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.

2. Task 2:

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?

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	

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. Code Appendix:

```
<<<<<<< HEAD
<<<<<<< HEAD
#include <SFML/Graphics.hpp>
```

```
#include <cmath>
#include <cstdlib>
#include <iostream>
#include <string>

using namespace std;

//
// Tetris
//
class Shape
{
public:
sf::Color tiles[4][4];
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[4][4];
for (int i = 0; i < 4; i++) {
for (int j = 0; j < 4; j++) {
tmp[i][j] = tiles[j][3 - i];
}
}
for (int i = 0; i < 4; i++) {
for (int j = 0; j < 4; 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 < 4; i++) {
for (int j = 0; j < 4; j++) {
tiles[i][j] = sf::Color::Black;
}
}


// two hardcoded shapes - this should be changed by you
switch (rand() % 7) {

case 0:
tiles[2][0] = sf::Color::White;
tiles[2][1] = sf::Color::White;
tiles[2][2] = sf::Color::White;
tiles[2][3] = sf::Color::White;
break;

case 1:
tiles[0][2] = sf::Color::Blue;
tiles[1][2] = sf::Color::Blue;
tiles[1][1] = sf::Color::Blue;
tiles[2][1] = sf::Color::Blue;
break;

case 2:
tiles[0][2] = sf::Color::Cyan;
tiles[1][2] = sf::Color::Cyan;
tiles[2][2] = sf::Color::Cyan;
tiles[1][1] = sf::Color::Cyan;
break;
```

```
case 3:
tiles[1][1] = sf::Color::Yellow;
tiles[1][2] = sf::Color::Yellow;
tiles[2][1] = sf::Color::Yellow;
tiles[2][2] = sf::Color::Yellow;
break;

case 4:
tiles[1][0] = sf::Color::Green;
tiles[1][1] = sf::Color::Green;
tiles[1][2] = sf::Color::Green;
tiles[2][2] = sf::Color::Green;
break;

case 5:
tiles[1][0] = sf::Color::Magenta;
tiles[1][1] = sf::Color::Magenta;
tiles[1][2] = sf::Color::Magenta;
tiles[2][0] = sf::Color::Magenta;
break;

case 6:
tiles[0][1] = sf::Color(229, 204, 255);
tiles[1][2] = sf::Color(229, 204, 255);
tiles[1][1] = sf::Color(229, 204, 255);
tiles[2][2] = sf::Color(229, 204, 255);
}

}

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 < 4; i++) {
for (int j = 0; j < 4; 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 < 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;
}
}

// 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;
}



=======
>>>>>>> f3a5158dadd55e343db3809c24d12fab18055acf
=======

shape.h
>>>>>>> 3cf73b260c573a7f25233eb267f114cca532638e

class Shape {
public:
    sf::Color tiles[4][4];
    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[4][4];
    for(int i = 0; i < 4; i++) {
        for(int j = 0; j < 4; j++) {
            tmp[i][j]=tiles[j][3-i];
        }
    }
    for(int i = 0; i < 4; i++) {
        for(int j = 0; j < 4; j++) {
            tiles[i][j]=tmp[i][j];
        }
    }
}

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

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

void Shape::draw(sf::RenderWindow& w) {
    sf::CircleShape s;
    s.setRadius(8);
    s.setOrigin(8,8);
    for(int i = 0; i < 4; i++) {
        for(int j = 0; j < 4; 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);
            }
        }
    }
}

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

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

    for(int i = 0; i < 4; i++) {
        for(int j = 0; j < 4; j++) {
            tiles[i][j] = sf::Color::Black;
        }
    }
    switch(rand() % 8) {
        case 0:
            tiles[1][1] = sf::Color::Yellow;
            tiles[1][2] = sf::Color::Yellow;
            tiles[2][1] = sf::Color::Yellow;
            tiles[2][2] = sf::Color::Yellow;
            break;
        case 1:
            tiles[2][0] = sf::Color(255,160,0);
            tiles[2][1] = sf::Color(255,160,0);
            tiles[2][2] = sf::Color(255,160,0);
            tiles[2][3] = sf::Color(255,160,0);
            break;
        case 2:
            tiles[0][2] = sf::Color::Blue;
            tiles[1][2] = sf::Color::Blue;
            tiles[1][1] = sf::Color::Blue;
            tiles[2][1] = sf::Color::Blue;
            break;
        case 3:
            tiles[0][2] = sf::Color::Green;
            tiles[1][2] = sf::Color::Green;
            tiles[2][2] = sf::Color::Green;
            tiles[1][1] = sf::Color::Green;
            break;
        case 4:
            tiles[2][3] = sf::Color::White;
            tiles[2][2] = sf::Color::White;
            tiles[1][2] = sf::Color::White;
            tiles[0][2] = sf::Color::White;
            break;
        case 5:
            tiles[2][3] = sf::Color(4,235,250);
            tiles[2][2] = sf::Color(4,235,250);
            tiles[1][2] = sf::Color(4,235,250);
            tiles[0][2] = sf::Color(4,235,250);
            break;
        case 6:
            tiles[2][1] = sf::Color(255,0,255);
            tiles[2][2] = sf::Color(255,0,255);
            tiles[1][2] = sf::Color(255,0,255);
            tiles[0][2] = sf::Color(255,0,255);
            break;
        case 7: {
            int x=rand() % 4; int y = rand() % 4;
            tiles[x][y] = sf::Color(75,0,150);
            x=rand() % 4; y = rand() % 4;
            tiles[x][y] = sf::Color(75,0,150);
            x=rand() % 4; y = rand() % 4;
            tiles[x][y] = sf::Color(75,0,150);
            x=rand() % 4; y = rand() % 4;
            tiles[x][y] = sf::Color(75,0,150);
            break;
        }
    }
}
```