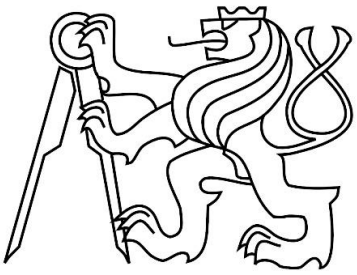


ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

FAKULTA DOPRAVNÍ

Bc. Petr Stříteský

GENERICKÝ NÁVRH PROVOZNÍCH DISPLEJŮ DRÁŽNÍCH VOZIDEL

Diplomová práce

2021

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta dopravní
děkan
Konviktská 20, 110 00 Praha 1

K620..... Ústav dopravní telematiky

ZADÁNÍ DIPLOMOVÉ PRÁCE
(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení studenta (včetně titulů):

Bc. Petr Střítеský

Kód studijního programu a studijní obor studenta:

N 3710 – IS – Inteligentní dopravní systémy

Název tématu (česky): **Generický návrh provozních displejů drážních vozidel**

Název tématu (anglicky): Generic Design of Operational Displays of Railway Vehicles

Zásady pro vypracování

Při zpracování diplomové práce se řiďte následujícími pokyny:

- Zpracujte teoretická východiska pro provozní zobrazení dražních vozidel
- S možností využití dosavadních veřejných knihoven navrhněte a vyviňte univerzální knihovny specificky určené pro vývoj zobrazení drážních vozidel ve zvoleném programovacím jazyce
- Na základě vyvinutých knihoven navrhněte či vytvořte SW zobrazení stanoveného kolejového vozidla či vozidel a ověřte jeho funkci
- Navrhněte komunikační rozhraní takového zobrazovacího zařízení a demonstrujte komunikaci na navrženém softwaru
- Vyvořte manuál k vytvořeným knihovnám pro potenciálního uživatele

Rozsah grafických prací: dle pokynů vedoucího práce

Rozsah průvodní zprávy: minimálně 55 stran textu (včetně obrázků, grafů a tabulek, které jsou součástí průvodní zprávy)

Seznam odborné literatury: Dokumentace programovacího jazyka C a jeho derivátů
Dokumentace grafických knihoven SDL a SDL 2.0
Dokumentace výrobce zvoleného zobrazovacího zařízení

Vedoucí diplomové práce: **doc. Ing. Martin Leso, Ph.D.**

Datum zadání diplomové práce: **1. června 2020**
(datum prvního zadání této práce, které musí být nejpozději 10 měsíců před datem prvního předpokládaného odevzdání této práce vyplývajícího ze standardní doby studia)

Datum odevzdání diplomové práce: **17. května 2021**
a) datum prvního předpokládaného odevzdání práce vyplývající ze standardní doby studia a z doporučeného časového plánu studia
b) v případě odkladu odevzdání práce následující datum odevzdání práce vyplývající z doporučeného časového plánu studia

Ing. Zuzana Bělinová, Ph.D.
vedoucí
Ústavu dopravní telematiky

doc. Ing. Pavel Hrubeš, Ph.D.
děkan fakulty

Potvrzuji převzetí zadání diplomové práce.

Bc. Petr Střítěský
jméno a podpis studenta

V Praze dne 1. června 2020

Poděkování

Děkuji vedoucímu diplomové práce doc. Ing. Martinovi Lesovi, Ph.D. za metodické vedení práce a poskytnutí odborných konzultací během jejího vzniku. Velké díky patří také kolektivu vývojového pracoviště VP01 AŽD Praha s.r.o. v čele s mým nadřízeným, Dr. Ing. Alešem Lieskovským, za vytvoření podmínek a prostoru pro studium při zaměstnání. Dále bych rád poděkoval své rodině a blízkým, zejména pak své přítelkyni Kateřině Neubauerové za nebetyčnou podporu, toleranci a poskytnutí příjemných podmínek pro vznik této práce.

Prohlášení

Předkládám tímto k posouzení a obhajobě diplomovou práci, zpracovanou na závěr studia na fakultě dopravní při ČVUT v Praze.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu § 60 Zákona č.121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 17. května 2021 ……...

Bc. Petr Stříteský

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta dopravní

GENERICKÝ NÁVRH PROVOZNÍCH DISPLEJŮ DRÁŽNÍCH VOZIDEL

Autor: **Bc. Petr Stříteský**

Studijní program: N3710 – Technika a technologie v dopravě a spojích

Obor: 3711T004 – Inteligentní dopravní systémy

Druh práce: Diplomová práce

Termín odevzdání: 17. května 2021

Vedoucí práce: **doc. Ing. Martin Leso, Ph.D.**

Ústav dopravní telematiky K620

Fakulta Dopravní, ČVUT v Praze

Abstrakt:

Předmětem práce je tvorba generických knihoven v programovacím jazyce C++ určených specificky pro provozní obrazovky kolejových vozidel či jejich simulátorů schopných funkce na širší množině běžně používaných operačních systémů. Vzniklé knihovny jsou následně použity k vývoji konkrétních provozních obrazovek, na nichž je jejich užití prakticky demonstrováno, a budou zpřístupněny pro další užití při vývoji obdobných aplikací včetně návodu pro vývojáře k jejich použití.

Abstract:

The major subject of this master's thesis is a development of generic C++ libraries designed specifically for use in displays of railway vehicles or their simulators capable of operation under various common operating systems. The resulting libraries are then used to develop specific screens, on which their use is practically demonstrated. These libraries will be made available for further use in the development of similar applications, including developer's guide for using them.

Klíčová slova:

Programování, vývoj, C++, SDL2.0, knihovny, železnice, kolejová vozidla, vlak, simulátor, scéna, displej, provozní obrazovka, grafické aplikace, 2D grafika, textury, geometrie, data, rychloměr, osa, diagnostika, poměrný tah

Keywords:

Programming, development, C++, SDL2.0, libraries, railroad, railway, rolling stock, train, simulator, scene, display, operational screen, graphic applications, 2D graphics, textures, geometry, data, speedometer, axis, diagnostics, relative thrust

Obsah

SEZNAM POUŽITÝCH ZKRATEK:

ACBM	Automatické cílové brzdění metra
ATO	Automatic Train Operation
AoE	ATO over ETCS
AŽD	AŽD Praha, s.r.o. (Automatizace železniční dopravy)
CCD	Hlavní ovládací displej (Central Control Display)
CD	Compact Disc
ČD	České Dráhy, a.s.
ČVUT	České vysoké učení technické v Praze
ETCS	European Train Control System
EU	Evropská Unie
IDE	Integrated Development Environment
IP	Internet Protocol
TDD	Technický a diagnostický displej
UDP	User Datagram Protocol
UIC	Mezinárodní železniční unie
USD	Americký dolar (měna)
VCU	Vehicle Control Unit
HKV	Hnací kolejové vozidlo
MHD	Městská hromadná doprava
MQTT	Message Queuing Telemetry Transport
SDL	Simple Direct-media Layer
SW	Software
TCP	Transmission Control Protocol

1 Úvod

Provozní obrazovky, které u moderních kolejových vozidel postupně nahradily starší analogové ručičkové ukazatele, kontrolky i některé tlačítkové ovládací prvky představují základní informační a diagnostické rozhraní mezi vozidlem a jeho obsluhou na stanovišti. Takové obrazovky jsou zpravidla běžným počítačovým programem s poměrně specifickým uživatelským rozhraním pro dané užití. S problematikou provozních obrazovek jsem se setkal již v rámci svého prvního bakalářského studia na Fakultě dopravní ČVUT v Praze, kdy jsem se podílel na vývoji provozní obrazovky P1 po vzoru lokomotivy Škoda 109E verze 3. V rámci své bakalářské práce na Vysoké škole logistiky, o.p.s. jsem pak navázal na tyto zkušenosti vývojem zobrazení diagnostické obrazovky systému ATO over ETCS ve spolupráci se svým zaměstnavatelem, firmou AŽD Praha. Během navazujícího magisterského studia však vyvstala nutnost věnovat se této problematice podrobněji. Konkrétně se jednalo o nutnost podílet se na vývoji hned několika provozních obrazovek simulátorů kolejových vozidel v rámci magisterského projektu. Právě tato skutečnost následně stála za základní ideou této práce, kterou se stal průzkum a realizace možností usnadnění a zavedení jisté míry standardizace v oblasti vývoje těchto aplikací. Předmětem této práce tedy jsou dílčí činnosti na vývoji generických knihoven v programovacím jazyce C++ specificky určených pro vývoj těchto provozních a diagnostických obrazovek kolejových vozidel s možností implementace na různých operačních systémech. Práce vzniká pod projektem Dopravního sálu Fakulty dopravní ČVUT v Praze s cílem využití jejích výstupů na vznikajících simulátorech kolejových vozidel i v budoucích projektech. Těmito výstupy jsou zejména funkční grafické knihovny pro vývoj provozních obrazovek kolejových vozidel a jejich využití například v rámci projektu univerzálního výcvikového simulátoru vozidla MHD[1] při vývoji konkrétních zobrazení simulátoru vozidla typu 81.71M provozovaného na trase A a B pražského metra. Na těchto reálných aplikacích bude použití i funkce navržených knihoven prakticky demonstrována a ověřena. Součástí práce dále je návrh konceptu rozhraní pro datovou komunikaci daných obrazovek, tvorba simulátoru zdrojových dat pro dynamickou demonstraci jejich funkce a vznik návodu pro programátory k využití knihoven pro vývoj podobných grafických aplikací. V první, teoretické části práce se tak po základním vhledu do problematiky obrazovek kolejových vozidel a jejich simulátorů budu zabývat zejména podrobným rozborem použitých metod a prostředků včetně stručného historického kontextu, jako jsou například použité knihovny, programovací jazyky nebo software. Zaměřím se také na problematiku datové komunikace

[1] CZ.07.1.02/0.0/0.0/17_049/0000841 Univerzální výcvikový simulátor vozidla MHD

v informačních technologiích. V následující, praktické části práce za pomoci grafického schématu popíši doporučenou strukturu aplikace a postup při jejím vývoji včetně implementace grafických knihoven SDL2.0 i knihoven, které jsou výstupem této práce. Následně se dle doporučeného schématu budu podrobně věnovat jednotlivým souborům a v nich obsaženým definicím a funkcím. Tyto metody budou demonstrovány na ukázkách programovacího kódu a jejich grafickém výstupu po sestavení aplikace. Následně popíši praktické metody k sestavení funkčního konceptu datové komunikace a vzniku simulátoru zdrojových dat. V závěru práce zhodnotím definované metody a vzniklý software, přičemž základem této zpětné vazby bude jeho instalace na obrazovky simulátoru pražského metra, na kterém bude tento software využíván a testován. Pakliže se demonstrovaný koncept prokáže jako funkční a efektivní, výstupy práce budou poskytnuty univerzitě, fakultě či projektu Dopravního sálu k dalšímu libovolnému užití dle potřeby. Vzhledem k odbornému zaměření práce včetně určité míry odbornosti metodiky popisu použitých metod se od čtenáře pro srozumitelné pochopení problematiky předpokládá základní znalost problematiky kolejové dopravy a programování, respektive pak programovacích jazyků C a C++.

1.1 Cíl práce

Mezi základní požadavky na výstupy dle zadání práce patří:

- Zpracování teoretických východisek pro provozní zobrazení drážních vozidel,
- s využitím dostupných prostředků navrhnout a vyvinout univerzální knihovny specificky určené pro vývoj zobrazení drážních vozidel ve zvoleném programovacím jazyce,
- na základě těchto knihoven navrhnout a vytvořit SW zobrazení stanoveného kolejového vozidla a ověřit jeho funkci,
- vytvořit programátorský manuál k daným knihovnám pro potenciálního uživatele.

Výstupní software, pro jehož vznik bude daných knihoven využito, by měl prokázat schopnost spolehlivého, nepřetržitého běhu a implementovatelnost do technologií simulátorů drážních vozidel. Vysoké požadavky jsou kladeny na nízký reakční čas daného zobrazení, neboť se jedná o zařízení informující strojvedoucího o základních a důležitých provozních stavech, jako je například aktuální rychlost vozidla. Daný software by měl rozumně nakládat se systémovými prostředky, aby byl schopen překreslovat výsledné zobrazení s ohledem na dodržení fyzikálních vlastnosti zobrazovacích zařízení. Samotné knihovny, respektive v nich obsažené funkce spolu s návodem k jejich použití by měly být

stavěny intuitivně s ohledem na minimalizaci množství času nutného pro vývoj i úpravu samotného softwaru v případě potřeby. Schopnost knihoven a výstupního softwaru vyhovět těmto požadavkům tak bude základním aspektem zpětného hodnocení úspěšnosti a efektivity této práce.

1.2 Metodika zpracování

Zpracování stanoveného tématu předpokládá jisté úrovně znalostí použitých metod. Právě proto jsem se rozhodl použít k vývoji programovací jazyk C++, a to zejména z důvodu nezanedbatelného množství mých předchozích zkušeností s tímto jazykem. S ohledem na zadání jsem zvolil jako základ práce veřejně dostupné grafické knihovny SDL 2.0, které představují velmi silné rozhraní pro vývoj grafických aplikací. Jejich použití však není zcela intuitivní, vyžaduje vyšší míru předchozích znalostí a zkušeností a zejména podrobnou znalost dokumentace. Naopak knihovny, které jsou výstupem této práce, budou koncipovány s maximálním ohledem na jednoduchost a možnost jejich užití bez nutnosti náročného studování dokumentací, byť je podrobné prostudování logiky daného řešení vždy tou zodpovědnější a doporučenou cestou. Je však nutné brát ohledy také na situace, kdy předdefinované funkce zcela nevyhovují požadavkům vývojáře na své chování či výsledný grafický výstup a bude tak nutné do jejich zdrojového kódu zasáhnout a upravit jej dle specifických požadavků. Typickým případem může být například zobrazení ukazatele poměrného tahu. Příklad řešení tohoto ukazatele bude součástí výstupních knihoven, v rámci kterých předdefinuji funkci, která vytvoří základ takového zobrazení, připouští se však nutnost její úpravy tak, aby grafický výstup odpovídal parametrům konkrétního zobrazení, například použitých barev, tloušťky jednotlivých linií, popisků os a podobně.

K programování a tzv. debuggingu, tedy ladění chyb, použiji vývojové prostředí Microsoft Studio Professional 2019, neboť se jedná o jedno z nejpoužívanějších vývojových prostředí současnosti, nativně podporuje vývoj desktopových aplikací v C++ a možnost jeho propojení s grafickými knihovnami SDL2.0 je podrobně na webových zdrojích zdokumentována. Před samotným aplikováním každé z uvedených metod zprvu provedu jejich analýzu, aby se prokázala vhodnost použití dané metody a došlo k pochopení jejích vnitřních i vnějších souvislostí.

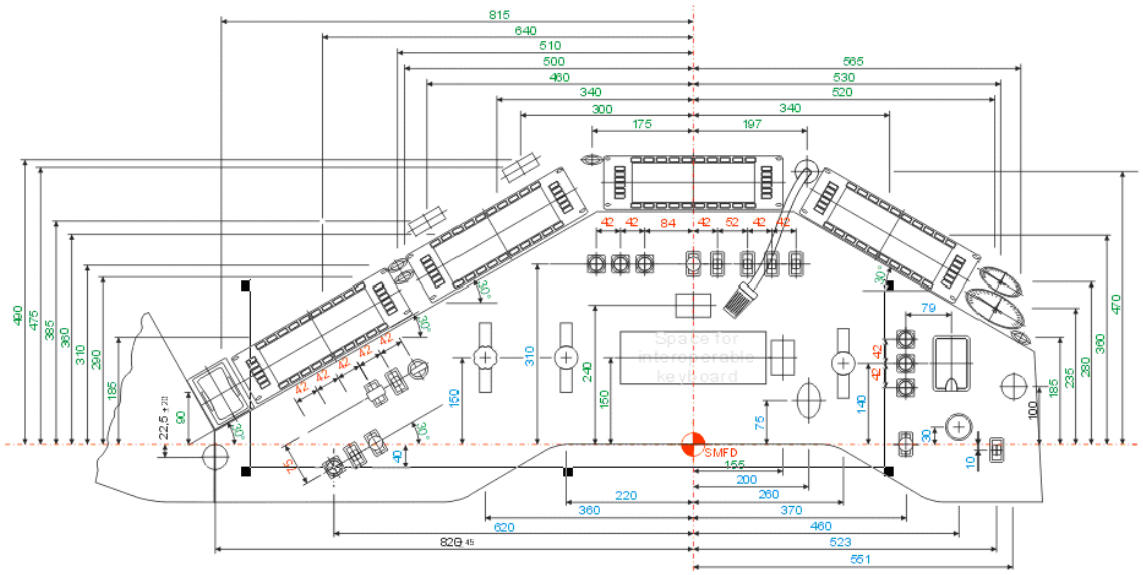
Ze zvolených jazyků a metod se odvíjí množina použité odborné literatury ke zpracování práce jak v její analytické, tak praktické části. V prvním případě půjde zejména o zdroje informací pro analýzy a uvedení čtenáře do kontextu dané problematiky, ve druhém případě se pak bude jednat o zisk nutných znalostí pro samotný vývoj aplikací studiem dokumentací, učebnic a návodů.

2 Teoretická východiska pro provozní zobrazení drážních vozidel

Problematikou provozních zobrazení kolejových vozidel, konkrétně elektrických či dieselových jednotek, lokomotiv a řídicích vozů se částečně zabývá například směrnice Mezinárodní železniční unie (UIC) 612. Ta se podrobně věnuje zejména problematice rozhraní mezi obsluhou lokomotivy a lokomotivou samotnou. Dokument obsahuje popis a požadavky struktury a rozvržení ovládacích prvků na pultu stanoviště strojvedoucího, dále se zabývá rozvržením strojovny a umístěním jednotlivých operačních elementů na hnacím kolejovém vozidle (HKV). Cílem je zejména standardizace prostředí obsluhy na v Evropě provozovaných hnacích a řídicích kolejových vozidlech. (1)

2.1 Rozvržení pultu strojvedoucího

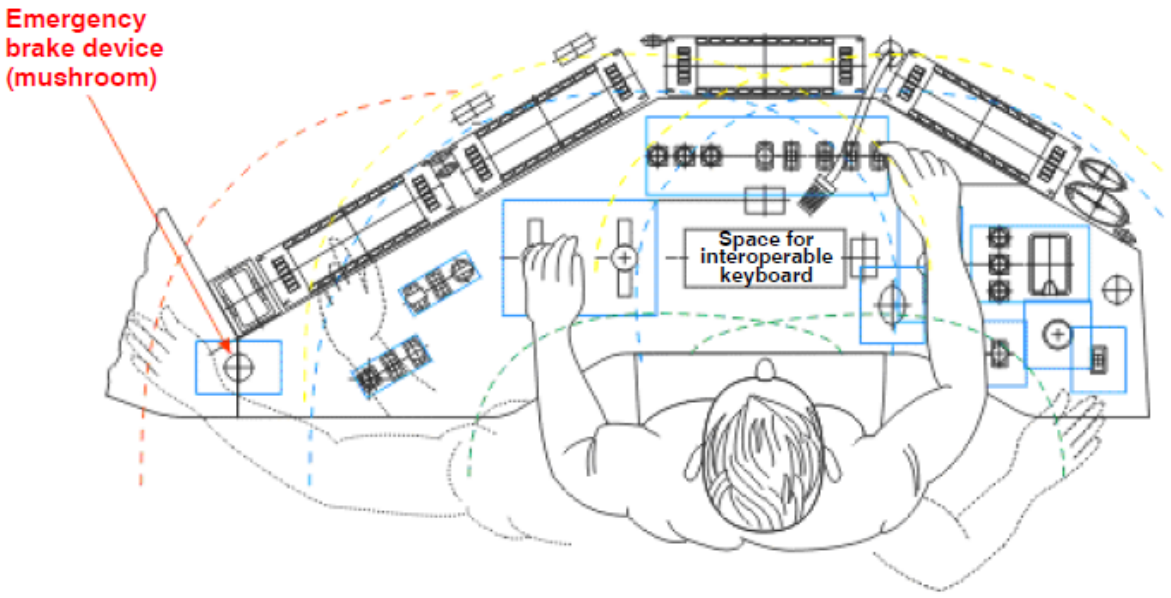
Ve svých prvních částech se UIC 612 zabývá zejména rozvržením pultu strojvedoucího. Ten dle směrnice může být na stanovišti umístěn na levé či pravé straně nebo veprostřed, jeho poloha na stanovišti však nesmí mít vliv na rozvržení jednotlivých ovládacích prvků. Doporučené vzdálenosti jednotlivých řídicích, diagnostických i bezpečnostních prvků jsou uvedeny na následujícím schématu: (Obr. 1).



