Jegyzőkönyv

Komputergrafika és képfeldolgozás

Útvonalkereső és képfeldolgozó

Készítette: **Bárdos Triszten Krisztofer**

Neptunkód: **CUNPO1**

Dátum: 2024.05.13

Sárospatak, 2024

Tartalomjegyzék

Bevezetés

A feladat leírása:

Az általam választott első feladat egy Java nyelvben megvalósított útvonalkereső algoritmus.

Az algoritmus neve „A*” (A star) algoritmus, mely egy heurisztikával ellátott legrövidebb útvonal kereső algoritmus.

Az applikáció felületén gombokat helyeztem el cellaszerű elrendezésben, melyek reprezentálják a mezőket, és a program figyel a felhasználó interakcióira.

A gombok megnyomásával azok állapota megváltoztatható, adott esetben egy gomb állapota lehet szabadon bejárható mező, akadály (vagy fal) mező, vagy a program futása után útvonal mező. Ezen mezők színtulajdonságai szintén meghatározottak, ezzel jelezve a felhasználó számára a különböző felvett állapotokat.

A feladathoz használt fejlesztői környezet: Apache NetBeans IDE 16

A második választott feladat egy képfeldolgozó program implementálása C++ nyelvben, majd ezen képekből kollázs létrehozása egy képszerkesztő programmal.

A program bemenetként kap egy bittérképes *.ppm* formátumú képet, és erre alkalmaz különböző filtereket és módosításokat.

Ezen képek mérete nagyobb az átlag képeknél, ugyanis itt álltalában nyers adatokat (*raw data*) tárol a számítógép az adott képről.

Fontos megjegyzés, hogy manapság ezeket a képeket ritkán használják, mert nem kompatibilisek a Windowsra írt szabványos szoftverekkel.

A feladathoz használt fejlesztői környezet: Code::Blocks 20.03.

A feladathoz használt képszerkesztő program: GIMP 2.10.34

1. Feladat – Útvonalkereső

A projekt neve: „pathfinder_A_star”, mely 4 Java osztályt tartalmaz:

- *Pathfinder_A_star.java*
- *Frame.java*
- *Grid.java*
- *Point.java*

package név: pathfinder_a_star

A *Pathfinder_A_star.java* importjai:

```
import java.awt.Color;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Random;
import java.util.Set;
import javax.swing.JFrame;
```

A *Frame.java* importjai:

```
import java.awt.GridLayout;
import javax.swing.JFrame;
```

A *Grid.java* nem rendelkezik külön importokkal.

A *Point.java* importjai:

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.SwingUtilities;
import static pathfinder_a_star.Pathfinder_A_star.end;
```

```
import static pathfinder_a_star.Pathfinder_A_star.start;
import static pathfinder_a_star.Pathfinder_A_star.diagonal_is_allowed;
import static pathfinder_a_star.Pathfinder_A_star.perform_with_heuristics;
import static pathfinder_a_star.Pathfinder_A_star.BUTTON_BORDER_COLOR;
```

Az eseménykezelések tehát a *Point.java* fájlban valósulnak meg.

Mindegyik osztály különböző funkciókat lát el, de létezik olyan változó vagy funkció, mely több osztályban is szerepet játszik a program működésében.

A *„main()”* függvény a *Pathfinder_A_star.java* fájlban található, és alapértelmezetten egy metódust hív meg: *performAstar(…)*

Mellette opcionálisan meghívható a *generateRandomObjects()* metódus, ami random generál akadály objektumokat.

A *main()* függvény:

```
public static void main(String[] args) throws InterruptedException
{
    //generateRandomObjects();

    performAstar(start, end, diagonal_is_allowed, perform_with_heuristics);
}
```

A *main()* függvény „dobhat” egy úgynevezett megszakítási kivételt (*interruptedException*), ami a szálkezelési folyamatokhoz szükséges.

a *performAstar()* metódus paraméterként várja a kezdőpontot (*start*), a végpontot (*end*), a diagonális vagy átlós mozgás engedélyezésének logikai értékét (*diagonal_is_allowed*), illetve a heurisztika engedélyezésének logikai értékét (*perform_with_heuristics*). Ha az utolsó paraméter hamis, akkor a program valójában *Dijkstra* algoritmussá válik.

A *Pathfinder_A_star.java* fájlban deklaráltam a cellák számát vertikális, illetve horizontális irányban.

