Szakdolgozat

Miskolci Egyetem

Keretrendszer készítése HTML5 Canvas-re fejlesztett alkalmazások hatékony futtatásához

Készítette:

Kláben Szabolcs Bence

Programtervező informatikus szak

Témavezető:

Piller Imre

Miskolc, 2022

Miskolci Egyetem
Gépészmérnöki és Informatikai Kar
Alkalmazott Matematikai Intézeti Tanszék **Szám:**

Szakdolgozat Feladat

Klában Szabolcs Bence (H46LPD) programtervező informatikus jelölt részére.

A szakdolgozat tárgyköre: HTML5, Canvas API, UNIX

A szakdolgozat címe: Keretrendszer készítése HTML5 Canvas-re fejlesztett alkalmazások hatékony futtatásához

A feladat részletezése:

Böngészős megjelenítő rendszerekben gyakran a HTML5 Canvas-t használják. Ez azt feltételezi, hogy a megjelenítés során egy webböngésző is fut a háttérben. A dolgozat bemutatja egy olyan keretrendszernek a megtervezését, implementációját és tesztelését, amely segítségével célzottan ilyen jellegű alkalmazások készíthetők és futtathatók. Ehhez áttekintésre kerül a HTML5 Canvas API-ja, a hozzá tartozó beviteli eseményekkel. Mérések segítségével összevetésre kerül, hogy a saját keretrendszer erőforrásigénye milyen a böngészős változathoz képest. A keretrendszer UNIX-szerű rendszerek alatt ad lehetőséget majd az alkalmazások natív futtatására.

Témavezető: Piller Imre (Tanársegéd)

A feladat kiadásának ideje: 2021. 09. 29.

. . . . . .
szakfelelős

Eredetiségi Nyilatkozat

Alulírott **Klában Szabolcs Bence**; Neptun-kód: `H46LPD` a Miskolci Egyetem Gépészmérnöki és Informatikai Karának végzős Programtervező informatikus szakos hallgatója ezennel büntetőjogi és fegyelmi felelősségem tudatában nyilatkozom és aláírásommal igazolom, hogy *Keretrendszer készítése HTML5 Canvas-re fejlesztett alkalmazások hatékony futtatásához* című szakdolgozatom saját, önálló munkám; az abban hivatkozott szakirodalom felhasználása a forráskezelés szabályai szerint történt.

Tudomásul veszem, hogy szakdolgozat esetén plágiumnak számít:

- szószerinti idézet közlése idézőjel és hivatkozás megjelölése nélkül;
- tartalmi idézet hivatkozás megjelölése nélkül;
- más publikált gondolatainak saját gondolatként való feltüntetése.

Alulírott kijelentem, hogy a plágium fogalmát megismertem, és tudomásul veszem, hogy plágium esetén szakdolgozatom visszautasításra kerül.

Miskolc, ...év ..... .hó ..... .nap

.....

Hallgató

1.

szükséges (módosítás külön lapon)

A szakdolgozat feladat módosítása

nem szükséges

....

dátum témavezető(k)

2. A feladat kidolgozását ellenőriztem:

témavezető (dátum, aláírás): konzulens (dátum, aláírás):

....

....

....

3. A szakdolgozat beadható:

....

dátum témavezető(k)

4. A szakdolgozat ..... szövegoldalt

..... program protokollt (listát, felhasználói leírást)

..... elektronikus adathordozót (részletezve)

.....

..... egyéb mellékletet (részletezve)

.....

tartalmaz.

....

dátum témavezető(k)

5.

bocsátható

A szakdolgozat bírálatra

nem bocsátható

A bíráló neve: .....

....

dátum szakfelelős

6. A szakdolgozat osztályzata

a témavezető javaslata: .....

a bíráló javaslata: .....

a szakdolgozat végleges eredménye: .....

Miskolc, ...

a Záróvizsga Bizottság Elnöke

Tartalomjegyzék

1. fejezet

Bevezetés

A szakdolgozatom egy már létező egyszerűen használható keretrendszer átültetésével foglalkozik. A cél egy olyan UNIX rendszereken futtatható keretrendszer, amire könnyen átrakhatók a Canvas API-val készített programok, és ennek segítségével kevesebb erőforrás használatával futtathatók.

Az inspiráció a szakdolgozatra a szakmai gyakorlat alatt végzett kutatásomból származik, ahol megismerkedtem mélyebben az UNIX megjelenítő rendszerekről.

A dolgozat 2. fejezete a böngésző alapú grafikus alkalmazásokat tekinti át. Ennek kapcsán láthatjuk majd, hogy a JavaScript milyen széles körben elterjedt, és így milyen előnyök származhatnak abból, hogy ha a JavaScript-ben készített alkalmazások később natív környezetben is futtathatóak lesznek majd.

A dolgozat tulajdonképpen a HTML5 technológiára és az X szerverhez tartozó alacsonyabb szintű implementációkra támaszkodik. A 3. fejezet ezeket taglalja, bemutatva a UNIX alapú rendszerekben a megjelenítés módját, és röviden áttekinti a fejlesztéshez használt eszközkészletet.

A 4. fejezetben az alkalmazás terveit, az egyes részek osztálydiagramjait láthatjuk. Ezek esetében OOP stílusú tervezés történ, így (még hogy ha technikailag nem is tekinthető helyesnek) helyenként egyszerűbbnek tűnt a logikai egységekre, mint osztályokra hivatkozni.

Az 5. fejezet a megvalósítás részleteire tér ki. Kódpéldákkal illusztrálva láthatjuk majd, hogy az elkészült C forráskód hogyan épül fel, hogyan működik, milyen technikai megoldásra volt szükség a HTML Canvas-hez hasonló funkcionalitás eléréséhez. A fejezet végén rövid áttekintést kapunk a C fejlesztéshez használt *cmocka* keretrendszerről és annak használatáról.

A 6. fejezet a kapott eredményeket vizsgálja, vagyis hogy az elkészített natív implementáció mennyivel lehet hatékonyabb, mint az, amelyik a böngészőben fut. Ehhez az Edge, Firefox és Chrome web-böngészők kerültek kiválasztásra, mint referenciák. Hisztogramok formájában láthatjuk majd, hogy hogy alakultak a futási idők a demó programokkal végzett mérések során.

A szakdolgozatom végén található egy használati útmutató, ami bemutatja hogyan lehet a keretrendszert és példákat futtatni Ubuntu és Windows rendszeren is.

2. fejezet

Böngésző alapú grafikus alkalmazások

2.1. A HTML5 szerepe

Napjainkban a böngésző egy univerzális futtatókörnyezetnek tekinthető. Mivel az alkalmazások egy jelentős része grafikus, így természetes módon adódott, hogy a HTML elemek között egy raszteres megjelenítésre alkalmas elem is szerepeljen. Ennek a szerepét a HTML5 Canvas eleme tölti be.

Számos olyan alkalmazással találkozhatunk, amelyek a 2 dimenziós megjelenítést használják. Ilyenek például a

- szerkesztőeszközök,
- folyamat, állapot megjelenítő eszközök,
- grafikon ábrázoló függvénykönyvtárak (például *Chart.js*),
- vektor grafikát használó függvénykönyvtárak (például *Paper.js*) és
- online játékok.

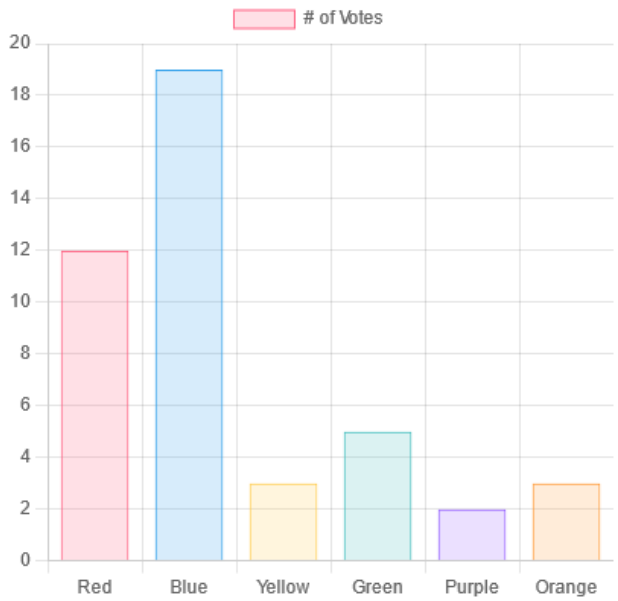
Példaként tekintsük a *Chart.js* függvénykönyvtárat [8]. Ezt alapvetően úgy használhatjuk, hogy egy HTML oldalban kijelölünk a függvénykönyvtár számára egy `canvas` elemet. A 2.1. ábrán láthatunk egy példát egy grafikon megjelenítésére. Ennek elkészítése az alábbi kóddal lehetséges. A helyes működéséhez egy olyan HTML dokumentumra van szükség, amely tartalmaz egy

```
<canvas id="myChart" />
```

elemet. A konkrét kirajzolást (tehát a Canvas API-t) közvetetten, a függvénykönyvtáron keresztül tudjuk használni.

```
const ctx = document.getElementById('myChart');
const myChart = new Chart(ctx, {
    type: 'bar',
    data: {
        labels: ['Red', 'Blue', 'Yellow', 'Green', 'Purple', 'Orange'],
        datasets: [{
            label: '# of Votes',
            data: [12, 19, 3, 5, 2, 3],
```

```
            backgroundColor: [
                'rgba(255, 99, 132, 0.2)',
                'rgba(54, 162, 235, 0.2)',
                'rgba(255, 206, 86, 0.2)',
                'rgba(75, 192, 192, 0.2)',
                'rgba(153, 102, 255, 0.2)',
                'rgba(255, 159, 64, 0.2)'
            ],
            borderColor: [
                'rgba(255, 99, 132, 1)',
                'rgba(54, 162, 235, 1)',
                'rgba(255, 206, 86, 1)',
                'rgba(75, 192, 192, 1)',
                'rgba(153, 102, 255, 1)',
                'rgba(255, 159, 64, 1)'
            ], borderWidth: 1
        }]
    },
    options: { scales: { y: { beginAtZero: true } } }
});
```



2.1. ábra. Chart.js, egy Canvas API-t használó grafikont ábrázoló függvénykönyvtár. A betöltéskor az oszlopok automatikusan animálva vannak. Forrás: [8].

