Gymnázium Christiana Dopplera, Zborovská 45, Praha 5

ROČNÍKOVÁ PRÁCE

Knihovna pro geometrii v rovině

Vypracoval: Jiří Cihelka
Třída: 8. M
Školní rok: 2023/2024
Seminář: Seminář z Programování

Prohlašuji, že jsem svou ročníkovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím s využíváním práce na Gymnáziu Christiana Dopplera výhradně pro studijní účely.

V Praze dne 17. 01. 2024 Jiří Cihelka

Obsah

1. Úvod

Pokud v dnešní době něco programujeme, málokdy píšeme všechen kód sami. Při práci nám totiž pomáhají nástroje a knihovny, které vytvořil již někdo před námi. Jejich přítomnost bereme za tak samozřejmou, že se často ani nezamýšlíme nad tím, jaké to je knihovny vytvářet.

Tato ročníková práce se bude zabývat tvorbou knihovny. Soustředit se, mimo jiné, bude i na rozdíly od tvorby klasického programu.

1.1 Cíl práce

Cílem práce je vytvořit jednoduchou, ale snadno rozšířitelnou knihovnu pro geometrii v rovině, vytvořit k ní dokumentaci a přiblížit čtenáři proces tvorby.

2. Nástroje a jazyk

2.1 Programovací jazyk

Pro tvorbu knihovny jsme si vybrali jazyk TypeScript[9]. Jedná se o nadstavbu jazyka JavaScript, která přidává statické typování a další vylepšení[42]. Jazyk je svou strukturou velmi podobný jazyku C#. TypeScript je kompilován do JavaScriptu[43], který je následně spouštěn v prohlížeči, zároveň zajišťuje lepší kompatibilitu napříč prohlížeči[1][45].

2.2 Nástroje

Při tvorbě kvalitní a stabilní knihovny je v podstatě nezbytné využít standardních nástrojů pro vývoj softwaru. Tato kapitola se bude zabývat nástroji, které jsme použili při tvorbě knihovny.

2.2.1 Verzovací systém

Verzovací systém je nástroj, který slouží k evidenci změn v kódu. Umožňuje nám také vytvářet větve, které můžeme následně sloučit do hlavní větve[25, 54]. Větve nám umožňují pracovat na více funkcionalitách zároveň, aniž bychom museli mít všechny hotové. Také nám umožňuje v případě potřeby se vrátit k předchozí verzi kódu. Pokud nalezneme chybu, můžeme v historii najít, jak chyba vznikla a podniknout kroky k eliminaci podobných chyb v budoucnu.

Pro tuto práci jsme zvolili verzovací systém Git. Jedná se o nejpoužívanější verzovací systém, který je zdarma a open-source[24, 9]. Zároveň má podporu prakticky ve všech standardních vývojových nástrojích a existuje mnoho serverů, které vám umožní zdarma hostovat váš repozitář. Jako hostovací server jsme vybrali GitHub, jelikož se jedná o nejčastěji používaný server pro open-source projekty[46].

2.2.2 Správce balíčků

Správce balíčků je nástroj, který slouží k instalaci a aktualizaci knihoven a dalších závislostí. Také slouží k alespoň částečné standardizaci struktury projektu[2]. U moderních jazyků je běžné, že většina knihoven je dostupná přes správce balíčků, který je součástí standardní instalace jazyka[3][26, 37, 34, 36].

[1] Kompatibilitu zajišťuje tím, že klíčová slova a API, která jsou dostupná až v novějších verzích JavaScriptu, transpiluje do starších verzí.

[2] Částečnou standardizací je myšleno, že definuje například kompilační kroky a parametry na jednom místě, nebo obsahuje soubor se standardizovaným názvem, který ve strojově i přirozeně čitelném formátu (typicky YAML nebo JSON) poskytuje informace o daném projektu.

[3] Nebo je ho možno velmi jednoduše přidat.

V případě TypeScriptu (ve finále spíše JavaScriptu) je toto poněkud složitější, jelikož nemá standardní instalaci a běžně se spouští přímo v prohlížeči. Existují však i neprohlížečová prostředí, která umožňují spouštět JavaScriptový kód přímo v příkazové řádce[4]. Jedním takovým prostředím je Node.js, které je založeno na jádře V8[35], které používá i prohlížeč Google Chrome[39]. Součástí Node.js je také správce balíčků npm[26, 9], který je nejčastěji používaným správcem balíčků pro JavaScript a TypeScript. Existuje k němu i pro veřejné balíčky bezplatný repozitář, který je dostupný na adrese `https://www.npmjs.com/`. Tento repozitář je používán i jinými správci balíčků, například pnpm[49].

2.2.3 Kompilátor

Důležitým „build stepem“ knihovny je kompilace. Kompilujeme zdrojový kód v TypeScriptu do JavaScriptu, který je spustitelný v prohlížeči i v Node.js. TypeScript je, obdobně jako C#, jazyk vytvořený Microsoftem[43]. Microsoft k jazyku tak poskytuje i kompilátor.

Existují i jiné kompilátory, které jsou v některých ohledech[5] lepší, ale vzhledem k tomu, že je kompilátor od Microsoftu nejrozšířenější, rozhodli jsme se ho použít.

