Part III: Maze Game (180p)

Part 3 of the exam is programming tasks. This section contains **13 sub-assignments** that together give a maximum score of approx. 180 points and counts approx. **60%** of the exam. Each task carries a maximum score of **5 - 20 points**, depending on difficulty and workload.

The handout code contains compilable (and executable) *.cpp* and *.h* files with pre-coded parts and a complete description of the exercises in Part 3 as a PDF. After downloading the handout code, you are free to use a development environment (VS Code) to work on the sub-assignments.

On the next page, you can download the *.zip* file if it is **NOT** working to access the handout code through the TDT4102 Extension, and later upload the new *.zip* file, including your own code.

Check List for Technical Issues

If the project with the handout code does not compile, you should ask for help. While you are waiting, try the following:

1. Check that the project folder is created in the folder `C:\temp`.
2. Check that the folder name does **NOT** include Norwegian letters (*Æ, Ø, Å*) or space, as this can cause problems when running the project in VS Code.
3. If it is **NOT** working to access the handout code through the TDT4102 Extension, you can download the zip file from Inspera:
   - Download the *.zip* file from Inspera and save the unzipped folder in `C:\temp` as **eksamenTDT4102_candidateNumber**. You should write *your own candidate number* behind the underline.
   - Open the folder in VS Code. Use the following TDT4102-commands to create a working project:
     (a) `Ctrl+Shift+P` → `TDT4102: Force refresh of the course content`
     (b) `Ctrl+Shift+P` → `TDT4102: Create project from TDT4102 template` → `Configuration only`
4. Assure you are inside the correct file in VS Code.
5. Run the following TDT4102-commands:
   (a) `Ctrl+Shift+P` → `TDT4102: Force refresh of the course content`
   (b) `Ctrl+Shift+P` → `TDT4102: Create project from TDT4102 template` → `Configuration only`
6. Try running the code again (`Ctrl+F5` or `Fn+F5`).
7. If it still does not work, try to close the VS Code window and open it again. Then repeat steps 5-6.

Introduction

This Maze Game is a simple game where the player is tasked with finding keys, unlocking doors, and making their way towards the exit. The handout code is lacking a lot of functionality. The internal gameplay logic is built on simple cells in a regular 2D grid. In the zip-file you will find the code skeleton for the maze game, with tasks clearly marked.

Before starting, check that the unmodified handout code runs without problems. You should see the same window as in Figure 1

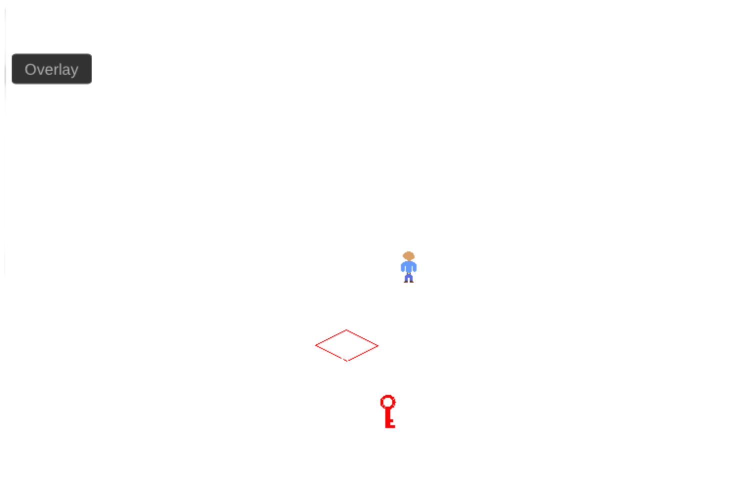

Figure 1: Screenshot of the application as it will appear when you run the unmodified handout code.

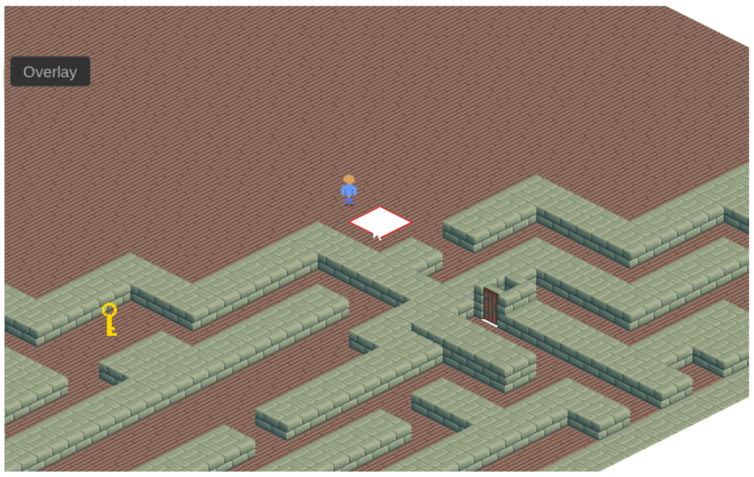

Figure 2: Screenshot of the complete application.