2.2. Natív teljesítmény

A JavaScript egy interpretált nyelv, amit a böngésző dolgoz fel. Mivel a böngésző egy univerzális futtatókörnyezet, így nem tud minden erőforrást a nyelv feldolgozására biztosítani. Segít az interpretáción a V8 JIT, ami bájtkódra (nem gépi kód) fordítja le

a JavaScript kódunkat, ami ténylegesen fel tudja gyorsítani a futtatást, de a probléma attól még fennáll [10].

2.3. Már létező megoldások

A probléma már egy évek óta fenntartó probléma, és már megoldás is született rá. A megoldást egy CrypticSwarm nevű GitHub felhasználó készítette el, de a megoldás már nem aktív 11 éve [5].

A megoldás egy modul *NodeJS*-ben. A technológia egy szintén ettől a felhasználótól származó másik technológia segítségével müködik. A XCB (pontosabban *Cairo* (ami XCB-t használ UNIX rendszereken) hívásokat elérhetővé teszi JS-ben [6]. Ennek a megoldásnak az előnye, hogy cross-platform és nem kell a rajzoló algoritmusokat újraimplementálni.

3. fejezet

Használt technológiák

3.1. HTML5 Canvas API leírása

A HTML5 Canvas egy olyan böngészőben használt JavaScript API, amelynek segítségével könnyen lehet rajzolni a weblapon elhelyezett `<canvas>` elemre. Használják animációkra, böngészős játékokra, adatábrázolásra, képszerkesztésre, és valós-idejű videó feldolgozásra [12].

A Canvas API 2D-s rajzolással foglalkozik, ahol vonalakat, íveket, görbéket, és alakzatokat lehet könnyen rajzolni és animálni. Emellett használható vele a WebGL API, ami hardveresen gyorsított 2D és 3D-s grafikát rajzol. Ez az OpenGL ES 2.0-ból származik [13].

Alább egy egyszerű zöld téglalap rajzolását mutatom be Canvas API-val.

```
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');

ctx.fillStyle = 'green';
ctx.fillRect(10, 10, 150, 100);
```

3.2. Megjelenítés UNIX rendszereken

Az UNIX rendszeren való megjelenítésre az X11 és a Wayland protokoll használható. Ezek közül Wayland viszont még nem terjedt el széleskörben [7].

Így X11 protokollnál maradva, Xlib és XCB függvénykönyvtár közül az utóbbit választottam a kisebb mérete és egyszerűsége miatt [2]. Az XCB-vel egyenesen a megjelenítő szerverrel beszélhetünk, megkerülve nagyobb grafikus könyvtárakat, mint GTK, Qt, vagy akár OpenGL. Ugyanakkor a hátrány is ez, hiszen a rajzolási lehetőségeink alapvetően minimális ezek segítsége nélkül. Említésre méltó így, hogy a WebGL API része a Canvas API-nak nem fogom a szakdolgozaton belül tervezni vagy implementálni.

3.3. JavaScript és C programozási nyelv

Egy JavaScriptben (JS) írt API-t tervezek áthelyezni C-be. Így érdemes az összehasonlításuk:

- JS gyengén típusos, a C erősen típusos
- JS futtáskor értelmezett nyelv, C fordított nyelv
- JS objektum-orientált (OOP), imperatív vagy akár funkcionális, C csak imperatív

Az utóbbi összehasonlítási pont fontos, mert a Canvas API OOP módon lett írva. Ez azt jelenti, hogy C-ben több kódot kell írnunk azért, hogy OOP módszertant használjuk [16].

3.4. Cmocka egységtesztelő keretrendszer

A cmocka egy elegáns egységtesztelő keretrendszer C nyelvhez, amely támogatja a mock objektumokat [17]. Csak a szabványos C könyvtárat igényli, így tökéletes a létrehozandó keretrendszerhez. Az egységtesztelés az a folyamat, amikor a forráskód egységeit elkülönítve, de teljes adatokkal (automatikusan) teszteljük.

3.5. Git verziókezelő rendszer és Github kódtárhely

A git egy verziókezelő rendszer, amely összehasonlítja a jelenlegi kódot az előző verzióval és a különbséget elmenti egyéb metaadatokkal (például ki és mikor írta), így egyszerűsítve a revizionálást, vagy a közreműködést [19].

3.6. Windows Subsystem for Linux

A Windows Subsystem for Linux (WSL) az egy kompatibilitási réteg, ami UNIX rendszerre írt kód futtatását teszi lehetővé Windows rendszereken. Ez a technológia szükséges a Windowson való fejlesztéshez és használathoz [3]. Windows 10-en önállóan viszont nem elég grafikus programok futtatásához.

3.7. VcXsrv

VcXsrv az egy X szerver Windows 10-hez, ami WSL-el együtt működve X11 grafikus alkalmazások futtatását teszi lehetővé Windows 10-en. Ez a technológia is szükséges a Windowson való fejlesztéshez és használathoz [11].

3.8. Github Copilot

A GitHub Copilot több milliárd sor nyilvános kódon alapuló mesterséges intelligencia, ami kódot tud generálni a megírt kód alapján vagy akár kommentekből is [9]. A használatával fel tudom gyorsítani a hasonló kódok írását.

4. fejezet

Tervezés

A fejezet részletesen bemutatja a HTML5 Canvas API C-s implementációjának tervezési folyamatát. Ebben bemutatásra kerülnek az objektum orientált szemlélet szerint felbontott elemek, melyeknek C nyelv esetében a struktúrák és a hozzájuk tartozó függvények felelnek meg.

4.1. Áttekintés

A keretrendszernek képesnek kell lennie

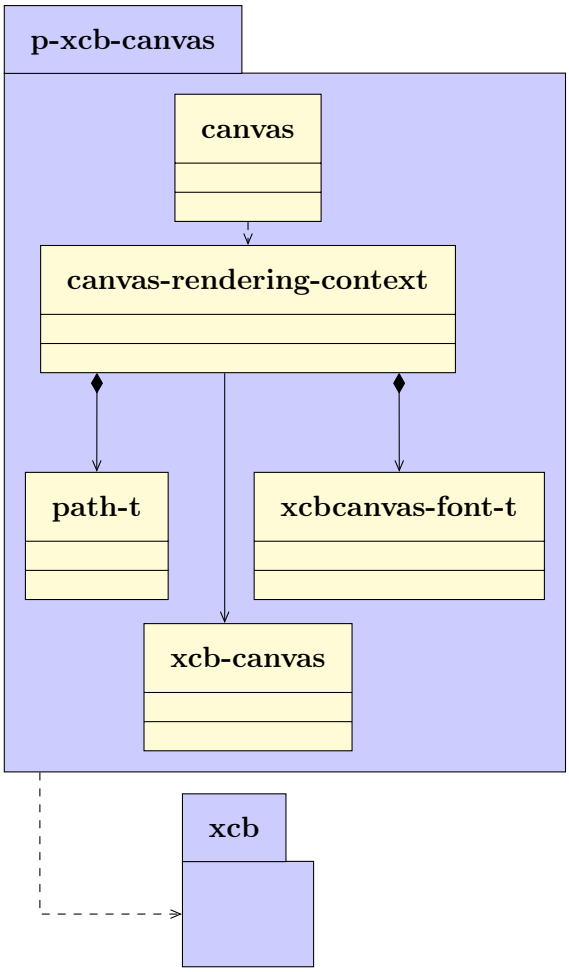
- a képernyőre rajzolni,
- egy programozói felületet adni, amivel egyszerű rajzolni és
- ehhez szükséges adatokat és állapotokat tárolni.

Erre a három feladatra jutott egy-egy osztály. Ezt a 4.1. ábra szemlélteti.

4.2. Képernyőre rajzolás

Az XCB keretrendszerrel való kommunikációt absztrahálja, és az ahhoz szükséges globális változókat tárolja. Ez pontosabban az X-szerverrel való kapcsolatot, az ablakunk ID-jét és a X-nek fontos grafikai állapotokat tartalmazó struktúrát jelenti. Kerültek továbbá ide függvények, ami XCB segítségével rajzol a képernyőre valamilyen primitívet (téglalap vagy ív), vagy az ablak egy tulajdonságát (például mérete vagy címe) módosítja. Emellett ide jutott az ablakot létrehozó kód is. Ezt a 4.2. ábra szemlélteti.

A 4.2. ábrán a plusz- és mínuszjel a láthatóságot jelzi, amely lehet publikus vagy privát. C-ben úgy kerül megvalósításra, hogy csak a forráskódban – a header fájlban nem – kerül definiálásra a struktúra [16].



