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 header file called "polyminos.h" and included it in the main program. This file will be available for users to create their own polyminos. The current setup is harcoded to be limited to 6 different polyminos. The main program calls a function with three parameters from the header file. We included some instructions in the very top of the header file on how to customize the shapes.

Here's an example of one of the shapes in the .h file. Explanation of the parameters is in the comments.

```
int polyminos(int x, int y, int z) {

switch (x) { //x = shape number

case 0:

switch (y) { //y = tile number, totalling at 6
case 0:
if (z == 1) //if z = 1 means its the x coordinate of the tile you want
return 2; //in this case the tile (2, 0) will be coloured
return 0;
case 1:
if (z == 1) //In this case tile (2,1) will be coloured
return 2;
return 1;
case 2:
if (z == 1)
return 2;
return 2;
case 3:
if (z == 1)
return 2;
return 3;
case 4:
if (z == 1)
return 2;
return 4;
case 5:
if (z == 1)
return 2;
return 5;
} //All this sums up to a shape that is 1x6 in size, which is a straigth line
```

Instead of moving the entire class to the .h file, which would also be a possible approach to the problem, we wanted to make a solution that was easier to grasp, and remove the users ability to screw up the functions in the class etc.

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.

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 3. The function loops over every row and coloumn, and if the random number equals 0 or 1, then the position is set to 0.
This function may result in an un-solveable game, but this very rarely happens.
The if statement can be changed to create easier or harder puzzles, depending on how many numbers are to be removed from board.

```
std::uniform_int_distribution<> dist(0,4);
x = dist(generator);
if (x==0 || x==1) { grid[r][c] = 0; }
```

4. Code Appendix:

Task 1 main program:

```
#include <SFML/Graphics.hpp>
#include <cmath>
#include <cstdlib>
#include <iostream>
#include <string>
#include "polyminos.h"

<<<<<<< HEAD

//Here you can make your Polyminos of choice.
//You can draw your polyminos like this and follow the explanation in the comments below
// 0  1  2  3 4  5
//0
//1    x
//2       x  x  x
//3    x        x
//4
//5

int polyminos(int x, int y, int z) {

switch (x) { //x = shape number

case 0:

switch (y) { //y = tile number, totalling at 6
case 0:
if (z == 1) //if z = 1 means its the x coordinate of the tile you want
return 2; //in this case the tile (2, 0) will be coloured
return 0;
case 1:
if (z == 1) //In this case tile (2,1) will be coloured
return 2;
return 1;
case 2:
if (z == 1)
return 2;
return 2;
case 3:
if (z == 1)
return 2;
return 3;
case 4:
if (z == 1)
return 2;
```

```
return 4;
case 5:
if (z == 1)
return 2;
return 5;
} //All this sums up to a shape that is 1x6 in size, which is a straigth line
/*


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][3-i];
        }
    }
    for(int i = 0; i < 6; i++) {
        for(int j = 0; j < 6; j++) {
            tiles[i][j]=tmp[i][j];
        }
    }
}
=======
using namespace std;
>>>>>>> b968024aa7b5f27a501459ec7471071ef25368bd

//
// Tetris
//
```

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

// times since last downward movement
float time;

<<<<<<< HEAD
#include <SFML/Graphics.hpp>
#include <cmath>
#include <cstdlib>
#include <iostream>
#include <sstream>

#ifndef _MYHEADER_H_INCLUDED
#define _MYHEADER_H_INCLUDED
#endif
#include "Shape.h"

using namespace std;

int score;  // Defining counter for tracking score.


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();

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

void Board::reduce() {
    for(int i = 3; i < 19; i++) {
        int count = 0;
        int tilecount = 0;
```

```
        for(int j = 1; j < 11; j++) {
            if (tiles[j][i] != sf::Color::Black) {
                count++;
                if(count == 10) {
                    score = score+100;
                    tilecount = i;
                    for(int k = tilecount; k >= 3; k--) {
                        for(int j = 1; j < 11; j++) {
                            tiles[j][k] = tiles[j][k-1];
                        }
                    }
                    break;
                }
            }
        }
    }
}

bool Board::intersect(Shape& shape) {
    bool intersect = false;
    for(int i = 0; i < 4; i++) {
        for(int j = 0; j < 4; 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 < 4; i++) {
        for(int j = 0; j < 4; 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;
        }
    }
    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::Clock clock;
    clock.restart();
    Shape shape;
    Board board;

    sf::Text hud;
    sf::Font font;
    font.loadFromFile("game_over.ttf");
    hud.setFont(font);
    hud.setCharacterSize(65);
    hud.setFillColor(sf::Color::White);
    hud.setPosition(350,200);

    while (window.isOpen())
    {
        // check all the window's events that were triggered since the last iteration of the lo
        sf::Event event;

        while (window.pollEvent(event))
        {
            // "close requested" event: we close the window
            if (event.type == sf::Event::Closed || sf::Keyboard::isKeyPressed(sf::Keyboard::Q))
                { 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) {
            shape.pos.y += 2;
            if(board.intersect(shape)) {
                shape.pos.y -= 1;
                cout << "intersect down" << endl;
            }
        }
    }
    std::stringstream ss;
    ss << "Score: " << score;
    hud.setString(ss.str());
}

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

shape.update(dt);

if(board.intersect(shape)) {
    shape.pos.y -= 1;
    board.add(shape);
    board.reduce();
    shape.init();
    if(board.intersect(shape)) {
        cout << "GAME OVER" << endl;
    }
```

```
        }
        window.clear(sf::Color::Black);
        board.draw(window);
        window.draw(hud);
        shape.draw(window);
        window.display();
    }
    return 0;
}


=======
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;
}
}

int shapenr = 0; // rand() % 7;

for (int i = shapenr; i < 6; i++) {
tiles[polyminos(shapenr, i, 1)][polyminos(shapenr, i, 2)] = sf::Color::White;
}
}

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 + 100));
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();

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

void Board::reduce()
{

}

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 loop
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);

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;
}

}
score2.setString(to_string(score1));


if (board.intersect(shape)) {
shape.pos.y -= 1;
board.add(shape);
board.reduce();
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 .h file:

```
//Here you can make your Polyminos of choice.
//You can draw your polyminos like this and follow the explanation in the comments below
// 0  1  2  3 4  5
//0
//1    x
//2       x  x  x
//3    x        x
//4
//5

int polyminos(int x, int y, int z) {

switch (x) { //x = shape number

case 0:

switch (y) { //y = tile number, totalling at 6
case 0:
if (z == 1) //if z = 1 means its the x coordinate of the tile you want
return 2; //in this case the tile (2, 0) will be coloured
return 0;
case 1:
if (z == 1) //In this case tile (2,1) will be coloured
return 2;
return 1;
case 2:
if (z == 1)
return 2;
return 2;
case 3:
if (z == 1)
return 2;
return 3;
case 4:
if (z == 1)
return 2;
return 4;
case 5:
```

```
if (z == 1)
return 2;
return 5;

//copy and paste the above and make up to 5 more cases/shapes.
} //All this sums up to a shape that is 1x6 in size, which is a straigth line

//
```