Application overview

As depicted in the initial state of the game world (figure 1), there is little visual activity. Only minimal graphic and functionality has been implemented. However, when the game is fully implemented (2), it becomes playable with graphics and logic.

At any time, it is possible to use the "Overlay" button to obtain a visual representation of the cells on which the player can move. Prior to implementing the logic, all cells will be filled with a red color. After the logic is implemented, the cells will be colored either red or green depending on whether the cell is blocked or open for movement, respectively (figure 3).

Maze Game Setup

The state of the player and the world is handled through the `main.cpp` file. Do **NOT** edit this file. The internal coordinate system is a regular 2D grid. Given a level of dimensions $w \times h$, the cell at position (i, j) is at the position $idx = j \times w + i$ (See figure 4).

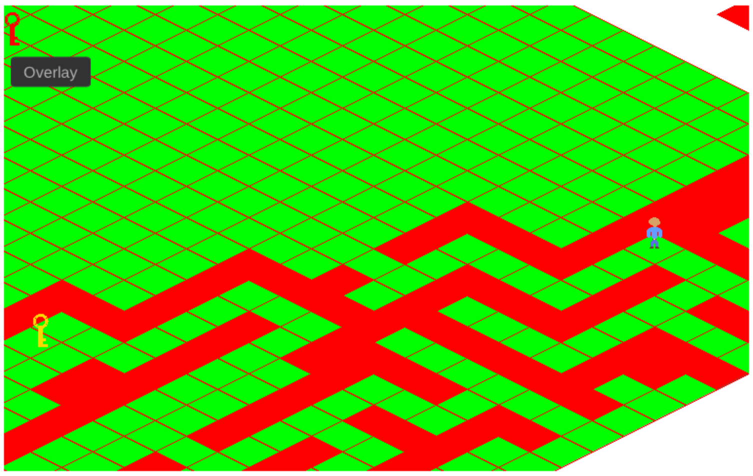

Figure 3: The overlay when the walkable logic is implemented.

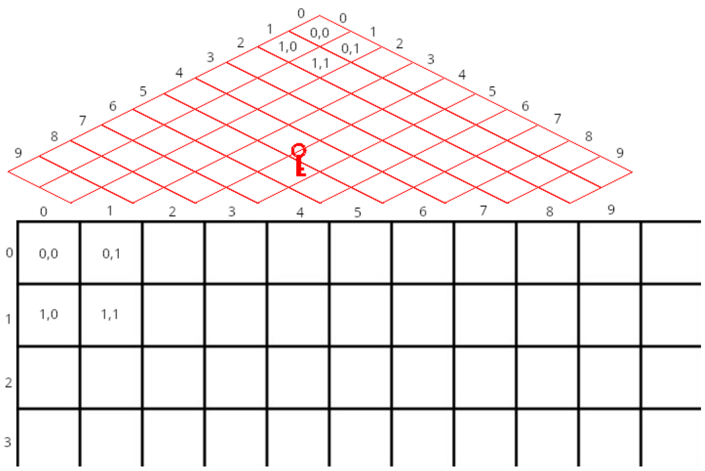

Figure 4: How the visual grid corresponds to the internal grid.

How to solve Part 3

Each task in part 3 has an associated unique code to make it easier to find where to write the answer. The code is in the format `<T><digit>` (*TS*), where the digits are between 1 and 13 (*T1 - T13*). In `Tasks.cpp`, you will find two comments that define respectively the beginning and the end of the code you will enter for each task,. The comments are in the format: `// BEGIN: TS` and `// END: TS`.