4.1. ábra. Áttekintő UML az osztályokról

4.3. Programozói felület

A felhasználó által használható metódusok és függvények ebben az osztályban kaptak helyet. Ezek a függvények követik a Canvas API szignatúráját azzal a különbséggel, hogy az ablakra rajzoláshoz extra információt tartalmazó osztályt is átadjuk a függvényeknek.

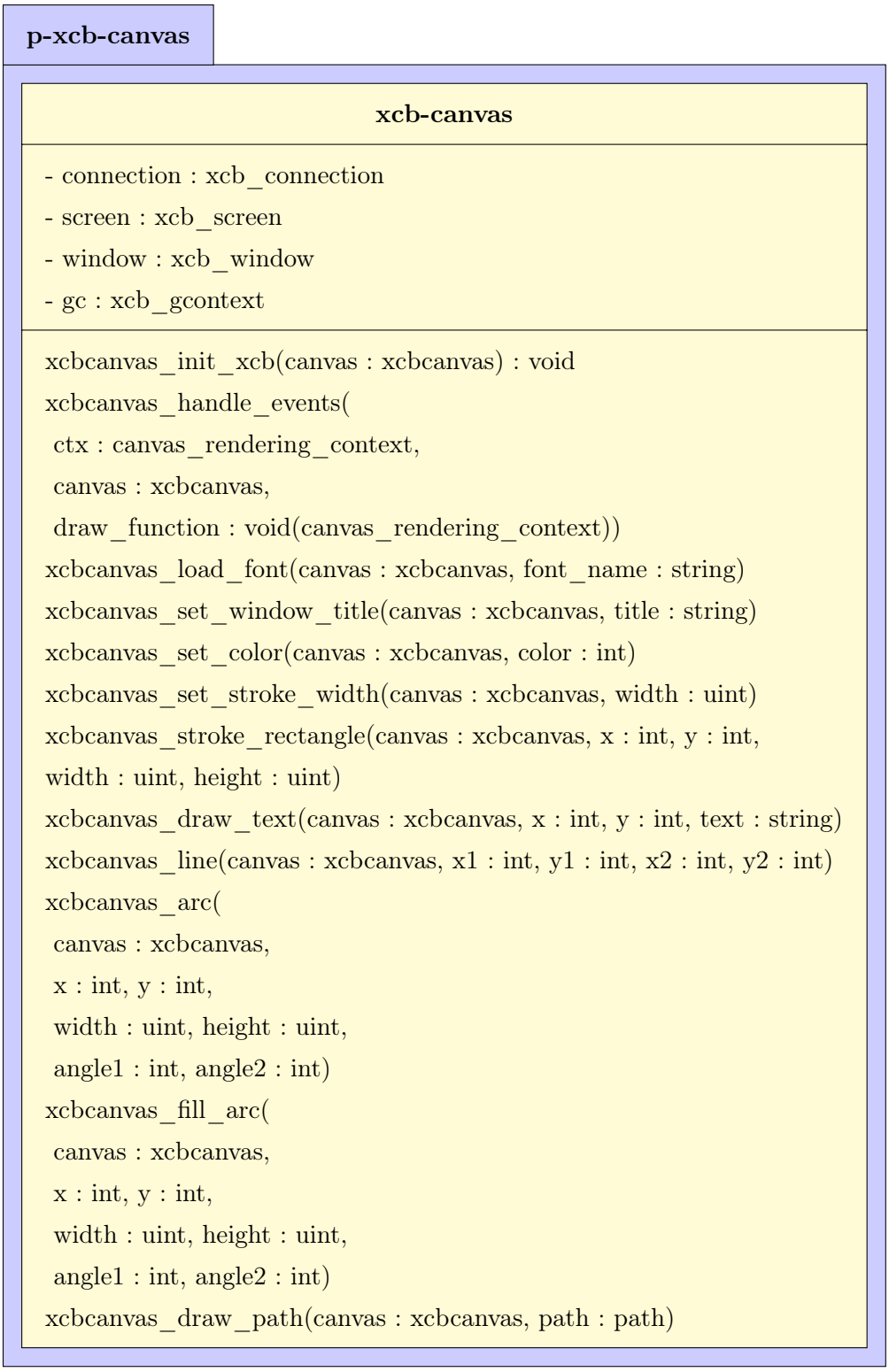
Előfordul, hogy egy névvel több szignatúrájú függvény van definiálva JS-ben, bár ez nem lenne probléma C++-ban, C-ben az. A probléma egyszerűen orvosolható a függvény nevének a módosításával, például az argumentumok számának változtatásával. Az OpenGL hasonló megoldást ad a problémára.

A `canvas` struktúrához tartozó függvényeket a 4.3. ábra szemlélteti.

4.4. Az állapotok kezelése

A különböző állapotok kezeléséhez szükséges adatok, metódusok és függvények osztálya a `canvas_rendering_context`. A különböző állapotok, amelyeket a keretrendszer kezel az a

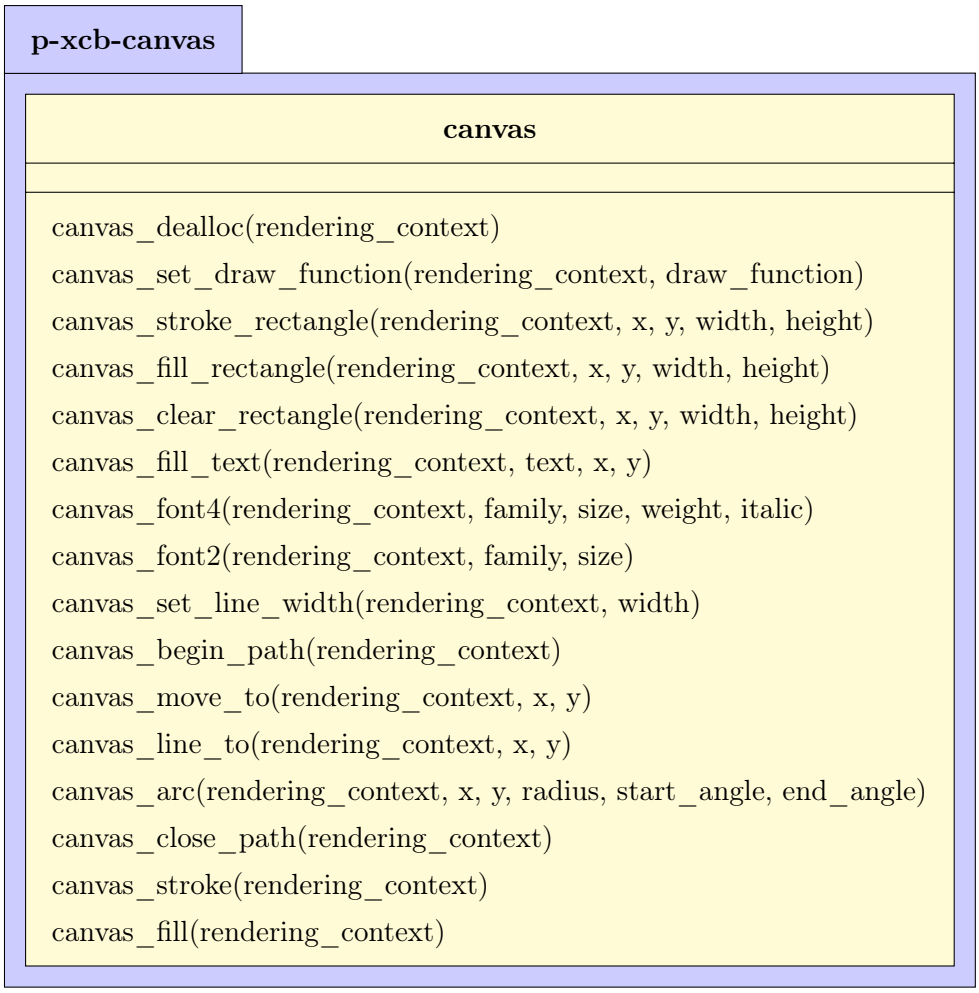
- betűtípus címe, mérete, kövérsége, dőlése, és
- `Path`-hez kiadott utasítások.



4.2. ábra. XCB kezelő osztály UML ábrája

A betűtípust kétféle függvénnyel lehet módosítani a `canvas` osztályból, majd a beállított betűtípus kerül felhasználásra szöveg kirajzolásakor. A `Path`-be sorként kerülnek az utasítások bele `sub_path`-ba, ami dinamikus tömbként van megoldva. Továbbá tárolásra kerül a tömb mérete, és hogy az alakzat kitöltött-e, vagy csak körvonal.

