Praktikumstermin Nr. 11, INF: Operator-Methode, Vererbung, Pong

Abgabe im GIP-INF Praktikum der Woche 3.1.-7.1.2022.

(Pflicht-) Aufgabe INF-11.01: Operator-Methode `operator*`

Kopieren Sie die Dateien `MyRectangle.h` und `MyRectangle.cpp` aus dem letzten Praktikum in ein neues Projekt.

Programmieren Sie eine Methode `draw_red()` durch Kopieren der Methode `draw()`, wobei Sie im Methodenrumpf dann das `blue` durch `red` als Farbe ersetzen.

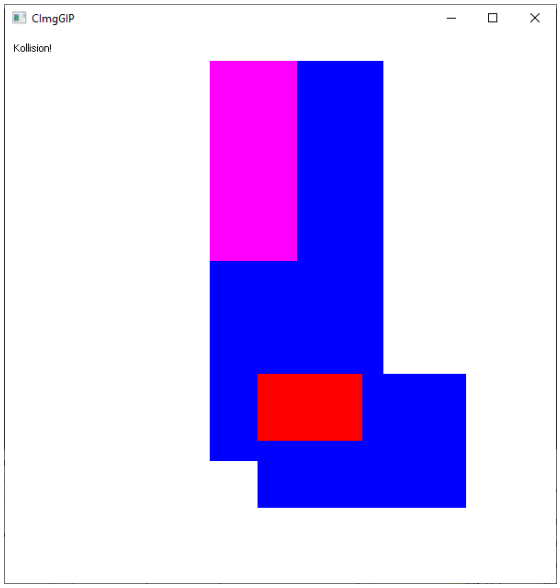
Programmieren Sie einen `operator * (double d)`, so dass man für ein `MyRectangle r1` einen Ausdruck `r1 * d` (z.B. `r1 * 0.5`) schreiben kann und dies bedeutet, dass ein neues Rechteck entsteht, welches die gleiche linke obere Ecke `(x1, y1)` hat wie das ursprüngliche Rechteck `r1`, aber um den Faktor `d` verlängerte oder verkürzte Kanten (z.B. im Fall `d = 0.5` nur halb so lange Kanten).

Ergänzen Sie das Hauptprogramm unterhalb des Kollisionstests (siehe auch Vorgabedatei in Ilias) um die Zeilen …

```
MyRectangle r3 = r1 * 0.5;
r3.draw_red();

(r2*0.5).draw_red();
```

Testen Sie Ihr Programm, indem Sie das Programm laufen lassen und die Ausgaben beobachten und visuell verifizieren.
Und wundern Sie sich nicht, wenn eines der roten Rechtecke ggfs. in rosa Farbe gezeichnet wird, das ist ein bekanntes Problem.



(Pflicht-) Aufgabe INF-11.02: Vererbung, Klasse `MyFilledRectangle`

Kopieren Sie die Dateien `MyRectangle.h` und `MyRectangle.cpp` in ein neues Projekt.

Legen Sie außerdem neue Dateien `MyFilledRectangle.h` und `MyFilledRectangle.cpp` an und programmieren Sie dort eine Klasse `MyFilledRectangle`, die von `MyRectangle` erbt.

Die Klasse `MyFilledRectangle` besitze keine eigenen Attribute, sondern nutze die Attribute der Basisklasse `MyRectangle`.
Wie muss der Zugriffsschutz der Basisklassen-Attribute sein (bzw. gegenüber dem vorigen Praktikum geändert werden), damit diese Attribute in der abgeleiteten Klasse mitgenutzt werden können und trotzdem nicht von außen zugreifbar sind?