It is essential that all your answers are written between such comment pairs, to support our censorship mechanics. If there is already some code written between the `BEGIN` and `END` comments in the files you have been given, then you can, and often should, replace that code with your own implementation unless otherwise specified in the sub-assignment description. All code that is *outside* the `BEGIN`- and `END` comments **SHOULD** be unmodified.

```
bool Level::is_walkable(const TDT4102::Point coordinate) const
{
    // BEGIN: T1
    // Write your answer to assignment T1 here, between the // BEGIN: T1
    // and // END: T1 comments. You should remove any code that is
    // already there and replace it with your own.

    // END: T1
}
```

After you have implemented your solution, you should end up with the following instead:

```
bool Level::is_walkable(const TDT4102::Point coordinate) const
{
    // BEGIN: T1
    // Write your answer to assignment T1 here, between the // BEGIN: T1
    // and // END: T1 comments. You should remove any code that is
    // already there and replace it with your own.

    /* Din kode her / Your code here */

    // END: T1
}
```

Note that the `BEGIN` and `END` comments should **NOT** be removed.

The sub-assignments

Representing the Level (30 Points)

A world is represented by the `Level` class. The declaration file for this class can be found in `Tasks.h`. The class has several functions to retrieve and modify the state of the world. In this section, you should only implement functions that allow external classes to retrieve state information from the world.

```
class Level {
public:
    /* ... Constructors ... */
    bool is_walkable(const TDT4102::Point coordinate) const;
    int tile_at(const TDT4102::Point coordinate) const;
    void set_tile_at(const TDT4102::Point coordinate, const int tile);
    void set_walkable_at(const TDT4102::Point coordinate, const bool walkable);

    unsigned int get_width() const noexcept;
    unsigned int get_height() const noexcept;

private:
    unsigned int width = 1;
    unsigned int height = 1;

    std::vector<int> tiles = {0};
    std::vector<bool> walkable = {false};

    // ...
};
```

Note the instances of `std::vector` for `tiles` and `walkable`. These are filled with information about which tiles are present in each cell and whether the cell is walkable respectively.

1. (5 points) **T1: Getting the width**

   Implement the code for getting the private `width` field in the `Level` class.

2. (5 points) **T2: Getting the height**

   Implement the code for getting the private `height` field in the `Level` class.

3. (10 points) **T3: Check whether a cell is walkable**

   Determine whether the cell in question is walkable, given the information available in the `walkable` field, and a `Point` parameter named `coordinate`. The coordinate parameter will always follow the internal coordinate system as seen in figure 4. The return value of the function should be a boolean representing whether the cell is walkable.

4. (10 points) **T4: Identify which tile occupies a cell**

   Determine which type tile occupies the the cell in question, given the information available in the `tiles` field, and the `Point` parameter named `coordinate`. The return value of the function should be an integer representing the tile's ID.

   Note! The code you write here will not be visually testable until assignment *T6* is completed.

Drawing the tiles (25 points)

The `Tile` class

```
struct Tile
{
    Tile(int id, bool walkable, const std::filesystem::path tile_image_path);

    Tile(const Tile &other);
    Tile &operator=(const Tile &other);

    Tile(Tile &&rhs);
    Tile &operator=(Tile &&rhs);

    bool has_image() const noexcept;

    int id;
    bool walkable;
    std::shared_ptr<TDT4102::Image> image;
};
```

The `Context` class

The `Context` class is an aggregate class providing access to the `AnimationWindow` and `Camera` that are used in, e.g., rendering contexts.The methods provided are `getWindow()` and `getCamera()` that each return references to `AnimationWindow` and `Camera` respectively.

5. (10 points) **T5: `Tile` copy constructor**

   In order to draw tiles, we need to be able to pass copies of the tile around such that the drawing logic gets proper instances of `Tile`. The copy constructor fails to do its job. The fields of the `Tile` class should be copied from one instance to the newly constructed one, otherwise all tiles that are copy-constructed will stay default initialized. In this assignment, you should add to the copy constructor such that the fields are properly copied.

   Note! The code you write here will not be visually testable until assignment *T6* is completed.

6. (15 points) **T6: Drawing tiles**

   Implement the code for drawing a tile at the location given by the `anchor` parameter. This gives a coordinate on the **screen** and not in the internal grid. The `AnimationWindow` instance to use must be fetched from the `Context` class. If there is no image contained in the `Tile` argument, do nothing.

   Note: Only one type of tile is available until question *T11* is done. Use the overlay to visualize the logic.

Moving stuff around (55 points)

A game without some sense of motion or motivation is a boring one. The Maze Game at this point should have graphics and might look nice, but there is little action going on. In this group of sub-assignments you will be tasked with ensuring the player can move to walkable cells and allowing the player to move the camera.

