Night-Collector

by

Andreas Ambühl & Louisa Reinger

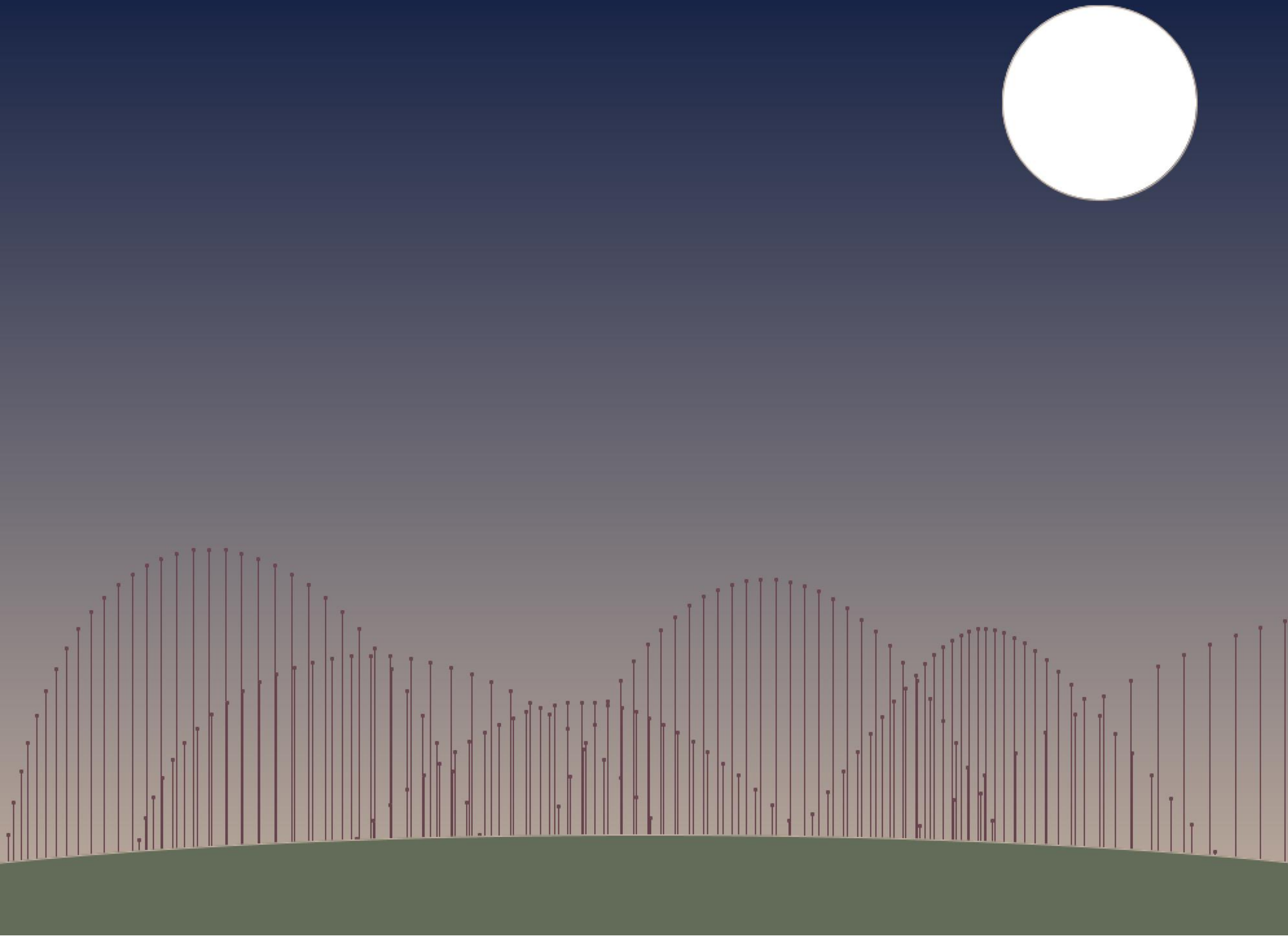


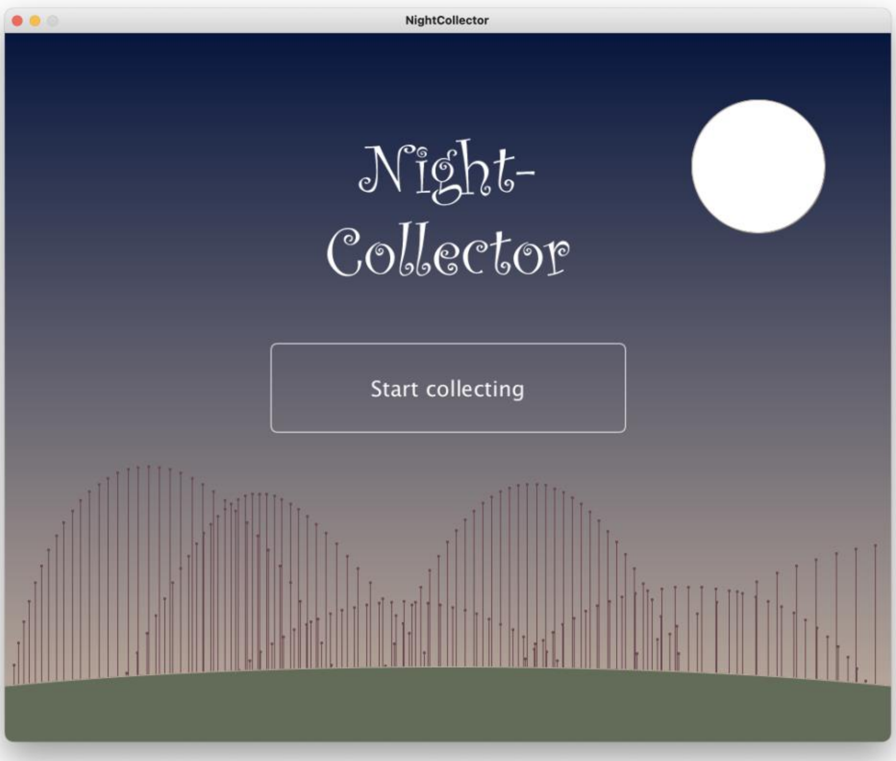
Table of Contents

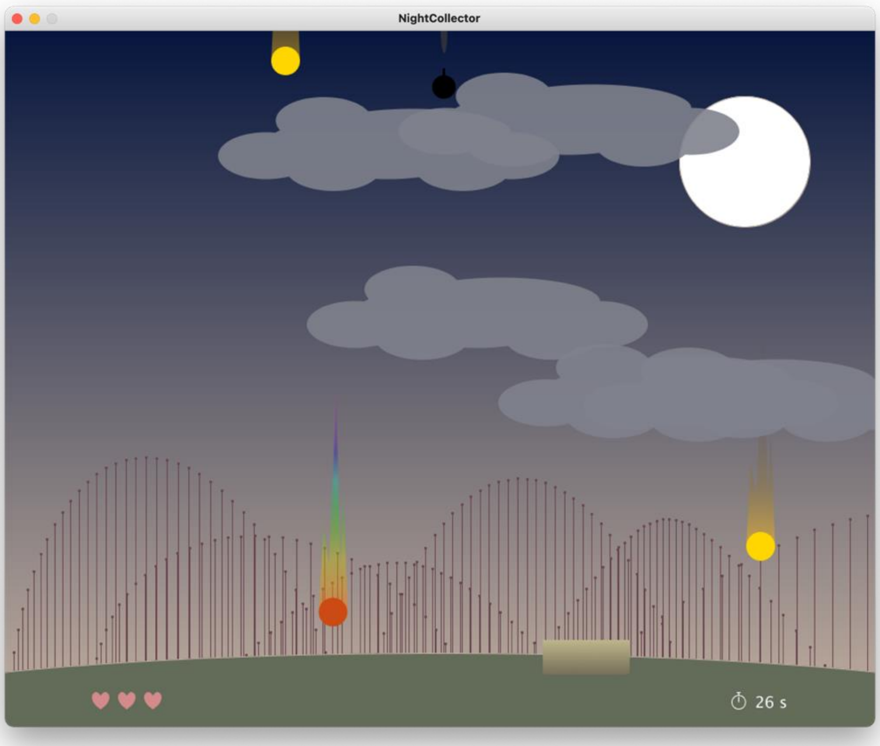
This project was made with Processing 3_5_4

Project description

Description

The goal of this game is to survive as long as possible. Initially, the player has 5 lives. These can decrease or increase during the game.
Using the mouse, the basket can be moved horizontally to catch the objects falling from the night sky.
There are 3 different objects that can be caught (see Objects).





Objects

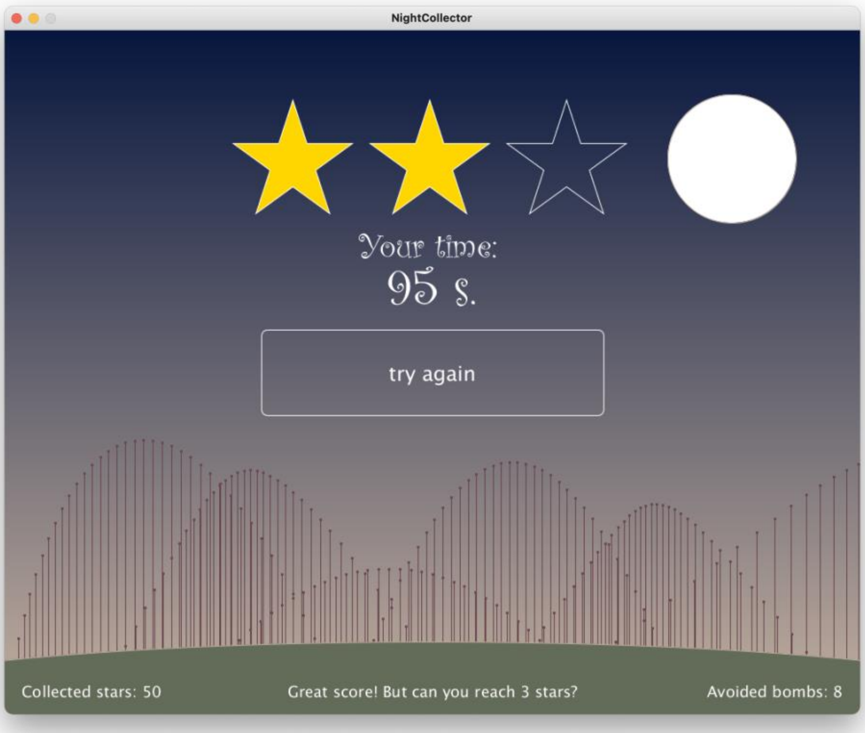
	Behavior on catching	Behavior on not catching
Star	(+ 1 star point for statistics)	**- 1 life**
Power-Star	**+ 1 life**	*Nothing happens*
Bomb	**Game Over**	(+ 1 avoided bomb for statistics)

Increase of the difficulty

As the game progresses, there will be more and more clouds appearing, limiting the view of the falling objects. In addition, every 20 seconds the “game speed” is increased by the factor 1.2, which influences the speed of the falling objects and the (randomized) time of spawning the falling. The feeling of this effect is further emphasized by the music getting faster by the factor 1.08.
Overall, there are various “game balancing parameters” which can be tweaked at whish; They can be found at the top of NightCollector.pde. With these game balancing parameters, all would be prepared for an additional future feature: Choosing different game difficulty settings before starting a new game.