2.2.4 Testování

Automatické testování je důležité pro zajištění kvality kódu v každém projektu. U knihoven je důkladné testování celého kódu ještě důležitější, jelikož se jedná o kód, na který se spoléhají jiné projekty.

Pro testování jsme zvolili framework Jest[9]. Důvodem byly především naše předchozí zkušenosti s tímto frameworkem. Protože má všechny potřebné funkce a je velmi jednoduchý na použití, neviděli jsme důvod proč zvolit jiný framework.

2.2.5 Dokumentace

Dokumentace je důležitá pro každý projekt. Dokumentace knihovny se dělí na dvě části.

Vývojářská dokumentace

Vývojářská dokumentace je určena pro hlubší pochopení kódu knihovny. Hodí se například pokud budeme chtít knihovnu rozšířit o nové funkce či hledat zdroje chyb. GitHub má zabudovanou podporu pro vývojářskou dokumentaci[47], která je psaná ve formátu Markdown[6]. Markdown je jednoduchý formát, který umožňuje psát text s krátkými formátovacími značkami[48]. Vývojářská dokumentace je dostupná na adrese `https://github.com/geometryjs/geometry.js/wiki/` a je manuálně aktualizována. Jedná se také o hlavní zdroj této práce.

[4]Spouští ho prostřednictvím interpreteru v prostředí uzpůsobeném pro fungování mimo prohlížeč.

[5]Například umí efektivněji paralelizovat kompilaci nebo využívat cache.

[6]GitHub wikis podporuje i jiné formáty než Markdown, jsou ale méně používané.

Uživatelská dokumentace

Uživatelská dokumentace (nebo také API dokumentace) je určena pro uživatele knihovny, tedy pro programátory, kteří knihovnu používají. Ta vychází přímo z kódu knihovny a komentářů v něm. Je automaticky generována systémem TypeDoc[7][9] a je dostupná na adrese `https://geometryjs.jiricekcz.dev/api/`. Obsahuje soupis všech tříd, funkcí a proměnných, které jsou dostupné pro uživatele knihovny a jejich popis.

[7]Spouštění generování je zajištěno přes GitHub Action, která aktivuje „rebuild“ stránky na Cloudflare Pages[2].

3. Struktura knihovny

3.1 Základní myšlenka

Knihovna je inspirována programem GeoGebra, který je určen pro tvorbu geometrických konstrukcí. Cílem je programátorovi umožnit snadno vytvářet geometrické objekty pomocí závislostí mezi nimi. Objekty by si zároveň měly pamatovat, které objekty na nich závisí, aby se při změně objektu automaticky přepočítaly všechny závislé objekty. Objekty by také měly obsahovat metody na výpočet nejběžnějších vlastností.

3.2 Architektura

Jasně stanovit principy architektury knihovny je důležité pro její přehlednost a udržitelnost. Stanovením těchto principů ze začátku se můžeme mimo jiné vyvarovat stavu, kdy se bude knihovna skládat z několika částí, jejichž struktura bude dána stavem mysli autora v době psaní kódu, nikoli rozumnou úvahou. Můžeme se tím však dostat do stavu, kdy jsme si v začátku stanovili principy, které se ukážou jako nevhodné. Protože jsme se těmito principy v takové situaci řídili při psaní kódu, může být obtížné je v průběhu projektu změnit bez nutnosti přepsat velkou část kódu.

Jelikož se nám koncept této knihovny jevil jako poměrně jednoduchý, rozhodli jsme se pro poměrně striktní nastavení pravidel architektury již ve velmi rané fázi projektu[3].

3.2.1 Dědičnost a kompozice

V objektově orientovaném programování máme ke strukturování dva hlavní přístupy - dědičnost[53] a kompozici[1].

TypeScript nám dovoluje používat oba přístupy, ale přiklání se spíše k dědičnosti. Dědičnost zajišťuje standardním způsobem pomocí klíčového slova `extends`[27]. Kompozici nám dovoluje pouze typovou a to pomocí rozhraní a klíčového slova `implements`[40]. Nedovoluje nám dědit z více tříd, ale umožňuje implementovat více rozhraní[40]. Přestože existují způsoby, jak obejít toto omezení[44], většinou je lepší se jich vyvarovat, jelikož přináší zbytečnou složitost a obecně méně přehledný a srozumitelný kód.

Pro knihovnu jsme tedy zvolili přístup jednoduše shrnutelný jako *kompozice pro strukturu, dědičnost pro implemetaci*[3]. V praxi se tento princip projevuje tak, že máme několik rozhraní, jejichž kombinacemi určujeme typ parametrů a návratových hodnot metod. Tyto kombinace jsou poté implementovány (většinou pomocí dědičnosti minimalizuje duplikaci kódu).

[1] Existuje několik typů kompozic, jedním je obsažení jednotlivých komponentů jako vlastnosti třídy, jiné může být prostřednictvím tzv. „traitů“[38]. Typová kompozice se spíše podobná metodě „traitů“.