7. (15 points) **T7: Restricting player movement**

   Implement the code for verifying a proposed player movement. The current player position can be read from the `position` field in the `PlayerControllable` class. Here is a list of requirements for a movement to be valid:

   - The new position is within the bounds of the level ($0 \leq x \leq w$ and $0 \leq y \leq h$).
   - The player is not moving a distance farther than 1 unit ($|x1 - x0| \leq 1$).
   - The target cell is walkable.

   The return value should be a boolean value denoting whether the movement is valid.

8. (20 points) **T8: Moving the camera**

   Up until this point you've been staring at the same region of the level. We would like the camera to be functional so that we can pan the camera around. Using the `Context` object, get references to the instances of `AnimationWindow` and `Camera` and use them to **a)** get keyboard input and **b)** move the camera based on said input. A description of how the function should work follows:

   Table 1: The keyboard keys used to pan the camera and how they should affect the camera.

Key	Effect
W	Translate the camera by `-speed` in the Y direction
S	Translate the camera by `+speed` in the Y direction
A	Translate the camera by `-speed` in the X direction
D	Translate the camera by `+speed` in the X direction

9. (20 points) **T9: Moving the player**

   The player can already move by clicking the left mouse. Clicking one cell at a time to move the player character can get tedious. Use the `AnimationWindow` argument to **a)** get keyboard input and **b)** move the player character based on said input. A description of how the function should work follows:

   Make sure to verify whether the move is valid before moving the player with the `moveTo` method. Both `canMoveTo` and `moveTo` expect a coordinate and a `Level` instance

Table 2: The keyboard keys used to pan the camera and how they should affect the camera.

Key	Effect
ARROW_UP	Move the player by -1 in the Y direction
ARROW_DOWN	Move the player by +1 in the Y direction
ARROW_LEFT	Move the player by -1 in the X direction
ARROW_RIGHT	Move the player by +1 in the X direction

Input / Output (70 points)

Thus far you've been working with a static level that is embedded within the binary as well as only one tile type. Ideally, for a game such as this, we want variety. In this group of assignments you will implement some of the functionality of a level loader and tile loader.

10. (15 points) **T10: Overloading the extraction operator for tiles**

    As part of the tile loader, the extraction operator provides the means of directing an input stream directly to a `TileDescriptor`. The `TileDescriptor`'s main concern is to provide a description of a loadable tile to the loading system. The tile descriptor file is described in the end of this document (p. 8).

    Write the overloaded operator such that the `TileDescriptor` instance is populated with the data as seen in the file type description.

```
struct TileDescriptor {
    int id;
    std::string filename;
    bool walkable;
};
```

11. (15 points) **T11: Reading a single tile descriptor**

    When loading tiles, the tile loader will read the descriptor file line by line.

    Write the overloaded operator such that the `TileDescriptor` instance is populated with the data as given in the file type description.

```
struct TileDescriptor {
    int id;
    std::string filename;
    bool walkable;
};
```

12. (20 points) **T12: Level Loading**

    Before starting this assignment, make yourself familiar with the Level File Structure in the end of this document (p. 8).

    The function you are working with in this assignment takes a file path to a file that describes a level. Its result should be a `Level` object that is constructed and filled with the data present in said file. If the function fails to load a level, it should throw a runtime error such as, e.g., *"Could not load [filename]"*.

13. (20 points) **T13: Comparison operator for coordinates**

    Implement the comparison operator for the `Point` class.

Level File Structure

A level is storred with the following file format:

- The first line contains the width and height of the level, separated by a tab character (`\t`).
- Following the dimensions is a block of WI × HE integers, each denoting a tile ID.
- The block is ended by a single line that says "END".
- The same block layout is used for the walkable status of each cell. A cell is walkable if the file says 1 and impassable if it is 0.

```
WI  HE
ID  ID  ID  ID  ID  ID  ID
ID  ID  ID  ID  ID  ID  ID
ID  ID  ID  ID  ID  ID  ID
ID  ID  ID  ID  ID  ID  ID
ID  ID  ID  ID  ID  ID  ID
END
WA  WA  WA  WA  WA  WA  WA
WA  WA  WA  WA  WA  WA  WA
WA  WA  WA  WA  WA  WA  WA
WA  WA  WA  WA  WA  WA  WA
WA  WA  WA  WA  WA  WA  WA
END
```

Tile Descriptor File Structure

Each line of the tile descriptor file follows this structure:

```
ID    IMAGE_PATH    WALKABLE
```

```
 0    house_00.png      0
 1    house_01.png      0
 2    house_02.png      1
 3    house_03.png      1
 4    house_04.png      1
10    house_10.png      0
20    house_20.png      0
28    house_28.png      0
...
```