Obr. 1 - Doporučené vzdálenosti ovládacích prvků pultu od obsluhy

zdroj: UIC 612 (1)

Důraz je kladen na to, aby veškeré interaktivní prvky byly s ohledem na bezpečnost a komfort umístěny v dosahu sedící obsluhy bez nutnosti výrazných pohybů celého těla. Spodní část panelu by měla obsahovat zejména prvky čistě ovládacího charakteru, tedy klávesnice, tlačítka, páky apod. V horní části pultu se pak mají nacházet zejména informativní elementy, jako jsou displeje a manometry. Ovladače trakce by neměly být umístěny na stejné straně jako hlavní ovladače pneumatické brzdy tak, aby nemohlo dojít k jejich záměně. Specifickým případem pak mohou být sdružené ovladače, které kombinují ovládání trakce i brzdy zároveň. Ovladač trakce (který je v případě sdružené jízdní páky zároveň ovladačem elektrodynamické brzdy) by se měl nacházet po levé straně obsluhy. Na levé straně by se dále neměly nacházet žádné další prvky, které by vyžadovaly ovládání dvou či více prvků zároveň, ty musí být pak logicky umístěny v dosahu druhé ruky na pravé straně. V těsné blízkosti ukazatelů tlaku v brzdové jímce by se měl vždy nacházet odbrzďovač pro případ přetlakování jímky. Tlačítko nouzové brzdy se musí také nacházet v blízkosti oblasti vyhrazené pro další osobu na stanovišti. Nutno však dodat, že doporučovaná řešení slouží primárně pro lokomotivy traťové služby. Řešení pro posunovací lokomotivy doposud nebylo navrženo. (1)

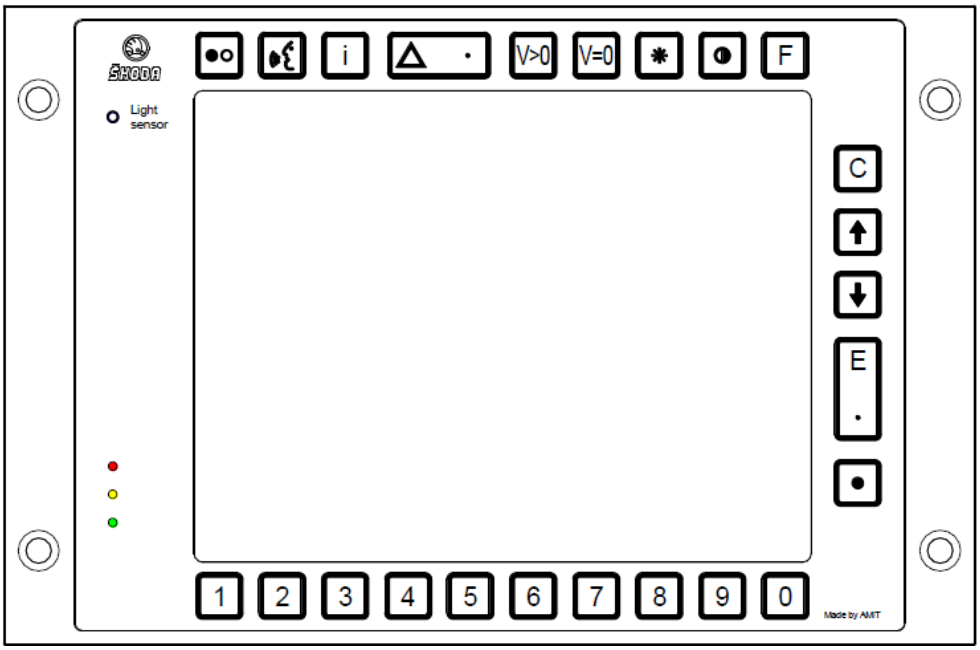


Obr. 2 - Nákres maximálních vzdáleností ovládacích prvků v dosahu ruky obsluhy

zdroj: UIC 612 (1)

2.2 Displeje strojvedoucího

K zobrazovacím zařízením na stanovišti se vyhláška UIC 612-0 vyjadřuje zejména ve smyslu, že na displejích by měly být obsaženy pouze informace nezbytně nutné k řízení vozidla, neboť jakékoliv nesouvisející informace by mohly negativně ovlivnit koncentraci a reakceschopnost obsluhy. Návrh jednotného pultu strojvedoucího počítá až se čtyřmi displeji na stanovišti – provozním displejem, zobrazujícím informace vztahující se k jízdě vlaku, jako je rychlost vlaku nebo poměrný tah, technickým a diagnostickým displejem, zobrazujícím provozní stavy vozidla a vlaku a doplňující diagnostické informace, displejem elektronického jízdního řádu a displejem vlakového rádia. U displejů se doporučuje úhlopříčka 10,4“ a musí disponovat schopností automatické i manuální regulace jasu. Příkladem takového displeje může být například palubní počítač s obvodovou klávesnicí AMiT APT9110T. Alespoň dva displeje – Hlavní ovládací displej (CCD) a Technický a diagnostický displej (TDD) musí být dle normy UIC 612 vzájemně zálohovatelné, tedy pokud dojde u jednoho z displejů k poruše, musí být jeho zobrazení možné vykreslit na druhém z nich. U displejů se také předpokládá možnost zobrazení obrazu z kamer jakožto alternativy zpětných zrcátek nebo nastavení vlakového informačního systému pro cestující. (1), (2)



Obr. 3 - Nákres přední strany počítače AMiT APT9110T

zdroj: (3)

3 Železniční simulátory v obecném kontextu

Nejen nehody a mimořádné situace na železnici, ale také vývoj a vznik nejnovějších technologií zejména v oblasti železniční zabezpečovací techniky zapříčinil rostoucí poptávku nejen železničních dopravců po interaktivních simulátorech kolejových vozidel zaměřených zejména na problematiku řízení a ovládání vozidla strojvedoucím. Simulátory kolejových vozidel se dnes běžně využívají zejména v západních zemích EU, a slouží především pro výcvik obsluhy hnacího vozidla ohledně řízení, řešení poruch a přípravu na řešení krizových situací. Simulátory však mohou také sloužit jako efektivní nástroj k testování a seznámení obsluhy s nejnovějšími technologiemi už ve fázi jejich vývoje a testování. Takových školících pracovišť, využívající simulátory kolejových vozidel pro odborně-vzdělávací účely však v současné době neexistuje mnoho. Můžeme se hojněji setkat s neprofesionálními simulátory sloužícími především k herním účelům, mající zejména audiovizuální charakter. Cílem projektu, v rámci kterého řeším tuto problematiku, je vyvinout zcela profesionální nástroj určený ke vzdělávacím a vývojovým účelům. Profesionální simulátory ve výcviku strojvedoucích v současné době, nově mezi zeměmi východní Evropy, zavedl například polský státní dopravce PKP Intercity využívající dvou simulátorů umístěných ve varšavském depu Olszynka Grochowska. První z nich je věrnou kopií stanoviště polského provedení elektrické lokomotivy řady EU 44, druhý simulátor je upraven jako zmodernizovaná rychlíková lokomotiva typu EP 09. (4)

Obr. 4 - Simulátory PKP Intercity

zdroj: (4)

Svůj záměr objednat dva obdobné trenažery ztvrdily také ČD ve druhé polovině roku 2017. Dopravce s podobným záměrem operoval již od doby tragické nehody ve Studénce roku 2008 za účelem již zmiňované přípravy strojvedoucích na mimořádné události, více reálných rozměrů záměr následně nabyl po tragické srážce jednotky Pendolino s kamionem opět ve Studénce roku 2015. Simulátory jsou založeny na vzhledu odpovídajícím stanovišti

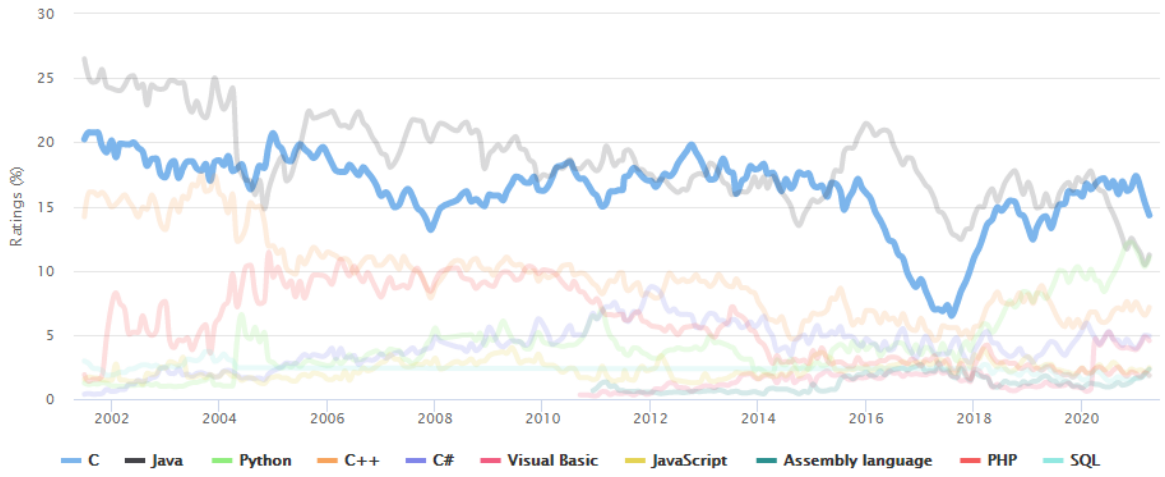
strojvedoucího elektrických jednotek typu 440/640/650/661 a dopravce je za účelem školení vlastních strojvedoucích umístil do dopravně-vzdělávacího střediska v České Třebové a v Praze, PJ Vršovice. Jedná se o statický simulátor, který sice má reálné ovládací prvky a je zde snaha o reálnou simulaci jízdy po funkční i vizuální stránce. Simulátor má však svá omezení ve vjemu reálného chování, zejména díky absenci pohybového systému navozujícího dynamiku jízdy vozidla. (5), (6)

4 Výběr použitých metod a jazyků pro návrh knihovny

Níže uvedu a popíši dílčí nástroje, metody a prostředky, jejichž bylo při vývoji knihoven využito.

4.1 Programovací jazyk C

Jazyk C patří mezi základní velmi populární programovací jazyk se širokou škálou možného využití. Dle indexu TIOBE se jedná o v současnosti (duben 2021) nejpopulárnější programovací jazyk na světě s procentuálním zastoupením 14.32%. (7)
Je uváděno, že C je jakousi vstupní bránou do světa programování – pokud programátor pochytí základy jazyka C, přizpůsobení se na jakýkoliv jiný programovací jazyk by posléze mělo být značně jednodušší. Ostatně je faktem, že spousta novějších programovacích jazyků je napsána právě v jazyce C. (8)



Obr. 5 - Vývoj popularity jazyka C v čase dle indexu TIOBE

zdroj: (7)

4.1.1 Základní syntaxe

Mezi základní syntaktická specifika jazyka C patří například direktiva #include sloužící k importu hlavičkového souboru z knihovny. Funkce jsou definovány uvedením datového typu, názvu funkce a závorkami obsahující případné vstupní argumenty funkce. Naopak výstup funkce je uveden příkazem return. Funkce nevracející žádnou hodnotu mají datový typ void. (8)

```
void SetColor(SDL_Color color)
{
	SDL_SetRenderDrawColor(gRenderer, color.r, color.g, color.b, color.a);
}
```

Ukázka kódu 1 - definice jednoduché funkce v jazyce C

4.1.2 Specifika

Jazyk C je kompilovaným jazykem, tedy jazykem, který využívá speciálního nástroje, tzv. kompilátoru, k překladu kódu do strojově čitelného programu. Výsledkem kompilace posléze jsou objektové soubory (typicky přípona .o nebo .obj), které jsou následně nástrojem zvaným linker zkombinovány do jednoho finálního spustitelného souboru (přípona .exe). Jak jsem již uvedl, C je jazykem nízké úrovně velmi blízkým samotnému strojovému kódu. V dnešní době je podporován většinou operačních systémů a hardwarových platforem. Jedná se o procedurální a strukturálně orientovaný programovací jazyk. (9)

4.2 Programovací jazyk C++

C++ je objektově orientovaný programovací jazyk přímo vycházející z C, se kterým je zpětně kompatibilní. Uvádí se, že přes 99% kódu v jazyce C je přenositelných a kompilovatelných také v jazyce C++. Oproti jazyku C nabízí C++ zejména rozšíření o možnost objektově orientovaného programování a s ním souvisejících tříd, jakožto základního typu objektu.

4.2.1 Důvody použití jazyka C++

Jazyk C++ jsem zvolil zejména díky jeho možnostem širokého využití napříč různými platformami. Byť cílovou platformou navrhovaného řešení je operační systém Windows, považuje se za vhodné udržovat možnost exportu daného řešení také pro ostatní platformy jako je například operační systém Linux či z něj vycházející OS Android. Konstrukce specifické pro jazyk C++ je však v případě vzniklých knihoven využíváno pouze okrajově a většina syntaxe vychází z nativního jazyka C.

4.2.2 Základní syntaxe

Není překvapením, že základní syntaxe jazyka C++ vychází přímo z jazyka C. Novinkou zde však jsou třídy (značené klíčovým slovem class), které představují popis objektů, jejich stavů a chování – tzv. metod.

```
#include <iostream>
using namespace std;

int main() {
	cout << "Hello World";
	return 0;
}
```

Ukázka kódu 2 - jednoduchá aplikace v jazyce C++

zdroj: vlastní zpracování úpravou z (10)

C++ také přichází s rozšířením o direktivu using, která určuje názvosloví, tedy jakousi sadu příkazů, kterou daný kód užívá. Výše můžeme vidět jednoduchou aplikaci, která vypíše do konzole text „Hello World!“. (10)

4.2.3 Specifika

C++ je otevřeným jazykem standardizovaným organizací ISO přímo vycházející z jazyka C, který s ním nabízí plnou zpětnou kompatibilitu. Jedná se taktéž o kompilovaný jazyk, který je při sestavení převáděn ze zdrojového kódu na strojový. Podporuje statické i dynamické typování a má jednu z největších bází dostupných knihoven. Jedná se o objektově orientovaný programovací jazyk. (11)

4.3 Grafické knihovny SDL 2.0

SDL 2.0 je multiplatformní vývojová knihovna navržená k poskytování snadného přístupu k ovládacím prvkům a grafickému hardwaru prostřednictvím OpenGL a Direct3D. SDL oficiálně podporuje platformy Windows, Mac OS X, Linux, iOS a Android. Samotné knihovny jsou napsány programovacím jazykem C, pracují však přirozeně s C++ a lze je při použití patřičných prostředků případně provázat i s jazyky C# nebo Python. SDL verze 2.0 je distribuována pod licencí zlib, která umožňuje volné užití těchto knihoven v libovolném softwaru. Autory licence jsou Jean-loup Gailly a Mark Adler. (12)

4.3.1 Historie

Za zrodem Simple Direct-media Layer stojí známý vývojář Sam Lantinga. Ten se proslavil mj. pro svou práci v herním studiu Blizzard, které stojí například za světoznámou počítačovou hrou World of Warcraft. V současnosti Lantinga pracuje pro taktéž velmi známou herní společnost Valve. (13) První verze knihovny byla vydána roku 1998 v době, kdy Lantinga pracoval pro společnost Loki Software. Myšlenka na vznik knihovny údajně vznikla během portování aplikace pro Windows na operační systém Macintosh – základní myšlenkou totiž je kompatibilita mezi různými platformami. Později použil Lantinga vzniklé knihovny k naportování populární hry Doom na operační systém BeOS. Roku 2008 založil

společnost Galaxy Frameworks za účelem rozšíření komerční pomoci vzniku a distribuce těchto knihoven. Knihovny procházely během času dalším vývojem, verze 1.3 například přinesla rozsáhlou podporu uživatelských vstupů a výstupů. Právě verze 1.3 byla pouze přečíslována na verzi 2.0, ta byla následně šířena pod licencí zlib. (14)

4.3.2 Specifika a princip

Slovo „layer“ (česky vrstva) v názvu knihoven představuje pomyslný obal funkcí specifických pro operační systém. Hlavním cílem SDL je poskytnout standardizované rozhraní pro přístup k těmto funkcím, nezávisle na operačním systému, pro který je daná aplikace kompilována. SDL obsahuje možnosti využití rozhraní OpenGL i Direct3D pro přístup k funkcím grafické karty. Využívá také knihoven OpenTTD pro poskytnutí rozhraní k uživatelským vstupům z myši, dotykové obrazovky, klávesnice apod. Knihovna je dále dělena do dílčích subsystémů, například knihovny SDL_Image poskytující podporu různých obrazových formátů, nebo knihovny SDL_ttf poskytující totéž pro vykreslování písma typu TrueType. Knihovna poskytuje návaznost na široké spektrum programovacích jazyků jako je C, C++, C# nebo Python až po méně běžné programovací jazyky typu Lua, Euphoria nebo Pliant. Syntaxe SDL je založená na funkcích – veškeré operace SDL jsou prováděny prostřednictvím předávání parametrů těmto funkcím. SDL taktéž obsahuje speciální datové struktury pro poskytování specifických informací, jako jsou například SDL_Surface nebo SDL_Texture. (14)

4.3.3 Vlastnosti

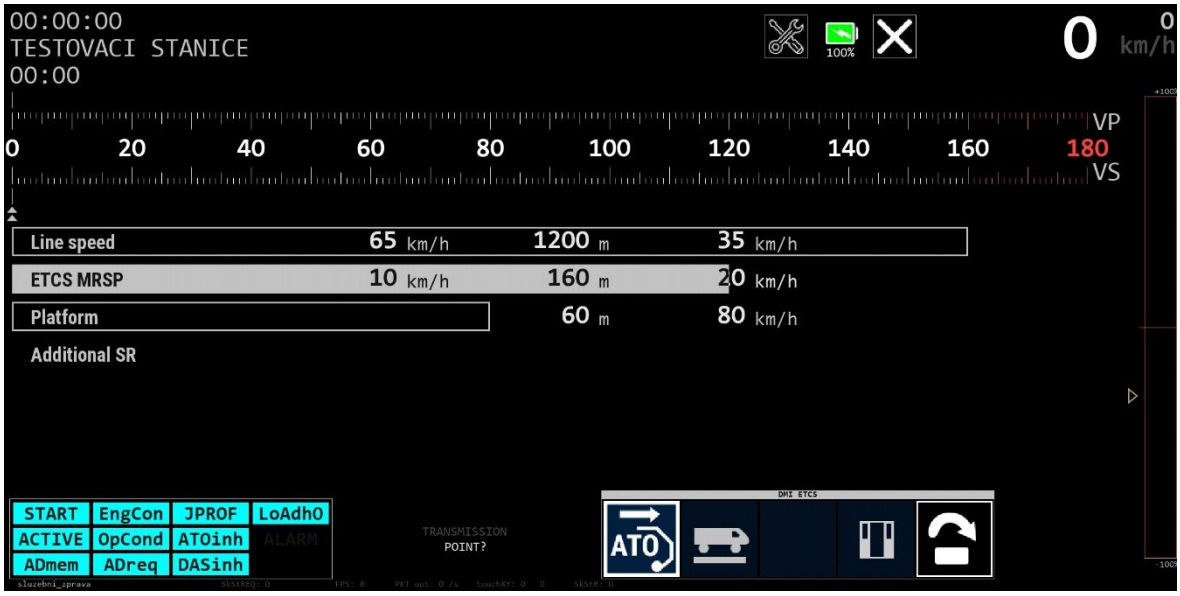
Knihovny SDL poskytují velmi silné rozhraní pro přístup ke grafickému rozhraní široké škály hardwaru a operačních systémů. Proto se jejich užití pro grafické aplikace jeví jako velmi vhodné. Jejich použití však vyžaduje jistou míru znalostí, které je nutné získat detailním nastudováním dokumentace či dostupných návodů. Pro usnadnění vývoje specifických druhů zobrazení tak vyvstala nutnost tyto knihovny zapouzdřit do vyšší vrstvy, která vykreslování dílčích geometrických obrazců, textů a dalších elementů do 2D scény značně usnadní.

4.3.4 Důvody použití jazyka C a knihovny SDL

Hlavním důvodem, proč jsem se rozhodl použít jazyk C, popřípadě C++ v kombinaci s grafickými knihovnami SDL 2.0, je jejich multiplatformní charakter. Dané knihovny jsou navrženy tak, aby vzniklý kód byl do maximální možné míry přenositelný mezi jednotlivými platformami a nebylo nutné jej za tímto účelem zásadním způsobem upravovat. Proto je

však při vývoji nutné vyhnout se specifickým konstrukcím pro konkrétní platformu a využívat výhradně vestavěných funkcí knihoven SDL. Ne vždy je však možné se takovým konstrukcím vyhnout. Typickým příkladem takové situace je implementace datové komunikace, kde se knihovny pro nastavení komunikačních soketů v rámci dílčích platforem značně liší. Například OS Windows k datové komunikaci v C či C++ využívá rozhraní Winsock, které v ostatních platformách pochopitelně není dostupné.

Možnost přenášení kódu v C++ s SDL 2.0 mezi platformami jsem měl možnost si osobně vyzkoušet například v případě provozního zobrazení ATO over ETCS, které jsem za účelem rozšíření zkušeností v programování pro platformy mimo OS Windows dokázal implementovat také na mobilní telefon s operačním systémem Android (Obr. 6). Mimo již zmíněné specifické jazykové konstrukce jsem ověřil, že funkce SDL vskutku bez problému fungují v rámci různých platforem, a to i při implementaci řešení poměrně specifického problému, jako je získání informací o napájení zařízení či stavu baterie.



Obr. 6 - Aplikace s využitím SDL 2.0 na mobilním telefonu s OS Android

zdroj: vlastní zpracování

4.4 Zdůvodnění volby IDE prostředí Microsoft Visual Studio

Vývojové prostředí (anglicky Integrated Development Environment) je jedním ze základních, nikoli však nezbytných nástrojů každého programátora. Jedná se o samostatný program, který je určen k programování dalších aplikací a poskytuje k tomu uživateli určité množství nástrojů. Standardními součástmi dnes jsou editor zdrojového

kódu, kompilátor a debugger. Pro vývoj aplikace v rámci této diplomové práce jsem zvolil právě Microsoft Visual Studio 2019 ve verzi Professional, a to hned z několika důvodů. Primárně je to má dlouhodobá osobní znalost a dobré zkušenosti s tímto prostředím, dále obrovské množství prostředků, které toto prostředí nabízí, podpora velkého množství programovacích jazyků a komponent a v neposlední řadě také vysoká míra individuální customizace. Právě toto osobní nastavení patří mezi určité charakteristiky každého programátora spolu s jeho autorským rukopisem. Byť takové věci, jako specifické nastavení zvýrazňování kódu nemá na funkci výsledného programu žádný vliv, pro programátora jde o velmi užitečnou pomůcku, která mu na čistě vizuální bázi pomáhá odlišit jednotlivé části kódu, druhy proměnných, definice od deklarací a další. Dle indexu PYPL se jedná v současnosti o nejvyužívanější vývojové prostředí na světě. (8)

Nepopiratelnou výhodou tohoto IDE je pak opět možnost využívat jej ke tvorbě kódu v různých jazycích pro různé platformy. Visual Studio lze formou volitelných součástí rozšířit o prostředky nutné ke kompilaci či vývoji různých druhů aplikací s využitím různých nástrojů. V rámci své diplomové práce na Univerzitě v Hradci Králové (8) jsem například prokázal jeho vhodnost ke tvorbě webových aplikací v jazyce C#. Možnosti Visual Studia sahají však mnohem dále a není tak problém jej provázat například s emulátorem operačního systému Android pro vývoj mobilních aplikací.

5 Síťová komunikace v IT (networking)

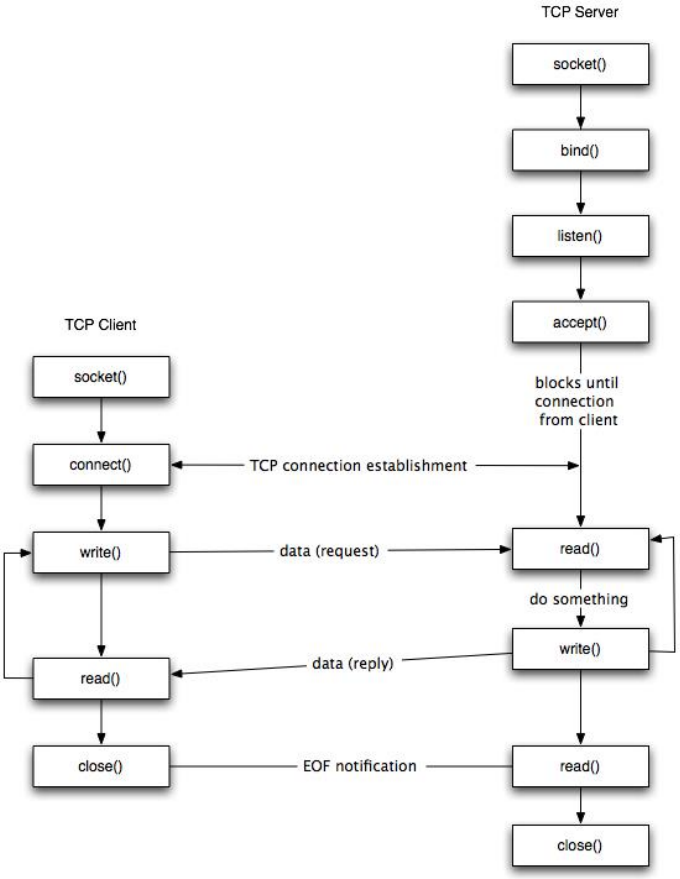
Síťová komunikace – tzv. programování soketů, představuje způsob propojení dvou koncových zařízení (uzlů) v počítačové síti pro jejich vzájemnou komunikaci. Zatímco jedno ze zařízení naslouchá na určeném portu, ostatní zařízení k danému portu přistupují, aby došlo k navázání spojení. V informační terminologii jsou tato dvě základní zařízení nazývána jako server a klient. (15) Mezi nejběžnější typy protokolů pro komunikaci zařízení v počítačové síti pak patří TCP/IP nebo UDP.