Tento přístup jsme zvolili především protože u dědičnosti se můžeme velmi rychle a jednoduše dostat do situace, kdy nebudeme schopni přidat novou funkcionalitu, aniž bychom zásadně změnili stávající strukturu kódu[2]. To je krajně nevhodné, pokud tuto strukturu odhalujeme uživateli knihovny, jelikož i přidání malé funkcionality může znamenat „breaking change“. Struktura daná kompozicí rozhraní je mnohem flexibilnější a transparentnější v tom, co umožňuje a co ne[3].

Knihovna je tedy většinově strukturována tak, aby k jejímu standardnímu používání bylo potřeba „importovat“ pouze rozhraní a funkce[3]. Výjimkou jsou třídy, které mají z podstaty věci vždy jen jednu implementaci s jednoduchou hierarchií dědičnosti (jedná se například o třídy představující průsečíky jiných objektů)[16].

[2]Při používání dědičnosti musíme balancovat hloubku „stromu dědičnosti“ - „mělký“ strom zaručí lepší přehlednost, „hlubší“ lépe popíše skutečné vztahy mezi třídami.

[3]V našem případě tomu tak je především protože kompozici pomocí rozhraní v TypeScriptu zajišťují typy využívané pouze při kompilaci. Nemá tedy žádné implikace při „runtimu“. To také znamená, že můžeme v krizovém případě kladené požadavky obejít.

4. Objekty

Základním objektem knihovny je `Plane`[1][20]. Jedná se o rovinu, ve které se budou všechny objekty nacházet. Zároveň je to objekt, který bude obsahovat metody na vytváření nových objektů[6].

4.1 Kategorizace objektů na základě některých vlastností

Pro přehlednost a snadnější orientaci v knihovně rozlišuje dokumentace několik kategorií objektů[2][6].

4.1.1 `ObjectWithType`

Do kategorie[3] `ObjectWithType` patří objekty, které reprezentují geometrické objekty. Jediným požadavkem na objekty v této kategorii je, aby měly vlastnost `objectType`, která určuje typ objektu, a vlastnost `virtual`, která určuje, zda se jedná o virtuální objekt (o tom více v další sekci). Pokud tedy někde bereme jako parametr objekt typu `ObjectWithType`, víme, že jsme rychle schopni zjistit, jaký geometrický objekt reprezentuje.[6, volný překlad]

4.1.2 Virtuální a nevirtuální objekty

Objekty z kategorie `ObjectWithType` mohou být virtuální[4] nebo nevirtuální. Virtuálními objekty se rozumí takovým objektům, které nereprezentují geometrický objekt, který by bylo možné umístit do roviny (například vektor nebo číslo). Nevirtuální jsou tedy objekty, které reprezentují geometrické objekty, které je možné umístit do roviny (například bod nebo přímka).[6, volný překlad]

Toto dělení se netýká objektů nespadajících do kategorie `ObjectWithType`.

4.1.3 „Bound“ a „Unbound“ objekty

Aby byla knihovna schopna správně reagovat na změnu objektu, musí vědět, které objekty od něj závisí. Toto je zajištěno promocí Grafu závislostí (viz sekce 5).

Objekty, které označujeme jako „bound“ jsou objekty, které jsou součástí grafu závislostí. Ty, které nejsou součástí grafu závislostí, označujeme jako „unbound“.

[1] Pokud se zde zmiňujeme o objektech `NázevObjektu`, myslíme tím vždy objekt, který implementuje rozhraní `NázevObjektu`.

[2] Rozdělením objektů je zde myšleno rozdělení tříd nebo rozhraní, nikoli konkrétních instancí těchto tříd nebo tříd implementujících tyto rozhraní.

[3] Jedná se zároveň i o rozhraní.

[4] Pojem virtuální se v jiných programovacích jazycích využívá k popisu metod, které lze „overridnout“ ve třídě, která tuto metodu dědí. V TypeScriptu nejsou žádné virtuální metody (respektive jsou všechny virtuální, a proto to není nutno specifikovat), využíváme tedy tohoto termínu pro jiný účel.

„Bound“ objekty jsou vytvářeny pomocí metod objektu `Plane` (viz sekce 4.2) a jsou brány za dlouho žijící objekty. Není tedy vhodné je vytvářet v cyklu či například v každém snímku animace. Jsou optimalizovány pro rychlé přepočítávání vlastností při změně závislých objektů, nikoli pro rychlé vytváření a mazání. Typicky z nich složíme scénu (to si můžeme představit stejně jako je tomu v GeoGebře). Musíme si taky při jejich vytváření dávat pozor na to, aby nedošlo k „memory leaku“[5]

„Unbound“ objekty slouží především pro přenos dat mezi funkcemi. Jejich hodnoty nejsou závislé na žádných jiných objektech a neaktualizují se při změně jiných objektů. Jejich vytváření nemá přílišný vliv na výkon[6], tudíž je možno je vytvářet v cyklu či v každém snímku animace a používat je k výpočtům.[6, volný překlad]

4.2 Vytváření objektů pomocí objektu `Plane`

Metody objektu `Plane` nám umožnují vytvářet nové objekty. Tyto metody se (i podle API dokumentace) dělí do několika skupin - metody typu `create`, `construct` a `extract`[7][6]. Těmito metodami vytváříme výhradně „bound“ objekty. „Unbound“ objekty vytváříme například metodami jiných tříd, které ovšem nemají vlastní kategorizaci.

