Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informačních technologií

CSS Houdini: Programové rozhraní k vlastnostem a hodnotám jazyka CSS

Bakalářská práce

Autor: Jakub Tichý

Studijní obor: Aplikovaná informatika(ai-3p)

Vedoucí práce: Mgr. Daniela Ponce, PhD.

Hradec Králové Duben 2021

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové, dne 27. dubna 2021 Jakub Tichý

Poděkování:

Děkuji vedoucí bakalářské práce Mgr. Daniele Ponce, PhD. za metodické vedení práce, praktické rady a zkušenosti, které byly velice užitečné při zpracování.

Anotace

Bakalářská práce má za úkol seznámit čtenáře s CSS Houdini a ukázat implementaci tohoto nového nativního API. Teoretická část popisuje všechny připravené i nepřipravené části API. Praktická část má za úkol ukázat implementaci API, a to pomocí jak připravených knihoven, tak vlastních řešení.

Klíčová slova: API , CSS, CSS Houdini, Javascript.

Annotation

Title: CSS Properties and Values API

The bachelor thesis aims to acquaint with CSS Houdini and show the implementation of this new native API. The theoretical part of the thesis describes all prepared and unprepared parts of this API. The practical part aims to show the implementation of API using already prepared libraries and their own solutions.

Key words: API , CSS, CSS Houdini, Javascript.

Obsah

A. ÚVOD ... 7

B. CÍL PRÁCE ... 8

C. METODIKA PRÁCE ... 9

D. ANALÝZA PŘÍSTUPU K VÝRAZŮM CSS PROSTŘEDNICTVÍM API ... 10

a) Úvod k CSS Houdini ... 10

b) CSS Object Model (CSSOM) ... 10

c) API v Houdini a jejich podpora ... 12

1) Worklety ... 12

2) Vysokoúrovňové API ... 13

3) Nízkoúrovňové API ... 13

d) Typed Object Model ... 13

1) CSSKeywordValueSerialization ... 15

2) StylePropertyMap ... 15

3) CSSUnparsedValue ... 18

4) CSSStyleValue ... 19

e) Custom Properties and Values API ... 28

f) Paint API ... 33

g) Animation API ... 38

h) Layout API ... 44

i) Font Metrics API ... 46

j) Parse API ... 47

k) Analýza implementace CSS Houdini ... 47

E. ANALÝZA PŘÍMÉHO PŘÍSTUPU K VÝRAZŮM CSS ... 49

a) Layout ... 49

b) Animace ... 52

c) Vykreslování ... 55

F. ZÁVĚR ... 57

G. MOŽNOSTI DALŠÍHO POKRAČOVÁNÍ ... 60

H. SEZNAM POUŽITÝCH ZDROJŮ A LITERATURY 61

I. SEZNAM POUŽITÝCH OBRÁZKŮ 63

J. PŘÍLOHY 65

A. Úvod

V dnešní době jsou webové stránky neodmyslitelnou součástí života a jejich poptávka jde strmě vzhůru. Stejnou měrou stoupají i požadavky na vývojáře. U statické webové stránky jde především o vzhled, o který se starají kaskádové styly. Bez kaskádových stylů si dnešní moderní webovou stránku neumíme představit. Dnes by taková stránka svým vzhledem rozhodně nezaujala. Jak se mění designové trendy, musejí vývojáři přicházet s novými nápady a modifikacemi. Webová stránka, která byla moderní před 10 lety, už dnes svým vzhledem nikoho nezaujme.

Některým vývojářům se ovšem povedlo dosáhnout potencionálního limitu kaskádových stylů. V tomto případě si musíme vypomoci javascriptem. Javascript k samotným CSS přidává spoustu dalších možností, jak oživit stránku. Převážně javascript umožňuje vylepšit chování animací. Pomocí tohoto přístupu můžeme spustit animaci pouze při prvním otevření úvodní stránky. Bez javascriptu by nebylo možné udělat responzivní rozbalovací menu, které se otevírá kliknutím myší. Pomocí javascriptu můžeme dále zachytit interakci uživatele, čímž stránka získá další plusové body. Zkrátka javascript a CSS se staly nedílnou součástí moderních webových technologií.

Bohužel spojení CSS a javascriptu má své limity, a to i přesto, jak silné tyto dva nástroje mohou být. Proto se začal vyvíjet CSS Houdini. Jedná se způsob programování CSS prostřednictvím javascriptu. CSS Houdini pomáhá lépe kontrolovat vykreslovací proces, zlepšuje výkon při vykreslovacím procesu a zároveň zprostředkovává příjemnější přístup k prvkům CSS z prostředí javascriptu.

Stále se však jedná o experimentální technologii, lze tedy očekávat komplikace při implementaci a to i u již připravených modulů.

B. Cíl práce

Cílem práce je analyzovat konkrétní řešenou problematiku kaskádovými styly, zvolit pro konkrétní problematiku vhodné API ze sady CSS Houdini a provést analýzu konkrétního API a jeho implementace.

Pokud by se nový způsob přístupu jevil jako vhodný a výhodnější, mohli bychom tuto metodu aplikovat a otestovat.