A `canvas_rendering_context` struktúra elemeit, és a hozzá tartozó függvényeket

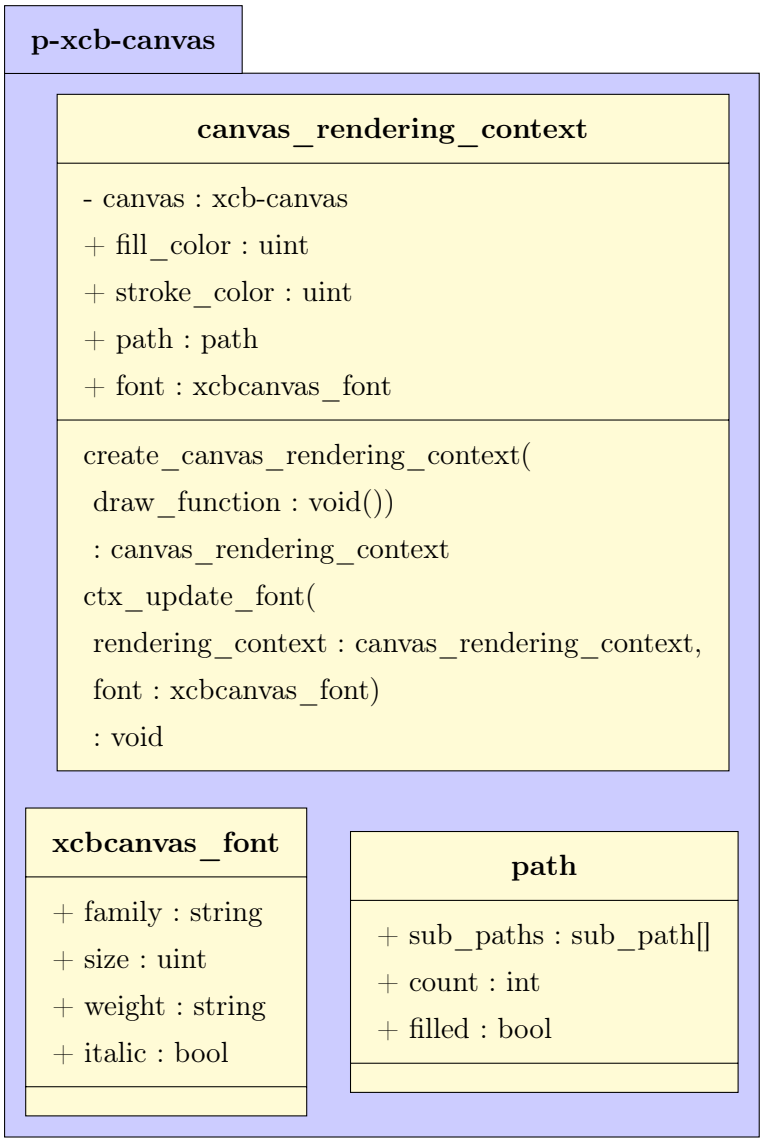


4.3. ábra. Felhasználó felőli osztályok UML ábrája

a 4.4. ábra szemlélteti.

4.5. XCB és Canvas hívások összehasonlítása

Az alábbi 4.1. táblázat összehasonlít néhány függvényt XCB és Canvas között ugyanarra a problémára. A táblázatból hamar feltünik, hogy át kell adni az állapotokat tároló változókat és az XCB-nek struktúraként kell átadni a színt és vastagságot. Valójában itt egy hívással tudnánk ezeket és többet változtatni egyszerre és a változtatások értékeit kell egy tömbbe rakni. Az értékek sorrendje adott. A `rectangle` struktúrája egy x, y, *szélesség* és *magasság*ból áll. A `points` pedig egy pontokból álló tömb, ahol a pont egy x és y-ból álló struktúra. A szöveg kiírásánál alacsony szintű programozás miatt a szöveg hosszát is át kell adni.



4.4. ábra. Adatok kezeléséhez szükséges osztályok UML ábrája

4.1. táblázat. Függvények összehasonlítása

Függvény	**XCB**	**Canvas**
Szín változtatása	`xcb_change_gc( c, gc, XCB_GC_FOREGROUND, &color);`	`ctx.strokeStyle = "#FF0000";`
Vonalvastagság változtatása	`xcb_change_gc( c, gc, XCB_GC_LINE_WIDTH, &width);`	`ctx.lineWidth = 20;`
Kitöltött téglalap kirajzolása	`xcb_poly_fill_rectangle( c, canvas->window, gc, 1, &rectangle);`	`ctx.fillRect(0, 0, 100, 100);`
Vonal rajzolása	`xcb_poly_line( c, XCB_COORD_MODE_ORIGIN, canvas->window, gc, 2, points);`	`ctx.lineTo(300, 200);`
Szöveg kiírása	`xcb_image_text_8_checked( canvas->connection, strlen(text), canvas->window,canvas->gc, x,y,text);`	`ctx.fillText( "Hello World", 500, 200);`

5. fejezet

Megvalósítás

5.1. Fejlesztőkörnyezet előkészítése Windowson

A feladat UNIX-alapú keretrendszer készítése, emiatt Windowson néhány előkészületre volt szükségem a fejlesztés elkezdése előtt. Először is, egy UNIX-alapú rendszerre van szükség WSL segítségével. Ehhez pedig, engedélyeznem kellett a WSL-t, majd telepíteni egy Linux disztribúciót [15]. Ubuntura esett a választásom egyszerűsége és elterjedtsége miatt. A WSL-lel való fejlesztéshez pedig a Visual Studio Code-t (VSCode) ajánlja Microsoft és lett használva, emiatt pedig egy kiegészítőt is kellett telepíteni VSCode-hoz [4].

Grafikus alkalmazások fejlesztéséhez viszont szükség van az alábbi csomagokra is:

```
sudo apt install build-essential libx11-xcb-dev pkg-config
```

Továbbá a grafikus felület látásához, szükség van egy X-szerver (például VcXsrv) telepítésére Windowson is, a fejlesztőfelület beállítására, hogy irányítsa át a megjelenítést a Windowson futó szervere, és végül a tűzfalon is állítani kellett [18]. Természetesen, egy új repository is létrehozásra került *GitHub*-on.

5.2. XCB használata

Egy ablak megjelenítéséhez és az abba való rajzoláshoz UNIX alapú rendszeren X11 protokollt használva az XCB függvénykönyvtár és keretrendszer van használva. A keretrendszer használata bár egyszerűbb a többinél, még így is bonyolult, ezért leegyszerűsítő metódusokat hoztam létre. Továbbá a megértéséhez az XCB oldalán elérhető útmutatót használtam fel [14]. Az eredmény egy fehér ablak a képernyőn. A használt példa kód az útmutatóban megtalálható.

5.2.1. Primitívek implementálása

A primitívek és a Path primitívei implementálása egyszerű volt – kivéve a görbéket – ugyanis XCB is támogatta őket. A nem támogatottokat kihagytam idő nyerése érdekében. Példának alább egy téglalap körvonalát rajzoló segítő függvény kódja olvasható a `xcb-canvas` osztályból. Ez a kód XCB könyvtár segítségével az X-szervernek küldi el, hogy rajzolja ki a téglalap körvonalát az adott pozícióra és az adott dimenziókkal.

```
void xcbcanvas_stroke_rectangle(
    xcbcanvas_t* canvas,
    int16_t x, int16_t y,
    uint16_t width, uint16_t height
)
{
  xcb_connection_t* c = canvas->connection;
  xcb_gcontext_t gc = canvas->gc;
  xcb_rectangle_t rectangle;
  rectangle.x = x;
  rectangle.y = y;
  rectangle.width = width;
  rectangle.height = height;
  xcb_poly_rectangle(c, canvas->window, gc, 1, &rectangle);
}
```

Munkámat támogatta a Github Copilot technológiája is. A technológia segítségével gyorsabban tudtam implementálni a primitíveket és a Path primitíveit, hiszen a kódok nagyon hasonlóak.

Ezek után a függvénykönyvtár-felhasználó által használt kódot is implementáltam a `canvas` osztályban, a felső példát használva az alábbi módon. Ezek a metódusok már a teljes `canvas_rendering_context` globális adatait is kéri, és az argumentumok sorrendje a Canvas API sorrendjében vannak. Továbbá, az ebbe az osztályba kerülő metódusok részletesebb dokumentációt kapnak a header fájlokban.

```
void canvas_stroke_rectangle(
    canvas_rendering_context_t* rendering_context,
    int16_t x, int16_t y,
    uint16_t width, uint16_t height
)
{
    xcbcanvas_stroke_rectangle(
        rendering_context->canvas, x, y, width, height);
}
```

5.2.2. Path implementálása

A Path implementálására, egy dinamikusan növekedő tömböt implementáltam, amibe a Path primitíveit vagy egyéb utasításait tárolom. A dinamikus tömb 10 elemszámonként növekedik. Az alábbi példa a Path kezdését mutatja, illetve a kódpéldát dinamikus növekedésre.

```
void canvas_begin_path(
    canvas_rendering_context_t* rendering_context
)
{
    rendering_context->path->sub_path_count = 0;
    rendering_context->path->sub_paths = malloc(sizeof(sub_path_t) * 10);
}
```

```
/* Továbbá minden egyéb Path utasítás elején: */
 if (rendering_context->path->sub_path_count % 10 == 9)
    {
        rendering_context->path->sub_paths = realloc(
            rendering_context->path->sub_paths,
            sizeof(sub_path_t) *
            (rendering_context->path->sub_path_count + 10));
    }
    int index = rendering_context->path->sub_path_count;
    sub_path_t* sub_path = &rendering_context->path->sub_paths[index];
```

