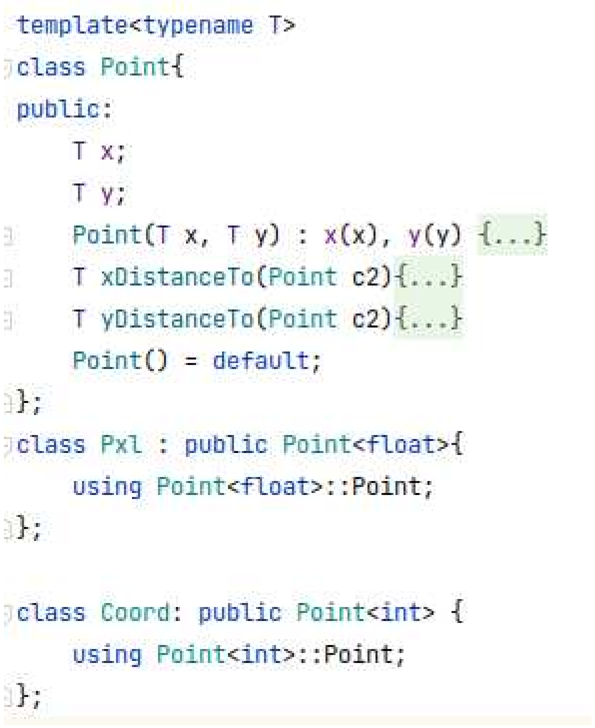
Classes

Point.h



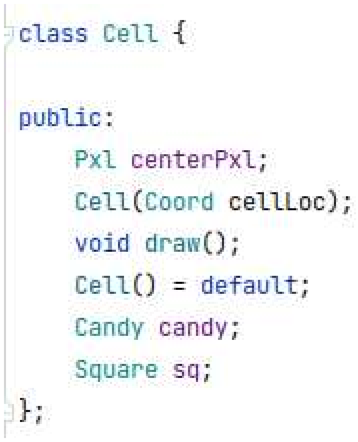
```
template<typename T>
class Point{
public:
    T x;
    T y;
    Point(T x, T y) : x(x), y(y) {...}
    T xDistanceTo(Point c2){...}
    T yDistanceTo(Point c2){...}
    Point() = default;
};
class Pxl : public Point<float>{
    using Point<float>::Point;
};

class Coord: public Point<int> {
    using Point<int>::Point;
};
```

The Point class is a class template, that is a wrapper for a pair of numerical values , (x,y).

The Pxl class, is specifically to refer to a Point of floating points that refer to a pixel on the window.

The Coord class, is specifically to refer to a Point of ints that refer to a coordinate on the board.

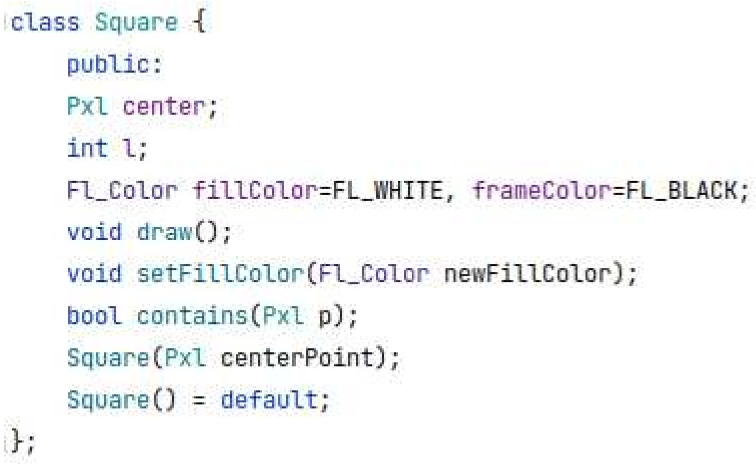
Cell.h



```
class Cell {

public:
    Pxl centerPxl;
    Cell(Coord cellLoc);
    void draw();
    Cell() = default;
    Candy candy;
    Square sq;
};
```

A cell contains a candy and a square. The containing square, is simply a white square that is centered at a specific pixel which is calculated given a Coord. Drawing a cell, simply draws its candy and square.

