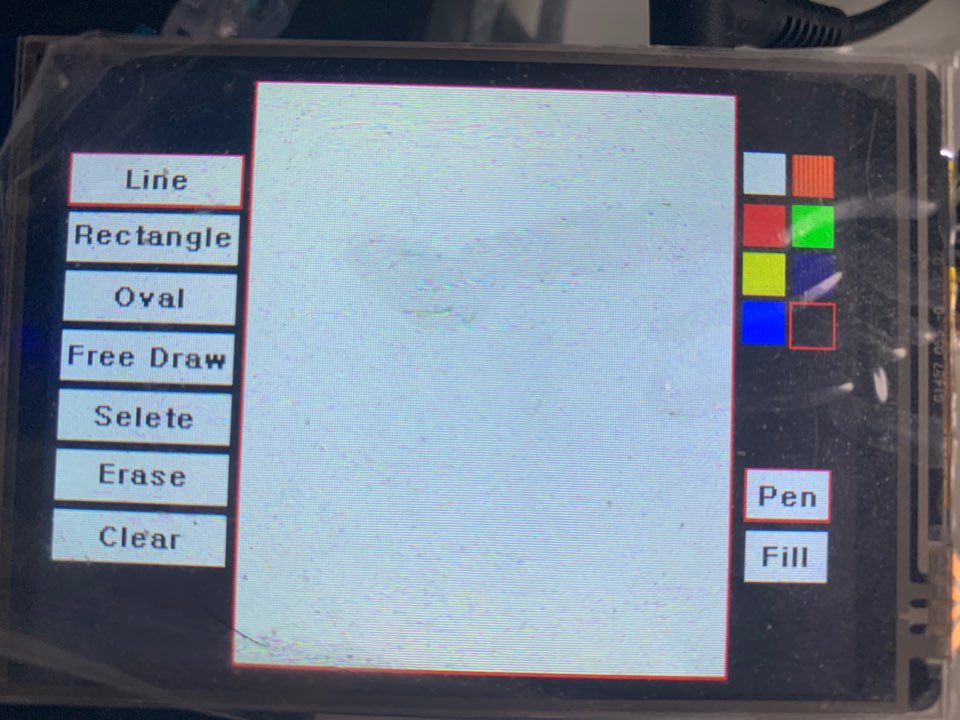
그림을 그리기 위해선 그리기 구역을 터치해 그리기 모드로 진입해야 한다.

그리기 모드로 진입했을시에는 위의 버튼과 마찬가지로 그리기 구역 주변에 빨간색 테두리가 생기게 된다.

그리기 모드 진입 전 ▼



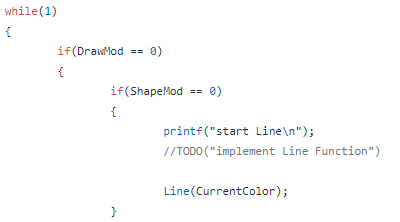
그리기 모드 진입 후 ▼



**3. 그리기**

그리기 모드에 진입했다면 이제 그림을 그릴 차례이다.

그림은 위에서 설정한 그리기 모드에 따라 그림이 그려지게 된다.



위의 코드는 위 동작 중 일부를 캡쳐한 것이다.

위의 코드를 보면 DrawMod가 0이고 ShapeMod가 0이면 Line 함수를 부르도록 되어있다

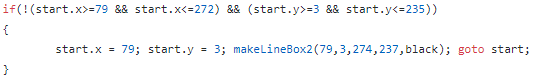
여기서 DrawMod가 0이라는 것은 펜모드에서 Pen이 선택되었다는 것이고

ShapeMod가 0이라는 것은 Line을 그리겠다는 것을 의미한다.

그리고 위에서 선택한 색상은 Line함수의 매개변수로 들어가 해당 색상을 그릴 수 있게 해준다.

**4. 그리기 구역 밖에 터치 시 그리기 모드를 빠져나감**

위의 반복문 안에는 아래의 코드를 넣어놓았다.



그리기 구역 밖을 터치하면 코드의 시작지점으로 돌아간다는 뜻이다.

즉 그리기 구역 밖을 터치하면 그리기 모드가 종료되고 1번부터 다시 코드가 반복이 된다는 것이다.

위의 1~4번을 반복하면서 그림판 프로그램은 동작을 하고 그에 맞게 함수를 불러와 올바른 동작을 하게끔 만들어준다.

그림을 그리는 알고리즘은 아래와 같다.

1. 현재 그린 그림을 tmp배열에 저장

2. Draw배열 출력

3. tmp배열 출력

4. 그리기 종료시 tmp배열의 내용을 Draw배열에 추가

Touch Screen으로의 출력은 mmap을 사용하여 바로 출력할 수 있었다.

하지만 다른 그림을 그리면 기존 그림이 지워지는 문제와 Select를 이용한 그림 옮기기를 위해서

배열을 사용하여 출력을 하도록 바꾸었다. 기존 mmap보다는 속도와 반응성이 떨어지지만 여러 문제점들을 손쉽게 해결할 수 있었다.

위의 알고리즘을 이해하기 쉽게 그림으로 표현해 보았다.

🡺 현재 그리고 있는 화면(tmp)

🡺 기존 그렸던 그림이 저장되어 있는 화면(Draw)

🡺 UI 구성 화면

이런 식으로 레이어를 나누어 표현을 하면 새로 그림을 그려도 새로운 그림이 기존 저장되어 있는 화면을 간섭하지 않고 그림을 그릴 수 있게 된다.

또한 rubber-band를 구현함에 있어서도 전체 화면을 간섭할 필요 없이 현재 그리고 있는 화면에 대한 컨트롤만 해주면 되기 때문에 상당히 편리 해졌다고 생각한다.

그리기가 종료되면 tmp에 있는 내용들을 Draw에 옮겨 줌으로써 변경내용을 반영하도록 해주었다.