A Path befejezésekor a körvonal vagy a kitöltött választás alapján a megfelelő rajzoló függvényeket hívom meg. Bonyolultságot adott a Path-bezárás függvény implementációja, ugyanis menteni kell, hogy melyik pontra ugorjon vissza, és hogy egyáltalán ugorjon-e vissza arra a pontra. Az alábbi kód bemutatja a Path kirajzolás függvényét. Érdekes határeset, amikor csak egy primitív van a Path-ben ugyanis azt mindig csak egy mozgás utasításnak van véve.

```
void xcbcanvas_draw_path(xcbcanvas_t* canvas, path_t* path)
{
  if (path->sub_path_count == 0) {
    printf("Error: No sub-paths in path\n");
    return;
  }

  /*  Technically valid, but would do nothing
      since the first sub-path is treated as just a move to.
  */
  if (path->sub_path_count == 1) return;

  xcb_point_t current_position = path->sub_paths[0].point;
  xcb_point_t closing_point = path->sub_paths[0].point;
  /* Keep track of all the points
     that make up the shape that would need to be closed */
  xcb_point_t points[path->sub_path_count + 1];
  points[0] = current_position;
  uint16_t moves_since_close = 0;
  /* Render an outline path */
  if (!path->filled) {
    for (int i = 0; i < path->sub_path_count; i++) {
      switch (path->sub_paths[i].type) {
        case SUBPATH_TYPE_MOVE:
          /* Move pen to the given position */
          current_position = path->sub_paths[i].point;
          /* Add the point to the list of points
             that need to be closed */
          if (moves_since_close > 1)
            points[moves_since_close + 1] = closing_point;
```

```
        /* Reset moves since close counter */
        moves_since_close = 0;
        break;
      case SUBPATH_TYPE_LINE:
        /* Draw line between current position and the given one */
        xcbcanvas_line(canvas,
          current_position.x,
          current_position.y,
          path->sub_paths[i].point.x,
          path->sub_paths[i].point.y);
        /* Update currrent position */
        current_position = path->sub_paths[i].point;
        points[++moves_since_close] = current_position;
        break;
      case SUBPATH_TYPE_ARC:
        xcbcanvas_arc(canvas,
          path->sub_paths[i].point.x,
          path->sub_paths[i].point.y,
          path->sub_paths[i].arc.radius * 2,
          path->sub_paths[i].arc.radius * 2,
          path->sub_paths[i].arc.start_radius_or_cp2_x,
          path->sub_paths[i].arc.end_radius_or_cp2_y);
        break;
      case SUBPATH_TYPE_ARC_TO:
      case SUBPATH_TYPE_QUADRATIC_CURVE:
      case SUBPATH_TYPE_CUBIC_CURVE:
        printf("Error: Unsupported path sub-path type: %d\n",
          path->sub_paths[i].type);
        break;
      case SUBPATH_TYPE_CLOSE:
        /* Close the shape */
        if (moves_since_close > 1)
          points[moves_since_close + 1] = closing_point;
        xcbcanvas_line(canvas,
          current_position.x, current_position.y,
          closing_point.x, closing_point.y);
        current_position = path->sub_paths[0].point;
        break;
    }
  }
}
/* Render a filled path */
else {
  /*  Same as above, but with the filled variants
  *   When closing a shape also need to make sure
  *   to correctly fill in the shape.
  */
  // [...]
```

```
  }
}
```

5.2.3. Szöveg implementálása

Szöveg implementálásra az XCB két lehetőséget ad; egy elavult egyszerű módot, ahol csak néhány betűtípus és méret elérhető, illetve egy újabb komplikáltabb és flexibilisebbet. Az egyszerűbb verziót választottam, ugyanis annak az implementálása van benne az általam használt útmutatóban.

Az egyszerűbb verzióban először frissíteni (be kell tölteni) az X-szerverben tárolt adatot a használandó betűtípusról. Érdekesség, hogy a betűtípus azonosítója csak egy betűlánc, ami tárolja a nevét, méretét és egyéb tulajdonságait.

```
void xcbcanvas_load_font(xcbcanvas_t* canvas, char* font_name)
{
  // First, create an id for the font
  xcb_font_t font = xcb_generate_id(canvas->connection);
  // Secondly, open the font
  xcb_void_cookie_t cookie =
    xcb_open_font_checked(
        canvas->connection, font, strlen(font_name), font_name);
  // Check for errors
  xcb_generic_error_t* error = xcb_request_check(
    canvas->connection, cookie);
  if (error)
  {
    fprintf(stderr, "Error opening font: %s\n", font_name);
    free(error);
    return;
  }
  // Then, assign font to our existing graphic context
  cookie = xcb_change_gc(
    canvas->connection, canvas->gc,
    XCB_GC_FONT, (uint32_t[]) { font });
  // Check for errors
  error = xcb_request_check(canvas->connection, cookie);
  if (error)
  {
    fprintf(stderr, "Error assigning font to graphic context\n");
    free(error);
    return;
  }
  // Close the font
  cookie = xcb_close_font_checked(canvas->connection, font);
  error = xcb_request_check(canvas->connection, cookie);
  if (error)
  {
    fprintf(stderr, "Error closing font\n");
    free(error);
```

```
    return;
  }
}
```

Miután megadtuk az X-szervernek a betűtípust, felhasználhatjuk szöveg megjelenítéséhez az alábbi módon.

```
void xcbcanvas_draw_text(
    xcbcanvas_t* canvas,
    int16_t x, int16_t y,
    const char* text)
{
  // Draw the text to the screen with the font that has been set
  xcb_void_cookie_t cookie_text = xcb_image_text_8_checked(
    canvas->connection, strlen(text),
    canvas->window, canvas->gc,
    x, y, text);
  xcb_generic_error_t* error =
    xcb_request_check(canvas->connection, cookie_text);
  if (error) {
    fprintf(stderr, "Error: Can't paste text: %d\n", error->error_code);
    free(error);
  }
}
```

5.2.4. További tervezett funkciók

Primitívek, amelyek nem voltak az XCB-ben nem kerültek implementálásra. Például

- a másodfokú és harmadfokú Bézier-görbe, ezekre léteznek már algoritmusok, amiket csak implementálni kell [1], vagy
- `arcTo`, ami egy ív a kezdőponttól, két pont és egy sugár alapján.

Ez az ív úgy képzelhető el, hogy a kezdő- és első pont, illetve az első és második pont érintőegyenest alkot az ívnek. Emellett, a kezdőpontot és az ív tényleges kezdését egy vonallal kell összekötni. Ha pedig a sugarat növeljük, akkor az ív kerekebb lesz és távolabb kerül az első ponttól. Egy példa `arcTo`-ra látható az 5.1. ábrán. Az `ArcTo`-ra nem tudom, hogy létezik-e hasonlóan megadott algoritmus, így lehetséges önállóan kell előállítani. Az ábrához használt kód alább olvasható.

```
const canvas = document.getElementById('canvas');
const ctx = canvas.getContext('2d');
// Tangential lines
ctx.beginPath();
ctx.strokeStyle = 'gray';
ctx.moveTo(200, 20);
ctx.lineTo(200, 130);
ctx.lineTo(50, 20);
ctx.stroke();
```

5.1. ábra. Példa `arcTo`-ra. A kék a kezdő pont, a piros pontok pedig a két másik végpont.

```
// Arc
ctx.beginPath();
ctx.strokeStyle = 'black';
ctx.lineWidth = 5;
ctx.moveTo(200, 20);
ctx.arcTo(200,130, 50,20, 40);
ctx.stroke();
// Start point
ctx.beginPath();
ctx.fillStyle = 'blue';
ctx.arc(200, 20, 5, 0, 2 * Math.PI);
ctx.fill();
// Control points
ctx.beginPath();
ctx.fillStyle = 'red';
ctx.arc(200, 130, 5, 0, 2 * Math.PI); // Control point one
ctx.arc(50, 20, 5, 0, 2 * Math.PI);   // Control point two
ctx.fill();
```

Animáció implementálása többszálassá tenné programot, ugyanis utólag tudtam meg, hogy XCB csak akkor futtattja a rajzoló kódot, amikor egy ablak eddig takart része előtűnik (például megnyitás, tálcára minimalizálás visszavonása, takaró ablak elhúzása, aktív ablakká választás, ablak méret változtatása). Ez azt jelenti, hogy animáció implementáláshoz például egy külön szálon lenne szükség az időt tartanom, és egy eseményt küldeni a főszálnak, hogy frissüljön. A többszálas programok komplexitása miatt így nem implementáltam. A bemenet kezelésére nem tekintettem rá.

Továbbá, a Canvas API hatalmas, és nem mindenre tekintettem rá. Például átláthatóság, színkeverés, színátmenet, vagy akár kép betöltése mind olyan, amit nem sikerült lefedni, és nem is tudom, hogy XCB-ben hogyan lehetséges vagy az algoritmusokat nekem kell implementálni az előbb felsorolt dolgokhoz. Végül, az egységtesztelés is csak minimálisan lett implementálva.

