BABEŞ–BOLYAI TUDOMÁNYEGYETEM KOLOZSVÁR
MATEMATIKA ÉS INFORMATIKA KAR
DIDAKTIKAI MESTERI – INFORMATIKA SZAK

Magiszteri dolgozat

Számítógépes grafika keretrendszer középiskolásoknak

TÉMAVEZETŐ:

DR. MEZEI ILDIKÓ-ILONA,
EGYETEMI ADJUNKTUS

SZERZŐ:

BEILAND ARNOLD

2023

BABEŞ–BOLYAI UNIVERSITY OF CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND INFORMATICS
SPECIALIZATION: MASTER IN DIDACTICS – COMPUTER SCIENCE

Master's Thesis

A computer graphics framework for high school students

ADVISOR:

ILDIKÓ-ILONA MEZEI, PHD.
UNIVERSITY LECTURER

AUTHOR:

ARNOLD BEILAND

2023

UNIVERSITATEA BABEŞ–BOLYAI, CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA MASTERAT DIDACTIC – INFORMATICĂ

Lucrare de disertație

Framework de grafică computațională pentru elevi de liceu

CONDUCĂTOR ŞTIINŢIFIC:
LECTOR DR. ILDIKÓ-ILONA MEZEI

ABSOLVENT:
ARNOLD BEILAND

2023

BABEŞ–BOLYAI UNIVERSITY OF CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND INFORMATICS
SPECIALIZATION: MASTER IN DIDACTICS – COMPUTER SCIENCE

Master's Thesis

A computer graphics framework for high school students

Abstract

We propose a low-level C++ computer graphics framework suitable for facilitating the teaching of some aspects of computer programming in a high school environment.

Students can color the pixels of a UI window and render text using simple function calls. No prerequisite knowledge of object-oriented programming or computer graphics is necessary, the exercises and problems proposed in this text can be solved using the basic features of C++ that are covered by the high school curriculum in Romania for students enrolled in the mathematics and computer science program, grades 9–11.

Starting with simple example code we provide a set of problems with gradually increasing difficulty that are solvable using the two functions exposed by the framework in the hope of sparking interest for computer graphics and programming in general and motivating students to deepen their knowledge. Here they are provided with a way to generate graphical output, which complements the text based I/O in programs that they are usually required to write (either using the standard I/O streams or input and output files).

The implementation of this framework is an adapter layer over FLTK, a cross-platform GUI toolkit. Relying on it made it easier for us to provide a platform-independent implementation that can be used on Microsoft Windows operating systems and GNU/Linux distributions as well.

I would like to thank my advisor, Ildikó-Ilona Mezei, who helped me shape these ideas into a form suitable for an educational setting and reviewed the text, providing feedback that is highly appreciated.

This work is the result of my own activity. I have neither given nor received unauthorized assistance on this work.

2023 ARNOLD BEILAND

ADVISOR:
ILDIKÓ-ILONA MEZEI, PHD.
UNIVERSITY LECTURER

Tartalomjegyzék

1. fejezet

Bevezető

Hazánkban a középiskolai programozásoktatás során a diákokat olyan (C++, C vagy Pascal) programok megírására tanítjuk, melyek szöveges bemenetből valamilyen algoritmussal szöveges kimenetet állítanak elő. Ez az elején csak a standard bemenet és kimenet felhasználását jelenti, később pedig beolvasást szöveges fájlokból és ezekbe való írást (illetve a standard ki/-bemenet és a fájlműveletek tetszőleges kombinációit). A hangsúly az információ feldolgozására, az alapvető algoritmusok megalkotására és megértésére helyeződik, az adatok megjelenési formája másodlagos.

Jelen dolgozatban szeretnénk egy könnyen használható C++ keretrendszert javasolni grafikus kimenet előállítására annak reményében, hogy a szöveges adatok használata mellé egy érdekes és látványos eszköz kerül a diákok kezébe, ösztönözve őket a kísérletezésre és gondolkodásra. Javasolunk néhány (kezdetben egyszerűbb, majd egyre bonyolultabb) feladatot is, melyek segítségével a tanárok kisebb-nagyobb kihívásokat állíthatnak diákjaik elé. A feladatok egy része interdiszciplináris jellegű, megoldásukhoz szükséges lesz felhasználni néhány alapismeretet az analitikus geometria területéről.

Bár számos nyílt forráskódú C++ grafikus keretrendszer és eszköztár áll a programozók rendelkezésére (Qt[1], GTK[2], WxWidgets[3], FLTK[4] stb.), ezek általában lényeges mértékben támaszkodnak az objektumorientált programozás elemeire (öröklés, polimorfizmus), illetve esetenként függvénymutatók és sablonok használatára. A középiskolai környezetben való alkalmazás viszont ebből a szempontból jelentős megszorításokkal jár, hiszen ezek a technikák már túlmutatnak a középiskolások nagy részének programozástudásán, megtanításukat a jelenleg érvényes tantervek nem írják elő. Ennek értelmében egy teljesen procedurális grafikus könyvtárra len-

[1] https://www.qt.io/
[2] https://gtk.org/
[3] https://www.wxwidgets.org/
[4] https://www.fltk.org/

ne szükségünk, amit a diákok a megszokott környezetből használhatnak (Code::Blocks IDE és GCC fordítócsomag, jellemzően Microsoft Windows operációs rendszeren vagy valamilyen GNU/Linux disztribúción).

Az alapötlet forrásai az Olive.c projekt [Kutepov] és a gfx nevű grafikus könyvtár [Thain]. Az előbbi bizonyos grafikus primitívek (pl. háromszög, ellipszis stb.) kirajzolásával foglalkozik, de csak memóriában állítja elő a kimenetet megjelenítés nélkül. Az utóbbi tud grafikus ablakokat kezelni, de csak X Window System[5] fölött működik. Azt is szeretnénk viszont elérni, hogy a grafikus primitívek (háromszögek, vonalak stb.) kirajzolása már a diákok feladata legyen, csak egy-egy pixel színének beállítására biztosítsunk nekik beépített függvényt.

Ennek érdekében egy FLTK-ra épülő procedurális keretrendszert állítottunk össze Simple Graphics Framework (SGF) néven, mely Microsoft Windows operációs rendszereken és GNU/-Linux disztribúciókon is könnyen használható néhány konfigurációs lépés elvégzése után.

A fejezet további részében ennek technikai részleteit mutatjuk be. A második fejezetben szeretnénk néhány fokozatosan nehezedő feladatot javasolni a keretrendszer használatára, a harmadik fejezetben pedig egy függvényábrázoló program elkészítésének lépeseit mutatjuk be, mely egy diákok által implementálható, a megszokott programjaiknál nagyobb kiterjedésű szoftverprojekt.

Köszönettel tartozom témavezetőmnek, Dr. Mezei Ildikó-Ilonának, akitől hasznos visszajelzéseket kaptam bemutatott ötletek, feladatok és a dolgozat összeállítására vonatkozóan.

1.1. A keretrendszer használata

A keretrendszerrel készíthető legegyszerűbb projekt három fájlból áll: a keretrendszer implementációja (*main.cpp* néven), egy forrásfájl, melyben a felhasználó kódja található (ennek neve tetszőleges) és egy header-állomány (*sgf.h*), melyben egyrészt a keretrendszer által biztosított, másrészt a felhasználó kódjától elvárt függvények deklarációja található néhány konstans kíséretében. Ezt a fejállományt mindkét forrásfájlba beillesztjük a megszokott `#include` direktíva segítségével.

A fordítás során a két forrásfájlt (akár külön modulként) tárgykóddá fordítjuk, majd egyetlen végrehajtható állományt készítünk belőle, megadva a szerkesztőnek az FLTK eszköztár bináris

[5] https://www.x.org/wiki/

fájljainak elérési útvonalát is. A diákoktól nem várjuk el, hogy a fordítás technikai részleteit ismerjék, ezekkel az előre elkészített Code::Blocks projekt build rendszere foglalkozik majd.

A közös header-állomány kódját a mellékletben megtalálható *src/sgf.h* fájl tartalmazza. Első része a keretrendszer által biztosított eszközök deklarációja (1.1. kódrészlet).

```
28 // represents the color of a pixel,
29 // color components are one-byte integers (0..255)
30 struct Color {
31     uint8_t r, g, b;
32 };
33
34 // can be used for logging status info
35 // (instead of cout)
36 extern std::ofstream logfile;
37
38 // drawing functions provided by the framework:
39
40 void draw_pixel(int row, int col, Color color);
41
42 void draw_text(int row, int col,
43                Color color, int size,
44                const char *text);
```

1.1. kódrészlet. A keretrendszer által biztosított eszközök

Az elején megadunk egy szín struktúrát, mely egy-egy pixel színét tudja tárolni. Ezt követi egy `ostream` típusú objektum, amibe naplóüzeneteket lehet írni, melyek az aktuális munkakönyvtárban létrehozott `logfile.txt` nevű állományba íródnak. Ide a keretrendszer is írhat üzeneteket (például helytelen koordinátákra való rajzolás esetén).

Ezek után a keretrendszer által biztosított rajzolófüggvények megadása következik. Az első függvénnyel egy adott pixelnek lehet a színét beállítani. A grafikus ablakot egy pixelekből álló mátrixként képzeljük el, így a pixel pozícióját a sor- és oszlopindexe azonosítja, színét pedig az előbb említett struktúra adja meg.

Mivel alacsony szinten akarjuk tartani a könyvtárat, az egyetlen rajzolófüggvényünk tulajdonképpen a `draw_pixel(...)` kellene legyen. Szerettünk volna ugyanakkor lehetőséget adni a diákoknak szöveges tartalom megjelenítésére is, de mivel a szöveg pixelekké alakítása már elég nehéz feladat, ezért erre is adunk kész függvényt (mely ugyanakkor az FLTK eszköztár hasonló funkcionalitására támaszkodik). Itt az előbbi függvény három paraméterén kívül még megadjuk a pixelekben kifejezett betűméretet, illetve a szöveget, amit ki akarunk írni. A sor- és oszlopindex ezen függvény esetén az első betű bal alsó sarkának koordinátáit jelenti.

A fejállomány következő része azon függvények bejelentését tartalmazza, melyek implementációja a felhasználó (diák) kódjában kell megjelenjen (1.2. kódrészlet). Az inicializáló függvényt csak a program futásának kezdetekor hívja majd meg a keretrendszer, ez alkalmas például bemeneti fájlok adatainak beolvasására vagy egyéb kezdőértékek beállítására.

```
47 // The following functions have to be implemented in user code:
48
49 // called once when the program starts
50 void initialize();
51
52 // called every time the window needs to be rendered,
53 // should render all graphical content via calls to
54 // draw_pixel and draw_text
55 void render(int width, int height);
56
57 // mouse events:
58 bool on_scroll(int delta); // delta can be positive or negative
59 bool on_move(int row, int col); // row and col are the current position
60 bool on_mouse_down();
61 bool on_mouse_up();
62
63 // key events, the key parameter can be any
64 // key code (including the constants below);
65 // regular characters have the same code as
66 // in ASCII
67 bool on_key_down(int key);
68 bool on_key_up(int key);
```

1.2. kódrészlet. Az implementálandó függvények

A `render(...)` függvény meg lesz hívva minden olyan esetben, amikor az ablak tartalmának kirajzolása szükségessé válik (ilyen az első megjelenítés, az átméretezések és minden olyan esemény, melynek következtében az operációs rendszer ablakkezelője ezt megköveteli). Minden meghívás előtt az ablak tartalma fehér pixelekkel inicializálódik, tehát minden elemet újból ki kell rajzolni (nem csak azt a részt, ami eddig nem létezett például egy átméretezés esetén).

A következő négy függvény az egéresemények kezelését szolgálja. Görgetéskor megkapjuk, hogy hány egységet mozgott a görgő (illetve az előjel megadja, hogy fel vagy le), mozgáskor azt, hogy éppen mely koordinátákon található az egér, illetve reagálhatunk a bal egérgomb megnyomására és felengedésére. A függvények visszatérési értéke azt jelzi, hogy újra ki kell-e rajzolni az ablak tartalmát az esemény feldolgozásának következtében. Ha `true` értéket térítünk vissza, akkor a keretrendszer újra meghívja a `render(...)` függvényt és frissíti a grafikus ablak tartalmát, ellenkező esetben mindez nem történik meg. Tehát ha valamely eseményre nem

szeretnénk reagálni, akkor az annak megfelelő függvény implementációja egyetlen sorból fog állni, melyben `false` értéket térítünk vissza.

A következő két függvény a neveiknek megfelelően a billentyűeseményekre való reagálást teszi lehetővé. Az általunk visszatérített érték hatása ugyanaz, mint az egéresemények esetén. A keretrendszer által küldött paraméter értéke általában a lenyomott vagy felengedett billentyűnek megfelelő karakter ASCII kódja, vagy valamilyen speciális érték a többi billentyű esetén. A fejállomány néhány speciális billentyűkódot tartalmazó konstanssal zárul.

A felhasználók által megírt kódot tehát a keretrendszer az itt felsorolt függvényeken keresztül hívja meg. Ezen függvények ugyanakkor használhatják a keretrendszer által rendelkezésükre bocsátott eszközöket (a rajzolófüggvények meghívásának természetesen csak a `render(...)` függvényben, vagy az innen meghívott segédfüggvényekben van értelme).

1.2. Fordítás, projektszerkezet

A keretrendszer implementációja az FLTK eszköztárra épülő adapter réteg, mely megtalálható a melléklet *src/main.cpp* állományában. Azért választottuk az FLTK eszköztárat mert a natív implementációval ellentétben nem kell külön kódot írjunk a két célplatformra. Natív megoldás esetén ugyanis a Microsoft Windows operációs rendszerek és a GNU/Linux disztribúciókra feltelepített X Window System teljesen különböző interfészeket adnak a programozóknak grafikus alkalmazások készítésére. Ugyanakkor más keretrendszerekhez képest az FLTK viszonylag kis méretű, és a nagyobb keretrendszerek által biztosított többletfunkcionalitásra nincs szükségünk.

Az iskolai környezetben való alkalmazhatóság nem csak a programozási nyelv felhasználható funkcióiban, hanem a függőségek és eszközök megválasztásakor is fontos szempont. Napjainkban a legtöbb iskolában a diákok Microsoft Windows operációs rendszer valamely verzióját használják az órákon. Előfordulhat, hogy a programcsomagok telepítésére viszont csak az arra kijelölt személynek van joga (egy-egy rendszergazdának), ezért szerettünk volna olyan megoldást találni, amihez nem kell adminisztrátori jogosultság a rendszeren.

