Střední průmyslová škola Třebíč

Maturitní práce

RUBIKOVA KOSTKA

Profilová část maturitní zkoušky

Studijní obor: Informační technologie

Třída: ITA4

Školní rok: 2023/2024 Lukáš Kurtin

Zadání práce

Střední průmyslová škola Třebíč
Manželů Curieových 734, 674 01 Třebíč

Zadání ročníkové práce

Obor studia: **18-20-M/01 Informační technologie**

Celé jméno studenta:	**Lukáš Kurtin**		
Třída:	**ITA4**	Školní rok:	**2023/2024**
Číslo tématu:	**23**		
Název tématu:	**Rubikova kostka**		
Rozsah práce:	**15 - 25 stránek textu**		

Specifické úkoly, které tato práce řeší:

Napište program pro simulaci skládání Rubikovy kostky. Zvolte vhodnou interní reprezentaci kostky a vizualizaci stran kostky. Po základní vizualizaci bude použit 2D pohled na rozvinutý povrch kostky, kromě toho můžete realizovat i 3D obraz. Zvolte vhodný způsob zadávání tahů kostkou. Tahy budou zaznamenány s možností uložit a pokračovat v již započatém skládání. Realizujte funkce "undo" a "redo". Využijte vývojové prostředí Visual Studio, pro správu verzí pak platfomu GitHub.

Termín odevzdání:	**27. března 2024, 23.00**
Vedoucí projektu:	**Ing. Ladislav Havlát**
Oponent:	**Ing. Marek Hrozníček**
Schválil:	**Ing. Petra Hrbáčková, ředitelka školy**

ABSTRAKT

Projekt je simulační prostředí Rubikovy kostky. Obsahuje základní funkce, jako jsou otáčení stran a provádění algoritmů podle celosvětové notace tahů. Kostka je vykreslena ve třídimenzionálním prostoru pomocí projekčních matic a lze ji otáčet okolo vertikální osy. Aplikace poskytuje i základní vysvětlení termínů Rubikovy kostky a zacházení s programem.

KLÍČOVÁ SLOVA

Maturitní práce, Rubikova kostka, C#, Windows Forms, .NET, 3D

ABSTRACT

This project is a simulational environment of the Rubik’s cube. It contains basic functions, like turning the sides and doing algorithms by the worldwide used notation of turns. The cube is drawn in three-dimensional space by using projection matrices and can be rotated around the vertical axis. The application also includes basic instructions on how to operate the program and teaches basic terms of the Rubik’s cube.

KEYWORDS

Graduation thesis, Rubik‘s cube, C#, Windows Forms, .NET, 3D

PODĚKOVÁNÍ

Děkuji Ing. Ladislavu Havlátovi za odborný pohled při procesu vytváření projektu a paní Natálii Pistovčákové, která mi pomohla se správností dokumentace.

V Třebíči dne 27. března 2024 podpis autora

PROHLÁŠENÍ

Prohlašuji, že jsem tuto práci vypracoval/a samostatně a uvedl/a v ní všechny prameny, literaturu a ostatní zdroje, které jsem použil/a.

V Třebíči dne 27. března 2024 podpis autora

Obsah

Úvod

Jako moji maturitní ročníkovou práci jsem chtěl vypracovat projekt nad moje zkušenosti, u kterého bych strávil více času než na jakémkoliv jiném projektu, který jsem v průběhu školních let dělal. Ideální je proto projekt, který je náročný nejen z hlediska programátorského, ale i grafického.

Vybral jsem si proto simulaci Rubikovy kostky, které jsou mi již léta blízké a vždy mě fascinovala jejich funkčnost. Pro uskutečnění bude potřeba projít několik způsobů, jak naprogramovat základní strukturu a funkce kostky.

Graficky bude kostka zobrazena v krychlové síti a ve 3D prostoru s možností otáčení pohledu. K tomu bude zapotřebí nastudovat projekční matice, které umožňují překlad mezi 3D prostorem a 2D obrazovkou.

Jako vývojové prostředí použiji Microsoft Visual studio a jeho balíček Windows Forms .NET framework, ve kterém se programuje v jazyku C# a který ulehčuje vykreslování na obrazovku a zadávání uživatelského vstupu. Pro řízení projektu a správu verzí využiji platformu GitHub, více popsanou v teoretické části.

Konečný stav projektu si představuji jako intuitivní aplikaci, která bude obsahovat samotné funkce Rubikovy kostky, ale i teoretický návod na její složení pomocí nejpoužívanější metody CFOP (pojmenované podle čtyř kroků, které aplikuje). Také bych chtěl implementovat optimalizovaný algoritmus pro automatické složení kostky, popřípadě ho využít jako doplněk naučné stránky.