Game Over

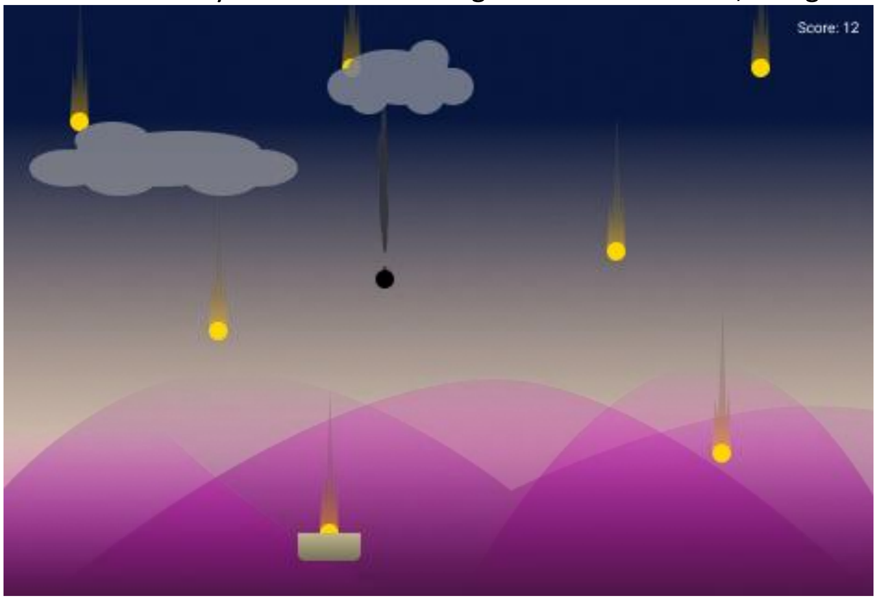
The game is over as soon as either a bomb is caught, or the player has no more lives. Then the time is stopped, and the end screen is displayed. Depending on how long the player survived, the player gets a better rating in the form of rating stars and a (hopefully) encouraging comment about the performance.



Collected stars: 50
Great score! But can you reach 3 stars?
Avoided bombs: 8

Approach

- **Iterative**
  We took an iterative approach and revised/improved the look, features, and game logic several times.

  This was the very first idea how the game could look like, designed on Figma:
  

- **Homemade**
  We drove with the approach to be creative and to create as much as possible ourselves and not to depend on external sources. Therefore, all visible objects and also the music came from ourselves (see Visual Objects, Music).

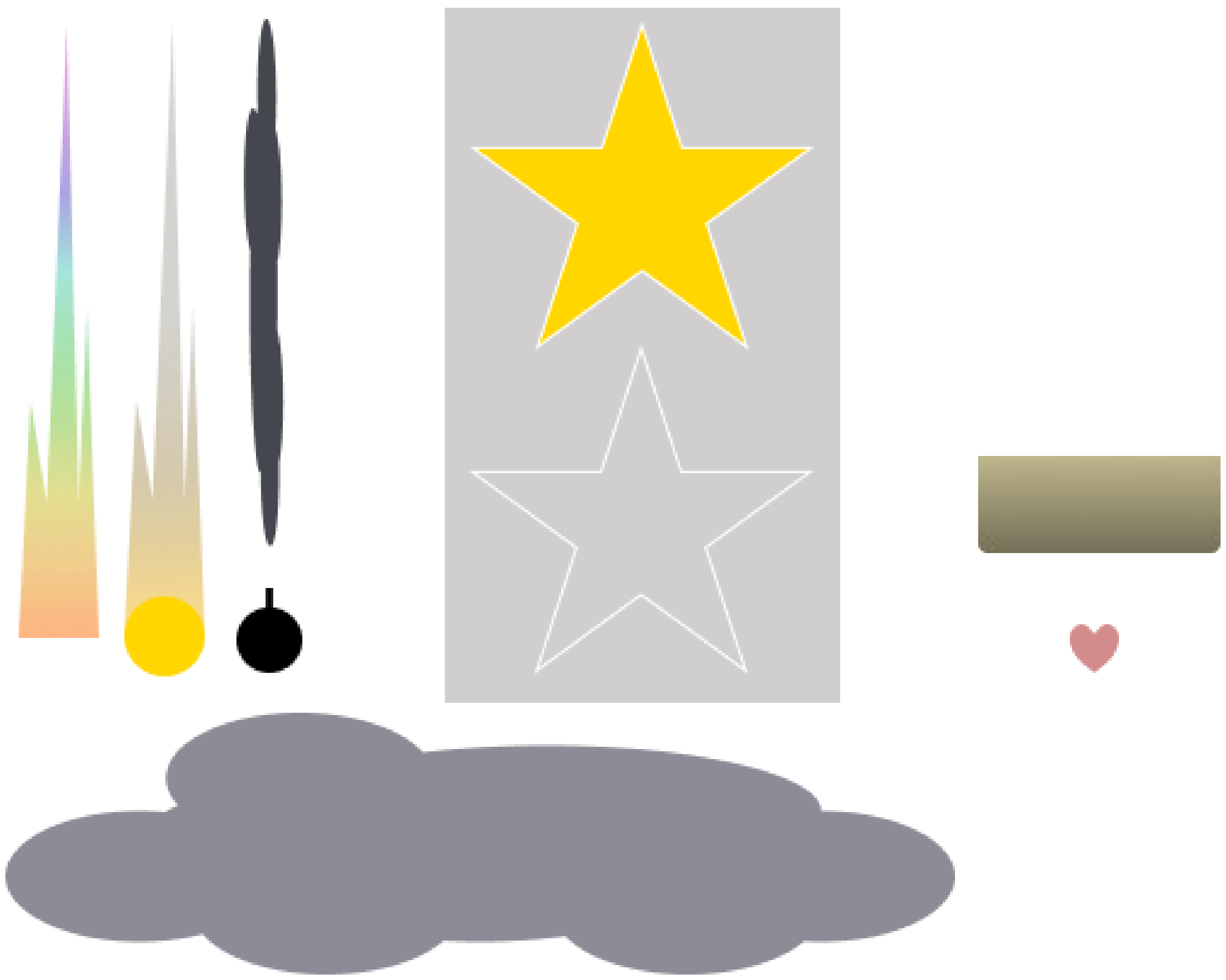
- **Approaches from the classroom**
  With this project we tried to try out and apply some approaches from our lessons (see Moving Basket, Mountains, Water).