5.3. Egységtesztelés

Az XCB programok egységteszteléshez például a *cmocka* függvénykönyvtár telepítésére és használatára van szükség (`https://cmocka.org`). A telepítéséhez letöltöttem a weboldaláról, kicsomagoltam, majd lefordítani a programot, végül a telepítőt rendszergazdaként futtattam. Ez a terminálban az alábbi módon tehető meg.

```
wget https://cmocka.org/files/1.1/cmocka-1.1.5.tar.xz
tar -xf cmocka-1.1.5.tar.xz
cd cmocka-1.1.5.tar.xz
mdkir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=/usr -DCMAKE_BUILD_TYPE=Debug ..
make
sudo make install
```

Ezután mockolással és tesztek készítésével, továbbá a `Makefile` módosításával lehetőség van automatikusan tesztelni a kódot minden fordításkor. A tesztek egy külön `tests` jegyzékben kaptak helyet. Alább a módosított `Makefile` olvasható.

```
tests: clean canvas.o xcb-canvas.o tests.o
        $(CC) $(STD) $(INCLUDE) -Wall -g tests/* \
        $(OUTPUT_FOLDER)/canvas.o $(OUTPUT_FOLDER)/xcb-canvas.o \
        $(OUTPUT_FOLDER)/tests.o \
        -o $(OUTPUT_FOLDER)/tests `$(PKG_CONF) xcb x11 cmocka` $(WRAP) && \
        ./$(OUTPUT_FOLDER)/tests
tests.o:
        $(CC) $(STD) $(INCLUDE) -Wall -g -c src/tests.c \
        -o $(OUTPUT_FOLDER)/tests.o `$(PKG_CONF) cmocka` $(WRAP)
```

Végül a teszteket meg is kellett írnom. Az egyik leggyakoribb teszt az a null mutatók elleni védelem. Egy ilyen tesztet mutat be az alábbi kód.

```
void xcbcanvas_draw_path_test(void** state)
{
    xcbcanvas_draw_path(NULL, NULL);
    path_t* path;
    path = malloc(sizeof(path_t));
    path->sub_paths = NULL;
    xcbcanvas_draw_path(NULL, path);
    free(path);
}
```

Mivel a kódom még nem volt védve null mutatók ellen, a sikertelen teszt üzenete várt:

```
[==========] Running 1 test(s).
[ RUN      ] xcbcanvas_draw_path_test
[  ERROR   ] --- Test failed with exception: Segmentation fault(11)
[  FAILED  ] xcbcanvas_draw_path_test
[==========] 1 test(s) run.
```

Cmocka kiírja, hogy melyik teszt lett sikertelen és miért. Ez esetben, a null mutató `Segmentation fault`-ot okozott a programomban. A programomat javítottam az alábbi kód hozzáadásával.

```
if (path == NULL) {
    printf("Error: Canvas path is NULL\n");
    return;
}
if (path->sub_paths == NULL) {
    printf("Error: Sub-paths is NULL\n");
    return;
}
```

A kód hozzáadása után a teszt sikeressé is vált, amit a *cmocka* az alábbi módon közölt velem. (Fontos megjegyezni, hogy a programomtól való hibaüzenetek várunk ebben az esetben.)

```
[==========] Running 1 test(s).
[ RUN      ] xcbcanvas_draw_path_test
Error: Canvas path is NULL
Error: Sub-paths is NULL
[       OK ] xcbcanvas_draw_path_test
[==========] 1 test(s) run.
```

Hozzáadható lehetőség a repository-hoz, hogy mikor új kód kerül hozzáadásra, akkor azon automatikusan futtassa le ezeket a teszteket, és értesítse a fejlesztőket, ha valamelyik sikertelen.

6. fejezet

Számítási teljesítmény mérése és összehasonlítása

Az erőforrásigény összehasonlítását a Firefox egyik böngészőlapjának, és az én általam elkészített keretrendszerben egy példaalkalmazás memóriaigényét és CPU kihasználtságát egy Linux Mint virtuális gépben.

6.1. Mérés böngészőkben

Először is a példa alkalmazást el készítettem egy HTML fájlként beágyazott Javascripttel. Miután biztosra mentem, hogy a két kód, ugyanazt az eredményt hozza grafikusan és a két ablak ugyanakkora, felhoztam a böngésző feladatkezelőjét. Az eredmény 511 KB memória és 0 CPU használat volt a böngészőlapnak. A futásideje a kódban lett mérve, amit tizenötször futattam, és *Python* segítségével ábrázoltam.

A böngészők verziói pedig

- Firefoxnak 100.0,
- Google Chrome-nak 101.0.4951.41,
- Microsoft Edge-nek 101.0.1210.32.

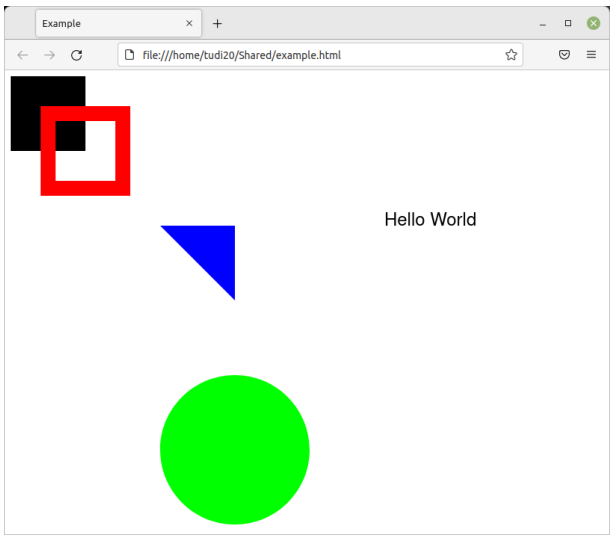
6.1.1. Mérés demó programmal

Továbbá, futtattam a kódot különböző böngészőkben a Windows 10-emben és a VM-ben is, amelynek a hisztogramja a 6.2. ábrán látható. Az ábráról a láthatóság kedvéért ki lettek hagyva a böngészők VM-ben való futtatása, mert azok futásideje már [1,60] ms intervallumba esett. Ezekből az adatokból arra lehet következtetni, hogy az összes Chromium-alapú böngésző hasonló futásidővel fog futni, és hogy Firefox gyorsabb a Canvas API esetén, mint a Chromium-alapú böngészők.

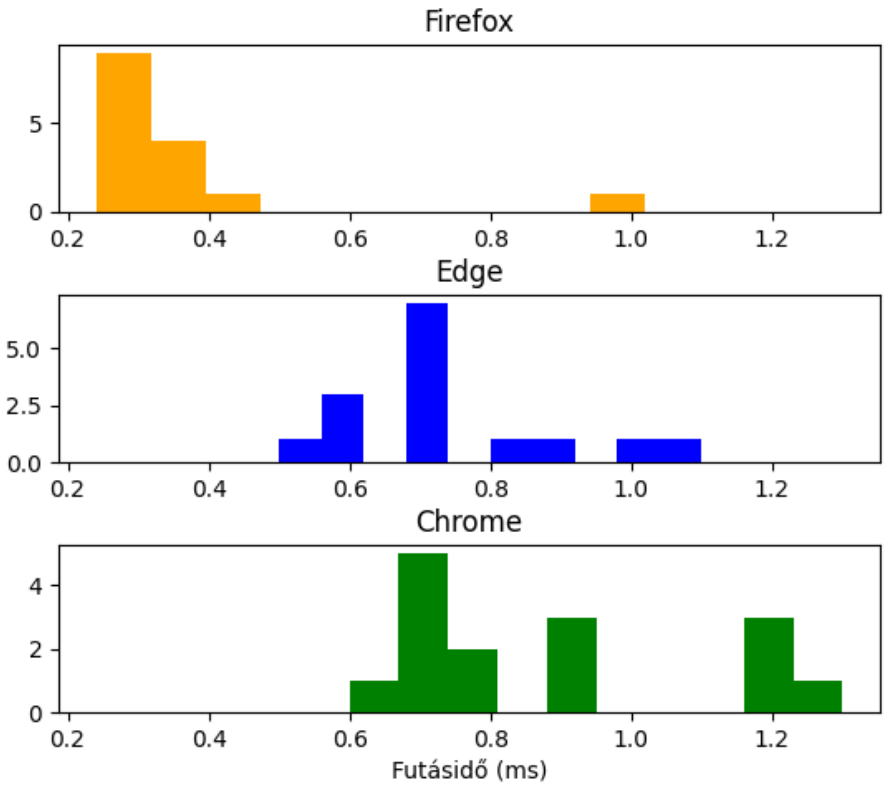
A használt példa az alábbi forráskód és az eredmény a 6.1 ábra.

```
<script>
    var canvas = document.getElementById("myCanvas");
    var ctx = canvas.getContext("2d");
    const start = performance.now();
    ctx.fillRect(0, 0, 100, 100);
```

```
    ctx.lineWidth = 20;
    ctx.strokeStyle = "#FF0000";
    ctx.strokeRect(50, 50, 100, 100);
    // Draw a blue triangle
    ctx.fillStyle = "#0000FF";
    ctx.lineWidth = 5;
    ctx.beginPath();
    ctx.moveTo(200, 200);
    ctx.lineTo(300, 200);
    ctx.lineTo(300, 300);
    ctx.closePath();
    ctx.fill();
    // Draw a green circle
    ctx.fillStyle = "#00FF00";
    ctx.lineWidth = 10;
    ctx.beginPath();
    ctx.arc(300, 500, 100, 0, Math.PI * 2, true);
    ctx.closePath();
    ctx.fill();
    // Write "Hello World"
    ctx.fillStyle = "#000000";
    ctx.lineWidth = 1;
    ctx.font = "24px Helvetica";
    ctx.fillText("Hello World", 500, 200);
    const duration = performance.now() - start;
    console.log("Time took: " + duration);
</script>
```



6.1. ábra. Böngészőben futatott példa eredménye



6.2. ábra. Futásidő hisztogramja különböző böngészőkben

6.1.2. Vonalhúzó méréssel

Elkészült egy program is, ami 1280 alkalommal (a vizsgálathoz használt felbontás alapján) húz egy vonalat különböző színnel, így egy színátmenetet létrehozva. A létrehozott színátmenet a 6.3. ábrán látható. A kód ami létrehozta a színátmenetet pedig alább olvasható. Érdekes, hogy a színátmenet nem az, amire azonnal számítanánk a kódból.

```
var canvas = document.getElementById("myCanvas");
var ctx = canvas.getContext("2d");
const start = performance.now();
ctx.strokeWidth = 2;
for (var i = 1; i < 1281; i++) {
    ctx.beginPath();
    ctx.moveTo(i - 1, (i - 1) % 2 ? 0 : 720);
    ctx.strokeStyle =
        "rgb(i / 1270.0 * 255,
        i / 1270.0 * 255,
        i / 1270.0 * 255)";
    ctx.lineTo(i, i % 2 ? 0 : 720);
    ctx.stroke();
}
const duration = performance.now() - start;
console.log("Time took: " + duration);
```