**Line 함수**

\* 기능 구현에 사용 된 알고리즘

저희의 LCD는 320 \* 240의 좌표로 이루어져 있습니다. 여기서, 직선을 긋는다면 시작점과 끝 점 사이에서 기울기를 구해서 그 기울기에 맞는 좌표들을 모두 찍어야 합니다. 일반적으로 기울기를 구하는 공식은,

(시작점 = (x1, y1) 끝점 = (x2, y2)



이러한 직선의 방정식을 떠올릴 수 있습니다. 하지만, 실수 연산이 모두 정수형으로 형 변환이 되기 때문에 이러한 식으로 직선을 구현한다면, 기울기가 정수인 경우에만 그려질 것입니다. 이를 해결하기 위해

‘ 브레슨 햄 알고리즘 ‘ 을 착안하였습니다.

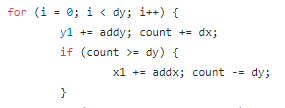


저희는 LCD에 시작점과 끝 점 사이의 최소의 거리들을 모두 점으로 표현하고 싶은 것이 목적입니다. 즉 시각적으로 직선에 가장 가깝게 보이도록 표현해야 합니다. 이를 표현하기 위해서 흔한 표현으로, 위의 사진처럼 직선 렌더링 기법을 사용하기로 결정했습니다.

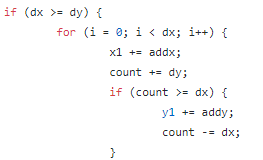
이를 표현하기 위해서는, 기울기가 1보다 작은 경우와, 기울기가 1보다 큰 두 가지 경우로 나뉩니다.



기울기가 1보다 클 때는, x좌표가 증가하는 기분 보단, y좌표가 증가하는 기준으로 점이 찍히게 됩니다. 이를 코드에 적용하면,

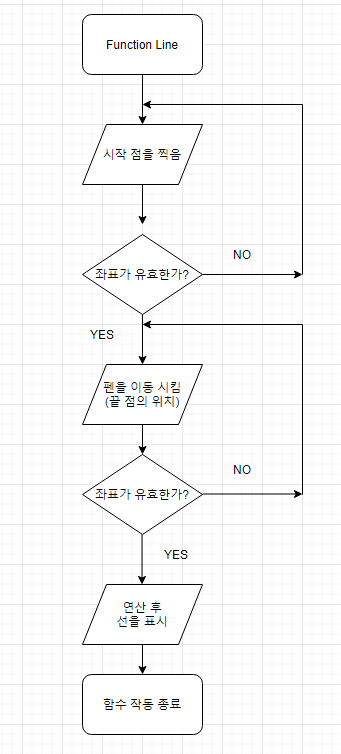


이러한 연산을 하게 됩니다, (y1 = 시작점의 y좌표, dx, dy =증가량, count = 증가량에 따라 x와 y 중 어떤 것을 더할 지 결정하는 변수,) + addx, addy 의 값은 1이며, 시작점 좌표가 끝 점 좌표보다 좌표 값이 낮은 위치에 있을 경우에 부호가 역전됩니다. Y가 증가함에 따라, count 변수에 dx를 계속 더합니다. 만약 count의 누적된 값이 y 증가량을 넘어선다면, x를 증가시키고 count에 dy만큼 뺍니다. 기울기가 1보다 작은 경우는,



위와 같은 연산을 합니다. 이러한 연산을 이용해, UI에 Line 함수가 정상적으로 작동하도록, 함수의 틀은 아래와 같이 구성하였습니다.

\* 함수 구조



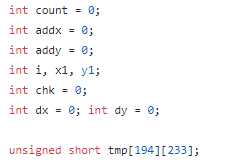
해당 flow chart를 기반으로, 그리는 공간에 유효한 좌표에 시작 점을 찍을 때 까지 반복을 하는 while문과,

그리는 공간에 유효한 좌표에 끝 점을 찍을 때 까지 반복을 하는 while문으로, 총 두개의 반복문을 가진 구조를 구현하기로 하였습니다.

\* 함수 코드 설계



우선 현재 색을 매개변수로 받아서 작동을 하는 Line 함수의 선언부 입니다.



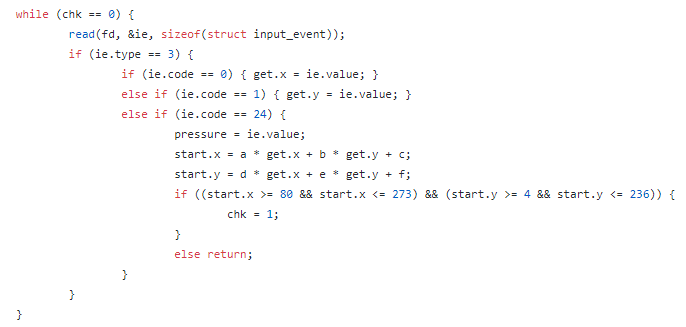
함수 작동을 위해 사용되는 지역변수들입니다.

Count : 앞서 설명 드린 증가량에 따라 x와 y 중 어느 것을 증가 시킬 지 판별하기 위한 조건변수

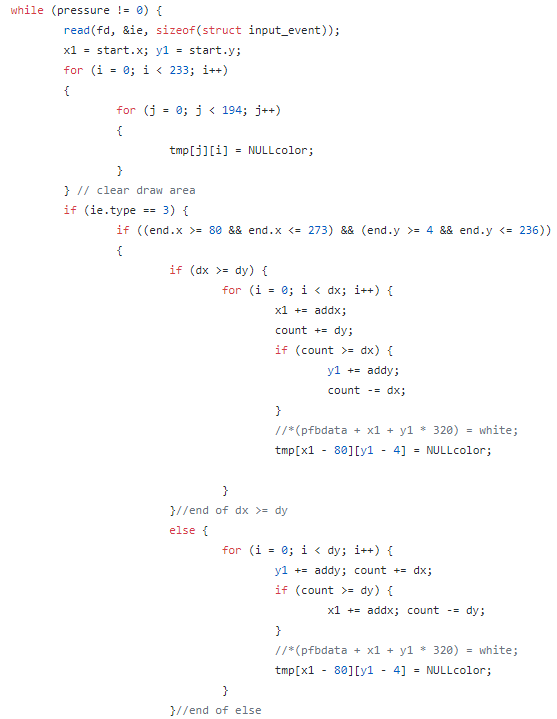
Addx, addy : count에 맞추어 x나 y를 증가시킬 때 더하는 변수.