- **Code structure**
  We tried not to let the classes become too large but divided too large classes into several clearly separated classes (see Code Structure).
  We also tried to remove some code duplication (see Approach to avoid code duplication).

Some realization details

Visual Objects

All visible objects are self-designed.
The following objects were designed in Figma and loaded into the project as PNGs:



All other visible objects have been implemented directly in Processing.

Music and Sounds

The music is from a CD of one of the authors of this game: Andreas Ambühl. More about this band: https://andreasmusic.ch/ararat-quintet.

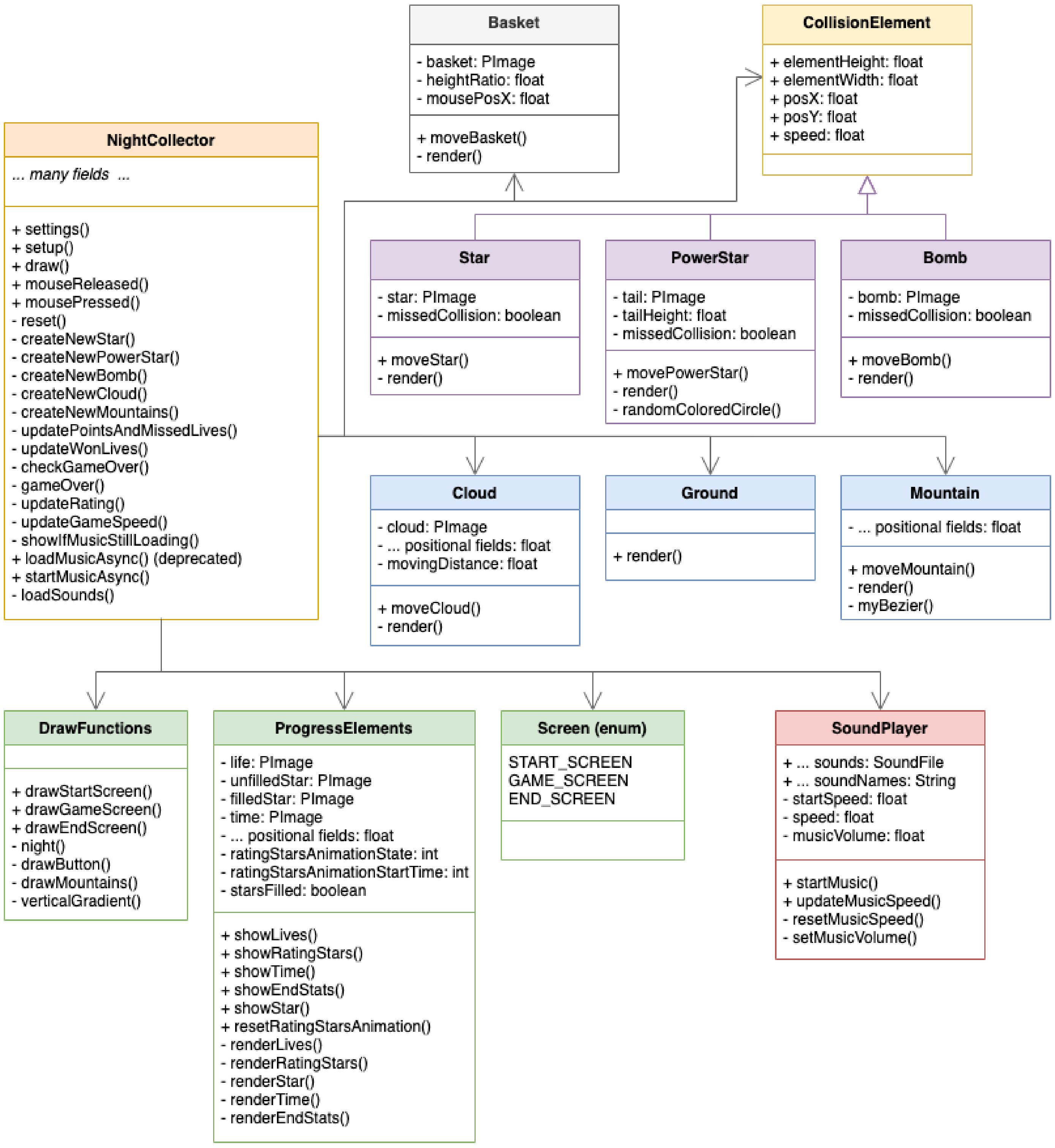
The game-sounds are all carefully selected from https://freesound.org. Every detail-link and the license details are specified in the class SoundPlayer.

Increasing Speed

As mentioned earlier, to give a more thrilling gaming experience once the game becomes more difficult, the speed of the music is also increased.

Code Structure

Our code structure can be seen in the UML-diagram. For a reasonable size of the UML-diagram, some fields were consolidated (e.g., ...positional fields in Class Cloud instead of all the individual fields) and parameters and return types of the methods were omitted.



Basket

User Interaction

The basket can be moved in x-direction via mouse.
And there is also a little logic built in that prevents the basket from running out of the picture.

In Class ***Basket***:

```
void moveBasket(){
    mousePosX = mousePosX+(mouseX-mousePosX)*speed; //action: roll to mouseX-Position

    render();
}


private void render(){

  posX = mousePosX-(elementWidth/2);

  //prevent basket from running beyond screen
  if(mousePosX <= elementWidth/2){
    posX = 0;
  }
  if(mousePosX >= width-(elementWidth/2)){
    posX = width-elementWidth;
  }

  push();
  translate(posX, posY);
  image(basket, 0, 0);
  pop();
}
```

Collision Detection

For the collision detection there must be proved if the bottom of each falling down element is at the y-position of the top of the basket.
If yes, there has to be proved if the element is inside the valid x-range of the basket. If yes there is a collision, so the element was cached by the basket, and it disappears. If not, the element was missed (and it should not immediately disappear).
There are three different types of falling down elements: Stars, PowerStars and Bombs.
Depending on what landed in the basket, different additional functions are executed, e.g. to update the number of lives.

