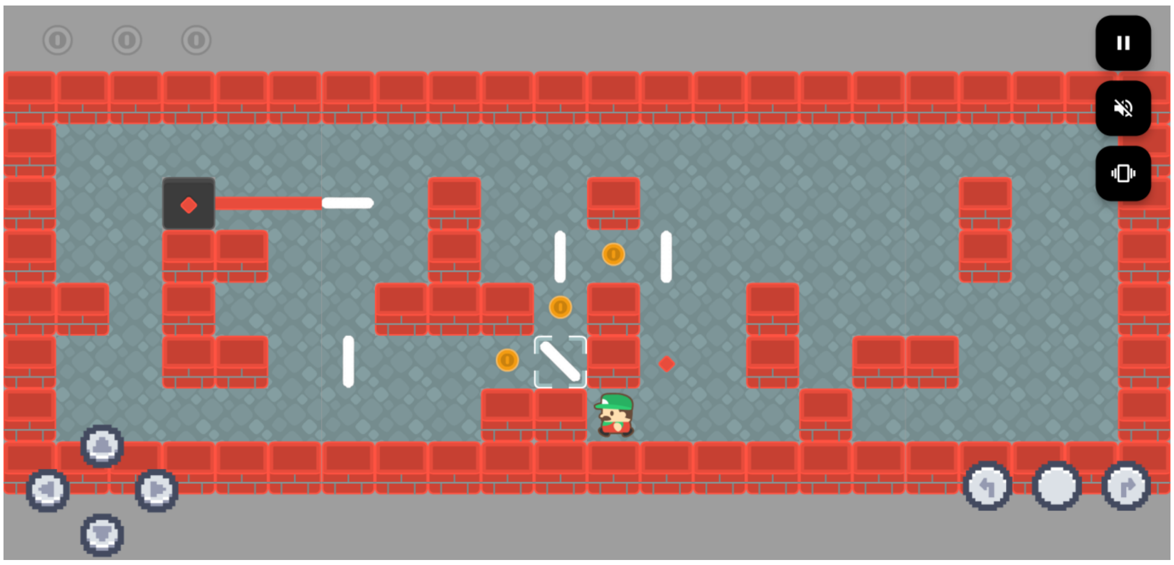
SMAGGHE Clément, PAQUET BOUSSARD Louis, DHAR Emon

Reflector quest



Object-oriented programming on mobile devices

INFO8011-1

2023-2024

Table of contents

Reflector Quest

Introduction

Reflector quest is a puzzle mobile game designed to challenge players with various levels. The application's name is indicative of its main objective at each level. Upon starting the app, users arrive on a home page with a few options: a prominent play button for accessing the level selection page, a settings button, and an about page.

Each level unfolds on a small map featuring lasers, coins, mirrors, and a player-controlled character. Players navigate through the game, moving around, pushing and turning mirrors strategically on the path of laser beams emanating from a light source, aiming to guide the laser beams to their endpoints. In addition to the primary goal, there is a side quest: collecting three coins across the map. Successfully collecting coins and completing the level results in achievements stored and displayed on the level selection page.

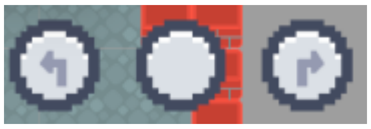
While playing a level, the player must avoid being struck by laser beams while moving around to collect coins or position mirrors. The laser beams not only have the potential to kill the player and end the game, but hitting coins with them results in instant destruction. The challenge is to navigate the map without dying from laser beams and to collect all coins without destroying them.

Reflector Quest offers the possibility to play on pre-designed levels or to play with randomly generated levels using a dynamic level generator. Choosing a random level from the list on the level selection page prompts the application to generate a unique, solvable level with everything on it: laser beams, three coins, a light source, laser endpoints, etc. This ensures that players can always enjoy new, fresh, and solvable challenges as they progress through the game.