Lze očekávat, že při implementaci nové experimentální metody se objeví určité problémy spojené s neúplnou podporou tohoto přístupu. V tomto případě bychom si alespoň ukázali, jak by se implementace měla provádět. Pokud to nebude možné, odkážeme se na zdroje, které měly možnost otestovat přístup, jenž se v našem případě nepovedlo implementovat.

C. Metodika práce

Při vytváření bakalářské práce byl zvolen postup analýzy, implementace, analýzy implementace a vyhodnocení vhodného řešení. Analýza zahrnovala všechny v práci použité technologie.

V implementační části jsme na základě informací získaných z dokumentace a dalších zdrojů, konkrétní řešení aplikovali. Společně s aplikací přístupu CSS Houdini jsme se snažili vytvořit stejný efekt pomocí standardního CSS nebo již známých javascriptových tříd pro práci s CSS. Implementace je znázorněna pomocí obrázku zachycujícího kód, který obsahuje probírané téma kapitoly.

Analytická část měla za úkol porovnat oba zvolené přístupu k vytvoření CSS efektů. Porovnání bylo v některých případech prováděno měřením času pomocí javascriptu. U efektů, které byly již někým otestovány, výsledek byl dohledatelný a důkladně popsaný, jsme se odkázali právě na výsledek tohoto testování.

Vhodné řešení bylo vybráno na základě výkonu, složitosti implementace, podpory technologie v prohlížeči a celkového přínosu.

D. Analýza přístupu k výrazům CSS prostřednictvím API

Tato kapitola je věnována popisu a znázornění rozdílů práce CSS Houdini a přístupů, které používáme dnes.

a) Úvod k CSS Houdini

Houdini je sbírka API pro prohlížeč. Určuje prostřednictvím kódu v javascriptu vykreslovacím strojům, jak má CSS vykreslovat. Tím se dá docílit významného progresu ve výkonu. Implementování této metodiky práce s prohlížečem nebylo v některých případech možné kvůli podpoře prohlížečů [17].

Tento nový přístup umožňuje vývojářům mít mnohem větší kontrolu nad stylováním v určitých momentech vykreslovacího procesu, během uživatelské interakce nebo na základě určitých událostí, které mohou při načítání nebo vykreslování stránky nastat. Houdini je nadále schopen v některých případech vylepšit výkon během vykreslovacího procesu.

b) CSS Object Model (CSSOM)

Aby bylo jasné, v čem může být CSS Houdini užitečný, měli bychom si představit princip fungování CSS. Stejně jako CSS Houdini je také CSS Object Model API sada, která slouží k manipulaci s CSS pomocí javascriptu. CSSOM disponuje, kromě prvků určených k přístupu CSS atributů, spoustou prvků určených pouze ke čtení.

CSS Object Model je API pracující podobně jako DOM pro HTML, ale tento model funguje obdobně pro CSS. Bohužel, co se týká práce javascriptu s CSSOM, je zde javascript dost limitován. Další nevýhodou je nízký výkon. Jde o to, že javascript může pracovat až po načtení DOM a CSSOM. Když k tomu připočteme implementaci CSS do HTML, může to někdy negativně ovlivnit výkon. Pokud se ještě přidají i uživatelské interakce, tak se nahromadí poměrně velký výkonnostní propad.

Ukázka práce CSSOM s atributy CSS:

```
//CSSOM
function CSSOMExamples(){

    let glass = document.querySelector( selectors: '.glass');
    let link = document.querySelector( selectors: '.link');
    let cssom = document.querySelector( selectors: '#cssom');


console.group()
console.log('%c CSSOM', 'color: red')
    console.log("Pozadí DOM elementu body= "
        +window.getComputedStyle(document.body).background);
    console.log("Délka elementů tříd 'link', které jsou použity v menu jako odkazy= "
        +window.getComputedStyle(link).getPropertyValue( property: 'width'));
    console.log("Pozadí elementu třídy 'glass', který slouží jako skleněné pozadí= "
        +window.getComputedStyle(glass).getPropertyValue( property: 'width'));
    console.log("Délka pseudo elementu ::before u třídy= "
        +window.getComputedStyle(link,  pseudoElt: '::before').width);

    //Setování vlastností pomocí CSSOM
    startTime = performance.now();
    cssom.style.setProperty( property: 'color',  value: 'blue');
    console.groupEnd()
    endTime=performance.now();
    console.log(endTime-startTime);
}
CSSOMExamples();
```

Obrázek 1: Ukázka práce CSSOM s HTML elementy.

Na obrázku je vidět ukázka zápisu a práce s CSSOM. Lze vidět práce s HTML DOM elementy, třídami i konkrétními prvky pomocí id selektoru. Zároveň je umožněno získat hodnoty pseudo elementů typu ::before a ::after.

Výstup z prohlížeče:

Changing property by CSSOM.

Color of text changed by CSSOM.

Obrázek 3: Ukázka výstupu práce CSSOM s textem.

Výstup z konzole:

```
▼ console.group                                                                    typedOM.js:8
    CSSOM                                                                          typedOM.js:9
   Pozadí DOM elementu body= rgba(0, 0, 0, 0) none repeat scroll 0% 0% / auto     typedOM.js:10
   padding-box border-box
   Délka elemntů tříd 'link', které jsou použity v menu jako odkazy= auto         typedOM.js:12
   Pozadí elemntu třídy 'glass', který slouží jako skleněné pozadí= 963.891px     typedOM.js:14
   Délka pseudo elementu ::before u třídy= auto                                   typedOM.js:16
```

Obrázek 4: Ukázka výstupu práce CSSOM do konzole.