Using the approach in Approach to avoid code duplication at collision detection an attempt was made to write a function that takes an array of CollisionElements (Star, PowerStar and Bomb became subclasses of CollisionElement) and performs the x/y position check.
Unfortunately, this didn't work, so now there is a separate function for each falling down element.

In Class ***NightCollector***:

```
private void updatePointsAndMissedLives(){
  for(int i = 0; i < stars.size(); i = i+1){ //all stars
  ....
}
```

```
 private void updateWonLives(){
   for(int i = 0; i < powerStars.size(); i = i+1){ //all powerStars
    ....
}


 private void checkGameOver(){
   for(int i = 0; i < bombs.size(); i = i+1){ //all bombs

     if(bombs.get(i).posY + bombs.get(i).elementHeight >= basket.posY){  //y-position >= basket

        if(bombs.get(i).posX + bombs.get(i).elementWidth / 2 >= basket.posX &&
           bombs.get(i).posX + bombs.get(i).elementWidth / 2 <= basket.posX + basket.elementWidth){ //x-range same as basket

          if(!bombs.get(i).missedCollision){

              soundPlayer.soundBomb.play(); //collision (correct x/y)
              gameOver();
              return;
          }
        }else{
          if(!bombs.get(i).missedCollision){
            avoidedBombs.add(bombs.get(i));
          }
          bombs.get(i).missedCollision = true; //missed (correct y / incorrext x)
        }
     }
   }
 }
```

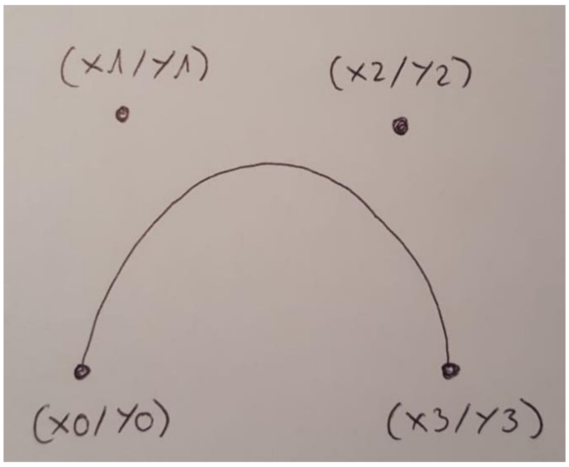
Since posY is a float, the elements may never have exactly the same y-position values as the basket. Therefore, the check is not done with "==" but with ">=" or ”<=”.
Since elements can be caught that are already below the basket, the property *missedCollision* was implemented. *missedCollision* is set to true once it has crossed the y-boundary and there was no collision with the basket. If there was a collision, the element is deleted directly, or it is not necessary as here with the bombs, because the game is over.

Mountains

The mountains were modeled using Bezier curves:

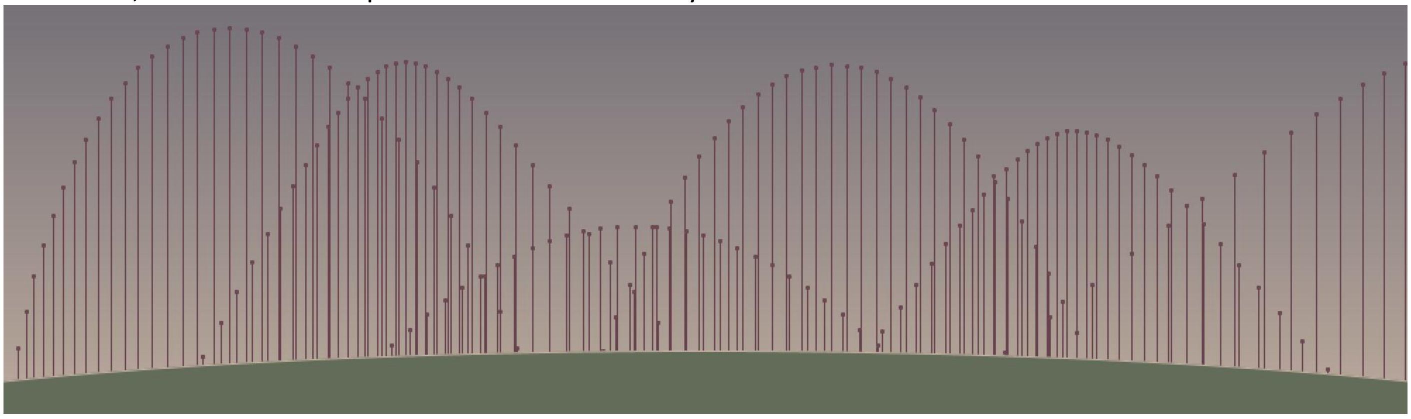
In Class ***Mountain***:

```
private void myBezier(float x0, float y0, float x1, float y1, float x2, float y2, float x3, float y3){
  int steps = 40; // Increase value to increase discretization
  int drawSteps = steps;

  for (int it=0; it<drawSteps; it++){
    float t=it/(float)steps;

    //bezier-spline
    float b0 =   -(t * t * t) + 3*(t * t) - 3*t + 1;
    float b1 =  3*(t * t * t) - 6*(t * t) + 3*t;
    float b2 = -3*(t * t * t) + 3*(t * t);
    float b3 =      t * t * t;

    //linear interpolation
    float x = b0*x0 + b1*x1 + b2*x2 + b3*x3;
    float y = b0*y0 + b1*y1 + b2*y2 + b3*y3;

    // render points
    circle(x,y,2);

    //vertical colored lines
    line(x,height, x, y);
    stroke(102, 66, 77);
   }
}
```



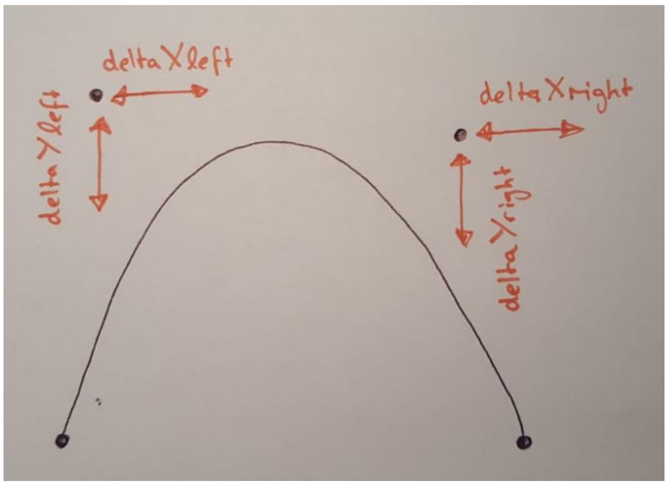
The parameters correspond to the image above.