5.1 TCP/IP

TCP/IP protokol cílí specificky na vysokou míru spolehlivosti a kontroly, zda byl přenos daných dat kompletní. Toho je docíleno například užitím třícestného handshake, tedy metody spojení dvou zařízení a ověření, zda jsou tato zařízení skutečně spojena. Datový přenos začíná až v momentě, kdy je spojení obou zařízení potvrzeno. Poté, co je přenos dat ukončen, dojde k ukončení spojení zavřením daného obvodu. Úspěšný přenos každého datového rámce, tzv. paketu, je potvrzen cílovým zařízením odesláním tzv. acknowledgementu. V případě, že zdrojové zařízení toto potvrzení po uplynutí určité doby neobdrží, odešle daný datový rámec znovu. Protokol také například umí pružně reagovat na přetížení sítě odložením vyslání paketů. TCP/IP je tedy považován za protokol orientovaný zejména na spojení. Pro ustálení spojení je nutná výměna celkem tří paketů mezi klientem a serverem. Hlavička standardního TCP/IP paketu má 20 bytů – zejména proto, že obsahuje rozšířené informace o odeslaném paketu pro kontrolu jeho doručení. TCP/IP protokol je tak o něco „těžší“ a pomalejší, než alternativní UDP. Schéma TCP/IP komunikace je možné vidět na Obr. 7. (16), (17)

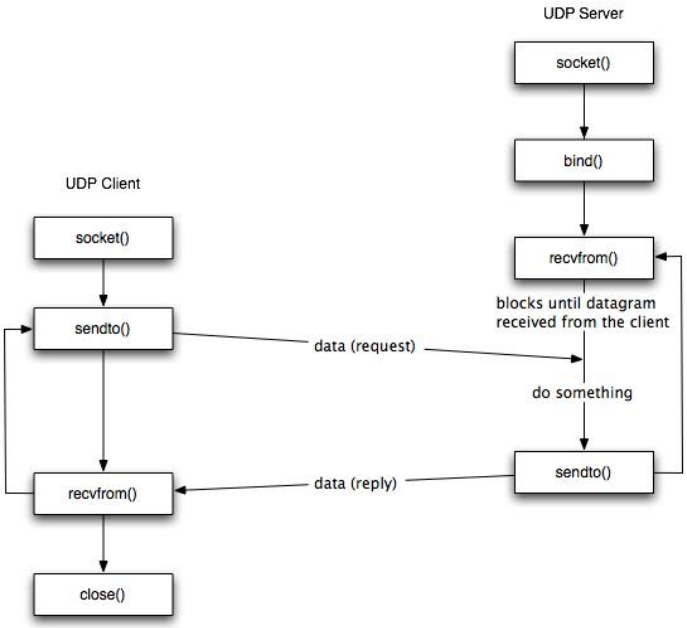
5.2 UDP

Protokol UDP je ve srovnání s TCP/IP značně jednodušší. Jedná se o datagramově orientovaný protokol, neobsahuje tedy kontrolu spojení ani doručení dat. Ustálení takového spojení je tak značně jednodušší, neboť nemusí proběhnout handshake. Protokol je vhodný zejména pro kontinuální vysílání dat, kdy například zdrojové zařízení nepřetržitě vysílá určité informace, aniž by se jakkoliv staralo o to, zda byla data někam doručena, zatímco koncová zařízení se dle potřeby mohou kdykoli k datovému toku připojit či odpojit. Hlavička UDP paketu má pouhých 8 bytů a spojení je značně rychlejší, to však na úkor absence jakékoli kontroly spolehlivosti. Schéma UDP komunikace je možné vidět na Obr. 8. (16), (17)



Obr. 7 - schéma komunikace prostřednictvím TCP/IP

zdroj: (16)

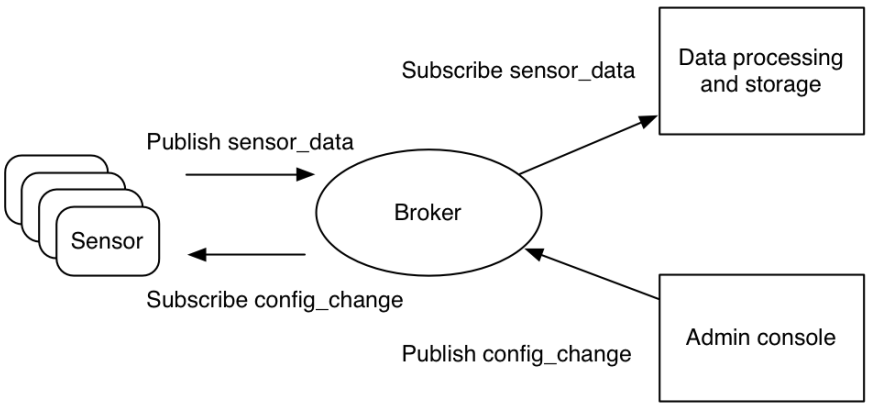


Obr. 8 - Schéma komunikace prostřednictvím UDP

zdroj: (16)

5.3 MQTT Technologie

MQTT (Message Queuing Telemetry Transport) představuje asynchronní komunikační protokol založený na TCP/IP navržený pro vzájemnou komunikaci zařízení zejména v rámci tzv. internetu věcí. Komunikace prostřednictvím této technologie je založená na existenci dvou zpráv typu publish/subscribe (zveřejnit/odebírat). To v praxi znamená, že cílová zařízení se přihlašují k odběru určitého typu informací, které si přejí od zdrojového zařízení získávat. Tak mohou učinit buď na základě vlastního požadavku či formou aktualizací při změně stavu od daného zdrojového zařízení. Takové řešení představuje základní výhodu zejména v přetížených či nepříliš spolehlivých sítích, neboť data jsou vysílána pouze jednorázově v případě potřeby iniciované buď zdrojovým, či cílovým zařízením. Samotný protokol je považován za velmi lehký a spolehlivý. (18) Schéma takové komunikace je možné vidět na Obr. 9. Technologie MQTT je v rámci projektů simulátorů vozidel využíváno pro přístup do datových skladů umožňujících sdílení proměnných a zpráv mezi jednotlivými komponentami daného simulátoru. Vstupy datových skladů představují například ovládací prvky pultu strojvedoucího, odkud si jejich stavy následně přebírá logika vozidlového počítače. Výstupů datového skladu dále mohou využívat například právě aplikace zobrazovacích zařízení pro přístup nejen k těmto informacím.



Obr. 9 - Schéma komunikační struktury prostřednictvím MQTT

zdroj: (18)

6 Základní charakteristika knihoven SDL 2.0

Při vývoji generických knihoven pro provozní zobrazení kolejových vozidel jsem se na základě vlastních kladných uživatelských zkušeností rozhodl využít grafických knihoven SDL 2.0. V této části proto představím některé ze základních datových typů a funkcí, kterými tyto knihovny disponují a jsou využity v mém návrhu.

6.1 Základní funkce SDL 2.0

SDL_Init()

datový typ: integer; parametry: Uint32 flags

Funkce SDL_Init slouží k obecné inicializaci knihoven SDL v programu. Tato funkce vrací datový typ integer. Hodnota 0 znamená úspěch, tedy informaci, že inicializace SDL proběhla bez problému. Ostatní hodnoty představují kód chyby, ke které při inicializaci došlo. Funkce má jeden vstupní parametr, a to sice 32-bitový unsigned integer flags. Tento parametr představuje předdefinovanou škálu hodnot, které určují subsystémy, které se mají v rámci SDL inicializovat. Typickými subsystémy jsou například subsystém video a audio. Jednotlivé chybové kódy i hodnoty parametru flags jsou dostupné v dokumentaci ke knihovnám SDL. (19)

SDL_SetHint()

datový typ: boolean; parametry: const char *name, const char * value

Funkce SDL_SetHint slouží k nastavení příznaků knihoven SDL. První parametr představuje ukazatel na vlastnost, kterou si přejeme nastavit, ve druhém parametru pak specifikujeme požadovanou hodnotu této vlastnosti. Seznamy těchto hodnot jsou opět dostupné v dokumentaci (19). Funkce vrací v případě, že nastavení dané vlastnosti proběhlo v pořádku, hodnotu true, v opačném případě pochopitelně vrátí hodnotu false. Příkladem příznaků, které je možné funkcí nastavit, je například požadovaný grafický ovladač (např. OpenGL nebo Direct3D) nebo kvalita vykreslovaného obrazu. Čím vyšší kvalitu nastavíme, tím bude obraz čistší, ale jeho vykreslení si vyžádá větší systémové prostředky. (19)

SDL_CreateWindow()

datový typ: SDL_Window*; parametry: const char* title, int x, int y, int w, int h, Uint32 flags

Tato funkce, jak může název napovědět, slouží k vytvoření grafického okna, do kterého budou definované prvky vykreslovány. Jejími argumenty jsou: název okna, souřadnice pro

pozici okna na obrazovce, šířka a výška okna a již známý parametr flags, který slouží k nastavení jistých vlastností tohoto okna. Souřadnice pro pozici okna nemusí být definovány konkrétním číslem, pro jejich stanovení může posloužit například hodnoty SDL_WINDOWPOS_CENTERED, která automaticky umístí okno do středu obrazovky dle dané souřadnice, nebo SDL_WINDOWPOS_UNDEFINED, která přenechá pozici okna výchozímu systémovému nastavení. Mezi nastavitelné příznaky patří například možnost zobrazení okna v popředí, v minimalizovaném zobrazení apod. (19)

SDL_SetWindowFullscreen()

datový typ: integer; parametry: SDL_Window* window, Uint32 flags

Účel této funkce je možné opět vytušit z jejího názvu. Funkce slouží k nastavení zobrazení daného okna na celou obrazovku. Základní parametr představuje SDL_Window, tedy dříve vytvořené okno, na které si uživatel přeje toto nastavení aplikovat, druhým parametrem pak jsou specifické příznaky tohoto okna. Hodnota druhého z parametrů 0 znamená zobrazení v okně, hodnota SDL_WINDOW_FULLSCREEN pak odpovídá zobrazení na celou obrazovku, hodnota SDL_WINDOW_FULLSCREEN_DESKTOP rozšíří velikost zobrazení na celou plochu obrazovky, ale ponechá jej v okně. (19)

SDL_CreateRenderer()

datový typ: SDL_Renderer*; parametry: SDL_Window* window, int index, Uint32 flags

Tato funkce vytváří tzv. renderer, tedy nástroj, který se postará o vykreslení definovaných prvků do stanoveného okna. Tomu odpovídají také parametry funkce. V prvním parametru uživatel stanovuje okno, které má renderer pro vykreslování používat, parametr index pak slouží k indexaci grafického ovladače, kterého má renderer využívat. Zde se doporučuje nastavení hodnoty na -1, která znamená, že renderer využije ovladače dříve inicializovaného v rámci funkce SDL_SetHint(). Mezi nastavitelné příznaky pak patří například volba, zda má renderer k vykreslení scény využívat softwarových či hardwarových prostředků. (19)

SDL_SetRenderDrawColor()

datový typ: integer; parametry: SDL_Renderer* renderer, uint8 r, g, b, a

Funkce slouží k nastavení barvy rendereru pro provedení některých základních vykreslovacích operací, jako je například kresba linie, bodu, obdélníku, nebo celé plochy okna. Parametry jsou: vytvořený renderer, u kterého se má barva použít a hodnoty RGBA dané barvy. (19)

SDL_PollEvent()

datový typ: integer; parametry: SDL_Event* event

Jedná se o funkci sloužící ke zpracování systémových událostí. Takovou událostí může být například stisknutí klávesy na klávesnici, pohyb myší nebo dotyk či uvolnění na dotykové obrazovce. Parametrem je definovaná instance událostí, o jejichž zpracování se stará vnitřní logika knihoven SDL. (19)

SDL_RenderFillRect()

datový typ: integer; parametry: SDL_Renderer* renderer, const SDL_Rect* rect

Tato funkce slouží k vykreslení obdélníku do scény. Základním parametrem je dříve definovaný renderer, který se má pro vykreslení využít, a struktura typu SDL_Rect, definující pozici a rozměry tohoto obdélníku. Před voláním funkce se, je-li to nezbytné, doporučuje volání funkce SDL_SetRenderDrawColor, která rendereru přiřadí barvu, kterou má dále definované objekty do scény vykreslit. V případě úspěchu funkce vrátí hodnotu 0, v případě neúspěchu pak kód chyby. (19)

SDL_RenderDrawLine()

datový typ: integer; parametry: SDL_Renderer* renderer, int x1, y1, x2, y2

Z názvu a parametrů funkce je v kontextu dříve uvedených funkcí patrně možné vytušit její účel. Tím je vykreslení linie pomocí daného rendereru, přičemž vstupními daty jsou souřadnice x a y pro začátek a konec této linie. Návratové hodnoty jsou zde stejné, jako v případě funkce SDL_RenderFillRect(). (19)

6.1.1 Textové funkce SDL_TTF

Vykreslení textu do scény s použitím knihoven SDL není tak jednoduché, jak by se mohlo na první pohled zdát. Pro potřeby vykreslování textů v českém jazyce je nutné například provést řadu konverzí, aby bylo možné požadovaným textem naplnit patřičnou funkci knihovny SDL_TTF. Tato knihovna nabízí hned několik metod, kterými lze vykreslit text do prostoru SDL_Surface, tedy do jakési dvourozměrné množiny pixelů. Tyto metody lze rozdělit dle dvou hlavních kritérií, a to dle hladkosti (kvality) vykreslení a dle kódování znaků.

Dle hladkosti vykreslení zde rozlišujeme tři úrovně, a to: *solid*, *shaded* a *blended*.

Úroveň *solid* představuje nejrychlejší, ale zároveň nejméně čistý způsob vykreslení. Vykreslení textu zde využívá pouze osmibitový prostor, přičemž povolené hodnoty

pro každý jeho pixel jsou pouze 1 a 0, kde 1 znamená vybarvený pixel a 0 stojí pro pixel průhledný. Z toho se dá poměrně snadno odvodit, že tato úroveň nepodporuje žádnou formu částečné průhlednosti a text se tak bude jevit typicky pixelově kostrbatý. Použití této úrovně je tak vhodné zejména v případech, kde nejsou kladeny vysoké nároky na obrazovou kvalitu výstupu, nebo se jedná o text, u kterého je předpoklad jeho velmi časté změny. Dalším důvodem pro použití této úrovně je pochopitelně její velmi nízká náročnost na paměť i výpočetní výkon. (20)

Úroveň *shaded* stále operuje v osmibitovém prostoru, podporuje však již v omezené míře částečnou průhlednost. Hodnota pixelu 0 znamená průhledný pixel, přičemž ostatní hodnoty od 1 do 7 odpovídají určité míře průhlednosti, od velmi průhledné až po zcela neprůhlednou. Díky tomu je již možné využít anti-aliasingu a text se tak jeví méně kostrbatý, byť stále v omezené míře, neboť množství úrovní průhlednosti je značně omezené. Takto vykreslený text se tak bude zobrazovat poměrně hladce na jednolitém pozadí, ale v případě složitějších obrazců na pozadí již bude omezení osmibitového prostoru poměrně znát. (20)

Poslední, nejkvalitnější úroveň *blended* oproti tomu operuje ve 32-bitovém ARGB prostoru, každému pixelu tak náleží 1 byte pro každou ze tří základních barev, přičemž poslední byte určující tzv. alfakanál náleží míře průhlednosti daného pixelu. Jen v případě alfakanálu tak máme k dispozici celkem 256 hodnot průhlednosti. Setkáváme se zde tak s vysokou úrovní anti-aliasingu a velmi hladkou podobou obrazového výstupu. Výpočetní výkon nutný pro generování takového prostoru je však o něco vyšší. (20)

Dalším hlediskem, ze kterého lze metody vykreslování dělit, je způsob kódování znaků. SDL_TTF podporuje celkem tři sady znaků, a to LATIN1, UTF-8 a UNICODE.

V případě kódování *LATIN1* jsou znaky vykreslovací funkci předávány jako datový typ *const char*. Sada znaků *LATIN1* obsahuje celkem 64 znaků, zahrnujících zejména základní malá a velká písmena a omezenou sadu speciálních znaků. (21)

V případě kódování *UTF-8* jsou znaky vykreslovací funkci předávány taktéž jako datový typ *const char*. Sada znaků *UTF-8* je oproti sadě *LATIN1* rozsáhlejší. Obsahuje celkem 256 základních znaků zahrnujících navíc také například některá písmena s diakritikou. (22)

V případě kódování *UNICODE* je však situace značně odlišná. Znaky jsou zde předávány jako 16-bitový unsigned integer (*Uint16*). Sada znaků *Unicode* je již ze své podstaty značně odlišná, neboť tato databáze obsahující široké množství nejen standardních znaků, ale například také ideogramů zvaných *Emoji*, se neustále rozrůstá. Ve své současné verzi 13.0 zahrnuje standard *Unicode* až 143 859 znaků. (23) Škála znaků použitelných v rámci

šestnáctibitové hodnoty (formát označovaný také jako *UTF-16*) je však pochopitelně omezená. Důležité však je, že sada znaků obsažená v *UTF-16* je jako jediná ze zmíněných dostatečně veliká na pokrytí základních znaků českého jazyka, a proto ji budeme používat.

TTF_Init()

datový typ: integer;

Obdobně jako funkce Img_Init() v případě doplňku SDL_IMG, slouží tato funkce k inicializaci jistého doplňku knihoven SDL. Tentokrát se však jedná o doplněk SDL_TTF, který přidává podporu písem typu TrueType. Funkce vrací hodnotu 0 v případě úspěchu, případně hodnotu -1 v případě výskytu jakékoliv chyby. (24)

TTF_OpenFont()

datový typ: TTF_Font*; parametry: const char* file, int ptsize)

Tato funkce slouží k načtení písma do paměti. Parametry jsou: cesta ke zdrojovému souboru písma TrueType a požadovaná velikost písma.

TTF_CloseFont()

datový typ: void; parametry: TTF_Font* font

Tato funkce slouží k uvolnění písma z paměti. Jediným parametrem je ukazatel na písmo instance TTF_Font, které si přejeme z paměti uvolnit.

TTF_RenderUNICODE_Solid()

datový typ: SDL_Surface*; parametry: TTF_Font* font, const Uint16* text, SDL_Color fg

Funkce slouží k vykreslení textu v kódování UNICODE do prostoru SDL_Surface metodou *Solid*. Parametry jsou: písmo, kterým má být text vykreslen, ukazatel na textový řetězec a barva písma.

TTF_RenderUNICODE_Shaded()

datový typ: SDL_Surface*; parametry: TTF_Font* font, const Uint16* text, SDL_Color fg

Funkce slouží k vykreslení textu v kódování UNICODE do prostoru SDL_Surface metodou *Shaded*. Parametry jsou: písmo, kterým má být text vykreslen, ukazatel na textový řetězec a barva písma.

TTF_RenderUNICODE_Blended()

datový typ: SDL_Surface*; parametry: TTF_Font* font, const Uint16* text, SDL_Color fg

Funkce slouží k vykreslení textu v kódování UNICODE do prostoru SDL_Surface metodou *Blended*. Parametry jsou: písmo, kterým má být text vykreslen, ukazatel na textový řetězec a barva písma.

6.1.2 Obrazové funkce SDL_IMG

SDL_Image je dodatková knihovna knihoven SDL umožňující programátorovi používat v rámci své aplikace většinu obrazových formátů. Představuje tak základní rozhraní pro podporu a čtení obrázků různých typů, jejich konverzi a vykreslení do scény prostřednictvím SDL 2.0. (25)

IMG_Init()

datový typ: integer; parametry: int flags

Tato funkce slouží k inicializaci doplňku SDL_IMG. Parametr *flags* slouží k určení konkrétního z těchto formátů, jehož podpora je vyžadována. Funkci je možné volat několikrát, s ohledem na to, kolik různých formátů je zapotřebí načítat. Funkce vrací hodnotu 0 v případě úspěchu, v opačném případě pak navrátí kód chyby. (25)

IMG_Load()

datový typ: SDL_Surface*; parametry: const char* file

Funkce načte ze zadané cesty soubor a převede jej na prostor pixelů SDL_Surface. V případě úspěšného načtení souboru ze zadané cesty funkce vrací ukazatel právě na tento pixelový prostor. V případě chyby, která může být zapříčiněna například chybnou cestou k souboru či neprovedením inicializace doplňku s potřebnými parametry, vrátí funkce hodnotu null.

6.1.3 Zvukové funkce SDL_Mixer

Doplňkové knihovny SDL_Mixer přidávají implementaci přehrávání zvuků.

Mix_Init()

datový typ: integer; parametry: int flags

Funkce zprostředkovává inicializaci doplňku při daných nastaveních prostřednictvím parametru flags. Hodnota parametru odpovídá typu zvukových souborů, který má být aplikací podporován. Typickou hodnotou je například MIX_INIT_MP3 pro podporu přehrávání souborů .mp3.

Mix_OpenAudio()

datový typ: integer; parametry: int frequency, Uint16 format, int channels, int chunksize

Tato funkce slouží k inicializaci přehrávání zvukových souborů. Mezi jeho parametry patří vzorkovací frekvence, která má být u zvukového výstupu použita, počet kanálů (2 pro stereo, 1 pro mono) a velikost zvukového bloku v bytech. Standardní zvukovou frekvencí je 22050 Hz, pro její nastavení lze použít také předdefinovanou hodnotu MIX_DEFAULT_FREQUENCY. Pro zlepšení zvukové kvality v případě komplexnějších zvukových výstupů (např. hudby) se však doporučuje použít hodnotu 44100 Hz, která odpovídá vzorkovací frekvenci hudebního CD. S tím však pochopitelně vzroste požadavek na výpočetní výkon. (26)

Mix_LoadWAV

datový typ: Mix_Chunk; parametry: char* file

Funkce, jak název napovídá, slouží k nahrání zvukového souboru .wav do paměti. Jejím jediným argumentem je cesta k tomuto souboru.

7 Vývoj knihoven pro zobrazení drážních vozidel

V následující části se již plně zaměřím na vývoj generických knihoven pro zobrazení drážních vozidel. Nejprve popíši základní funkce obsažené v grafických knihovnách SDL2.0, kterých budu využívat jak při tvorbě samotných generických knihoven pro provozní zobrazení kolejových vozidel, tak například k vytvoření grafického okna. Právě v tomto okně se posléze budou definované prvky vykreslovat. Postupně popíši veškeré z těchto funkcí a na ukázkách kódu a jejich grafickém výstupu popíši jejich logiku a funkce. Uvedený text v této kapitole, společně s následující kapitolou 8, bude sloužit uživatelům knihovny jako stručný návod pro použití a tvorbu aplikací.