Tato ukázka slouží pouze jako nastínění práce CSSOM, abychom si mohli udělat lepší představu o tom, jak funguje CSS Houdini. Ukázky popisující funkcionality v rámci CSSOM zdaleka nezahrnují všechny funkcionality CSSOM API.

c) API v Houdini a jejich podpora

CSS Houdini je ve své podstatě sada nativních API. Celkem se jedná o 7 sad API, z toho 2 jsou momentálně v rané fázi vývoje. Zbývajících 5 sad je částečně nebo plně podporováno v závislosti na prohlížeči. Největší míru podpory mají prohlížeče Google Chrome, Opera a Microsoft Edge.

Každá sada API má své funkce a vlastnosti. Samotné API se dále dělí na 2 druhy, vysokoúrovňové a nízkoúrovňové API. Všechna vysokoúrovňová API fungují na principu workletů a nízkoúrovňová API slouží spíše jako jejich podpora.

1) Worklety

„Rozhraní Worklet je odlehčená verze Web Workers a poskytuje vývojářům přístup k nízkoúrovňovým částem vykreslovacího kanálu. S Worklety můžete spustit javascript a WebAssembly kód, abyste mohli provádět grafické vykreslování nebo zpracování zvuku tam, kde je vyžadován vysoký výkon.“[2].

Dnes jsou k dispozici 4 druhy workletů. Každý z nich má svůj specifický účel a nelze ho použít pro libovolné výpočty.

- Paint worklet
    - Používá se pro programové generování obrazu. Lze vytvářet vlastní grafické efekty.

- Animation worklet
    - Slouží k výpočtům animací. Lze pomocí workletu vytvářet vlastní animace.
- Audio worklet
    - Je určeno pro zpracování audia.
- Layout worklet
    - Využívá se pro definování umístění elementů.

2) Vysokoúrovňové API

Úzce souvisí s procesem vykreslování v prohlížeči.

- Paint API
    - Slouží k určení vizuálních vlastností jako je barva, pozadí, ohraničení a různé pokročilé vykreslovací efekty.
- Layout API
    - Používá se k určení rozměrů prvků, umístění, poloze a zárovnávání.
- Animation API
    - Využívá se k obsluze animací a vrstev.

3) Nízkoúrovňové API

Aplikuje se jako podpora pro vysokoúrovňové API.

- Typed Object Model API
- Custom Properties & Values API
- Font Metrics API
- Parser API

d) Typed Object Model

Typed Object Model (TOM) zjednodušuje manipulaci s vlastnostmi CSS tím, že namísto připisování atributů jako textových řetězců můžeme pomocí getterů a setterů pracovat s javascriptovými objekty a zapisovat hodnoty v konkrétním formátu. Tento přístup má za následek 2 věci, jednodušší manipulaci s atributy v CSS a z ní plynoucí pozitivní dopad na výkon.

Běžně se k manipulaci CSS v javascriptu používá Element CSS Inline Style, u kterého jsme museli pro práci se styly v javascriptu používat *element.style.property*. Tento přístup je velmi pomalý a těžkopádný. Oproti tomu TOM reprezentuje stejné hodnoty jako klasické javascriptové objekty, s nimiž lze manipulovat a rozumět jejich zápisu snáze než pomocí textových řetězců.

Pomocí TOM lze rozsáhle manipulovat s CSS obdobně jako s jakýmkoliv objektem v javascriptu. Abychom toho mohli docílit, obsahuje v sobě TOM zabudované nativní metody, které umožňují manipulaci s CSS ve velmi rozsáhlém spektru.

Na obrázcích níže lze porovnat přístup CSSOM a TOM. Oba přístupy se používají na změnu průhlednosti elementu třídy „box". První obrázek ukazuje přístup k vlastnostem pomocí metodiky TOM.

```
function opacityOn(){
    let box = document.querySelector( selectors: ".box");
    box.attributeStyleMap.set('opacity', CSS.number(.5));
}
```

Obrázek 5: Ukázka práce pomocí StylePropertyMap s parametry CSS.

Naopak druhý obrázek znázorňuje přístup pomocí Element CSS inline Style. Z obrázku je patrné, jak tento přístup funguje. Jako první musíme pomocí qeurrySelectoru vybrat konkrétní elementy. V dalším kroku už s nimi můžeme pracovat.

Na tomto obrázku můžeme vidět přístup zápisu pomocí inline zápisu.

```
function opacityOut(){
    let box = document.querySelector( selectors: ".box");
    box.style.opacity = "1";
}
```

Obrázek 6: Ukázka práce s CSS parametry pomocí takzvaného „inline" zápisu.

Zde jsou vypsány všechny třídy, které lze v rámci TOM využít. Společně s třídami TOM jsou zde popsány jejich metody a rozhraní konkrétních tříd, jejichž funkce a syntaxe jsou vzájemně propojeny. To znamená, že rozhraní CSSMathMax můžeme najít hned v několika třídách:

1) CSSKeywordValueSerialization

Třída CSSKeywordValueSerialization disponuje těmito metodami:

- CSSKeywordValue()
  - Jedná se o konstruktor, kterým vytvoříme instanci třídy CSSKeywordValue().
  - Je důležité, aby vstupní argument byl textový řetězec.
- value()
  - Pomocí metody value() získáváme hodnotu uloženou v objektu, který jsme vytvořili pomocí konstruktoru CSSKeywordValue().

