

ITP & PDM FINAL PROJECT  
COVER LETTER

|  |  |  |  |
| --- | --- | --- | --- |
| Student Information | Surname:  Hassan | Given Name:  Hassan | Student ID:  2440105785 |

Course Code: COMP6699

Course Name: Object Oriented Programming

Class: L2BC

Major: Computer Science

Title of Assignment: Candy Crush

Name of Lecturer: Jude Joseph Lamug Martinez

Plagiarism/Cheating

Binus International seriously regards all forms of plagiarism, cheating and collusion as academic offenses which may result in severe penalties, including loss/drop of marks, course/class discontinuity and other possible penalties executed by the university. Please refer to the related course syllabus for further information.

Declaration of Originality

By signing this assignment, I understand, accept and consent to BiNus International terms and policy on plagiarism. Herewith I declare that the work contained in this assignment is my own work and has not been submitted for the use of assessment in another course or class, except where this has been notified and accepted in advance.

Signature of Student: Hassan Mohamed Hassan

**Program Description**:

The first thing to load up is the Menu screen. It has two buttons, "New Game" starts a new game and loads up a level selection scree. In the level screen you can choose between Easy, Normal and Hard all loading up the game file with the selected difficulty . The there is the "Quit" button to exit the program(which can also be done using the Escape(Esc) button).

Once the game starts the candy grid is loaded. The player must match the 5 types of candy in groups of 3, 4 and 5s. The more candy that is matched up the more points you get. The goal is to fill up the score bar before you run out of moves. You replenish moves with purple candy. All the other candies give the same amount of points. Ones you finished the level you can go back to try other levels.

**Game Interface:**

Game Menu:



Difficulty selection:



Game running:

****

**Class Diagram:**

Diagram attached separately, too big.

**Libraries:**

Canvas, Graphics, Canvas: to make the window and the frames

BufferStrategy, BufferedImage, LocatedImage;: to load sprites

Music: to load music and sound

Color: get different colors for text

HashMap: to make a hashmap of music and sound

MouseAdapter, MouseEvent: to get mouse inputs

Random: to generate random numbrs

Rectangle: make the collision boxes for the candy

KeyAdapter: get keyboard inputs  
**Lessons Learned:**

I learned about game logic and how to render it on the screen. When I was making the code for moving the candy around, I had hard time figure out how to update the matrix to have the candy move to its new location but also show this change on the screen for the player. The same thing when I was adding new candy to matrix, I had to find a way to show this change to the player. And it only got worse since I didn’t know witch java libraries I will be using at latter parts of the code. So there where multiple times I had to abandon the solution I made because it interfered with my new code I just added.

One thing that was interesting to play around with was the disconnect between the variables I am working with in code and the graphics I am rendering in the game. Like I can play with a variable in the background without a single change happening to code until I update the graphics with the new data. The same thing with

I also learned a lot about game development, especially about frames and game loops. It was my first time working with a code that is constantly running and for a while, it kept throwing my calculations off. As "a = a+2" is not run one time but once every second. I learned about using class methods and if statements to overcome these and hurdles and keep the game running. All and all it was a really fun and educational project and certainly I want to keep working. I got to add that magic at some point.

**Code Explained:**

The code for the game is mostly straight forward with the most interesting part being the game logic for moving the candy and updating the candy when 3 or more of them line up. So this will be that part I will explain the most.

The first thing I will go through the class that I will not be explain their code, mainly because the program code is long and not all of them are worth it.

* The window class is our basic game screen we will showing the game through that we made using jframe and canvas.
* GameState & ID are enuns that we use to keep track of the different game objects we made with ID and the state the game is with GameState.
* ImageLoader is the same as Audio, a class to load up our sprites and background using BufferedImage.
* Audio class loads up our audio files with the Music Library.
* Keyinput we use to capture the keyboard inputs of the user, our game is mainly mouse based. mainly used it for exit the game quickly during testing and developing with the “Esc” key.

As for the rest of the code I will just go through it and talk about the important bits.

Let us first quicly go over the menu and player class. They are both fairly straight forward class. The menu class renders the all the menu options on the screen. It also checks the game state to know which menu to show. If it is in menu it shows the main menu, level shows the different level options and End shows the end-screen. The End menu also checking if our score is above or equal to goal. If it is we show the victory screen otherwise we show the Game over screen.