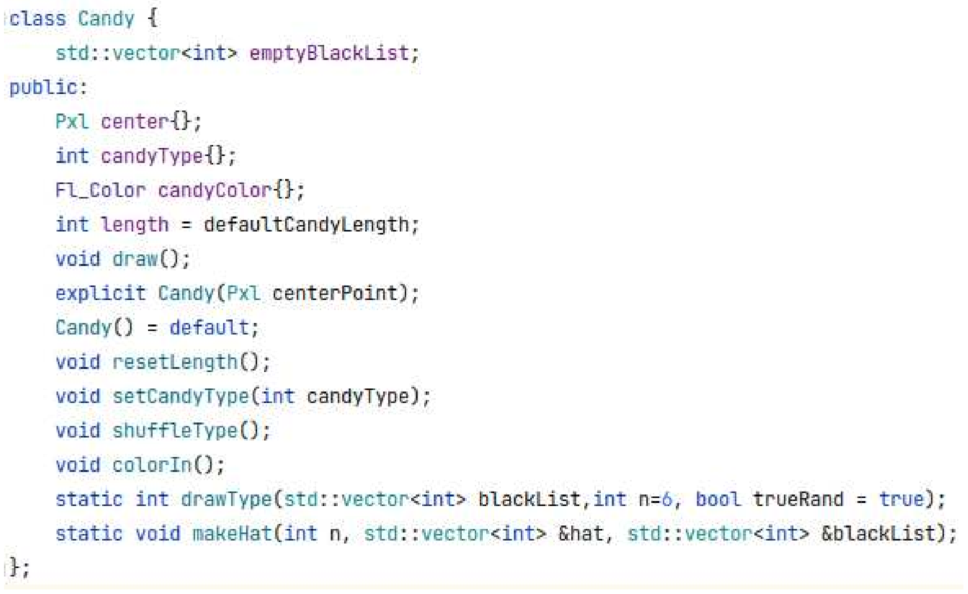
Square.h



```
class Square {
    public:
    Pxl center;
    int l;
    Fl_Color fillColor=FL_WHITE, frameColor=FL_BLACK;
    void draw();
    void setFillColor(Fl_Color newFillColor);
    bool contains(Pxl p);
    Square(Pxl centerPoint);
    Square() = default;
};
```

The square class is used to draw a square, with a specific location and color. It is the basis of identifying the corresponding cell at which the user clicked. This process of identification is done in Game.

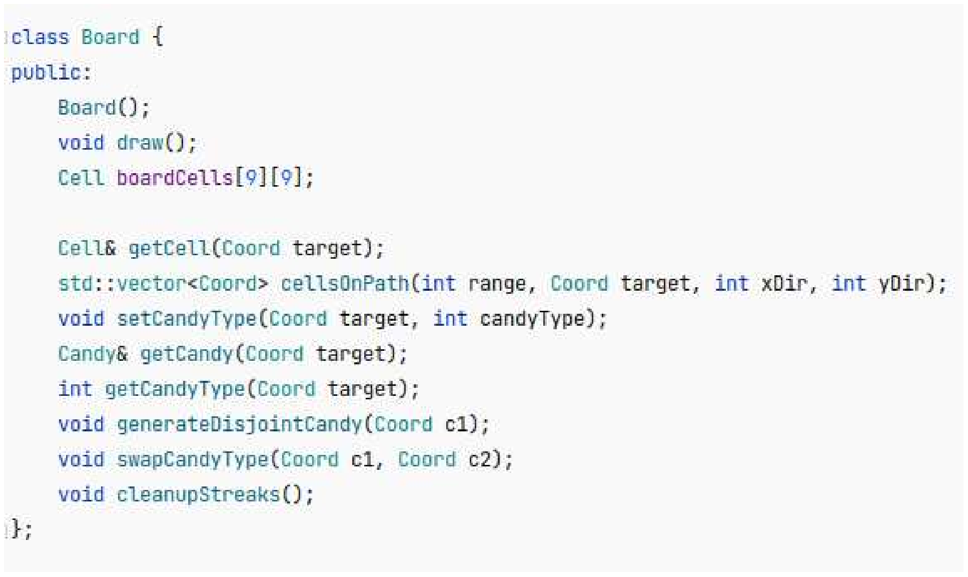
Candy.h



```
class Candy {
    std::vector<int> emptyBlackList;
public:
    Pxl center{};
    int candyType{};
    Fl_Color candyColor{};
    int length = defaultCandyLength;
    void draw();
    explicit Candy(Pxl centerPoint);
    Candy() = default;
    void resetLength();
    void setCandyType(int candyType);
    void shuffleType();
    void colorIn();
    static int drawType(std::vector<int> blackList,int n=6, bool trueRand = true);
    static void makeHat(int n, std::vector<int> &hat, std::vector<int> &blackList);
};
```