1 Teoretická část

Rubikova kostka je považována za náročný hlavolam. Jakmile se však člověk snaží přijít na to, jak vlastně funguje její mechanismus, zjistí, že to není tak složité.

Kostek je několik variant: 2x2, 3x3, 4x4, 5x5 a mnohem více. Čísla určují, kolik jedna řada z rohu do rohu obsahuje kostiček. Tato práce se zabývá tou původní a nejznámější verzí: 3x3.

1.1 Rubikova kostka

Velký počet lidí si o Rubikově kostce myslí, že tahy zaměňují barvy, což není plně pravda. Skutečně se totiž neotáčí s barvami, ale s jednotlivými kostičkami. Kostka jich má celkově 22, 8 rohů se třemi barvami a 12 hran se dvěma. Jednobarevné středy se do nich nezapočítávají, neboť jejich poloha se nikdy nemění. Jsou totiž středem osy každého otočení.

Každá kostička má distinktivní kombinaci barev, které za normálních okolností nemohou mít duplikát. Může se měnit pouze jejich poloha a tím i její otočení. Silou otáčet jednotlivé kostičky bez použití algoritmu učiní kostku neskladatelnou, dokud se neotočí zpátky.

1.2 Algoritmizace

Jedním z prvků finální verze projektu by měl být algoritmus pro metodické složení kostky. Pokud se kostkou bude náhodně otáčet, složení by zabralo průměrně $43{,}252\times10^{15}$ tahů. Existuje proto několik algoritmických sad, ze kterých se vytváří skládací metody. Nejznámějšími a nejpoužívanějšími z nich jsou CFOP a ROUX. Kvůli svojí znalosti o této metodice bude použita metoda CFOP.

Jako zdroj algoritmů se využije veřejná databáze SpeedCubeDB. [1]

1.2.1 Metodika CFOP

Název označuje zkratky postupů, podle jakých se Rubikova kostka skládá. Jsou to: Cross (kříž), tedy jako první krok se poskládá kříž na jedné straně (barvě) kostky. Dále se poskládají první 2 vrstvy okolo vytvořeného kříže. Odtud název F2L (first two layers, první dvě vrstvy). Tyto kroky se většinou provádí podle intuitivního sledování a manipulování kostky, jejich algoritmy se téměř nepoužívají, pouze na nejvyšší

kompetitivní úrovni. Zbývající dva kroky jsou naopak pouze algoritmické. Pro sadu OLL (Orientating the Last Layer, orientace poslední vrstvy) je celkem 57 algoritmů. Jejich cílem je pootáčet kostičky tak, aby celá vrchní strana byla kompletní, avšak kostka neposkládaná. O to se stará poslední krok, PLL (Permutating the Last Layer, permutace poslední vrstvy), který přemístí všechny zbylé kostičky. Výsledkem je poskládaná kostka. [2]

1.2.2 Značení tahů

Algoritmy se zapisují podle celosvětové notace, kde každý tah má své písmeno z anglického názvu. Základní jsou R, L, U, D, F, B (pravá, levá, horní, spodní, přední, zadní). Otáčí se po směru hodinových ručiček. Každý tah se také řídí pravidlem, že pokud se za písmenem nachází apostrof nazývaný „prime“, tah se provede v opačném směru a pokud číslice 2, provede se posun 2krát. Rozšiřující jsou pak široké tahy w, rotace pohledu na kostku x, y, z a průřezové M, E, S. [3]

1.3 Programová struktura kostky

Velmi důležitý faktor je, jak vytvořit algoritmus, který bude kostkou otáčet. Na základě toho se vytvoří celý systém a struktura kostky. Po troše hledání jsem došel ke dvěma možným způsobům. Kvůli převažujícím výhodám bude využit způsob 2.

1.3.1 Způsob 1

Síla tohoto způsobu je nízká náročnost programovatelnosti.

Kostka by se skládala z 6 polí o dvou dimenzích neboli 3x3 matice. Každý prvek by znamenal jedno barevné pole na kostce.

Spočívá v tom, že při zadání příkazu na otočení strany se každá barva na každé ovlivněné kostičce natvrdo posune na její požadovanou pozici. Nevýhodou je neaplikovatelnost v chytrých algoritmech, neboť by se při nich musely samostatně kontrolovat všechny barvy. To by začinilo extrémní časovou náročnost programu a zbytečně dlouhý a nepřehledný kód.