Chk : 시작 점이 찍혔는지, 안 찍혔는지 판별하기 위한 조건변수

Tmp 배열 : 그리는 공간의 좌표와 해당 배열 값들을 매핑 시키기 위해 선언합니다.

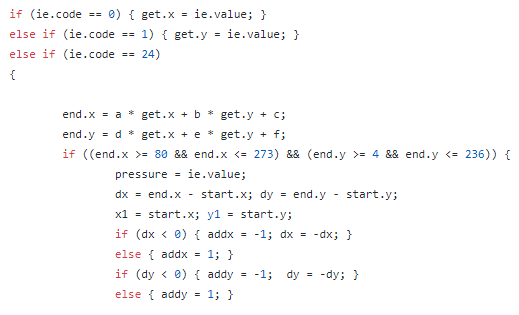


함수의 시작부분에서, chk 값을 조건으로 걸어서 첫 시작 좌표를 찍을 때 까지 반복합니다. 또, 그 좌표가 DrawArea (80,4) ~ (273,236) 에 들어갈 경우에만, chk 값을 바꿔서 해당 반복문을 빠져나가게 합니다.

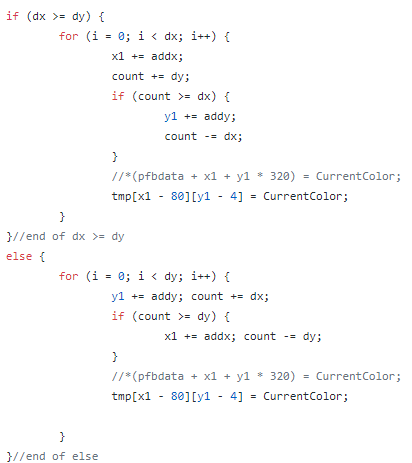


시작 점이 조건에 맞게 찍혔다면, 그 다음에는 터치 펜을 스크린에서 뗄 때 까지(pressure가 0이 될 때 까지) 반복을 하는 반복문에 들어가게 됩니다. 첫 점을 찍은 이후, 스크린에 pressure가 계속 들어오고, 좌표가 계속 읽히기 때문에 과제 direction의 rubber-band를 이에 맞춰 구현 할 수 있다고 생각하였습니다.

끝 점 처리의 시작부분에서는, 먼저 스크린의 draw area와 매핑되는 tmp 배열의 원소들을 초기화합니다. Rubber – band 방식을 하게 되면, 펜을 움직일 때 마다 그려져 있던 선을 지우고, 다시 그리는 작업을 반복해야 하기 때문입니다.



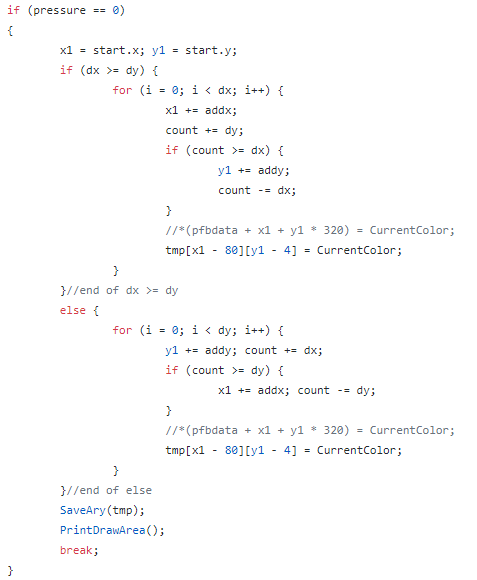
지우는 과정을 완료한 이후, 펜에 입력되는 정보들을 가져와서 변수에 대입합니다. 이 때, 예외 처리가 들어가게 되는데, 만약 끝 점의 좌표 값이 시작 점의 좌표보다 값이 작다면, 증가량은 음수를 띄게 될 겁니다. 그렇다면, 이에 따라 계산을 다르게 해 주어야 하는데, 저는 계산을 다르게 하지 않고, 증가량과 addx, addy 자체의 부호를 역전시켜 동일한 계산을 해도 문제가 없도록 처리하였습니다.



변수에 값들을 선언 하고, 부호에 대한 예외 처리 이후, 선을 그리기 위해 계산을 합니다. CurrentColor 라는 색상 정보를 매개변수로 받았기 때문에, 해당하는 x좌표와 y좌표를 tmp 배열의 인덱스에 매핑시켜서, 해당 좌표는 CurrentColor 라는 정보를 넣게 됩니다. 그 이후,



해당 함수로 사용자가 보는 Draw area에 시각적으로 표현을 해줍니다. 그리고 이 과정은 펜을 뗄 때 까지(Pressure 값이 0이 될 때 까지) 지우고 그리기를 반복합니다. -> rubber band 구현



펜을 뗐다면, 마지막으로 읽은 좌표가 남아있으니 그 좌표를 이용해 다시 계산을 해서, tmp 배열에 저장 한 다음, 출력을 하게 됩니다. 이를 수행 하고 Line 함수는 종료됩니다.

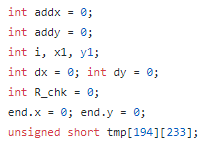
**Rectangle 함수**

Rectangle 함수의 구조는 위의 Line 함수의 구조와 유사합니다.

사용 한 알고리즘은, 임베디드 수업 touchscreen 에서 사용한 makeLineBox를 참고하여, 시작점과 끝 점 사이의 거리를 이용해 x 축과 y축 서로 두 번씩 직선을 긋는 알고리즘을 사용했습니다.



Rentangle 함수의 선언, CurruntColor를 매개변수로 받습니다.

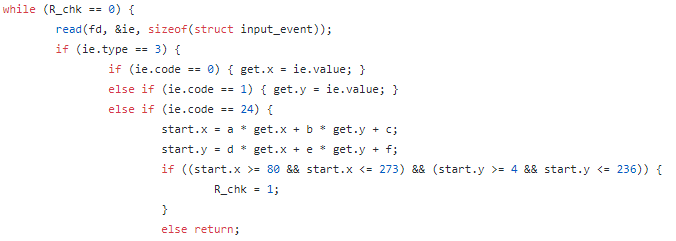


Addx, addy : 시작점과 끝 점의 차이가 음수인지, 양수인지에 따라 부호가 갈립니다.