This class represents the candy. A candy is part of a cell. Each candy has a type, which is a numerical value between 0-6. Each type has a 1:1 relation with a color. Type 0 represents a blank candy. Here the color is the same as the cell, white. So, each "deletion" doesn't really delete the candy, but rather just makes it white.

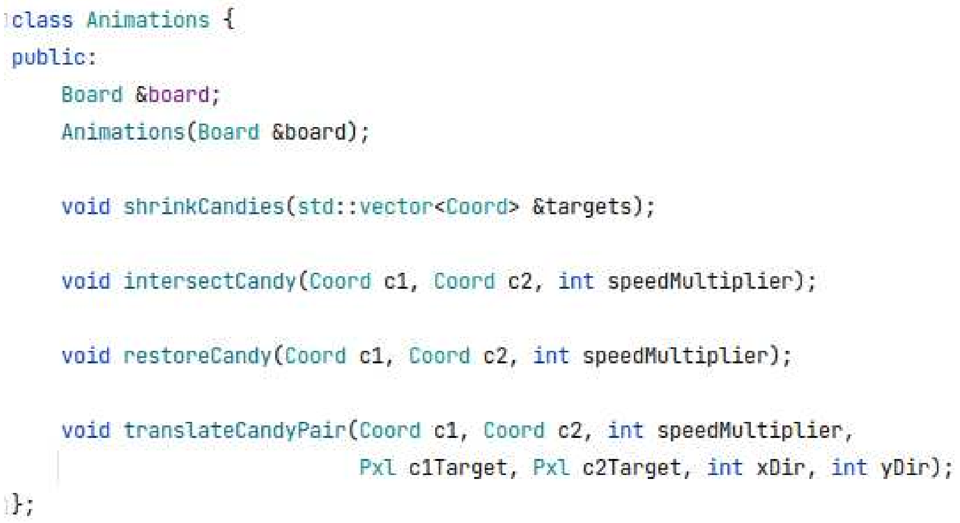
Board.h



```
class Board {
public:
    Board();
    void draw();
    Cell boardCells[9][9];

    Cell& getCell(Coord target);
    std::vector<Coord> cellsOnPath(int range, Coord target, int xDir, int yDir);
    void setCandyType(Coord target, int candyType);
    Candy& getCandy(Coord target);
    int getCandyType(Coord target);
    void generateDisjointCandy(Coord c1);
    void swapCandyType(Coord c1, Coord c2);
    void cleanupStreaks();
};
```

The board class holds a 9x9 array of Cells. It represents the internal structure of the game. Each method, depending on a Coord(x,y), accesses cells through boardCells[x][y], and modifies or retrieves the values of the data members of the targeted cell. No cells ever exchange position. If a candy must be "swapped", the only thing swapped is the value of their type.