4.2.1 Metody typu `create`

Metody typu `create` slouží k vytváření objektů z jednoho nebo více objektů a/nebo hodnot zjevnou[8] cestou[6]. Názvy metod tohoto typu jsou většinou odvozeny od názvů vstupních objektů a/nebo hodnot a mohou vypadat například takto:

```
create[TypVýsledku]From[TypParametru1]And[TypParametru2]()
```

4.2.2 Metody typu `construct`

Metody typu `construct` slouží k vytváření objektů z jednoho nebo více objektů a/nebo hodnot nezjevnou[9] cestou [6]. Názvy metod tohoto typu se neřídí striktními pravidly, pouze se snaží být co nejvíce popisné, například:

[5] JavaScript je jazyk s „garbage collectorem“[30], který se stará o uvolňování paměti. Jedná se o velmi složitý proces, ale lze ho snadno shrnout tak, že pokud na objekt není žádná reference, je uvolněn. Na „bound“ objekty je vždy držena reference, tudíž se nikdy neuvolní.

[6] Vliv není větší než při standardní instanciaci objektů v JavaScriptu.

[7] Metodou typu `typMetody` se v tomto případě chápe metoda jejíž název začíná sekvencí znaků *typMetody*. Zároveň má odpovídající „tag“ v API dokumentaci[1]

[8] Zjevnou cestou se zde rozumí tomu, že pokud známe typy vstupních objektů a typ výstupu, je nám jasné, jakým procesem ze vstupu výstup získáme. Jedná se například o vytvoření přímky ze dvou bodů (je zjevné, že přímku vytvoříme tak, aby oběma body procházela).

[9] Nezjevnou cestou se rozumí takové konstrukci, u které je nutné k jednoznačnosti procesu přidat i název konstrukce. Jedná se například o vedení kolmice k přímce bodem (pokud bychom zadali pouze přímku a bod, není jednoznačně jasné, jak k výsledné přímce máme přijít).

`constructPerpendicularLineFromPoint()`

4.2.3 Metody typu `extract`

Metody typu `extract` slouží k „boundnutí“ objektů. Typicky jsou využity v případech, kdy chceme získat nějakou vlastnost objektu, která je běžně vrácena metodou jako „unbound“ objekt, jako „bound“ objekt[6]. Názvy metod tohoto typu jsou většinou odvozeny od názvů vstupního objektu a extrahované vlastnosti a mohou vypadat například takto:

`extract[PopisVýsledku][TypVýsledku]From[TypParametru]()`

5. Graf závislostí

Graf závislostí obsahuje informace o vztazích mezi „bound“ objekty. Jedná se o orientovaný necyklický graf[51], jehož vrcholy jsou objekty a hrany reprezentují závislosti mezi nimi[4]. Orientovanost grafu je implikována tím, že se jedná o graf necyklický.

Graf je v paměti uložen tak, že každý objekt typu `DependencyNode` obsahuje `Iterable`[1] objektů, které na něm závisí, a `Iterable` objektů, na kterých závisí on[11, line 17, 21]. Elementem tohoto grafu je vždy objekt typu `DependencyNode`.

5.1 Přidávání objektů do grafu

O přidávání objektů do grafu závislostí se nemusí starat uživatel knihovny. Objekt je přidán vždy při instanciaci třídy. Toto je zajištěno pomocí parametru `dependencies` konstruktoru třídy `DependencyNode`[12, line 13-17]. Tento parametry je při dědění tříd vždy předán konstruktoru rodičovské třídy. Můžeme tak mít libovolný počet „mezitříd“, dokud se nedostaneme k dost konkrtétní třídě, která už má stanoveno, na kterých objektech závisí.

O přidání reference druhého směru (tedy z objektu na kterém závisí nově vytvořený objekt na nově vytvořený objekt) se stará metoda `registerDependant`[11, line 32].

5.2 Procházení grafem

Procházení grafem je možné libovolným algoritmem pro procházení grafu. Pokud budeme iterovat přes `deepDependencies`[11, line 17] nebo přes `deepDependants`[11, line 21], využíváme algoritmu pro prohledávání do hloubky[50]. Jelikož se jedná o orientovaný graf, nikoli strom, je nutné si pamatovat již navštívené vrcholy, abychom se nezacyklili. K tomu využíváme vestavěný datový typ `Set`.

5.3 Zajištění necykličnosti grafu

Protože při výpočtu vlastností jednotlivých objektů budeme rekurzivně počítat potřebné vlastnosti všech objektů, na kterých dané objekty závisí, je nutné zamezit cyklickým závislostem. Tohoto snadno docílíme tak, že nedovolíme změnit závislosti po vytvoření objektu a budeme všechny závislosti požadovat již při vytváření objektu. V kódu tohoto docílíme tak, že metodu na přidávání závislostí (metoda `registerDependency`[12, line 20-23]) uděláme `protected` a zavoláme ji pouze v konstruktoru[2].