7.1 Vývoj funkcí pro zobrazení kolejových vozidel

V předchozí kapitole jsem popsal základní funkce z knihovny SDL, kterých budu v rámci nových grafických funkcí specificky určených pro zobrazení kolejových vozidel hojně využívat. Cílem těchto nově vzniklých funkcí, které jsou hlavním předmětem této kapitoly, však není pouze definovat metody specifických zobrazení pro kolejová vozidla, ale také značně zjednodušit syntaxi jejich tvorby. Proto zde prvně uvedu funkce více obecného charakteru, jako je např. kreslení geometrie, obrázků či textu do scény. Následně pak popíši funkce, které jsou svým charakterem zaměřené na specifické zobrazení kolejových vozidel.

7.1.1 Vykreslení geometrie

Níže popsané funkce slouží k vykreslení základních geometrických obrazců:

Rectangle()

datový typ: void; parametry: různé

Rectangle() je funkce určená k vykreslení obdélníku s výplní do scény. Uvnitř knihoven je dostupná ve dvou přetíženích:

```
void Rectangle(SDL_Color color, int posX, int posY, int width, int height)
{
	SDL_Rect a = { posX, posY, width, height };
	SDL_SetRenderDrawColor(gRenderer, color.r, color.g, color.b, color.a);
	SDL_RenderFillRect(gRenderer, &a);
}
```

Ukázka kódu 3 - přetížení 1 funkce Rectangle()

V případě prvního přetížení jsou vstupními argumenty: barva, počáteční souřadnice x, počáteční souřadnice y, šířka a výška požadovaného obdélníku. Funkce se automaticky stará o přiřazení barvy rendereru a vytvoření instance typu SDL_Rect, která je nezbytným parametrem pro použití funkce SDL_RenderFillRect().

```
void Rectangle(SDL_Color color, SDL_Rect rectangle)
{
	SDL_SetRenderDrawColor(gRenderer, color.r, color.g, color.b, color.a);
	SDL_RenderFillRect(gRenderer, &rectangle);
}
```

Ukázka kódu 4 - přetížení 2 funkce Rectangle()

V případě druhého přetížení je funkce upravená tak, aby byla schopná brát jako vstupní argument již dříve popsanou obdélníkovou strukturu SDL_Rect. To je vhodné k použití například v situacích, kdy si uživatel přeje obdélníkem vyplnit či podkreslit konkrétní globálně definované pole mřížky, nebo si naopak nepřeje při volání funkce vyplňovat dílčí položky této struktury.

Příkladem může být vykreslení červeného čtverce na pozici 100; 300 o délce stran 50 px. Volání funkce pak vypadá následovně:

```
Rectangle(red, 100, 300, 50, 50);
```

Ukázka kódu 5 - volání funkce Rectangle()

Grafický výstup funkce můžeme vidět níže na Obr. 10.

Obr. 10 - Ukázka grafického výstupu funkce Rectangle()

zdroj: vlastní zpracování

StrokedRectangle()

datový typ: void; parametry: SDL_Color outer, inner, uint8_t StrokeSize, SDL_Rect rectangle

Funkce StrokedRectangle() slouží k vykreslení obdélníku s ohraničením. Parametry jsou: barva výplně, barva ohraničení, šířka ohraničení a instance obdélníku SDL_Rect.

```
void StrokedRectangle(SDL_Color outer, SDL_Color inner, uint8_t StrokeSize, SDL_Rect
rectangle)
{
	SDL_Rect a = { rectangle.x, rectangle.y, rectangle.w, rectangle.h };
	SDL_Rect b = { rectangle.x + StrokeSize, rectangle.y + StrokeSize,
rectangle.w - (2 * StrokeSize), rectangle.h - (2 * StrokeSize) };
	SDL_SetRenderDrawColor(gRenderer, outer.r, outer.g, outer.b, outer.a);
	SDL_RenderFillRect(gRenderer, &a);
	SDL_SetRenderDrawColor(gRenderer, inner.r, inner.g, inner.b, inner.a);
	SDL_RenderFillRect(gRenderer, &b);
}
```

Ukázka kódu 6 - tělo funkce StrokedRectangle()

Uvnitř funkce se odehrává vytvoření dvou obdélníků – vnějšího a vnitřního na základě vstupních parametrů tak, aby bylo docíleno zobrazení ohraničení a výplně požadovaného obdélníku. Díky překrytí obou obdélníků není možné použít funkci pouze k vykreslení ohraničení, a proto funkce není vhodná ke kreslení obdélníků s průhledností. K tomu ostatně funkce ani není určená a možnosti vykreslení takového zobrazení popíši později v rámci jiných funkcí. Tato funkce je vhodná výhradně k použití v situacích, kdy je u obdélníku požadována specifická barva výplně s odlišnou barvou ohraničení. Příkladem takové situace může být například tvorba jednoduchého tlačítka.

Níže lze vidět podobu volání funkce v případě, že si uživatel přeje vykreslit modrý obdélník se žlutým ohraničením o rozměrech stran 100 px a 30 px a šířkou ohraničení 5 px:

```
StrokedRectangle(yellow, blue, 5, { 100, 300, 100, 30 });
```

Ukázka kódu 7 - volání funkce StrokedRectangle()

Grafický výstup takto volané funkce je možné vidět na Obr. 11.

Obr. 11 - Ukázka grafického výstupu funkce StrokedRectangle()

zdroj: vlastní zpracování

Frame()

datový typ: void; parametry: SDL_Color color, int posX, posY, width, height, thickness

Tato funkce slouží k vykreslení samotného ohraničení bez výplně. Vstupními parametry jsou: barva ohraničení, počáteční pozice, šířka a výška pomyslného ohraničeného obdélníku a tloušťka ohraničení.

```
void Frame(SDL_Color color, int posX, int posY, int width, int height, int
thickness)
{
	SDL_Rect a = { posX, posY, thickness, height };
	SDL_Rect b = { posX, posY, width, thickness };
	SDL_Rect c = { posX + width, posY, thickness, height };
	SDL_Rect d = { posX, posY + height, width + thickness, thickness };
	SDL_SetRenderDrawColor(gRenderer, color.r, color.g, color.b, 0xFF);
	SDL_RenderFillRect(gRenderer, &a);
	SDL_RenderFillRect(gRenderer, &b);
	SDL_RenderFillRect(gRenderer, &c);
	SDL_RenderFillRect(gRenderer, &d);
}
```

Ukázka kódu 8 - tělo funkce Frame()

Uvnitř funkce dochází k vytvoření celkem čtyřech obdélníků, které představují jednotlivé strany ohraničení. Následně je rendereru přiřazena barva a dochází k jejich vykreslení do scény. Použití této funkce je vhodné zejména v případě, kdy uživatel požaduje plochu uvnitř ohraničení nevyplňovat a zachovat ji zcela průhlednou. Takového zobrazení lze využít například v případě vykreslení zpětné vazby dotykové obrazovky při stisknutí určitého ovládacího prvku, jako je např. tlačítko či interaktivní osa.

Funkci budu demonstrovat například na již dříve vytvořeném červeném čtverci, přičemž ohraničení jsem umístil tak, aby tento čtverec překrývalo pouze částečně. Pro ohraničení jsem zvolil oranžovou barvu a jeho tloušťku nastavil na 5 px: (Obr. 12)

```
Rectangle(red, 100, 200, 50, 50);
Frame(orange, 125, 225, 50, 50, 5);
```

Ukázka kódu 9 - volání funkce Rectangle() a Frame()

Obr. 12 - Ukázka grafického výstupu funkce Rectangle() a Frame()

zdroj: vlastní zpracování

SDL_RenderDrawCircle()

datový typ: void; parametry: SDL_Renderer* renderer, int midpointX, midpointY, radius

Dle názvu funkce by se mohlo zdát, že se jedná o nativní funkci knihovny SDL. Tak tomu však není, ve skutečnosti se jedná o mnou vytvořenou funkci. Vzhledem však k tomu, že knihovny SDL zcela postrádají nativní možnosti vykreslování kruhu či elipsy, a tyto možnosti byly vyhodnoceny za natolik zásadní, že bez jejich existence by bylo velmi komplikované se obejít, bylo rozhodnuto využít názvosloví knihoven SDL pro jejich základní definici a implementaci do základního souboru funkcí. Funkce SDL_RenderDrawCircle() slouží k vykreslení ohraničení kruhu s následujícími parametry: Renderer, který se má pro vykreslení použít, souřadnice x a y středu a poloměr tohoto kruhu. Tělo funkce má následující podobu:

```
void SDL_RenderDrawCircle(SDL_Renderer* renderer, int midpointX, int midpointY, int
radius)
{
int oX, oY, d;
int sWitch;

oX = 0;
oY = radius;
d = radius - 1;
sWitch = 0;

while (oY >= oX) {
	sWitch += SDL_RenderDrawPoint(renderer, midpointX + oX, midpointY + oY);
	sWitch += SDL_RenderDrawPoint(renderer, midpointX + oY, midpointY + oX);
	sWitch += SDL_RenderDrawPoint(renderer, midpointX - oX, midpointY + oY);
	sWitch += SDL_RenderDrawPoint(renderer, midpointX - oY, midpointY + oX);
	sWitch += SDL_RenderDrawPoint(renderer, midpointX + oX, midpointY - oY);
	sWitch += SDL_RenderDrawPoint(renderer, midpointX + oY, midpointY - oX);
	sWitch += SDL_RenderDrawPoint(renderer, midpointX - oX, midpointY - oY);
	sWitch += SDL_RenderDrawPoint(renderer, midpointX - oY, midpointY - oX);

	if (sWitch < 0) {
		sWitch = -1;
		break;
	}

	if (d >= 2 * oX) {
		d -= 2 * oX + 1;
		oX += 1;
	}
	else if (d < 2 * (radius - oY)) {
		d += 2 * oY - 1;
		oY -= 1;
	}
	else {
		d += 2 * (oY - oX - 1);
		oY -= 1;
		oX += 1;
	}
   }
}
```

Ukázka kódu 10 - tělo funkce SDL_RenderDrawCircle()

SDL_RenderFillCircle()

datový typ: void; parametry: SDL_Renderer* renderer, int midpointX, midpointY, radius

Podobně jako v případě funkce SDL_RenderDrawCircle() se taktéž jedná o mnou vytvořenou funkci. Na rozdíl od předchozí funkce tato funkce nevykresluje křivku pouze po obvodu kruhu, ale celou výplň daného kruhu. Parametry jsou: Vykreslující renderer, souřadnice x a y středu a poloměr kruhu. Tělo funkce má následující podobu:

```
void SDL_RenderDrawCircle(SDL_Renderer* renderer, int midpointX, int midpointY, int
radius)
{
	int oX, oY, d;
	int sWitch;

	oX = 0;
	oY = radius;
	d = radius - 1;
	sWitch = 0;

	while (oY >= oX) {
	   sWitch += SDL_RenderDrawPoint(renderer, midpointX + oX, midpointY + oY);
	   sWitch += SDL_RenderDrawPoint(renderer, midpointX + oY, midpointY + oX);
	   sWitch += SDL_RenderDrawPoint(renderer, midpointX - oX, midpointY + oY);
	   sWitch += SDL_RenderDrawPoint(renderer, midpointX - oY, midpointY + oX);
	   sWitch += SDL_RenderDrawPoint(renderer, midpointX + oX, midpointY - oY);
	   sWitch += SDL_RenderDrawPoint(renderer, midpointX + oY, midpointY - oX);
	   sWitch += SDL_RenderDrawPoint(renderer, midpointX - oX, midpointY - oY);
	   sWitch += SDL_RenderDrawPoint(renderer, midpointX - oY, midpointY - oX);

	   if (sWitch < 0) {
		sWitch = -1;
		break;
	   }

	   if (d >= 2 * oX) {
		d -= 2 * oX + 1;
		oX += 1;
	   }
	   else if (d < 2 * (radius - oY)) {
		d += 2 * oY - 1;
		oY -= 1;
	   }
	   else {
		d += 2 * (oY - oX - 1);
		oY -= 1;
		oX += 1;
	   }
	}
}
```

Ukázka kódu 11 - tělo funkce SDL_RenderFillCircle()

Circle()

datový typ: void; parametry: int midpointX, midpointY, thickness, radius, SDL_Color color, bool fill

Funkce Circle() představuje jakési zapouzdření předchozích dvou funkcí tak, aby nebylo nutné je volat napřímo a došlo k usnadnění syntaxe například v případě, kdy uživatel požaduje vykreslení křivky po obvodu kruhu o určité tloušťce. Funkce je nově koncipována tak, aby s její pomocí bylo možné jednoduše kreslit jak prázdný, tak plný kruh. Argumenty jsou: souřadnice středu, tloušťka obvodu, poloměr kruhu, barva kruhu a přepínač plného či prázdného kruhu. Struktura funkce je následující:

```
void Circle(int midpointX, int midpointY, int thickness, int radius, SDL_Color
color, bool fill)
{
	SDL_SetRenderDrawColor(gRenderer, color.r, color.g, color.b, color.a);
	if (fill)
	{
		SDL_RenderFillCircle(gRenderer, midpointX, midpointY, radius);
	}
	else
	{
		for (int i = 0; i < thickness; i++)
		{
		    SDL_RenderDrawCircle(gRenderer, midpointX, midpointY, radius - i);
		}
	}
}
```

Funkce je tak univerzálně použitelná pro širší škálu využití. Své uplatnění v případě provozních zobrazení kolejových vozidel může najít například v případě kruhového ukazatele rychlosti, jehož implementaci budu demonstrovat později. Níže je možné vidět příklad, kdy pomocí této jediné funkce vykreslím vedle sebe dva červené kruhy s poloměrem 30 px, první prázdný s obvodem o tloušťce 5 px, druhý plný:

```
Circle(450, 100, 5, 30, red, false);
Circle(520, 100, 0, 30, red,  true);
```

Ukázka kódu 12 - volání funkce Circle()

Grafický výstup takto volané funkce můžeme vidět na Obr. 13.

Obr. 13 - Ukázka grafického výstupu funkce Circle()

zdroj: vlastní zpracování

7.1.2 Vykreslení textů

Text()

datový typ: void; parametry: wstring text, TTF_Font* font, int posX, posY, SDL_Color color

V kapitole 6.1.1 jsem uvedl, že vykreslení textu do scény v českém jazyce není ani s využitím knihoven SDL příliš jednoduchou záležitostí. Za tímto účelem byla vyvinuta funkce Text(). Složitost řešení nutného pro takovou akci je možné demonstrovat přímo na obsahu těla této funkce:

```
void Text(std::wstring text, TTF_Font* font, int posX, int posY, SDL_Color color)
{
	typedef std::basic_string<Uint16, std::char_traits<Uint16>,
std::allocator<Uint16> > u16string;
	std::wstring a = text;
	u16string b(a.begin(), a.end());
	SDL_Surface* c = TTF_RenderUNICODE_Blended(font, b.c_str(), color);
	SDL_Texture* d = SDL_CreateTextureFromSurface(gRenderer, c);
	int TxW, TxH;
	SDL_QueryTexture(d, NULL, NULL, &TxW, &TxH);
	SDL_Rect textrect = { posX, posY, TxW, TxH };
	SDL_RenderCopy(gRenderer, d, NULL, &textrect);
	SDL_FreeSurface(c);
	SDL_DestroyTexture(d);
}
```

Ukázka kódu 13 - tělo funkce Text()

V rámci aplikace používám v C++ pro definici textových řetězců datový typ std::wstring. Tento datový typ odpovídá formátu UTF-16 a je jedním z mála, který uživateli umožní kompilovat textové řetězce s českými znaky. Po předání tohoto řetězce funkci je třeba nejdříve definovat nový specifický datový typ, který jsem nazval u16string. Tento typ se posléze naplní předaným řetězcem. To se ukázalo být jediným možným řešením, jak předat tento řetězec funkci TTF_RenderUNICODE_Blended(). Je až s podivem, z jakého důvodu je v rámci SDL takto složitá konverze nutností a proč vykreslovací funkce neumí nativně pracovat s pro C++ zcela běžnými formáty. Použitím funkce TTF_RenderUNICODE_Blended() získám pixelový prostor SDL_Surface, který je následně třeba převést na typ SDL_Texture. Tím však není všem nutným úkonům konec. SDL potřebuje pro vykreslení získané textury do scény znát také obdélník, do kterého se má daný text vykreslit. Opět zůstává pouze s podivem, proč není výchozím předpokladem, že text má být vykreslen ve standardních rozměrech bez nutnosti změnit nepoměrově jeho dimenze tak, aby odpovídal rozměrům předem definovaného obdélníku. Díky této skutečnosti je třeba užít také funkce SDL_QueryTexture, která naplní stanovené proměnné originálními rozměry textury. Teprve po získání těchto rozměrů lze vyplnit obdélník textem tak, aby byly jeho rozměry zachovány a nedocházelo k jeho deformaci. K vykreslení textu nakonec použiji funkci SDL_RenderCopy, která přesune texturu do renderované scény.

Dalšími parametry kromě vstupního řetězce, je pochopitelně také písmo, kterým se má text vykreslit, počáteční souřadnice x a y vykreslovaného textu a jeho barva. Výše uvedený postup má jeden nedostatek, a to sice, že text lze touto metodou vykreslovat pouze zarovnaný doleva, respektive směrem doprava dolů od počáteční souřadnice. Za účelem možnosti zarovnání textu na střed či doleva od definovaného bodu jsem vytvořil také modifikované přetížení funkce Text(). Variace funkce podporující metody zarovnání textu vypadá následovně:

```
void Text(std::wstring text, TTF_Font* font, int posX, int posY, SDL_Color color,
byte alignment)
{
	typedef std::basic_string<Uint16, std::char_traits<Uint16>,
std::allocator<Uint16> > u16string;
	std::wstring a = text;
	u16string b(a.begin(), a.end());
	SDL_Surface* c = TTF_RenderUNICODE_Blended(font, b.c_str(), color);
	SDL_Texture* d = SDL_CreateTextureFromSurface(gRenderer, c);
	int TxW, TxH;
	SDL_QueryTexture(d, NULL, NULL, &TxW, &TxH);
	SDL_Rect textrect;
	if (alignment == 0)
	{
		textrect = { posX, posY, TxW, TxH };
	}
	else if (alignment == 1)
	{
		textrect = { posX - (TxW / 2), posY - (TxH / 2), TxW, TxH };
	}
	else if (alignment == 2)
	{
		textrect = { posX - TxW, posY, TxW, TxH };
	}
	SDL_RenderCopy(gRenderer, d, NULL, &textrect);
	SDL_FreeSurface(c);
	SDL_DestroyTexture(d);
}
```

Ukázka kódu 14 - přetížení funkce Text() pro implementaci zarovnání textu

Je možné si povšimnout, že parametry funkce byly rozšířeny o proměnnou alignment. Její hodnota 0 odpovídá standardnímu zarovnání doleva, hodnota 1 zarovnání na střed a hodnota 2 zarovnání doprava. Příklad vykreslení pomocí funkce do středu obrazovky včetně grafického výstupu lze vidět níže: (Obr. 14)

```
Text(L"Fakulta dopravní ČVUT v Praze", Arial_24, SCREEN_WIDTH / 2, 18, white, 1);
```

Ukázka kódu 15 - ukázka použití funkce Text()

Obr. 14 - Ukázka grafického výstupu funkce Text()

zdroj: vlastní zpracování

TextFill()

datový typ: void; parametry: SDL_Rect rectangle, wstring text, TTF_Font* font, SDL_Color color

Tato funkce slouží k vyplnění specifického obdélníku textem. Její použití je vhodné například ke tvorbě tlačítek s popiskem. Parametry jsou: objekt rectangle, tedy definovaný obdélník, znakový řetězec, kterým se má dané pole vyplnit, font a barva písma.

Funkce je definována následovně:

```
void TextFill(SDL_Rect rect, std::wstring text, TTF_Font* font, SDL_Color color)
{
	Text(text, font, rect.x + (rect.w / 2), rect.y + (rect.h / 2), color, 1);
}
```

Ukázka kódu 16 - definice funkce TextFill()

Je možné si všimnout, že funkce pouze vhodně využívá již dříve uvedenou funkci Text(). Pro možnost výplně obdélníku dvěma řádky textu bylo vytvořeno také následující přetížení podporující dva vstupní texty pro každý řádek:

```
void TextFill(SDL_Rect rect, std::wstring text_upper, std::wstring text_lower, TTF_Font* font,
SDL_Color color)
{
    Text(text_upper, font, rect.x + (rect.w / 2), rect.y + (rect.h / 4), color, 1);
    Text(text_lower, font, rect.x + (rect.w / 2), rect.y + (rect.h / 2) + (rect.h / 4), color, 1);
}
```

Ukázka kódu 17 - přetížení funkce TextFill()

Vytvoření obdélníku s popisem uvnitř je díky těmto funkcím posléze poměrně jednoduchou a intuitivní záležitostí:

```
SDL_Rect testRect = { 50, 50, 150, 50 };
StrokedRectangle(white, ddblue, 2, testRect);
TextFill(testRect, L"text uvnitř", ArialBold_18, white);
```

Ukázka kódu 18 - příklad použití funkce TextFill()

Výše si lze všimnout, že prvně dojde k vytvoření obdélníku s určitými rozměry, následně je pomocí již známé funkce StrokedRectangle() tento obdélník vykreslen včetně jeho ohraničení, a prostřednictvím funkce TextFill() je následně vyplněn textem o libovolných parametrech. Výsledný grafický výstup lze vidět na Obr. 15.

Obr. 15 - Grafický výstup ukázky funkce TextFill()

zdroj: vlastní zpracování

SizeText()

datový typ: Size; parametry: wstring text, TTF_Font* font

Tato funkce slouží zejména k analytickým účelům. Nejen pro její účely jsem nadefinoval nový datový typ – strukturu Size, obsahující dvě hodnoty odpovídající šířce a výšce daného objektu.

```
struct Size
{
	int w;
	int h;
};
```

Ukázka kódu 19 - definice struktury Size

Účelem této funkce je vrátit rozměry požadovaného textu, respektive šířky a výšky vykresleného textového řetězce v pixelech. Její definice je následující:

```
Size SizeText(std::wstring text, TTF_Font* font)
{
	typedef std::basic_string<Uint16, std::char_traits<Uint16>,
	std::allocator<Uint16> > u16string;
	std::wstring a = text;
	u16string b(a.begin(), a.end());
	SDL_Surface* c = TTF_RenderUNICODE_Blended(font, b.c_str(), white);
	SDL_Texture* d = SDL_CreateTextureFromSurface(gRenderer, c);
	int TxW, TxH;
	SDL_QueryTexture(d, NULL, NULL, &TxW, &TxH);
	Size size = { TxW, TxH };
	return size;
}
```

Ukázka kódu 20 - tělo funkce SizeText()

Je možné zpozorovat, že tělo této funkce je značně podobné části těla funkce Text(), konkrétně pak části, kde probíhá právě zjištění výchozích rozměrů vykreslovaného textu. Funkce může sloužit například ke zjištění, zda se požadovaný text vejde do oblasti, do které jej uživatel zamýšlí vykreslit. Jejími parametry jsou: znakový řetězec a font daného textu.