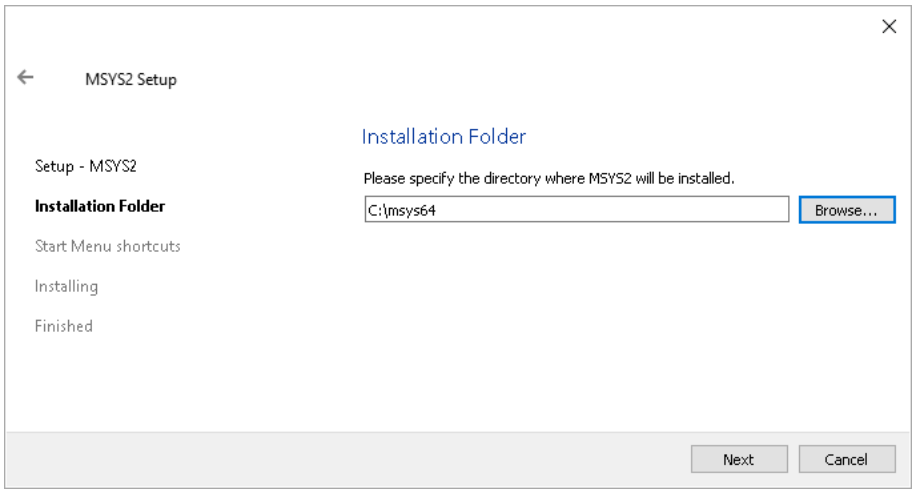
Az FLTK keretrendszer lefordítható forráskódból is adott célrendszeren (erre vonatkozó részletek megtalálhatók a fejlesztők által publikált dokumentációban[6]). A könnyebb telepítés

[6] https://www.fltk.org/documentation.php

végett viszont az MSYS2 eszközrendszer csomagjai között megtalálható bináris változatot[7] fogjuk használni.

Az MSYS2 olyan könyvtárak és build eszközök gyűjteménye, melyekkel natív Windows programokat lehet elkészíteni. Megfelel a célkörnyezetünknek, mert tetszőleges könyvtárba telepíthető, így telepítéséhez nem kell adminisztrátori jogosultság a rendszeren. Továbbá számos csomagot és függőséget telepíthetünk vele, többek között a fordítóprogramokat, a C++ standard könyvtárat és az FLTK keretrendszert is.

A projektjeink előkészítéséhez tehát először töltsük le a legfrissebb telepítőfájlt az MSYS2 weboldaláról[8], majd indítsuk el a telepítést tetszőleges útvonalat megadva célkönyvtárként (ld. 1.1. ábra).

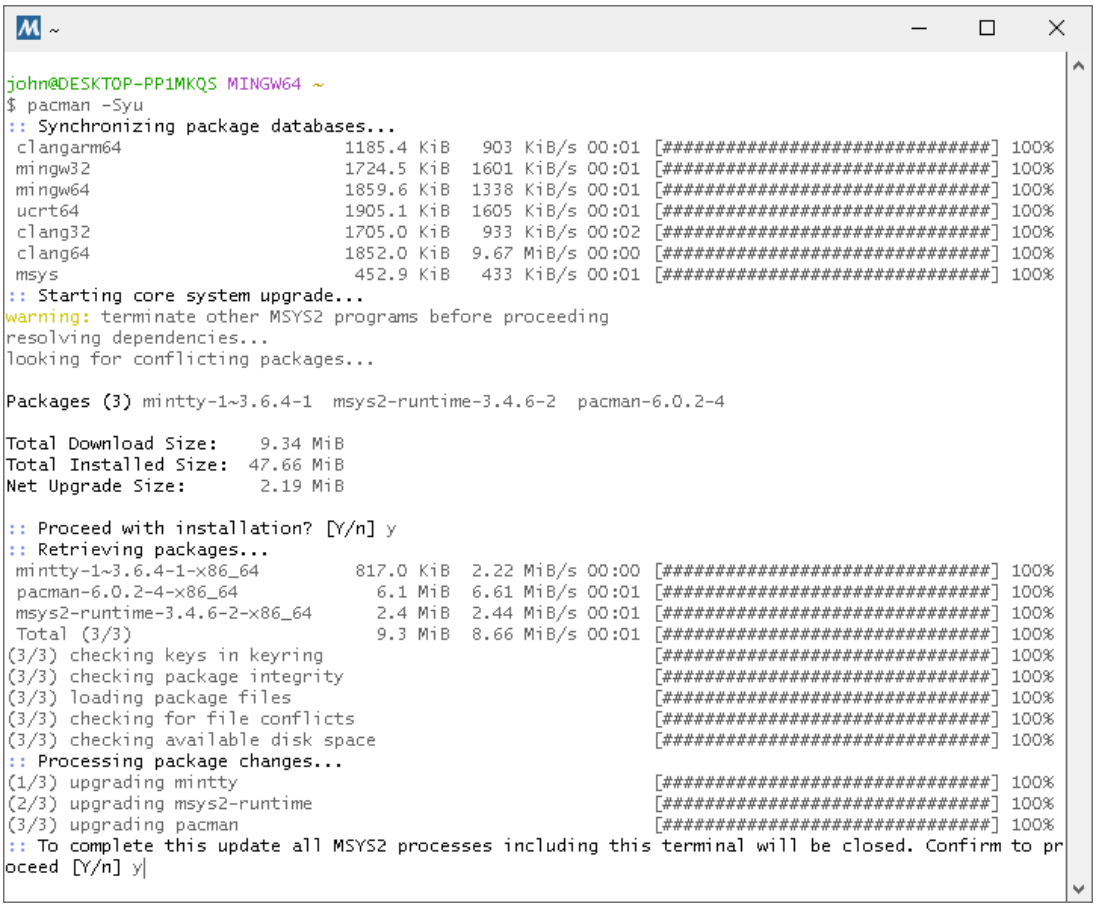


1.1. ábra. MSYS2 telepítési útvonal

A telepítőn végighaladva fogadjuk el az alapértelmezett beállításokat, a végén pedig ne hagyjuk megjelölve az MSYS2 futtatására vonatkozó jelölőnégyzetet.

Az MSYS2 csomagban több alrendszer található, mindegyik a neki megfelelő parancssorral és feltelepített szoftvercsomag-gyűjteménnyel. Ezek közül itt a *MinGW64* nevűt fogjuk használni, tehát indítsuk el a telepítési könyvtárból a `mingw64.exe` nevű programot, majd adjuk ki a `pacman -Syu` parancsot és hagyjuk jóvá a telepítést. Ez frissíti a csomagokról tárolt információkat és a rendszer saját csomagjait. A telepítés végén valószínűleg újra kell majd indítani a programot (ld. 1.2. ábra).

[7] https://packages.msys2.org/package/mingw-w64-x86_64-fltk
[8] https://www.msys2.org/



1.2. ábra. MSYS2 frissítése

Ezek után az újraindított MinGW parancssorban adjuk ki az alábbi parancsot és hagyjuk jóvá a telepítést:

```
pacman -Syu mingw-w64-x86_64-toolchain mingw-w64-x86_64-fltk
```

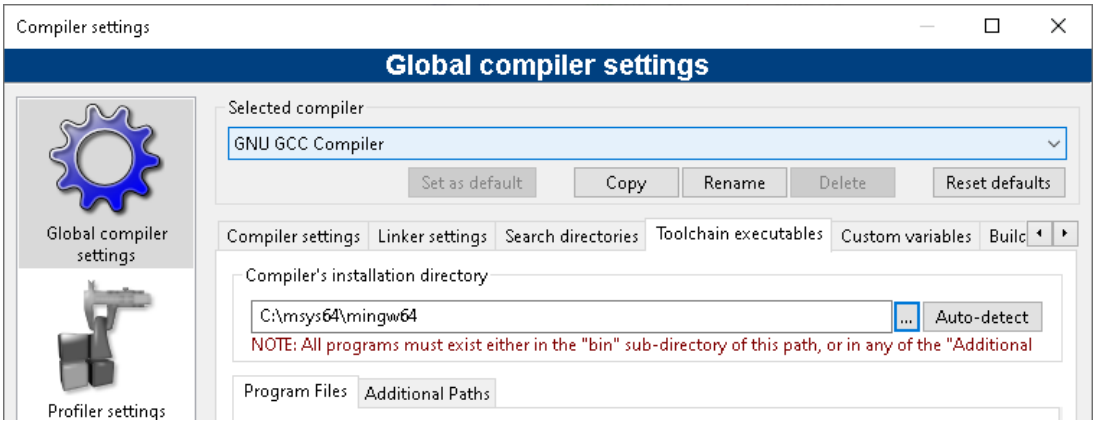
Ezzel feltelepítjük GCC fordítót és GDB debuggert (a *toolchain* csomag részeként), illetve az FLTK keretrendszert is.

A következőkben szükségünk lesz a Code::Blocks integrált fejlesztői környezetre, mely a legtöbb iskolában rendelkezésre áll (illetve a diákok gépére is fel van telepítve). Ha mégsem, akkor a weboldalukról[9] letölthető telepítő és telepítést nem igényló változat is (letöltéskor választhatjuk azon változatokat is, amelyekben nincs meg a MinGW csomag, hiszen az MSYS2 alatti fordítót fogjuk majd használni).

Következő lépésben a Code::Blocks-ban beállítjuk, hogy az MSYS2 alatti fordítót és debuggert szeretnénk használni. Ezért először a Settings menü Compiler menüpontját választva

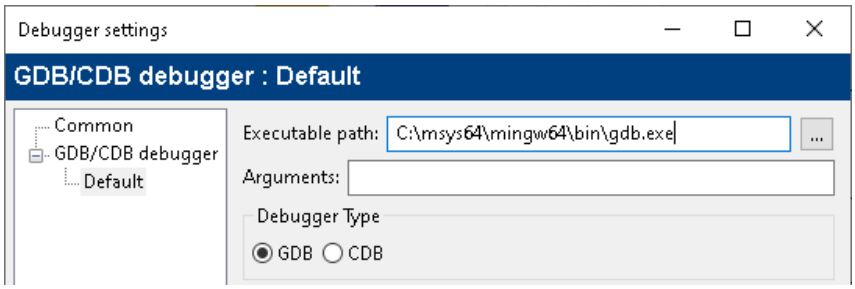
[9] https://www.codeblocks.org/

a megjelenő beállításablak *Toolchain executables* fülén beállítjuk a GCC fordítónk telepítési útvonalát (ld. 1.3. ábra).



1.3. ábra. Fordító beállítása Code::Blocks-ban (amennyiben az MSYS2 rendszert a `C:\msys64` útvonalra telepítettük)

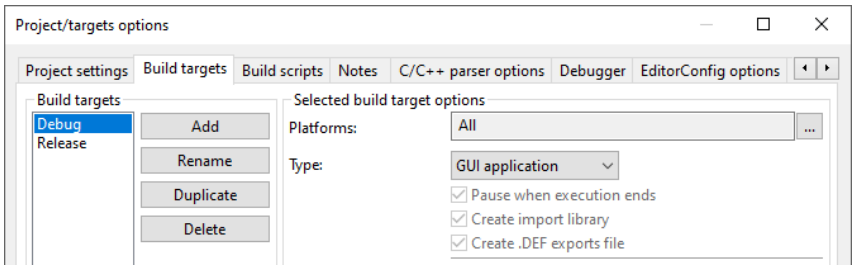
A debugger elérési útvonalát hasonlóan megadhatjuk a Settings / Debugger menüpontra kattintva, itt már a `gdb.exe` útvonalát kell beállítani (ld. 1.4. ábra).



1.4. ábra. Debugger beállítása Code::Blocks-ban (amennyiben az MSYS2 rendszert a `C:\msys64` útvonalra telepítettük)

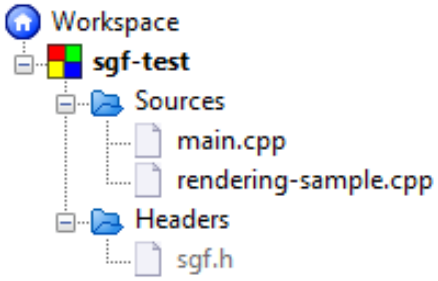
Ezen beállítások után elkészíthetjük az első Code::Blocks projektet, ami használja a keretrendszerünket. Készítsünk egy új projektet az *Empty project* sablont választva a varázslóból, majd a Project / Properties menüpontra kattintva felnyíló ablak *Build targets* fülén állítsuk át a típust *Console application*-ről *GUI application*-re úgy a Debug, mint a Release konfiguráció esetén (ld. 1.5. ábra). Ez azért szükséges, hogy ne nyíljon meg minden indításkor egy üres konzolablak is a grafikus programunk mellett.

Másoljuk be a mellékletben található *src/main.cpp*, *src/sgf.h* és *src/examples/rendering-sample.cpp* fájlokat a Code::Blocks projekt könyvtárába, majd adjuk hozzá őket a projekthez (utóbbit megtehetjük jobb egérgombbal kattintva a projekt nevére és az *Add files* menüpontot



1.5. ábra. Projekt típusának beállítása Code::Blocks-ban

választva). Ezek után az 1.6. ábrának megfelelő projektszerkezet fog látszani a bal oldali panelen.



1.6. ábra. Code::Blocks projekt szerkezete

Még el kell végezzünk két beállítást azért, hogy egyrészt fordításkor legyenek elérhetőek az FLTK fejállományai, másrészt pedig a szerkesztő számára legyenek elérhetőek az FLTK bináris állományai.

Futtassuk az előbb használt `mingw64.exe` parancssorban az `fltk-config --cxxflags` parancsot és ennek kimenetét másoljuk be a vágólapra. A Code::Blocks projektünk Project / Build options menüpontján megnyíló ablak bal oldali panelén válasszuk ki a projekt nevét (azért, hogy ne csak az egyik build konfiguráció beállításait módosítsuk, hanem mindkettőt), majd az *Other compiler options* fül alatti mezőbe illesszük be az előbb bemásolt kimenetet, kitörölve belőle a `-O2` opciót (azaz a könnyebb debuggolás céljából egyenlőre nem kapcsoljuk be a GCC optimizációkat).

Majd futtassuk szintén a MinGW parancssorban az `fltk-config --ldflags` parancsot és kimenetét másoljuk be a vágólapra, aztán illesszük be az előbbi beállításablak *Linker settings* fülének *Other linker options* mezőjébe, és írjuk még a végére a `-lm` opciót.

Mentsük le az beállításablakon elvégzett módosításokat, majd a projekt *rendering-sample.cpp* nevű fájljában az *sgf.h* fejállománynak megfelelő `#include` direktívából töröljük

ki a `../` előtagot (ez ugyanis csak azért került oda, mert a mellékletben a példaprogramok egy alkönyvtáron belül helyezkednek el).

Ezek után kellene működjön a program fordítása és futtatása, egy grafikus ablak kell megjelenjen, melyben a „Hello, world!” szöveg látszik.