[1] `Iterable` je vestavěné rozhraní, které reprezentuje objekt, kterým lze iterovat.

[2] Je stále teoreticky možné vytvořit cyklickou závislost, ale lze prakticky vyloučit, že bychom ji vytvořili „omylem“, jelikož kód k tomu potřebný je tak nestandardní, že si nelze neuvědomit, že děláme něco špatně.

6. Procedury

Knihovna potřebuje (nebo alespoň v budoucnu může potřebovat) velké množství matematických výpočtů. Abychom se vyhnuli opakování a umožnili lepší segmentaci kódu, abstrahujeme tyto výpočty do tzv. „procedur“[8].

6.1 Definice procedury

Procedurou nazveme „pure“ funkci, která bere jako parametr jeden JavaScriptový objekt a vrací jeden JavaScriptový objekt[8].

6.2 Implementace procedury

Procedury budeme implementovat jako třídy implementují rozhraní `Procedure`[21]. Pro zjednodušení implementace máme k dispozici třídu `Procedure`[22].

6.2.1 Využití třídy

Někteří si možná mohou položit otázku, proč využíváme třídu, když se jedná o pouhou funkci. Důvodem je, že procedury jsou takto objekty a můžeme jim tedy přidávat další metody nebo vlastnosti. Zatím této možnosti knihovna nevyužívá[22], ale v budoucnu by mohla být užitečná.

6.3 Dokumentace procedur

Jelikož jádrem procedur budou většinou matematické výpočty, jejichž kódová implementacenemusí být dostatečně deskriptivní, je důležité, aby procedury byly dobře zdokumentované. K tomu máme dedikovanou stránku dokumentace na wiki[8].

6.4 Typování procedur

Jelikož procedury dědí z jedné třídy, ale každá procedura má jiný typ vstupního a výstupního objektu, je nutné využít generických typů[41]. Vstupní a výstupní typ procedury jsou definovány jako generické typy rozhraní `Procedure`[21, line 6].

6.5 Dělení procedur

Procedury dělíme do dvou kategorií - procedury „základní“ a procedury „odvozené“[8].

6.5.1 Základní procedury

Základními procedurami jsou procedury, které nepotřebují specifickou dokumentaci. Jejich cíl a postup je běžně známý nebo často popisovaný v jiných zdrojích a stačí nám k tomu pouze odkaz na tyto zdroje.

Příkladem může být procedura `VectorAddition`[23, line 9-23], která sečte dva vektory, nebo procedura `VectorDotProduct`[23, line 88-99], která spočítá skalární součin dvou vektorů.

6.5.2 Odvozené procedury

Odvozené procedury jsou procedury, které potřebují specifickou dokumentaci[8]. Tyto procedury jsou specifičtější a často například nemají vlastní název. Jako jejich název použijeme několik slov vystihující, co procedura dělá.

Příkladem může být procedura `LineLineIntersection`[18, line 11-27] nebo procedura `LineCCoefficient`[17, line 11-20].

7. Pomocné struktury

Každý rozsáhlejší softwarový projekt má své pomocné funkce a třídy. Jedná se typicky o funkce a třídy využívané na více místech, které ale zároveň nejsou specifické pro daný projekt. Často to jsou například funkce pro práci s datovými strukturám nebo přímo implementace těchto datových struktur.

7.1 Pomocné třídy

Jelikož zatím knihovna nepoužívá žádné složitější datové struktury, pomocná třída je pouze jedna[1][19].

7.1.1 Pomocná třída Cache

Jak již bylo zmíněno v kapitole 5, knihovna využívá graf závislostí. Když už je tento graf k dispozici a všechny objekty jsou v něm uloženy, je vhodné si výpočty, které již byly provedeny, někde uložit. Každý objekt implementující rozhraní `DependencyNode` obsahuje metodu `update`[11, line 26]. Tato metoda je volána vždy, když se změní nějaký objekt, na kterém daný objekt závisí, můžeme ji proto spojit s čištěním cache.

Rozhraní `Cache`

Rozhraní `Cache`[10] nám definuje jak by implementacepomocných tříd měla vypadat. Rozlišuje, zda je cache „NonEmpty“. Pokud je cache „NonEmpty“, musí všechny jeho prvky být vždy definované. Pokud cache „NonEmpty“ není, mohou jeho prvky být `undefined`[2].

Rozšířením rozhraní `Cache` je rozhraní `IterableCache`[10, line 49 - 54], které nám zajišťuje, že cache je iterovatelná.

Implementace rozhraní `Cache`

Implementace rozhraní `Cache` je třída `MemoryMapCache`[19, line 7 - 33] a její rozšíření `MemoryMapCacheWithAutomaticCalculation`[19, line 38 - 59]. Obě tyto třídy jsou implementovány pomocí vestavěného rozhraní `Map`[29].

Třída `MemoryMapCacheWithAutomaticCalculation` v konstruktoru přijímá funkce pro výpočet hodnoty, pokud není v cache nalezena. Tím tato třída zajišťuje, že vždy, když se pokusíme získat hodnotu z cache, získáme ji (pokud existuje, tak z cache, pokud neexistuje, tak ji vypočítáme)[7].