1.3.2 Způsob 2

Další varianta je těžší na programování, ale je kompatibilnější s dynamickým vývojem.

Namísto šesti matic se využijí pouze 3, jedna na každou vrstvu kostky. Jejich objekty by byly třídy kostičky, které by obsahovaly vlastnosti otočení a pevné očíslování základní pozice. Kostičky by měnily svoji aktuální polohu po směru hodinových ručiček relativně k ostatním. Pro získávání stavů kostky pro užití vhodného skládacího algoritmu by se kontrolovala pouze rotace a očíslování kostičky (vztahy mezi větším počtem kostiček).

1.4 Grafické vykreslení

Pro vykreslení kostky se použijí 2 způsoby: síťové zobrazení kostky pro 2D a zobrazení ve 3D s možností rotace.

1.4.1 2D

Zobrazení celé kostky ve dvou dimenzích je možné pouze oddělením všech jejích stran tak, aby se mohla rozložit na plochu. Takové zobrazení je sice lehké vytvořit, je však velmi nepřehledné.

1.4.2 3D

Vykreslení ve třech dimenzích je optimální pro lidské pochopení. V takovém prostředí žijeme, tudíž je to pro nás přírodní. Dosáhnutí toho je však náročnější z programové stránky.

Je nutné vytvořit digitální prostředí pro objekty, které se budou pohybovat po třech světových osách (x, y, z). Obrazovka počítače je dvoudimenzionální, tudíž je nemožné prostor bez úprav zobrazit. Tento problém řeší ortodoxní projekce.

1.4.3 Ortodoxní projekce

Prostor je zrealizován pomocí třídimenzionálních vektorů. Ty se ale nedají vykreslit na 2D plochu, neměly by hloubku a nic by z nich nešlo poznat. Proto se jednotlivé vektory vynásobí tzv. projekčními maticemi a výsledek je 2D bod na obrazovce s vypočítanými faktory pohledu, jako je vzdálenost od kamery, zorné pole rotace atd.[4]

1.5 Použité technologie

Při vývoji se používalo několik nástrojů a technik pro vývoj software a verzování aplikace.

1.5.1 GitHub

GitHub je online služba, která poskytuje ukládání projektů a jejich přenos mezi zařízeními. Dělá se tak pomocí repositářů neboli prostorů na internetu, kam se ukládají data.[5]

1.5.2 GitHub Desktop

Aplikace, která ulehčí práci s platformou GitHub. Detekuje všechny změny v souborech daného staženého repositáře na zařízení a umožňuje jejich nahrání nebo stažení pomocí jednoho tlačítka.[6]

1.5.3 .NET

.NET (nazýván DOTNET) je spolehlivá aplikační platforma, ve které se programuje většinou v jazyku C#. Je dostupná na téměř všech operačních systémech. Poskytuje tzv. Full-stack neboli vývoj na straně klienta i serveru. Runtime, knihovny a jazyky jsou základem .NET struktury.[7]

1.5.4 Windows Forms .NET Framework

Tento framework používá programovací jazyk C#. Ulehčuje grafickou stránku aplikací přes svoje komponenty UI (uživatelské rozhraní) a formulářový základ. Lehce se implementuje s programem pomocí událostí pro jednotlivé akce na komponentách formuláře. Také poskytuje nástroje pro kreslení vlastních tvarů do aplikace. [8]

Lze kreslit na samotný formulář, ale tato metoda je velice neefektivní pro výkonnostní prostředky počítače. Využije se pro to komponentu PictureBox, která je mnohem lépe optimalizovaná pro vykreslování.

1.5.5 C#

C# je programovací jazyk na vysoké úrovni, odvozený z jazyka C. Struktura programu je převážně objektově orientovaná, pro většinu akcí se tedy využívají třídy a struktury. Funkční programový způsob znamená, že program operuje ve funkcích, které se dají použít jako jakákoliv jiná entita.[9]

2 Praktická část

Popíše se zde celý program, jeho hlavní metody a algoritmy. Začne se od nejdůležitějších tříd a postupně se bude procházet dál podle toho, jak postupuje program.

2.1 Třídy 3D prostoru

Pro správnou funkci kostky je potřeba vytvořit 3D prostor pro její vykreslení. Skládá se třemi třídami, každá z nich posouvá prostor o dimenzi výše.

Při jejich vývoji jsem narazil na jeden zásadní problém, a to je kopírování objektů. Pokud se objekt přiřadí do nového pouze pomocí ‚=' jako pro normální proměnnou, zkopíruje se odkaz na daný objekt, takže nyní 2 kostičky měly identickou polohu a všechny ostatní vlastnosti.