Mivel a projekt elkészítési és konfigurálási lépéseinek végrehajtásakor sok a hibalehetőség, ezeket legközelebb már megspórolhatjuk, ha a projektfájlt (`*.cbp`) és a forrásfájlokat lemásoljuk, majd a projektfájlt szövegszerkesztővel megnyitva kicseréljük a kezdeti projekt nevének minden előfordulását a kívánt új projektnévre, illetve ennek megfelelően átnevezzük magát a projektfájlt is.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<CodeBlocks_project_file>
    <FileVersion major="1" minor="6" />
    <Project>
        <Option title="sgf-test-2" />
        <Option pch_mode="2" />
        <Option compiler="gcc" />
        <Build>
            <Target title="Debug">
                <Option output="bin/Debug/sgf-test-2" prefix_auto="1"
                    extension_auto="1" />
                <Option object_output="obj/Debug/" />
                <Option type="0" />
                <Option compiler="gcc" />
                <Compiler>
                    <Add option="-g" />
                </Compiler>
            </Target>
            <Target title="Release">
                <Option output="bin/Release/sgf-test-2" prefix_auto="1"
                    extension_auto="1" />
                <Option object_output="obj/Release/" />
```

1.3. kódrészlet. Részlet a Code::Blocks projektfájlból

Amennyiben szeretnénk a létrejött végrehajtható programot egyenesen a fájlrendszerből vagy akár más számítógépen elindítani, akkor az MSYS2 telepítési könyvtárának *mingw64/bin* alkönyvtárából mellé kell másolni a szükséges run-time függőségeket (libfltk.dll, libgcc_s_seh-1.dll, libstdc++-6.dll és libwinpthread-1.dll). Amennyiben valamelyikük hiányzik, egy Windows hibaablak értesít majd erről minket.

Ha a projektet GNU/Linux operációs rendszer alatt akarjuk használni, akkor annak függvényében, hogy van-e jogunk csomagokat telepíteni a rendszerre, egyik lehetőség, hogy az adott

disztribúció szoftvergyűjteményéből feltelepítjük az FLTK keretrendszer csomagját (pl. https://archlinux.org/packages/community/x86_64/fltk/, https://packages.ubuntu.com/jammy/libfltk1.3-dev stb.), vagy pedig lefordítjuk kézzel forráskódból a dokumentációban leírtaknak megfelelően (ld. https://www.fltk.org/doc-1.3/intro.html). Ezek után a Code::Blocks projekt felépítése hasonló, annyi különbséggel, hogy MinGW helyett az adott rendszerre feltelepített parancssort, fordítót és debuggert használjuk.

A mellékletben az *src* nevű könyvtárban megtalálható a keretrendszer és a példaprogramok forráskódja mellett egy *Makefile* is, ennek segítségével a GNU Make build rendszert is használhatjuk programjaink lefordítására (amennyiben az FLTK telepítése már megtörtént), illetve egyszerre lefordíthatjuk a példaprogramokat, melyek között szerepel a következő fejezetekben tárgyalt feladatok egy részének megoldása is. Ezt inkább tanároknak ajánljuk, hiszen a diákok esetében pont az lenne a lényeg, hogy ők jöjjenek rá a feladatok megoldására, ne kapják meg készen azokat.

Szintén a melléklet része az FLTK jelenlegi verziójának (1.3.8) forráskódja és dokumentációja az *fltk* könyvtárban, továbbá egy *bin-win64* nevű könyvtár a példaprogramok bináris változatával (64 bites Windows célrendszerre lefordítva) és a futtatáshoz szükséges függőségekkel.

2. fejezet

Gyakorlatok, feladatok

Ebben a fejezetben bemutatunk néhány különböző nehézségű feladatot, melyek megoldhatók a keretrendszer segítségével. A tanár szemszögéből tárgyaljuk bizonyos feladatok egy-egy megoldási ötletét is, a megoldás leírásában megjelenő gondolatok és kódrészletek közül a diákoknak általában csak annyit érdemes odaadni, amennyire szükség van (vagyis szeretnénk, ha a megoldások minél nagyobb részben tőlük származnának). A feladatokat felhasználó tanár a körülményeknek megfelelően eldöntheti, hogy mit és mennyit mond el a megoldás menetéből.

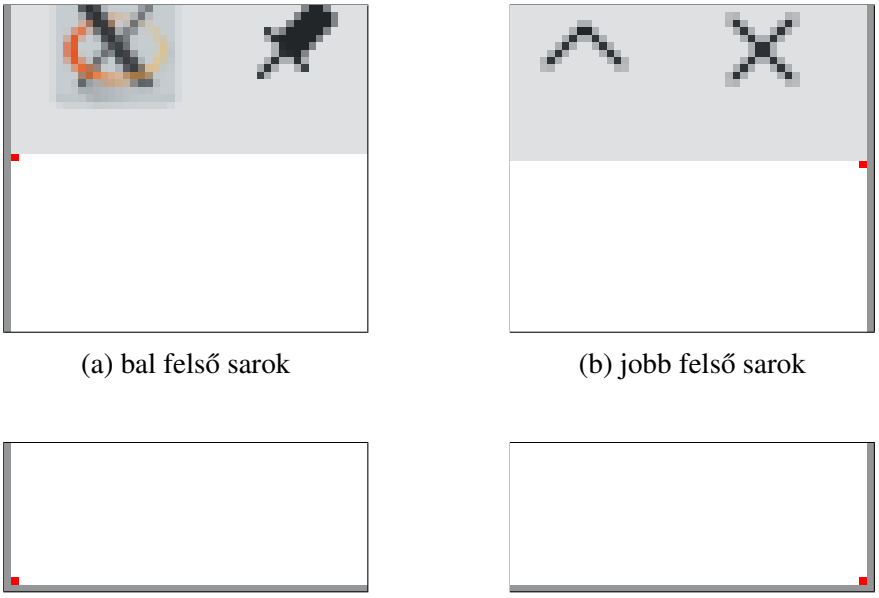
Mivel egy-egy pixel színét egy struktúra tárolja, illetve a képernyőre rajzolás C++ függvények meghívásán keresztül történik, ezért a jelenlegi informatika tantervnek megfelelően ezeket a feladatokat leginkább 11. osztályban, illetve az intenzív informatika tantervnek megfelelően már 10. osztályban ajánlott alkalmazni, miután a szükséges nyelvi elemeket a diákok megismerték. A feladatok alkalmat adnak úgy az általános gondolkodási képesség fejlesztésére, mint a korábban elsajátított ismeretek felhasználására (pl. struktúrák és karakterláncok alkalmazása).

2.1. A példakód közelebbről

Nézzük meg az előző fejezetben futtatott példakód `render` függvényét (2.1. kódrészlet). Figyeljük meg, hogy a megjelenő ablakban látható „Hello, world!” szöveget kirajzoló `draw_text` függvényhívás mellett még négy darab pixelt is kirajzolunk az ablak négy sarkának megfelelő sor- és oszlopkoordinátákra. Ezek színe piros, mert a `Color` struktúra *r*, *g* és *b* adattagjainak ebben a sorrendben a 255, 0, 0 értékeket adjuk (a színkomponensek értéke egy-egy 0 és 255 közötti egész szám kell legyen). Ha a megjelenő ablakról képernyőképet készítünk, veszteségmentesen lementjük (például PNG formátumban), majd egy képszerkesztővel ránagyítunk, akkor a címsor alatti rész négy sarkában megfigyelhető a négy piros pixel (ld. 2.1. ábra).

```
30 void render(int width, int height)
31 {
32     draw_pixel(0, 0, {255,0,0});
33     draw_pixel(height-1, 0, {255,0,0});
34     draw_pixel(0, width-1, {255,0,0});
35     draw_pixel(height-1, width-1, {255,0,0});
36
37     draw_text(40, 20, {0,0,0}, 20, "Hello, world!");
38 }
```

2.1. kódrészlet. A példakód `render(...)` függvénye



(a) bal felső sarok

(b) jobb felső sarok

(c) bal alsó sarok

(d) jobb alsó sarok

2.1. ábra. Nagyított képernyőfelvételek a grafikus ablak sarkairól

A keretrendszer által nyújtott funkcionalitás jobb megértése érdekében elvégezhetjük a következő gyakorlatokat:

1. Nyissuk meg a színpalettát egy tetszőleges képszerkesztőben (pl. GIMP, MS Paint) és keressük meg, hogy honnan tudjuk adott szín piros, zöld és kék komponenseinek értékét leolvasni. Válasszunk egy színt és színezzük ki ezzel a programunkban a teljes hátteret. Próbáljuk megadni a színek egyes komponenseit hexadecimális konstansként is (például 255 helyett a `0xFF` literált használva).

2. Helyezzünk el egy `logfile << "render "<< width << "x"<< height << endl;` kiírást a `render` függvény elejére, majd a program elindítása után végezzünk el egy-egy átmé-

retezést. Code::Blocks-ból való futtatás esetén a projektkönyvtárban, ellenkező esetben pedig az aktuális katalógusban létrejött-e egy *logfile.txt* nevű állomány, melyben nyomon követhetjük, hogy hányszor és milyen méretekkel volt meghívva a `render` függvény. A kiírásban az `endl` fontos, mert új sorra térés mellett még a kimeneti puffert is üríti, így rögtön az utasítás végrehajtása után a fájlba kerül az kiírt szöveg.

3. Helyezzünk el megfelelő kiírásokat az egér- és billentyűesemények kezelőfüggvényeiben is, majd a naplófájl tartalma alapján figyeljük meg, hogy melyik függvényt mikor és milyen értékekkel hívja meg a keretrendszer.

2.2. Négyzetek rajzolása

A következő feladat, hogy rajzoljunk ki a grafikus ablakba egy sötétzöld négyzetet, a lehető legnagyobb méretben és középre igazítva. A négyzet oldala nem lehet nagyobb sem az ablak szélességénél, sem a magasságánál. Ha az ablak belseje nem pont négyzet alakú, akkor vagy a négyzet két oldalán, vagy pedig alatta és felette valamekkora részt üresen kell hagyni.

Egy lehetséges megoldás található az alábbi kódrészletben, a kimenet két különböző ablakméret esetén pedig a 2.2. ábrán.

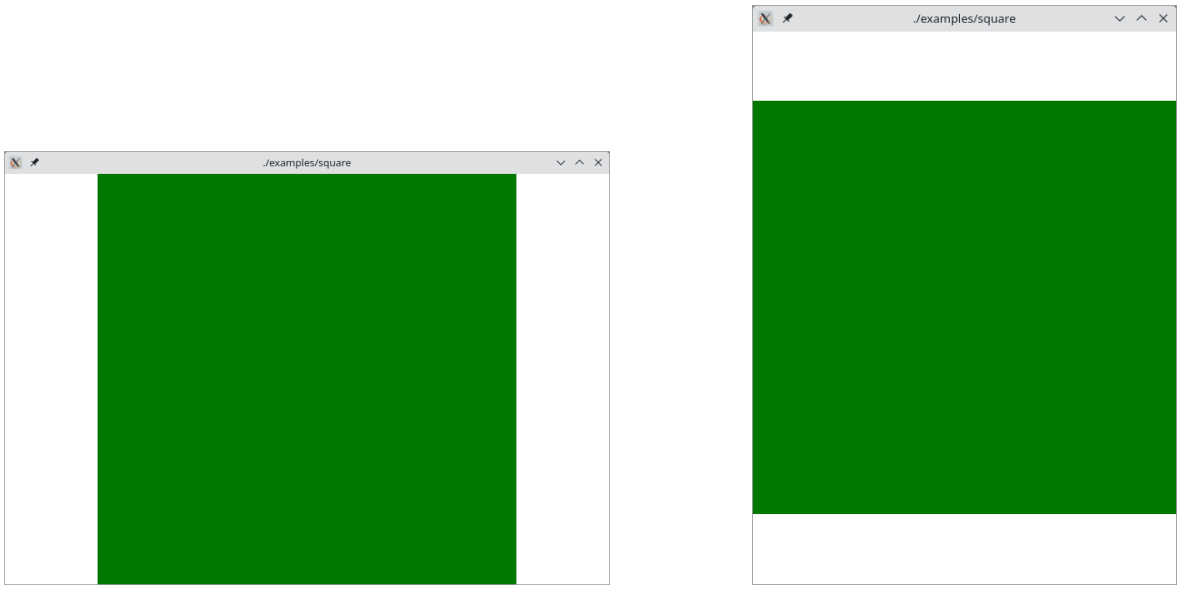
```
31 void render(int width, int height)
32 {
33     int side = min(width, height);
34     int margin_cols = (width - side)/2;
35     int margin_rows = (height - side)/2;
36 
37     for (int i = 0; i < side; ++i)
38         for (int j = 0; j < side; ++j)
39             draw_pixel(margin_rows + i, margin_cols + j, { 0, 120, 0 });
40 }
```

2.2. kódrészlet. Zöld négyzet kirajzolásához használt `render` függvény

Ha ez sikerült, akkor a négyzetes mátrixokban a főátló és mellékátló fogalmának átismétlésére javasolhatjuk például a következőket:

1. Színezzük ki a négyzet főátló feletti és főátló alatti részét két különböző színnel.

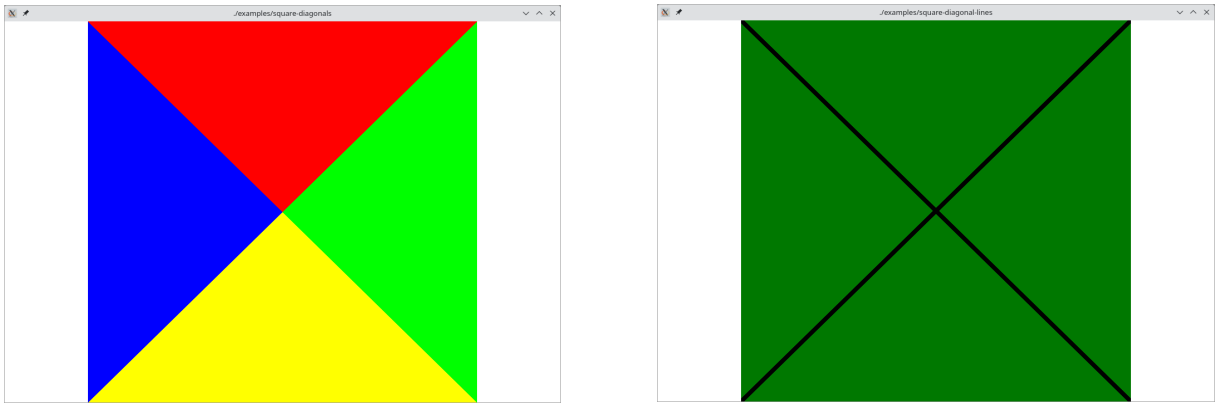
2. Az előbbi két zónában még tegyünk különbséget a mellékátló feletti és alatti részek között



2.2. ábra. Zöld négyzetek

is, állítsunk elő a 2.3a. ábrához hasonló kimenetet. A főátlón és mellékátlón levő pixelek színe megegyezhet vagy a felettük, vagy az alattuk található zóna színével.

3. Színezzük feketére a főátló és mellékátló vonalaihoz közel álló pixeleket a 2.3b. ábrának megfelelően (a vastagság lehet tetszőleges vagy felnagyítva megszámolhatjuk az ábrán a fekete pixeleket minden soron).

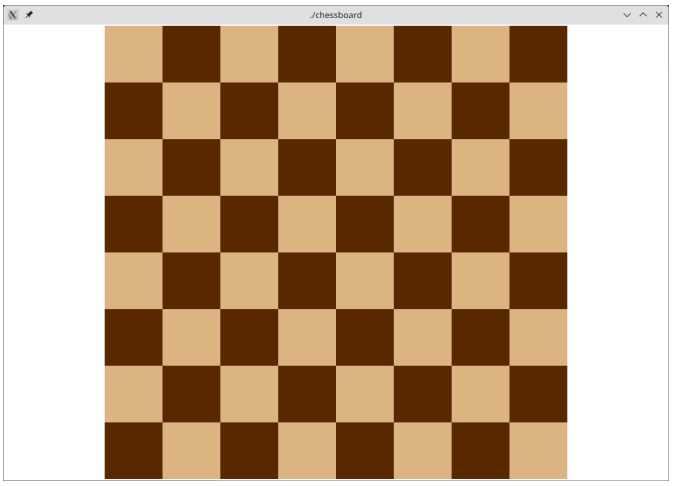


(a) Felosztás főátló és mellékátló mentén (b) Átlóvonalak

2.3. ábra. Négyzetekkel kapcsolatos feladatok kimenetei

2.3. Rajzoljunk sakktáblát

Ebben a feladatban egy fokkal több tervezésre lesz szükség a rendelkezésre álló hely beosztásához. Egy sakktáblát szeretnénk kirajzolni a 2.4. ábrának megfelelően.



2.4. ábra. Sakktábla

Első lépésben megnézhetjük, hogy legfeljebb mekkora lehet a négyzetünk. Az előző feladatban szereplő két megkötésen kívül (az oldalhossz nem lehet nagyobb a szélességnél és a magasságnál) még azt is szeretnénk, ha az oldalhossz nyolc többszöröse lenne, hogy mindegyik kis négyzet ugyanakkora legyen. Miután kiszámoltuk a szükséges hosszúságokat, egy világos és egy sötét színt választva néhány ciklus segítségével kirajzolhatjuk a pixeleket (egy lehetséges megoldás a 2.3. kódrészletben).

Javasoljuk még a következő feladatokat:

1. Általánosítsuk a rajzolást $n \times n$ méretű tábla esetére (ahol az n pozitív egész számot az `initialize` függvényen belül egy fájlból olvassuk be).

2. Próbáljuk meg eltüntetni a maradékos osztás miatt megjelenő vékony fehér csíkokat (a 2.4. ábrán ezek a sakktábla felett és alatt látszanak). Ezt megoldhatjuk például úgy, hogy az osztási maradéknak megfelelő pixeleket az első vagy utolsó sor négyzeteinek magasságához adjuk hozzá, vagy elosztjuk azokat majdnem egyenlően a négyzetek között (persze nem jut majd mindegyiknek).

3. Oldjuk meg, hogy kattintáskor a világos és sötét színek felcserélődjenek a táblán (újabb kattintáskor pedig visszaálljanak eredeti állapotukba. Ehhez implementáljuk az

`on_mouse_down` eseménykezelőt, mely egy globális változóban számolja a kattintások számát (vagy nyilvántartja azok paritását) és `true` értéket térít vissza. Majd a `render` függvényben a kattintások számának megfelelően cseréljük fel a színeket, ha szükséges.