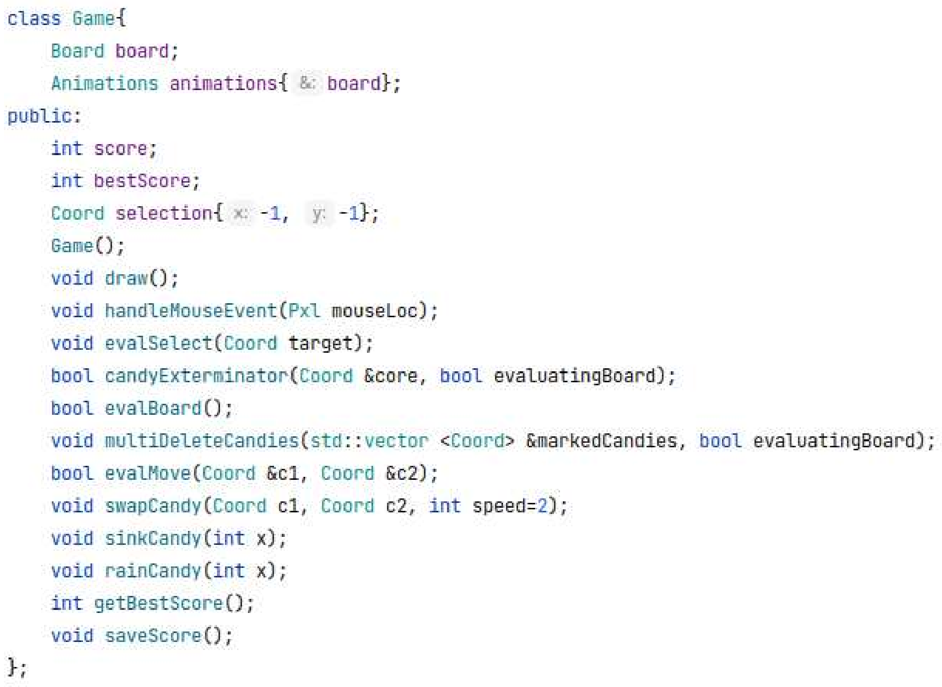
Animation.h



```
class Animations {
public:
    Board &board;
    Animations(Board &board);

    void shrinkCandies(std::vector<Coord> &targets);

    void intersectCandy(Coord c1, Coord c2, int speedMultiplier);

    void restoreCandy(Coord c1, Coord c2, int speedMultiplier);

    void translateCandyPair(Coord c1, Coord c2, int speedMultiplier,
                            Pxl c1Target, Pxl c2Target, int xDir, int yDir);
};
```

This class is responsible for animating the “view” of the game. All it does is modify the length of the candy and its position where it's supposed to be drawn and redraws it. It accesses specific candies via the board.

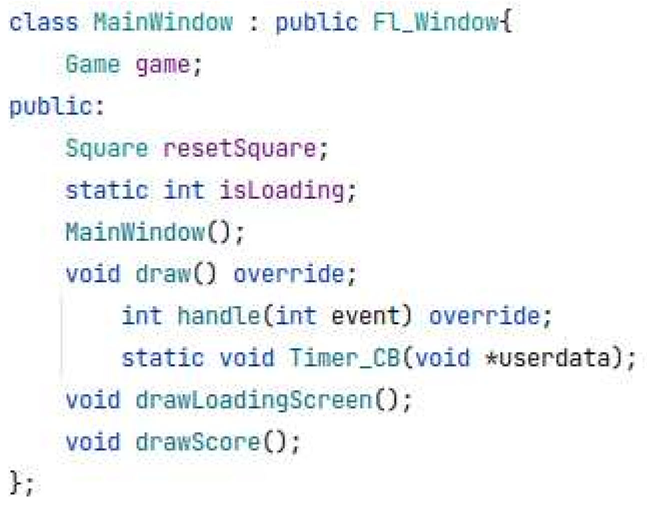
Game.h



```
class Game{
    Board board;
    Animations animations{ &: board};
public:
    int score;
    int bestScore;
    Coord selection{ x: -1,  y: -1};
    Game();
    void draw();
    void handleMouseEvent(Pxl mouseLoc);
    void evalSelect(Coord target);
    bool candyExterminator(Coord &core, bool evaluatingBoard);
    bool evalBoard();
    void multiDeleteCandies(std::vector <Coord> &markedCandies, bool evaluatingBoard);
    bool evalMove(Coord &c1, Coord &c2);
    void swapCandy(Coord c1, Coord c2, int speed=2);
    void sinkCandy(int x);
    void rainCandy(int x);
    int getBestScore();
    void saveScore();
};
```

This class is responsible for processing user input that it receives from the MainWindow class and respectively modifying the state of the board object. After which, the appropriate animations are called.

MainWindow.h



```
class MainWindow : public Fl_Window{
    Game game;
public:
    Square resetSquare;
    static int isLoading;
    MainWindow();
    void draw() override;
        int handle(int event) override;
        static void Timer_CB(void *userdata);
    void drawLoadingScreen();
    void drawScore();
};
```

The MainWindow class calls all the draw functions on a timer and makes a “view” of the board. A click and release will send two signals of the locations of the mouse respectively to the game. Before making a view of the board, it will draw a window that states our names, and the name of the game for a few seconds.