[1] Exportovány jsou dvě, ale obě jsou velmi podobné a není třeba je rozebírat jednotlivě.

[2] TypeScript zajišťuje „null safty“ pomocí „type unions“ s typem `undefined`.

7.2 Pomocné funkce

Nejčastějším typem pomocných struktur jsou funkce.

7.2.1 Funkce pro zjišťování rovnosti

Úkon, který se na první pohled může zdát triviální, je zjištění rovnosti dvou čísel. V JavaScriptu existuje pouze jeden typ čísel, a to `number`. Specifikace nám nezaručuje, jakým datovým typem bude číslo reprezentováno[3], musí být ale schopné dosáhnout přesnosti a rozsahu 64-bitového čísla s plovoucí desetinnou čárkou[31].

Porovnávání čísel s plovoucí desetinnou čárkou je však známý problém[55]. Pokud k výsledku, který by matematicky měl být ekvivalentní, dojdeme dvěma různými cestami, může se stát, že výsledky budou rozdílné[4]. To je způsobeno tím, že mezivýsledky jsou vždy zaokrouhlovány na konečný počet desetinných míst.

Řešením tohoto problému je porovnávání s přesností menší než je maximální přesnost proměnné dle standardu *IEEE 754*. Přesnost je nastavována globálně. Výchozí hodnota je 43 bitů, to je o 10 bitů méně než je maximální přesnost proměnné typu `number` v JavaScriptu. Toto nastavení je vhodné pro většinu případů.

Ve speciálních případech (například `NaN`, `Infinity`, `-Infinity`) se řídíme standardním chováním operátoru `===`.[7]

Funkce pro porovnávání s nulou

Jediným okrajovým případem je porovnávání s nulou. V tomto případě nelze porovnat pouze s určitou přesností, jelikož absolutní přesnost nuly není známa. Pro tento případ existuje globální nastavení `unit`. To nám udává číslo, kolem kterého se budou výsledky pohybovat. Platí, že číslo x je rovno 0 právě tehdy, když $x + unit$ je rovno $unit$ podle podmínek stanovených pro obecnou rovnost čísel výše.[13]

7.2.2 Získávání názvu typu objektu

Naše knihovna je napsána v TypeScriptu a tak v dokumentaci i ve zdrojovém kódu pracujeme s typy. JavaScript však typy nezná[28] a tak pokud dojde k typové chybě, nejsme schopni uživateli popsat, co za typovou chybu udělal. Abychom ale mohli uživateli alespoň částečně napověděli, co udělal za chybu, vytvořili jsme funkci `getTypeString`[15], která se pokusí odhadnout, jak by vypadal TypeScriptový typ objektu, který ji poskytneme.

[3] JIT může v některých případech zkompilovat čísla do 32-bitových „integerů“. Zajišťuje nám však, že chování bude ekvivalentní jako při použití 64-bitového „floatu“

[4] Rozdlínými výsledky jsou myšleny výsledky s rozdílnou bitovou reprezentací dle standardu *IEEE 754*[52]

7.3 Pomocné typy

TypeScript má velmi silný typový systém, který nám umožňuje vytvářet vlastní typy. Komplexita typů může být tak vysoká, že by snížila čitelnost kódu, kdyby byly všechny vyvářeny na místech, kde se používají. Některé typy se také mohou hodit na více místech. Nejčastěji používané typy, které přímo nesouvisí s náplní knihovny, jsou proto vytvořeny jako pomocné typy.

7.3.1 Typ `Some`

Často se nám stane, že chceme typově popsat libovolnou hodnotu, která není `undefined` nebo `null`. Takový typ v TypeScriptu vestavěný není, ale můžeme si ho definovat sami. Definujeme proto typ `Some`[14, line 4], který popisuje všechny hodnoty kromě `undefined` a `null`.

8. Chybové hlášky

Další nezbytnou součástí knihovny jsou chybové hlášky. Ne vždy uživatel knihovny použije všechny metody správně a obzvláště v jazyce bez striktního statického typování[1]. Když k takové chybě dojde, je vhodné uživateli poskytnout co nejvíce informací o tom, proč k chybě došlo, popřípadě, pokud chybu očekává, mu umožnit ji vyřešit v kódu. V TypeScriptu se „error handling“ standardně řeší pomocí `try` a `catch` bloků a klíčového slova `throw`[32].

My rozlišujeme dva typy chyb - fatální a nefatální. Pokud dojde k problému nefatálnímu (například nedefinovaný výraz[2] ve výpočtu), vrátí daná metoda nebo funkce `NaN`[5]. Nefatální chyba se propaguje výpočty a projeví se tedy pouze pokud ovlivní hodnotu, kterou uživatel knihovny používá[5].

Fatální chyba je chyba, která způsobí, že některá z metod nebo funkcí nemůže pokračovat v práci, nebo je to chyba, která je považována za tak závažnou, že by mohla způsobit obecně chaotické chování knihovny. Takové chyby jsou řešeny pomocí `throw` a jednoho z „error typů“, které knihovna poskytuje. Tyto chyby jsou pak řešeny pomocí `try` a `catch` bloků.