```
32 void render(int width, int height)
33 {
34     int side = min(width, height)/8;
35     int margin_cols = (width - 8*side)/2;
36     int margin_rows = (height - 8*side)/2;
37
38     Color light { 220, 180, 130 };
39     Color dark { 90, 40, 0 };
40
41     for (int i = 0; i < 8; ++i)
42         for (int j = 0; j < 8; ++j) {
43             Color color = (i+j)%2 == 0 ? light : dark;
44             for (int row = 0; row < side; ++row)
45                 for (int col = 0; col < side; ++col)
46                     draw_pixel(
47                         margin_rows + i*side + row,
48                         margin_cols + j*side + col,
49                         color);
50         }
51 }
```

2.3. kódrészlet. Sakktábla kirajzolásához használt `render` függvény

2.4. Színátmenetek

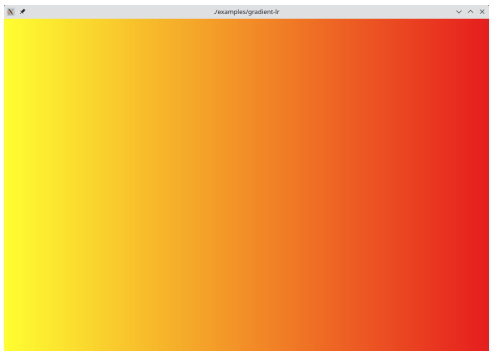
A következő feladattal szeretnénk az eddig leginkább csak konstans értékként használt színek algoritmikus manipulálását gyakoroltatni.

Írjunk egy színkeverő függvényt, mely két színt és egy valós számot kap paraméterként és egy (kevert) színt térít vissza. A valós szám a $[0, 1]$ intervallum eleme és azt szeretnénk, ha a függvény a két színt $(1 - r) : r$ arányban keverné (például $r = 0$ esetén az első színt, $r = 1$ esetén a második színt, $r = 0.5$ esetén a két szín számtani középarányosát térítse vissza, $r = \frac{1}{3}$ esetén pedig azt a színt, mely a két szín $\frac{2}{3}$-dal és $\frac{1}{3}$-dal súlyozott számtani középarányosa). A színekkel végzett műveleteket komponensenként értelmezzük, tehát külön végezzük el a keverést az egyes komponensekre. Egy lehetséges implementációja a színkeverő függvénynek a 2.4. kódrészletben látható. A keverés komponensenként történik és az eredmény nyolc bites egésszé való kerekítését (pontosabban csonkolását) a beépített típuskonverzióra bízzuk.

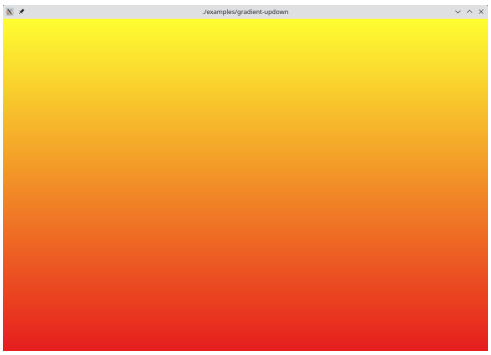
```
1 Color combine(Color c1, Color c2, double ratio)
2 {
3     Color c;
4     c.r = c1.r * (1-ratio) + c2.r * ratio;
5     c.g = c1.g * (1-ratio) + c2.g * ratio;
6     c.b = c1.b * (1-ratio) + c2.b * ratio;
7     return c;
8 }
```

2.4. kódrészlet. Színkeverő függvény

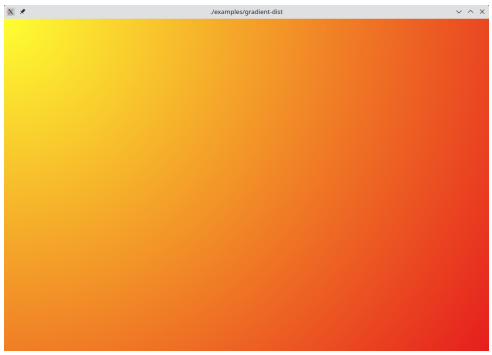
Ezt a keverőfüggvényt felhasználva állítsunk elő a 2.5. ábrán látható színátmeneteket. Mind a négy képernyőfelvételen egy sárga ({255,255,50}) és egy piros ({230,30,30}) szín közötti átmenet látható.



(a) balról jobbra



(b) fentről le



(c) bal felső saroktól mért távolság

(d) középponttól mért távolság

2.5. ábra. Színátmenetek

Az első ablak esetén balról jobbra haladva szeretnénk egyik színből a másikba átmenni, tehát a keverési arány meghatározásában csak az oszlopindex játszik szerepet. A második ablak esetén ugyanakkor fentről lefelé halad az átmenet, ezért itt csak a sorindexnek van hatása az

arányra. A harmadik átmenet előállításakor a pixelek bal felső saroktól mért távolságának és a lehetséges legnagyobb ilyen távolságnak (azaz a jobb alsó sarok távolságának) az arányát használtuk keverési arányként, míg a negyedik átmenet esetén a középponttól mért távolságot arányítjuk a lehetséges legnagyobbhoz (azaz valamelyik sarok középponttól mért távolságához).

A négy átmenet egy-egy lehetséges implementációja megtalálható a melléklet *src/gradient-lr.cpp*, *src/gradient-updown.cpp*, *src/gradient-dist.cpp* és *src/gradient-circular.cpp* állományaiban.

A *CSS*[1]-ben megszokott színátmenetekből inspirálódva egy fokkal nehezebb feladatokat is javasolhatunk. Egyik ötlet, hogy implementáljuk az előbb bemutatottakhoz hasonló átmeneteket több mint két színnel. Például három szín esetén a balról jobbra haladó átmenet a legbaloldalibb oszloptól a középsőig átmegy az első színből a másodikba, a középső oszloptól a jobboldaliig pedig a második színből a harmadikba. A színek számát, a színkódokat és az átmenet típusát megadhatjuk egy bemeneti fájlban. Másik nehezítési lehetőség, hogy a balról jobbra és fentről lefelé haladó átmenetek általánosításaként állítsunk elő adott szögben eldőlt lineáris színátmenetet.

2.5. Ceruza

Az alábbiakban szeretnénk a rajzolóprogramokban általában *ceruza* néven ismert funkcióhoz hasonló viselkedést előállítani. Tehát amíg a felhasználó lenyomva tartja a bal egérgombot, az egér mozgását követő vonal jelenjen meg a vásznon.

Ezt megvalósíthatjuk a keretrendszer által kezelt egéreseményekkel, de szükséges lesz tárolni az aktuális képet, hiszen esetleges újrarajzolásokkor már nem fogjuk tudni az egér régebbi pozícióit. Tehát vegyünk egy elég nagy (pl. 1920×1080-as vagy akár még nagyobb felbontásnak megfelelő) kétdimenziós tömböt, és ebben fessük be a pixeleket valahányszor az egér úgy halad át egy pixelen, hogy a bal egérgomb le van nyomva (illetve akkor is befesthetjük az aktuális pixelt, hogyha éppen most nyomódik le). A `render` függvényben meg csak rajzoljuk ki a grafikus ablakba a lementett kép azon részét, amelyik elfér benne (például a bal felső saroktól indulva). Az előbbiek alapján az első implementációnk a melléklet *src/examples/pencil-1.cpp* állományában található és a 2.6a ábrának megfelelő viselkedést látjuk, ha megpróbálunk rajzolni valamit az egérrel.

[1] Cascading Style Sheets, https://www.w3.org/TR/CSS/

(a) Egy-egy pixel rajzolása

(b) 3×3-as négyzetek rajzolása

2.6. ábra. Ceruza kezdeti implementációi

Ez a vonal túl vékony, így alig látható, ezért próbáljunk meg ne csak egy-egy pixelt elhelyezni a képen az egéreseményekor során, hanem mondjuk egy-egy 3×3-as, vagy akár tetszőleges méretű négyzetet, melynek közepe (vagy páros méret esetén egyik középponti pixelje) az egér pozíciója. Ennek implementációja a melléklet *src/examples/pencil-2.cpp* állományában található, egy rajz pedig a 2.6b ábrán.

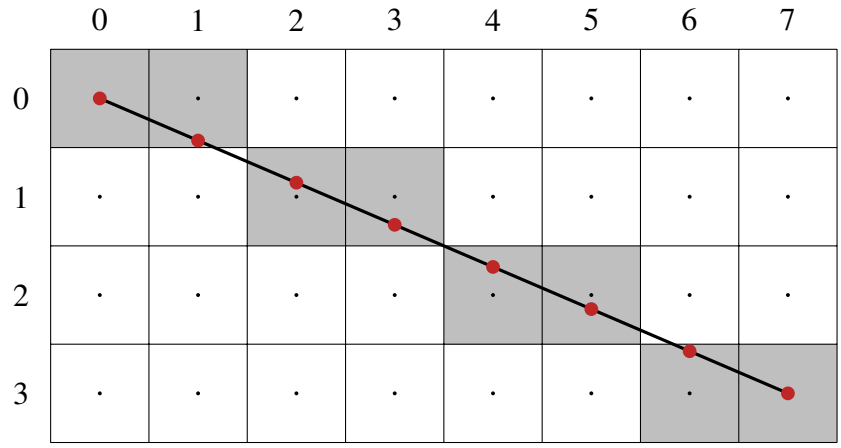
Még mindig hagy kívánni valót maga után a rajzolóprogramunk viselkedése, ugyanis az egérrel leírt görbe azon részein, ahol viszonylag gyorsan mozgattuk az egeret, szaggatott lett a kirajzolt vonal. Ezt az okozza, hogy ilyen esetekben nem hívódik meg az `on_move` függvény minden pixelre, amin az egér áthalad, ezek közül csak néhányat kapunk meg.

Ki kell találjunk tehát valamilyen megoldást arra, hogy amennyiben az egérgomb nem lett felengedve, de nem két egymás melletti koordinátapár érkezik be az `on_move` függvény két egymásutáni meghívásába, akkor a két pont közötti helyet valamilyen módon töltsük ki. Az első ötlet, hogy egy egyenes szakasz mentén színezzük be a pixeleket, viszont mivel ezek csak egész koordinátákra kerülnek, ezért egy kis kerekítésre is szükség lesz.

Tegyük fel például, hogy az első hívásban a $(sor_1, oszlop_1) = (0, 0)$ pozíciót, a másodikban pedig a $(sor_2, oszlop_2) = (3, 7)$ pozíciót kapjuk meg az `on_move` függvény paraméterein keresztül. Ha a szakasz függőleges vagy vízszintes lenne, akkor egyértelmű, hogy mely pixeleket kell beszínezni. Jelen esetben viszont ez nem áll fenn, úgyhogy írjuk fel a végpontokra illeszkedő egyenes explicit egyenletét kifejezve például a sorindexet az oszlopindex segítségével:

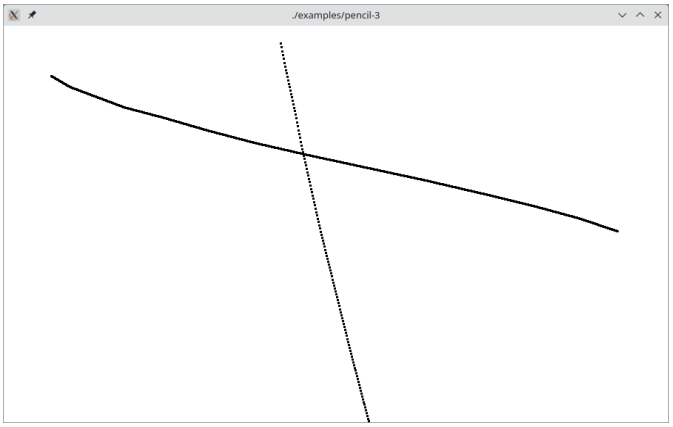
$$sor = \frac{sor_2 - sor_1}{oszlop_2 - oszlop_1} \cdot oszlop + \frac{sor_1 oszlop_2 - sor_2 oszlop_1}{oszlop_2 - oszlop_1}.$$

Ha ebbe az egyenletbe rendre behelyettesítjük az $oszlop_1$ és $oszlop_2$ közötti egész számokat, akkor megkapjuk, hogy azon paraméterértékekre melyik sorindexet kellene választani, viszont ez nem lesz mindig egész szám, ezért kénytelenek vagyunk kerekíteni (ld. 2.7. ábra). Ezt úgy is felfoghatjuk, hogy azt a pixelt fogjuk beszínezni, amelynek a középpontja közelebb van az egyenesen található ponthoz (amennyiben a pixelek középpontjait tekintjük a sík egész koordinátájú pontjainak).



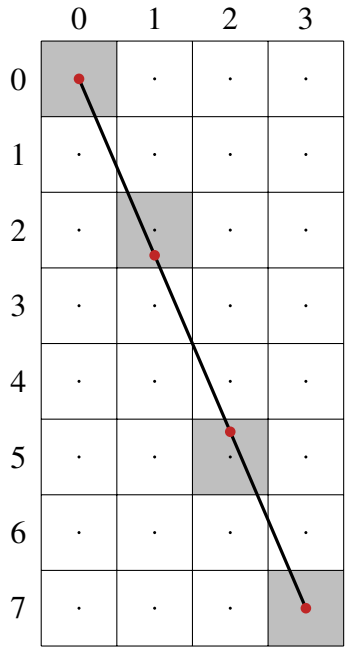
2.7. ábra. Pixelek színezése egy egyenes mentén

Ezen ötlet megvalósítása során tehát a diákok analitikus mértan ismereteiket is fel kell használják a cél eléréséhez. Egy lehetséges implementáció a melléklet *src/examples/pencil-3.cpp* állományában található, azonban ennek viselkedése még mindig nem kielégítő. Figyeljük meg, hogy bár a kevésbé meredek vonalak folytonosnak látszanak, ha meredekebben próbálunk rajzolni (lefelé vagy felfelé), még mindig bizonyos helyeken szaggatott lesz a vonal (ld. 2.8. ábra).



2.8. ábra. Egérpozíciók közötti helyek kitöltése egyenes mentén

Lássunk egy példát arra, hogy ez miként történhet meg. Tegyük most fel, hogy az első hívásban a $(sor_1, oszlop_1) = (0, 0)$ pozíciót, a másodikban pedig a $(sor_2, oszlop_2) = (7, 3)$ pozíciót kapjuk meg az `on_move` függvény paraméterein keresztül. Az előző megközelítést használva a 2.9. ábrának megfelelő pixeleket színezné ki a kódunk, viszont látszik, hogy hézagok maradnak: nem minden 0 és 7 közötti egész szám jelenik meg (kerekített) sorindexként, az egyenes ehhez túl meredek.



2.9. ábra. Pixelek színezése egy egyenes mentén

Ezen probléma elkerülésére az egyik módszer, hogy ilyenkor fordítva írjuk fel az egyenes egyenletét, tehát az oszlopindexet fejezzük ki a sorindex függvényében, majd minden sorindexen végighaladva kerekítés után megkapjuk, hogy ezen a soron melyik oszlopindexű pixelt kell kifesteni (vagy esetleg több pixelt az adott pozíció körül, lásd a vastagság tárgyalását a feladat elején). Az egyenlet alakja ekkor:

$$oszlop = \frac{oszlop_2 - oszlop_1}{sor_2 - sor_1} \cdot sor + \frac{oszlop_1 sor_2 - oszlop_2 sor_1}{sor_2 - sor_1},$$

és az egyenes meredekségének függvényében vagy ezt, vagy az előzőt használjuk majd. Pontosabban, ha két egymás után megkapott pozíció esetén a sorindexek különbsége nagyobb abszolút értékű mint az oszlopindexek különbsége, akkor ezt a felírást (ilyenkor az egyenes meredek), ellenkező esetben pedig az előzőt.

Ez a módszer digital differential analyzer néven ismert (lásd például [Godse és Godse, 2007], 2.3.1. rész), egy lehetséges implementációját a melléklet *src/examples/pencil-4.cpp* állománya tartalmazza. Az alábbiakban javasolunk még néhány feladatot:

1. Az „R” billentyű lenyomására töröljük a képernyő tartalmát, az „1”, „2”, stb. gombok megnyomása esetén változtassuk az aktuális vonalvastagságot, a „P”, „Z”, „K”, „F” billentyűk lenyomása esetén pedig változtassuk az aktuális színt (pirosra, zöldre, kékre, illetve feketére) és az ez után következő vonalakat az így kapott vastagsággal és színnel rajzoljuk.

2. Rajzoljunk ki egy (megfelelő színű és méretű) pöttyöt az egér aktuális pozíciójára (ez nem része a képnek, csak egérgomb lenyomása esetén váljon majd azzá).

3. A raszterezéshez (a kifestendő pixelek megállapításához) a digital differential analyzer helyett implementáljuk Bresenham algoritmusát (ld. [Bresenham, 1965] vagy [Godse és Godse, 2007], 2.3.2. rész).

2.6. Szövegvezérelt rajzolóprogram

A vektorgrafikát használó rajzolórendszerek egy része szöveges formátumban megadott bemenettel dolgozik. Ennek egyik legelterjedtebb példája az SVG (Scalable Vector Graphics)[2] néven ismert fájlformátum. Jelen feladatban olyan program előállítását tűzzük ki célul, ami egy ennél sokkal egyszerűbb formátumú szöveges bemenet alapján állítja elő a képet megfelelő `draw_pixel` hívások során.

Minden kirajzolandó geometriai alakzat esetén adott annak típusa, az elhelyezéshez szükséges információk és a kitöltés színe. Tekintsük például az alábbi három alakzatot:

– *téglalap*: a megadáshoz elég ha megadjuk a bal felső és jobb alsó sarok koordinátáit, illetve a kitöltés színét;

– *kör(lap)*: a megadásához szükség van a középpont koordinátáira, a sugárra és a kitöltés színére;

– *háromszög(lap)*: ennek megadásakor leírjuk a csúcsok koordinátáit és a kitöltés színét.

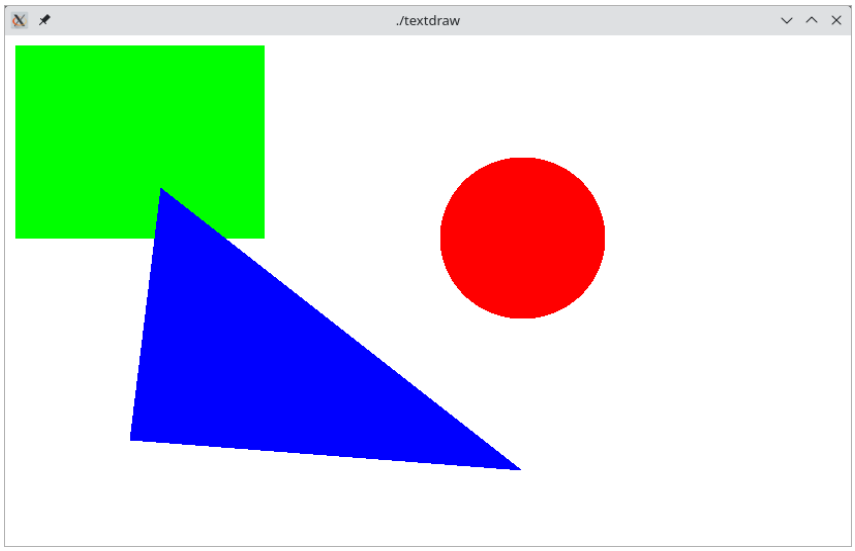
[2] https://www.w3.org/Graphics/SVG/

Az egyszerűség kedvéért az előbb felsorolt adatokat szóközzel elválasztott egész számok formájában felsorolhatjuk egy szöveges fájlban, például a 2.5. kódrészletnek megfelelően. Ebben az esetben három alakzatot szeretnénk kirajzolni: egy zöld téglalapot, egy piros kört és egy kék háromszöget.

