# Backlog CBL

By Diederik Webster 1957880 and Matthijs Smulders 1973169

Must have Should have Could have

|  |  |  |
| --- | --- | --- |
| Name: | How to demo: | Notes: |
| Run button is pressed = game starts | Start the application, the game window should appear and a level with a player will be shown on screen. | For this we need a player class, a level class, and a collision class. The rendering is done by a JPanel component which is put inside a game-loop function.  Goals:  - Learning how to render objects by using Swing components  - Basic programming  - Collision detection  - Methods utilization |
| The level gets rendered | When the application is running, you see a level. | For this we need to read a txt file with inside the level. Then this level needs to be rendered to the frame.  Goals:  - work with Swing components  - read from a file |
| When the D button is pressed move right | When you start the application, and you move the player to the right using the d key, the player should smoothly accelerate and decelerate. | For this we need to create a method in the player class. This checks if the key was pressed the in the previous loop and accelerates the player.  Goals:  - Learning how to detect an input in the JPanel component.  - Learning how a player moves smoothly. |

|  |  |  |
| --- | --- | --- |
| When the A button is pressed move left | When you start the application, and you move the player to the left using the a key, the player should smoothly accelerate and decelerate. | For this we need to create a method in the player class. This checks if the key was pressed the in the previous loop and accelerates the player.  Goals:  - Learning how to detect an input in the JPanel component.  - Learning how a player moves smoothly. |
| The player jumps when the space key is pressed | When you start the application and you move the player upwards using the space key, the player should accelerate smoothly upwards and then accelerate downwards again until there is an obstacle underneath the player. | For this we need to create a method inside the player class. This checks if the space key is pressed and accelerates the player smoothly upwards and the player should fall back down again. The camera shouldn’t move along with the player.  Goals:  - Learning how to use gravity in a game.  - Making a player accelerate smoothly upwards and downwards. |
| When the player touches an obstacle, it collides and can’t move further | The player can stand on obstacles and does not fall through them. Additionally, it won't be able to run through walls. | If the player wants to move, we should first check if, when it is moved, it will share coordinates with an obstacle. If so, don't use the inputted movement.  Goals:  - Learning to use the coordinates from the JFrame component. |
| The level should end when the player pressed W and touches a door | When a player touches the door at the end of the level and the W key is pressed, the next level should appear. Or, in case of the last level, the end screen. | For this to work we need to make an end level function, this checks when the W key is pressed and the player is near the coordinates of a door, a new level needs to be shown.  Goals:  - Learning how to get coordinates from the JFrame. |

|  |  |  |
| --- | --- | --- |
| When space and a sidekey are pressed together, the player moves up and to the side simultaneously | When you start the application and press the right or left and up button simultaneously the player should jump and move in the air. | For this the jumping method and moving method should be able to move simultaneously.  Goals:  - Making functions work together. |
| Moving camera | The level should move along with the player. | The camera should stop moving when near the end of the level and when the player is at the beginning.  Goals:  - Learning how to make the camera move |
| The player respawns when touching a spike | When a player touches a spike, the player should be reset to the beginning of the level. | For this to work we need to make a restart function, along with a function which detects when the player dies.  Goals:  - Resetting a level |
| Multiple levels | When a level is completed, there are move levels to complete than only one. | For this to work we need to levels using arrays, in a array an integer represents an object.  Goals:  - Working with arrays |
| If the player exits the screen above or below, he dies (below) or nothing happens (above) | When falling below the level, the lose screen should appear and when above the level, the player should move further without getting an error. | Introduce if statement to check if the y coordinate is too big or too small when the player moves.  Goals:  - Preventing errors in our code |
| Game-state screens | When the game is paused, won, lost, started or ended, the appropriate screen is shown | We need to create a variable to save the gamestate in level. Check this variable when drawing the level.  Goals:  Using swing |