Controls

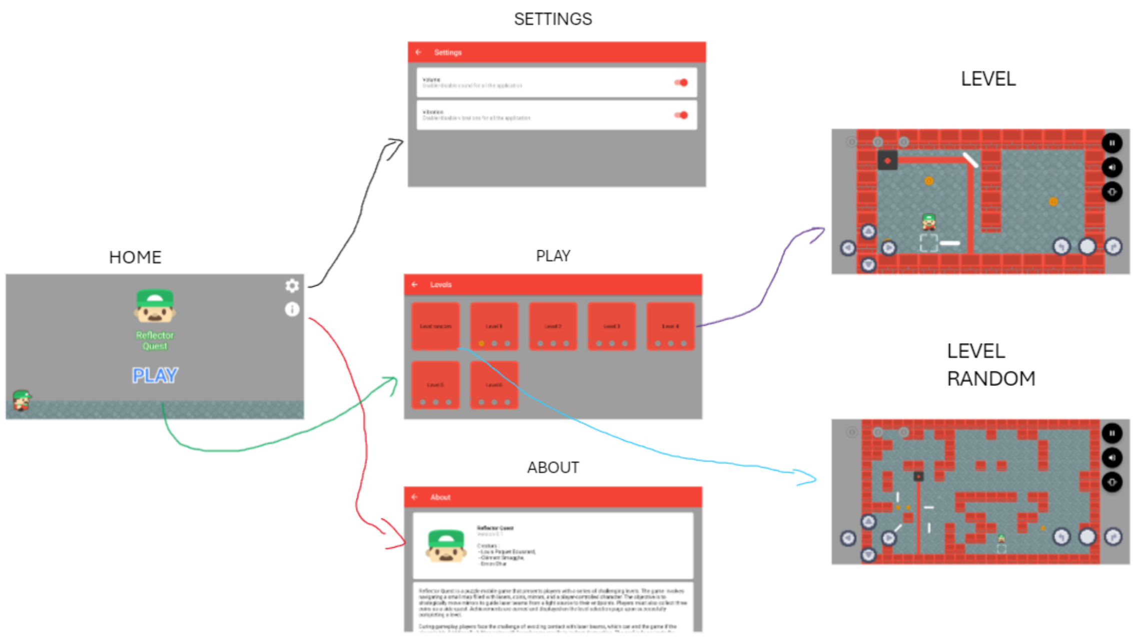
In Reflector Quest, during a game, the player can use various controls. The user interface incorporates a cross button for character movement, a sound and a vibration button to adjust these, and a pause button for pausing the game.

There are also three buttons related to the manipulation of the mirrors. The two buttons with an arrow on them allow the player to turn the mirrors. In the middle of these 3, the player can find a third button which is associated with changing the focus to another mirror near the player. We made this addition later in the creation of the game because without it, the game’s handling was much less enjoyable.

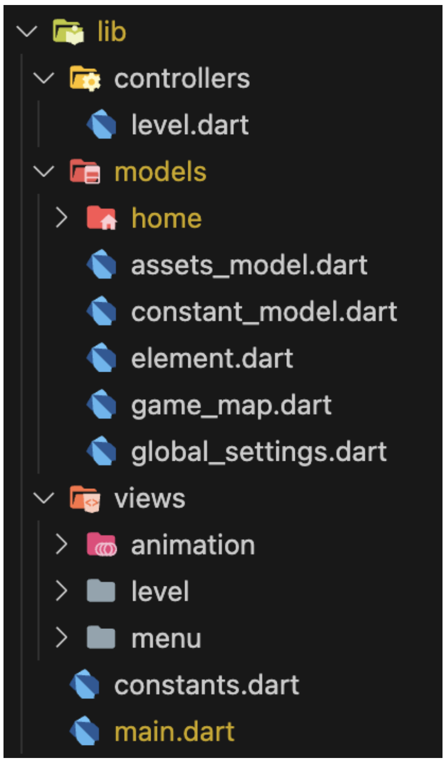


App Structure (routing)

Our application comprises five distinct views that either overlay or replace each other based on specific actions. Upon launching the app, users are directed to the "home" view, serving as the initial point and providing access to other views. Clicking on various buttons overlays additional views onto the home view, such as level selection, settings, and about. When users access the level selection page, they can choose from predefined levels or opt for a random level. Clicking on any level invokes the same view, replacing the previous ones instead of overlaying them. The level itself is either read from a file or generated randomly. Following a win or loss, the level view is replaced by the main menu.