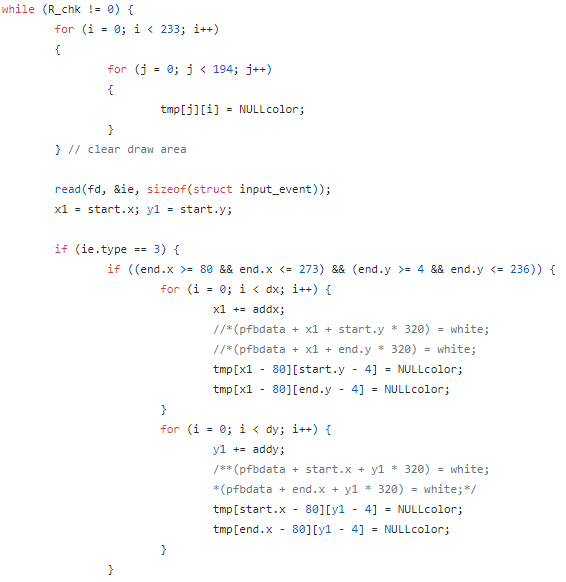
Dx, dy : 시작점과 끝 점의 차이를 나타내는 값입니다. 음수일 경우, addx와 addy가 음수가 됩니다.

R\_chk : 앞서 설명드린 Line 함수에서의 Chk 변수와 같은 역할입니다.

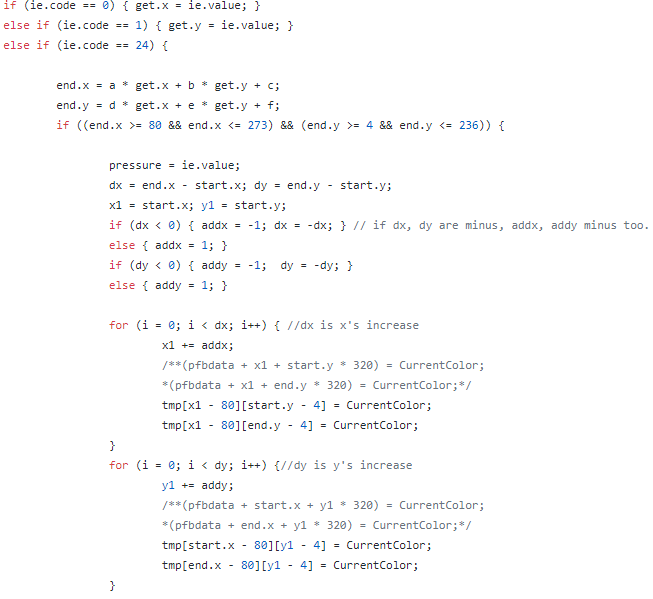
Tmp : 위와 동일합니다.



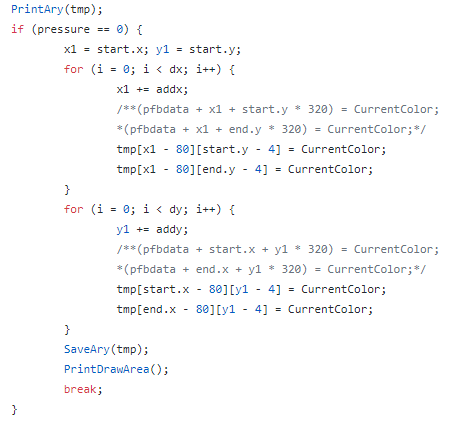
우선 시작점을 찍을 때 까지, R\_chk 변수를 조건으로 걸고 Draw Area에 만족하는 점을 찍을 때 까지 기다립니다. 찍게 된다면, 반복문을 빠져나갑니다.



R\_chk가 0이 아닐 경우, 끝 점을 찍는 반복문으로 들어오고, 우선 모든 공간을 tmp 배열에 nullcolor 값으로 저장합니다.(끝 점을 계속 이동시키면서 rubber-band를 구현하기 위해, 그리기 이전 현재 그림판의 모든 픽셀 값을 default 값으로 생각하고 시작) 그 후, 펜이 찍고 있는 점의 좌표를 읽기 위해 이벤트 값을 불러옵니다. 끝 점의 좌표를 읽기 전에, rubber-band를 구현하기 위해 ie가 발생할 때 마다 해당 좌표로 그려진 사각형을 지웁니다.



x축과 y축에 대해 변화량을 계산하고, 해당 값의 부호를 판별하여 증가값의 부호를 결정합니다. 그리고 변화량과 증가값을 사용해 시작점부터 끝점 까지 각 축마다 2번 반복하여 사각형을 그립니다.



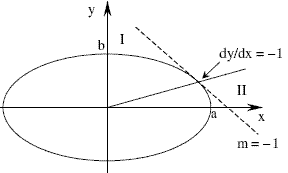
펜을 뗀 이후, 완전히 해당 사각형을 배열에 저장하여 Draw area와 매핑하는 코드입니다, 이 동작을 수행하면, Rectangle 함수는 종료합니다.

**Oval 함수**

\* 기능 구현에 사용 된 알고리즘

타원 알고리즘 또한 ‘ 브레슨 햄 알고리즘 ‘을 착안하였습니다. ‘ 브레슨 햄 알고리즘 ‘을 이용하여 타원을 그릴 때 현재 좌표 기준으로 계산을 하여 점으로 표현하면 원하는 결과가 안나와 타원의 중심을 (0, 0)으로 가정하고 타원위의 점들의 위치를 계산한 다음 옮기기 전의 위치와 (0, 0)의 위치의 차이만 큼 픽셀을 옮겨 LCD상에서 타원이 표현되도록 구현했습니다.

타원을 그릴때에는 위에서 말했듯이 그리려는 타원을 (0,0) 기준으로 옮겨서 픽셀을 옮기는 방식을 이용합니다. 그럴려면 먼저 픽셀의 위치를 계산하는 판별식을 정해야하는데 이 식을 도출하는 과정은 아래와 같습니다.