```
public static final int NUMBER_OF_ROWS = 15;
public static final int NUMBER_OF_COLUMNS = 30;
```

a „*final*” kulcsszóval elérhető, hogy ne lehessen módosítani a változót a program további részeiben. Mindkét változó egész szám típusú, reprezentálva a pálya méretét.

Az azt követő sorokban a különböző gomb állapotok színe kerül megvalósításra, és mindegyik egy-egy változóhoz van hozzárendelve a *java.awt.Color* használatával.

Példa:

```
public static Color DEFAULT_TILE_COLOR = new Color(155, 199, 228);
```

Ez egy szabadon bejárható mező színe.

A színbeállítások alatt a rácsszerkezetű pálya inicializálása történik:

```
public static Grid grid = new Grid(NUMBER_OF_ROWS, NUMBER_OF_COLUMNS);
```

A *Grid.java* fájlban az osztály konstruktora két egész számot vár, és azok alapján hozza létre a pályát úgy, hogy két egymásba ágyazott ciklussal létrehoz egy-egy *Point* objektumot, melynek osztály konstruktora szintén két egész számot vár, mégpedig a sor és az oszlopindexet.

```
for ( int i = 0; i < this.number_of_rows; i ++ )
{
    for ( int j = 0; j < this.number_of_columns; j ++ )
    {
        this.points[i][j] = new Point(i,j);
    }
}
```

A *Point.java* fájlban mindegyik paraméterként kapott sor és oszlopindex párhoz létrehoz egy gombot („*button*” változónévvel) .

Ehhez a gombhoz minden iterációban rendel egy ActionListener interfészt és egy *MouseListener* interfészt.

Az első a gombokra való kattintás eseményeinek kezeléséhez, a másik az egér által megvalósított események kezeléséhez szükséges.

```
button.addActionListener(this);
```

```
button.addMouseListener(new MouseAdapter()
{
    @Override
    public void mouseEntered(MouseEvent e)
    {
        .
        .
        .
    }
});
```

```
@Override
public void actionPerformed(ActionEvent e)
{
    if ( e.getSource() == button )
    {
        .
        .
        .
    }
}
```

Az algoritmus menete

Az algoritmus úgy működik, hogy létrehoz egy olyan ponthalmazt, amelybe azok a cellák kerülnek be, melyeket már meglátogatott a program.

Kezdetben mindegyik pont értéke „végtelen”:

```
public double local_score = Double.POSITIVE_INFINITY;
public double global_score = Double.POSITIVE_INFINITY;
```

Ez azért szükséges, mert az algoritmus miután a kezdőponttól elkezdi felfedezni a szomszédos cellákat, akkor mindegyik ponthoz egy új távolság értéket fog hozzá rendelni oly módon, hogy ha az aktuális cella távolság értéke + a vizsgált szomszédcelláig való eljutás költsége (ami pontosan 1.0 egy ilyen rács szerű pályánál) kisebb, mint a vizsgált szomszédcella távolságértéke, akkor frissíti azt a kisebb távolságértékre, így biztosított, hogy az iterációk végén felfedezhető lesz az az útvonal, melynek a legkisebb ez az értéke.

Megjegyzés: erre példa programrészletet nem illesztek be annak méretére való tekintettel.

Heurisztika

A heurisztika az alapértelmezett euklideszi pont távolság alapján határozza meg, hogy milyen messze van a végponttól az aktuális pont.

Ezt a következő függvény határozza meg:

```
static double euclideanDistanceBetweenTwoPoints(int x1, int y1, int x2, int
y2) //Heurisztika
{
    return Math.sqrt(Math.pow(x1 - x2, 2) + Math.pow(y1 - y2, 2));
}
```

Visszatérési értékként egy lebegőpontos számot kapunk, így elkerülve a sor-oszlop differenciákból adódó potenciális pontatlanságot (Manhattan-distance), majd ezen értékek alapján a program sorba rendezi a cellákat, úgy, hogy amelyiknek ezen távolságértéke kisebb, az előrébb kerül a vizsgálandó cellák listájában.

Tehát az algoritmus feltételezi, hogy jó helyen jár, ha ez a távolság kisebb, de ez inkább azon esetekben érvényes, amikor a síkbeli elrendezés nagyrészt követi az alap geometriai szabályokat.

A heurisztika alkalmazása:

```
if ( perform_with_heuristics )
{
    for ( int i = 0; i < queue.size() - 1; i ++ )
    {
        for ( int j = i + 1; j < queue.size(); j ++ )
        {
            if ( queue.get(i).global_score > queue.get(j).global_score )
            {
                Point P = queue.get(i);
                queue.set(i, queue.get(j));
                queue.set(j, P);
            }
        }
    }

    //Az adatok rendezése: Collections.sort(queue);
```

```
    if ( visited_points_set.contains(grid.points[end.row_coordinate][end.column_coordinate]) )
    {
        break;
    }
}
```

Felhasználói irányítás

A program futtatása után megjelenő ablakban a gombokra jobb egérgomb lenyomásával akadály objektum helyezhető el, illetve vehető le.

Egér elhúzása esetén ugyanezen műveletek valósíthatók meg, viszont nem kell egyesével minden gombot megnyomni annak állapotának megváltoztatásához.

2. Feladat – Képfeldolgozó program

Projekt neve: *image*

A programkódot tartalmazó *.cpp* fájl neve: *image.cpp*

Az általam megvalósított program a *landscape.ppm* bittérképes képre alkalmaz különböző filtereket, illetve konvolúciót, azaz a képpontokra és azok környezetére a különböző mátrix műveleteket.

A program nem kizárólag csak feldolgoz, hanem létre is hoz ilyen formátumú bittérképes képeket.

A használt könyvtárak:

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <iomanip>
#include <math.h>
#include <cstdlib>
#include <ctime>
```

Programkód működése

A programban globálisan létrehoztam 2 egész konstanst, melyek a létrehozott képek szélességének, illetve magasságának értékét tárolják.

```
const int height = 100;
const int width = 100;
```

Létrehoztam egy „*color*” struktúrát, amelyben 3 egész típusú változót deklaráltam értékadás nélkül:

- *r*: red
- *g*: green
- *b*: blue

Ezen változók a piros, zöld és kék színek előfordulását tartalmazzák egy képpontban értelmezve.

```
struct color
{
    int r, g, b;
};
```

A kép beolvasását egy *input file stream*-el valósítottam meg (*ifstream*), ami tulajdonképpen egy fájlok beolvasására használt adatfolyam, mely képes szöveges formátumú adatállományokkal is dolgozni.

A kép kimenetet egy *output file stream*-el valósítottam meg (*ofstream*), mely hasonlóan az *ifstream*-hez, egy adatfolyam, viszont ez nem beolvasásra használt, hanem az adatok kivitelére vagy fájlok írására.

```
ifstream image_in_stream;
ofstream image_out_stream;
```

Az ezekhez szükséges könyvtár az *fstream* könyvtár (*file stream*).

A bemeneti adatfolyam típusú változóval beolvastam a „*test_image.ppm*” képet:

```
image_out_stream.open("test_image.ppm");
```

Ezután definiáltam a pixelek elrendezésének szabályait, mely információt a „*P3*” karakterlánc tartalmaz, majd megadtam a kimeneti kép szélességét és magasságát is, illetve a maximum felvehető színértéket:

```
image_out_stream << "P3" << endl;
image_out_stream << 255 << " " << 255 << endl;
image_out_stream << 255 << endl;
```

Az adatfolyamot célszerű minden különböző művelet után lezárni:

```
image_out_stream.close();
```

Képek filterezésének megvalósítása

Egy példafunkción keresztül bemutatom a képek filterezésének működését.