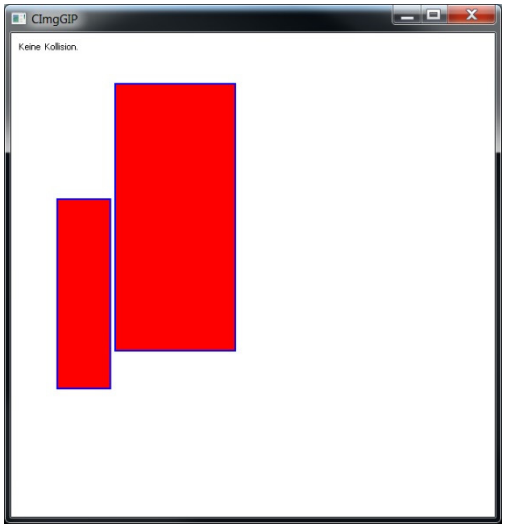
Der Konstruktor von `MyFilledRectangle` habe die gleichen Parameter wie der Konstruktor der Basisklasse `MyRectangle` und initialisiere die Attribute durch Aufruf des Basisklassen-Konstruktors mit den Parametern des eigenen Aufrufs.

Die `draw()` Methode von `MyFilledRectangle` soll erst einmal mit Hilfe der `draw()` Methode von `MyRectangle` ein blaues Rechteck zeichnen.
Dann sollen mit Hilfe der in `CImgGIP06.h` realisierten Funktion …

```
gip_draw_line(unsigned int x1, unsigned int y1,
              unsigned int x2, unsigned int y2,
              gip_color color);
```

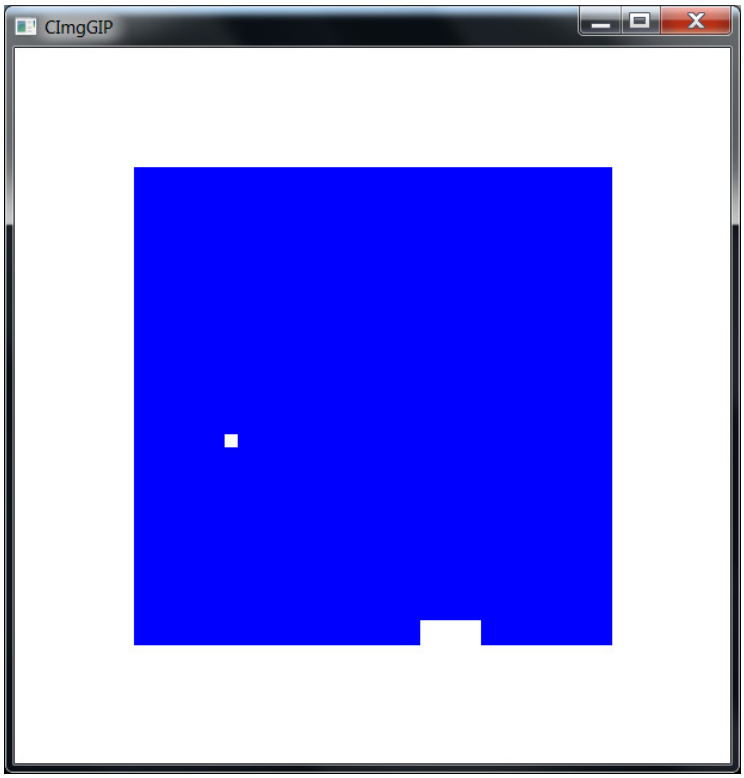
… lauter horizontale rote (`red`) Linien gezeichnet werden, von „ganz oben

im Kästchen" (linker bis rechter Rand) bis „ganz unten im Kästchen" (linker bis rechter Rand), allerdings mit jeweils einem Abstand von 2 Pixeln zum Rand (an allen vier Rändern). D.h. das Rechteck wird durch lauter horizontale rote Linien ausgefüllt und außen befindet sich ein blauer Rand mit einer Breite von 2 Pixeln.
Ihr Code soll prüfen, ob das `MyFilledRectangle` auch wirklich hoch und breit genug ist, um einen Abstand von 2 Pixeln zum Rand zu erlauben. Falls nicht, so soll das „Füllen mit Linien" nicht gemacht werden (und nur das `MyRectangle` gezeichnet werden).