Zprvu bylo implementováno řešení přes dědění třídy ICloneEnumerable, neboli přidání klonovací metody, ale objekty stále kopírovaly odkazy. Žádaný výsledek dosáhla nová deklarace tříd, která převzala jako jediný prvek svoji vlastní třídu a kopírovala vlastnost po vlastnosti do nového objektu.

2.1.1 Vector3

Nejnižším možným prvkem ve všech dimenzionálních prostorách je bod neboli vektor. Tato třída obsahuje tři hodnoty pro posun po osách X,Y a Z. Spojením několika vektorů se dají vytvořit složitější tvary. Většinou se však používají pouze trojúhelníky nebo čtverce, ze kterých se dané tvary skládají.

Dalšími vlastnostmi třídy Vector3 jsou výpomocné proměnné, které určují, jak má vektor měnit svoji pozici při otáčení kostky. Všechny tři vlastnosti jsou vlastní třídy Vector3. Každý vektor kostky má daná pravidla pro každou osu zvlášť.

LengthFrom0 – Vzdálenost vektoru od středu kostky;

Displacement – Udává úhel (v radiánech), který vektor svírá s danou osou;

AnimState – Ukládá změněnou pozici neboli otočení okolo osy

Metoda CalcDisplacement() vypočítává délku a úhel od nuly. Nelze je na pevno nastavit, protože po každém otočení se hodnoty mění a je potřeba je přepočítat.

2.1.2 Square

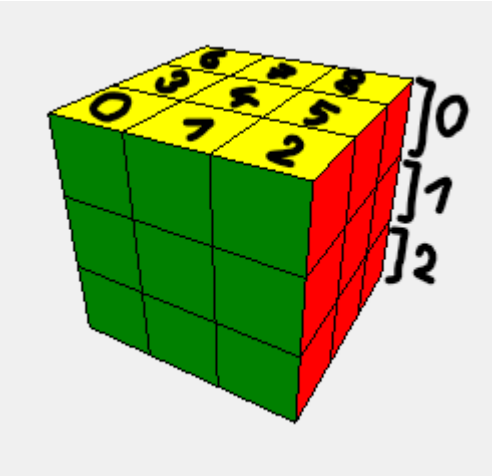
Ze čtyř vektorů se skládá jeden čtverec. Jediná nová vlastnost je barva čtverce, která se nastavuje manuálně při vytváření kostiček.

2.1.3 Cube

Přidáním třetí dimenze čtverci vzniká kostka – požadovaný tvar. Konkrétněji se jedná o kostičku, ze kterých se sjednocuje celá Rubikova kostka. Obsahuje pole šesti čtverců, které se dají nastavit osmi vektory a šesti barvami při vytváření objektu. Vlastnost cubeIndex udává správnou pozici kostičky. Její hodnoty jsou unikátní čísla 0 až 26, pro každou vrstvu 9. Využívá se především pro algoritmické skládání. Nezbytnou vlastností je rotateAxis neboli osa, po které se mají kostičky otáčet. Nastavuje se manuálně a pouze pro středové kostičky, které mají pouze jednu venkovní (nečernou) stranu. Pokud se s kostkou otočí tak, aby se změnila přední strana, tyto hodnoty se zkopírují společně se zbytkem vlastností a zajistí se tak správná rotace pro každou stranu. Bez tohoto otáčení bylo náročné s Rubikovou kostkou pracovat i přes to, že je na ni člověk zvyklý. Pro laika by uživatelský zážitek byl velice nepříjemný.

2.2 Struktura kostky

Fyzická Rubikova kostka se skládá z 22 kostiček, které se mohou pohybovat, 12 rohů a 4 hrany. Poloha středů se nikdy nemění. Programově je nutné tyto kostičky zahrnout, takže jich bude celkově 27. Jsou 3 vertikální vrstvy a každá z nich má 9 kostiček.



Obrázek 1: Grafické znázornění pole kostky

Kostka je strukturována jako dvoudimenzionální pole kostiček. První index je 0–8 pro vodorovné kostičky, druhý 0-2 pro vybrání svislé vrstvy.