7.1.3 Vykreslení obrázků

Pro vykreslení obrázků do scény, kterými můžou být například různé piktogramy či ikony, byla taktéž vytvořena specifická sada funkcí. Tyto funkce využívají především rozšíření SDL_Img a základních funkcí knihoven SDL.

Img()

datový typ: void; parametry: SDL_Texture* texture, int posX, posY, width height

Tato jednoduchá funkce předpokládá, že již došlo prostřednictvím funkce IMG_Load() nebo IMG_LoadTexture() k vytvoření textury z daného zdrojového obrazového souboru. Následně stačí funkci formou argumentů předat název této textury, pozici a rozměry textury k vykreslení. Definice vypadá následovně:

```
void Img(SDL_Texture* texture, SDL_Rect rectangle)
{
	SDL_RenderCopy(gRenderer, texture, NULL, &rectangle);
}
```

Ukázka kódu 21 - základní definice funkce Img()

O něco komplexnější pak je přetížení této funkce, které umožňuje zadat navíc také požadované rozměry obrázku. Základní funkce předpokládá, že tyto rozměry budou dodrženy původní a nebude docházet k roztažení či zmenšení textury.

```
void Img(SDL_Texture* texture, SDL_Rect rectangle, float sizeX, float sizeY)
{
	SDL_Rect img_rect;
	img_rect.x = rectangle.x + (rectangle.w / 2) - sizeX / 2;
	img_rect.y = rectangle.y + (rectangle.h / 2) - sizeY / 2;
	img_rect.w = sizeX;
	img_rect.h = sizeY;
	SDL_RenderCopy(gRenderer, texture, NULL, &img_rect);
}
```

Ukázka kódu 22 - přetížení funkce Img() jejím rozšířením pro podporu změny velikosti textury

Pro potřeby demonstrace níže uvedu definici textury settings obsahující obrázek „settings.png“ získaný z platformy FlatIcon (27) a jeho vykreslení do scény:

```
SDL_Texture* settings = IMG_LoadTexture(gRenderer, ".../settings.png");
Img(settings, 400, 350, 64, 64);
```

Ukázka kódu 23 - příklad použití funkce Img()

V případě reálné aplikace se pochopitelně předpokládá, že inicializace textury prostřednictvím IMG_LoadTexture() proběhne již na začátku běhu programu. Při programování specifické obrazovky tak zpravidla používám pouze funkci Img(). Výše uvedený příklad tak má ryze ilustrativní charakter. Grafický výstup níže uvedeného příkladu lze vidět na Obr. 16.

Obr. 16 - Příklad grafického výstupu funkce Img()

zdroj: vlastní zpracování a (27)

7.1.4 Vykreslení tlačítek

S množinou funkcí, kterou jsem dříve nadefinoval, je poměrně snadné vytvořit další funkce, které jich budou komplexně využívat. Takovými funkcemi mohou být například ty, které poslouží k vykreslení tlačítek, která bývají běžnou součástí dotykových obrazovek řídicího pultu kolejových vozidel.

TextButton()

datový typ: void; parametry: SDL_Rect rectangle, SDL_Color strokeColor_outer, strokeColor_inner, uint8_t strokeSize, wstring text, TTF_Font* font, SDL_Color textColor

Funkce TextButton() vykreslí do scény základní tlačítko s textovým popiskem. To se skládá z obdélníku s ohraničením a vnitřního textu. Mezi parametry tedy náleží definice obdélníku pro stanovení rozměrů, barva ohraničení a výplně a šířka ohraničení tlačítka. Dále vstupní textový řetězec, font, kterým má být text vykreslen a barva tohoto textu.

Definice:

```
void TextButton(SDL_Rect rectangle, SDL_Color strokeColor_outer, SDL_Color
strokeColor_inner, uint8_t strokeSize, wstring text, TTF_Font* font, SDL_Color
textColor)
{
     StrokedRectangle(strokeColor_outer, strokeColor_inner, strokeSize, rectangle);
     TextFill(rectangle, text, font, textColor);
}
```

Ukázka kódu 24 - definice funkce TextButton()

Ukázku volání funkce a její grafický výstup můžeme vidět níže na Obr. 17.

```
TextButton(button1_rect, lgrey, ddblue, 3, L"Tlačítko", Arial_18, lgrey);
```

Ukázka kódu 25 - ukázkové volání funkce TextButton()

Obr. 17 - Příklad grafického výstupu funkce TextButton()

zdroj: vlastní zpracování

Výše uvedené funkce představují základní metody, kterými lze plnit scénu obrazovky kolejového vozidla triviálními prvky. Některá zobrazení však jsou svým provedením poněkud více specifická a ze strany uživatele tak může nastat nutnost tyto funkce upravit, popřípadě z nich vycházet při tvorbě funkcí zcela nových. V rámci knihoven, jejichž vývoj je tématem práce, jsem definoval alespoň jednu takovou funkci, která je typická například pro zobrazení displeje DMI mezinárodního vlakového zabezpečovače ETCS. Jedná se o tlačítko, které provedením svého ohraničení evokuje prostorový dojem. Funkci vykreslující takové tlačítko jsem pojmenoval _3dButton():

_3dButton()

datový typ: void; parametry: SDL_Rect rectangle, SDL_color strokeColor_light, strokeColor_shade, wstring text, TTF_Font* font, SDL_Color textColor

Funkce _3dButton() vytváří tlačítko s textovým popisem, poskytuje však specifický, trojrozměrný vzhled, který je pro některá zobrazení kolejových vozidel, jako je například obrazovka DMI ETCS, typický. Podoba funkce je následující:

```
void _3dButton(SDL_Rect rectangle, SDL_Color strokeColor_light, SDL_Color strokeColor_shade, wstring text,
TTF_Font* font, SDL_Color textColor)
{
	SDL_SetRenderDrawColor(gRenderer, strokeColor_light.r, strokeColor_light.g, strokeColor_light.b,
strokeColor_light.a);
	SDL_RenderDrawLine(gRenderer, rectangle.x, rectangle.y, rectangle.x + rectangle.w, rectangle.y);
	SDL_RenderDrawLine(gRenderer, rectangle.x + 1, rectangle.y + 1, (rectangle.x + rectangle.w) - 1,
rectangle.y + 1);
	SDL_RenderDrawLine(gRenderer, rectangle.x, rectangle.y, rectangle.x, rectangle.y + rectangle.h);
	SDL_RenderDrawLine(gRenderer, rectangle.x + 1, rectangle.y + 1, rectangle.x + 1, (rectangle.y +
rectangle.h) - 1);
	SDL_SetRenderDrawColor(gRenderer, strokeColor_shade.r, strokeColor_shade.g, strokeColor_shade.b,
strokeColor_shade.a);
	SDL_RenderDrawLine(gRenderer, rectangle.x + rectangle.w, rectangle.y, rectangle.x + rectangle.w,
rectangle.y + rectangle.h);
	SDL_RenderDrawLine(gRenderer, (rectangle.x + rectangle.w) - 1, rectangle.y + 1, (rectangle.x +
rectangle.w) - 1, (rectangle.y + rectangle.h) - 1);
	SDL_RenderDrawLine(gRenderer, rectangle.x, rectangle.y + rectangle.h, rectangle.x + rectangle.w,
rectangle.y + rectangle.h);
	SDL_RenderDrawLine(gRenderer, rectangle.x + 1, (rectangle.y + rectangle.h) - 1, (rectangle.x +
rectangle.w) - 1, (rectangle.y + rectangle.h) - 1);
	TextFill(rectangle, text, font, textColor);
}
```

Ukázka kódu 26 - definice funkce _3dButton()

Volání funkce pak vypadá podobně, jako v případě tlačítka typu TextButton():

```
_3dButton(button2_rect, DMIblue_light, DMIblue_dark, L"3D tlačítko", Arial_14, lgrey);
```

Ukázka kódu 27 - volání funkce _3dButton()

Grafický výstup:

Obr. 18 - Grafický výstup funkce _3dButton()

zdroj: vlastní zpracování

7.1.5 Vykreslení charakteristických elementů obrazovek kolejových vozidel

S definovanými metodami vykreslování textu, geometrických obrazců, obrázků ze souboru a dalších je následně možné tyto metody využít k vykreslování složitějších obrazců. Vzhledem k tomu, že u různých provozních zobrazení kolejových vozidel se počítá s některými elementy natolik triviálními a společnými pro velkou většinu vozidel, jako je například rychloměr či ukazatel poměrného tahu, definoval jsem za tímto účelem funkce, jejichž cílem je vytvořit základ takového zobrazení.

Speedometer()

datový typ: void; parametry: SDL_Point midpoint, int radius, int circleRadius, float ZeroAngle, int no_BLines, no_TLines, len_Blines, lineThickness, maxSpeed, float currentSpeed, SDL_Color color

Funkce vykreslí do scény základní radiální rychloměr. Vzhledem k její komplexnosti jsem se rozhodl popsat jednotlivé parametry podrobněji:

SDL_Point midpoint

- Bod, který je středem rychloměru a všech jeho elementů

int radius

- poloměr rychloměru v pixelech

int circleRadius

- poloměr středu ukazatele rychlosti v pixelech

float ZeroAngle

- úhel, který svírá počátek a konec osy rychloměru s osou x

int no_BLines

- počet silných dílků osy na rychloměru

int no_TLines

- počet slabých dílků osy mezi každými dvěma silnými dílky

int len_BLines

- délka silných dílků osy. Délka slabých dílků se rovná polovině této délky.

int maxSpeed

- maximum osy rychloměru

float currentSpeed

- aktuální zobrazovaná rychlost

SDL_Color color

- barva elementů rychloměru

Definice funkce vypadá následovně:

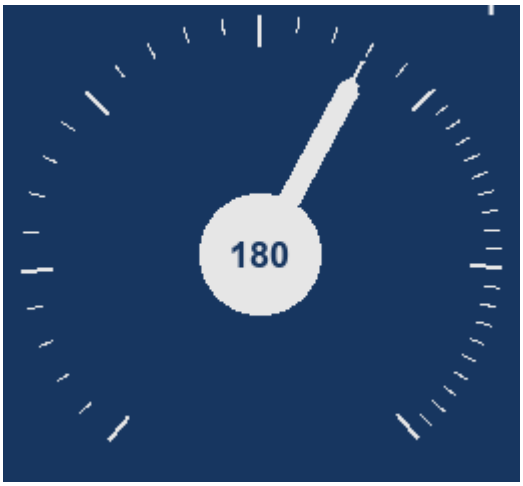
```
void Speedometer(SDL_Point midpoint, int radius, int circleRadius, float ZeroAngle, int
no_BLines, int no_TLines, int len_BLines, int maxSpeed, float currentSpeed, SDL_Color
color)
{
	Circle(midpoint.x, midpoint.y, 30, 30, lgrey, 1);

	SDL_Texture* b = SDL_CreateTexture(gRenderer, SDL_PIXELFORMAT_RGBA8888,
	SDL_TEXTUREACCESS_TARGET, 30, 10);
	SDL_SetRenderTarget(gRenderer, b);
	SetColor(color);
	SDL_RenderClear(gRenderer);
	SDL_SetRenderTarget(gRenderer, NULL);
	SDL_Rect t1 = { (midpoint.x - radius) + 1, midpoint.y + 1, len_BLines, 2 };
	SDL_Rect t2 = { (midpoint.x - radius) + 1, midpoint.y + 1, len_BLines / 2, 1 };
	SDL_Point m2 = { radius, 0 };
	float ZA0 = ZeroAngle;
	float ZA2 = ZeroAngle;
	float ZA3 = ZeroAngle;
	float degrees = ( 360 - (2 * (90 + ZeroAngle)));
	float pcs = ((float)degrees / (float)(no_BLines - 1));
	float s_pcs = pcs / (no_TLines + 1);
	float xs_pcs = s_pcs / 2;

	for (int i = 0; i <= no_BLines - 1; i = i + 1)
	{
		SDL_RenderCopyEx(gRenderer, b, NULL, &t1, ZeroAngle, &m2, SDL_FLIP_NONE);
		ZeroAngle += (pcs);
	}

	for (int j = 0; j <= ((no_BLines - 1) * (no_TLines + 1)); j = j + 1)
	{
		SDL_RenderCopyEx(gRenderer, b, NULL, &t2, ZA2, &m2, SDL_FLIP_NONE);
		ZA2 += s_pcs;
	}

	for (int j = 0; j <= (((no_BLines - 1) * (no_TLines + 1))) * 2; j = j + 1)
	{
		if (ZA3 >= (degrees / ((no_BLines - 1 ) / 3)))
		{
			SDL_RenderCopyEx(gRenderer, b, NULL, &t2, ZA3, &m2, SDL_FLIP_NONE);
		}
		ZA3 += xs_pcs;
	}
	SDL_DestroyTexture(b);

	SDL_Texture* strelka = SDL_CreateTexture(gRenderer, SDL_PIXELFORMAT_RGBA8888,
	SDL_TEXTUREACCESS_TARGET, SCREEN_WIDTH, SCREEN_HEIGHT);
	SDL_SetTextureBlendMode(strelka, SDL_BLENDMODE_BLEND);

	SDL_SetRenderTarget(gRenderer, strelka);

	Arrow(midpoint.x, midpoint.y, 110, 11, 0, lgrey);

	SDL_SetRenderTarget(gRenderer, NULL);

	SDL_RenderCopyEx(gRenderer, strelka, NULL, NULL, ((currentSpeed * (0.5975 -
	(ZA0/10 * 0.0675)))- 180) + ZA0, &midpoint, SDL_FLIP_NONE);
	SDL_DestroyTexture(strelka);

	Text(std::to_wstring((int)currentSpeed), ArialBold_18, midpoint.x, midpoint.y,
	DMIblue, 1);
}
```

Ukázka kódu 28 - definice funkce Speedometer()

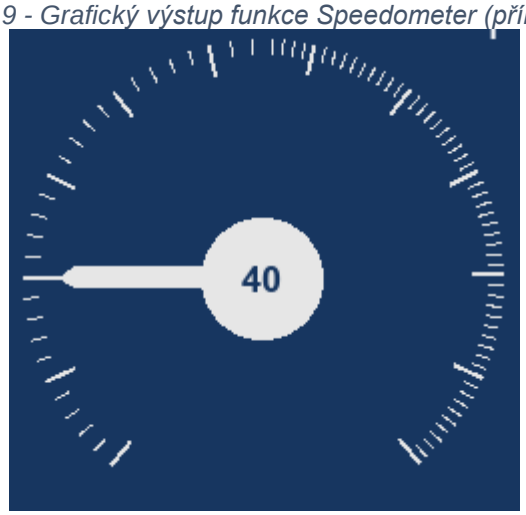
Níže nastíním příklady volání funkce v případě použití různých parametrů a jejich grafických výstupů:

```
Speedometer({ 150, 150 }, 120, 30, -50,  7, 4, 16, 1, 400, 180, lgrey); // Obr. 19
Speedometer({ 150, 150 }, 120, 30, -50, 12, 4, 16, 1, 220,  40, lgrey); // Obr. 20
Speedometer({ 150, 150 }, 120, 30, -50,  6, 5, 16, 3, 100,  20, lgrey); // Obr. 21
```

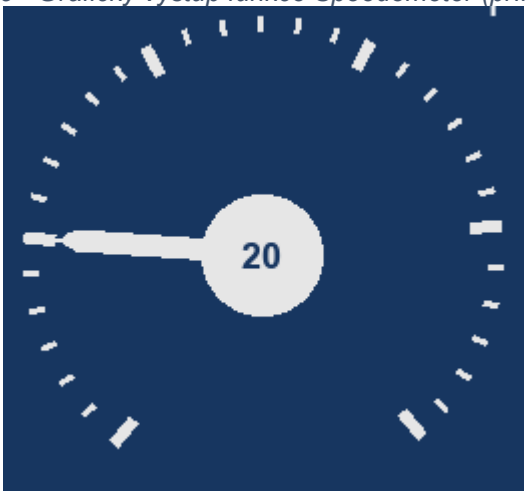
Ukázka kódu 29 - různé parametry při volání funkce Speedometer()



Obr. 19 - Grafický výstup funkce Speedometer (příklad 1)



Obr. 20 - Grafický výstup funkce Speedometer (příklad 2)



Obr. 21 - Grafický výstup funkce Speedometer (příklad 3)
zdroj: vlastní zpracování

RelativeThrust()

datový typ: void, parametry: bool type, uint18_t posX, posY, width, float scale, int16_t value, SDL_Color color_main, SDL_Color color_bar, TTF_Font* font

Běžnou součástí provozních obrazovek kolejových vozidel jsou také různé formy bargrafů a ukazatelů. Tím pravděpodobně nejtypičtějším je ukazatel poměrného tahu. K vykreslení tohoto ukazatele slouží funkce RelativeThrust(). Funkce nabízí zobrazení dvou typů ukazatele – liniového a sloupcového. K tomuto nastavení slouží parametr type. Zatímco hodnota 0 odpovídá ukazateli liniovému, hodnota 1 odpovídá ukazateli sloupcovému. Dalšími parametry jsou: počáteční pozice x a y ukazatele, měřítko, aktuální hodnota poměrného tahu, barva osy ukazatele, barva sloupce poměrného tahu (pouze pro sloupcový typ) a písmo popisků (také pouze pro sloupcový typ). Definice funkce je následující:

```
void RelativeThrust(bool type, uint16_t posX, uint16_t posY, uint16_t width, float scale, int16_t value, SDL_Color
color_main, SDL_Color color_bar, TTF_Font* font)
{
   SDL_SetRenderDrawColor(gRenderer, color_main.r, color_main.g, color_main.b, color_main.a);
   int16_t i = -100;
   float y = 0;
   if (type == 0)
   {
     for (i; i <= 100; i++)
     {
     if (i % 100 == 0)
     {
        SDL_RenderDrawLine(gRenderer, posX, posY+i+100+y, posX+width, posY + i + 100 + y);
         SDL_RenderDrawLine(gRenderer, posX, posY+i+100+y+1, posX + width, posY + i + 100 + y + 1);
         SDL_RenderDrawLine(gRenderer, posX, posY+i+100+y-1, posX + width, posY + i + 100 + y - 1);
     }
     else if (i % 20 == 0)
     {
         SDL_RenderDrawLine(gRenderer, posX+(width/2)+2, posY+i+100+y, posX+(width), posY+i+100+y);
     }
     y += scale;
     }
   SDL_RenderDrawLine(gR,posX+width/2,posY+((i+100+y)/2),posX+width/2,posY+((i+100+y)/2)-(value+value*scale));
   SDL_RenderDrawLine(gR,posX+width/2-1,posY+((i+100+y)/2),posX+width/2-1,posY+((i+100+y)/2)-(value+value*scale));
   }
   else if (type == 1)
     {
     int tW, tH;
     for (i; i <= 100; i++)
     {
     if (i % 100 == 0)
       {
         SDL_RenderDrawLine(gRenderer, posX - 6, posY + i + 100 + y, posX + width, posY + i + 100 + y);
         if (i == 0)
         {
                  TTF_SizeText(font, "0", &tW, &tH);
                  Text(L"0", font, posX + width + 4, posY + 100 + i + y - tW + 1, grey);    }
         }
         else if (i % 20 == 0)
         {
                  SDL_RenderDrawLine(gRenderer, posX - 3, posY + i + 100 + y, posX, posY + i + 100 + y);
         }
         y += scale;
       }
    Rectangle(color_bar, posX, posY + ((i + 100 + y) / 2), width, -(value + value * scale));
    TTF_SizeText(font, "+100%", &tW, &tH);
    Text(L"+100%", font, posX + width - tW + 4, posY - tH, grey);
    TTF_SizeText(font, "-100%", &tW, &tH);
    Text(L"-100%", font, posX + width - tW + 4, posY + i + 100 + y, grey);
    Frame(color_main, posX, posY, width, i + 100 + y - 2, 1);
    }
}
```

Ukázka kódu 30 - Definice funkce RelativeThrust()

Níže jsem uvedl příklady volání této funkce jak pro sloupcový, tak liniový ukazatel:

```
RelativeThrust(0, 600, 20, 30, 0.3, 50, lgrey, cyan, Consolas_12);//liniovy-Obr. 22
RelativeThrust(1, 560, 20, 20, 0.3, 50, lgrey, cyan, Consolas_12);//sloup. -Obr. 23
```

Ukázka kódu 31 - volání funkce RelativeThrust()

Obr. 22 - Liniový ukazatel jakožto výstup funkce RelativeThrust()

zdroj: vlastní zpracování

Obr. 23 - Sloupcový ukazatel jakožto výstup funkce RelativeThrust()

zdroj: vlastní zpracování

8 Vývoj aplikací s využitím knihoven

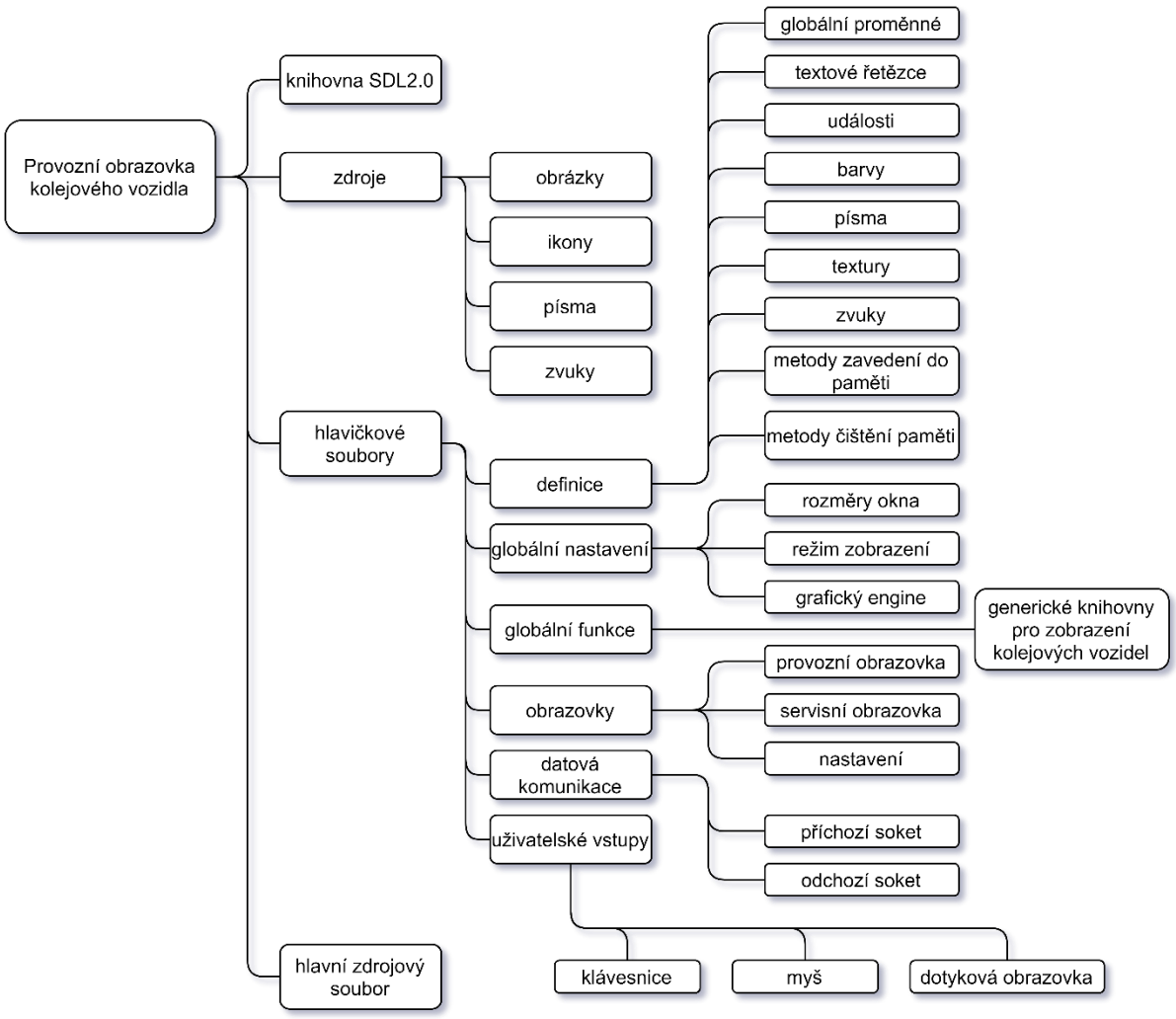
V následující části práce uvedu doporučené postupy pro využití vytvořených funkcí ke tvorbě grafických aplikací pro kolejová vozidla. Aby bylo použití těchto knihoven a funkcí pokud možno co nejefektivnější, popíši nejprve množinu doporučených postupů pro tvorbu struktury souborů a jejich obsahu. Následně popíši také instalaci knihoven do vývojového prostředí Visual Studio. Použití vzniklých funkcí budu následně demonstrovat na několika konkrétních příkladech aplikací využívajících vytvořené knihovny.