Schreiben Sie ein Hauptprogramm analog zum Hauptprogramm des vorherigen Praktikums, welches alle 4 Sekunden zwei neue `MyFilledRectangle` an zufälligen Positionen zeichnet und die Kollisionserkennung durchführt.

(Pflicht-) Aufgabe INF-11.03: Single-Player Pong



Programmieren Sie ein Spiel, bei dem ein kleiner weißer „Ball" (als Kästchen 10x10 Pixel) von einem Schläger und den Wänden links, rechts und oben abprallt. Wenn der Ball den Schläger verfehlt und das Spielfeld nach unten verlässt, soll das Spiel enden (Fenster schließt sich).

Der Schläger bewege sich in x-Richtung in Richtung der aktuellen Mausposition. Die x-Koordinate der aktuellen Mausposition erhalten Sie als Rückgabewert der parameterlosen Funktion …
`unsigned int gip_mouse_x()`.

Spielfeld:
linke obere Ecke an Position (100,100),
rechte untere Ecke an Position (500,500).
Schläger: 50 Pixel breit, 20 Pixel hoch.
Ballgeschwindigkeit: 3 Pixel in x- und y-Richtung je Schleifendurchlauf der „alles neu zeichnen" Schleife.
Der Ball starte etwas rechts der Mitte des Spielfelds, mit diagonaler Bewegung in Richtung links oben, so dass er erst an der oberen Wand

abpralle und kurz danach an der linken Wand. Abprallen links und rechts kehren die x-Bewegungsgeschwindigkeit und –richtung um, abprallen oben und unten (am Schläger) kehren die y-Bewegungsgeschwindigkeit und –richtung um.

Benutzen Sie die entsprechenden Anteile aus vorherigen Lösungen: Abprallen am Rand und die Kollisionserkennung mit Rand (oder Schläger) ähnlich dem Abprallen des Kästchens in der ersten Aufgabe mit der CImg Bibliothek.

Mögliche schrittweise Entwicklung der Lösung:

1. Spielfeld zeichnen. Ball an Startposition zeichnen. Ball bewegt sich nach links oben.

2. Kollisionserkennung mit der oberen Wand hinzufügen. Dann y-Bewegungsrichtung des Balls umkehren, so dass er in y-Richtung abprallt.

3. Kollisionserkennung mit linker Wand hinzufügen. Dann x-Bewegungsrichtung des Balls umkehren, so dass er in x-Richtung abprallt.

4. Kollisionserkennung mit rechter Wand hinzufügen. Dann x-Bewegungsrichtung des Balls umkehren, so dass er in x-Richtung abprallt. Dabei ggfs. initiale Startrichtung des Balls in „nach rechts oben" ändern, so dass Abprallen an der rechten Wand vorkommt.

5. Schläger zeichnen.

6. Bewegung des Schlägers in x-Richtung hin zur Mausposition realisieren.

7. Kollisionserkennung des Balls mit dem Schläger hinzufügen und entsprechendes Abprallen nach oben.

Viele dieser Aspekte lassen sich jeweils mit einer oder wenigen Programmzeilen Code realisieren, so dass das resultierende Programm gar nicht so umfangreich ist …

Programmgerüst (siehe auch Datei in Ilias):

```
// Datei: pong.cpp

#define CIMGGIP_MAIN
#include "CImgGIP06.h"
using namespace std;

int main()
{
    // Für das "blaue Spielfeld" ...
    const unsigned int x0 = 100, y0 = 100;
    const unsigned int x1 = 500, y1 = 500;
    // Für Position und Ausdehnung des weißen Balls ...
    int xb = 200, yb = 300;
    const int ball_size = 10;
    // Geschwindigkeit des Balls ...
    int delta_x = -3, delta_y = -3;
    // Ausdehnung und Position des Schlaegers ...
    const int schlaeger_size_x = 50, schlaeger_size_y = 20;
    int xs = 300, ys = y1 - schlaeger_size_y;

    gip_white_background();
    while (gip_window_not_closed())
    {
        // Später nötig, damit die Graphik "schneller" wird …
        // gip_stop_updates();

        // Blaues "Spielfeld" neu zeichnen ...

        // Ball zeichnen ...

        // Schlaeger zeichnen ...

        // Schlaeger verschieben ...

        // Falls der Schläger außerhalb des Spielfelds => zurücksetzen ...

        // Ball-Kollisionen mit dem Rand ...

        // Kollision mit Schlaeger ...

        // Unterer Rand erreicht => Abbruch

        // Bewege Ball ...

        // Später nötig, damit die Graphik "schneller" wird …
        // gip_start_updates();

        // Später nötig, wenn die Graphik "schneller gestellt" ist …
        // Etwas Pause, damit das Spiel nicht zu schnell läuft ...
        // gip_wait(50);
    }
    return 0;
}
```