Code Structure



```
lib
  controllers
    level.dart
  models
    home
    assets_model.dart
    constant_model.dart
    element.dart
    game_map.dart
    global_settings.dart
  views
    animation
    level
    menu
  constants.dart
  main.dart
```

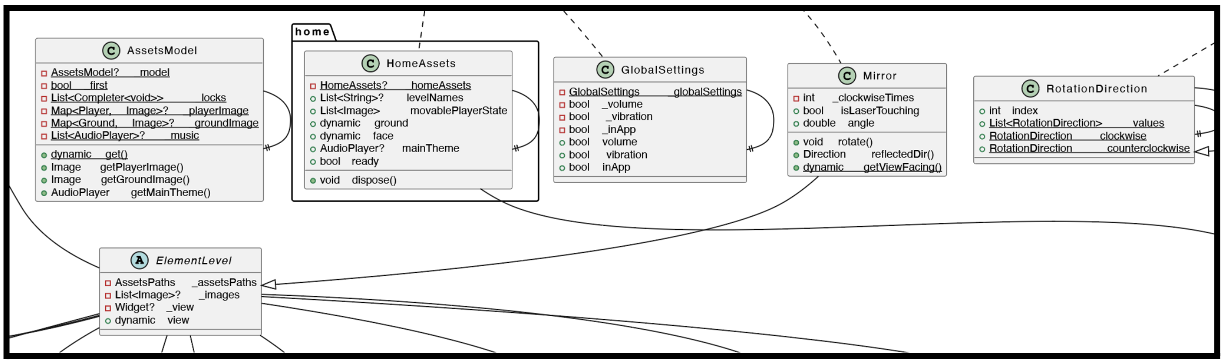
UML

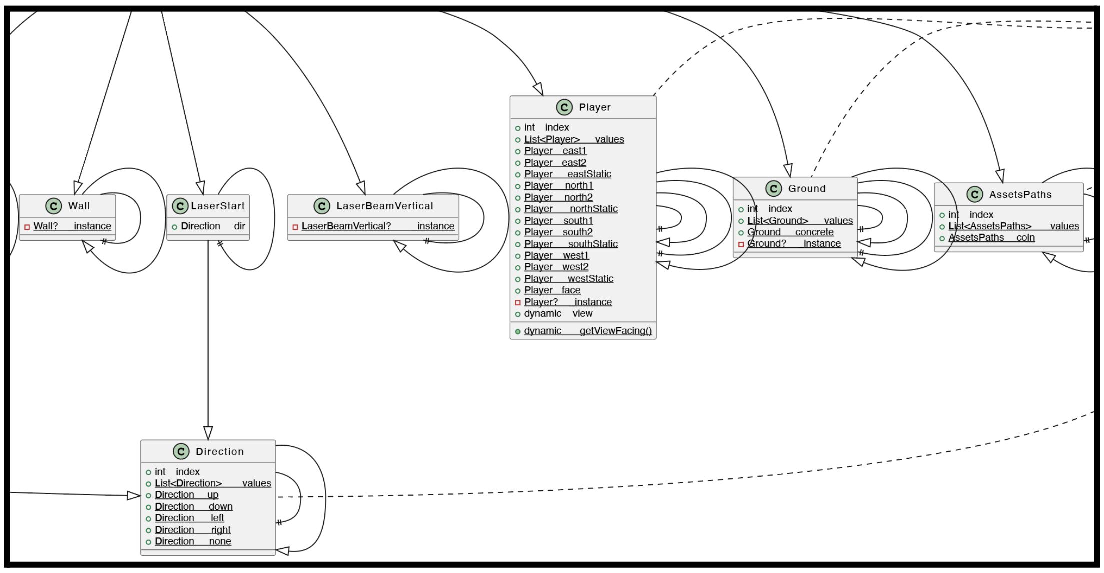
views
level
_GameButtonState
PlayPauseButton
LevelController controller
Widget build()
OverlayLevel
LevelController controller
Widget build()
HapticsButton
Widget build()
SoundButton
Widget build()
LevelView
int levelID
Widget build()
GameButton
ImageProvider<Object> icon
LevelController controller
void Function(TapDownDetails) tapDownFunction
State<StatefulWidget> createState()
menu
LevelSelection
Widget build()
About
Widget build()
Settings
Widget build()
Loading
Widget build()
Home
State<StatefulWidget> createState()
CommonPage
Widget content
String titlePage
Widget build()
Common
Widget content
Widget build()
animation
MovingPathAnimation
Image _ground
Duration duration
MovingPathAnimationState createState()
SpriteAnimation
List<Widget> listSprites
Duration duration
State<StatefulWidget> createState()
main.dart
MainApp
Widget build()
controllers
LevelController
GameMap map
bool isGameRunning
void movePlayer()
void rotateMirror()
void changeCursorPosition()
bool isGameFinished()
flutter
State
StatelessWidget
StatefulWidget
Widget
ChangeNotifier
dart
core
Enum
models
AssetsModel
AssetsModel? _model
bool _first
List<Completer<void>> _locks
Map<Player, Image>? _playerImage
Map<Ground, Image>? _groundImage
List<AudioPlayer>? _music
dynamic get()
Image getPlayerImage()
Image getGroundImage()
AudioPlayer getMainTheme()
home
HomeAssets
HomeAssets? _homeAssets
List<String>? levelNames
List<Image> movablePlayerState
dynamic ground
dynamic face
AudioPlayer? mainTheme
bool ready
void dispose()
GlobalSettings
GlobalSettings _globalSettings
bool _volume
bool _vibration
bool _inApp
bool volume