Chyby jsou navrženy tak, aby při standardním používání knihovny nebylo nutné použít `catch` bloky[5], protože značně zpomalují kód[33].

[1] Nejčastěji nastává problém při použití JavaScriptu. Může nastat i v TypeScriptu při nadužívaní `any` typu nebo s použitím například *ts-ignore* a *ts-expect-error* direktiv.

[2] Za nedefinovaný výraz se považuje například $\frac{0}{0}$

9. Závěr

Výsledkem této práce je knihovna *GeometryJS* dostupná na *GitHubu*[1] a *NPM*[2] spolu s její dokumentací. Knihovna dokáže pracovat s body, vektory, přímkami a hodnotami, ale je snadno rozšířitelná o další geometrické objekty.

9.1 Porovnání se standardním softwarovým projektem

Při vytváření knihovny si můžeme všimnout mnoha rozdílů oproti vytváření klasického softwarového projektu.

Musí být kladený zvýšený důraz na stabilitu exportovaných struktur, při tom ale musí být zachována flexibilita pro rozšíření. Také je nutné myslet na to, že knihovna bude používána v mnoha různých prostředích, tudíž je nutné minimalizovat požadavky na prostředí, ve kterém knihovna funguje. Deskriptivní názvy tříd, vlastností a metod jsou také důležitější, jelikož se v nich bude muset orientovat mnoho uživatelů knihovny. Je též potřeba dbát na optimalizaci výkonu, protože nemůžeme efektivně předpovídat v jakých prostředích a jak budou uživatelé knihovnu používat a co za požadavky na výkon na ni budou mít.

Na druhou stranu není třeba vytvářet grafické uživatelské rozhraní a při tvorbě negrafického uživatelského rozhraní můžeme předpokládat alespoň základní znalost programování uživatele. Pro publikaci a dokumentaci můžeme používat nástroje, které by uživatel-neprogramátor nemusel být schopen použít, ale pro nás jako vývojáře jsou jednodušší a rychlejší.

[1] `https://github.com/geometryjs/geometry.js`

[2] `https://www.npmjs.com/package/@jiricekcz/geometry.js`

Literatura

[1] Jiří Cihelka. Geometryjs - api reference. https://geometryjs.jiricekcz.dev/api/, Dec 2023.

[2] Jiří Cihelka. Geometryjs - wiki (api documentation). https://github.com/geometryjs/geometry.js/wiki/API-Documentation, Dec 2023.

[3] Jiří Cihelka. Geometryjs - wiki (code structure). https://github.com/geometryjs/geometry.js/wiki/Code-structure, Dec 2023.

[4] Jiří Cihelka. Geometryjs - wiki (dependency graph). https://github.com/geometryjs/geometry.js/wiki/Dependency-graph, Dec 2023.

[5] Jiří Cihelka. Geometryjs - wiki (errors). https://github.com/geometryjs/geometry.js/wiki/Errors, Dec 2023.

[6] Jiří Cihelka. Geometryjs - wiki (geometry objects). https://github.com/geometryjs/geometry.js/wiki/Geometry-objects, Dec 2023.

[7] Jiří Cihelka. Geometryjs - wiki (helpers). https://github.com/geometryjs/geometry.js/wiki/Helpers, Dec 2023.

[8] Jiří Cihelka. Geometryjs - wiki (procedures). https://github.com/geometryjs/geometry.js/wiki/Procedures, Dec 2023.

[9] Jiří Cihelka. Geometryjs - wiki (technologies). https://github.com/geometryjs/geometry.js/wiki/Technologies, Dec 2023.

[10] Jiří Cihelka. Geometryjs - cache.ts. https://github.com/geometryjs/geometry.js/blob/v1.0.0/src/interfaces/cache.ts, Jan 2024.

[11] Jiří Cihelka. Geometryjs - dependencynode.ts. https://github.com/geometryjs/geometry.js/blob/v1.0.0/src/interfaces/dependencyNode.ts, Jan 2024.

[12] Jiří Cihelka. Geometryjs - dependencynode.ts. https://github.com/geometryjs/geometry.js/blob/v1.0.0/src/geometryObjects/dependencyNode.ts, Jan 2024.

[13] Jiří Cihelka. Geometryjs - float.ts. https://github.com/geometryjs/geometry.js/blob/v1.0.0/src/helpers/equality/float.ts, Jan 2024.

[14] Jiří Cihelka. Geometryjs - general.ts. https://github.com/geometryjs/geometry.js/blob/v1.0.0/src/helpers/types/general.ts, Jan 2024.

[15] Jiří Cihelka. Geometryjs - gettypestring.ts. https://github.com/geometryjs/geometry.js/blob/v1.0.0/src/helpers/getTypeString.ts, Jan 2024.

[16] Jiří Cihelka. Geometryjs - intersections. https://github.com/geometryjs/geometry.js/tree/v1.0.0/src/geometryObjects/intersections, Jan 2024.

[17] Jiří Cihelka. Geometryjs - lineequation.ts. https://github.com/geometryjs/geometry.js/blob/v1.0.0/src/procedures/derived/lineEquation.ts, Jan 2024.