Für das Aufgaben-Tutorium am Freitag 17.12.2021, als freiwillige Aufgabe:

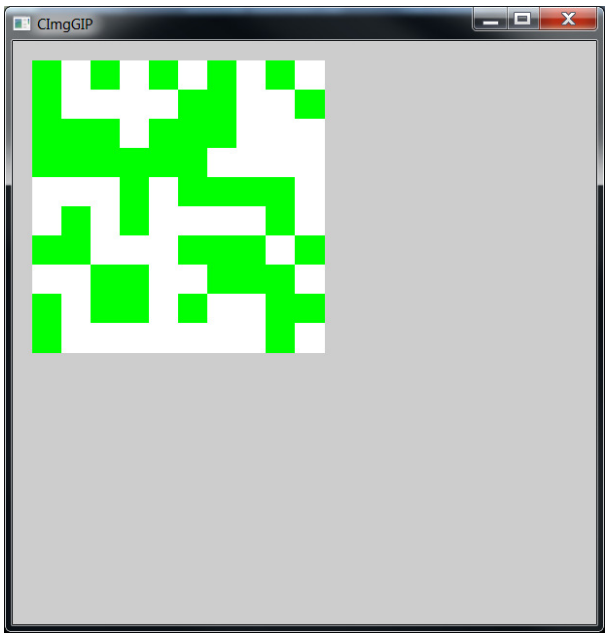
(Freiwillige) Aufgabe INF-11.04: Conway's *Game of Life*

Schreiben Sie unter Benutzung der `CImg` Library (mittels der Headerdatei `CImgGIP06.h`, die Sie schon aus einem vorherigen Praktikumsversuch kennen) ein C++ Programm, welches das Spiel *Game of Life* realisiert. Dieses stellt die zeitliche Entwicklung einer „Zellkolonie" dar.

https://de.wikipedia.org/wiki/Conways_Spiel_des_Lebens

Die Größe des Feldes sei als Konstante `grid_size` vorgegeben. Benutzen Sie ein zweidimensionales statisches Array zur Realisierung des Spielfeldes.

Das Spiel startet je nach Benutzereingabe entweder mit einer zufälligen Belegung des Feldes oder mit einer vorgegebenen Belegung. Benutzen Sie für die zufällige Belegung den Funktionsaufruf `gip_random(0,1);` um jeweils eine Zufallszahl „entweder 0 oder 1" zu erzielen.