Az *AddRedFilterToImage()* funkció:

A függvény várt paraméterei:

- *image_name* (*string*)
- *add_value* (*int*)
- *image_in* (*ifstream*, referencia hivatkozással)
- *image_out in* (*ofstream*, referencia hivatkozással)

A feldolgozandó kép a bemeneti adatfolyammal megnyitásra kerül „párhuzamosan” a kimeneti, feldolgozott képpel együtt:

```
image_in.open(image_name + ".ppm");
image_out.open("red_filtered_" + image_name + ".ppm");
```

Az *is_open()* metódussal ellenőrizhető, hogy az adatfolyamok sikeresen elvégezték a műveleteket, így a bemeneti és a kimeneti adatállomány is megnyitásra került.

Először a fájl *header* (vagy fejléc) adatai kerülnek beolvasásra:

```
string type;
int w, h, maximum_color_value_of_the_image;

image_in >> type >> w >> h >> maximum_color_value_of_the_image;

image_out << type << endl;
image_out << w << " " << h << endl;
image_out << maximum_color_value_of_the_image << endl;
```

A kép típusa (*type*, P3), a kép szélessége (*w*), magassága (*h*), illetve a pixelek színskálájának értéke (*maximum_color_value_of_the_image*).

Ezután az alábbi ciklus elvégzi a pixelekben szereplő piros szín értékének módosítását, pontosabban ennek az értéknek a növelését. Ha a növelt érték a maximum megengedett színskála értékénél nagyobb lenne, akkor a program hozzáadja a két érték differenciáljával elért értéket az adott pixelben szereplő piros szín értékéhez.

```
for ( int i = 0; i < h; i ++ )
{
    for ( int j = 0; j < w; j ++ )
    {
        image_in >> r >> g >> b;

        if ( r + add_value <= maximum_color_value_of_the_image )
        {
            r += add_value;
        }else r += maximum_color_value_of_the_image - (r + add_value);

        image_out << r << " " << g << " " << b << endl;
    }
}
```

Majd ezután az adatfolyamok lezárása következik:

```
image_in.close();
image_out.close();
```

Az *AddGreenFilterToImage()* és az *AddBlueFilterToImage()* funkciók ugyanezen műveleteket hajtják végre, azzal a különbséggel, hogy a pixeleknek más színértékeit változtatják.

Konvolúció

A *ConvolveImage()* funkció bemeneti paraméterként a bemeneti képfájl nevét (image_*name*), a bemeneti adatfolyamot (*ifstream*), és a kimeneti adatfolyamot (*ofstream*) várja.

A függvényben létrehoztam egy mátrixot (*mul*), melynek mérete: n * *m* és lebegőpontos értékek tárolását valósítja meg .

```
int n = 3, m = 3;

float mul[n][m];

mul[0][0] = 0.25;
mul[0][1] = 0.0;
mul[0][2] = -0.25;
mul[1][0] = 0.5;
mul[1][1] = 0.0;
mul[1][2] = -0.5;
mul[2][0] = 0.25;
mul[2][1] = 0.0;
mul[2][2] = -0.5;
```

Ezután a képfájl beolvasásra kerül az adatfolyammal.

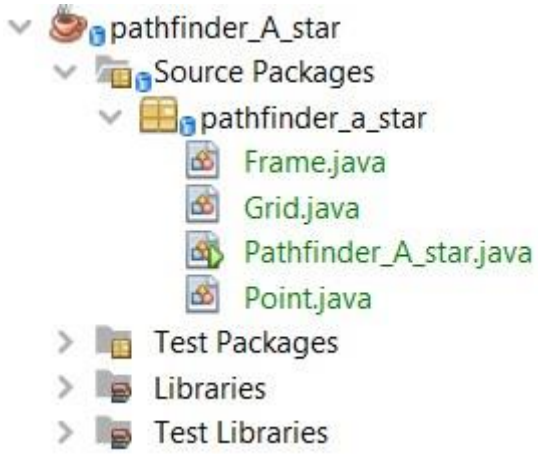
A beolvasást követő négy egymásba ágyazott ciklus a bemeneti képfájl pixeleinek piros, zöld, illetve kék színértékeire alkalmazza a mátrixban megadott értékekkel való szorzást.

Fontos megjegyezni, hogy a program a mátrix méretéhez igazítja az iterációkat, így elkerülve a helytelen indexelési hibákat.