2.2.1 Deklarace kostky

Při deklaraci je pro každou z 27 kostiček zadáno 8 vektorů, 6 barev a její číselná pozice. Celá kostka má rozměry -1.5 až 1.5 ve všech směrech. Kdyby bylo potřeba, program se dá lehce přepsat na zadání délky strany a vypočítat jednotlivé kostičky.

```
public void GenerateCubes()
{
    {
        cubes[0, 0] = new Cube(new Vector3(-1.5, 0.5, 0.5), new Vector3(-1.5, 1.5, 0.5),
                               new Vector3(-0.5, 1.5, 0.5), new Vector3(-0.5, 0.5, 0.5),
                               new Vector3(-1.5, 0.5, 1.5), new Vector3(-1.5, 1.5, 1.5),
                               new Vector3(-0.5, 1.5, 1.5), new Vector3(-0.5, 0.5, 1.5),
                               Color.Black, Color.Orange, Color.Green,
                               Color.Black, Color.Black, Color.Yellow,
                               0);
```

Obrázek 2: Deklarace kostičky

Zde jde vidět deklarace jedné kostičky a ručně zadané hodnoty. Představuje kostičku vepředu, nahoře a vlevo, viz. Obrázek 1, a barevně je žluto-zeleno-oranžová, se třemi vnitřními černými stranami.

2.3 Vykreslování

Vše se vykresluje v komponentě PictureBox. Má vlastní událost pro vykreslování s argumentem PaintEventArgs. Tento argument obsahuje třídu Graphics, skrze kterou se na komponentu kreslí. Obsahuje několik metod pro vykreslení mnoha tvarů.

```
public void DrawSquare(Graphics g, Square s)
{
    Brush b = GetBrush(s.color);
    g.FillPolygon(b, new PointF[]{
                                        new PointF((float)s.vectors[0].X, (float)s.vectors[0].Y),
                                        new PointF((float)s.vectors[1].X, (float)s.vectors[1].Y),
                                        new PointF((float)s.vectors[2].X, (float)s.vectors[2].Y),
                                        new PointF((float)s.vectors[3].X, (float)s.vectors[3].Y)});
    g.DrawPolygon(new Pen(Brushes.Black, 1), new PointF[]{
                                        new PointF((float)s.vectors[0].X, (float)s.vectors[0].Y),
                                        new PointF((float)s.vectors[1].X, (float)s.vectors[1].Y),
                                        new PointF((float)s.vectors[2].X, (float)s.vectors[2].Y),
                                        new PointF((float)s.vectors[3].X, (float)s.vectors[3].Y)});
}
```

Obrázek 3: Vykreslení čtverců

První metoda nakreslí vyplněný čtyřúhelník. Jako barvu se musí zadat třída Brush, která naštěstí obsahuje metodu pro přejetí z barvy (třída Color).

Přes vyplněný polygon se následně překreslí metodou DrawPolygon černý obrys, který vizuálně odděluje kostičky od sebe.

2.3.1 Projekční matice

Překlad prostoru z 3D na 2D, aby se mohla kostka správně vykreslit, řeší projekční matice pro ortodoxní pohled.

```
public void RenderMatrix()
{
    float znear = 0.1f;
    float zfar = 1000;
    float fov = 90;
    float aspectRatio = pictureBox1.Height / pictureBox1.Width;
    if (pictureBox1.Height < pictureBox1.Width)
        aspectRatio = pictureBox1.Width / pictureBox1.Height;

    double fovRad = Math.Tan(fov * 0.5 / 180 * Math.PI);
    projectionMatrix[0, 0] = aspectRatio  / fovRad;
    projectionMatrix[1, 1] = fovRad;
    projectionMatrix[2, 2] = zfar / (zfar - znear);
    projectionMatrix[3, 2] = (-zfar * znear) / (zfar - znear);
    projectionMatrix[2, 3] = 1;
```

Obrázek 4: Ortodoxní projekční matice

Na obrázku je vypočítaní hodnot matice. Jednotlivé hodnoty určují, jak se má vykreslovat prostor.

```
public Vector3 MultiplyMatrixVector(Vector3 input, double[,] m)
{
    Vector3 output = new Vector3();
    output.X = input.X * m[0, 0] + input.Y * m[1,0] + input.Z * m[2,0] + m[3,0];
    output.Y = input.X * m[0, 1] + input.Y * m[1, 1] + input.Z * m[2, 1] + m[3, 1];
    output.Z = input.X * m[0, 2] + input.Y * m[1, 2] + input.Z * m[2, 2] + m[3, 2];
    double w = input.X * m[0, 3] + input.Y * m[1, 3] + input.Z * m[2, 3] + m[3, 3];
    if(w != 0)
    {
        output.X /= w;
        output.Y /= w;
        output.Z /= w;
    }
    return output;
}
```

Obrázek 5: Vynásobení vektoru maticí

Takto vypadá program na násobení vektoru maticí, výsledkem této operace je odpovídající bod ve 2D. Protože je matice 4x4 a vektor má pouze 3 hodnoty, vzniká navíc bod označený w, který určuje vzdálenost objektu od kamery.