```
3
rectangle   10 10      200 250              0 255 0
circle      200 500    80                   255 0 0
triangle    150 150    400 120    430 500   0 0 255
```

2.5. kódrészlet. Rajzolóprogram bemenete

Amennyiben az ablakunk elég nagy ahhoz, hogy ezek elférjenek benne, akkor azt szeretnénk, ha ezen bemenet esetén a program futtatásakor a 2.10. ábrán látható kimenet jelenne meg az ablakban.



2.10. ábra. A kirajzolt alakzatok

A megvalósítás egyik módja, hogy a `render` függvény meghívásakor végigjárjuk az ablak pixeleit, majd ha egy pixel valamelyik síkidom belső tartományában helyezkedik el, akkor annak a színére színezzük ki (illetve ha több olyan is van, akkor ezek közül az utolsónak a színére, így elérjük, hogy a később kirajzolt alakzatok legyenek felül az eredményben).

Ehhez a diákoknak szüksége lesz a matematikából tanult analitikus mértan ismeretek egy részére. Téglalap esetén könnyű a döntés, mert azok a pontok vannak a belső tartományban,

melyeknek a sor- és oszlopkoordinátája is a bal felső illetve jobb alsó sarok sor- és oszlopkoordinátái által meghatározott intervallumokban található. Kör esetén azon pontok vannak a belső tartományban, melyek távolsága a középponttól nem nagyobb, mint a sugár. Háromszög esetén pedig felírhatjuk az oldalegyenesek egyenleteit, majd megnézzük hogy a kérdéses pont mindhárom oldalegyenesnek ugyanazon az oldalán található-e, mint az egyeneseken nem szereplő csúcs (teszteléskor sajátos esetként érdemes figyelmet fordítani a függőleges és vízszintes oldalegyenesekre).

Az ötlet egy lehetséges implementációja a melléklet *src/examples/textdraw-1.cpp* állományában található.

Transzláció

Az eddigiekben a képernyőn és a rajzon ugyanazt a koordináta-rendszert használtuk, tehát a képernyő r-edik sorának c-edik pixelje egyben a rajzolás síkjának (r, c) koordinátájú pontja, a bal felső saroktól lefelé és jobbra számolva. Ha azonban az ablakméret nem elég nagy, akkor előfordulhat, hogy a rajznak csak egy része látszik (a negatív tartományok pedig egyáltalán nem).

Erre azt a megoldást javasoljuk, hogy a rajz váljon mozgathatóvá, vagyis ha bal egérgombot lenyomva tartva húzzuk az egeret, akkor a mutató elmozdulásának megfelelően mozogjon a rajz is. Ezt implementálhatjuk úgy, hogy az `on_move` függvény segítségével folyamatosan nyilvántartjuk az egér kurzor legutóbbi helyét és amennyiben a bal egérgomb le van nyomva, akkor a legutóbbi és aktuális hely közötti elmozdulásnak megfelelően eltoljuk a képet, tehát a rajzolás és a képernyőfelület koordináta-rendszere el lesz tolva egymáshoz képest.

Az aktuális eltolás nyilvántartására elég két változót használni. Legyen tehát tr_{row} és tr_{col} a rajz azon sor és oszlopkoordinátája, melyet a képernyő $(0, 0)$ pozíciójára fogunk rajzolni. Ez azt jelenti, hogy amikor a képernyő (i, j) koordinátájú pixelének akarjuk a színét megállapítani, akkor a rajzolási síkban az $(i + tr_{row},\ j + tr_{col})$ koordinátájú pontot vizsgáljuk (azaz ellenőrizzük, hogy benne van-e a síkidomok valamelyikében), így tetszőlegesen eltolt állapotot tudunk megjeleníteni a `render` függvényben.

Amennyiben az egérrel mozgatjuk a képet és azt vesszük észre, hogy az egérmutató az

(r_1, c_1) pozícióról az (r_2, c_2) pozícióra mozdult el, akkor tr_{row} és tr_{col} értéke a

$$tr_{row} \leftarrow tr_{row} - (r_2 - r_1), \text{ illetve}$$
$$tr_{col} \leftarrow tr_{col} - (c_2 - c_1)$$

szabályoknak megfelelően kell változzon. Például ha az egér lefelé és jobbra mozdul el, akkor a tr_{row} és tr_{col} koordinátáknak egyaránt csökkenniük kell, mert a rajznak már a kisebb sor és oszlop-koordinátájú részeit is látni szeretnénk. Ugyanakkor a rajz elmozdulásának a mértéke így egyenlő az egér elmozdulásának mértékével, ezért a felhasználónak úgy tűnik majd, hogy az a pont, amit „megfogott” az egérrel, helyben marad az egérmutató alatt.

Ezen ötlet implementációja nem kellene nehézséget okozzon, egy lehetséges változata a melléklet *src/examples/textdraw-2.cpp* állományában megtalálható. Sajnos a mozgatás nem túl hatékony, mert folyton újra kell számolni az összes képpontot (és egyes rendszereken érezhetően akadozva követi a rajz mozgása az egérmutatót).

Javasoljuk a következő gyorsítási ötletet: minden alakzat esetén mentsünk le egy legkisebb és legnagyobb sor és oszlopindexet, melyeken kívül eső pontok biztosan nem részei az alakzat belső tartományának (például háromszög esetén a csúcsok sor-koordinátájának minimumánál kisebb sor-koordinátájú pont biztosan nem része a háromszögnek stb.). Ezen információkat előre kiszámolva minden alakzat esetén kapunk egy gyors elutasítási feltételt a képernyő pontjainak nagy részére (tehát a bonyolultabb és itt-ott lebegőpontos számokkal végzett műveleteket igénylő mértani számításokat nem kell majd minden pontra végrehajtani). Az ötlet egy implementációja a melléklet *src/examples/textdraw-3.cpp* forrásfájljában található és érezhetően javít a rajzolás hatékonyságán (vagyis az előző implementációnál lényegesen gyorsabban követi itt a rajz az egérmutatót).

Egyéb optimalizációs ötletek is felmerülhetnek, például nem kellene feltétlenül újraszámolni a képpontok színét azokban a zónákban, melyek eltolás előtt és után is részei a megjelenített képnek, ezek eltolt megjelenítéséhez fel lehetne használni az előző képet (amennyiben előzőleg elmentettük azt).

További feladatok

1. Implementáljuk az egyenes szakasz (adottak a végpontok, vastagság és szín), a paralelogramma (adott három szomszédos csúcsa, vagy egy csúcspontot és két vektor), illetve

a konvex sokszög (adottak a csúcspontok) primitíveket.

2. A grafikus primitívek esetén adjunk lehetőséget adott vastagságú és adott színnel színezett keret létrehozására (például fekete körvonalon belül pirosra színezett körlap).

3. A bemenet formátumát tegyük könnyebben olvashatóvá. Például a `rectangle 10 10 200 250 0 255 0` bemenet helyett fogadjuk el a bemeneten a `rectangle (10,10) (200,250) rgb(0,255,0)` szöveget és dolgozzuk fel karaktersorként a C++-ban rendelkezésre álló eszközökkel. A fehér karakterek (szóköz, tabulátor) számának ne legyen jelentősége. A szín megadásánál fogadjunk el hexadecimális konstanst is a CSS-ben megszokott formátumnak megfelelően, például `rgb(0,255,0)` helyett `#00FF00` alakban.

4. Készítsünk egy `circle2` nevű alakzattípust, mely olyan körlapot rajzol, amelynek körvonala áthalad három megadott nem kollineáris ponton.

5. A 2.10. ábrán látható lépcsősen recézett éleket szeretnénk elkerülni. Ezt a jelenséget *aliasing*-nek, az elkerülésére használt módszereket pedig *anti-aliasing*-nek nevezi a szakirodalom (lásd például [Akenine-Möller et al., 2018], 5.4. rész). Implementáljunk egyszerű felbontásduplázásra épülő supersampling típusú anti-aliasing algoritmust: duplázzuk meg az ismert pontok sor és oszlopindexeit és képzeljük el, hogy a képernyő szélessége és magassága is kétszer akkora, mint amit megkaptunk. Ekkor tulajdonképpen kétszer akkora felbontásban tudjuk kirajzolni a képet (sorok és oszlopok szempontjából egyaránt). Ezen a nagy képen négy-négy pixel színét átlagoljuk (2×2-es négyzeteket) és megkapunk egy, az eredeti felbontásnak megfelelő képet, de már kevésbé éles színátmenetekkel. Egy lehetséges (memória szempontjából hatékony) implementációt tartalmaz a melléklet *src/examples/textdraw-4.cpp* állománya, a kimenet változása pedig a 2.11. ábrán látható.

(a) anti-aliasing nélkül (b) kétszeres felbontással

2.11. ábra. A fentebbi 2.10. ábra kék háromszögének jobb alsó sarka (nagyítva)

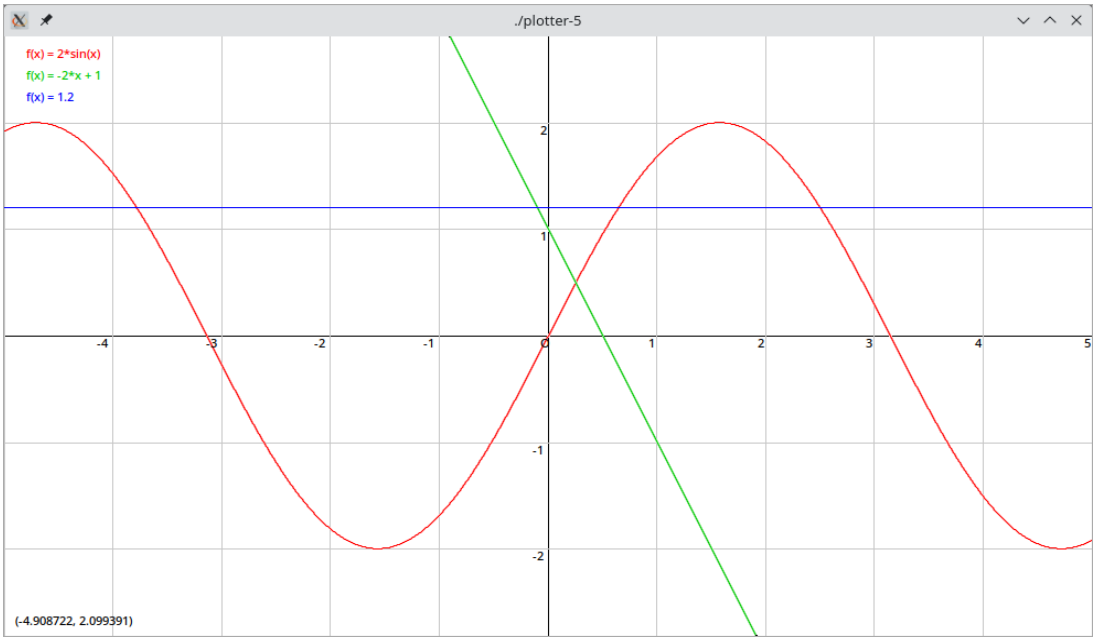
3. fejezet

Függvényábrázoló program készítése

Ebben a fejezetben egy függvényábrázoló program elkészítését tűzzük ki célul. A bemenet az ábrázolni kívánt egyváltozós valós számfüggvények megfeleltetési szabályait megadó szöveges állomány (egy példa a 3.1. kódrészletben). A kimeneten pedig egy, a 3.1. ábrán láthatóhoz hasonló képet szeretnénk előállítani, melyet később egérrel mozgathatóvá (és akár nagyíthatóvá) is teszünk majd.