…

if(Game.*gameState*==*GameState*.***Menu***) {

g.setFont(fnt);

g.setColor(Color.***blue***);

g.drawString("Candy Crush",350, 100);;

g.setFont(fnt2);

g.setColor(Color.***green***);

g.drawString("New Game", 0, 300);;

g.setColor(Color.***red***);

g.drawString("Exit", 0, 500);;

} else if(Game.*gameState*==*GameState*.***End***) {

g.setFont(fnt);

if(player.getScore()<player.getGoal()) {

g.setColor(Color.***red***);

g.drawString("Game Over",350, 100);;

g.setFont(fnt2);

...

What actually gives our menus and really all other classes functionality is the Mouseinput class. This checks where the player mouse is clicking on that and what the game state is. For example if the game state is on Level and we click on the coordinates where our Easy mode option is then the game is launched on easy mode. The way we do it is that Mouse input class is a subclass of MouseAdapter class. From that class we get the functions mouse pressed, to code for events when the mouse is pressed. Moused released when the mouse is released and mouse dragged. Mouse dragged being the function we use when to move the candy around(will discuss down bellow). So when press on the mouse we get the coordinates of the where we clicked on. Then we check the state of the game and after it is we use mouseOver function to now if our mouse actually clicked on something and we run the corresponding code if it is.

public void mousePressed(MouseEvent e) {

int mx = e.getX();

int my = e.getY();

if(Game.*gameState*==*GameState*.***Game***) {

if(!Game.*Pause*) {

*dragging*= true;

*Imx* =mx/(500/8);

*Imy* = my/(500/8);

*Cmap* = createClonemap();

}

}if(Game.*gameState*==*GameState*.***Menu***) {

if(mouseOver(mx,my,0, 250, 250, 50)) {

game.*gameState*=*GameState*.***Level***;

}

if(mouseOver(mx,my,0, 450, 200, 50)) {

System.*exit*(1);

}

}

if(Game.*gameState*==*GameState*.***Level***) {

//easy

if(mouseOver(mx,my,380, 215, 150, 50)) {

Mapmaker();

//printmap(Game.*map*);<< did this for test ignore

player.setMoves(20);

player.setGoal(80);

player.setScore(0);

Audio.*getMusic*("Menu").stop();

Audio.*getMusic*("Game").play();

Game.*gameState*=*GameState*.***Game***;

for(int y=0;y<8;y++) {

for(int x=0;x<8;x++) {

handler.addObject(new Candy(x,y,*ID*.***Candy***,handler));

}

}

player.setScore(0);

As you can see when we “click” on the easy mode any mode other mode the game is launched. The number of moves and goal are set, score is reset. After that Audio class is run to play the music and stop the other music. Then the MapMaker() function is called to make a new random map for the level(we will get there). Then the game state is changed before the new candy objects are created and put in the handler list.

The handler being a linked list that contains all our game objects.

public class Handler {

LinkedList<GameObject> object = new LinkedList<GameObject>();

it has a render and update function that loop through the list updating and rendering all the objects with in it. And a clear function to empty the list when a new game is starated.

public void update() {

for(int i =0;i<object.size();i++) {

GameObject temp = object.get(i);

temp.update();

}

}

public void render(Graphics g) {

for(int i =0;i<object.size();i++) {

GameObject temp = object.get(i);

temp.render(g);

}

}public void clear() {

this.object.clear();

}

After that that our game is launched. One import thing about the game class is the game loop. This allows us loop through all the objects we have in the game, update them and the render the changes on the screen. For our loop the game makes sure to update 60 times before rendering or drawing all the objects. This makes sure that all the variables are update correctly before anything is show on the screen. Though of course the more time it take to update the less frames we have, but that is not a problem for our game as it is not that big.

this.requestFocus();

double ns = 1000000000 / 60d; << how many updates we want per second

long lastTime = System.*nanoTime*();<< get the time before the start of loop

long timer = System.*currentTimeMillis*();

double delta =0;

while(running) {

long currentTime = System.*nanoTime*();<< get the time

delta += (currentTime - lastTime) / ns; << Calculate the number of update we are going to do

lastTime = currentTime;

while(delta>=1) {

update(); << update the game

delta--;