Podobně vypadají 2 další matice, které počítají rotaci prostoru po osách X a Y. Vektory se tedy vynásobí všemi třemi maticemi postupně a poté se vykreslí na obrazovku.

2.3.2 Pořadí vykreslení

Metoda Middle() z třídy Čtverec vypočítá ze svých vektorů střed, podle kterého se řadí všechny čtverce do listu a popořadě se vykreslují. První prvek v listu se vykreslí jako první, ale brzy ho překreslí nový, který zaobírá na obrazovce stejné místo, je však blíž ke kameře. Výsledkem je tedy požadovaný pohled na kostku s pouze nejmenšími záblesky nevhodného překreslení.

2.4 Tahy

Nezbytnou funkcí programu je otáčení stran kostky. Postup provádění tahů je kvůli struktuře kostiček obrácený od očekávaného postupu. Při příkazu tahu se znaky postupně zadají do fronty, kterou vytváří proměnná List<char>. V události komponenty Timer, která se provede každých 10 ms, se fronta kontroluje a nevhodné znaky vypustí. Pak se spustí animace, na jejíž konci se kostka vrátí do stejného tvaru, tah se vymaže z fronty a provede se programový posun.

2.4.1 Vizuální změna polohy

Tahy se animují pomocí goniometrických funkcí sinu a cosinu, kterými se mění vlastnosti všech ovlivněných vektorů tak, aby po x krocích dosáhly nové pozice.

```
switch (animateTurn[0])
{
    case 'U':
        for (int i = 0; i < 9; i++)
        {
            foreach (Square s in cubes[i, 0].squares)
            {
                foreach (Vector3 v in s.vectors)
                {
                    v.animState.Y += prime;
                    v.X = Math.Sin(-v.animState.Y * Math.PI / 2 / turnStep + v.displacement.Y) * v.lengthFrom0.Y;
                    v.Z = Math.Cos(-v.animState.Y * Math.PI / 2 / turnStep + v.displacement.Y) * v.lengthFrom0.Y;
                }
            }
        }
        break;
```

Obrázek 6: Animace tahu

Na obrázků je výstřižek pro animaci tahu U. Cykly for se vyberou kostičky, které jsou tahem ovlivněny a chceme jejich pozici změnit. Dvěma foreach se z kostiček vybere každý vektor všech čtverců. Hodnota prime se nastaví před switchem a udává, zda se má otáčet pro nebo proti směru hodinových ručiček. Vypočítá se upravenou hodnotou nová pozice v prostoru. Kostičky se otáčí okolo osy Y, takže měníme pozici X a Z pomocí hodnot vlastností vektoru pro osu Y.

Tahy F,B,R,L potřebují čtyři takové výpočty, kdy se provede pouze jeden určený tím, na jakou stranu je otočený pohled. Pokud s kamerou posunu o 45° na jakoukoliv stranu neboli na hranu kostky, tato orientace se změní a provede se tah x.

Počet kroků animace je ovlivněn rychlostí, která se upravuje v označené číselné komponentě (Počet kroků = 101 – hodnota).

2.4.2 Programový pohyb

Druhý způsob z teoretické části sice pracuje na stejném principu jako ten první, kde se kostičky kopírují na své nové pozice bývalých kostiček, avšak je účinnější a kratší. Také se skrze něj dají snadno a dynamicky hledat pozice kostiček.

Všechny tahy se provádí ve funkci Turn(string), jejíž parametrem je požadovaný pohyb. Může obsahovat pouze písmeno tahu nebo i apostrof či 2 navíc. Podle toho program určí, jestli se má provést tah po či proti směru hodinových ručiček. Pokud obsahuje 2 za písmenem, přidá právě provedený tah na začátek fronty tahů a provede se ihned znovu.

```
case 'U':
    buffCorner = new Cube(cubes[0, 0]);
    cubes[0, 0] = new Cube(cubes[2, 0]);
    cubes[2, 0] = new Cube(cubes[8, 0]);
    cubes[8, 0] = new Cube(cubes[6, 0]);
    cubes[6, 0] = new Cube(buffCorner);
    buffEdge = new Cube(cubes[1, 0]);
    cubes[1, 0] = new Cube(cubes[5, 0]);
    cubes[5, 0] = new Cube(cubes[7, 0]);
    cubes[7, 0] = new Cube(cubes[3, 0]);
    cubes[3, 0] = new Cube(buffEdge);
    cubes[4, 0] = new Cube(cubes[4, 0]);
    break;
```

Obrázek 7: Programový pohyb kostiček při tahu

Ukázka tahu U. Prvně se přesunou všechny rohy o jednu pozici dopředu nebo dozadu (prime tahy jsou v jiném switchi) a pak hrany. Kvůli neznámé chybě se musí kopírovat i střed, který se nikam nehýbe.