6.3. ábra. Létrehozott színátmenet vonalhúzó méréssel

6.2. Mérés az általam elkészített keretrendszerben

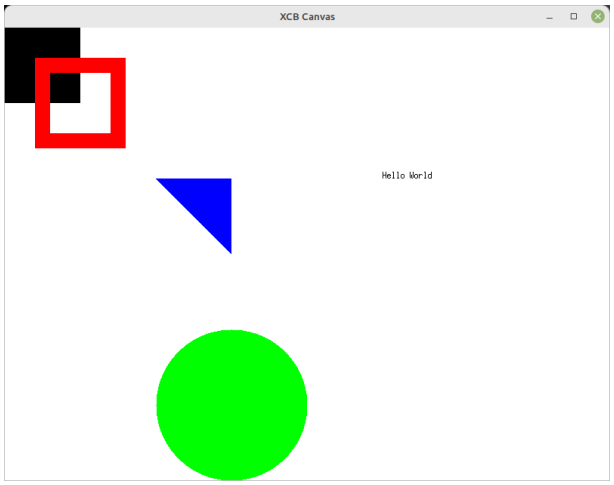
A saját példaprogramomat lefordítottam és futtatam a háttérben majd felhoztam a rendszer figyelő alkalmazását. 104 KiB memória és 0 CPU használat mutatott. A HTML és JS kód megfelelője a keretrendszeremben az alábbi forráskód és az eredménye a 6.4. ábra.

6.2.1. Demó programmal

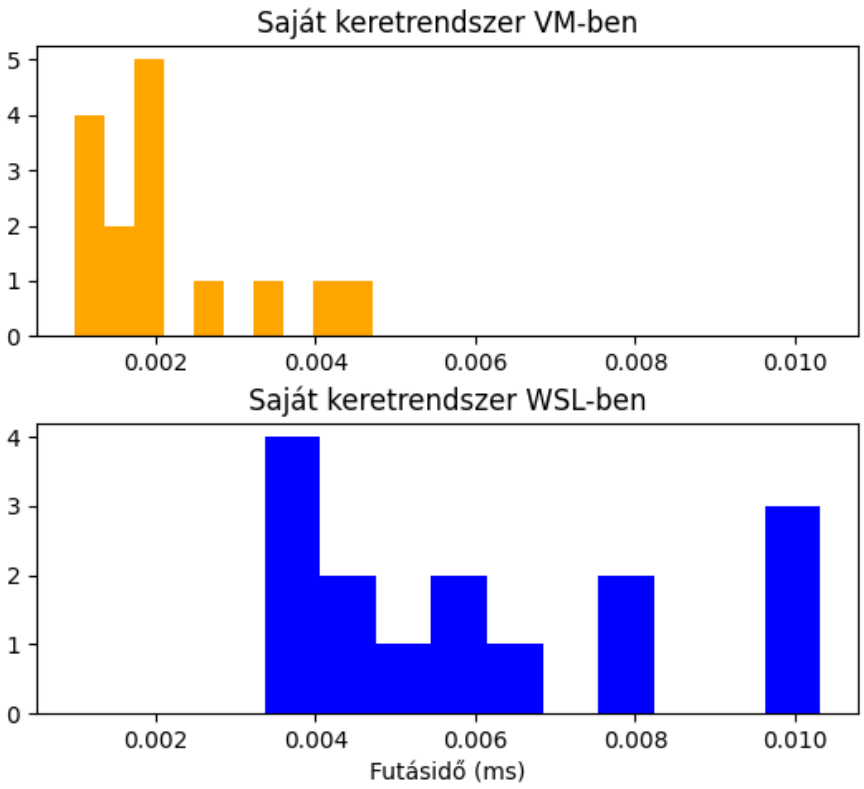
A futásidő is mérve lett a kódban hasonló módon – a VM-ben és WSL-lel is – majd hisztogramként a 6.5. ábrán ábrázolva is lett *Python* segítségével.

```
void draw(canvas_rendering_context_t* rendering_context)
{
    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start);
    canvas_fill_rectangle(rendering_context, 0, 0, 100, 100);
    canvas_set_line_width(rendering_context, 20);
    canvas_set_color(rendering_context, 255, 0, 0);
    canvas_stroke_rectangle(rendering_context, 50, 50, 100, 100);
    /* Draw a blue triangle */
    canvas_set_color(rendering_context, 0, 0, 255);
    canvas_set_line_width(rendering_context, 5);
    canvas_begin_path(rendering_context);
    canvas_move_to(rendering_context, 200, 200);
    canvas_line_to(rendering_context, 300, 200);
    canvas_line_to(rendering_context, 300, 300);
    canvas_close_path(rendering_context);
    canvas_fill(rendering_context);
    /* Draw a green circle */
    canvas_set_color(rendering_context, 0, 255, 0);
    canvas_set_line_width(rendering_context, 10);
    canvas_begin_path(rendering_context);
    canvas_arc(rendering_context, 300, 500, 100, 0, 360, 0);
    canvas_close_path(rendering_context);
    canvas_fill(rendering_context);
    /* Write Hello World */
```

```
    canvas_set_color(rendering_context, 0, 0, 0);
    canvas_set_line_width(rendering_context, 1);
    canvas_font2(rendering_context, "helvetica", 24);
    canvas_fill_text(rendering_context, "Hello World", 500, 200);
    clock_gettime(CLOCK_MONOTONIC, &end);
    printf("Time taken: %f\n", (end.tv_sec - start.tv_sec) +
    (end.tv_nsec - start.tv_nsec) / 1000000000.0);
}
```



6.4. ábra. Példa a saját keretrendszerben



6.5. ábra. Futásidő hisztogramja a saját keretrendszerben

6.2.2. Vonalhúzó méréssel

A saját keretrendszerben is meg lett mérve az idő a vonalhúzó mérésben. A kódja hasonló módon készült el a fentihez. A keretrendszerem a böngésző esetében kapottal megegyező eredményt adott.

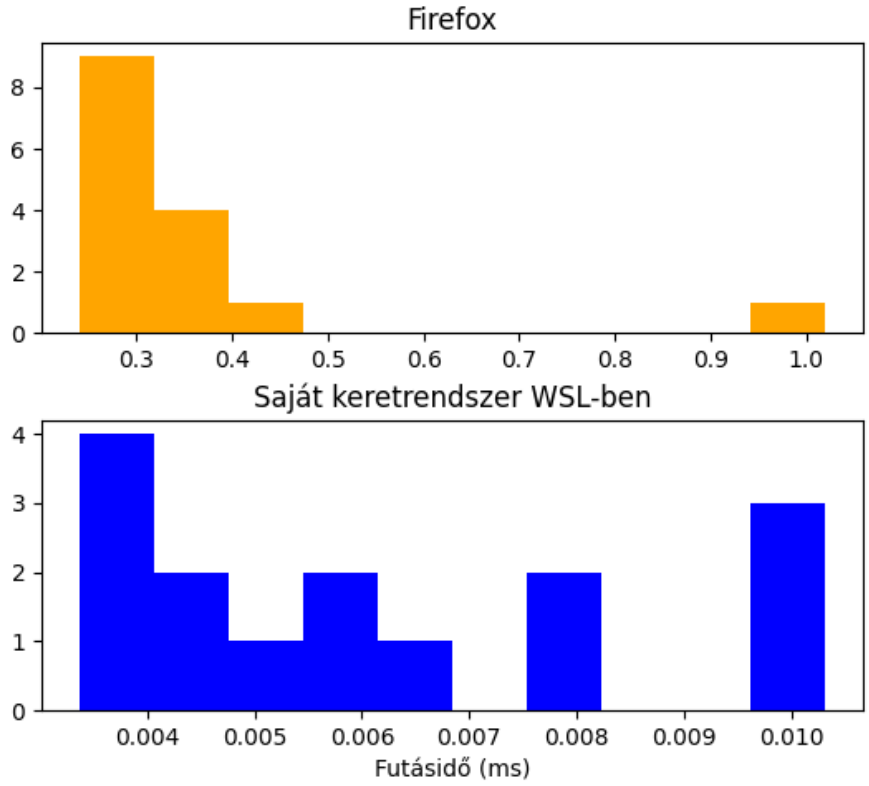
```
struct timespec start, end;
xcbcanvas_set_window_size(rendering_context->canvas, 1280, 720);
clock_gettime(CLOCK_MONOTONIC, &start);
canvas_set_line_width(rendering_context, 2);
for (int i = 1; i < 1281; i++) {
    canvas_begin_path(rendering_context);
    canvas_move_to(rendering_context,
        (i - 1), (i - 1) % 2 ? 0 : 720);
    canvas_set_color(rendering_context,
        (int)(i / 1280.0 * 255),
        (int)(i / 1280.0 * 255),
        (int)(i / 1280.0 * 255));
    canvas_line_to(rendering_context, i, i % 2 ? 0 : 720);
    canvas_stroke(rendering_context);
}
clock_gettime(CLOCK_MONOTONIC, &end);
printf("Time taken: %f\n", (end.tv_sec - start.tv_sec) +
(end.tv_nsec - start.tv_nsec) / 1000000000.0);
```

6.3. Összehasonlítás

A keretrendszerem csak a Canvas API primitívek rajzolását, Path rajzolását görbék nélkül, és a szöveg írásának minimális lehetőségét fedi le, és a memóriának csak a 20%-át használja. Ellentétben viszont, a keretrendszeremben lassabban lehet tesztelni, hogy helyes kódot írtunk, ugyanis a gépi kódra fordítás lassabb, mint egy böngészőben betölteni egy helyi weboldalt. Továbbá össze akartam hasonlítani a már létező xcb-canvas-val, de nem sikerült a használata.