The mountains couldn't be colored completely, because this required too much computing power and caused the game to stutter.
Therefore, we decided to represent the mountains by vertical lines and circles.



Movement

To move the mountains, a function was created, which can be passed 4 parameters, with which the upper two points of the mountains are moved:



In Class ***Mountain***:

```
void moveMountain(float deltaXleft, float deltaYleft, float deltaXright, float deltaYright){
  leftX1  += deltaXleft;
  leftY1  += deltaYleft;
  rightX1 += deltaXright;
  rightY1 += deltaYright;

  render();
}
```

To make the mountains move constantly, the function is called as follows:

In Class ***DrawFunctions***:

```
private void drawMountains(){
  for(int i = 0; i < mountains.size(); i = i+1){

      float deltaXleft  =  0.5 * sin(millis()/((i+1)*1000));
      float deltaYleft  = -0.2 * sin(millis()/((i+1)*1000));
      float deltaXright = -0.5 * sin(millis()/((i+1)*1000));
      float deltaYright = -0.2 * sin(millis()/((i+1)*1000));

      mountains.get(i).moveMountain(deltaXleft, deltaYleft, deltaXright, deltaYright);
    }
}
```

The continuous time is packed into a sine function to generate oscillating values.

Clouds

Appearing Pattern

A new cloud can appear every 5 seconds. However, so that it is not too predictable a cloud does not have to appear every 5 seconds.
Since the playtime is a float, there are inaccuracies which are exploited here for randomness:

In Class ***DrawFunctions***:

```
float playTime = millis()*0.001f;

 ...

//clouds
if (playTime % 5 <= 0.01) { //create a new cloud after a certain time
  createNewCloud();
}
```

Movements

To make it a bit more exciting for the viewer, there are 3 different movements a cloud can have:

In Class ***Cloud***:

```
void moveCloud(float t, int movement){
  float m;
  if(movement == 1){
    m = sin(t/6 * PI);
  }
  else if(movement == 2){
    m = cos(t/8 * PI);
  }
  else{
    m = 1;
  }

  cloudPosX = cloudInitXPos + (movingDistance/2) * m; //action: horizontal movement
  render();
}
```

The moveCloud function is called in the draw class, with each new cloud having a different move than the previous two (by (i%3)):

In Class ***DrawFunctions***:

```
for(int i = 0; i < clouds.size(); i = i+1){ //existing clouds
  clouds.get(i).moveCloud(playTime, i%3);
}
```

Color changing Power Star

The PowerStar is composed of an image (with the tail of the star) and a circle which changes the color.
To change the color on each draw() call the HSB color mode was used. With each call the hue value is set to a different random number between 0 and 360.

In Class ***PowerStar***:

```
private void render(){

  push();
  image(tail, posX, posY);
  randomColoredCircle(int(posX - 0.5 + (elementWidth/2)),
                      int(posY + tailHeight),
                      elementWidth);
  pop();
}

private void randomColoredCircle(int x, int y, float width){
  colorMode(HSB, 360, 100, 100);
  noStroke();
  float hue = random(0, 360);
  fill(hue, 90, 80); // Hue, Saturation, Brightness
  circle(x, y, width);
}
```

Falling Objects

The falling objects (Bombs, Stars, PowerStars) appear at random intervals.
The intervals remain random but become shorter as the game time increases.

In Class ***DrawFunctions***:

```
//stars
if (millis() - starTimer >= millisBetweenStars) { //create a new star after a certain time
  createNewStar();
  millisBetweenStars = starSpawnFactor + random(1000);
  starTimer = millis();
}
```

In Class ***NightCollector***:

```
private void updateGameSpeed() {
  if (inNewSecond && seconds > 2 && seconds % secondsUntilSpeedIncrease == 0) {
    gameSpeed            *= gameSpeedIncreaseFactor;
    starSpawnFactor      /= gameSpeedIncreaseFactor;
    powerStarSpawnFactor  /= gameSpeedIncreaseFactor;
    bombSpawnFactor      /= gameSpeedIncreaseFactor;
    //increase music speed with different factor:
    soundPlayer.updateMusicSpeed(soundSpeedIncreaseFactor);
  }
}
```

The speed of the objects is also getting faster with increasing game time.
The speed of the different objects are different but they all include random factor.
This keeps the game exciting and different every time.

In Class ***Star***:

```
this.speed          = gameSpeed * 4 + random(3);
```

Multiple Screens

There are 3 different screens. The currentScreen controls which screen is displayed.

In Class ***NightCollector***:

```
void draw(){

  switch(currentScreen){
    case START_SCREEN :
      drawFuncs.drawStartScreen();
      break;
    case GAME_SCREEN :
      drawFuncs.drawGameScreen();
      showIfMusicStillLoading();
      break;
    case END_SCREEN :
      drawFuncs.drawEndScreen();
      break;
  }
}
```

Boundaries

The following features were attempted to be implemented, but for different reasons they could not be realized and were in some cases replaced by other solutions.

Moving Basket

The basket can be moved in x-direction via mouse.
Similar to the soccer ball from class, a small delay was first built in so that the ball rolls smoothly behind the mouse.