2.4.3 Historie

Po provedení se tah zapíše do proměnné pro historii všech tahů. Vzhledem k tomu, že skládání je většinou dlouhý proces, historie by se celá vypsaná na formulář nevlezla. Proto se vypisuje pouze prvních 30 tahů. Nad výpisem jsou tlačítka, jedno slouží pro otevření poznámkového okna MessageBox, kde se historie vypíše celá. Po kliknutí na druhé se historie zkopíruje do schránky. To se dá využít například pro ukládání stavu kostky, kdyby uživatel potřeboval vypnout aplikaci, ale nechtěl ztratit pokrok.

2.5 Automatické poskládání

V projektu je důležité, aby se kostka mohla vrátit na složenou pozici. Uživatel může chtít experimentovat ze stavu poskládané kostky. To zajišťují skládací algoritmy.

2.5.1 Vrácení všech tahů

Využívá se proměnná historieTahu, která zapisuje veškerou akci na kostce. Algoritmus prochází tah po tahu od nejnovějšího po nejstarší a uzná vhodnou reverzi tahu. Například z jednoduchých tahů R2UL‘ převede řetězec na LU’R2 a použije ho v metodě Algorithm(), která překládá kostkové algoritmy na tahy.

2.5.2 Algoritmická metoda CFOP

Začalo se zezadu, tedy krokem PLL. Pozice horní vrstvy se uspořádaly do jednoho řetězce, podle kterého se následně hledaly algoritmy.

```
case "81052763": offsetTurn = "U"; output += "U'"; goto case "01652783";
case "61058723": offsetTurn = "U2"; output += "U2"; goto case "01652783";
case "61250783": offsetTurn = "U'"; output += "U"; goto case "01652783";
case "01652783":
    output += Form1.PLL[0];//Aa
    found = true;
    break;
```

Obrázek 8: Detekce algoritmu

Na obrázku lze vidět formát složeného řetězce. Lze na něm vidět, že 4 různé stavy odkazují na stejný algoritmus, v tomto případě označený „Aa“. Jeden PLL algoritmus může na kostce existovat v 16 různých stavech a těchto algoritmů je 21. Další částí je OLL, kde se namísto pozic kostiček formátuje jejich rotace. Tyto algoritmy už mají pouze 4 stavy, ale existuje jich 57. Napsat detekce všech 78 algoritmů by zabralo značně času. Nejtěžší jsou však kroky kříž a F2L. Ty se totiž provádějí intuitivně, a to počítač bez umělé inteligence nedokáže. Je nutné napsat algoritmus, který detekuje desítky až stovky možných pozic všech kostiček a pak najde vhodný algoritmus kostky pro jejich dosazení na správnou pozici se správnou rotací.

K tomu bohužel z časových nedostatků nedošlo. Algoritmus pro PLL funguje bezchybně, avšak kvůli neúplnosti metodiky není do aplikace implementován. Vrátilo se tedy z většiny funkční poskládání pomocí obrácení všech provedených tahů a opravily zbylé chyby.

2.5.3 Reset

Pokud se cokoliv stane špatně, nastane chyba vykreslení, kostičky nezmění svoji pozici, nebo uživatel touží po instantním poskládání kostky, může se stisknout tlačítko RESET.

```
private void buttonReset_Click(object sender, EventArgs e)
{
    animateTurn.Clear();
    GenerateCubes();
    RenderMatrix();
    historieTahu = " ";
    label2.Text = "Historie:";
    turnAnim = 0;
}
```

Obrázek 9: Metoda RESET

Spustí se metody a nastaví vlastnosti stejným způsobem, jako se provádí při startu aplikace. Šetří to čas a zlepšuje uživatelský zážitek naproti manuálnímu vypnutí a spuštění aplikace.

Závěr

Za stanovený čas jsem dokázal vytvořit aplikaci, se kterou jsem spokojený. Obsahuje všechny potřebné funkce a funguje, jak má. Hned při prvním testování proti chybám nebo nežádoucím výsledkům se žádnému z účastníků nepodařilo najít jedinou závadu.