```
1 2*sin(x)
2 -2*x + 1
3 1.2
```

3.1. kódrészlet. Függvényábrázoló program bemenete



3.1. ábra. A függvényábrázoló program egy kimenete

Mivel a funkcionalitások száma és bonyolultsága itt jelentősen nagyobb annál, mint amivel az eddigi feladatok során találkoztunk, ezért egyrészt a célközönség már inkább csak az érdeklődő, átlagnál jobb programozói készséggel rendelkező diákokra szűkül, másrészt pedig a függvényábrázoló program implementációját leginkább projekt jelleggel ajánljuk (az osztályban feladott kötelező megoldandó feladat helyett).

Első lépésben figyeljük meg, hogy a bemeneten megadott függvényeket számos pontban ki akarjuk majd értékelni. Tehát szükség lesz egy programrészre, mely a megadott karaktersorok alapján (illetve azokat tetszőleges adatszerkezetté alakítva) képes lesz a felhasználó által beírt függvények tetszőleges argumentumhoz tartozó függvényértékét meghatározni.

Amint ki tudjuk értékelni a függvényeket el kell majd dönteni, hogy milyen pixeleket kell a képernyőn beszínezni ahhoz, hogy a matematika órákon történő függvényábrázoláskor megszokott konvencióknak megfelelő képet állítsunk elő (például szeretnénk, hogy az összes függvényt ugyanabban a koordináta-rendszerben legyen ábrázolva, láthatóak legyenek a koordinátatengelyek, az egységnek ugyanakkora hossz feleljen meg ezeken).

Végül annak érdekében, hogy a program minél hasznosabb legyen a felhasználói számára, szeretnénk elérni, hogy egérműveletekkel tetszőlegesen mozgatható és nagyítható legyen a grafikus kép, vagyis a felhasználó ennek bármely részét meg tudja vizsgálni akár nagy pontossággal is (megfelelően ránagyítva arra részre, amiben érdekelt).

3.1. Megfeleltetési szabályok kiértékelése

A szöveges adatok értelmezése, beleértve akár az aritmetikai kifejezések kiértékelését is nem idegen a programozói versenyeken részt vevő diákok számára. Szükség van rá például az alábbi feladatok megoldásához (a két *infoarena.ro*-n megtalálható feladat leírása román nyelvű, a többi angol):

- https://codeforces.com/problemset/problem/188/H,
- https://infoarena.ro/problema/dir,
- https://infoarena.ro/problema/evaluare,
- https://codeforces.com/problemset/problem/778/B,
- https://codeforces.com/problemset/problem/7/E

Esetünkben egy-egy függvény megfeleltetési szabálya olyan kifejezés lesz, melyben a számok, a négy alapművelet és a csoportosítást szolgáló zárójelek mellett még megjelenhet az x változószimbólum, mely a függvény argumentumát jelenti. Szeretnénk továbbá néhány ismert függvényt is elérhetővé tenni a felhasználóknak (pl. $\sin(x)$, $\sqrt{x}$, stb.).

Első lépésben tehát az lenne a cél, hogy készítsünk egy programot, amely ilyen kifejezéseket képes kiértékelni. Például minden beolvasott kifejezés esetén írjuk ki a képernyőre az általuk megadott függvény -2, -1, 0, 1, 2 valós argumentumokhoz rendelt függvényértékét (már amennyiben értelmezett ezekben az argumentumértékekben, ellenkező esetben pedig például a `NAN` értéket).

A nyelvi elemzés (parsing) megvalósítására a szakirodalomban több megközelítés megjelenik (lásd például [Aho et al., 2007] 4. fejezetét). Jelenlegi feladatunkhoz egy rekurzív leszállásra épülő elemző megírását javasoljuk, többnyire azért, mert implementálható indirekt rekurzió segítségével néhány függvény között, és az implementáció megértéséhez nincs szükség a formális nyelvek elméletéhez kapcsolódó háttérismeretekre.

Amennyiben azt szeretnénk, hogy a diákok maguk álljanak elő a teljes implementációval, ahhoz jó eszköz akár a fentebb említett versenyfeladatok végigoldása, akár egy specifikusan rekurzív leszállással foglalkozó anyag tanulmányozása (pl. [Clarke, 1986] vagy [Stroustrup, 2013] 10.2. rész, illetve más online elérhető anyagok és oktatóvideók). Azt is megtehetjük, hogy odaadunk a diákoknak egy részleges implementációt (például olyat, amiben csak az összeadás, szorzás és részkifejezések zárójelezése támogatott), majd annak bővítése lesz a feladatuk.

A mellékletben megtalálható *src/examples/plotter-0.cpp* állomány egy lehetséges implementációt add olyan elemzőre, mely a négy alapműveletet, a zárójelezést és a szinusz függvényt ismeri. A rekurzív leszállás módszerével egy kifejezésfát építünk, majd ezt bejárva határozzuk meg a függvényértéket az egyes argumentumok esetén. Néhány alap szintű hiba is kezelésre kerül.

Az alábbi nyelvtannak megfelelően minden nemterminális szimbólumot egy-egy függvény kezel (a szóközök nem kötelezőek):

$$
\begin{aligned}
expr: &\quad term\ [+|-]\ term\ [+|-]\ ...\ [+|-]\ term \\
term: &\quad token\ [*|/]\ token\ [*|/]\ ...\ [*|/]\ token \\
token: & \\
&\quad \text{egy valós szám} \\
&\quad x \\
&\quad (expr) \\
&\quad \sin(expr)
\end{aligned}
$$

Ennek megfelelően az elemzést a `parse_expr`, `parse_term` és `parse_token` függvények végzik, menet közben pedig egy kifejezésfát építünk (mindegyik függvény visszatéríti a részkifejezésének megfelelő részfa gyökércsomópontjának címét, vagy egy hiba típusú csomópont címét megfelelő üzenettel). Az `evaluate_expression` függvény pedig az elkészült fát bejárva egy-egy valós x argumentumra ki tudja értékelni a kifejezést.

Amennyiben a gyökeres fák, illetve azok bejárásának fogalma még nem ismert, elkerülhetjük ezeket a fordított lengyel forma bevezetésével és a shunting yard algoritmus használatával (ld. [Norvell, 1999]).

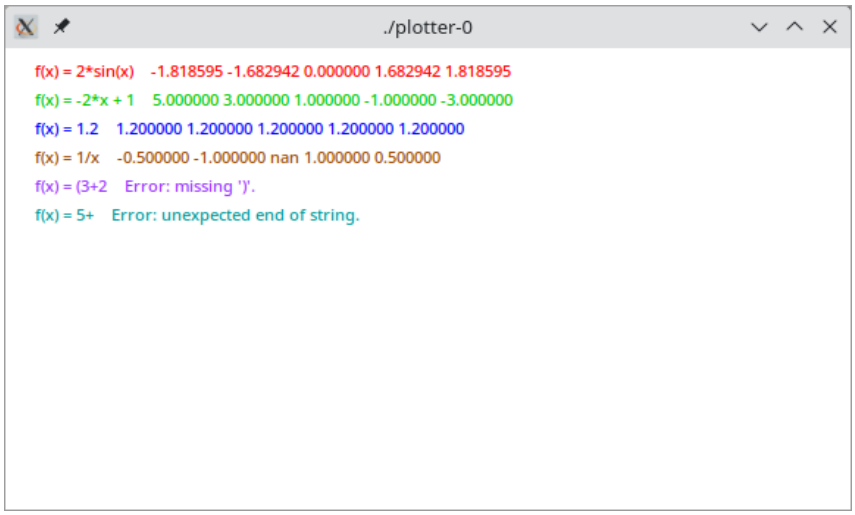
A 3.2. ábrán látható az elemző program egy kimenetének részlete (kiírtuk a beolvasott kifejezéseket, majd azok értékét az $x \in \{-2, -1, 0, 1, 2\}$ pontokban). A kiíráshoz, választottunk néhány különböző színt, hogy később a megfelelő függvénygörbék is ilyen színűek legyenek (ha túl sok függvény van megadva, akkor egy idő után újra felhasználjuk a színeket).

Az elemző és a kiértékelés funkcionalitását számos irányba bővíthetjük. Javasoljuk a következő feladatokat:

1. A sin függvény mellett implementáljuk még a cos, tg, ctg, arcsin, arccos, arctg, arcctg, ln, sqrt, exp függvényeket. Kiértékeléskor az értelmezési tartományon kívül eső argumentumokra térítsünk vissza `NAN` értéket.

2. Adjunk hozzá hatványozást (például az `x^3` kifejezés jelentse x-nek a harmadik hatványát). Figyeljünk oda a műveletek sorrendjére (a hatványozás bevezetése tulajdonképpen újabb függvény bevezetését igényli, hasonlóan, mint az összeadás és szorzás viszonyának esetében).

3. A kerek zárójelpár mellett (mely a csoportosítást jelenti), vezessük be a szögletes és kap-



3.2. ábra. Az elemző kimenete

csos zárójelpárokat, melyek az egészrészt és törtrészt számítják majd ki, illetve a függőleges vonalakat, melyek az abszolút értéket. Például $[x]$ az x egészrésze, $\{2x\}$ a $2x$ törtrésze és $|\sin(x)|$ a $sin(x)$ abszolút értéke.

3.2. Koordináta-rendszer és függvénygörbék

A függvények megfeleltetési szabályainak értelmezése után szeretnénk kirajzolni a koordináta-tengelyeket úgy, hogy azok nagyjából az ablak vízszintes és függőleges szimmetriatengelyei legyenek (azért csak nagyjából, mert páros szélesség vagy magasság esetén nem tudunk egy pontosan középső sor és oszlopindexet meghatározni). A `render` függvénybe beérkező *width* és *height* paramétereken keresztül megkapjuk az aktuális ablak szélességét és magasságát, ennek megfelelően legyen az Ox tengely a $\left\lfloor \frac{height}{2} \right\rfloor$ indexű soron, az Oy tengely pedig a $\left\lfloor \frac{width}{2} \right\rfloor$ indexű oszlopon (ezen sort és oszlopot feketére festjük). Továbbá szeretnénk még egy szürke négyzetrácsot, ahol a rácspontok az egész koordinátájú pontok. Ehhez meg kell tudjuk mondani, hogy milyen oszlopindexekre kerülnek az egész x koordináták, és milyen sorindexekre kerülnek az egész y koordináták.

Építsünk fel tehát egy koordináta-transzformációt a függvényábrázolás koordináta-rendszere illetve a pixelek kirajzolásának koordináta-rendszere között. A függvényábrázolás koordináta-rendszerében az origó az ablak közepe (pontosabban a $\left\lfloor \frac{height}{2} \right\rfloor$ sorindexű és $\left\lfloor \frac{width}{2} \right\rfloor$ oszlopindexű pixel), az Ox tengely vízszintes és jobb oldalra haladva nő, az Oy tengely pedig