In Class ***Basket***:

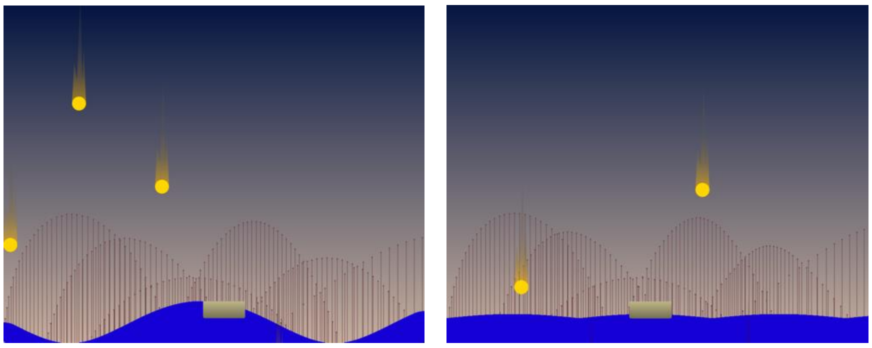
```
void moveBasket(){
    mousePosX = mousePosX+(mouseX-mousePosX)*speed; //action: roll to mouseX-Position

    render();
}
```

However, since we decided to hide the mouse during the game, this version became too heavy to play the game well. The basket now represents the mouse and is always in the same X position as the mouse.

Look of Mountains

The mountains could not be colored fully, because the computational effort was too high and the game started to stutter. (see Mountains).

Water



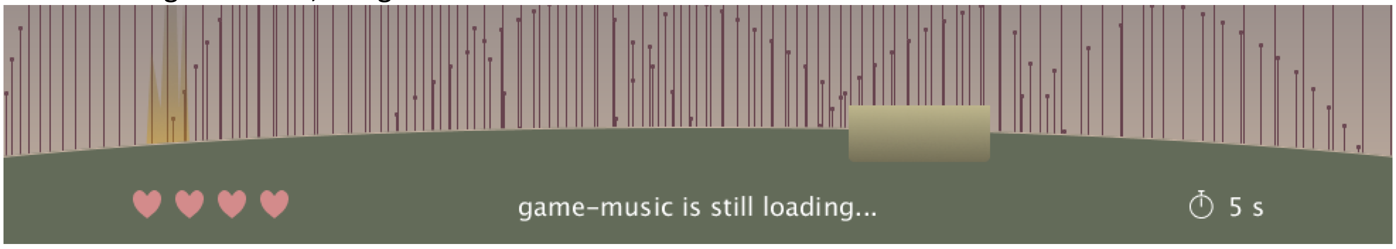
One idea was to implement moving water with sine instead of the grass.
However, a flat filling was not possible for the same reason as with the mountains. A second attempt to display the waves only via strokes did not work either, because the game only ran smoothly up to a thickness of 10. With thicker strokes the game also started to stutter.
Therefore the water was replaced by a green grass surface.

In Class ***Water***: (doesn't exist anymore)

```
private void sinus(float amplitude, float xDirection, float yPos, float xStretch){

  float xPrevious = 0;
  float yPrevious = amplitude * sin(0 - xDirection) + yPos;
  float y;

  for(int x = 0; x<width; x = x+1){ //for each x value...

    y = amplitude * sin(xStretch * x - xDirection) + yPos;

    line(xPrevious, yPrevious, x, y);
    strokeWeight(10); //too slow if thicker
    stroke(0, 0, 205);

    xPrevious = x;
    yPrevious = y;
  }
}
```

Music took too long to load (solved)

At first the music file was an mp3 file with a size of 7.2 MB.
The mp3 file took about 8 to 10 seconds for the game music to load. Therefore, it was not possible to hear the music from the start of the application. The music had to be loaded asynchronously. If the player started the game before the music was loaded, it was indicated at the bottom of the screen that the music is still loading. However, the game could still start.



In Class ***NightCollector***:

```
// Will run on a separate thread -- called by thread("loadMusicAsync");
void loadMusicAsync() {
  //soundPlayer.music = new SoundFile(this, soundPlayer.musicFileName);
}

// will run on a separate thread -- called by thread("startMusicAsync");
void startMusicAsync() {
  while (soundPlayer.music == null) {
    delay(100);
  }
  gameMusicLoaded = true;
  soundPlayer.startMusic();
}
```

Finally, we were able to solve the problem by converting the mp3 file into a wav file. Now the music is loaded immediately.

Music didn't work on Windows (solved)

Initially, the music file was in mp3 format and loaded asynchronously as mentioned above.
This worked perfectly on the mac, not blocking the game, and starting to play the music once the music was loaded. On Windows, however, this led to a NullPointerException resulting in Crashing of the whole Sound-functionality: The game was still fully playable but without any sound. It seems that Windows

cannot load music in mp3 asynchronously. It can load mp3 when not on a separate thread, but then we must wait again for a few seconds and the UI would be completely blocked.