Orientieren Sie sich an folgendem Programmgerüst (welches auch als Datei in Ilias vorliegt) für Ihre Lösung:

```
#include <iostream>
#define CIMGGIP_MAIN
#include "CImgGIP06.h"
using namespace std;
```

```
using namespace cimg_library;

const int grid_size = 10; // Anzahl an Kaestchen in x- und y-Richtung
const int box_size = 30;  // size der einzelnen Kaestchen (in Pixel)
const int border = 20;   // Rand links und oben bis zu den ersten Kaestchen (in Pixel)

// Prototyp der Funktionen zum Vorbelegen des Grids ...
void grid_init(bool grid[][grid_size]);

int main()
{
    bool grid[grid_size][grid_size] = { 0 };
    bool next_grid[grid_size][grid_size] = { 0 };

    // Erstes Grid vorbelegen ...
    grid_init(grid);

    while (gip_window_not_closed())
    {
        // Spielfeld anzeigen ...
        // gip_stop_updates(); // ... schaltet das Neuzeichnen nach
                               //     jeder Bildschirmänderung aus

        // TO DO

        // gip_start_updates(); // ... alle Bildschirmänderungen (auch die
                                //     nach dem gip_stop_updates() ) wieder anzeigen
        gip_sleep(3);

        // Berechne das naechste Spielfeld ...
        // Achtung; Für die Zelle (x,y) darf die Position (x,y) selbst *nicht*
        // mit in die Betrachtungen einbezogen werden.
        // Ausserdem darf bei zellen am rand nicht über den Rand hinausgegriffen
        // werden (diese Zellen haben entsprechend weniger Nachbarn) ...

        // TO DO


        // Kopiere das naechste Spielfeld in das aktuelle Spielfeld ...

        // TO DO

    }
    return 0;
}

void grid_init(bool grid[][grid_size])
{
    int eingabe = -1;
    do {
        cout << "Bitte waehlen Sie die Vorbelegung des Grids aus:" << endl
            << "0 - Zufall" << endl
            << "1 - Statisch" << endl
            << "2 - Blinker" << endl
            << "3 - Oktagon" << endl
            << "4 - Gleiter" << endl
            << "5 - Segler 1 (Light-Weight Spaceship)" << endl
            << "6 - Segler 2 (Middle-Weight Spaceship)" << endl
```

```
            << "? ";
        cin >> eingabe;
        cin.clear();
        cin.ignore(1000, '\n');
    } while (eingabe < 0 || eingabe > 6);

    if (eingabe == 0)
    {
        // Erstes Grid vorbelegen (per Zufallszahlen) ...

        // TO DO

    }
    else if (eingabe == 1)
    {
        const int pattern_size = 3;
        char pattern[pattern_size][pattern_size] =
        {
            { '.', '*', '.' },
            { '*', '.', '*' },
            { '.', '*', '.' },
        };
        for (int y = 0; y < pattern_size; y++)
            for (int x = 0; x < pattern_size; x++)
                if (pattern[y][x] == '*')
                    grid[x][y+3] = true;
    }
    else if (eingabe == 2)
    {
        const int pattern_size = 3;
        char pattern[pattern_size][pattern_size] =
        {
            { '.', '*', '.' },
            { '.', '*', '.' },
            { '.', '*', '.' },
        };
        for (int y = 0; y < pattern_size; y++)
            for (int x = 0; x < pattern_size; x++)
                if (pattern[y][x] == '*')
                    grid[x][y+3] = true;
    }
    else if (eingabe == 3)
    {
        const int pattern_size = 8;
        char pattern[pattern_size][pattern_size] =
        {
            { '.', '.', '.', '*', '*', '.', '.', '.' },
            { '.', '.', '*', '.', '.', '*', '.', '.' },
            { '.', '*', '.', '.', '.', '.', '*', '.' },
            { '*', '.', '.', '.', '.', '.', '.', '*' },
            { '*', '.', '.', '.', '.', '.', '.', '*' },
            { '.', '*', '.', '.', '.', '.', '*', '.' },
            { '.', '.', '*', '.', '.', '*', '.', '.' },
            { '.', '.', '.', '*', '*', '.', '.', '.' },
        };
        for (int y = 0; y < pattern_size; y++)
            for (int x = 0; x < pattern_size; x++)
                if (pattern[y][x] == '*')
```

```
                grid[x][y+1] = true;
    }
    else if (eingabe == 4)
    {
        const int pattern_size = 3;
        char pattern[pattern_size][pattern_size] =
        {
            { '.', '*', '.' },
            { '.', '.', '*' },
            { '*', '*', '*' },
        };
        for (int y = 0; y < pattern_size; y++)
            for (int x = 0; x < pattern_size; x++)
                if (pattern[y][x] == '*')
                    grid[x][y+3] = true;
    }
    else if (eingabe == 5)
    {
        const int pattern_size = 5;
        char pattern[pattern_size][pattern_size] =
        {
            { '*', '.', '.', '*', '.' },
            { '.', '.', '.', '.', '*' },
            { '*', '.', '.', '.', '*' },
            { '.', '*', '*', '*', '*' },
            { '.', '.', '.', '.', '.' },
        };
        for (int y = 0; y < pattern_size; y++)
            for (int x = 0; x < pattern_size; x++)
                if (pattern[y][x] == '*')
                    grid[x][y+3] = true;
    }
    else if (eingabe == 6)
    {
        const int pattern_size = 6;
        char pattern[pattern_size][pattern_size] =
        {
            { '.', '*', '*', '*', '*', '*' },
            { '*', '.', '.', '.', '.', '*' },
            { '.', '.', '.', '.', '.', '*' },
            { '*', '.', '.', '.', '*', '.' },
            { '.', '.', '*', '.', '.', '.' },
            { '.', '.', '.', '.', '.', '.' },
        };
        for (int y = 0; y < pattern_size; y++)
            for (int x = 0; x < pattern_size; x++)
                if (pattern[y][x] == '*')
                    grid[x][y+3] = true;
    }
}
```