1. 현재의 픽셀의 좌표를 (x, y)라고 가정할 때 다음 점은 (x+1, y) 혹은 (x+1, y-1) 입니다.
2. 위 두 점의 중간점이 그리려는 타원의 내부인지 외부인지를 판별해서 두 점 중 하나를 선택하면 됩니다.
3. 타원의 중심이 (0,0)이라고 가정했을 시 타원의 방정식은 x^2 / a^2 + y^2 / b^2 = 1입니다. 이것을 F(x, y)로 정리하면 F(x, y) = b^2 \* x^2 + a^2 \* y^2 – a^2 \* b^2입니다.

여기서 저희가 사용할 판별식은 x 매개 독립변수 구간에서 d = F(x+1, y – 1/2)입니다. d가 0보다 작을 경우 (x+1, y)를 선택하고 0보다 클 경우 (x+1, y-1)를 선택하면 됩니다. 선택이 이뤄진 후에 판별식의 값도 갱신하여 다음 좌표에서도 사용하게 만들어주어야 합니다.

* d < 0 였을 경우, 새로운 판별식은 F(x+2, y)가 되고 이것은 원래 d에 b^2 \* (2 \* (x+1) +1)을 더해준 값과 동일합니다.
* d > 0 였을 경우, 새로운 판별식은 F(x+2, y – 3/2)가 되고 이것은 원래 d에 b^2 \* (2 \* (x+1) +1) – a^2 \* (2 \* (y - 1))을 더해준 값과 동일합니다.

그리고 타원의 접선의 기울기가 -1인 부분을 기점으로 그 전까지는 x를 독립매개변수로 사용하고 그 후에는 y를 독립매개변수로 사용합니다. 이것을 판별하기 위해 타원식을 미분하고 기울기를 통해 범위를 설정해주어야 합니다. X 독립매개변수 구간과 초기값 설정 과정은 아래와 같습니다.

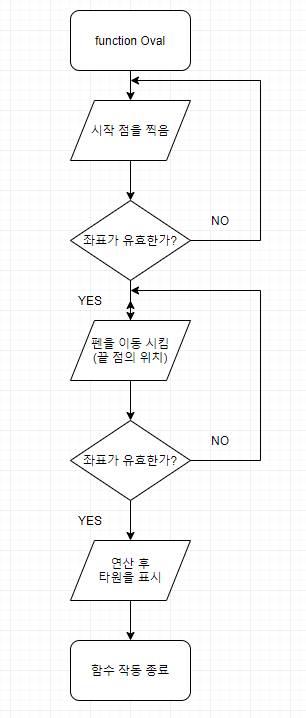
1. b^2 \* x + a^2 \* y \* (dy/dx) (= 기울기) = 0
2. b^2 \* x = a^2 \* y \* (dy/dx) → a^2 \* y \* |dy/dx| <= a^2 \* y, -1 <= dy/dx <= 0
3. b^2 \* x <= a^2 \* y
4. 위 조건이 만족하는 동안 x를 독립매개변수로 사용하고 아닐 경우 y독립매개변수 구간으로 넘어갑니다.
5. x 독립매개변수 구간에서의 초기값은 x=0, y = b(타원의 단축의 +좌표), d = F(0, b – 1/2) = (4 \* b^2 + a^2 \* (1 – 4 \* b)) / 4 입니다.

y 독립매개변수 구간은 x와 y의 역할을 맞바꾸어 현재 픽셀의 좌표가 (x, y)라 가정하면 다음 점의 좌표는 (x, y+1), (x-1, y+1)이 두 점 중 하나입니다. 두 점의 중간점은 (x - 1/2, y + 1)이므로 d를 이 중간점의 좌표로 다시 계산(= F(x – 1/2, y + 1))해주면 됩니다. 그래서 판별식 갱신과 초기값 설정 과정은 다음과 같습니다.

* d < 0일 경우 새로운 판별식은 F(x – 1/2, y+2)가 되고 이것은 원래 d에 a^2 \* (2 \* (y + 1) +1)을 해준 값과 동일합니다.
* d > 0일 경우 새로운 판별식은 F(x – 3/2, y + 2)가 되고 이것은 원래 d에 (-2) \* b^2 \* (x-1) + a^2 \* (2 \* (y+1) +1)을 더해준 값과 동일합니다.
* 초기값의 경우 x와 y의 역할이 서로 맞바뀌므로 x = a(타원의 장축의 +좌표), y =0, d = F(a-1/2, 0) = (4 \* a^2 + b^2 \* (1 – 4 \* a)) / 4 입니다.

위의 계산 결과를 코드에 대입하면 타원의 중심이 (0, 0)에 있다고 가정할 때 x, y이 변화하면서 픽셀이 x >0, y > 0 구간의 좌표에만 새로 점을 찍는데 이 문제는 타원의 대칭성을 이용하여 (x, y), (-x, y), (x, -y), (-x, -y)에 찍으면 해결됩니다.

\* 함수 구조



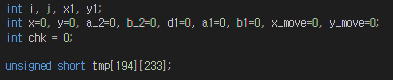
해당 flow chart를 기반으로 그리는 공간에 유효한 좌표에 시작점을 찍을 때까지 반복을 하는 while문과 그리는 공간에 유효한 좌표에 끝 점을 찍을 때까지 반복을 하는 while문으로 총 두개의 반복문을 가진 구조를 구현하기로 하였습니다.

여기서 연산과정에서 두 번의 while문 즉, x 독립 매개변수 구간, y 독립 매개변수 구간이 연산 과정에 포함되어 있지만 flow chart에서는 포함시키지 않았습니다.

\* 함수 코드 설계



우선 현재 색을 color 매개변수로 받아서 작동을 하는 Oval 함수의 선언부입니다.