[18] Jiří Cihelka. Geometryjs - lineline.ts. https://github.com/geometryjs/geometry.js/blob/v1.0.0/src/procedures/derived/intersections/lineLine.ts, Jan 2024.

[19] Jiří Cihelka. Geometryjs - memorymapcache.ts. https://github.com/geometryjs/geometry.js/blob/v1.0.0/src/helpers/cache/memoryMapCache.ts, Jan 2024.

[20] Jiří Cihelka. Geometryjs - plane.ts. https://github.com/geometryjs/geometry.js/blob/v1.0.0/src/interfaces/plane.ts, Jan 2024.

[21] Jiří Cihelka. Geometryjs - procedure.ts. https://github.com/geometryjs/geometry.js/blob/v1.0.0/src/interfaces/procedure.ts, Jan 2024.

[22] Jiří Cihelka. Geometryjs - procedure.ts. https://github.com/geometryjs/geometry.js/blob/v1.0.0/src/procedures/procedure.ts, Jan 2024.

[23] Jiří Cihelka. Geometryjs - vectoroperations.ts. https://github.com/geometryjs/geometry.js/blob/v1.0.0/src/procedures/foundational/vectorOperations.ts, Jan 2024.

[24] Git Community. Git - about. https://git-scm.com/about.

[25] Git Community. Git - branches in a nutshell. https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell.

[26] NPM Community. Npm - documentation. https://docs.npmjs.com/.

[27] MDN contributors. extends - javascript | mdn. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/extends, 1998-2024.

[28] MDN contributors. Javascipt data types and data structures - javascript | mdn. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#objects, 1998-2024.

[29] MDN contributors. Map - javascript | mdn. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map, 1998-2024.

[30] MDN contributors. Memory management - javascript | mdn. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_management, 1998-2024.

[31] MDN contributors. Number - javascript | mdn. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number, 1998-2024.

[32] MDN contributors. try...catch - javascript | mdn. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/try...catch, 1998-2024.

[33] cprcrack, Esailija, asthomas, and Akseli Palén. In javascript, is it expensive to use try-catch blocks even if an exception is never thrown. https://stackoverflow.com/questions/19727905/, Nov 2013.

[34] Golang developers. Go - packages. https://pkg.go.dev/.

[35] Nodejs developers. Nodejs - the v8 javascript engine. https://nodejs.org/en/learn/getting-started/the-v8-javascript-engine.

[36] Python developers. pip - the python package installer. https://pypi.org/project/pip/.

[37] The Rust Project Developers and Open Source Contributors. Rust book - cargo guide. https://doc.rust-lang.org/cargo/guide/.

[38] The Rust Project Developers and Open Source Contributors. Rust book - traits guide. https://doc.rust-lang.org/book/ch10-02-traits.html.

[39] Google and Open Source contributors. v8 javascript engine. https://v8.dev/.

[40] Microsoft. Typescript classes. https://www.typescriptlang.org/docs/handbook/2/classes.html.

[41] Microsoft. Typescript generics. https://www.typescriptlang.org/docs/handbook/2/generics.html.

[42] Microsoft. Typescript handbook. https://www.typescriptlang.org/docs/handbook/intro.html.

[43] Microsoft. Typescript language. https://www.typescriptlang.org/.

[44] Microsoft. Typescript mixins. https://www.typescriptlang.org/docs/handbook/mixins.html.

[45] Microsoft. Typescript tsconfig reference. hhttps://www.typescriptlang.org/tsconfig.

[46] GitHub (Microsoft). About. https://github.com/about.

[47] GitHub (Microsoft). About wikis. https://docs.github.com/en/communities/documenting-your-project-with-wikis/about-wikis.

[48] GitHub (Microsoft). Basic writing and formatting syntax. https://docs.github.com/en/get-started/writing-on-github/getting-started-with-writing-and-formatting-on-github/basic-writing-and-formatting-syntax.

[49] pnpm developers. pnpm - fast, disk space efficient package manager. https://pnpm.io/.

[50] Wikipedia contributors. Depth-first search — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Depth-first_search&oldid=1186499373, 2023. [Online; accessed 4-January-2024].

[51] Wikipedia contributors. Directed graph — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Directed_graph&oldid=1186925478, 2023. [Online; accessed 4-January-2024].

[52] Wikipedia contributors. Double-precision floating-point format — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Double-precision_floating-point_format&oldid=1185375352, 2023. [Online; accessed 5-January-2024].

[53] Wikipedia contributors. Inheritance (object-oriented programming) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Inheritance_(object-oriented_programming)&oldid=1192201504, 2023. [Online; accessed 5-January-2024].

[54] Wikipedia contributors. Version control — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Version_control&oldid=1192261201, 2023. [Online; accessed 5-January-2024].

[55] Wikipedia contributors. Floating-point arithmetic — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Floating-point_arithmetic&oldid=1193574336, 2024. [Online; accessed 5-January-2024].

Přílohy

https://github.com/geometryjs/geometry.js/tree/v1.0.0

https://github.com/geometryjs/geometry.js/wiki

https://geometryjs.jiricekcz.dev/api/