|  |  |  |
| --- | --- | --- |
| The players position in which level should be saved | If you stop the application, and start it up again, you start at the point you stopped. | We must save the level number and player coordinates to a file.  Goals:  Reading and writing a file |
| The levels should be constructed from a file, so anyone can easily modify the levels | If you change the level files, the level displayed in the application should also change | We must read from a file in the constructor of the Level class  Goals: reading from files |
| Randomly generated levels | When you start the application, an infinitely long level is present with checkpoints along the way. | Create predetermined blocks and putting them after each other to create an infinite long level.  Goals:  - Using the random in Java  - Creating levels |
| Moving enemy | In a level a moving enemy is present. The player dies when touching it. | Create an enemy class in the object class. And making him move on a predetermined path.  Goals:  - Basic programming  - Making an algorithm |
| Spring obstacle | When a player jumps on a spring it propels the player a certain amount of blocks upwards. | Create a spring class in the object class.  Goals:  - Using a spring and adding an effect to a player  - adding extra movement on top of other functions |
| Life system | The player starts with 5 lives, when dying a life is subtracted and a player can earn lives by collecting 10 coins. | Create a variable life and using this in the other methods.  Goals:  - Using a life system |

Challenges:

* Smooth Physics
* Reading and writing files

# Decisions and process (“REAME.txt”)

## Physics

We will use acceleration variables to create dynamic speeds.

We use a chart in the style of the above. The acceleration is constant up to a certain speed, after which it steps up or down. (In the game, the values differ, this graph explains it clearest. For horizontal speed and acceleration, the acceleration steps up from a certain speed).

We also implement a max speed: this would be the resulting graph:

We use this concept separately for horizontal and vertical acceleration and speed.

If you want to change direction at high speeds, we want that to be snappy. For this, we make the deceleration higher. If we wanted the ground to feel “slippery”, we could turn this deceleration down.

## Save Data and Level files.

In save.txt, we save in this order:

* Levelnumber
* Player x
* Player y
* X0 (for the camera)

An empty file will start the game from the beginning, a file with only a level number will spawn the player at the default location.

The level files describe a level with:

* “RB” for a row of blocks
* “CB” for a column of blocks
* “RS” for a row of spikes
* “CS” for a column of spikes
* “ED” for the end door

The first two following numbers are the x and y coordinate of:

* Collums: the “lowest” block, highest in value.
* Rows: the leftmost block, lowest in value.
* Door: the x and y coordinate.

The third number is the length/height of the row/column.

The game only reads 3 level files, with the names:

Level1.txt, level2.txt, level3.txt.

If you want to increase this, you need to change some variables in the code.

## Collision detection

The following is the checklist where we based the collision detection on.

If a side-key is pressed:

* check which side
* How much do you move? (speed)
* Check if you collide
* if collide 🡪 move as far as you can go
* if you don’t collide à move

If up-key is pressed:

* check if on ground
* start jump
* You have to be able to execute other functions during jump
* if you collide with something: stop jump
* If Jump is done: start falling

If you walk of platform:

* start falling
* during fall: be able to execute other functions

Fall:

* Check if you can fall the max fall distance
* If not, move as far as you can
  + Stop the fall
* If you can, keep falling

On ground check has to be checked all the time

When you move while you jump, you don’t have to account for the y-direction when checking for collisions in the x-direction, and vice-versa. This is because we implement the functionality of moving as far as you can go if a collision is detected on the intended trajectory.

🡪 if it would happen that you can overshoot the collision because of your increasing/decreasing your y-position, the game would receive a new keyEvent, check for collisions again and not detect a collision anymore.

When falling, we found that you do have to check for collisions in the x and y direction simultaneously. We use the same for-loop principles with moving as far as the player can, as in the directions separately, but combine them now. We check this in the move function to be sure it is executed.

## Main class

At first, we restarted the game by creating a new game object, and disposing the frame of the original game. We soon found, however, that the game started to lag quite a bit after restarting or going to the next level.

After some testing, we concluded that this happened because of the multiple game objects that existed together. To solve this problem, we changed the structure of our code, by introducing a “Main” class, which could function as a sort of constructor of the Game class. Now we could keep using one game instance, and reset the values on restarting, or going to the next level.

## Bugs

We had some issues with the JPanel, where the code runs too fast to dispose the frame and to create a new frame. To resolve this issue we added a sleep statement to resolve the issue. If the problem still persists, it is easily resolved by restarting the code.