8.1 Doporučená struktura systému souborů

Aby byl kód pokud možno co nejpřehlednější a vývoj samotné aplikace nejplynulejší, je vhodné při vývoji aplikace dodržovat určitá pravidla. Zatímco samotné definice jednotlivých proměnných či rozvržení dané scény se mohou zprvu jevit jako poměrně zdlouhavé a nezáživné, pokud již na začátku vývoje tuto činnost nezanedbáme, následné umisťování prvků do scény bude naopak velmi rychlé a intuitivní. Je však nutné zmínit, že uvedené postupy mají ryze doporučující charakter a předpokládá se nutnost jejich přizpůsobení konkrétním požadavkům na danou aplikaci. Za účelem grafické demonstrace navržené struktury jsem vytvořil schéma popisující jednotlivé dílčí části aplikace. Patří sem zejména:

- Zdrojové soubory, jako jsou obrázky, ikony, písma nebo zvuky,
- hlavičkové soubory, kde se nachází veškerý kód, od definic proměnných, mřížek, textur, nastavení obrazu, funkcí pro vykreslování, zpracování dat, datové komunikace, zpracování uživatelského vstupu a další,
- hlavní zdrojový soubor, ve kterém probíhá inicializace a základní smyčka aplikace,
- veškeré knihovny nutné pro správný běh aplikace.

Vytvořené schéma je možné vidět na Obr. 24:



Obr. 24 - Schéma doporučené struktury aplikace.

zdroj: vlastní zpracování

8.1.1 Soubor definic

Hlavičkový soubor s definicemi je určený pro definice globálních proměnných, textových řetězců, událostí, barev, písem, textur, zvuků a dalších. Co je však neméně důležité, obsahuje funkce, které zavedou výše zmíněné do paměti, nebo ji posléze vyčistí.

Součástí souboru tak jsou například globální proměnné, do kterých se ukládají data o uživatelském vstupu, jako je např. pozice myši na obrazovce:

```
/* INPUT VARIABLES */
int MOUSE, MOUSEx, MOUSEy, TOUCHx, TOUCHy;
bool click;
```

Ukázka kódu 32 - definice proměnných pro uživatelský vstup

Následují také definice událostí. Mezi události patří například stisk tlačítka na klávesnici:

```
/* EVENTS */
SDL_Event e;
SDL_KeyboardEvent k;
```

Ukázka kódu 33 - definice proměnných událostí

Zásadní je také inicializace samotného rendereru a dalších klíčových proměnných aplikace

```
/* APPLICATION */
SDL_Renderer* gRenderer = NULL;
bool quit;

/* APP DATA */
int screen_mode = 0;

/* DATA TYPES */
struct Size
{
	int w;
	int h;
};
```

Ukázka kódu 34 - definice globálních proměnných programu

Následují definice barev. Aby bylo možné se vyhnout zdlouhavému vypisování RGBA hodnot při každém volání funkce mající za vstupní parametr barvu, lze barvy definovat v hlavičkovém souboru následujícím způsobem:

```
/* COLORS */
SDL_Color white =   { 255, 255, 255, 0xFF };
SDL_Color yellow =  { 255, 234,   0, 0xFF };
SDL_Color red =     { 255,   0,   0, 0xFF };
SDL_Color green =   {   0, 255,   0, 0xFF };
SDL_Color blue =    {   0,   0, 255, 0xFF };
SDL_Color cyan =    {   0, 255, 255, 0xFF };
SDL_Color black =   {   0,   0,   0, 0xFF };
SDL_Color orange =  { 255, 150,   0, 0xFF };
SDL_Color grey =    { 195, 195, 195, 0xFF };
SDL_Color mgrey =   { 128, 128, 128, 0xFF };
SDL_Color lgrey =   { 230, 230, 230, 0xFF };
SDL_Color dgrey =   {  73,  73,  73, 0xFF };
SDL_Color ddgrey =  {  20,  20,  20, 0xFF };
```

Ukázka kódu 35 - definice barev

Podobná situace je také s písmy. Ty doporučuji nejprve globálně nadefinovat, aby pro ně již bylo vyhrazené místo v paměti. O samotnou inicializaci v podobě načtení písem ze zdrojových souborů do těchto struktur se budu zajímat až během inicializační funkce.

```
/* FONTS */
TTF_Font* Consolas_12 = NULL;
TTF_Font* Consolas_16 = NULL;
TTF_Font* ConsolasBold_16 = NULL;
TTF_Font* Consolas_24 = NULL;
TTF_Font* ConsolasBold_24 = NULL;
TTF_Font* Consolas_36 = NULL;
TTF_Font* ConsolasBold_36 = NULL;
```

```
TTF_Font* Consolas_48 = NULL;

TTF_Font* Arial_14 = NULL;
TTF_Font* Arial_18 = NULL;
TTF_Font* ArialBold_18 = NULL;
TTF_Font* Arial_24 = NULL;
TTF_Font* ArialBold_24 = NULL;
```

Ukázka kódu 36 - definice písem

Podobným způsobem doporučuji vytvořit také instance textur, které má uživatel v rámci své aplikace v plánu globálně využívat.

```
/* TEXTURES */
SDL_Texture* grid;
SDL_Texture* settings;
```

Ukázka kódu 37 - definice textur

Inicializace dat

Inicializace dat probíhá prostřednictvím funkce loadMedia(). Výhodou řešení inicializace vstupních dat prostřednictvím funkce je jednak možnost kontroly, zda tato inicializace proběhla v pořádku. Je možné se tak vyhnout vzniku neošetřených výjimek ústících v pád aplikace bez uživateli zřejmého důvodu. Na začátku funkce proto doporučuji definovat proměnnou success, čímž získám možnost z výstupní hodnoty funkce zjistit, zda se inicializace zdařila. Výhodou také je, že tuto funkci posléze stačí zavolat pouze jednou při inicializaci aplikace a není nutné se o správu prostředků dále starat. Níže uvedu zkrácenou podobu této funkce, kde postupně dochází k načítání jednotlivých textur a písem:

```
/* MEDIA LOADING FUNCTION */
bool loadMedia()
{
	bool success = 1;
	/* textures */
	grid = IMG_LoadTexture(gRenderer, "grid.png");
	settings = IMG_LoadTexture(gRenderer, "ico/settings.png");

	/* fonts */
	Consolas_12 = TTF_OpenFont("fonts/consola.ttf", 12);
	Consolas_16 = TTF_OpenFont("fonts/consola.ttf", 16);
	ConsolasBold_16 = TTF_OpenFont("fonts/consolab.ttf", 16);
	Consolas_24 = TTF_OpenFont("fonts/consola.ttf", 24);
	ConsolasBold_24 = TTF_OpenFont("fonts/consolab.ttf", 24);

	Arial_14 = TTF_OpenFont("fonts/arial.ttf", 14);
	Arial_18 = TTF_OpenFont("fonts/arial.ttf", 18);
	ArialBold_18 = TTF_OpenFont("fonts/arialbd.ttf", 18);
	Arial_24 = TTF_OpenFont("fonts/arial.ttf", 24);
	ArialBold_24 = TTF_OpenFont("fonts/arialbd.ttf", 24);
	return success;
}
```

Ukázka kódu 38 - funkce pro načtení multimédií

Uvolnění paměti

K uvolnění paměti doporučuji použít obdobný přístup, jako k inicializaci vstupních dat. Za tímto účelem lze vytvořit funkci close(), která uvolní veškeré dříve načtené globální prostředky. Za povšimnutí stojí také volání funkcí čištění knihoven SDL a jejich doplňků. V případě funkce close(), vzhledem k jejímu charakteru, není nutné provádět kontrolu úspěšnosti. Pokud některý z uvolňovaných prvků nebyl do paměti dříve načten, žádost o uvolnění vyhrazených prostředků takového prvku nepředstavuje chybu. Podobu funkce je možné ve zkrácené podobě vidět níže:

```
void close()
{
	/* textures */
	SDL_DestroyTexture(grid);
	SDL_DestroyTexture(settings);

	/* fonts */
	TTF_CloseFont(Consolas_12);
	TTF_CloseFont(Consolas_16);
	TTF_CloseFont(ConsolasBold_16);
	TTF_CloseFont(Consolas_24);
	TTF_CloseFont(ConsolasBold_24);

	TTF_CloseFont(Arial_14);
	TTF_CloseFont(Arial_18);
	TTF_CloseFont(ArialBold_18);
	TTF_CloseFont(Arial_24);
	TTF_CloseFont(ArialBold_24);
	TTF_CloseFont(Arial_36);

	IMG_Quit();
	SDL_Quit();
	TTF_Quit();
}
```

Ukázka kódu 39 - funkce pro uvolnění paměti

K definici, načítání a čištění různých prvků, které nebyly v uvedené demonstraci zmíněny, ať již se jedná například o zvuky či jiné globální proměnné, pochopitelně lze přistupovat obdobným způsobem.

8.1.2 Soubor funkcí

V souboru funkcí jsou definovány veškeré funkce, které uživatel hodlá v mezích vznikajícího programu často využívat a nepřeje si, aby mu svými definicemi složitě znepřehledňovaly zdrojový kód. Mezi takové funkce pochopitelně patří například funkce grafické, pakliže nejsou do souboru importovány přímo prostřednictvím knihovny. Vzhledem k tomu, že grafické funkce jsem již podrobně zdokumentoval v kapitole 7.1, bylo by silně neproduktivní se o nich znovu okrajově zmiňovat zde. Grafické funkce však

pochopitelně nejsou zdaleka jediné, které budeme při tvorbě aplikace potřebovat. V závislosti na charakteru aplikace je potřeba až hypoteticky nekonečné množiny funkcí různého určení. Může jít například o funkce, které se postarají o převody jednotek, čtení bitových polí, spouštění zvuků, změnu dynamických prvků ve scéně a podobně.

Níže nastíním například funkci, která se postará o zaokrouhlení čísla na stovky:

```
float roundTo100(float x) {
	x /= 100;
	return floor(x) * 100;
}
```

Ukázka kódu 40 - funkce zaokrouhlení celých čísel na stovky

, nebo kupříkladu funkci, která převede datový typ integer na pole bytů:

```
vector<unsigned char> intToBytes(int paramInt)
{
	vector<unsigned char> arrayOfByte(4);
	for (int i = 0; i < 4; i++)
		arrayOfByte[3 - i] = (paramInt >> (i * 8));
	return arrayOfByte;
}
```

Ukázka kódu 41 - funkce pro převod čísla na pole bytů

, nebo do třetice funkci, která vrátí uživateli řetězec znaků obsahující aktuální čas:

```
std::string TimeStamp()
{
	char str[32]{};
	time_t a = time(nullptr);
	struct tm time_info;
	if (localtime_s(&time_info, &a) == 0)
	{
		strftime(str, sizeof(str), "%H:%M:%S", &time_info);
	}
	return str;
}
```

Ukázka kódu 42 - funkce pro získání řetězce s aktuálním časem

Některé funkce jsou však svým charakterem natolik specifické, že se pro přehlednost vyplatí je umístit do samostatného souboru.

8.1.3 Soubor s funkcemi zpracování uživatelského vstupu

Příkladem takových funkcí mohou být například funkce, které se starají o zpracování uživatelského vstupu. Tím mám na mysli například sledování vstupu dotykové obrazovky, kdy při dotyku prstu na povrch obrazovky dojde k zaznamenání události, že došlo ke stisku na určitém místě plochy. Další událostí, která pravděpodobně bude následovat, je pohyb prstu po dotykové ploše a jeho následné zvednutí. Všechny tyto události jsou nějakým způsobem zpracovány, ať se již jedná o uložení aktuální pozice prstu do určitých globálních

proměnných či jednoduché informace, zda se prst stále dotýká obrazovky či nikoliv. Za účelem zpracování těchto vstupů je definována například funkce getClicks().

```
DWORD WINAPI getClicks(LPVOID lpParameter)
{
	while (1)
	{
		MOUSE = SDL_GetMouseState(&MOUSEx, &MOUSEy);
		if (e.type == SDL_MOUSEBUTTONDOWN || e.type == SDL_FINGERDOWN)
		{
			TOUCHx = e.tfinger.x * SCREEN_WIDTH;
			TOUCHy = e.tfinger.y * SCREEN_HEIGHT;
			click = 1;
		}
		else if (e.type == SDL_FINGERMOTION)
		{
			TOUCHx = e.tfinger.x * SCREEN_WIDTH;
			TOUCHy = e.tfinger.y * SCREEN_HEIGHT;
		}
		else if (e.type == SDL_MOUSEBUTTONUP || e.type == SDL_FINGERUP)
		{
			TOUCHx = 0;
			TOUCHy = 0;
			MOUSEx = 0;
			MOUSEy = 0;
			TOUCHx = e.tfinger.x * SCREEN_WIDTH;
			TOUCHy = e.tfinger.y * SCREEN_HEIGHT;
			click = 0;
			e.type = 0;
		}
	}
}
```

Ukázka kódu 43 - funkce zpracování uživatelského vstupu

Tato funkce běží v samostatném vlákně a její provádění tak není závislé na hlavní smyčce programu. To se děje zejména z toho důvodu, že hlavní smyčka, která se stará o vykreslování prvků do scény, nemusí běžet zcela kontinuálně. K dočasnému pozastavení jejího běhu dochází například v případě, kdy si přejeme překreslit scénu pouze několikrát za daný časový interval, abychom zbytečně nepřetěžovali hardwarové prostředky aktualizací scény maximální možnou rychlostí. Takto častá aktualizace by jednak mohla být pro lidské oko nezaznamenatelná, stejně tak navíc existuje určitý předpoklad, že vstupní data zobrazení přicházejí s určitou periodicitou a překreslení scény bez její změny by tak bylo zcela zbytečné. Funkce zpracovává jak uživatelský vstup v podobě myši, tak dotykové obrazovky. Pokud dojde ke stisknutí pravého tlačítka myši, dojde k zaznamenání pixelové pozice kurzoru vzhledem k oknu aplikace do patřičných proměnných. Stejně je tomu v případě dotyku či pohybu prstu na dotykové ploše. Při uvolnění tlačítka či vzdálení prstu od obrazovky dojde k vynulování těchto proměnných.

Další takovou funkcí, která je však již součástí hlavní smyčky, je vzniklá funkce Click(). Ta jednak prostřednictvím globální proměnné click komunikuje s dříve zmíněným vláknem

a funkcí getClicks(), aby zjistila, zda je právě aktivní stisk tlačítka myši či dotyk obrazovky. Funkce dále kontroluje, zda se pozice kurzoru či prstu v okně nachází uvnitř zadané oblasti. Pokud ano, vrátí hodnotu 1, v opačném případě 0.

```
bool Click(SDL_Rect field)
{
	if (((click) &&
		MOUSEx > field.x &&
		MOUSEx < (field.x + field.w) &&
		MOUSEy > field.y &&
		MOUSEy < (field.y + field.h)) || (((click)) &&
			TOUCHx > field.x &&
			TOUCHx < (field.x + field.w) &&
			TOUCHy > field.y &&
			TOUCHy < (field.y + field.h)))
	{
		return 1;
	}
	else
	{
		return 0;
	}
}
```

Ukázka kódu 44 - funkce detekce poklepání na plochu

V tuto chvíli již lze vytušit, že se jedná o funkci, která je využívána zejména pro detekci stisku některého z tlačítek či jiných ovládacích prvků obrazovky. Její použití budu demonstrovat na příkladu tlačítka z kapitoly 7.1.4, kdy v případě detekce poklepání myší či dotyku prstu v oblasti tlačítka dojde k zobrazení zpětné vazby v podobě vybarvení tohoto tlačítka bílou barvou:

```
SDL_Rect button1_rect = { 10, 390, 80, 80 }; //oblast tlačítka
//vykreslení tlačítka:
TextButton(button1_rect, lgrey, DMIblue, 3, L"Tlačítko", Arial_18, lgrey);

if (Click(button1_rect)) //detekce stisku
   {
       Rectangle(white, button1_rect); //vybarvení bílou barvou
   }
```

Ukázka kódu 45 - podbarvení tlačítka v případě jeho stisku

8.1.4 Soubor datové komunikace

Dalším příkladem specifických funkcí, které je pro svůj charakter vhodné umístit do samostatného souboru, jsou funkce, které zprostředkovávají datovou komunikaci. Níže nastíním pro příklad funkci UDPCom(), která zpracovává příchozí data prostřednictvím komunikačního protokolu UDP.

```
DWORD WINAPI UDPCom(LPVOID lpParameter)
{
	printf("Vlakno UDPCom  bezi. Cekam na spojeni. \n");
	WSACleanup();
	static struct sockaddr_in server, si_other; //adresa serveru
	static bool initsocket = FALSE;
	WSADATA wsaData;
	if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) //start windows socketu
	{
		printf("Failed. Error Code : %d", WSAGetLastError());
	}

	if (initsocket == FALSE)
	{
		//vytvoření socketu
		if ((serverPC_UDP = socket(AF_INET, SOCK_DGRAM, 0)) == INVALID_SOCKET)
		{
			printf("Could not create socket : %d", WSAGetLastError());
		}
	}

	server.sin_family = AF_INET;
	server.sin_addr.s_addr = INADDR_ANY; //akceptování libovolné IP adresy
	server.sin_port = htons(1880); //nastavení portu

	//přiřazení socketu
	if (bind(serverPC_UDP, (struct sockaddr *)&server, sizeof(server)) == SOCKET_ERROR)
	{
		printf("\n Bind failed with error code : %d", WSAGetLastError());
	}

	sockaddr_in clientAddr;
	int addrlen = sizeof(clientAddr);

	while (1)
	{
		memset(Buffer, 0, sizeof(Buffer)); //čištění paměti

		//kontrola, zda pravidelně přichází pakety
		if (timeout_recvfrom(serverPC_UDP, (struct sockaddr *)&clientAddr, recv_buf,
		    sizeof(recv_buf), addrlen, 2) == FALSE)
		{
			//detekována ztráta komunikace
			printf("Probehl 2s timeout... Ztrata komunikace.\n");
			memset((char*)&Buffer, 0, sizeof(Buffer));
		}
		else
		{
			//komunikace OK
			if (sizeof(recv_buf) < 388)
			{
				//detekce vadného či neočekávaného paketu
				warning = L"Received too short or incomplete packet.";
			}
			else
			{
				//kopírování příchozích dat do paměti
				memcpy((char*)&Buffer, &recv_buf[0], sizeof(Buffer));
			}
		}
	}

	std::cout << "Connection with client was destroyed!\n";
	closesocket(serverPC_UDP); //zavření socketu
	WSACleanup();
}
```

Ukázka kódu 46 - příjem a zpracování UDP paketu

Uvedená funkce opět běží v samostatném vlákně, neboť kontrolu příchozích dat je zapotřebí provádět neustále. Ta však není zásadně náročná na systémové prostředky, neboť samotná funkce recvfrom(), která přijímá data, je blokovací. To znamená, že funkce čeká na příchozí data a dokud na vstupu není žádný nový příchozí paket, běh funkce se pozastaví, dokud nedojde k detekci nového příchozího paketu. Po přijetí paketu dochází k jeho zběžné kontrole dle stanovených požadavků (například na minimální délku), a pokud se jeho obsah jeví v pořádku, data z paketu jsou uložena do stanoveného bufferu v paměti.

