Part III: Level Editor

Part 3 of the exam is programming tasks. This section contains **13 sub-assignment**s 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 level editor is a simple editor where the user can make a maze for a maze game. The internal logic is built on simple cells in a regular 2D grid. The handout code is lacking a lot of functionality to make the editor fully featured, or even visually pleasing. 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

Application overview

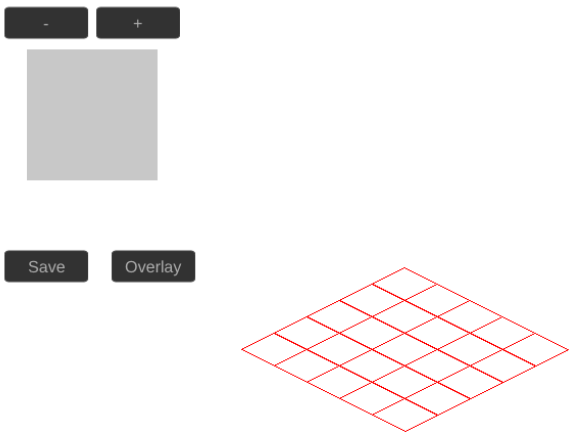

Figure 1: Screenshot of the application as it will appear when you run the unmodified handout code.

Figure 2: Screenshot of the complete application.

As depicted in the initial state of the game editor (figure 1), there is little visual activity. Only minimal graphic and functionality has been implemented. However, when the application is fully implemented (2), it becomes fully featured 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. The cells are colored either red or green depending on whether the cell is blocked or open for movement, respectively (figure 3).

Editor 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).

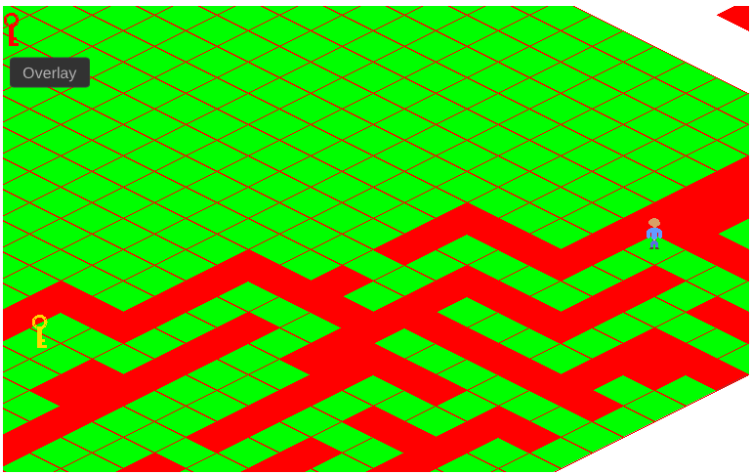

Figure 3: The overlay when the walkable logic is implemented.

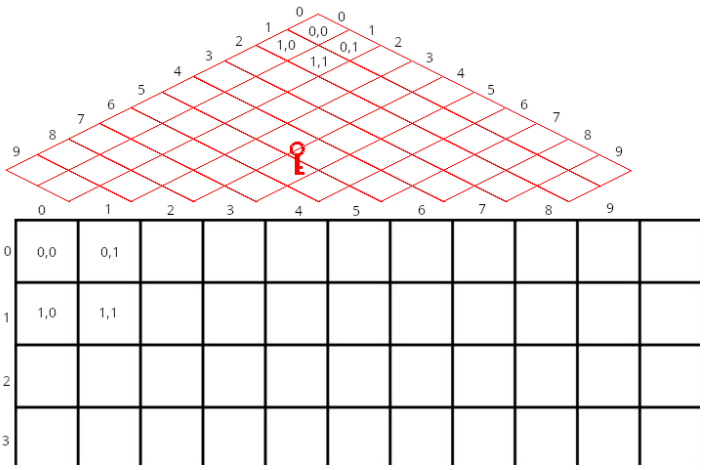

Figure 4: How the visual grid corresponds to the internal grid.

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`, for each task, you will find two comments that define respectively the beginning and the end of the code you will enter. 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 `Level.hpp`. 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: Checking whether a Tile has an image**

   The `Tile` class has a field named `image`. It is a shared pointer to an instance of `TDT4102::Image`.

   Given the shared pointer, write a function that return a boolean value depending on whether the shared pointer is valid.

4. (10 points) **T4: `Region` constructor**

   The purpose of the Region struct is to represent a multi-tile area. It has two fields: `begin` and end, both of which are instances of `TDT4102::Point`. The constructor should assign the lowest x- and y-values among the two points to the `begin` field, and the highest x- and y-values to the `end` field.

   Write an implementation of this constructor.

Fixing the Camera (20 Points)

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. (20 points) **T5: 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

Placing Tiles (60 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;
};
```

In addition to the declaration of the `Tile` struct, you should be aware of the following two auxiliary methods for rendering images and quads:

- `TileRenderer::render(Context &ctx, const Tile &tile, TDT4102::Point anchor)`
  - Draws a tile at the screen coordinates specified by `anchor`
- `QuadRenderer::render(Context &ctx, TDT4102::Point anchor, TDT4102::Color color)`
  - Draws a quad at the screen coordinates specified by `anchor` with the specified color

Note: To avoid program crashes when clicking outside the level, make sure the function does not try to write outside the bounds of the vector.

6. (10 points) **T6: Setting one tile**

   The coordinate conforms to the internal coordinate representation as shown in the end of this documnt (p. 8). A coordinate (x, y) in a world that is $W \times H$ corresponds to index $y \times W + x$.

   Modify the function that takes in a coordinate and a tile ID so a selected tile should occupy the clicked cell. Write the function so that the `tiles` vector in the `Level` instance reflects the change of tiles.

7. (15 points) **T7: Setting a cell to be walkable**

   Write the function so that the `walkable` vector in the `Level` set the walkable flag for a selected cell.

8. (15 points) **T8: Placement overlay**

   The placement overlay should already be visible. The tile following your cursor is the placement overlay. When you hold the left mouse button, however, you will notice that the tile stays behind. This is because the placement overlay is supposed to show you the region you have selected.

   Extend the function to render the same tile over the region spanned by the two fields, `begin` and `end`.

9. (20 points) **T9: Updating a region**

   Given the two points, `begin` and `end`, apply the previously implemented `set_tile_at` to the entire region the two points span.

Operator overloading and I/O (70 Points)

The `TileDescripto` class

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

10. (15 points) **T10: Copy Assignment Operator**

    Write the copy assignment operator such that all fields in `Tile` are copied from an existing instance.

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

    Write the function `process_line` that should take a string and construct a `TileDescriptor` from it. The structure of a tile descriptor line can be seen in the end of this document (p. 8).

12. (20 points) **T12: Dynamic Loading of Tiles**

    In the `tiles/` directory, there is a file named `tile_info`. It describes a set of tiles by their IDs, image names, and walkable flags.

    Write the function so that it opens a file stream, reads the file line by line, processes each line, constructs a `Tile`, and adds it to the image pool. If the function fails to open the file, throw a runtime error signalling this.

13. (20 points) **T13: Saving the level**

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

    Implement a save function for a `Level` instance by open a file stream, throw an error if the file failed to open, write the width and height header, write the tile ID block, write the walkable flag block.

    NOTE: The beginning of each line is always an alphanumeric character, i.e., there should not be a tab at the beginning of a line.

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 $\times$ 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
...
```