HAND IN MODULE 2

MARIUS HAAKONSEN, OLE K LARSEN

1. Task 1:

Exercise A:
The both of us have already played Tetris, so we'll skip this part.

Exercise B:

Implementing the function void Board::reduce() to remove the lines when completed.

Looping over row number i, from top to bottom.

```
void Board::reduce() {
    for(int i = 3; i < 19; i++) {
```

Defining variables to use while looping over j number of columns.

```
        int count = 0;
        int tilecount = 0;
        for(int j = 1; j < 11; j++) {
            if (tiles[j][i] != sf::Color::Black) {
                count++;
```

If all tiles in row number 'i' is inequal to the color black, the program loops from that row and upwards, setting the current row to be equal to the row above, giving the impression that the rows "falls down".

```
                if(count == 10) {
                    tilecount = i;
                    for(int k = tilecount; k >= 3; k--) {
                        for(int j = 1; j < 11; j++) {
                            tiles[j][k] = tiles[j][k-1];
                        }
                    }
                    break;
                }
```

As a little coding-exercise from out part, we also added the functionality of pressing spacebar to make the shapes move all the way down until it intersects with other shapes. We also added a score.

Exercise C:

To solve this problem we created a text called "polyminos.txt" and created a filestream to read from it. This file will be available for users to create their own polyminos. On the very first line of the txt file is the number of shapes in the file. The ifstream starts of with reading this number into a global called nrOfShapes, which is used in the random selection of shapes in the file.

Here's an example of one of the shapes in the .txt file.

```
0 0 0 0 0 0
0 0 1 0 0 0
0 0 1 1 0 0
0 0 1 0 0 0
0 0 1 1 0 0
0 0 0 0 0 0
```

2. Task 2:

Exercise A:

How was this puzzle created?
Puzzling.stackexchange.com was utilized to get the correct specifications of the puzzle and inspiration for the statements to be made by the three people in the encounter.

Knight: Always tells the truth.
Knave: Always tells a lie.
Spy: Tells either the truth or a lie.

The puzzle involves encountering three different people, person A, B and C.

They all have their own statements:

A: "C is the knight."
B: "A is the knight."
C: "B is the knave."

Who is the knight, who is the knave, and who is the spy among the three?

To reach the solution of this, the statements are put in a table.

Knight, Knave and Spy

Combinations A B C	Is it true? A B C	Would it be said? A B C	Solution A B C
0 1 x	0 0 0	1 0 x	
1 0 x	0 1 1	0 0 x	
0 x 1	1 0 0	0 x 0	
1 x 1	0 1 0	0 x 1	
x 0 1	1 0 1	x 1 1	1
x 1 0	0 0 0	x 0 0	

The solution is found by asking the two following questions: 'Is it true?' and 'Would it be said?'.

Based on the combinations of these columns, and based on whether or not the person actually tells the truth or lies, we find the correct combination of knight knave and spy. The next to last row is the only possible solution, as shown by the 'x 1 1' outcome. This means that C is the knight, B is the knace and A is the spy.

Exercise B:
Reformulated statements, giving the same answer:

A: "C may tell the truth."
B: "C is not a knight."
C: "B may tell a lie."

Knight, Knave and Spy			
Combinations A B C	Is it true? A B C	Would it be said? A B C	Solution A B C
0 1 x	1 1 0	0 1 x	
1 0 x	1 1 1	1 0 x	
0 x 1	1 0 1	0 x 1	
1 x 1	0 1 1	0 x 0	
x 0 1	1 0 1	x 1 1	1
x 1 0	0 1 0	x 1 0	

3. Task 3:

The forums at Stackoverflow.com was used for the solution to this task.

References

[1] https://stackoverflow.com/questions/40702099/solve-sudoku-with-backtracking-c

The logic for checking each individual 3x3 block was found by looking at 'the 3x3 check' in source, and the logic for the 'cleared()' function was found by utilizing the 'find()' function in the source. In the Basic Programming Course the purpose of nested functions was adviced, which is the reasoning for the program setup.

The program is written entirely using C++, as we are most familiar with this syntax. In addition it was limited time from the lecture to when a complete implementation using GeCode would have had to be finished, this also made it more practical to use C++.