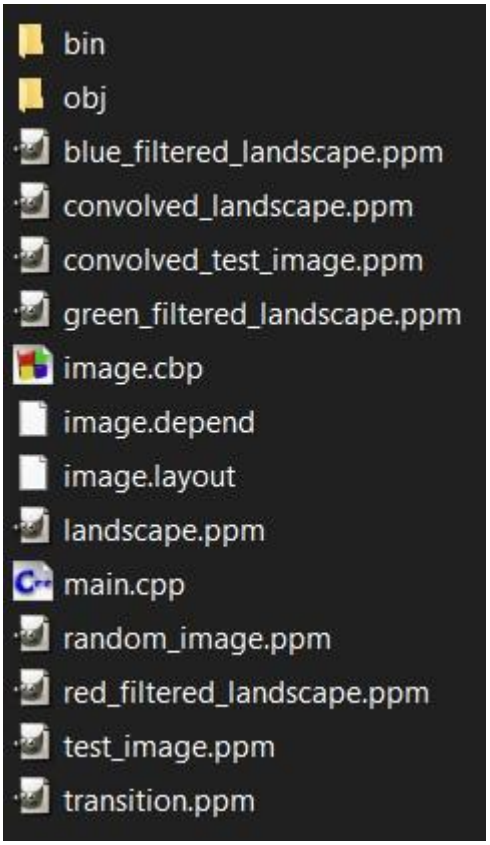
Az így kapott kép egy külön fájlba kerül kimenetként.