bool vibration
bool inApp
Mirror
int _clockwiseTimes
bool isLaserTouching
double angle
void rotate()
Direction reflectedDir()
dynamic getViewFacing()
RotationDirection
int index
List<RotationDirection> values
RotationDirection clockwise
RotationDirection counterclockwise
audioplayers
AudioPlayer
ElementLevel
AssetsPaths _assetsPaths
List<Image>? _images
Widget? _view
dynamic view
GameMap
GameMap? _map
Map<Position, ElementLevel> _levelMap
int _maxCoinsNbr
Direction playerFacing
int width
int height
int _levelID
int _reachedLaserEnds
Position _initialPlayerPosition
Position _playerLastPosition
List<Position> _mirrorsNeighborsOfPlayer
bool isReady
bool _isLose
bool _isWon
List<Position> initialMirrorsPosition
Map<Position, ElementLevel> levelMap
Position initialPlayerPosition
Position playerPosition
List<Position> mirrorsPositions
Position cursorCurrentPosition
Position cursorNextPosition
bool isWon
bool isLose
int pickedCoins
void notifyAllListeners()
int getPlayerFacingAsInt()
void refreshLasers()
Map<Position, ElementLevel> _generateRandomLevel()
LaserBeamHorizontal
LaserBeamHorizontal? _instance
dynamic view
LaserBeamCross
LaserBeamCross? _instance
dynamic view
LaserEnd
LaserEnd? _instance
Coin
Coin? _instance
Wall
Wall? _instance
LaserStart
Direction dir
LaserBeamVertical
LaserBeamVertical? _instance
Player
int index
List<Player> values
Player east1
Player east2
Player eastStatic
Player north1
Player north2
Player northStatic
Player south1
Player south2
Player southStatic
Player west1
Player west2
Player westStatic
Player face
Player? _instance
dynamic view
dynamic getViewFacing()
Ground
int index
List<Ground> values
Ground concrete
Ground? _instance
AssetsPaths
int index
List<AssetsPaths> values
AssetsPaths coin
Position
int x
int y
int hashCode
bool ==()
Position translate()
dynamic ()
bool isNeighborOf()
Direction
int index
List<Direction> values
Direction up
Direction down
Direction left
Direction right
Direction none