Logic of the Game

Initialization

Upon launching the game, a MainWindow is created, which will cause the Game class to instantiate, which will in turn make a Board object. The board object contains a 9x9 array called boardCells of cells, initialized with a random candy type.

After the generation, a cleanup is done. Each cell is scanned horizontally then vertically. If boardCells detects a streak (consecutive types) of 3 candies, then the 3rd candy is made disjoint (ie: different from its neighbors).

A welcoming screen stating our names and the name of the game “Rhombus Crush” will be drawn first. Then, everything is erased, and the “game” is drawn, which calls boardCells to be drawn, as well as the score counter is drawn, along with the reset button.

Input processing

Upon the click of the user on a cell, MainWindow will transmit the input to Game. Then, Game will find and select the cell, by looking into the square contained within each cell, and checking if the mouse click location is within it. Upon finding the square, the coordinate is calculated and saved as a “selection”. The release will do the same.

If the user has released the mouse on a cell that is NOT 1 away, horizontally or vertically, the selections are reset. Else, the game will cause a swap of the candy types to occur. Then, candyExterminator, is called on both locations of the user selections. candyExterminator basically goes left, right, up and down, and marks candies of the same type, provided they are forming a streak with the core, then deletes the candy. The candies sink, and new ones are generated at the top. If candies are deleted, then the board is evaluated for any new streaks formed by the recent generation or deletion. candyExterminator is called for each cell. This loop is finished when there are no more streaks left to delete. The selection is reset, and it is the user’s turn again.

If candyExterminator finds no streaks, then the candies are swapped back, the selection is reset, and the board has returned to its former state.

Swap

A swap is animated by causing the two candies to move towards each other, forcing an intersection. Once the intersection has taken place, the types are swapped, and the candies are moved back towards their original cell. This makes it seem as if the candy1 has translated to the spot of candy2, without actually moving the candy to be part of another cell,or positioning it where another candy belongs. This avoids a lot of unnecessary calculations and methods to keep track of where each candy has shifted. It will also avoid the need to delete candies (see below)

Deletion and Generation

The “deletion” of a candy doesn’t really take place. What happens is that the candy type is changed to 0, which corresponds to a white candy that blends with the background of the cell. This makes it seem like the candy disappeared.

The animation of the deletion is to make the length of the candies become smaller and smaller, before changing to its type to 0 and recoloring it.

When a candy has been deleted (ie: its type is 0), its column is scanned downwards. The first candy with type 0 is swapped upwards, until it the topmost candy is white. This makes it seem to the user that the candies are sinking. After, the candy is set to a valid type and it is colored in. To the user, it appears (due to illusion) as if the candy has fallen from above (since the rest of them fell). The process repeats until the column has no more deleted candies.

Score

The best score is stored in a file called “score”. For each deleted candy, the score increases by 1. Clicking the red square at the top, sends a signal to Game, to reset the score. The bestScore is thus equal to the current score. Each modification of the bestScore rewrites the file “score”.

MVC

I did start the project considering the MVC conception model. However, each class is interlaced with the view role, because I designed the draw methods to be implemented within each class. The reason being that it makes so much more sense for each object of a class to have its very own draw function. I also find that it is more readable. I did make it so that each class does have a primary role, and this is outlined below:

The Cells, Square and Candy class were primarily responsible for holding a model of the game This model, is represented concretely by Board.boardCells as a 9x9 grid of cells.

Animation and Mainwindow are the view classes. MainWindow is what calls all draw methods. MainWindow paints the image of what the Board should look like. Animation simply changes how things look on the MainWindow upon an action, without changing any values on the model end. The MainWindow receives input and transmits it to the Game class.

The Game and Board class is responsible for controlling the state of the model. The Board class simply changes the values of the data members of boardCells. The Game class is responsible for calling upon animations and changing the state of boardCells, via Board, depending on the input from MainWindow. The two methods could technically be merged, but it made more sense to divide the methods such that the Board class has the methods that change the state of the board, and the Game class has the methods that process user input and then change the state via the Board class.

The remaining class, Point, is just a wrapper for an {x,y} numerical pair.

To completely obey the MVC conception model, I could implement a separate class that is responsible for drawing and a separate class that holds only one data member and nothing else: boardCells (seeing that board is a controller class and thus should not have the model of the board within it). But I did not think an entire class with just one data member warrants being created.