Příklad užití CSSKeywordValueSerialization, kde prostřednictvím konstruktoru vytvoříme objekt, který nese informaci o délce ohraničujícího elementu a z toho dopočítává délku vnitřního elementu, jenž funguje jako „progressbar“. Díky tomuto postupu jsme zaručili responzivní chování elementů. Je zjevné, že tato třída slouží pouze pro čtení hodnot, případně jejich transformaci a přípravu pro budoucí práci s javascriptem, nikoli k manipulaci s daty.

```
function calcWidth(){
    const number = document.querySelector( selectors: '#number');
    const fill = document.querySelector( selectors: '.fill');
    const bar = document.querySelector( selectors: '.bar');

    let value  = new CSSUnparsedValue( ['30'] );

    bar.attributeStyleMap.set('width', CSS.vw(value[0]));
    const barWidth = bar.computedStyleMap().get('width')
    const keyword = new CSSKeywordValue(barWidth.value).value;

    document.getElementById( elementId: "percents").innerHTML = number.value+"%";
    number.value > 100 ? number.value=100 : number.value
    fill.attributeStyleMap.set('width', CSS.px(((number.value*keyword)*0.99)/100));
}
calcWidth();
```

Obrázek 7: Ukázka práce CSSKeywordValueSerialization.

2) StylePropertyMap

Pomocí StylePropertyMap jsme schopni provádět změnu atributů CSS pomocí metod, které jsou pro tyto účely definovány. Tyto metody jsou:

- set()
  - Metodu set používáme, pokud chceme danému elementu nebo třídě nastavit libovolnou vlastnost.
  - Na příkladu si ukážeme, jak lze pomocí uživatelské interakce měnit element pomocí CSS Houdini.

```
function calcWidth(){
    const number = document.querySelector( selectors: '#number');
    const fill = document.querySelector( selectors: '.fill');
    const bar = document.querySelector( selectors: '.bar');

    let value  = new CSSUnparsedValue( ['30'] );

    bar.attributeStyleMap.set('width', CSS.vw(value[0]));
    const barWidth = bar.computedStyleMap().get('width')
    const keyword = new CSSKeywordValue(barWidth.value).value;

    document.getElementById( elementId: "percents").innerHTML = number.value+"%";
    number.value > 100 ? number.value=100 : number.value
    fill.attributeStyleMap.set('width', CSS.px(((number.value*keyword)*0.99)/100));
```

Obrázek 8: Ukázka dopočítávání délky elementu na základě rodičovského elementu pomocí CSS Houdini.

  - Uživatel může do textového pole zadávat číselné hodnoty. Na základě hodnot se vypočítá velikost „loading baru“. Vše funguje na principu přístupu TOM a metody set(). Zároveň zde využíváme metodu get(), se kterou získáváme aktuální velikost ohraničení „loading baru“. Tuto metodu výpočtu volíme kvůli responzivitě.
  - V příkladu jsme zároveň použili jednu z dalších možností CSS Houdini. Jedná se o CSSKeywordValue.
  - Druhý příklad ukazuje, jak lze přistupovat k CSS atributu „color“ a jak ho posléze změnit využitím přístupu TOM. Podobný příklad jsme aplikovali v rámci CSSOM. Lze tedy porovnat oba přístupy.
  - Ukázka kódu:

```
//TOM
function TOMChangeTextColor(){
    let tom = document.querySelector( selectors: '#tom');
    tom.attributeStyleMap.set('color', 'red');
    console.log(tom.attributeStyleMap.get('color'))
}
TOMChangeTextColor();
```

Obrázek 9: Ukázka změny barvy textu pomocí CSS Houdini.

- Výstup v prohlížeči:

Changing property by CSSOM.
Color of text changed by CSSOM.

Changing property by TOM.
Color of text changed by TOM.

Obrázek 10: Ukázka výstupů změny barvy textu pomocí přístupů CSSOM a TOM.

- Na obrázku výše můžeme vidět výstup obou zmiňovaných přístupů.

- delete()
  - Metoda delete() slouží pro odstranění konkrétního atributu v elementu, který se udává jako argument v metodě delete(). Atribut se zadává jako textový řetězec.
  - Kód, který vidíme níže, provádí změnu CSS atributu color na základě změny proměnné typu boolean, která se mění po kliknutí tlačítka.
  - Na obrázku níže jsme se pokusili demonstrovat užití metody delete() a set().

```
function changeStyle(){
    let isActive = true
    button1=document.querySelector( selectors: "#example");
    button1.addEventListener( type: 'click',  listener: ()=>{
        isActive = !isActive;
        isActive ?  button1.attributeStyleMap.set('color', 'white') : button1.attributeStyleMap.delete('color');
    })
}
changeStyle()
```

Obrázek 11: Demonstrace metod delete() a append().

- clear()
  - Metoda clear() zbavuje element veškerého stylování.
- append()
  - Metoda append() připojuje k elementu zadaný atribut včetně hodnoty. Atribut i hodnotu zadáváme jako argumenty v podobě textového řetězce, append(atribut, hodnota).