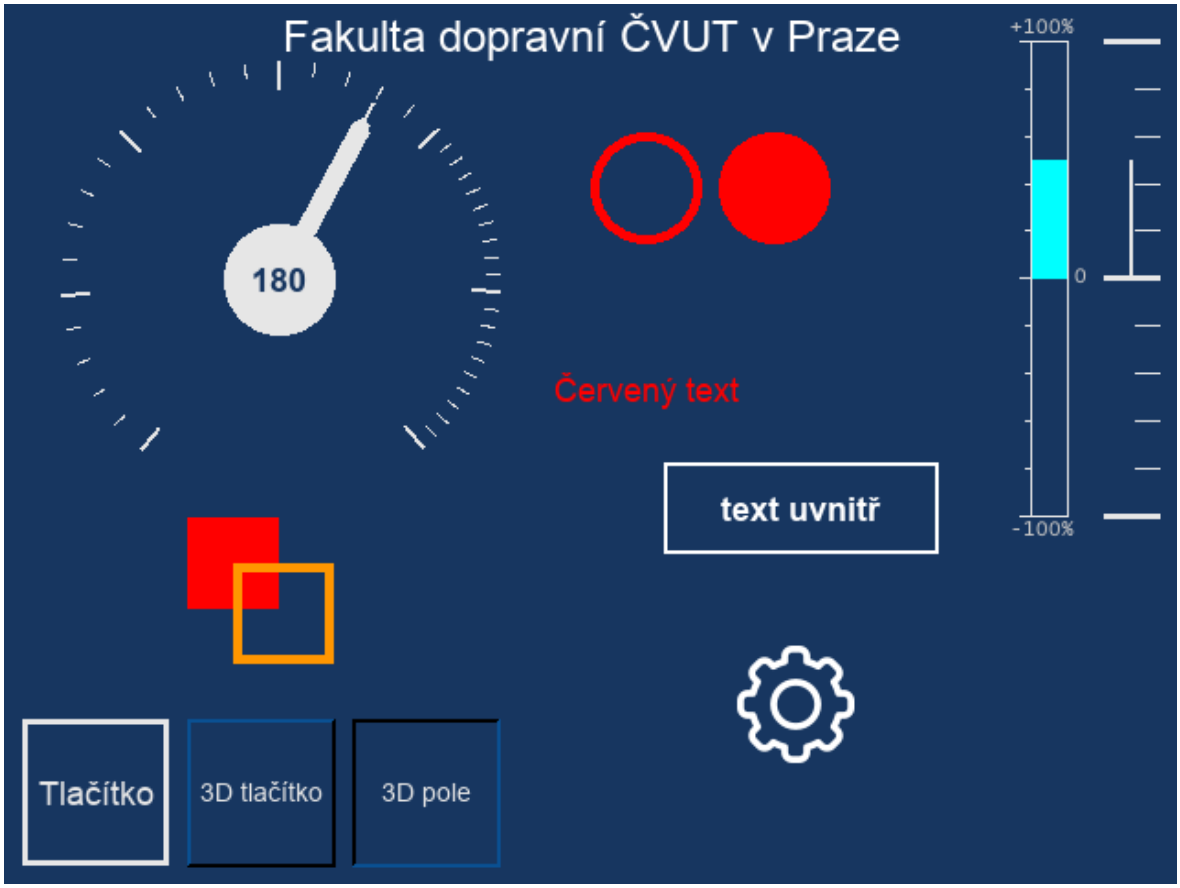
8.1.5 Soubor obrazovek

Souborem, který jsem záměrně ponechal až na závěr této kapitoly, je takzvaný soubor obrazovek. Ten v sobě prostřednictvím funkcí zaštiťuje veškerá zobrazení, která se v rámci dané aplikace mohou vykreslovat. Těmi může být například provozní obrazovka, diagnostická obrazovka, obrazovka s nastavením apod. O volání těchto funkcí se stará přepínač, který na základě zvoleného módu zobrazení zavolá patřičnou funkci s obsahem obrazovky. V předchozí kapitole jsem jednotlivě popsal různé prvky, které lze za pomoci vyvinutých knihoven do scény vykreslit. Nyní uvedu, jak vypadá definice kompletní scény se všemi vykreslovanými prvky. Příkladem může být funkce Screen_MAIN(), která obsahuje definice všech prvků, které se mají zobrazit na hlavní obrazovce aplikace:

```
void Screen_MAIN() //hlavní obrazovka
{
	SetColor(DMIblue); //modré pozadí obrazovky
	SDL_RenderClear(gRenderer); //vykreslení pozadí

	Text(L"Fakulta dopravní ČVUT v Praze", Arial_24, SCREEN_WIDTH / 2, 18, white, 1); //nadpis

	Text(L"Červený text", Arial_18, 300, 200, red); //vykreslení červeného textu

	Speedometer({ 150, 150 }, 120, 30, -50, 6, 5, 16, 3, 100, 20, lgrey); // rychloměr

	RelativeThrust(0, 600, 20, 30, 0.3, 50, lgrey, cyan, Consolas_12); //Tah liniově
	RelativeThrust(1, 560, 20, 20, 0.3, 50, lgrey, cyan, Consolas_12); //Tah sloupcově

	//tlačítka a pole
	SDL_Rect button1_rect = { 10, 390, 80, 80 };
	SDL_Rect button2_rect = { 100, 390, 80, 80 };
	SDL_Rect button3_rect = { 190, 390, 80, 80 };
	TextButton(button1_rect, lgrey, DMIblue, 3, L"Tlačítko", Arial_18, lgrey);
	_3dButton(button2_rect, DMIblue_light, DMIblue_dark, L"3D tlačítko", Arial_14, lgrey);
	_3dField(button3_rect, DMIblue_light, DMIblue_dark, L"3D pole", Arial_14, lgrey);

	//detekce kliknutí na tlačítko
	if (Click(button1_rect))
	{
		Rectangle(white, button1_rect);
	}

	// vykreslení obrázku
	Img(settings, 400, 350, 64, 64);
	//kruhy s ohraničením a s výplní
	Circle(350, 100, 5, 30, red, false);
	Circle(420, 100, 0, 30, red,  true);
}
```

Ukázka kódu 47 - definice hlavní obrazovky

Výsledný grafický výstup takto definované obrazovky je možné vidět níže na Obr. 25.



Obr. 25 - Příklad výstupu definice hlavní obrazovky

zdroj: vlastní zpracování

Kromě hlavní obrazovky však předpokládám, že v dané aplikaci budou existovat také určitá další zobrazení. Příkladem může být obrazovka s nastavením. Nyní uvedu, jak by mohla vypadat definice takové obrazovky, která je pro demonstrační účely poněkud jednodušší. Jedná se o obrazovku Nastavení, o jejíž vykreslení se zaslouží funkce Screen_SETTINGS()

```
void Screen_SETTINGS()
{
	SetColor(DMIblue);
	SDL_RenderClear(gRenderer);

	Text(L"Nastavení", Arial_24, SCREEN_WIDTH / 2, 18, white, 1);

	SDL_Rect goBack = { SCREEN_WIDTH / 2 - 30, 200, 60, 60 };
	TextButton(goBack, lgrey, ddblue, 3, L"ZPĚT", Arial_18, lgrey);
	if (ClickOnce(goBack))
	{
		Rectangle(white, goBack);
		screen_mode = 0;
		e.type = 0;
	}
}
```

Ukázka kódu 48 - definice obrazovky nastavení

Tato funkce záměrně neobsahuje mnoho prvků k zobrazení. Na obrazovce se nachází pouze nadpis a tlačítko s popisem „ZPĚT“ (Obr. 26). Je však možné si ve zdrojovém kódu povšimnout logiky přepínání mezi obrazovkami. K tomu zde slouží funkce ClickOnce(), která je svým charakterem velmi podobná již popsané funkci Click(). Tato funkce taktéž detekuje poklepání na daný objekt, avšak s tím rozdílem, že po zaznamenání události je tato událost bezprostředně ukončena, aby k přepnutí obrazovky došlo pouze jednou a nemohla nastat situace, že by po přepnutí do jiné obrazovky událost poklepání nadále trvala. To by mohlo uživateli způsobit nepříjemnosti zejména v případě, kdy by se na nové obrazovce na stejném místě nacházelo jiné tlačítko. Při stále aktivní události poklepání by tak byla bezprostředně po přepnutí do nové obrazovky spuštěna logika stisku tohoto jiného tlačítka, což považuji za silně nežádoucí.

Obr. 26 - Jednoduchá obrazovka se stisknutelným tlačítkem

zdroj: vlastní zpracování

O přepínání mezi obrazovkami se stará již zmíněný přepínač, který se nachází uvnitř funkce Draw(). Ten na základě proměnné screen_mode určuje, která z funkcí obrazovek se bude aktuálně volat.

```
void Draw()
{
	switch (screen_mode)
	{
	case 0: Screen_MAIN(); break;
	case 1: Screen_SETTINGS(); break;
	case 2: Screen_SERVICE(); break;
	case 3: Screen_FAULTS(); break;
	}
	SDL_RenderPresent(gRenderer);
}
```

Ukázka kódu 49 – Funkce Draw() zodpovědná za volbu vykreslované scény

8.1.6 Hlavní zdrojový soubor

Hlavní zdrojový soubor aplikace je zodpovědný za její inicializaci, ukončení a běh. Na začátku běhu dochází k inicializaci a nastavení knihoven SDL, jejich doplňujících součástí a vytvoření okna. Následně je volána funkce Init() pro načtení zdrojových multimédií do paměti. Dále zde dochází k definici a spuštění všech postranních vláken, jako je detekce uživatelského vstupu či příjem dat. Po této inicializaci vstoupí program do hlavní smyčky, kde je periodicky volána funkce Draw() vykreslující aktuální scénu. V případě požadavku na ukončení aplikace se ještě patřičná funkce postará o uvolnění paměti a vypnutí všech subsystémů.

8.2 Instalace knihoven do prostředí MS Visual Studio

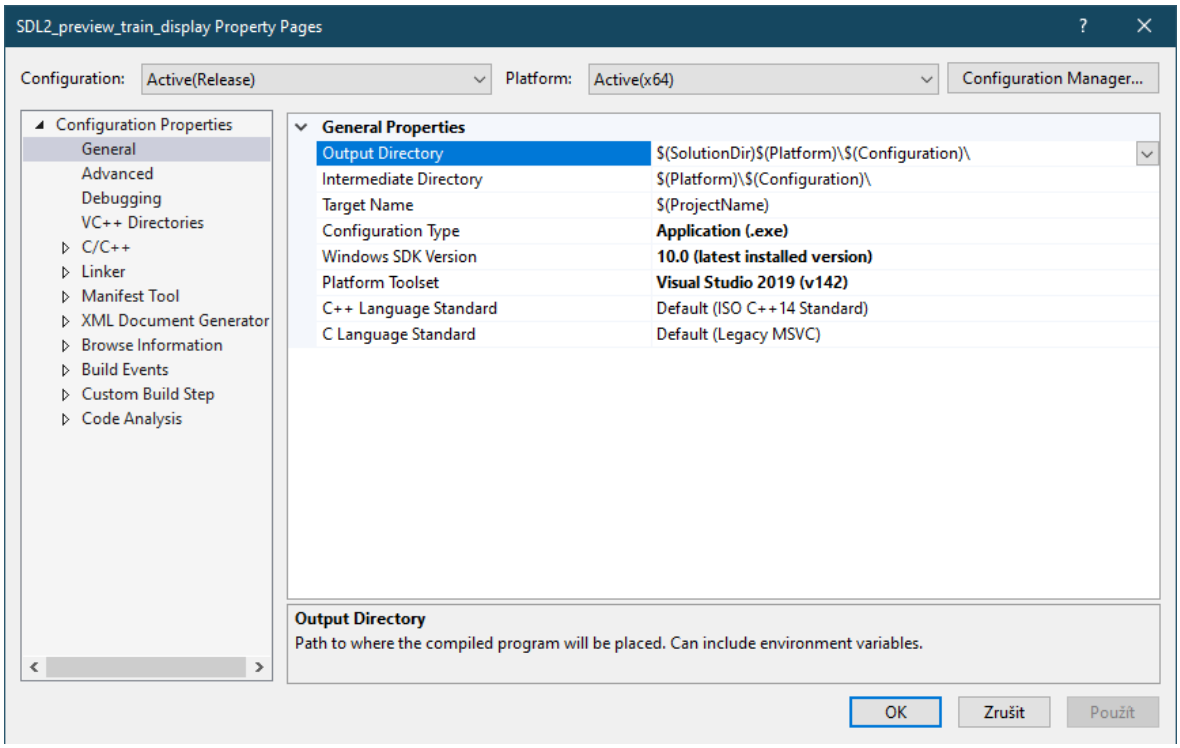
Následující postup představuje obecný a jednoduchý návod zavedení knihoven do prostředí Microsoft Visual Studio. Instalaci knihoven zde budu demonstrovat například na již mnohokrát zmiňované a četně využívané grafické knihovně SDL 2.0 a jejích doplňcích.

8.2.1 Stažení knihoven pro vývojáře

Knihovny SDL 2.0 jsou dostupné ke stažení na domovské stránce projektu SDL. (28) Je možné zde sehnat jak samotné zdrojové kódy, tak knihovny specificky určené pro kompilaci na konkrétních operačních systémech či v konkrétních vývojových prostředích. V případě kompilace pro platformu Windows stáhneme ze stránky knihovny pro 32-bitovou či 64-bitovou verzi tohoto operačního systému, v závislosti na tom, pro jakou verzi tohoto systému si přejeme kompilovat. Knihovny pro 32-bitovou verzi je pochopitelně možné používat pro kompilaci či debugging i v 64-bitové verzi, avšak s omezenými možnostmi využití systémových prostředků. Stažené knihovny extrahujeme a umístíme do libovolného adresáře v počítači. Podobně postupujeme také v případě všech doplňků knihoven, které si přejeme používat.

8.2.2 Vytvoření a nastavení projektu v MS Visual Studio

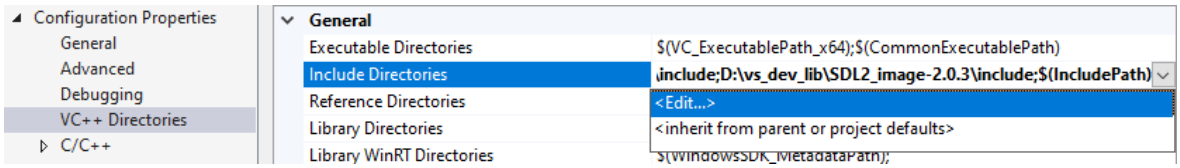
Ve vývojovém prostředí vytvoříme nový prázdný projekt v jazyce C++. Následně otevřeme okno s podrobnostmi projektu: (Obr. 27)



Obr. 27 - Okno s podrobnostmi projektu v MS Visual Studio

zdroj: vlastní zpracování

V okně podrobností nastavíme také dle verze stažených knihoven platformu, pro kterou si přejeme kompilovat. Následně se přesuneme na záložku s názvem „VC++ Directories“. V této záložce najdeme položku „Include Directories“ a zvolíme možnost „<Edit…>“ (Obr. 28)

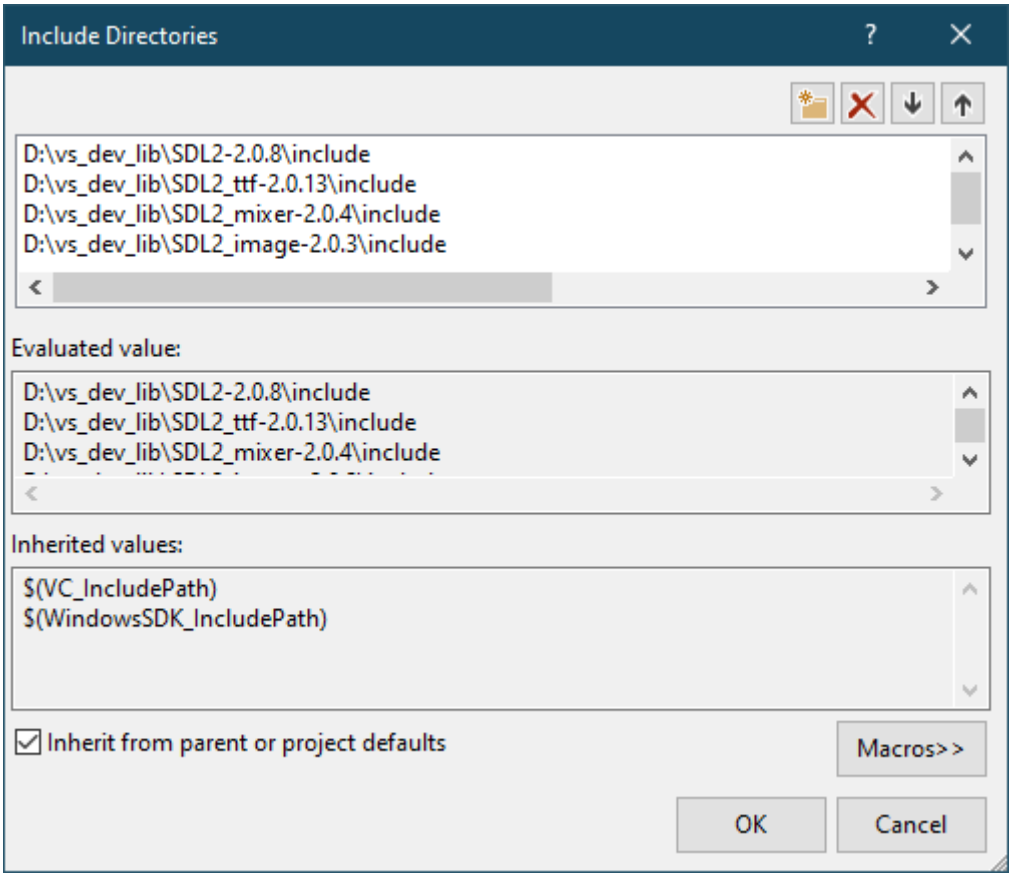


Obr. 28 - Záložka "VC++ Directories" v nastavení projektu

zdroj: vlastní zpracování

Po zvolení této možnosti se otevře dialogové okno „Include Directories“, ve kterém uvedeme cesty ke všem adresářům Include ve složkách s knihovnami či jejich rozšířeními.

Výslednou podobu tohoto okna po zavedení všech žádoucích adresářů můžeme vidět na Obr. 29.



Obr. 29 - Okno "Include Directories" v nastavení projektu

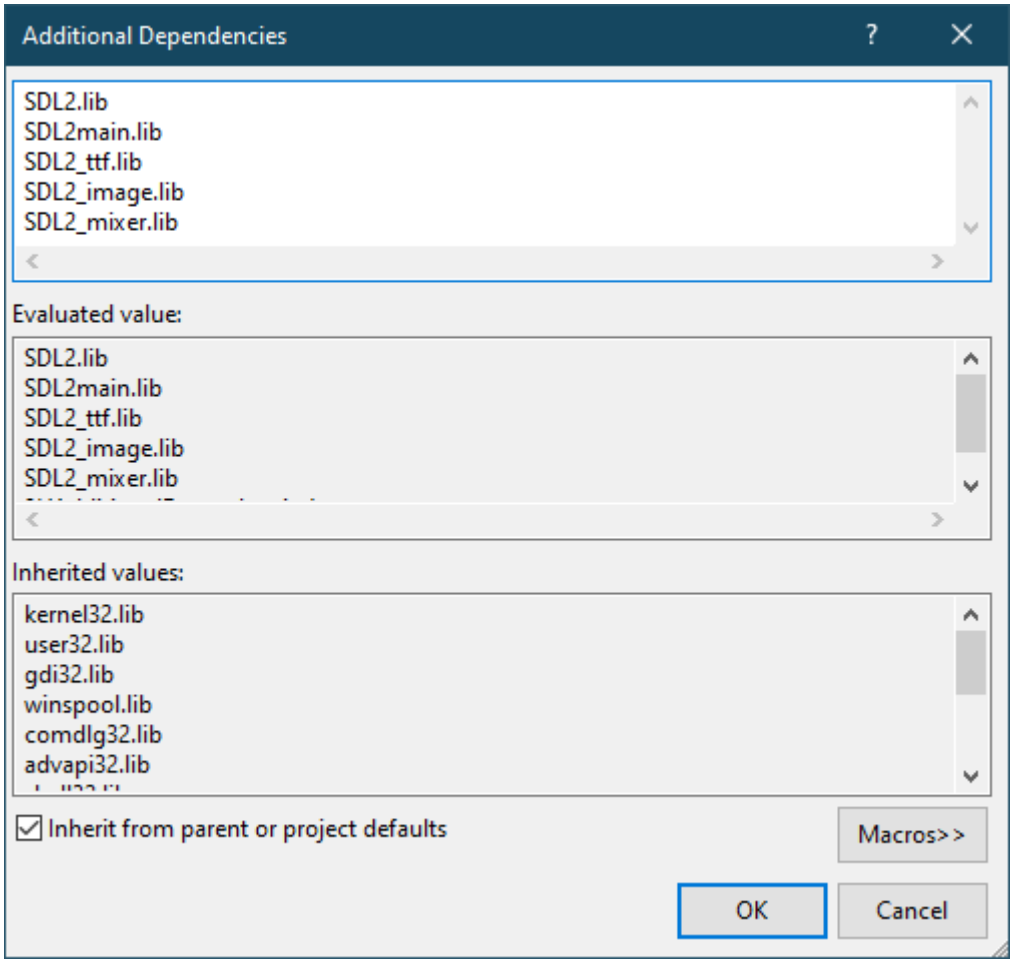
zdroj: vlastní zpracování

Po nastavení všech cest k požadovaným direktivám „Include“ postupujeme obdobně také v případě zavedení cest k samotným knihovnám. V rámci záložky vybereme položku „Library Directories“ a stejným způsobem poklepáním na možnost „<Edit…>“ zadáme také cesty k podadresářům „lib“ ve stažených adresářích s knihovnami.

Posledním krokem je otevření položky „Input“ v záložce „Linker“. Zde se nachází pole „Additional Dependencies“. Opětovným výběrem položky „<Edit…>“ otevřeme dialogové okno, ve kterém namísto cest ke zdrojovým podadresářům tentokrát vyplníme názvy konkrétních souborů s příponou „.lib“ ve stejnojmenných složkách. Výslednou konfiguraci v případě zahrnutí knihoven SDL a jejich součástí SDL_Img, SDL_TTF a SDL_Mixer můžeme vidět na Obr. 30.

Po zavedení knihoven tímto způsobem by již prostředí Visual Studio mělo při zahrnutí knihovních souborů prostřednictvím direktivy #include tyto soubory bez problému

rozpoznat, čímž získáme možnost využívat knihovny ve svém projektu a námi navržené řešení bez problému kompilovat.



Obr. 30 - Nastavení dodatečných závislostí v MS Visual Studio

zdroj: vlastní zpracování

9 Ukázka vytvořených projektů, využívající navrženou knihovnu

Za pomocí knihoven, které jsou hlavním výstupem této práce, jsem postupně vytvořil několik dílčích zobrazení displejů kolejových vozidel, zejména pak těch, které jsou součástí simulátoru metra DSFD. Jedním z těchto zobrazení je například obrazovka tachografu.