Eine Zelle wird in einem leeren Feld neu geboren, wenn im vorigen Grid drei der umgebenden Nachbarzellen belebt waren.

Nachbarzellen: Eine Zelle, die nicht am Rand des Spielfelds liegt, hat 8 Nachbarn: oben, unten, links, rechts und 4x diagonal. Die Zelle selbst zählt nicht zu ihren eigenen Nachbarn. Zellen am Rand des Spielfelds haben weniger Nachbarn.

Eine Zelle bleibt in einem Feld am Leben, wenn im vorigen Grid zwei oder drei der umgebenden Nachbarzellen belebt waren.

In allen anderen Fällen wird keine Zelle geboren bzw. eine dort lebende Zelle stirbt wegen zu vieler oder zu weniger Nachbarn, d.h. das Feld wird im nächsten Spielfeld unbewohnt sein.

Stellen Sie das Spielfeld über die Zellgenerationen hinweg graphisch dar.

Hinweis: Sie werden sehen, dass die Aktualisierungen des "Spielfelds" recht langsam "Kästchen für Kästchen" passieren … Das ist eine Konsequenz der GIP-spezifischen Änderungen der CImg Library (http://cimg.eu/), die ich vorgenommen habe: Ich "zwinge die Library", jede Graphik-Operation auch sofort auf dem Bildschirm sichtbar zu machen. Das bremst die Library sehr stark, hat aber den Vorteil, dass Sie im Debugger jeden Schritt des Programms einzeln nachverfolgen können und wirklich auf dem Bildschirm sehen, was ihre einzelnen Graphik-Operationen wie `gip_draw_rectangle()` *tun. Normalerweise wäre das nicht der Fall …*
Wenn Sie dann sicher sind, dass ihr Code wohl funktioniert, dann fügen Sie die im Programmrahmen noch auskommentierten Funktionsaufrufe `gip_stop_updates()` *und* `gip_start_updates()` *hinzu.*
`gip_stop_updates()` *"schaltet CImg in seinen üblichen Modus um": Graphik-Änderungen ("Updates") werden* ***nicht*** *mehr direkt angezeigt, sondern nur noch intern durchgeführt und gespeichert.*
Erst mit dem Aufruf von `gip_start_updates()` *wird die Library wieder in den "GIP Modus geschaltet": Alle "in der Zwischenzeit" intern gespeicherten Änderungen werden dann mit einem Schlag sichtbar gemacht und alle danach stattfindenden Änderungen werden dann auch wieder sofort sichtbar gemacht …*

Testläufe: *(Animation der vorgegebenen Muster siehe Wikipedia Seite)*

```
Bitte waehlen Sie die Vorbelegung des Grids aus:
0 - Zufall
1 - Statisch
2 - Blinker
3 - Oktagon
4 - Gleiter
5 - Segler 1 (Light-Weight Spaceship)
6 - Segler 2 (Middle-Weight Spaceship)
?
```