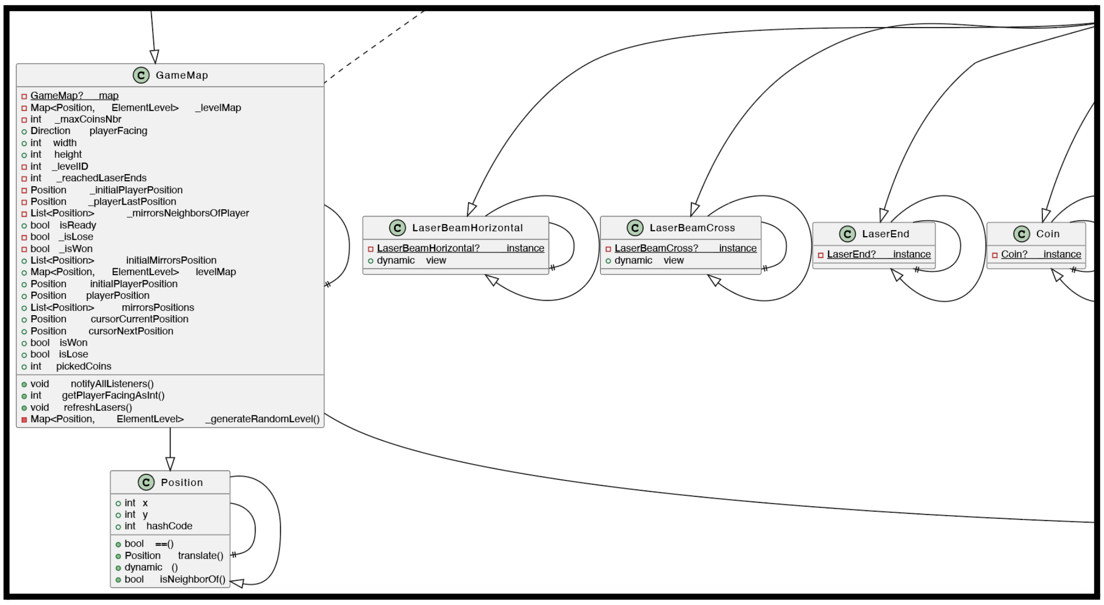
MVC

We have implemented the Model-View-Controller (MVC) architectural pattern. The MVC pattern helps to organize the code and separates the concerns of data manipulation, UI rendering, and user interactions.

Models:

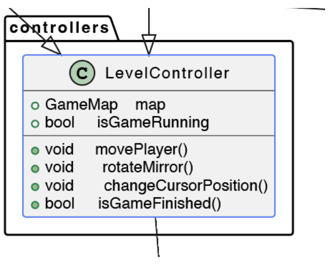




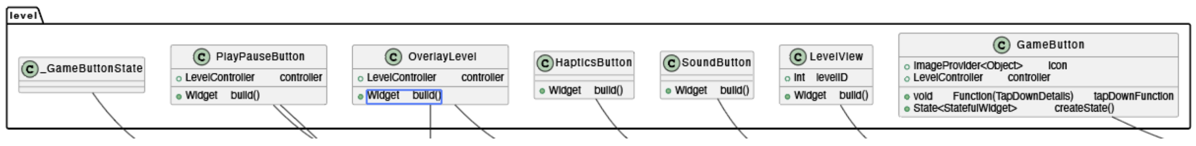


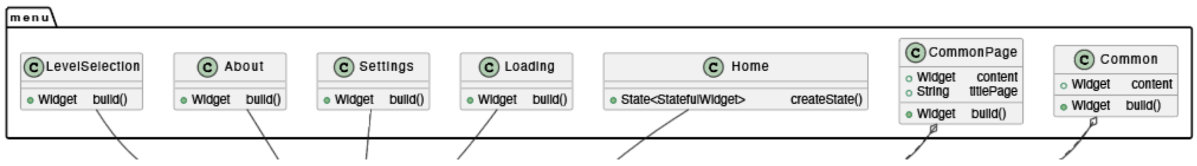
GameMap
GameMap? map
Map<Position, ElementLevel> _levelMap
int _maxCoinsNbr
Direction playerFacing
int width
int height
int _levelID
int _reachedLaserEnds
Position _initialPlayerPosition
Position _playerLastPosition
List<Position> _mirrorsNeighborsOfPlayer
bool isReady
bool _isLose
bool _isWon
List<Position> initialMirrorsPosition
Map<Position, ElementLevel> levelMap
Position initialPlayerPosition
Position playerPosition
List<Position> mirrorsPositions
Position cursorCurrentPosition
Position cursorNextPosition
bool isWon
bool isLose
int pickedCoins
void notifyAllListeners()
int getPlayerFacingAsInt()
void refreshLasers()
Map<Position, ElementLevel> _generateRandomLevel()
Position
int x
int y
int hashCode
bool ==()
Position translate()
dynamic ()
bool isNeighborOf()
LaserBeamHorizontal
LaserBeamHorizontal? instance
dynamic view
LaserBeamCross
LaserBeamCross? instance
dynamic view
LaserEnd
LaserEnd? instance
Coin
Coin? instance

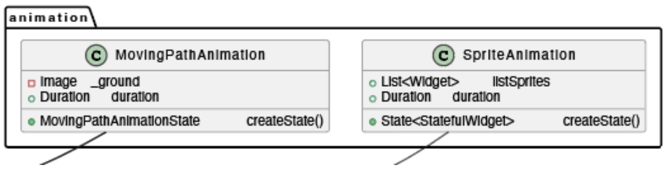
Controllers:



Views:


level
_GameButtonState
PlayPauseButton
LevelController controller
Widget build()
OverlayLevel
LevelController controller
Widget build()
HapticsButton
Widget build()
SoundButton
Widget build()
LevelView
int levelID
Widget build()
GameButton
ImageProvider<Object> icon
LevelController controller
void Function(TapDownDetails) tapDownFunction
State<StatefulWidget> createState()





Patterns

We have also used several design patterns in our project to improve code efficiency and maintainability. One of them is the singleton.
The singleton is a nice feature for the elements of the level, as they should only load the sprite asset *once*.
We also used the singleton pattern to ensure some security in the app. For example, the global settings class is a singleton since we do not want to have multiple instances of this class. The same thing applies to the GameMap, the Player classes and some more.

Another pattern we have used is the observer/provider.
We have implemented this pattern using the Provider package to check for changes on specific variables.
For instance, when a button is pressed, we notify the player controller to perform player movement.
We used this pattern on some features of the application. The “biggest” is the settings. We want the entire application to be able to access the parameters.

Technical challenges

MVC structure

Unfamiliar with the Model-View-Controller architecture pattern, we started by writing all our code in a few files. This allowed us to be faster (at the *beginning*), and we didn't have to do complicated git merges (with conflicts).
But, later, code structure refactoring was really needed, and added an extra cost (time) to fix.

Random level generator

The biggest challenge of this project was the level generator, because of all the constraints to satisfy: feasible, not too easy to solve, quite large map, …

It seemed quite straightforward at first: placing randomly several walls, then a laser start and the induced laser path.
The most difficult part was the laser (beam) path.

We make several choices during the level creation (walls/laser/coins/… positions), so in order to create a (feasible) level, we always loop through all the different possibilities (so we're guaranteed to get a result).

We came with a solution in several steps.

Walls:

Randomly placed walls (without any constraint) is not ideal, as it would result in a sparse map in which it would be difficult to navigate for the player.
Instead we create *aggregates* of walls: we choose random locations where we put several walls close to each other. This way feels more “natural” when we look at the result.
We also need to set a threshold for the number of walls to place, but that can be done by trial-error (and is more prone to subjectivity).

Each wall is only added to the map if it does not split the level in 2 distinct zones (or a zone would be inaccessible for the player). **This constraint is also applied to the other Elements (laser start, beam, end and mirrors).**

Laser start:

We simply place the laser at a position where it has a ground next to it (for the laser beam to be on - or a mirror).
For simplicity with the generator, we preferred to create a long single laser path rather than multiple smaller paths.