9.1.1 Grafická replika analogového tachografu metra

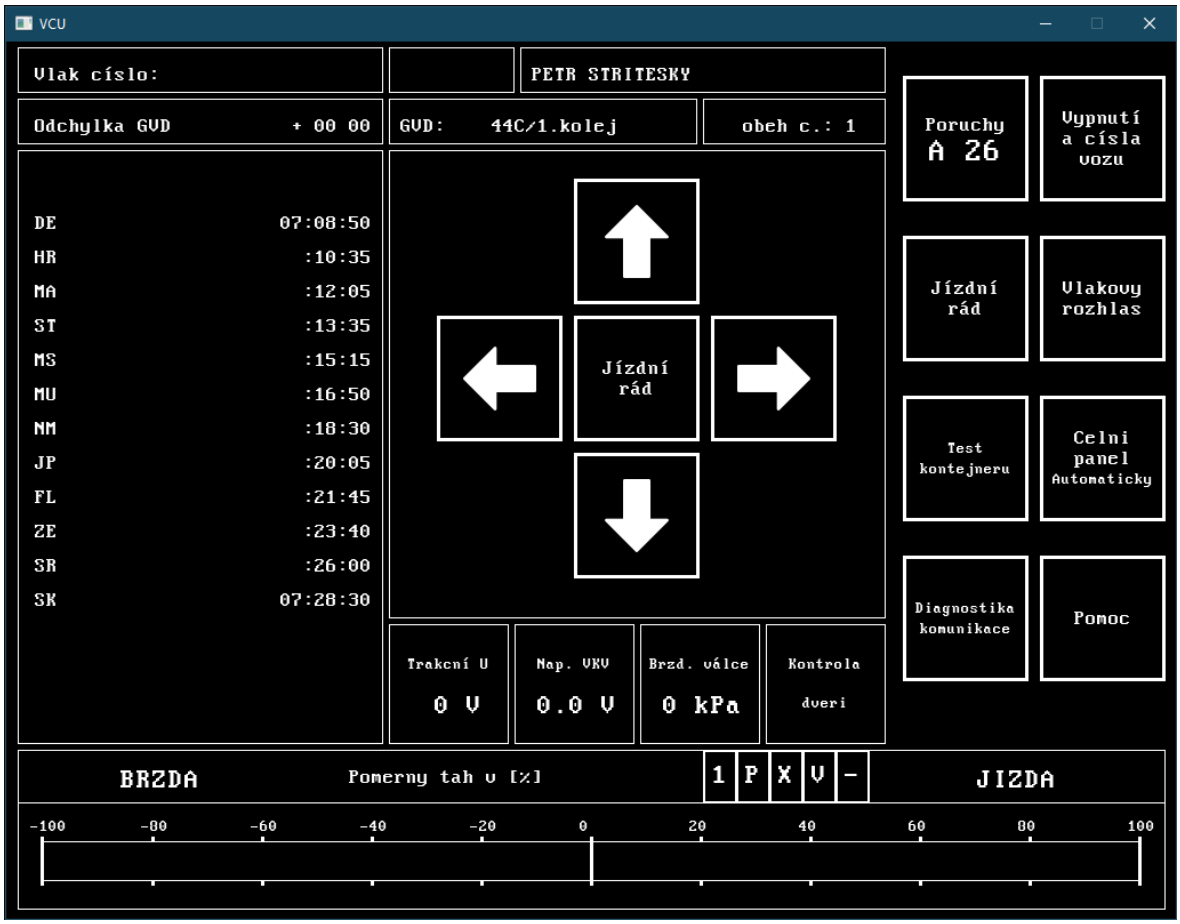
Vzhledem k nedostupnosti původního analogového rychloměru vozidla 81-71M pro použití v simulátoru soupravy metra došlo po konzultaci s vedoucím projektu k rozhodnutí tento analogový rychloměr nahradit digitální obrazovkou. Takové řešení však vyžaduje nutnost toto zobrazení vytvořit co nejvěrohodněji, aby do nejvyšší možné míry dokázalo navodit dojem skutečného analogového rychloměru. Za tímto účelem jsem pro tvorbu základních textur zvolil metodu použití tzv. fototextur, tedy textur, které nejsou vytvořeny genericky, nýbrž na základě reálných fotografií daného objektu. Tato metoda vyžaduje vyšší míru zkušeností s grafickými editory, neboť fotografie před svojí transformací na textury musí být do značné míry upraveny. Tyto úpravy zahrnují například geometrické zarovnání, odstranění odlesků, vytvoření separátních textur pro prvky v pozadí a popředí, řešení stínování apod. Zdrojové fotografie však není vhodné upravovat příliš, neboť právě nečistoty, stíny a různé nedokonalosti jsou tím, co odlišuje fototextury od generických textur a dodává tak výslednému zobrazení dojem reálného prvku. Grafickou podobu vytvořené aplikace je možné vidět na Obr. 31.

Obr. 31 - Výsledná podoba aplikace zobrazení tachografu

zdroj: vlastní zpracování

9.1.2 Obrazovka řídicího počítače VCU

VCU (Vehicle control Unit) představuje základní řídicí počítač soupravy metra. S ním strojvedoucí komunikuje prostřednictvím dedikované dotykové obrazovky na stanovišti. Pro toto zobrazení jsem vytvořil aplikaci do maximální možné míry odpovídající reálnému zobrazení, a to včetně simulace nižšího rozlišení displeje a užití systémových písem typických pro starší grafická zobrazení. Výslednou podobu hlavní obrazovky vzniklé aplikace je možné vidět níže na Obr. 32.

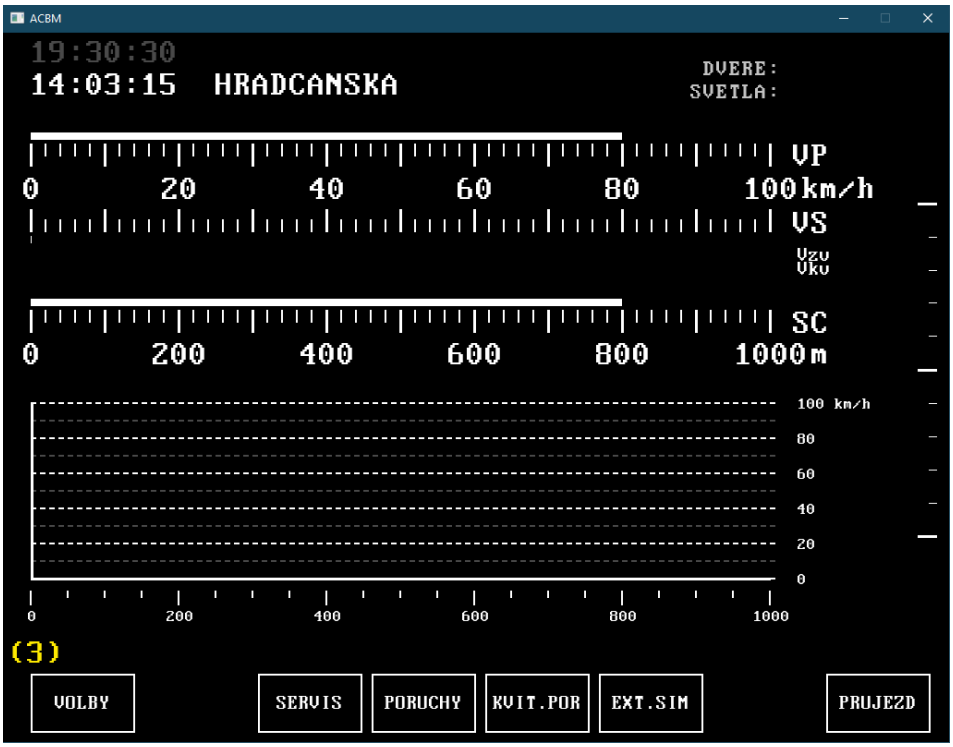


Obr. 32 - Výsledné zobrazení hlavní obrazovky VCU.

zdroj: vlastní zpracování

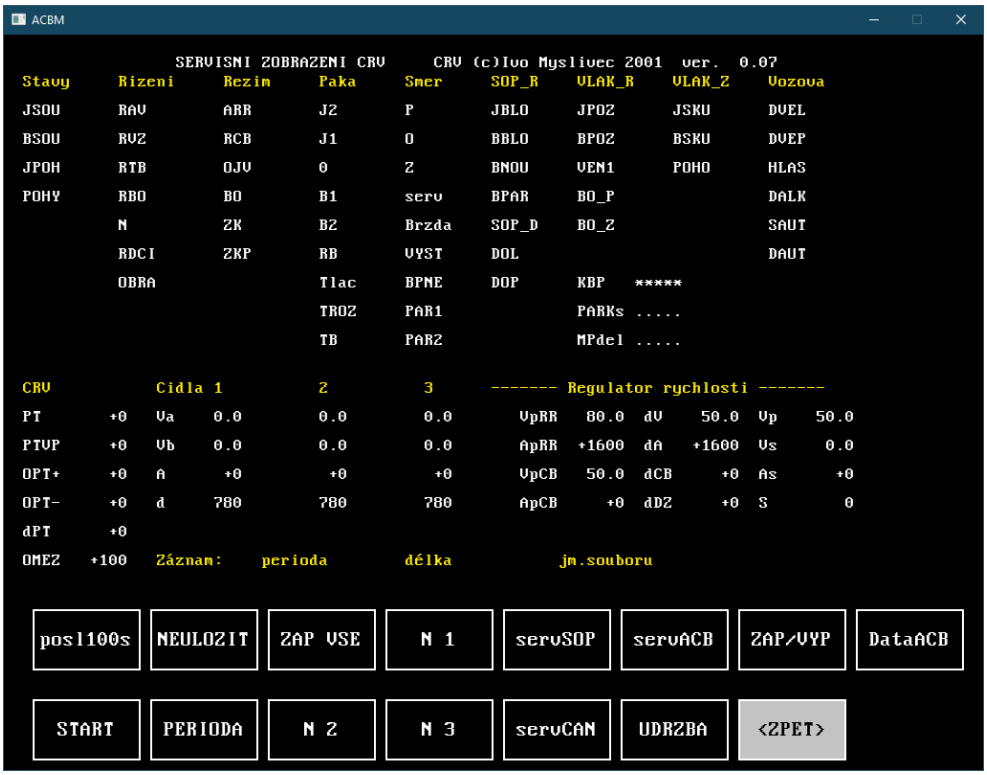
9.1.3 Obrazovka automatizačního zařízení ACBM-3

ACBM-3, neboli automatické cílové brzdění metra je systém dodávaný společností AŽD Praha pro automatizaci jízdy a schopnost cílového brzdění soupravy ke stanoveným prvkům, jako jsou omezení rychlosti, zastávky a prvky zabezpečovacího zařízení LZA. Podobně jako v případě VCU, také při implementaci tohoto řešení jsem kladl důraz na maximální možnou míru připodobnění zobrazení reálné předloze. Příklady výsledné podoby vzniklé grafické aplikace je možné vidět na Obr. 33 a Obr. 34.



Obr. 33 - Hlavní provozní obrazovka systému ACBM-3

zdroj: vlastní zpracování



Obr. 34 - Servisní obrazovka systému ACBM-3

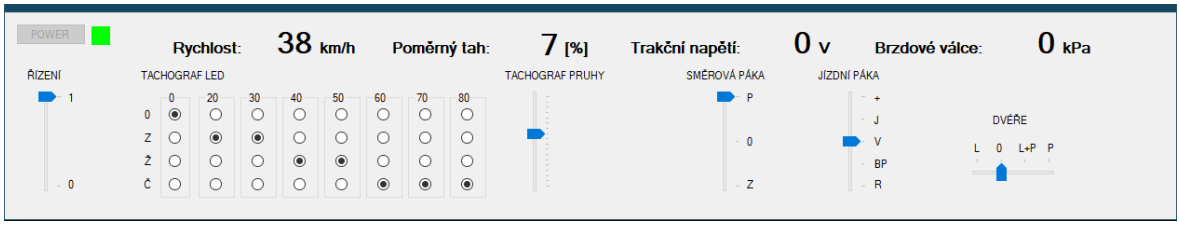
zdroj: vlastní zpracování

10 Vývoj simulátoru zdrojových dat pro ověření funkce displejů

Zobrazení určená pro použití v simulátoru metra typu 81-71M bylo vlivem teprve vznikající simulace centrálního počítače během vývoje a ladění nutné připravit na budoucí implementaci do pultu simulátoru, průběžně je zkoušet a následně demonstrovat vhodným způsobem jejich funkci a reakceschopnost na změnu obsahu dat. Za tímto účelem jsem sestavil primární návrh komunikační datové struktury. Na základě této struktury jsem následně definoval komunikační rozhraní založené na přenosu dat prostřednictvím protokolu UDP a implementoval do dílčích aplikací metody pro příjem a zpracování těchto paketů. Chyběl však jeden zásadní element, a to zdroj těchto testovacích dat. Rozhodl jsem se tedy vytvořit jednoduchou formulářovou aplikaci, která bude schopná simulovat některé ze základních ovládacích prvků pultu strojvedoucího a kontrolovat zobrazení dalších specifických prvků, dokud nedojde k vytvoření jejich vlastní simulační logiky. Vzniklá aplikace je schopná ovládáním přepínače jízdní páky simulovat zadání poměrného tahu, jehož prostřednictvím je díky velmi zjednodušenému modelu chování možné simulovat jízdu či brzdění daného vozidla, ovládat některé diagnostické elementy či například simulovat stav otevření dveří. Dovolím si zdůraznit, že se jedná o simulaci s primárním cílem demonstrovat interaktivní chování obrazovek, nikoliv napodobit skutečný fyzikální model vozidla. Uživatelské rozhraní této aplikace je vyobrazeno na Obr. 35.

```
/* OVLADANI */
sbyte / int 8   (  0 ): rizeni ON/OFF
sbyte / int 8   (  1 ): ACBM ON/OFF
sbyte / int 8   (  2 ): poloha smerove paky
sbyte / int 8   (  3 ): poloha jizdni paky
sbyte / int8    (  4 ): pomerny tah [%]
double/float64 ( 5-12 ): rychlost [km/h]
double/float64 (13-20): trakcni napeti [V];
double/float64 (21-28): trakcni proud [A];
sbyte / int8  (  29): dvere //-1:L 0:0 1:L+P 2:P
```

Ukázka kódu 50 - výňatek z návrhu komunikační datové struktury



Obr. 35 - Uživatelské rozhraní simulátoru zdrojových dat

zdroj: vlastní zpracování

11 Závěr

Ve své práci jsem si stanovil za cíl vytvořit generické knihovny pro provozní zobrazení kolejových vozidel. Základní motivací za výběrem tohoto tématu byla zejména snaha o usnadnění a zavedení jisté míry standardizace do procesu vývoje těchto silně specifických zobrazení, neboť se tvorbou zobrazení kolejových vozidel jsem se měl možnost setkat již několikrát v minulosti, a to jak v rámci svých studijních, tak pracovních aktivit. Velmi jsem však na tomto poli postrádal existenci obecně aplikovatelného konceptu, jak k návrhu takových zobrazení přistupovat. Díky tomu vznikla myšlenka vytvořit sadu specifických funkcí, které budou univerzálně využitelné pro tvorbu stabilního základu zobrazení kolejových vozidel ve stanoveném programovacím jazyce použitelného na různých platformách. Za tímto účelem jsem na základě dříve osvojených postupů využil možností multiplatformních knihoven SDL 2.0 a zapouzdřil je do specifických jazykových konstrukcí zahrnujících metody, jejichž jednotlivá implementace by byla nejen časově náročná, ale vyžadovala by také detailní znalost knihoven SDL.

Před popisem vzniklých funkcí a jejich vývoje jsem nejprve podrobně popsal problematiku zobrazení kolejových vozidel a uvedl veškeré použité prostředky a metody spolu s důvody jejich výběru. Pro možnost využití softwaru na širším množství operačních systémů bez nutnosti přepracování příliš velkého množství kódu jsem se rozhodl použít k vývoji programovací jazyk C++. Z obdobného důvodu jsem zvolil vývojové prostředí MS Visual Studio, neboť dnes již poskytuje možnosti kompilace pro většinu běžně využívaných platforem a nabízí nadstandardní možnosti diagnostiky a ladění chyb.

Při popisu samotných funkcí jsem se soustředil na jejich srozumitelnou interpretaci zejména pro programátory, neboť vzniklé knihovny byly poskytnuty pro současné i budoucí užití dalším osobám zejména v rámci projektu Dopravního sálu a univerzity. Tuto část práce lze považovat za programátorskou příručku k použití vytvořených knihoven. Velký potenciál knihoven do budoucna však shledávám také pro své osobní užití v rámci následujících studijních i pracovních činností.

Vytvořené knihovny jsem již během závěrečné fáze jejich vývoje využil ke tvorbě několika konkrétních zobrazení simulátoru soupravy pražského metra. Tím jsem získal důležitou zpětnou vazbu v podobě uživatelských zkušeností, na základě kterých bylo možné dané funkce optimalizovat a dokončit tak, aby byly splněny podmínky jednoduchosti, intuitivity a efektivity jejich použití. Na vzniklých zobrazeních jsem následně demonstroval příklady konkrétních výstupů, které lze s použitím knihoven získat.

Vzniklé knihovny a aplikace jejich prostřednictvím vyvinuté představují zásadní výstup této práce, na jehož základě je možné zhodnotit, zda bylo naplněno zadání práce. Za zásadní kritérium v tomto ohledu považuji uplatnitelnost výstupu v praxi. Toho bylo z mé strany dosaženo tím, že vzniklá zobrazení jsou nedílnou součástí simulátoru soupravy pražského metra, kde mohou být využívány pro potřeby vědeckého výzkumu a školení strojvedoucích. Daných knihoven hodlám dále využívat také v rámci své další činnosti ve společnosti AŽD Praha. Vzniklé knihovny se však na základě vlastní volby z několika nabízených možností rozhodli použít také studenti Fakulty informačních technologií ČVUT pro svoji práci na projektu tvorby zobrazení DMI evropského vlakového zabezpečovače ETCS. Toto rozhodnutí pro mne představuje důležitý aspekt při hodnocení smysluplnosti práce, neboť předpoklad využití knihoven dalšími uživateli byl naplněn ještě před samotným odevzdáním práce.

V kontextu výše uvedených informací si tak troufám tvrdit, že všechny body zadání byly naplněny. Zpracování této práce pro mne vyjma hmatatelných výstupů představuje množinu nových zkušeností, na které hodlám dále navázat v rámci svého dalšího pracovního či studijního rozvoje.

Obr. 36 – Vzniklé aplikace po instalaci na cílová zařízení simulátoru

zdroj: vlastní zpracování

12 Použité zdroje

1. **International Union of Railways.** *UIC 612-0: Driver Machine Interfaces for EMU/DMU, Locomotives and driving coaches - Functional and system requirements associated with harmonised Driver Machine Interfaces.* 1. Paříž : UIC, 2009.

2. **NENUTIL, Dobromil.** Řídicí systémy vlaku. *AUTOMA - časopis pro automatizační techniku.* [Online] https://automa.cz/Aton/FileRepository/pdf_articles/52954.pdf.

3. **ŠTĚPÁN, Václav.** *Panelový počítač pro kolejová vozidla, návod na obsluhu.* 1.00. místo neznámé : AMiT, spol. s.r.o., 2016. TD008922.

4. **HARÁK, Martin.** Poláci zaškolují strojvedoucí na speciálních trenažerech. *Železničář.* [Online] 30. Duben 2015. https://zeleznicar.cd.cz/zeleznicar/zahranici/polaci-zaskoluji-strojvedouci-na-specialnich-trenazerech/-7587/22,0,,/.

5. **SŮRA, Jan.** České dráhy kupují dva trenažéry pro strojvedoucí za 30 milionů korun. *ZDopravy.cz.* [Online] 18. Září 2017. https://zdopravy.cz/ceske-drahy-kupuji-dva-trenazery-pro-strojvedouci-za-30-milionu-korun-1869/.

6. **HOLEK, Josef.** V simulátorech strojvedoucí natrénují komunikaci. *Časopis Železničář.* Leden 2018, ISSN 0322-8002.

7. **TIOBE.** TIOBE Index for April 2021. *TIOBE.* [Online] [Citace: 26. Duben 2021.] https://www.tiobe.com/tiobe-index/.

8. **STŘÍTESKÝ, Petr.** *Zpracování a vizualizace diagnostických hlášení z řídicího systému kolejového vozidla.* Hradec Králové. Diplomová práce : Univerzita Hradec Králové, Fakulta informatiky a managementu, 2021.

9. **Guru99.** What is C Programming Language? Basics, Introduction, History. *Guru99.* [Online] https://www.guru99.com/c-programming-language.html.

10. **TutorialsPoint.** C++ Basic Syntax. *TutorialsPoint.* [Online] https://www.tutorialspoint.com/cplusplus/cpp_basic_syntax.htm.

11. **cplusplus.com.** A Brief Description. *cplusplus.com.* [Online] https://www.cplusplus.com/info/description/.

12. **SDL: Simple DirectMedia Layer.** About SDL. *SDL: Simple DirectMedia Layer.* [Online] https://www.libsdl.org.

13. **LinkedIn.** Sam Lantinga - Senior Software Engineer - Valve corporation. *LinkedIn.* [Online] LinkedIn Corporation. [Citace: 26. Duben 2021.] https://www.linkedin.com/in/sam-lantinga-02771b10.

14. **Academic.** Simple DirectMedia Layer. *Academic.* [Online] https://en-academic.com/dic.nsf/enwiki/17977.

15. **GeeksforGeeks.** Socket Programming in C/C++. *GeeksforGeeks.* [Online] 31. Květen 2019. https://www.geeksforgeeks.org/socket-programming-cc/.

16. **Dartmouth College.** Lecture 3 and 4: Socket Programming. *CS 60 Computer Networks.* [Online] https://www.cs.dartmouth.edu/~campbell/cs60/socketprogramming.html.

17. **Guru99.** TCP vs UDP: What's the Difference? *Guru99.* [Online] 2021. https://www.guru99.com/tcp-vs-udp-understanding-the-difference.html#3.

18. **YUAN, Michael.** Getting to know MQTT. *IBM Developer.* [Online] IBM, 7. Leden 2020. [Citace: 28. Duben 2021.] https://developer.ibm.com/technologies/messaging/articles/iot-mqtt-why-good-for-iot/.

19. **SDL: Simple DirectMedia Layer.** SDL 2.0 API - dokumentace knihoven SDL 2.0. *SDL Wiki.* [Online] https://wiki.libsdl.org/CategoryAPI.

20. —. SDL_ttf Documentation - kapitola 3.4 Render. *LibSDL.org.* [Online] 13. Listopad 2009. [Citace: 28. Duben 2021.] https://www.libsdl.org/projects/SDL_ttf/docs/SDL_ttf_frame.html.

21. **Indiana University.** What is the Latin-1 (ISO-8859-1) character set? *Knowledge Base.* [Online] Indiana University, 18. Leden 2018. [Citace: 28. Duben 2021.] https://kb.iu.edu/d/aepu.

22. **JONES, Kelly.** ASCII and UTF-8 Table. [Online] [Citace: 28. Duben 2021.] https://kellykjones.tripod.com/webtools/ascii_utf8_table.html.

23. **Unicode, Inc.** Basic Questions. *Unicode – The World Standard for Text and Emoji.* [Online] Unicode, Inc., 2021. [Citace: 28. Duben 2021.] https://www.unicode.org/faq/basic_q.html.

24. **SDL: Simple DirectMedia Layer.** SDL_ttf Documentation. *LibSDL.org.* [Online] [Citace: 28. Duben 2021.] https://www.libsdl.org/projects/SDL_ttf/docs/.

25. —. SDL_image Documentation. *LibSDL.com.* [Online] [Citace: 28. Duben 2021.] https://www.libsdl.org/projects/SDL_image/docs/.

26. —. SDL_mixer Documentation. *LibSDL.org.* [Online] [Citace: 28. Duben 2021.] https://www.libsdl.org/projects/SDL_mixer/docs/.

27. **Freepik.** Settings free icon. *Flaticon.com.* [Online] [Citace: 28. Duben 2021.] https://www.flaticon.com/free-icon/settings_2099058?term=settings&page=1&position=2&page=1&position=2&related_id=2099058&origin=search.

28. **SDL: Simple DirectMedia Layer.** SDL Releases - Download. *SDL: Simple DirectMedia Layer.* [Online] [Cited: Duben 28, 2021.] https://www.libsdl.org/download-2.0.php.

29. **NOVAK, István, a další.** *Visual Studio 2010 and .NET 4 Six-in-One: Visual Studio, .NET, ASP.NET, VB.NET, C#, and F#.* místo neznámé : Wrox, 2010. ISBN: 978-0-470-49948-1.

30. **Albatross.** History of C++. *cplusplus.com.* [Online] https://www.cplusplus.com/info/history/.

31. **Microsoft.** Microsoft Details Pricing and Licensing for Visual Studio 2005 and Simplifies Microsoft Developer Network Subscriptions. *Microsoft.* [Online] Microsoft, 21. Únor 2005. https://news.microsoft.com/2005/03/21/microsoft-details-pricing-and-licensing-for-visual-studio-2005-and-simplifies-microsoft-developer-network-subscriptions/.

32. **HEROUT, Pavel.** *Učebnice jazyka C.* České Budějovice : Kopp, 2009. ISBN 978-80-7232-383-8.

13 Seznam obrázků