함수 작동을 위한 지역 변수들입니다.

x1, y1 : 시작 지점 (start.x, start.y)를 저장하는 변수

x, y : 현재 계산중인 x, y의 좌표 값을 저장하는 변수

a1, b1 : 각각 현재 그리는 타원의 장축, 단축의 좌표를 저장하는 변수

a\_2, b\_2 : a1, b1의 제곱을 각각 저장하는 변수

d1 : 판별식이 저장될 변수

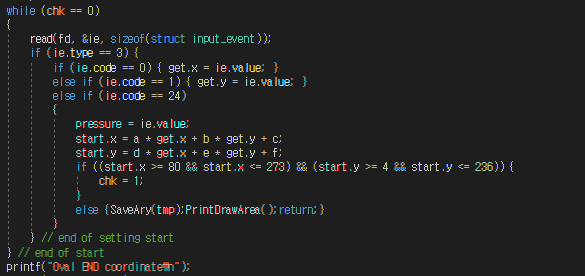
x\_move, y\_move : 현재 그리는 타원을 현재 좌표에서 계산하면 제대로 계산이 안되기 때문에 타원의 중심

을 (0, 0)으로 이동시켜 계산하기 위해 타원이 (0, 0)을 기준으로 x축, y축으로 이동한 값을

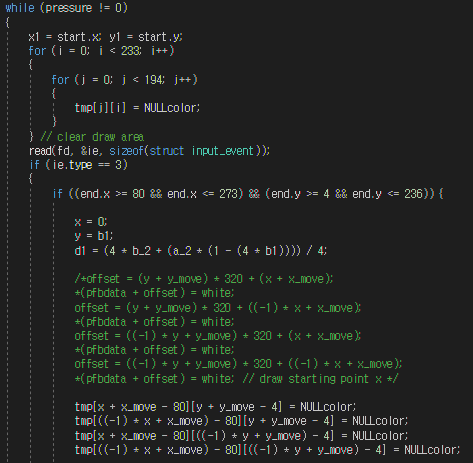
저장하는 변수

chk : 시작 점이 찍혔는지, 안 찍혔는지 판별하기 위한 조건변수

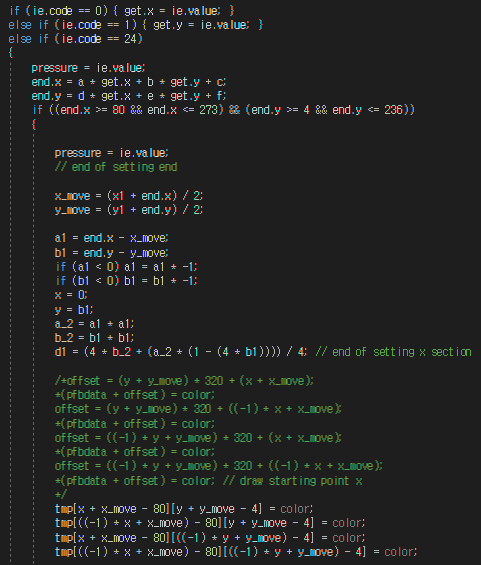
tmp 배열 : 그리는 공간의 좌표와 해당 배열 값들을 매핑 시키기 위해 선언한 배열입니다.



함수의 시작부분에서 chk 값을 조건으로 걸어서 첫 시작 좌표를 찍을 때까지 반복합니다. 그 좌표 DrawArea (80,4) ~ (276, 236)에 들어갈 경우에만 chk 값을 바꿔서 해당 반복문을 빠져나가게 합니다.

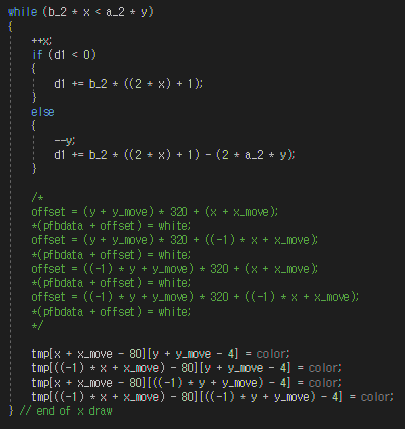


시작점이 조건에 맞게 찍혔다면, 그 다음의 과정은 Line, Rectangle과 유사하게 구현했습니다. 다른 점은 타원 알고리즘을 이용해 그려져 있던 타원을 그리는 데 타원 알고리즘은 아래에서 설명하겠습니다.



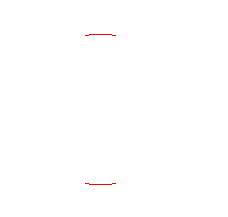
지우는 과정을 완료하고 터치로 입력되는 정보들을 가져와서 변수에 대입합니다. x\_move, y\_move에는 x,y가 이동한 값, 즉 start.x와 end.x, start.y와 end.y의 평균값을 저장합니다. 그리고 a1, b1에는 각각 end의 x,y값을 move의 값만큼 빼주게 되는데 이 값이 음수 일 경우 -1을 곱하여 부호를 역전시켜 계산에 문제가 없도록 처리합니다.

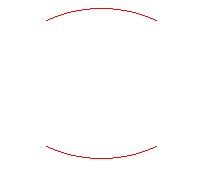
그 후, x 독립매개변수 구간 계산을 위한 초기값 설정을 해줍니다. x는 0, y는 b1, a\_2, b\_2는 각각 a1,b2를 제곱한 값, 참고 알고리즘에서 계산한 초기 판별식 F(0, b – 1/2)을 대입시켜 (4 \* b\_2 + (a\_2 \* (1 - (4 \* b1)))) / 4 을 대입시킵니다. 그리고 이 변수들을 가지고 초기 점들을 찍어주는데 계산한 결과가 타원의 중심이 (0, 0)일 경우의 결과이므로 계산된 좌표(= (x, y) , (-x, y) , (x, -y) , (-x, -y))들을 x\_move, y-move 만큼 이동시키고 end의 x, y의 최솟값 80, 4를 감소시켜서 tmp 배열 인덱스에 매핑시킨 후, color의 정보를 넣습니다.