Zpětný ohlas byl také pozitivní, uživatelé neměli problém ovládání aplikace pochopit a s kostkou manipulovat.

Bohužel jsem nedokázal uskutečnit naučný režim, jak Rubikovu kostku poskládat, ani algoritmické poskládání pomocí metody CFOP. Důvodem byl nedostatek vypracovaných hodin na projektu a náročnost algoritmu pro skládání. Na rozdíl od těchto cílů jsem svůj osobní splnil velice dobře, a to zlepšení svých schopností v programování a programové grafice. Vývoj v prostředí Windows Forms je mnohem více zamířen na základy jazyka a programování celkově, než např. vývojové prostředí pro hry Unity, se kterým jsem vytvářel předešlé projekty.

Největší problémy spočívaly v animacích kostky. Upravovat všechny hodnoty výpočtů goniometrických funkcí tak, aby byly přehledně aplikovatelné a samotné výpočty vypsat, bylo náročnější, než jsem očekával. Jakmile jsem se ale vypořádal se začátkem, při dalších chybách jsem téměř ihned věděl, kde se špatný výpočet nachází a čím ho opravit.

Projekt od začátku prošel třemi různými způsoby vykreslení kostky. V první verzi bylo využito síto kostky ve 2D a natvrdo naprogramovaný 3D pohled, avšak z jednoho úhlu pohledu a bez otáčení. Jak projekt přešel na projekční matice, neviděl jsem důvod znovu vytvářet 2D síť, protože je značně nepřehledná a zabírá zbytečně mnoho místa na aplikaci.

Při vývoji projektu se struktura programu, jeho postupy a algoritmy značně měnily. V příštích projektech strávím více času přemýšlením a tzv. „brainstormingem“, abych netrávil zbytečný čas na algoritmech, které se stejně změní.

V budoucnu bych se k projektu chtěl rozhodně vrátit. Vidím v něm potenciál pro další rozvoj svých programátorských schopností, a to nejen dosáhnutím stanovených cílů.

Seznam použitých zdrojů

[1] ZUSSMAN, Gil. *3x3*. SpeedCubeDB. SpeedCubeDB. Dostupné z: https://www.speedcubedb.com/a/3x3 ,[cit. 2024-03-26]

[2] FERENC, Denes. *Rubik's Cube solution with advanced Fridrich (CFOP) method*. Ruwix. Ruwix. Dostupné z: https://ruwix.com/the-rubiks-cube/advanced-cfop-fridrich/ ,[cit. 2024-03-26]

[3] J PERM. *Rubik's Cube Move Notation*. J Perm. J Perm, 2024. Dostupné z: https://jperm.net/3x3/moves ,[cit. 2024-03-26]

[4] *Code-It-Yourself! 3D Graphics Engine Part #1 - Triangles & Projection*. YouTube, c2019. Dostupné z: https://www.youtube.com/watch?v=ih20l3pJoeU ,[cit. 2024-02-08]

[5] GITHUB, INC. *About repositories*. GitHub Docs. GitHub, c2024. Dostupné z: https://docs.github.com/en/repositories/creating-and-managing-repositories/about-repositories , [cit. 2024-03-26]

[6] GITHUB, INC. *About GitHub Desktop*. GitHub Docs. GitHub, c2024. Dostupné z: https://docs.github.com/en/desktop/overview/about-github-desktop , [cit. 2024-03-26]

[7] MICROSOFT. *Introduction to NET. Microsoft*. Learn. Microsoft, c2024. Dostupné z: https://learn.microsoft.com/en-us/dotnet/core/introduction?WT.mc_id=dotnet-35129-website ,[cit. 2024-03-26]

[8] MICROSOFT. *Průvodce pro desktop (model Windows Forms s .NET)*. Learn. Microsoft, c2024. Dostupné z: https://learn.microsoft.com/cs-cz/dotnet/desktop/winforms/overview/?view=netdesktop-8.0 ,[cit. 2024-03-26]

[9] MICROSOFT. *C# documentation*. Learn. Microsoft, c2024. Dostupné z: https://learn.microsoft.com/en-us/dotnet/csharp/?WT.mc_id=dotnet-35129-website ,[cit. 2024-03-26]

Seznam použitých symbolů a zkratek

Zkratka	Celý název
CFOP	Kříž F2L OLL PLL, metodika skládání
ROUX	Metodika skládání
F2L	První 2 vrstvy
OLL	Orientace poslední vrstvy
PLL	Permutace poslední vrstvy
C#	Programovací jazyk
2D	Dvoudimenzionální prostor
3D	Třídimenzionální prostor

Seznam obrázků