Exercise A:

in 'main()' the function 'solved()' is called on the grid with an unknown number of spots set to 0. 'clear()' returns solved as true if the position != 0.
'isSafe()' utilizes three functions to make sure the placement of the number i (in the loop) is not already placed in the relevant row, column and block. A block is 1 of 9 3x3 sized grids within the 9x9 grid.

```
if (!clear(grid, r, c)) { return true; }
for (int i = 1; i <= 9; i++) {
if (isSafe(grid, r, c, i)) { grid[r][c] = i;
```

```
  if (solve(grid)) { return true; }
  grid[r][c] = 0;
  }
```

Exercise B:

To create a new puzzle, a function that removes an unknown amount of numbers is cast on the completed grid inside 'main()'. A random number engine is used to create a number between 0 and 2. The function loops over every row and coloumn, and if the random number equals 0, then the position is set to 0.
In the 'main()' part of the program a solved puzzle is set to be the grid, giving the solver only one unique solution to solve, after 1/3 of the spots are set to 0.

```
std::uniform_int_distribution<> dist(0,3);
x = dist(generator);
if (x==0) { grid[r][c] = 0; }
```

4. Code Appendix:

Task 1 main program:

```
#include <SFML/Graphics.hpp>
#include <cmath>
#include <cstdlib>
#include <iostream>
#include <string>
#include <fstream>


using namespace std;

//
// Tetris
//

int nrOfShapes;

class Shape
{
public:
    sf::Color tiles[6][6];
    sf::Vector2i pos;

    // times since last downward movement
    float time;

    Shape();
    // reinitialise the shape: move to top and change shape
    void init();

    // move downwards once per second
    void update(float dt);

    // render the shape
    void draw(sf::RenderWindow& w);

    // rotate the shape
    void rotateLeft();
    void rotateRight();
};

void Shape::rotateLeft()
{
    sf::Color tmp[6][6];
    for (int i = 0; i < 6; i++) {
        for (int j = 0; j < 6; j++) {
            tmp[i][j] = tiles[j][5 - i];
        }
    }
    for (int i = 0; i < 6; i++) {
        for (int j = 0; j < 6; j++) {
            tiles[i][j] = tmp[i][j];
        }
    }
}

void Shape::rotateRight()
{
    rotateLeft();
    rotateLeft();
    rotateLeft();
}

Shape::Shape()
{
    init();
}

void Shape::init()
{
    // move to top and reset timer
    pos.y = 0;
    pos.x = 4;
    time = 0.0f;

    // fill with black tiles
    for (int i = 0; i < 6; i++) {
        for (int j = 0; j < 6; j++) {
            tiles[i][j] = sf::Color::Black;
        }
    }

    ifstream in("polyminos.txt");
    in >> nrOfShapes; in.ignore();
    int nr; char buf[16];
    int skip = (rand() % nrOfShapes);
    while (skip > 0) {
    for (int s = 0; s < 7; s++)
            in.getline(buf, 16);
        skip -= 1;
    }
    for (int i = 0; i < 6; i++) {
        for (int j = 0; j < 6; j++) {
            in >> nr;
            if (nr == 1)
                tiles[j][i] = sf::Color::Blue;
        }
    }
    in.close();

}


void Shape::update(float dt)
{
    time += dt;
    if (time > 1) {
        time = 0;
        pos.y += 1;
    }
}

void Shape::draw(sf::RenderWindow& w)
{
    sf::CircleShape s;
    s.setRadius(8);
    s.setOrigin(8, 8);
    for (int i = 0; i < 6; i++) {
        for (int j = 0; j < 6; j++) {
            if (tiles[i][j] != sf::Color::Black) {
                s.setFillColor(tiles[i][j]);
                s.setPosition(sf::Vector2f(pos.x * 16 + 16 * i + 100, pos.y * 16 + 16 * j + 10
                w.draw(s);
            }
        }
    }
```

```
}

class Board
{
public:
    sf::Color tiles[12][20];

    Board();

    // add a shape to the board
    void add(Shape& shape);

    // check if a shape intersects with the board
    bool intersect(Shape& shape);

    // remove full lines - should be implemented by you
    void reduce(int & score1, Board & board);

    // render board
    void draw(sf::RenderWindow& w);
};

void Board::reduce(int & score1, Board & board)
{
    int k = 20 - 1;
    for (int i = 20; i > 0; i--) {
        int count = 0;
        for (int j = 1; j < 12; j++) {
            if (board.tiles[j][i] != sf::Color::Black && board.tiles[j][i] != sf::Color::Red)
                count++;
            board.tiles[k][j] = board.tiles[i][j];
        }
        if (count == 10) {
            for (int d = 1; d < 11; d++)
                board.tiles[d][i] = sf::Color::Black;
            for (int f = i; f > 0; f--)
                for (int w = 1; w < 11; w++)