Laser beams, mirrors and laser end:

First we decide first the number of mirrors to use for the complete laser path.

Then we place a corner of the laser beam path (i.e. a mirror) between the laser start and a wall. We repeat the operation for each straight line of the laser path.
If we end up with an impossible placement - not enough space - we change the positions of corners. If it’s still not enough we repeat the same operation by changing the laser start position or decreasing the number of corners the laser path is composed of.

If a solution was found a laser end is just placed at the end of the laser path.

Now if we play the level directly, it will be automatically won, since the laser start is connected to the laser end.
Thus, we rotate all the mirrors at least once with a random angle.

Again, for ease, mirrors stay in their position. It would require far more tests and logic to be able to place the mirrors at random position and still create a feasible level (since we can push the mirrors).

Coins:

Coins can be placed randomly on any ground of the map.
But, if we want to “spice up” the level, we can put 2 coins on the laser path, so the user should pay attention before rotating the mirrors (or they might be burnt by the lasers).

Player:

No need for advanced logic, we can place the player on any remaining ground.

An example of a randomly generated level can be found at the cover page.

Problems, bugs, changes

What did not work correctly (and got fixed)

- The random level generator would only create a level with a rather short laser path (now, a longer one is generated, and 2 coins are placed on the path so it's more difficult for the player to collect the 3 coins).
- Player not killed when the mirrors are rotated and the updated laser touch it
- Superimposed lasers would never be display in the same direction (resulting in a display with laser crosses when a mirror would be placed on a lasers intersection)
- Mirror pushed by the player would keep its (laser) reflection sprite if it was touched by a laser (even if it no longer touches a laser)
- Coins touched by the lasers were effectively removed from the display, but it was the same as if the player collected them (while they should have burnt)
- When the game was over (whether the player died or succeeded), it could continue to play
- The level 2 did not have any coins in its map
- Buttons to rotate the mirrors were swapped
- The level 1 was sometimes solved automatically
- Home screen title was underlined when user finished a level

What still does not work (or still is not implemented)

- A mirror put at a lasers intersection will only show (in its sprite) a single side with a reflected laser (graphical bug - there is effectively a reflected laser on each side of the mirror, but it's not shown because of the limited sprite)
- Some rare assets (sprites/images) are not loaded asynchronously

As it can be seen, there were quite a lot of problems in the released application.
The majority of these are small bugs that can be fixed quickly.

Why all these problems?

We decided to be honest about everything.

We made significant mistakes, and we are committed to learning from them. With this new experience, we are determined not to repeat the same errors again. Recognizing our shortcomings is the first step towards improvement, and we will apply these lessons to future projects.

Addressing the issues in the released application required a thorough examination of both technical and organizational aspects. Bugs were identified and fixed, and features that were not working as intended were addressed. For example, we rectified instances where the player was not killed when mirrors were rotated and the updated laser touched them. Between other things, we also fixed the display of superimposed lasers, ensuring they were shown in the correct direction.

Some issues, however, still persist or have not been fully implemented. For instance, a graphical bug prevents mirrors placed at laser intersections from displaying both sides with reflected lasers. We also acknowledge that certain assets, such as sprites and images, are not loaded asynchronously, impacting the overall performance.

Let’s look at the root causes of these problems: we were at first slow to produce code, since it was the first time we had to “play” with Flutter and Dart (programming languages we did not know back then). Learning asynchronous loads posed a challenge for some team members who hadn't encountered this concept before. Additionally, there was a period where the project was put on the back burner, leading to a lack of consistent progress (we still worked on it from time to time, but not enough). We underestimated the time required and only fully grasped the urgency near the deadline.

In the end, we devoted focused, full-time effort to address the issues and meet the submission deadline. However, we acknowledge that our approach was far from good. Instead of building and refining the project progressively, we found ourselves adding content at the last minute, resulting in a working application with numerous bugs.

The key takeaway from this experience is our realization of the importance of organization and planning in the development process. We are committed to applying these lessons to future projects, ensuring a more successful outcome.
Transparency is crucial, and we believe that by openly acknowledging our mistakes, we can make improvements within our team.