}

render();<< render/draw the changes on to the screen

if(System.*currentTimeMillis*()- timer >1000) {

timer+=1000;

fps= 0;

ups=0;

}

}

Now let us look at the update() function:

The update function checks if the game is in the Game state if its then it is then it updates our handler, which a linked-list that contains all game objects and the handler loops through that list update them all in the handler update() function. Then the Player is update(will show down bellow as well). Then it checks if the score is the same the goal or if we run out of moves which are game ending conditions. It clears our handler which removes all the elements this way we can add new objects when the game is rerun. Then the game state is changed to End. And if the game is not in Game state then it Menu class updated. The menu class contains all the other states as they all use the same base code and just print stuff on the screen.

private void update() {

ups++;

if(*gameState*==*GameState*.***Game***) {

if(!*Pause*) {

handler.update();

player.update();

checkmap();

if(player.getMoves()==0||player.getScore()==player.getGoal()) {

handler.clear();

*gameState*=*GameState*.***End***;

}

}

}else if(*gameState*!=*GameState*.***Game***) {

menu.update();

}

After the everything is updated the objects are drawn/rendered on to screen. First we set up the buffer strategy. This determines how we order and organise the different canvas we show. The game is using triple buffering which means the are three canvas to show. The one currently being show, the next one in line and the third one is the one being rendered at the moment. After that we check the state of the game. If it is in the game state we render our our objects through the handler and if not we render the menu.

private void render() {

fps++;

BufferStrategy bs = this.getBufferStrategy();

if(bs == null) {

this.createBufferStrategy(3);

return;

} g = bs.getDrawGraphics();

g.setColor(Color.***black***);

g.fillRect(0, 0, ***Width***, ***Height***);

g.drawImage(*background3*,0,0,null);

if(*gameState*==*GameState*.***Game***) {

g.drawImage(*background2*,0,0,null);

handler.render(g);

player.render(g);

if(*Pause*){

g.drawString("Paused",500, 500);

}

}else if(*gameState*!=*GameState*.***Game***) {

menu.render(g);

}

g.dispose();

bs.show(); }

Now our handler has candy objects let us look at the candy class. First thing our candy class is a subclass for our game object class. Game objects being an abstract class that contains getters and setters for our x, y and ID; ID being an enum we use to differentiate between all the different objects within the handler list once we loop through it. Upon instillation we first get the array Map which is an 8 by 8 2d array that represents the placing of our candy on the screen. We set our x and y to the candies index on the Map array times (500/8), which is the distance between the candies on the screen. The variable a, b are the x , y index on the Map array and ab is the unique code for the candy. We also load up our sprites for the candy.

public GameObject(int x, int y, *ID* id) {

this.id = id;

if(this.id==*ID*.***Candy***) {

this.a =x;

this.b=y;

this.ab=Integer.*parseInt*(Integer.*toString*(b)+Integer.*toString*(a));

this.x=x\*(500/8);

this.y=y\*(500/8);

}else {

this.x = x; this.y = y;

…

public Candy(int x,int y, *ID* id,Handler handler) {

super(x, y, id);

this.handler = handler;;

this.type=Game.*map*[this.b][this.a];

ImageLoader loader = new ImageLoader();

*candy1*= loader.loadImage("/candy1.png");

*candy2*= loader.loadImage("/candy2.png");

….

We also create an invisible box within the candy. This box is the hit-box for the candy and we use it to check for collusion with other candy. This is also why we have the variable , ab. You see if loop through our handler check for all the id Candy, our collusion code we always return true because our candy is also within that loop (took me a while to realize out that error). So we also check for the ab code of the candy to make sure that it is not our own. I will shoe the collision code later.

…

public Rectangle getBounds() {

return new Rectangle(x,y,54,54);

...

There is the update function for the candy. This one being one of the most import codes of the program.