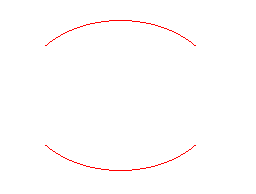


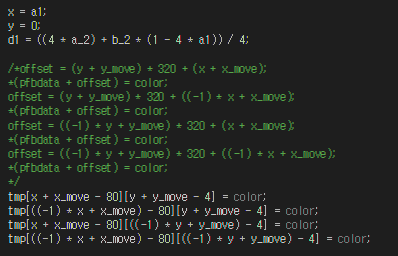
위의 코드는 x 독립 매개변수 구간에서 판별식을 이용한 점 선택 코드입니다. 먼저 b\_2 \* x가 a\_2 \* y보다 작을 때까지만 반복하게 합니다. 그리고 우선 x를 1증가시켜서 다음 x좌표를 가리키게 만들고 판별식이 0보다 작으면 판별식을 갱신하고 0보다 크거나 작을 경우에는 y를 1 감소시키고 판별식을 갱신합니다. 그 후, 초기 점 설정때와 같은 과정으로 tmp 배열 인덱스에 매핑시킵니다. 판별식에 더하는 식의 경우 위 알고리즘에서 설명한 증가 값은 b^2 \* (2 \* (x + 1) +1)이지만 위에서 이미 x를 1증가시켰으므로 x의 값을 대입시키면 원래 목적과 맞게 판별식이 갱신이 됩니다.

이 과정을 따를 경우 아래 그림과 같이 좌표가 설정되게 됩니다.

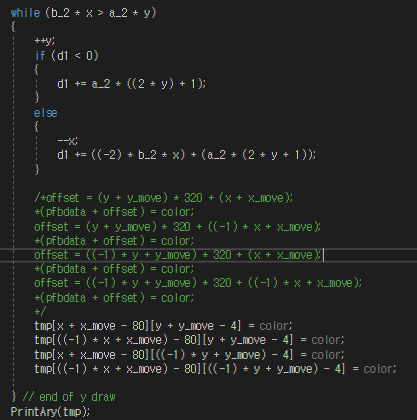






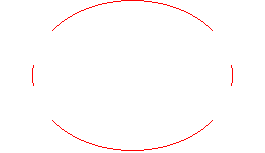


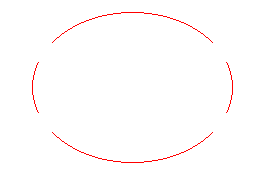
이제 다음은 y 독립매개 변수일때의 계산 과정입니다. 초기 값 설정을 x, y의 역할을 바꾸어 설정해주고 판별식의 값은 F(a-1/2, 0)의 값 ((4 \* a\_2) + b\_2 \* (1 - 4 \* a1)) / 4 을 대입시킵니다. 그 후, x 독립 매개변수 일 때와 동일하게 초기 좌표를 tmp 배열 인덱스에 color의 값으로 추가합니다.

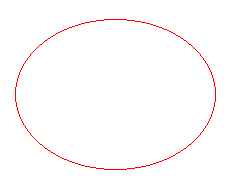


y 독립 매개변수 구간의 반복문의 조건은 x 독립 매개변수일 때와 반대로 b\_2 \* x 가 a\_2 \* y보다 클 때까지 반복하도록 정의해줍니다. 반복문 안에서는 먼저 y를 1 증가시키고 판별식이 0보다 작을 경우에는 판별식만 갱신하고 0보다 클 때에는 x를 1 감소시키고 판별식을 갱신해줍니다. 그리고 판별식에 더해지는 식의 경우 x 독립 매개변수 구간 에서와 같이 y가 이미 1 증가했으므로 y를 대입시켜 주면 원래 목적과 맞게 판별식이 갱신됩니다.

이 과정을 따를 경우 아래 그림과 같이 좌표가 설정되게 됩니다.

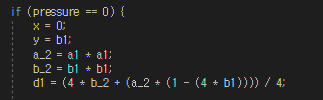




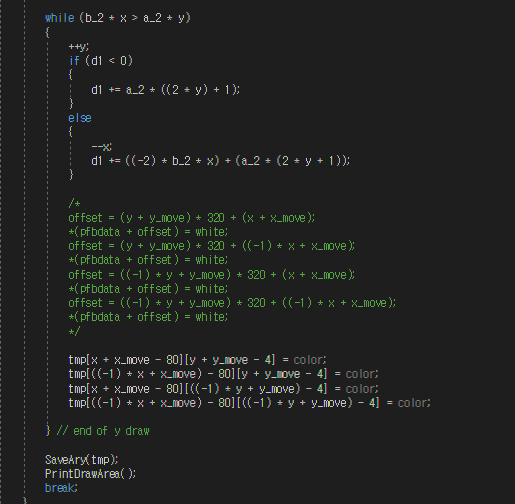


위 코드를 실행후 PrintAry() 함수로 Draw area에 시각적으로 표현해 줍니다. 이 과정은 Line, Rectangle과 같이 펜을 뗄 때까지 지우고 그리기를 반복합니다.

위와 같이 x 독립 매개변수 구간, y 독립 매개변수 구간에서 결정된 좌표들을 결정하는 코드를 그려져 있던 도형을 지울 때와 저장할 때에도 같은 과정을 거쳐서 구현해주면 됩니다.



펜을 뗐다면 마지막 end의 x, y 좌표의 값을 이용해 다시 a1, b1의 값을 구하고 x, y, a\_2, b\_2, d1의 값을 x 독립 매개변수 구간에서와 같은 초기값으로 다시 설정해줍니다.



마지막 좌표를 이용해 x 독립 매개변수 구간, y 독립 매개변수 구간을 다시 계산해서 tmp 배열에 저장한 다음 출력하고 Oval 함수를 종료시킵니다.

**Free Draw**

Free Draw의 기본적인 알고리즘은 아래와 같다.