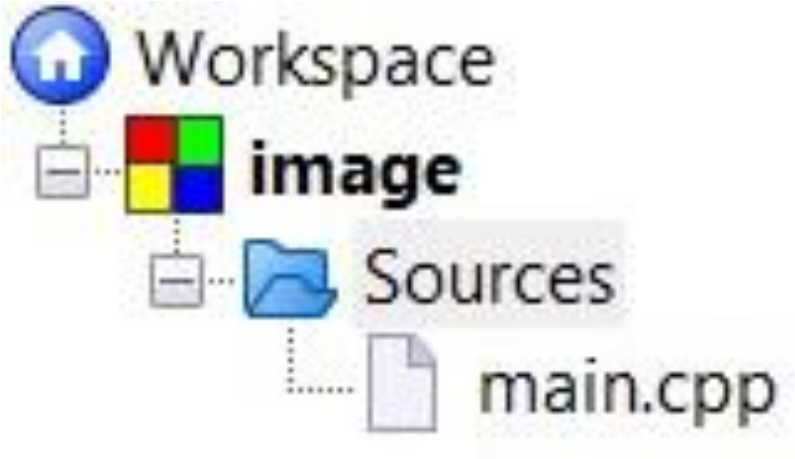
Struktúra szerkezet

1. Feladat

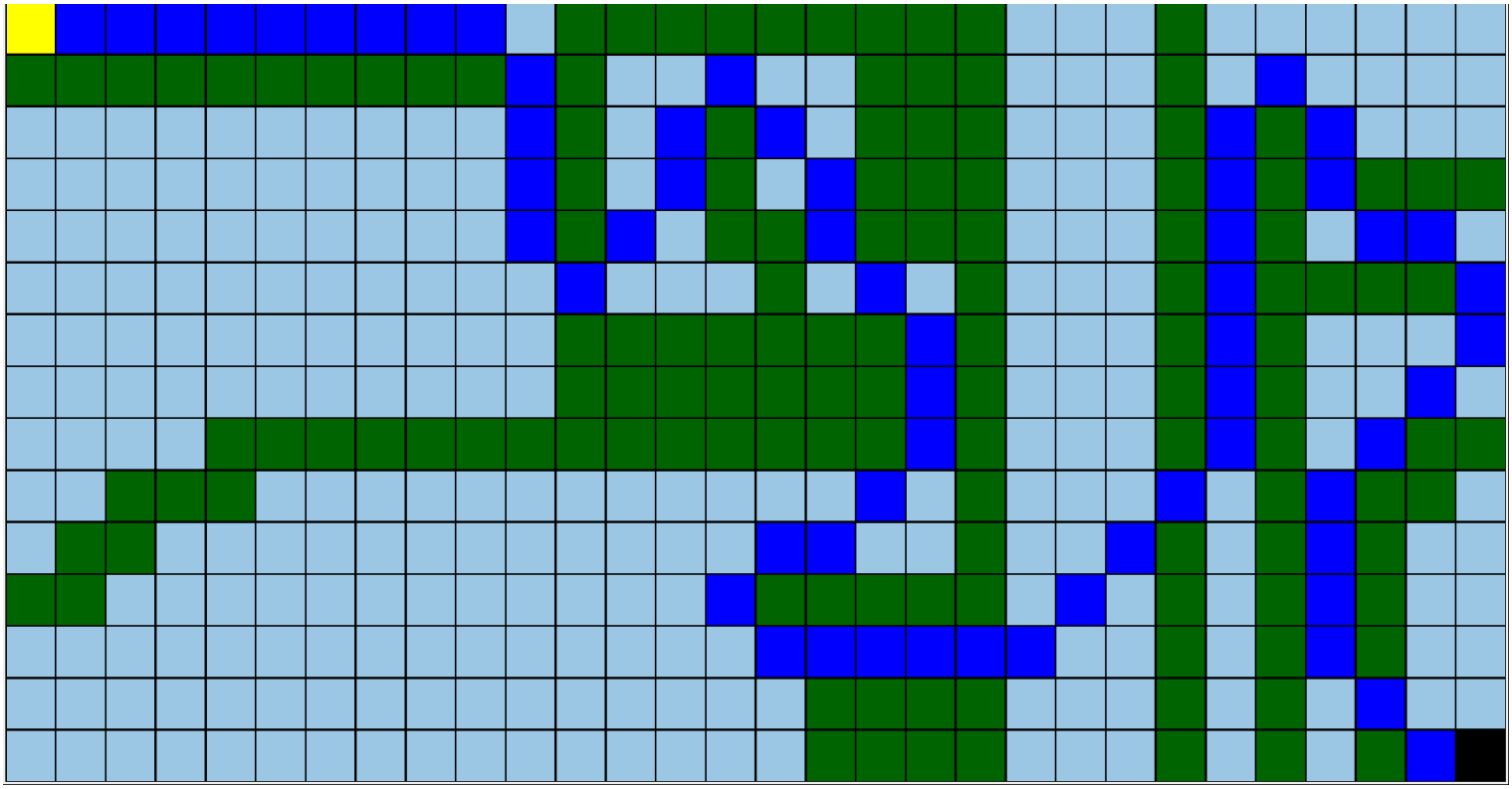


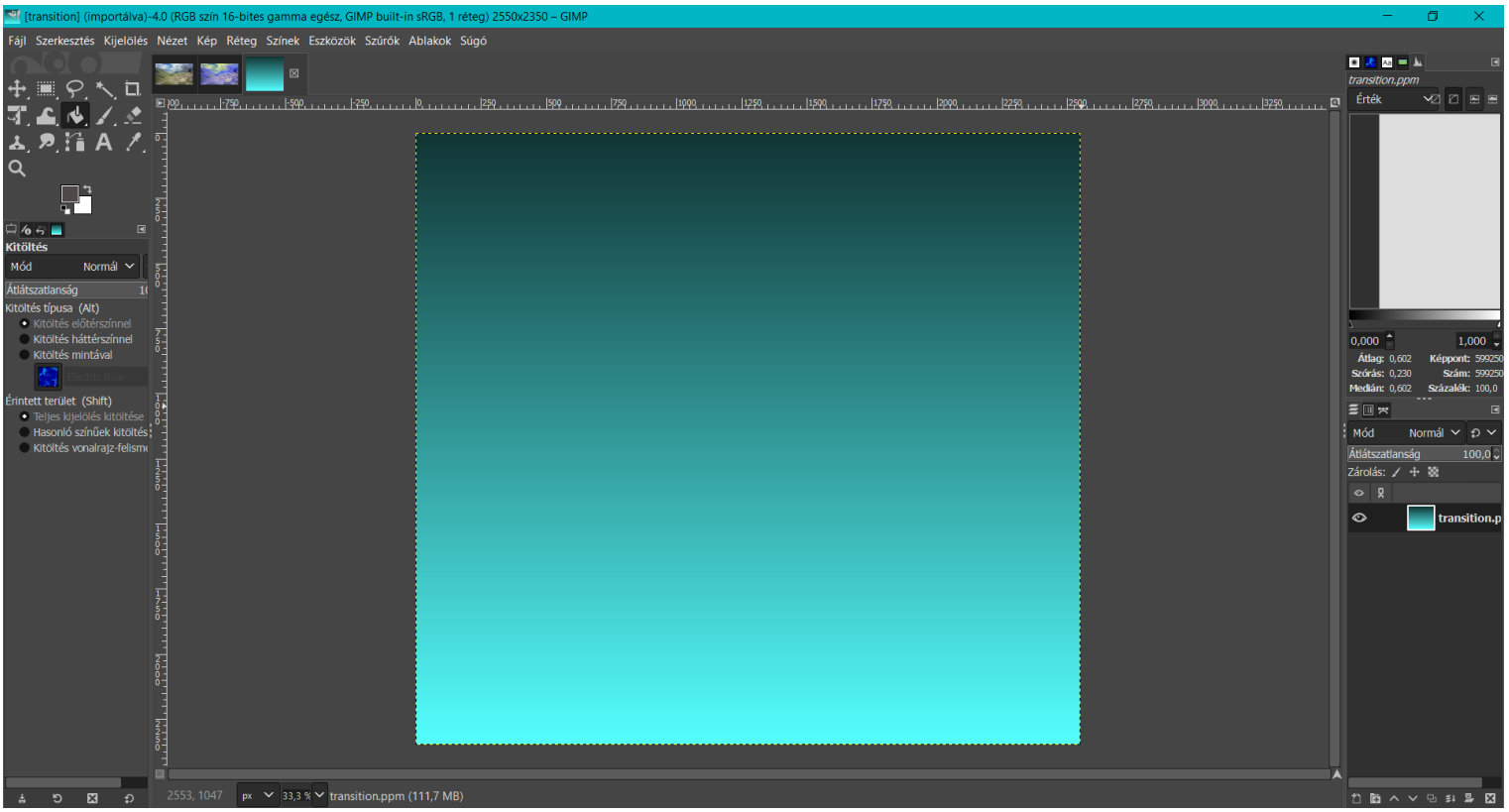
2. Feladat

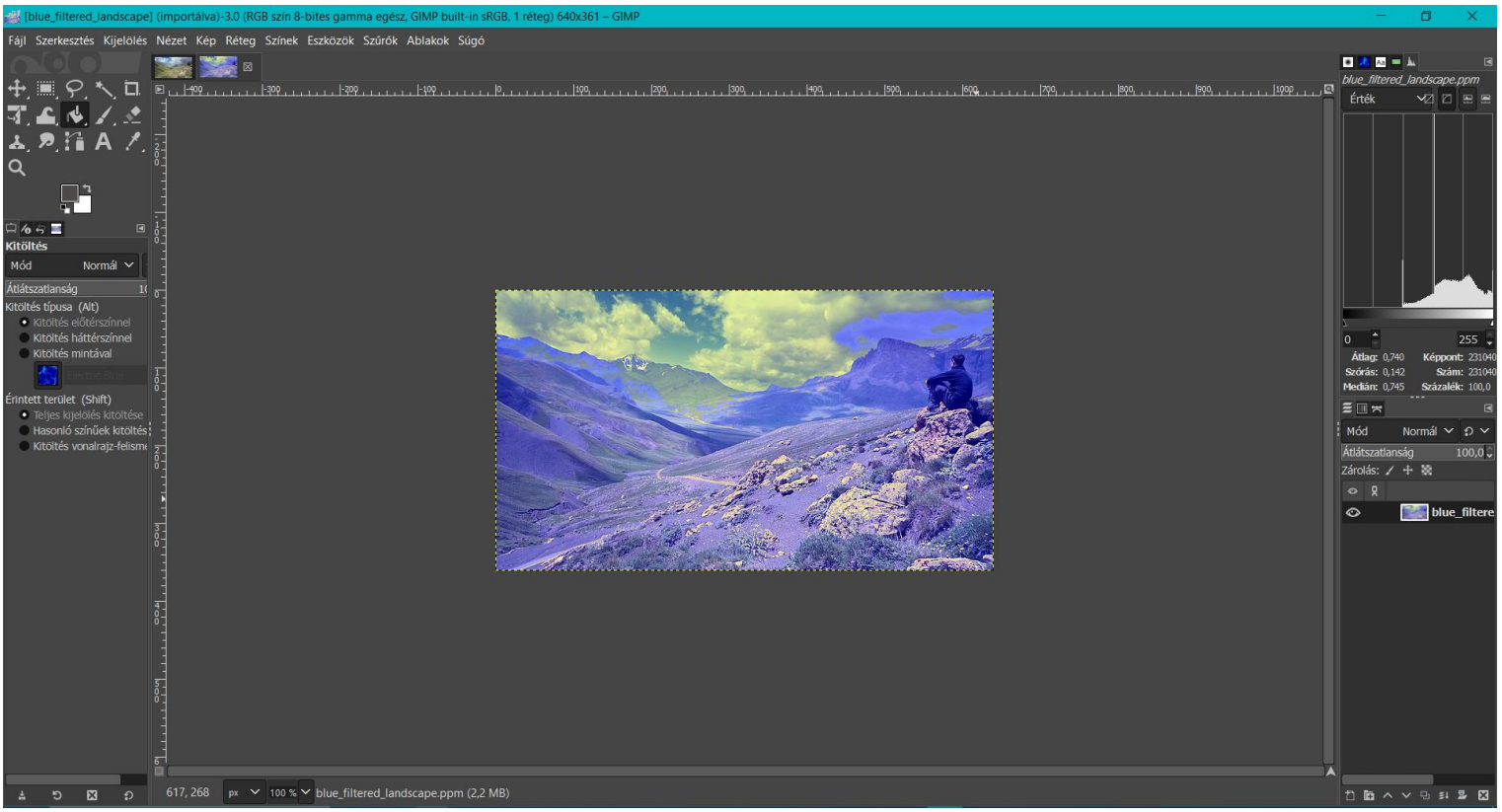




Futtatás eredményei




[transition] (importálva)-4.0 (RGB szín 16-bites gamma egész, GIMP built-in sRGB, 1 réteg) 2550x2350 – GIMP
Fájl Szerkesztés Kijelölés Nézet Kép Réteg Színek Eszközök Szűrők Ablakok Súgó
Kitöltés
Mód Normál
Átlátszatlanság
Kitöltés típusa (Alt)
Kitöltés előtérszínnel
Kitöltés háttérszínnel
Kitöltés mintával