3) CSSUnparsedValue

Na stránce MDN Web Docs tuto třídu popisují takto:

„Rozhraní CSSUnparsedValue představuje hodnoty vlastností, které odkazují na Custom properties. Skládá se ze seznamu fragmentů řetězců a odkazů na proměnné.“[3]

„Custom properties jsou reprezentovány CSSUnparsedValue a odkazy var() jsou reprezentovány pomocí CSSVariableReferenceValue.“[3]

Tato třída disponuje těmito metodami:

- entries()
- forEach()
  - Tato metoda se neliší od standardního cyklu forEach, kterým disponuje javascript.
- keys()
- CSSUnparsedValue()
  - Tímto konstruktorem vytváříme objekt, jehož argumenty jsou atributy instance třídy, kterou vytvoříme. Pro demonstraci tento konstruktor aplikujeme do příkladu kalkulace šířky na základě uživatelského vstupu.

```
function calcWidth(){
    const number = document.querySelector( selectors: '#number');
    const fill = document.querySelector( selectors: '.fill');
    const bar = document.querySelector( selectors: '.bar');

    let value  = new CSSUnparsedValue( ['30'] );

    bar.attributeStyleMap.set('width', CSS.vw(value[0]));
    const barWidth = bar.computedStyleMap().get('width')
    const keyword = new CSSKeywordValue(barWidth.value).value;

    document.getElementById( elementId: "percents").innerHTML = number.value+"%";
    number.value > 100 ? number.value=100 : number.value
    fill.attributeStyleMap.set('width', CSS.px(((number.value*keyword)*0.99)/100));
}
```

Obrázek 12: Ukázka konstruktoru CSSUnparsedValue, který používáme v ukázce kalkulace šířky elementu.

4) CSSStyleValue

CSSStyleValue je základní třídou TOM. Instance této třídy se dají používat téměř kdekoli, kde se očekává textový řetězec. Tato třída obsahuje tyto metody:

- parse()
    - Z hodnot, které jsou poskytnuty jako argument u některého z rozhraní, tato metoda vytvoří objekt, který nese právě informace poskytnuté argumenty.
    - Ukázka zápisu:

```
function CSSStyleValue(){
    const transform = CSSTransformValue.parse(
        s: 'transform', 'translate3d(10px,10px,10px) scale(0.1)');
        console.log(transform)

}
CSSStyleValue();
```

Obrázek 13: Ukázka metody parse() u třídy CSSTransformValue.

- Výstup z konzole:

```
▼CSSTransformValue                                              typedOM.js:137
  ▼0: CSSTranslate
     is2D: false
    ▼x: CSSUnitValue
       unit: "px"
       value: 10
      ▶__proto__: CSSUnitValue
    ▼y: CSSUnitValue
       unit: "px"
       value: 10
      ▶__proto__: CSSUnitValue
    ▼z: CSSUnitValue
       unit: "px"
       value: 10
      ▶__proto__: CSSUnitValue
    ▶__proto__: CSSTranslate
  ▼1: CSSScale
     is2D: true
    ▼x: CSSUnitValue
       unit: "number"
       value: 0.1
      ▶__proto__: CSSUnitValue
    ▼y: CSSUnitValue
       unit: "number"
       value: 0.1
      ▶__proto__: CSSUnitValue
    ▼z: CSSUnitValue
       unit: "number"
       value: 1
      ▶__proto__: CSSUnitValue
    ▶__proto__: CSSScale
   is2D: false
   length: 2
  ▶__proto__: CSSTransformValue
```

Obrázek 14: Ukázka výpisu konzole pro metodu parse() použitou u třídy CSSTransformValue.

- parseAll()
    - Tato metoda nastaví všechny výskyty konkrétní vlastnosti CSS na zadanou hodnotu. Jako výstup se očekává pole objektů, z nichž každý obsahuje jednu z dodaných hodnot.
    - Ukázka kódu:

```
const transformAll = CSSTransformValue.parseAll(
    'transform', 'translate3d(10px,10px,10px) translate3d(5px,5px,5px) scale(0.1) scale(0.5)');
console.log(transformAll)
```

Obrázek 15: Ukázka metody parseAll u třídy CSSTransformValue.

- Výstup z konzole:

```
▼Array(1)                                                    typedOM.js:142
 ▼0: CSSTransformValue
  ▼0: CSSTranslate
     is2D: false
    ▼x: CSSUnitValue
       unit: "px"
       value: 10
      ▶__proto__: CSSUnitValue
    ▶y: CSSUnitValue {value: 10, unit: "px"}
    ▶z: CSSUnitValue {value: 10, unit: "px"}
    ▶__proto__: CSSTranslate
  ▼1: CSSTranslate
     is2D: false
    ▼x: CSSUnitValue
       unit: "px"
       value: 5
      ▶__proto__: CSSUnitValue
    ▶y: CSSUnitValue {value: 5, unit: "px"}
    ▶z: CSSUnitValue {value: 5, unit: "px"}
    ▶__proto__: CSSTranslate
  ▼2: CSSScale
     is2D: true
    ▼x: CSSUnitValue
       unit: "number"
       value: 0.1
      ▶__proto__: CSSUnitValue
    ▶y: CSSUnitValue {value: 0.1, unit: "number"}
    ▶z: CSSUnitValue {value: 1, unit: "number"}
    ▶__proto__: CSSScale
  ▼3: CSSScale
     is2D: true
    ▼x: CSSUnitValue
       unit: "number"
       value: 0.5
      ▶__proto__: CSSUnitValue
    ▶y: CSSUnitValue {value: 0.5, unit: "number"}
    ▶z: CSSUnitValue {value: 1, unit: "number"}
    ▶__proto__: CSSScale
   is2D: false
   length: 4
  ▶__proto__: CSSTransformValue
  length: 1
 ▶__proto__: Array(0)
```

Obrázek 16: Ukázka výpisu konzole pro metodu parseAll() pro třídu CSSTransformValue.