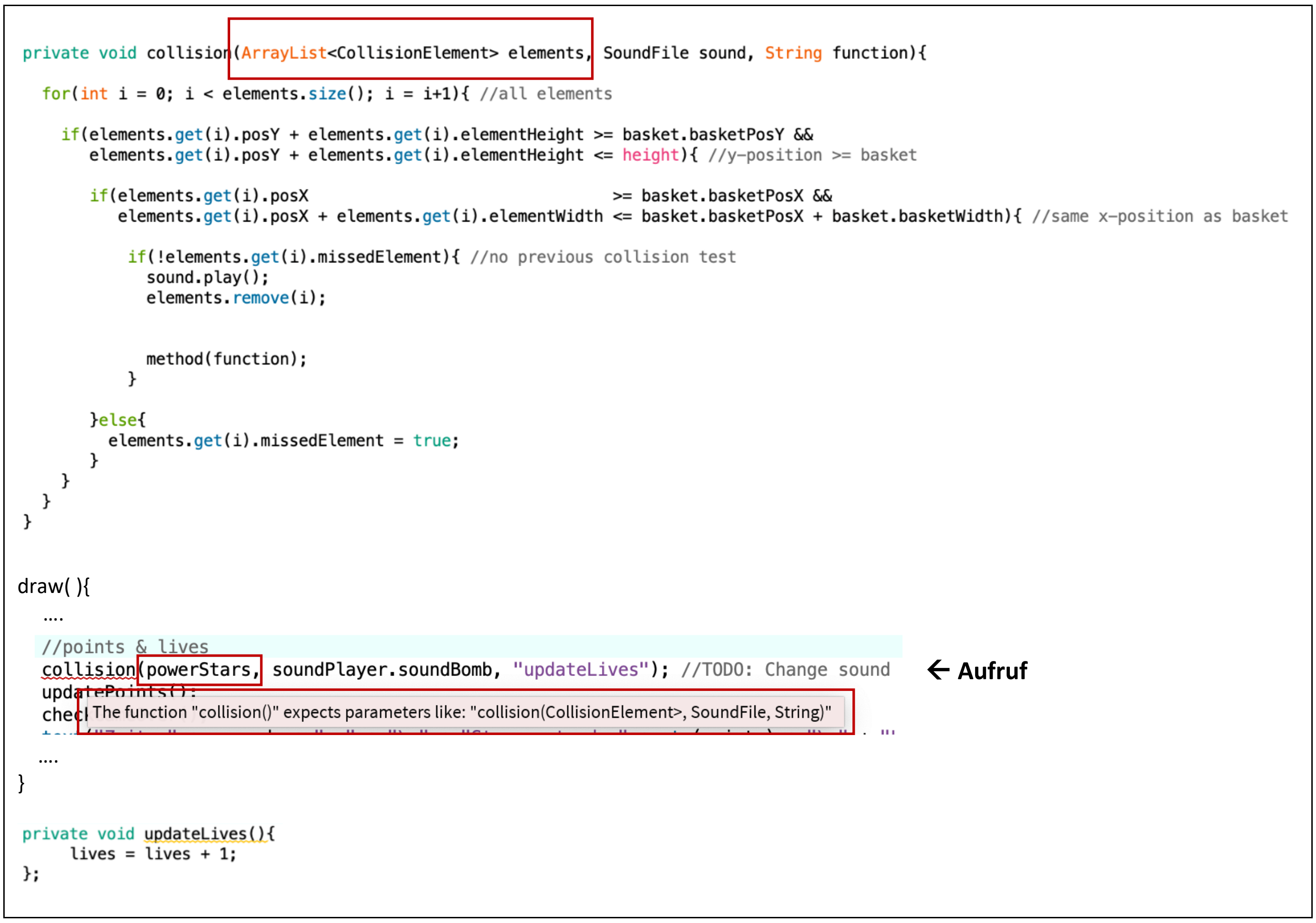
As a solution for both Mac and Windows platform, we converted the mp3 file to a wav file (in Audacity) resulting in a quite large file of 50MB. But now, we can load this file nearly instantly into Processing and thus we do not have to load it asynchronously anymore.

In Class ***NightCollector***:

```
void loadSounds() {
  soundPlayer.music                = new SoundFile(this, soundPlayer.musicFileName);
  soundPlayer.soundCollect         = new SoundFile(this, soundPlayer.soundCollectName);
  soundPlayer.soundBomb            = new SoundFile(this, soundPlayer.soundBombName);
  soundPlayer.soundMissed          = new SoundFile(this, soundPlayer.soundMissedName);
  soundPlayer.soundPowerUp         = new SoundFile(this, soundPlayer.soundPowerUpName);
  soundPlayer.soundMissedPowerUp = new SoundFile(this, soundPlayer.soundMissedPowerUpName);
  soundPlayer.soundGameOver        = new SoundFile(this, soundPlayer.soundGameOverName);
  soundPlayer.soundStar            = new SoundFile(this, soundPlayer.soundStarName);
  soundPlayer.soundMissedPowerUp.amp(0.5);  // otherwise this sound is too loud
}
```

Approach to avoid code duplication at collision detection

For the collision detection (see Collision Detection) we tried to write a generally valid function that can be called several times for the different CollisionElements.



```
private void collision(ArrayList<CollisionElement> elements, SoundFile sound, String function){

  for(int i = 0; i < elements.size(); i = i+1){ //all elements

    if(elements.get(i).posY + elements.get(i).elementHeight >= basket.basketPosY &&
       elements.get(i).posY + elements.get(i).elementHeight <= height){ //y-position >= basket

       if(elements.get(i).posX                              >= basket.basketPosX &&
          elements.get(i).posX + elements.get(i).elementWidth <= basket.basketPosX + basket.basketWidth){ //same x-position as basket

          if(!elements.get(i).missedElement){ //no previous collision test
            sound.play();
            elements.remove(i);


            method(function);
          }

       }else{
         elements.get(i).missedElement = true;
       }
    }
  }
}


draw( ){
  ....
  //points & lives
  collision(powerStars, soundPlayer.soundBomb, "updateLives"); //TODO: Change sound
  updatePoints();
  chec
  ....
}

private void updateLives(){
     lives = lives + 1;
};
```

← **Aufruf**

The function "collision()" expects parameters like: "collision(CollisionElement>, SoundFile, String)"

This approach was not possible for the following reason:

A **PowerStar** is a subclass of **CollisionElement**, but an **ArrayList<PowerStar>** is not a subclass of **ArrayList<CollisionElement>**. Therefore, a separate function was written for each group of elements (for stars, powerStars & bombs) (see Collision Detection).

Beat Detection

Since we decided to let the music get faster, this made beat detection more difficult, since the pitch of the music gets higher and higher as the speed increases.
Because of the many features already implemented, we decided to leave the beat detection out and not look for another solution.
The mountains and clouds therefore now do not move to the beat of the music.

Conclusion

We had a lot of fun implementing the game. Due to the clear code structuring (e.g., a separate class for each object), the code could easily be extended with new features. We reached some limits, but in the end, we are very satisfied with the result. The teamwork was also very pleasant. We would have liked to implement more features, but that would probably have gone beyond the scope of this project.