This is the part simulates the movement of the candy down to the lower level ones we player matches the candy up and removes them. What it does it checks if dragging is true, dragging being the Boolean value of player dragging and move a candy around. If it false and it candy is not on the lowest row then it checks if the spot directly beneath it empty, an empty spot being represented by a 4 in the Map matrix. If it is then the candy starts going down till reaches that spot. After that the map matrix is updated. With the previous spot of the candy being marked empty and the new spot given it candies old values. After that the y value is corrected to fit perfectly into the space. With all the candies being updated to the new Map array.

public void update() {

if(MouseInput.*dragging*==false) {

this.a=this.getX()/(500/8);

this.b= this.getY()/(500/8);

this.ab=Integer.*parseInt*(Integer.*toString*(b)+Integer.*toString*(a));

}

if(b<7) {

if(Game.*map*[b+1][a]==4&&y!=(b+1)\*(500/8)) {

this.setY(this.getY()+5);

if(y>=(b+1)\*(500/8)) {

Game.*map*[b+1][a]=Game.*map*[b][a];

Game.*map*[b][a]=4;

this.setY((b)\*(500/8));

this.type=Game.*map*[b][a];

And if the candy is at the top most row and is empty and random color candy is generated.

if(this.type==4 && b==0) {

int randm = rand.nextInt(4) + 0;

if(randm!=Game.*map*[b+1][a])Game.*map*[b][a]= randm;

else Game.*map*[b][a]=4;

}

After that the candy is rendered on to the map. With its sprite and x and y coordinates.

public void render(Graphics g) {

if(this.type==4) {

}else {

if(this.type==0) {

g.drawImage(*candy1*,x,y,null);

}

else if(this.type==3) {

g.drawImage(*candy4*,x,y,null);}

..

Now we talked about the candy moves down let us talk about the player can move them. For that we use the mouse pressed, dragged and released.

We first check where the player pressed on the screen. If it is on a candy great! We set dragging to be true and save the location of the candy. Then we create a new array Cmap, a cone of the Map array we or rather the player can play around and change as he likes.

public void mousePressed(MouseEvent e) {

int mx = e.getX();

int my = e.getY();

if(Game.*gameState*==*GameState*.***Game***) {

if(!Game.*Pause*) {

*dragging*= true;

*Imx* =mx/(500/8);

*Imy* = my/(500/8);

*Cmap* = createClonemap();

}

…

private int[][] createClonemap() {

for(int y=0;y<8;y++) {

for(int x=0;x<8;x++) {

*Cmap*[y][x]=Game.*map*[y][x];

}

}

return *Cmap*;

}

...

Now that dragging is set to true. We go to mouse dragged function. This runs when the mouse button is pressed and held and the mouse is moved around. First we save the x,y of where we clicked then loop through our handler to find the candy that is at that location. Once we do that we lock the candy set the candy's position the same as the mouse, making sure to lock it either horizontally or veracity. We make sure to save where the candy came from and where it is at right now

…

public void mouseDragged(MouseEvent e) {

if(*dragging*=true) {

int cx= e.getX()/(500/8);

int cy= e.getY()/(500/8);

GameObject temp;

for(int i =0;i<handler.object.size();i++) {

temp = handler.object.get(i);

if(temp.id==*ID*.***Candy***&& Candy.*updatemap*==false) {

if(cx<8&&cy<8&&cx>=0&&cy>=0)

if(temp.getA()==*Imx*&&temp.getB()==*Imy*) {

if(cx==*Imx*&&Math.*abs*(cy-(temp.getY()/(500/8)))==1) {

int temy= temp.getY();

int temx= temp.getX();

temp.setY(cy\*(500/8));

collision(temp,temx,temy,cy);

…

Once the mouse has moved to a new box, we run the collision function. The collision function takes our candy and checks if its hit-box is hitting any other candy’s hit-box by looping through the handler. If it is the candy that it is hitting is given our candy’s old possession and is moved there while Cmap matrix we have created is updated to reflect this change.

…

public void collision(GameObject a,int oldx,int oldy,int old) {

GameObject temp;

for(int i =0;i<handler.object.size();i++) {

temp = handler.object.get(i);

if(temp.id==*ID*.***Candy***&&a.getAb()!=temp.getAb()) {

if(a.getBounds().intersects(temp.getBounds())) {

if(oldy==temp.getY()) {

oldx = oldx/(500/8);

temp.setX((oldx)\*(500/8));

int t = *Cmap*[temp.getY()/(500/8)][old];

*Cmap*[temp.getY()/(500/8)][old] = *Cmap*[temp.getY()/(500/8)][oldx];

*Cmap*[temp.getY()/(500/8)][oldx] = t;

...