CSSStyleValue disponuje také několika rozhraními:

a) CSSNumericValue

- Rozhraní, které nám dává možnost procesování numerických operací.
- Metody:
    - parse()
        - Metoda převádí řetězec hodnot na objekt.
    - add()
        - Metoda add() slouží pro přičítání hodnot.
    - sub()
        - Metoda sub() je používána pro odečítání hodnot.
    - mul()
        - Metoda, kterou lze násobit členy mezi sebou.
    - div()
        - Metoda, která je určena pro dělení.
    - max()
        - Metoda max() navrací maximální hodnotu. Podmínkou je, aby hodnoty byly stejného typu. Je tedy vhodné, někdy i žádoucí, použít metodu to() pro převedení jednotek na požadovaný tvar.
    - min()
        - Metoda min() funguje na stejném principu jako metoda max() s rozdílem, že navrací minimální hodnotu.
    - equals()
        - Pomocí metody equals() porovnáváme 2 hodnoty, které do metody vstupují jako argumenty. Tyto hodnoty jsou porovnávány a metoda vrací hodnotu true typu boolean, pokud jsou hodnoty shodné, false, pokud jsou neshodné. V tomto případě se jedná o striktní porovnání, které v javascriptu popisujeme takto: „===“.
    - to()
        - Metoda to() převádí zadané jednotky na jednotky, které určíme.
    - toSum()

        - Metodou toSum() můžeme sečíst všechny dodané hodnoty a převést je na požadovanou jednotku.
    - type()
        - Metoda type() vrací objekt, který nese informace o vlastnostech dané proměnné nebo jiného nositele informace o hodnotách typu CSSNumericValue. To znamená, že pokud budeme mít proměnnou nesoucí záznamy o délce v hodnotách pixelů, výstupem bude o jaký typ hodnot se jedná (například length, time, angel, ...).

- Vzhledem k počtu metod, kterými třída disponuje, si ukážeme pouze některé příklady, které fungují na podobném principu.

```
console.log(CSS.px("10").to("cm").toString());

let sum = CSS.cm("23").add(CSS.in("4")).add(CSS.px("3")).add(CSS.percent("9"));
console.log(sum.toSum("px", "percent").toString())
```

Obrázek 17: Kód ukazující metody pro sčítání hodnot a převádění jednotek v rámci třídy

- Na ukázce jsme demonstrovali, jak fungují metody toSum() a to(). Metoda toSum() sečte všechny prvky na požadovaný formát. Druhý vstupní argument říká, jakou hodnotu budeme k sumě přičítat. Druhá metoda to() pouze převádí jednotky mezi sebou. Ukázka:

```
function CSSNumericValue(){
    let mathSum = CSS.px("23").add(CSS.em("3")).add(CSS.percent("4")).mul(CSS.cm("9")).div(CSS.rem("4"));
    console.log(mathSum.toString());
    console.log(mathSum);
}
CSSNumericValue()
```

Obrázek 18: Ukázka kódu, ukazující pokročilejší metody třídy CSSNumericValues.

- Na následujícím obrázku je vidět výstup v konzoli z ukázek, které jsme viděli na předchozím obrázku.

```
0.264583cm                                              typedOM.js:179
calc(1256.29px + 9%)                                    typedOM.js:182
```

Obrázek 19: Výstup do konzole z ukázek na 19. obrázku.

- Na ukázce kódu výše (obrázek číslo 19) jsou znázorněny metody add(), mul() a div(). Tyto metody provádějí přičítání hodnoty, násobení a dělení. Po provedení operace

metoda vrátí hodnoty v požadovaném tvaru. Níže vidíme výpis z konzole. V prvním případě se vrátí textový řetězec, který lze použít jako hodnotu pro atribut CSS.

```
calc((23px + 3em + 4%) * 9cm / 4rem)                                typedOM.js:185
```

Obrázek 21: Výpis do konzole s metodou toString() z 18. obrázku.