6.3.1. Demó programnál

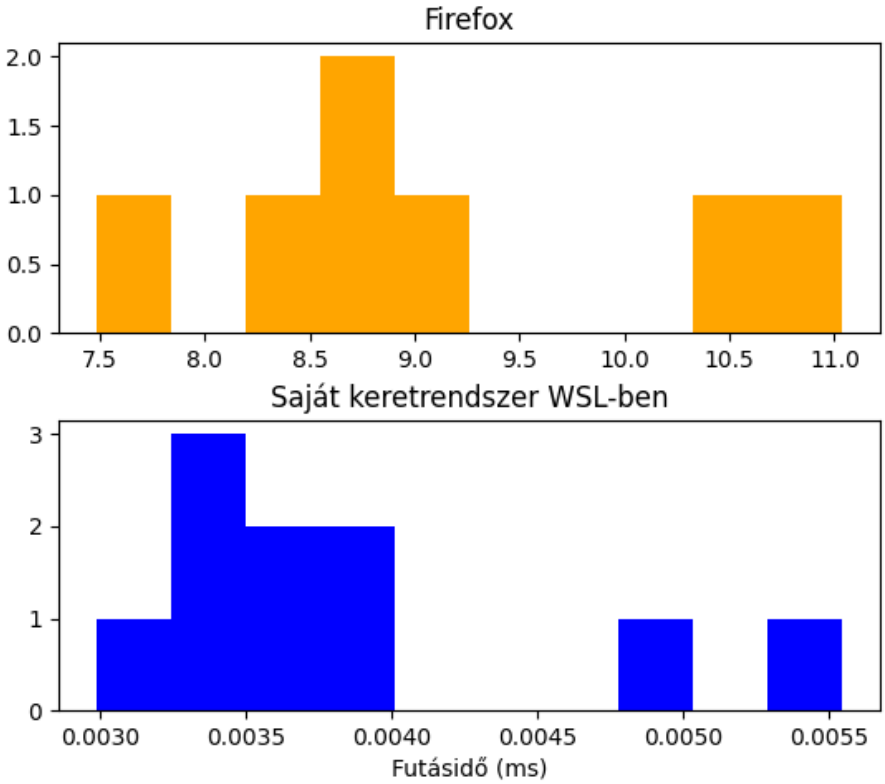
A futási időt tekintve a keretrendszerem 0.28 ms-ről 0.007 ms-re csökkent, azaz negyvenszer lett gyorsabb. Az ezt összehasonlító hisztogram látható a 6.6. ábrán.



6.6. ábra. Futásidőt összehasonlító hisztogram. Míg Firefox 0.2 ms alá nem esik, addig WSL 0.01 ms-nél csak gyorsabb szokott lenni.

6.3.2. Vonalhúzó mérésnél

A mérés ebben az esetben csak Firefoxban és WSL-ben történt tízszer. Ha ezt a mérés vesszük figyelembe akkor a WSL 2558-szor gyorsabb, mint Firefox. Ezeknek a mérésneknek a hisztogramjai a 6.7. ábrán tekinthető meg.



6.7. ábra. Futásidőt összehasonlító hisztogram a vonalhúzó mérés alkalmazásban.

7. fejezet

Összefoglalás

A dolgozat írása közben nagyon feltűnővé vált, hogy milyen nagy a Canvas API, és hogy mennyire nehéz ilyen alacsony szinten grafikusan programozni. A másik észrevétel, hogy a böngészőben futó kód az cross-platform, amit még nehezebb lenne elérni.

A dolgozat a HTML Canvas API néhány egyszerűbb rajzolási módját részletezte tervezés és implementáció szintjén egyaránt. Sor került a JavaScript-ből elérhető függvények és az XCB adta lehetőségek összevetésére. Ebből jól látszott, hogy közel sincs 1:1 kapcsolat a kétféle API funkcióit illetően.

A dolgozat megírása során elkészült egy XCB alapú keretrendszer, mely az API kialakításában hasonló, mint a HTML5 Canvas, továbbá natív kódra fordítható.

A Canvas API-nak a kb. a 3/8-a van lefedve, így bőven lenne még mit implementálni, kezdve a görbék rajzolásától, saját betűtípus használhatóságán, áttetszőség, színkeverés, és színátmeneteken keresztül, animációk és felhasználói bementtel bezárva. Ezen kívül a cross-platform megoldás és a fejlesztett program gyorsabb ellenőrzésének – például "hot-swap"-pal – elérhetővé tétele. További tesztelésre (például egységtesztek írására) szintén szükség lenne.

A minimális siker ellenére, sikernek sikert ért el a program abban, hogy kisebb erőforrásigényű legyen.

Irodalomjegyzék

[1] Max K. Agoston. *Computer Graphics and Geometric Modelling: Implementation & Algorithms*, chapter 11.4, page 401. Springer Science+Business Media, 2005. https://books.google.hu/books?id=TAYw3LEs5rgC&pg=PA401.

[2] Alan Coopersmith. The X New Developer's Guide: Xlib and XCB, 2013. https://xorg.freedesktop.org/wiki/guide/xlib-and-xcb/.

[3] craigloewen-msft. What is the Windows Subsystem for Linux?, 2021. https://docs.microsoft.com/en-us/windows/wsl/about.

[4] crloewen. Get started using VS Code with WSL, 2021. https://docs.microsoft.com/en-us/windows/wsl/tutorials/wsl-vscode.

[5] CrypticSwarm. Github - CrypticSwarm/xcb-canvas, 2011. https://github.com/CrypticSwarm/xcb-canvas.

[6] CrypticSwarm. Github - CrypticSwarm/XCBJS, 2011. https://github.com/CrypticSwarm/XCBJS.

[7] Wayland csapat. Wayland toolkits, 2021. https://wayland.freedesktop.org/toolkits.html.

[8] Helder Da Rocha. *Learn Chart. js: Create interactive visualizations for the web with chart. js 2*. Packt Publishing Ltd, 2019.

[9] Dave Gershgorn. Github and OpenAI launch a new AI tool that generates its own code. *The Verge*, June 2021. https://www.theverge.com/2021/6/29/22555777/github-openai-ai-tool-autocomplete-code.

[10] Jakob Gruber. JIT-less V8, 2019. https://v8.dev/blog/jitless.

[11] Computer Hope. What is VcSsrv?, 2019. https://www.computerhope.com/jargon/v/vcxsrv.htm.

[12] MDN közremüködők. Canvas API, 2021. https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API.

[13] MDN közremüködők. WebGL: 2D and 3D graphics for the web, 2022. https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API.

[14] XCB közremüködők. xcb/tutorial, 2014. https://xcb.freedesktop.org/tutorial/.

[15] mattwojo. Get started with the Windows Subsystem for Linux, 2020. https://docs.microsoft.com/en-us/learn/modules/get-started-with-windows-subsystem-for-linux/.

[16] Michael Safyan. Object-Oriented Programming (OOP) in C, 2017. https://www.codementor.io/@michaelsafyan/object-oriented-programming-in-c-du1081gw2.

[17] Andreas Schneider. cmocka - unit testing framework, 2020. https://cmocka.org/.

[18] whme. How to set up working X11 forwarding on WSL2, 2022. https://stackoverflow.com/a/61110604.

[19] Scott Chacon és egyéb közreműködők. Git, 2021. https://git-scm.com/.

CD Használati útmutató

A fordításhoz Ubuntun a a szükséges csomagok letöltéséhez az alábbi parancsot kell kiadni: `sudo apt install build-essential libx11-xcb-dev pkg-config` . Ezután `cmocka`-t kell telepíteni, aminek a módja elolvasható 5.3. szakaszban. Végül a példa program elkészítéséhez a `make example` parancs kiadása után a `example` programot kell futattni a megjelent `build` jegyzékben.

Amennyiben Windows 10 vagy Windows 11-en vagyunk WSL-t telepítenünk kell, majd WSL-ben Ubuntu-t telepíteni és a fenti utasításokat követni[15]. Továbbá Windows 10-en grafikus program megjelenítéséhez VcXsrv telepítéséhez és felállításához is szükség van[18]. Windows 11-en tudtommal már lehet grafikus Linux programokat futtatni WSL segítségével, de nem teszteltem.

A CD-n található jegyzékek:

- .vscode/ – a VsCode fejlesztő környezet beállításait tartalmazó jegyzék
- include/ – a header fájlokat tartalmazó jegyzék
- src/ – a forráskódot tartalmazó jegyzék
- tests/ – az egységteszteket tartalmazó jegyzék
- build/ – ebben a jegyzékbe lesz létrehozva a futtatható tartomány és a keretrendszer függvénykönyvtár
- thesis/ – ebben a jegyzék található a szakdolgozat LaTeX kódja és PDF fájlja.