függőleges és felfelé haladva nő. Az pixelek megjelenítésének koordináta-rendszerében pedig az origó a bal felső sarok, innen lefelé nőnek a sorindexek (amik ráadásul egész számok), jobbra haladva pedig az oszlopindexek.

Szabadon megválaszthatjuk, hogy egy pixelnyi távolság mekkora távolságnak felel meg a valós számsíkon. Tegyük fel például, hogy ezt úgy akarjuk megválasztani, hogy az ablak teljes szélessége x_range egységnyi távolságnak feleljen meg az Ox tengelyen (x_range-et megválaszthatjuk mondjuk 10-nek, de az általánosság kedvéért szimbolikus alakban használjuk). Ekkor $width$ pixelnyi távolság megfelel x_range darab egységnek a vízszintes tengelyen, ezért egy pixelnyi távolság $\frac{x\_range}{width}$ lesz és fordítva, egy egységnek $\frac{width}{x\_range}$ darab pixel felel meg.

Ekkor adott c oszlopindex esetén az x koordináta az

$$x = \left(c - \left\lfloor \frac{width}{2} \right\rfloor\right) \cdot \frac{x\_range}{width}$$

összefüggéssel, illetve adott x koordináta esetén a neki megfelelő c oszlopindex a

$$c = \left\lfloor \frac{width}{2} \right\rfloor + \left\lfloor x \cdot \frac{width}{x\_range} \right\rfloor$$

összefüggéssel adható meg. Ez utóbbi persze csak egy közelítő érték, mert kénytelenek vagyunk egész számot kihozni oszlopindexnek, viszont az ebből származó hibát elhanyagolhatónak tekintjük. Sőt, kerekítés helyett alsó egész részt használunk az implementáció egyszerűsége miatt (ekkor ugyanis nem kell `round` függvényt hívni, elég ha egy egész típusú változóba mentjük a kiszámolt lebegőpontos értéket).

Hasonlóan megállapíthatjuk az y koordináta és az r sorindex közötti összefüggéseket, csak arra kell vigyázni, hogy az y koordináta felfelé haladva nő, a sorindex pedig lefelé. Ennek megfelelően

$$y = \left(\left\lfloor \frac{height}{2} \right\rfloor - r\right) \cdot \frac{x\_range}{width},$$

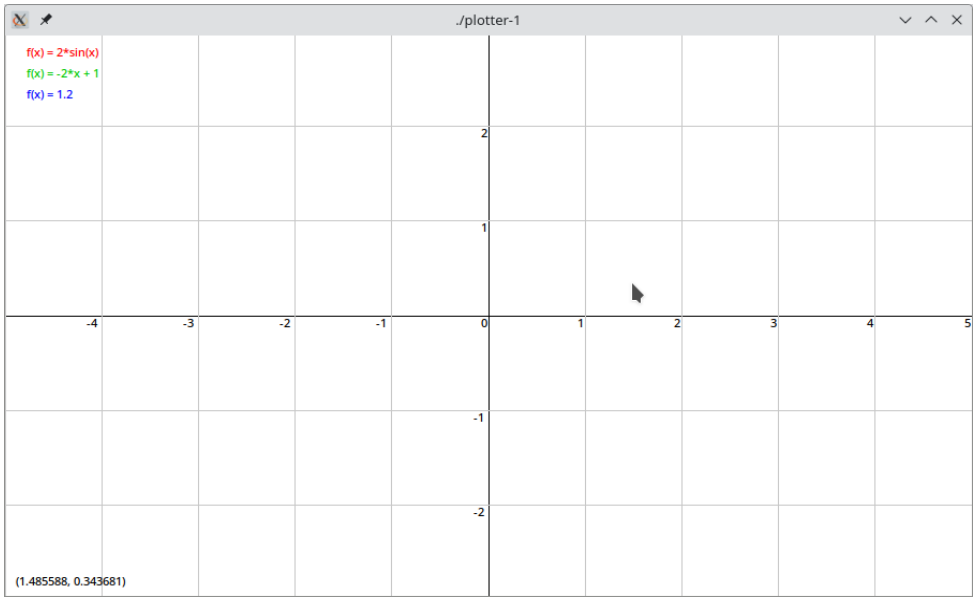
illetve

$$r = \left\lfloor \frac{height}{2} \right\rfloor - \left\lfloor y \cdot \frac{width}{x\_range} \right\rfloor.$$

Figyeljük meg, hogy ugyancsak az $\frac{x\_range}{width}$ és $\frac{width}{x\_range}$ váltószámokat használtuk mert azt szeretnénk, ha az egységnyi távolság ugyanakkora lenne mindkét tengelyen (tehát tulajdonképen megszabtuk, hogy a képernyő szélessége x_range távolságot fedjen le, a magassága pedig a

képaránynak megfelelően ennél nagyobb vagy kisebb távolságot fed le y irányban).

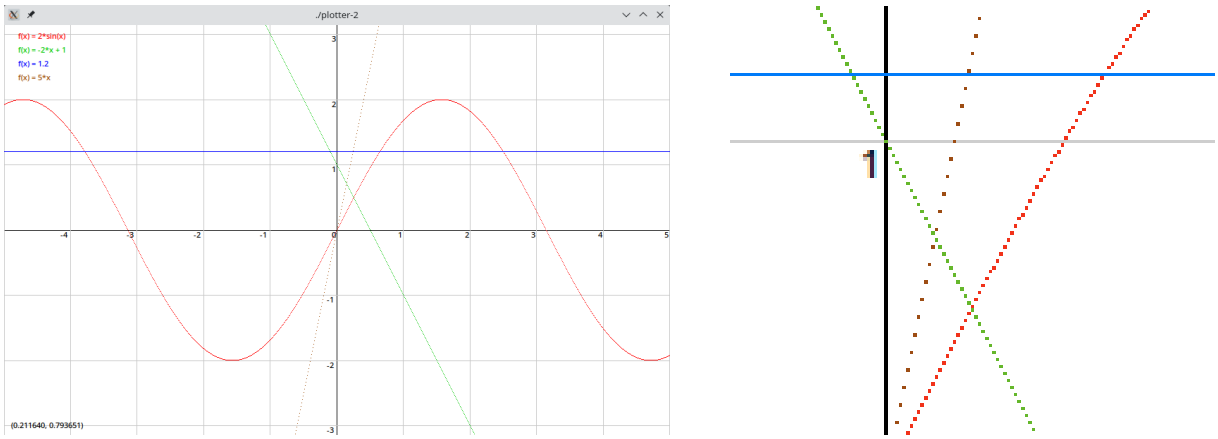
Ezzel megvan az eszköztárunk ahhoz, hogy a négyzetrácsot is kirajzoljuk. A koordinátatranszformációk további ellenőrzése céljából még jelenítsük meg azokat a koordinátákat is, melyek az egér aktuális pozíciójának felelnek meg a függvényábrázolás koordinátarendszerében, a 3.3. ábrához hasonló kimenetet előállítva (természetesen az egér pozícióját aktualizálni kell minden mozgási esemény során). Egy lehetséges implementáció a melléklet *src/examples/plotter-1.cpp* állományában található. Az említett elemek mellett még kirajzoltuk a négyzetrács egyeneseinek megfelelő koordinátákat, illetve meghagytuk az előző részben tárgyalt elemzőt és a függvénykifejezések kiírását (ezek ugyanis részei maradnak a végterméknek).



3.3. ábra. Koordináta-tengelyek és egér pozíciója

Ha ezzel megvagyunk, elkezdhetjük magát a függvényábrázolást. Egy viszonylag természetesnek tűnő ötlet a következő: az f függvény ábrázolásához számoljuk ki rendre az egyes oszlopindexeknek megfelelő x koordinátát, a kifejezéskiértékelő függvénnyel számoljuk ki az ennek megfelelő $f(x)$ függvényértéket, majd keressük meg, hogy mely sorindex felel meg ennek (már amennyiben az a látható részben található) és fessük be a megfelelő pixelt a függvény színére. Ezen ötlet implementációját tartalmazza a melléklet *src/examples/plotter-2.cpp* állománya, az eredmény viszont korántsem kielégítő (ld. 3.4. ábra).

A probléma hasonló, mint a ceruza eszközt implementáló feladat esetében az előző fejezet-



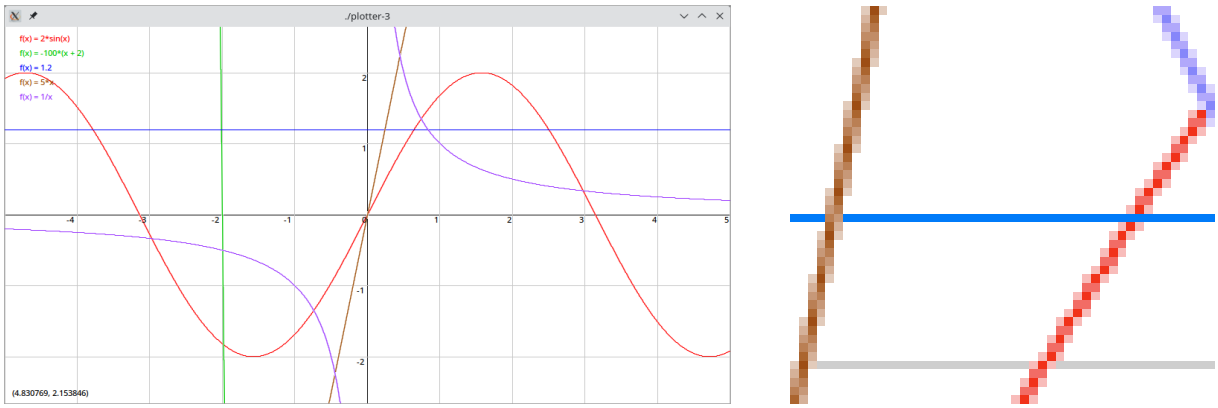
(a) teljes kimenet (b) egy felnagyított rész

3.4. ábra. Első próbálkozás a függvények ábrázolására

ben. Bár egymás melletti oszlopoknak megfelelő x koordinátákban értékeljük ki a függvényeket, előfordulhat, hogy a függvényértékek különbsége akkora, hogy nem egymásutáni sorindexeket kapunk, hanem hézagok maradnak. Egy lehetséges megoldás az előző fejezetben tárgyalt digital differential analyzer algoritmus implementálása, viszont javasolunk itt egy másik módszert, mellyel valamivel simábbnak tűnő kimenetet tudunk előállítani (anélkül, hogy bonyolult anti-aliasing algoritmusok implementálásában gondolkodnánk).

Tegyük fel, hogy a j indexű oszlop i indexű sorába kell egy pixelt kirajzoljunk, a $j+1$ indexű oszlopban pedig már az $i+k$ indexű sorba esik a befestendő képpont (azok az esetek zavarnak, amelyekben $k \geq 2$). Eljárhatunk úgy, hogy a j és $j+1$ indexű oszlopokon egyaránt befestjük az $i, i+1, ..., i+k$ sorokon levő pixelek mindegyikét, de a j indexű oszlopon egyre csökkenő, a $j+1$ indexű oszlopon pedig egyre növekvő intenzitással. Tehát a függvényértéknek megfelelő képpont lesz a legerősebb színű mindkét oszlop esetén, de elérjük azt is, hogy a sorindexek közötti átmenet fokozatosan történjen meg (ne legyenek ugrások). Persze ez csak a valóban folytonos függvények ábrázolásakor jó stratégia, de ettől most eltekintünk (vagyis feltételezzük, hogy folytonos függvényekkel dolgozunk, csak az értelmezési tartományon kívül eső argumentumok esetét vizsgáljuk). Az ötlet implementációját a melléklet *src/examples/plotter-3.cpp* állománya tartalmazza, az eredményt pedig a 3.5. ábra szemlélteti.

További feladat lehetne, hogy a négyzetrács beosztásait ne mindig egységenként vegyük fel, hanem igazodjunk az ablak méretéhez, hogy a szürke vonalak ne legyenek se túl sűrűn, se túl ritkán. A két rácsvonal közötti távolságot beállíthatjuk például kettőnek a megfelelő hatványára



(a) teljes kimenet (b) egy felnagyított rész

3.5. ábra. Folytonos függvénygörbék előállítása

(tehát egy, kettő, négy stb. egységre, vagy másik irányban fél, negyed stb. egységre annak megfelelően, hogy melyik esetén van nem túl nagy és nem túl kicsi távolság két rácsvonal között). Ez különösen hasznos lesz majd a nagyítás implementálása után.

3.3. Geometriai transzformációk

Hasznosabbá tenné a függvényábrázoló programot, ha a függvények grafikus képének nem mindig ugyanazt a tartományát mutatnánk. Ebben a részben transzlációt és nagyítást implementálunk, hogy a felhasználó a megfelelő egérműveletekkel el tudja mozgatni a grafikus képet, illetve rá tudjon nagyítani annak egy-egy részére.

Az eddigiekben a pixelek kirajzolásának koordináta-rendszere mellett használtunk egy xOy ortonormált koordináta-rendszert, melyet a függvényábrázolás koordináta-rendszerének neveztünk és amelynek középpontja az ablak közepe, illetve benne az egységhossz akkora, hogy az ablak teljes szélessége x_range egység hosszú legyen.

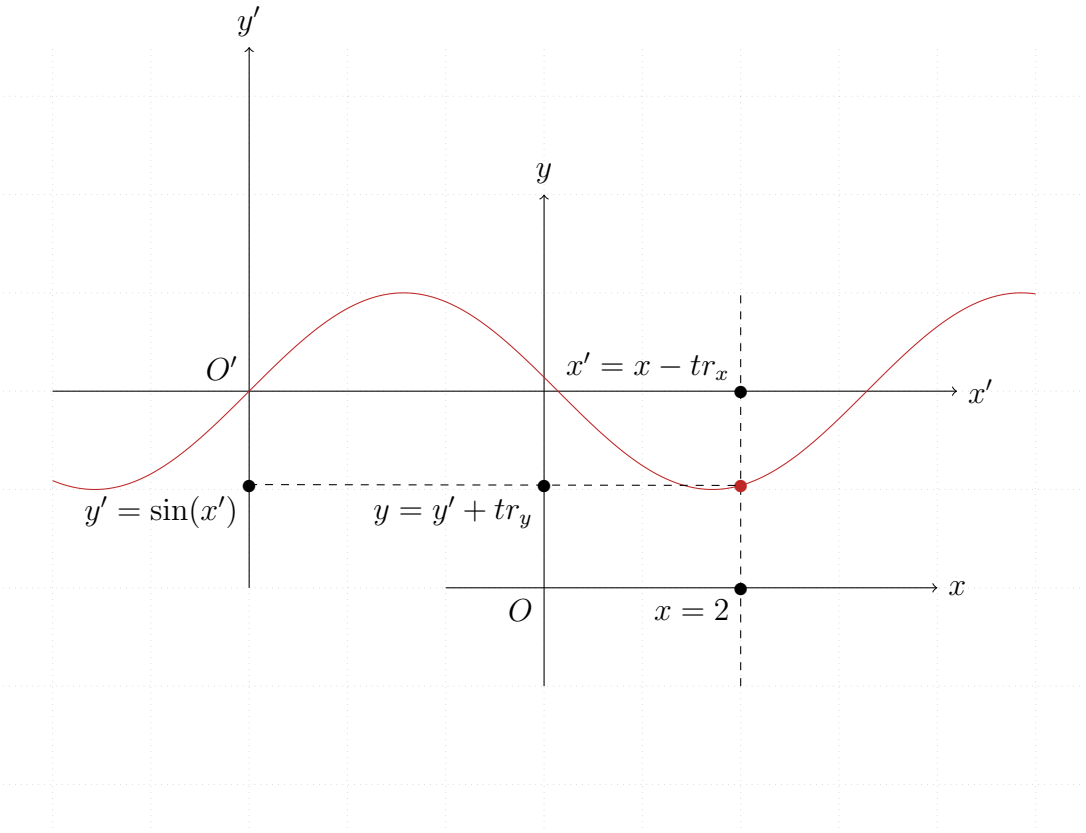
Ezúttal a függvények ábrázolását áttesszük egy másik, $x'O'y'$ koordináta-rendszerbe, mely az alkalmazott transzformációknak megfelelően viszonyul majd az xOy koordináta-rendszerhez (hasonló megoldást már láttunk az előző fejezetben, de ott a pixelek kirajzolásának koordináta-rendszeréhez képest volt egy eltolásunk).

Transzláció

Kezeljük először az eltolást. Jelölte tr_x és tr_y az $x'O'y'$ koordináta-rendszer O' kezdőpontjának koordinátáit az xOy koordináta-rendszerben. Ekkor érvényesek az

$$\begin{cases} x' = x - tr_x \\ y' = y - tr_y \end{cases}, \text{ illetve } \begin{cases} x = x' + tr_x \\ y = y' + tr_y \end{cases}$$

összefüggések. Ez azt jelenti, hogy ha a grafikus ablak c oszlopindexe az x koordinátának felel meg az xOy koordináta-rendszerben, akkor a függvényt nem az x argumentumban értékeljük ki, hanem az annak megfelelő x' argumentumban. Továbbá az $y' = f(x')$ függvényértékhez társított y koordinátának megfelelően tudjuk majd a színezni kívánt pixel sorindexét meghatározni (ld. 3.6. ábra). A színezni kívánt pont (x, y) koordinátáiból az oszlop- és sorindexre való átszámolást már az előbbiekben elvégeztük.