Ve druhém výpisu je znázorněn vnořený objekt, který ale rozděluje konkrétní výstupy metod do 3 sekcí podle toho, k jakému rozhraní patří. Viz obrázky níže.

```
▼CSSMathProduct                                                     typedOM.js:186
   operator: "product"
  ▼values: CSSNumericArray
    ▼0: CSSMathSum
       operator: "sum"
      ▼values: CSSNumericArray
        ▼0: CSSUnitValue
           unit: "px"
           value: 23
          ▶__proto__: CSSUnitValue
        ▼1: CSSUnitValue
           unit: "em"
           value: 3
          ▶__proto__: CSSUnitValue
        ▼2: CSSUnitValue
           unit: "percent"
           value: 4
          ▶__proto__: CSSUnitValue
         length: 3
        ▶__proto__: CSSNumericArray
      ▶__proto__: CSSMathSum
    ▼1: CSSUnitValue
       unit: "cm"
       value: 9
      ▶__proto__: CSSUnitValue
    ▼2: CSSMathInvert
       operator: "invert"
      ▼value: CSSUnitValue
         unit: "rem"
         value: 4
        ▶__proto__: CSSUnitValue
      ▶__proto__: CSSMathInvert
     length: 3
    ▶__proto__: CSSNumericArray
  ▶__proto__: CSSMathProduct
```

Obrázek 20: Výpis do konzole z obrázku číslo 18.

b) CSSImageValue

Rozhraní CSSImageValue představuje hodnoty pro vlastnosti, které reprezentují obrázek. Příkladem může být obrázek na pozadí elementu. Tato třída může nabývat pouze CSS hodnot url() nebo image(). Třída CSSImageValue nedisponuje žádnými specifickými metodami, proto zde nejsou uvedeny ukázky, které by třídu demonstrovaly. Třída pouze využívá metody typu get() a set(). Tyto metody jsme již dříve používali. Třída je nezbytná pro práci s obrázky. Pokud bychom je nechali vypsat pomocí třídy get() u CSS atributu „background-image', dostali bychom objekt, ve kterém by byl popsán obrázek a zároveň třída, pod kterou tento atribut spadá.

c) CSSMathValue

Tato třída nepřidává žádné nové metody, pouze je dědí z nadřazených tříd. Třída CSSMathValue jako taková má sama o sobě jeden společný atribut. Atribut operator nese informaci o tom, který matematický operátor v dané třídě je použit. Když se podíváme na 22. obrázek, všimneme si atributu operator. Tento atribut je pouze u instancí tříd z rodiny CSSMath, v tomto případě CSSMathProduct a CSSMathSum. Také si všimněme, že atribut operator nabývá hodnoty podle toho, z jaké třídy pochází.

- CSSMathValue je třída zahrnující práci s rozhraními určenými pro matematické operace. Tato rozhraní si níže popíšeme.
    - CSSMathProduct se aplikuje při kombinaci několika metod pro matematické operace. Protože jsme použili několik metod společně, v příkladu na obrázku 20 můžeme vidět, že operator je „product", protože se o tyto jevy stará rozhraní CSSMathProduct.
    - CSSMathSum se používá pro sčítání několika hodnot. Protože jsme použili metodu sum(), v příkladu na obrázku 22 můžeme vidět, že tato metoda byla použita právě v rámci tohoto rozhraní a operator je „sum".
    - CSSMathInvert by mělo reprezentovat CSS funkci calc(). Tato funkce je použita spolu se vstupním argumentem, který je použit ve funkci calc() tímto způsobem: calc(1 /argument).
    - CSSMathMax reprezentuje matematickou funkci max().
    - CSSMathMin reprezentuje matematickou funkci min().
    - CSSMathNegate by mělo ze vstupního argumentu, který vkládáme do konstruktoru, vytvořit negovanou hodnotu onoho argumentu.

- Třída CSSMathMax a CSSMathMin fungují na stejném principu jako CSSMathSum a CSSMathProduct. To znamená, že reprezentují jednu konkrétní funkci pro klasické početní operace.

d) CSSPositionValue

Rozhraní slouží pro získávání souřadnic elementu. Jako ukázku zvolíme příklad z webové stránky develloper.moziila.org [11].

Na stránce vývojáři demonstrují umístění obrázku pomocí rozhraní CSSPositionValue.

```
let replacedEl = document.getElementById( 'image' );
let position = new CSSPositionValue( CSS.px(35), CSS.px(40) );

replacedEl.attributeStyleMap.set( 'object-position', position );
console.log( position.x.value, position.y.value );
console.log( replacedEl.computedStyleMap().get('object-position') );
```

Obrázek 22: Ukázka použití CSSPositionValue z webové stránky https://developer.mozilla.org.

Prostřednictvím tohoto rozhraní můžeme dobře ovlivňovat umístění elementu na základě souřadnic druhého elementu, a to tak, že díky computedStyleMap.get() zjistíme souřadnice elementu, na základě kterého budeme určovat pozici druhého elementu. Druhému elementu poté přiřadíme ony souřadnice s požadovaným rozdílem souřadnic prvního elementu.

e) CSSUnitValue

CSSUnitValue slouží pro jasné deklarování jednotek. Rozdíl oproti CSSNumericValue je hlavně v syntaxi a datové reprezentaci.

f) CSSTransformValue

Toto rozhraní nám umožňuje kombinovat několik transformací najednou. Demonstraci použití si uvedeme na příkladu.

1. Deklarujeme funkci, kterou budeme spouštět.
2. V dané funkci vytvoříme proměnné, které budou držet hodnoty transformací.
3. Pomocí selektoru deklarujeme proměnné, které budou obsahovat referenci na elementy v html souboru.
4. Ke všem html elementům, které chceme použít k interakci, definujeme event listnery.