1. 터치가 이루어짐

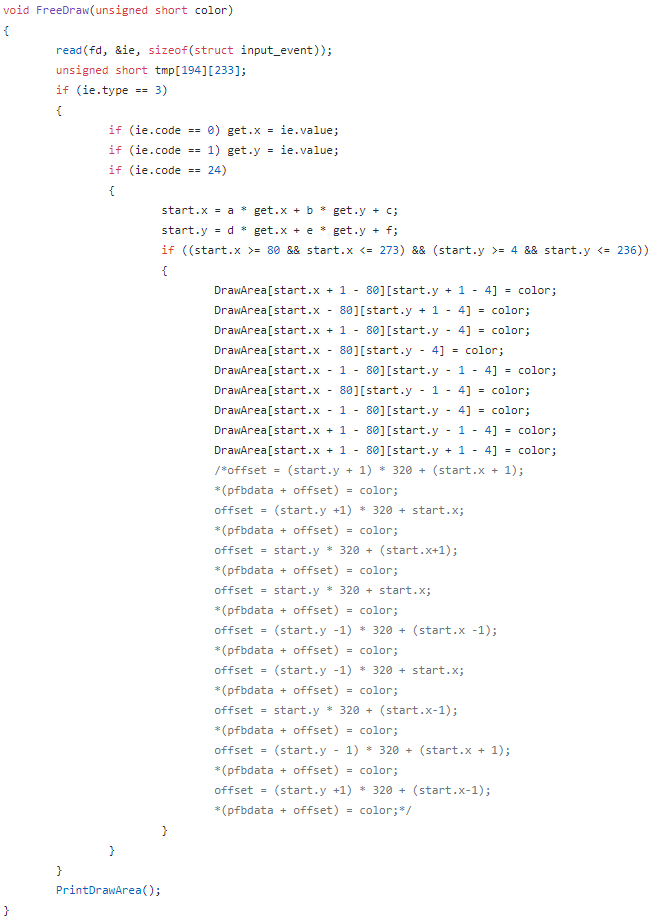
2. 해당 좌표를 기준으로 3\*3만큼의 크기만큼 배열에 저장

3. 배열의 출력

Free Draw의 구성은 간단하다.

현재 터치가 되고 있는 좌표를 읽은 후 출력을 해주면 된다.

코드는 아래와 같다.



기존 출력 알고리즘은 tmp에 저장을 해야 했지만 FreeDraw는 출력내용이 바로 화면에 반영이 되어도 상관없기 때문에 tmp에 저장하는 것이 아닌 DrawArea에 곧바로 저장하도록 만들어 주었다.

**Select**

Select함수의 알고리즘은 아래와 같다.

1. 시작좌표 입력받음

2. 종료좌표 입력받음

3. 시작좌표와 종료좌표를 바탕으로 사각형을 그림

4. 시작좌표 입력받음

5. 종료좌표 입력받음

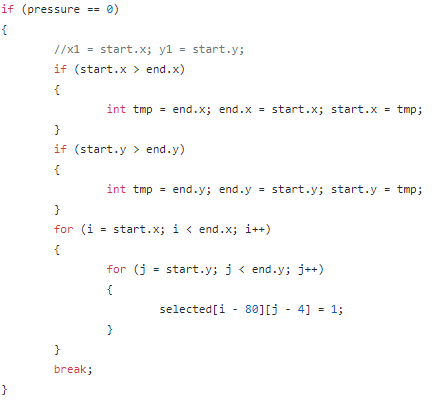
6. 시작좌표와 종료좌표간 차이만큼 사각형 안의 그림을 옮김

시작좌표와 종료좌표를 입력받는 것은 다른 코드와 동일하다. 그렇기 때문에 3번과 6번을 위주로 설명하도록 하겠다.

3. 시작좌표와 종료좌표를 바탕으로 사각형을 그림

시작좌표와 종료좌표를 바탕으로 사각형을 그리는 것은 Ractangle함수와 동일하다.

하지만 Select에서는 하나의 과정이 더 추가된다.



바로 위의 코드가 Rectangle의 함수와 다른 부분이다.

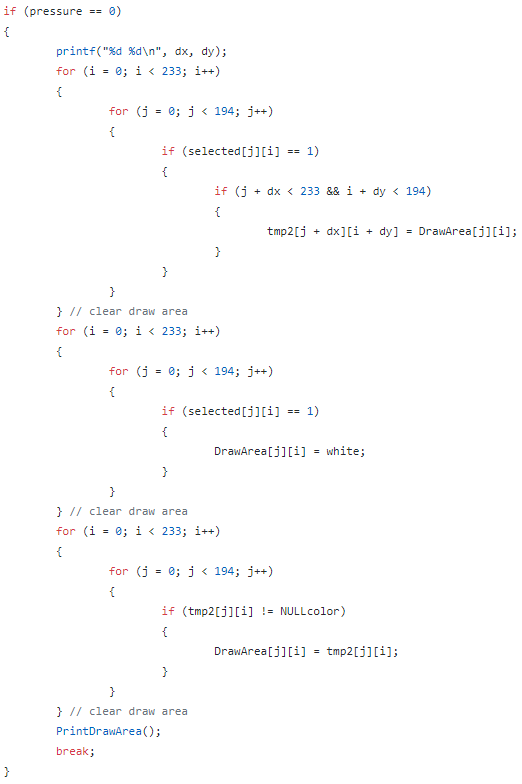
select 함수에는 selected배열이 있다. 이 배열의 역할은 선택된 영역에 대해 flag를 설정해주는 역할을 한다.

즉 선택된 영역 안의 플래그를 모두 1로 설정을 해준다는 이야기이다.

이런 식으로 설정해 놓은 것들은 뒤이어 이동을 시키는 부분에서 사용이 된다.

다시 시작좌표와 종료좌표를 입력 받고 선택된 영역을 이동을 시킨다.

이동시키는 부분의 코드이다.



위의 코드의 주된 내용은 flag로 설정해 두었던 selected배열을 바탕으로 1로 설정되어 있는 부분에 대해 DrawArea의 값을 tmp2에 저장을 한다.

이때 tmp2에 저장하는 인덱스는 앞서 입력받았던 시작좌표와 종료좌표의 차이만큼 변동시킨다.

그렇게 되면 tmp2에는 값들이 이동된 것처럼 저장이 된다.

그리고 나서 Selected배열이 1로 되어있는 곳의 DrawArea부분을 하얀색으로 바꿔주고

DrawArea에 tmp2의 값을 저장해주면 DrawArea에는 선택된 부분이 이동되어 저장이 되고 함수는 종료가 된다.

**Erase**

**Clear**

**Fill**

**기타 등등**

**결론**