Érintett terület (Shift)
Teljes kijelölés kitöltése
Hasonló színűek kitöltés
Kitöltés vonalrajz-felism
transition.ppm
Érték
Átlag: 0,602 Képpont: 599250
Szórás: 0,230 Szám: 599250
Medián: 0,602 Százalék: 100,0
Mód Normál
Átlátszatlanság 100,0
Zárolás:
transition.p
2553, 1047 px 33,3 % transition.ppm (111,7 MB)


[blue_filtered_landscape] (importálva)-3.0 (RGB szín 8-bites gamma egész, GIMP built-in sRGB, 1 réteg) 640x361 – GIMP
Fájl Szerkesztés Kijelölés Nézet Kép Réteg Színek Eszközök Szűrők Ablakok Súgó
Kitöltés
Mód Normál
Átlátszatlanság
Kitöltés típusa (Alt)
Kitöltés előtérszínnel
Kitöltés háttérszínnel
Kitöltés mintával
Érintett terület (Shift)
Teljes kijelölés kitöltése
Hasonló színűek kitöltés
Kitöltés vonalrajz-felism
blue_filtered_landscape.ppm
Érték
Átlag: 0,740 Képpont: 231040
Szórás: 0,142 Szám: 231040
Medián: 0,745 Százalék: 100,0
Mód Normál
Átlátszatlanság 100,0
Zárolás:
blue_filtere
617, 268 px 100 % blue_filtered_landscape.ppm (2,2 MB)

Források

Inspiráció a képfeldolgozó programhoz:

https://www.youtube.com/watch?v=HGHbcRscFsg

A landscape.ppm képfájl:

https://github.com/ferrabacus/p3images