Once the player is done moving the candy and releases the mouse button. The checkmap function is called to check if the changes the player made result in any candy matching up. If it doesn’t then nothing happens and since the main Map matrix has not been changed,(we have been using Cmap so far) all the candies we go back to their old position.

…

public void mouseReleased(MouseEvent e) {

if(Game.*gameState*==*GameState*.***Game***&&*updatemap*==true) {

if(checkmap()==true) {

Audio.*getSound*("nice").play();

updatemap();player.setMoves(player.getMoves() - 1);

}

if(*dragging*==true) *dragging*= false;

*updatemap*=false;

}

...

But if did result in candy matching then updatemap function is called and Game.Map is set to equal the Cmap matrix the player has been changing around. And the player moves go down by 1.

Now that let us look at the cmapmaker, updatemap and mapmaker functions. Createmap creates a random map at the start of the game. Checkmap, checks if the map has 3 or more matching.They all use the same principle of looping through an array and counting the number of times an element happens.

Createmap loops through the matrix assigning a random number to that position from a list of choices. If an element is the same number as the one before then that number is removed from choices and the next element is given a random number with out that number. After that number is but back int the list and the loop continues.

…

public void Mapmaker() {

ArrayList<Integer> type = new ArrayList();

int[][] array= new int[8][8];

type.add(0);type.add(1);type.add(2);type.add(3);type.add(5);

int varx = 0;

int vary=0;

int countx=1;

int county=1;

for(int y=0;y<8;y++) {

for(int x=0;x<8;x++){

if(countx<2) {

array[y][x] = type.get(rand.nextInt(type.size()));

if(array[y][x]==varx) {

countx++;

}

varx=array[y][x];

}else {

type.remove(Integer.*valueOf*(varx));

array[y][x] = type.get(rand.nextInt(type.size()));

type.add(varx);

varx=array[y][x];

countx=1;

}

}

...

Checkmap does exact same thing, except it checks if three or more of the elements are the same. Once it finds ones that are it adds to the score and changes the elements to 4 which is an empty candy according to game.

private int checkmap() {

int varx= Game.*map*[0][0];

int vary= Game.*map*[0][0];

int countx =0;

int county =0;

int q=0;

int p=0;

for(int y=0;y<8;y++) {

for(int x=0;x<8;x++) {

if(Game.*map*[y][x]==varx && x>0&&varx!=4) {

countx++;

}else {

varx=Game.*map*[y][x];

if(countx>=3) {

if(x==0&&y>0) {

q=8;

p=y-1;

}else{

q=x;

p=y;

}

if(Game.*map*[p][q-1]==5) {

player.setMoves(player.getMoves() + 1);

}else {

player.setScore(player.getScore() + county);

}

for(int i=countx;i>0;i--) {

Game.*map*[p][q-i]=4;

}

**Project Link & Video:**

**https://github.com/hassan701/OOP-project**

[**https://youtu.be/CcOWzA8Eqwg**](https://youtu.be/CcOWzA8Eqwg)

**Project Reference:**

Candy Crush - For idea and art design

Huniepop - Leveling design

**Code and Tutorials used/flowed to develop this program:**

Java Programming: Let's Build a Game series by RealTutsGML : YouTube <https://www.youtube.com/watch?v=1gir2R7G9ws&list=PLWms45O3n--6TvZmtFHaCWRZwEqnz2MHa>

Java Game Programming for Beginners - #9 - Mouse Moved Events by TheJavaHub - YouTube, <https://www.youtube.com/watch?v=CZ6i8RM6Uxg&t=350s>

**Sprites:**

Crystals(all): Craftpix - <https://craftpix.net/freebies/free-crystals-2d-game-items/>

**Background:**

Forest background(both): Craftpix, <https://craftpix.net/freebies/free-cartoon-forest-game-backgrounds/>

**Music and Sound:**

Candy sound – Jingle\_Win\_Synth\_03: Free sound, <https://freesound.org/people/LittleRobotSoundFactory/sounds/274177/>

Memories by Benjamin Tissot: Bensound, <https://www.bensound.com/royalty-free-music/track/creative-minds>

Creativeminds by Benjamin Tissot: Bensound, <https://www.bensound.com/royalty-free-music/track/memories>