3.6. ábra. Az $f(x) = \sin(x)$ függvény grafikus képe eltolva ($tr_x = -3,\ tr_y = 2$) és a két koordináta-rendszer

Természetesen a grafikus ablakban csak az $x'O'y'$ koordináta-rendszert és az ennek megfelelő négyzetrácsot mutatjuk majd a felhasználónak, hiszen ennek van információértéke a függvényábrázolás szempontjából.

Az implementációhoz még hozzá tartozik, hogy az egérmozgásnak megfelelően változtassuk tr_x és tr_y értékét (feltéve, hogy a bal egérgomb le van nyomva, azaz a felhasználó mozgatni akarja az ábrát). Ha például az egér a mozgás során d_r sorral került lejjebb és d_c oszloppal jobbra az előző pozíciójához képest, akkor a

$$tr_x \leftarrow tr_x + d_c \cdot \frac{x\_range}{width},$$

illetve

$$tr_y \leftarrow tr_y - d_r \cdot \frac{x\_range}{width}$$

átalakításokat kell elvégezzük (tehát az egér elmozdulását átalakítjuk az xOy koordináta-rendszerben megfelelő elmozdulássá, ezzel azt az érzést keltjük, hogy az a pont, amin állva a felhasználó lenyomta az egérgombot, végig az egérmutató alatt marad, együtt mozog vele). Amennyiben az egér felfelé és/vagy balra mozgott, akkor a d_r és d_c értékek közül valamelyik vagy mindkettő negatív lesz.

A koordinátatranszformációkat egybevonva felírhatunk megfeleltetéseket egyenesen az $x'Oy'$ és a pixelek rajzolásának koordináta-rendszere között (ahol r-rel jelöljük a sorindexet, c-vel az oszlopindexet). Egyrészt már tudjuk, hogy

$$x = \left(c - \left\lfloor \frac{width}{2} \right\rfloor\right) \cdot \frac{x\_range}{width}, \qquad c = \left\lfloor \frac{width}{2} \right\rfloor + \left\lfloor x \cdot \frac{width}{x\_range} \right\rfloor,$$
$$y = \left(\left\lfloor \frac{height}{2} \right\rfloor - r\right) \cdot \frac{x\_range}{width}, \qquad r = \left\lfloor \frac{height}{2} \right\rfloor - \left\lfloor y \cdot \frac{width}{x\_range} \right\rfloor.$$

Amennyiben ezekben áttérünk az x' és y' koordinátákra (a tr_x és tr_y eltolásnak megfelelően), azt kapjuk, hogy:

$$x' = \left(c - \left\lfloor \frac{width}{2} \right\rfloor\right) \cdot \frac{x\_range}{width} - tr_x, \qquad c = \left\lfloor \frac{width}{2} \right\rfloor + \left\lfloor (x' + tr_x) \cdot \frac{width}{x\_range} \right\rfloor,$$
$$y' = \left(\left\lfloor \frac{height}{2} \right\rfloor - r\right) \cdot \frac{x\_range}{width} - tr_y, \qquad r = \left\lfloor \frac{height}{2} \right\rfloor - \left\lfloor (y' + tr_y) \cdot \frac{width}{x\_range} \right\rfloor.$$

Ezeknek megfelelően tulajdonképpen már nincs is szükség az xOy-beli koordinátákra az imp-

lementációban.

A transzlációval kiegészített implementáció megtalálható a melléklet *src/examples/plotter-4.cpp* állományában. Az említett elemek mellet persze még a négyzetrácson is alkalmazni kell a megfelelő eltolást, illetve az egér aktuális pozíciójának koordinátáit is az $x'O'y'$ koordinátarendszernek megfelelően kell kiírni.

Nagyítás

Szeretnénk olyan nagyítást implementálni, mely az egér aktuális pozícióját is figyelembe veszi (tehát az egér alatt található pont maradjon helyben, de az ábra nőjön meg minden alkalommal, amikor például görgetünk egyet felfelé, illetve legyen kisebb lefelé görgetéskor). Ez pont az a viselkedés, ami képszerkesztők vagy térképszoftverek esetében a felhasználóknak megszokott.

Azt szeretnénk tehát, hogy az x_range adott arányban nőjön vagy csökkenjen görgetéskor (például minden `on_scroll` függvényhívás esetében 5%-kal változzon a *delta* paraméter előjelének megfelelően), de az egér alatti képpont maradjon helyben. Viszont azt, hogy hova és mekkorára rajzoljuk a grafikus képeket és a négyzetrácsot, jelenleg három állapotparaméter jellemzi: x_range, tr_x és tr_y.

Tegyük fel, hogy ezen paraméterek jelenlegi értékei $x\_range^{(1)}$, $tr_x^{(1)}$ és $tr_y^{(1)}$, a nagyítás utáni értékek pedig $x\_range^{(2)}$, $tr_x^{(2)}$ és $tr_y^{(2)}$, melyek közül $x\_range^{(2)}$-ről már tudjuk, hogy milyen értéke kell legyen (az előbbi gondolatmenetnek megfelelően). A feladatunk tehát, hogy megtaláljuk azokat az értékeket $tr_x^{(2)}$-nek és $tr_y^{(2)}$-nek, amelyek mellett az éppen az egér alatt levő pont helyben marad a képernyőn.

Ha az egér sor- és oszlopindexe jelenleg m_r és m_c, akkor $x'O'y'$-ben a neki megfelelő pont koordinátái

$$m_{x'} = \left(m_c - \left\lfloor \frac{width}{2} \right\rfloor\right) \cdot \frac{x\_range^{(1)}}{width} - tr_x^{(1)}, \tag{3.1}$$

illetve

$$m_{y'} = \left(\left\lfloor \frac{height}{2} \right\rfloor - m_r\right) \cdot \frac{x\_range^{(1)}}{width} - tr_y^{(1)}. \tag{3.2}$$

Azt szeretnénk, ha nagyítás után is a sík $(m_{x'}, m_{y'})$ pontja az (m_c, m_r) pixelkoordinátákra ke-

rülne, vagyis teljesülne, hogy

$$m_{x'} = \left(m_c - \left\lfloor \frac{width}{2} \right\rfloor\right) \cdot \frac{x\_range^{(2)}}{width} - tr_x^{(2)}, \tag{3.3}$$

illetve

$$m_{y'} = \left(\left\lfloor \frac{height}{2} \right\rfloor - m_r\right) \cdot \frac{x\_range^{(2)}}{width} - tr_y^{(2)}. \tag{3.4}$$

.

Ekkor a 3.1 és 3.3 összefüggésekből

$$\left(m_c - \left\lfloor \frac{width}{2} \right\rfloor\right) \cdot \frac{x\_range^{(1)}}{width} - tr_x^{(1)} = \left(m_c - \left\lfloor \frac{width}{2} \right\rfloor\right) \cdot \frac{x\_range^{(2)}}{width} - tr_x^{(2)},$$

ahonnan

$$tr_x^{(2)} = tr_x^{(1)} + \left(m_c - \left\lfloor \frac{width}{2} \right\rfloor\right) \cdot \frac{x\_range^{(2)} - x\_range^{(1)}}{width}.$$

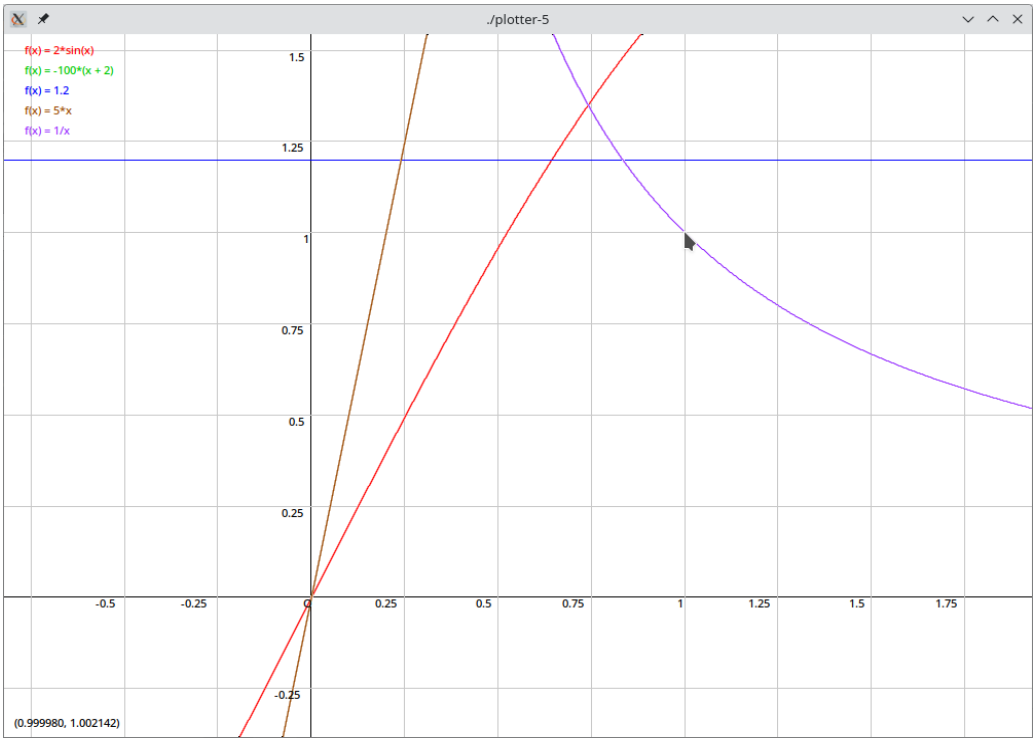
Hasonlóan a 3.2 és 3.4 összefüggésekből

$$tr_y^{(2)} = tr_y^{(1)} + \left(\left\lfloor \frac{height}{2} \right\rfloor - m_r\right) \cdot \frac{x\_range^{(2)} - x\_range^{(1)}}{width}.$$

Így abból a feltételből, hogy az egér alatti pont maradjon helyben a képernyőn meg tudtuk határozni a $tr_x^{(2)}$ és $tr_y^{(2)}$ értékeket, vagyis már minden paraméter ismert a nagyítás utáni kirajzoláshoz. Ezeknek kiszámítását az `on_scroll` függvényben célszerű elvégezni. Egy lehetséges implementáció a melléklet *src/examples/plotter-5.cpp* állományában található. A 3.7. ábrán látható a grafikus ablak tartalma miután az $(x, y) = (1, 1)$ pontra ránagyítottunk.

Bár az itt végigvitt gondolatmenet nehéznek bizonyulhat a diákok számára, érdemes engedni, hogy próbálkozzanak saját ötletekkel még akkor is, ha azok eleinte nem helyesek (például implementálhatják a nagyítást úgy, hogy csak az x_range értékét állítják be, az eltolási értékeket változatlanul hagyják, aztán teszteléskor majd kiderül, hogy mi ezzel a gond).

A függvényábrázoló program tovább bővíthető tetszőleges függvényekkel vagy akár billentyűeseményekkel is (pl. adott gomb megnyomására térjünk vissza az eredeti nézetre). Remélhetőleg sikerül elérni vele azt a célt, hogy a diákok az analitikus mértan ismereteiket valamilyen konkrét gyakorlatban megjelenő probléma megoldására alkalmazzák.



3.7. ábra. Nagyított kimenet

Irodalomjegyzék

Aho, A. V., Lam, M. S., Sethi, R., és Ullman, J. D. *Compilers. Principles, Techniques & Tools*. Pearson Education, 2nd edition, 2007.

Akenine-Möller, T., Haines, E., Hoffman, N., Pesce, A., Iwanicki, M., és Hillaire, S. *Real-Time Rendering*. CRC Press, 4th edition, 2018.

Bresenham, J. E. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4 (1):25–30, 1965.

Clarke, K. The top-down parsing of expressions. 1986. URL https://www.antlr.org/papers/Clarke-expr-parsing-1986.pdf.

Godse, A. P. és Godse, D. A. *Computer Graphics*. Technical Publications Pune, 2007.

Kutepov, A. Olive.c. URL https://github.com/tsoding/olive.c.

Norvell, T. Parsing expressions by recursive descent. 1999. URL https://www.engr.mun.ca/~theo/Misc/exp_parsing.htm.

Stroustrup, B. *The C++ Programming Language*. Pearson Education, 4th edition, 2013.

Thain, D. gfx: A simple graphics library (v2). URL https://www3.nd.edu/~dthain/courses/cse20211/fall2013/gfx/.