                    board.tiles[w][f] = board.tiles[w][f - 1];
            score1 += 10;
        }
    }
}

bool Board::intersect(Shape& shape)
{
    bool intersect = false;
    for (int i = 0; i < 6; i++) {
        for (int j = 0; j < 6; j++) {
            if (tiles[i + shape.pos.x][j + shape.pos.y] != sf::Color::Black &&
                shape.tiles[i][j] != sf::Color::Black)
                intersect = true;
        }
    }
    return intersect;
}

void Board::draw(sf::RenderWindow& w)
{
    sf::CircleShape s;
    s.setRadius(8);
    s.setOrigin(8, 8);
    for (int i = 0; i < 12; i++) {
        for (int j = 0; j < 20; j++) {
            s.setFillColor(tiles[i][j]);
            s.setPosition(sf::Vector2f(16 * i + 100, 16 * j + 100));
            w.draw(s);
        }
    }
}

void Board::add(Shape& shape)
{
    for (int i = 0; i < 6; i++) {
        for (int j = 0; j < 6; j++) {

            if (shape.tiles[i][j] != sf::Color::Black) {
                tiles[i + shape.pos.x][j + shape.pos.y] = shape.tiles[i][j];

            }
        }
    }
}

Board::Board()
{
    // fill with black
    for (int i = 0; i < 20; i++) {
        for (int j = 0; j < 12; j++) {
            tiles[j][i] = sf::Color::Black;
        }
    }

    // boundary
    for (int i = 0; i < 12; i++) {
        tiles[i][19] = sf::Color::Red;
    }
    for (int i = 0; i < 19; i++) {
        tiles[0][i] = sf::Color::Red;
        tiles[11][i] = sf::Color::Red;
    }
```

```
}


int main()
{
    sf::RenderWindow window(sf::VideoMode(512, 512), "Tetris");
    sf::Font font; font.loadFromFile("joystix monospace.ttf");
    sf::Text score; score.setString("Score: ");
    score.setCharacterSize(30);
    score.setFont(font);
    score.setPosition(300, 200);
    sf::Clock clock;
    clock.restart();
    int score1 = 0;
    sf::Text score2;
    score2.setFont(font);
    score2.setCharacterSize(50);
    score2.setPosition(320, 250);

    Shape shape;

    Board board;

    // run the program as long as the window is open
    while (window.isOpen())
    {
        // check all the window's events that were triggered since the last iteration of the l
        sf::Event event;

        while (window.pollEvent(event))
        {
            // "close requested" event: we close the window
            if (event.type == sf::Event::Closed)
                window.close();


            if (event.type == sf::Event::KeyPressed) {
                if (event.key.code == sf::Keyboard::Left) {
                    shape.pos.x -= 1;
                    if (board.intersect(shape)) {
                        shape.pos.x += 1;
                        cout << "intersect left" << endl;
                    }
                }
                if (event.key.code == sf::Keyboard::Right) {
                    shape.pos.x += 1;
                    if (board.intersect(shape)) {
                        shape.pos.x -= 1;
                        cout << "intersect right" << endl;
                    }
                }
                if (event.key.code == sf::Keyboard::Down) {
                    shape.pos.y += 1;
                    if (board.intersect(shape)) {
                        shape.pos.y -= 1;
                        cout << "intersect down" << endl;
                    }
                }
                if (event.key.code == sf::Keyboard::Up) {
                    shape.rotateLeft();
                    if (board.intersect(shape)) {
                        shape.rotateRight();
                        cout << "intersect rotate" << endl;
                    }
                }
                if (event.key.code == sf::Keyboard::Space) {
                    while (!board.intersect(shape)) {
                        shape.pos.y += 1;
                        cout << "fast down" << endl;
                    }
                }
            }
        }

        float dt = clock.restart().asSeconds();

        shape.update(dt);

        score2.setString(to_string(score1));


        if (board.intersect(shape)) {
            shape.pos.y -= 1;
            board.add(shape);
            board.reduce(score1, board);
            shape.init();
            if (board.intersect(shape)) {
                cout << "GAME OVER" << endl;
            }
        }
        window.clear(sf::Color::Black);

        board.draw(window);
        shape.draw(window);
        window.draw(score);
        window.draw(score2);
        window.display();
    }

    return 0;
```

```
}
```

Task 1 .txt file:

```
3
0 0 1 0 0 0
0 0 1 0 0 0
0 0 1 0 0 0
0 0 1 0 0 0
0 0 1 0 0 0
0 0 1 0 0 0

0 0 0 0 0 0
0 0 1 0 0 0
0 0 1 1 0 0
0 0 1 0 0 0
0 0 1 1 0 0
0 0 0 0 0 0

0 0 0 0 0 0
0 0 1 0 0 0
0 0 1 1 1 0
0 0 1 1 0 0
0 0 0 0 0 0
0 0 0 0 0 0
```

Task 3 code:

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <cstdlib>
#include <random>

const int N = 9;

bool solve(int grid[N][N]);
bool clear(int grid[N][N], int &row, int &col);
bool usedRow(int grid[N][N], int row, int num);
bool usedCol(int grid[N][N], int col, int num);
bool usedBlock(int grid[N][N], int blockRow, int blcokCol, int num);
bool isSafe(int grid[N][N], int row, int col, int num);
void printGrid(int grid[N][N]);

int main() {

    std::random_device rand;
    std::default_random_engine generator(rand());
    std::uniform_int_distribution<> dist(0,3);
    int x;
    int grid[N][N] =
    { // An already solved grid, which will have a certain number
// of spots set to 0 by the following function.
// This is the only valid solution, shown in the grid below.
        {2, 9, 5,   7, 4, 3,    8, 6, 1},
        {4, 3, 1,   8, 6, 5,    9, 2, 7},
        {8, 7, 6,   1, 9, 2,    5, 4, 3},

        {3, 8, 7,   4, 5, 9,    2, 1, 6},
        {6, 1, 2,   3, 8, 7,    4, 9, 5},
        {5, 4, 9,   2, 1, 6,    7, 3, 8},

        {7, 6, 3,   5, 2, 4,    1, 8, 9},
        {9, 2, 8,   6, 7, 1,    3, 5, 4},
        {1, 5, 4,   9, 3, 8,    6, 7, 2}
    };
// Loops over the grid and sets a spot to 0.
        for (int r = 0; r < N; r++) {
            for (int c = 0; c < N; c++) {
                x = dist(generator);
                if (x==0) { grid[r][c] = 0; }
            } // 1/3 of the spots, each in a random
        }    // position, will be set to 0.

    printGrid(grid); std::cout << "\n";

    if(solve(grid)) { printGrid(grid); }
    return 0;
}

bool solve(int grid[N][N]) {
    int r, c;
    if (!clear(grid, r, c)) { return true; } // spot cleared?
    for (int i = 1; i <= 9; i++) {
        if (isSafe(grid, r, c, i)) { grid[r][c] = i;
            if (solve(grid)) { return true; } // grid solved? return.
            grid[r][c] = 0; // else, return false.
        }
    } return false;
}

bool clear(int grid[N][N], int &r, int &c) {
    for (r = 0; r < N; r++) { // Looping over the grid.
        for (c = 0; c < N; c++) {
            if (grid[r][c] == 0) { return true; }
        } // Returns true if the spot is open and clear.
    } return false;
}
```

```
bool usedRow(int grid[N][N], int r, int n) {
    for (int c = 0; c < N; c++) { // Looping over the columns in 1 row.
        if (grid[r][c] == n) { return true; } // Is number in he row?
    } return false;
}

bool usedCol(int grid[N][N], int c, int n) {
    for (int r = 0; r < N; r++) { // Looping over the rows in 1 column.
        if (grid[r][c] == n) { return true; } // Is number in the col?
    } return false;
}

bool usedBlock(int grid[N][N], int blockRow, int blockCol, int n) {
    for (int r = 0; r < 3; r++) { // Looping over the grid.
        for (int c = 0; c < 3; c++) {
            if (grid[r + blockRow][c + blockCol] == n) { return true; }
        } // Returns true if specified block is safe /
// the number is not already placed there.
    } return false;
}

bool isSafe(int grid[N][N], int r, int c, int n) {
// Returns true if the spot is not already used the row, column
// or specified block.
    return !usedRow(grid, r, n) && !usedCol(grid, c, n) &&
    !usedBlock(grid, r - r % 3 , c - c % 3, n);
}

void printGrid(int grid[N][N]) { // Loops over the grid and prints
    for (int r = 0; r < N; r++) { // the numbers inhabitating each spot.
        for (int c = 0; c < N; c++) {
            std::cout << grid[r][c] << " ";
        } std::cout << "\n";
    }
}
```