5. Do těchto listnerů vložíme funkci, která bude provádět samotné transformace. Jako argumenty funkce použijeme proměnné určené k uchování hodnot pro transformace. Tyto proměnné se v rámci listneru budou měnit dle potřeby. Pro rotaci vpravo budeme po každém kliknutí přidávat jedničku k definované proměnné.
6. Vytvoříme funkci pro transformace. V našem případě je funkce pro změnu měřítka a rotace. Jako vstupní argumenty máme první pro rotaci a druhý pro změnu měřítka, jeden po ose X a druhý po ose Y. Ve funkci deklarujeme proměnnou transform, která je instance třídy CSSTransformValue. Tato třída má jako vstupní argument pole s transformacemi, které chceme aplikovat. My jsme zvolili CSSRotate, do které jako argument vkládáme objekt s metodou deg(). Tato metoda uchovává proměnnou, která drží údaje o rotaci. Metoda deg() definuje, že se jedná o stupně. Podobně pracujeme i s třídou pro změnu měřítka CSSScale, jenom s tím rozdílem, že tato metoda přijímá dva argumenty. Jeden pro osu X a druhý pro osu Y. Tyto argumenty obalíme do metody CSS.number(), abychom upřesnili, o jakou jednotku se jedná.
7. Výsledkem je, že můžeme pomocí tlačítek měnit měřítko a rotaci objektu. Tyto transformace lze aplikovat najednou.

```
//TOM transform
function rotateTOM(){
    let rotateNum = 0;
    let leftScale = 1;
    let rightScale = 1;
    rotatePlus=document.querySelector( selectors: "#rotatePlus");
    rotateMinus=document.querySelector( selectors: "#rotateMinus");
    scalePlusR=document.querySelector( selectors: "#scalePlusR");
    scaleMinusR=document.querySelector( selectors: "#scaleMinusR");
    scalePlusL=document.querySelector( selectors: "#scalePlusL");
    scaleMinusL=document.querySelector( selectors: "#scaleMinusL");


    rotatePlus.addEventListener( type: 'click',  listener: ()=>{
        doTransorm( rotate: rotateNum+=1, leftScale, rightScale)
    })
    rotateMinus.addEventListener( type: 'click',  listener: ()=>{
        doTransorm( rotate: rotateNum-=1, leftScale, rightScale)
    })

    scaleMinusL.addEventListener( type: 'click',  listener: ()=>{
        doTransorm(rotateNum,  scaleAdd: leftScale-=0.1, rightScale)
    })
    scalePlusL.addEventListener( type: 'click',  listener: ()=>{
        doTransorm(rotateNum,  scaleAdd: leftScale+=0.1, rightScale)
    })
    scaleMinusR.addEventListener( type: 'click',  listener: ()=>{
        doTransorm(rotateNum, leftScale,  scaleRemove: rightScale-=0.1)
    })
    scalePlusR.addEventListener( type: 'click',  listener: ()=>{
        doTransorm(rotateNum, leftScale,  scaleRemove: rightScale+=0.1)
    })

    function doTransorm(rotate, scaleAdd, scaleRemove){
    const transform = new CSSTransformValue([
        new CSSRotate(CSS.deg(rotate)),
        new CSSScale(CSS.number(scaleAdd), CSS.number(scaleRemove))
      ])
      box.attributeStyleMap.set('transform', transform);
    }
}
rotateTOM();
```

Obrázek 23: Ukázka práce uživatelské interakce pomocí CSS Houdini.

K ukázce na obrázku číslo 21 jsme také vypracovali výkonnostní porovnání Typed Object Modelu a CCS Object Modelu. Tento příklad jsme vybrali, protože byl ze všech procesně nejnáročnější a zároveň na něm bylo možné dobře porovnat oba přístupy. Výsledek srovnávacího testu výkonu byl jasně prokazatelný. Na obrázcích je viditelné, že Typed Obejct Model dokáže prokazatelně rychleji provádět i tak nenáročné operace, jako jsme ukázali na příkladu. Rozdíl se může zdát zanedbatelný, ale je potřeba brát v potaz, že se stále jedná pouze o nízkoúrovňové API. Jde pouze o jakýsi druh podpory pro vysokoúrovňová API, u kterých by poté výsledný rozdíl byl více znatelný.

```
0.09000000136438757                                    typedOM.js:129
```

Obrázek 24: Výsledek výkonnostního testu mezi Typed Object Model a CSSObject Model – Typed Object Model.

```
0.2749999985098839                                     typedOM.js:176
```

Obrázek 25: Výsledek výkonnostního testu mezi Typed Object Model a CSSObject Model – CSS Object Model.

e) Custom Properties and Values API

Custum Property and Values API umožňuje vytvářet CSS proměnné, které mohou uchovávat konkrétní, často opakovaně použitou hodnotu napříč celým dokumentem. Tento druh API umožňuje vývojářům rozšířit proměnné v CSS o typ, dědičnost a počáteční hodnotu. Aby bylo možné provést takové rozšíření, musí se použít metoda registerProperty(). Metoda říká prohlížeči, jak pracovat s přechody a jak se případně chovat během výskytu erroru. Ke konkrétním proměnným pak přistupujeme pomocí funkce var(). Jediný vstupní argument této funkce je právě požadovaná proměnná.

Již zmíněná metoda registerProperty() přijímá argumenty v podobě objektu. Tento objekt obsahuje několik povinných atributů. Jedním z nich je atribut syntax, určující, o jaký typ proměnné se bude jednat. Atribut syntax tak může nabývat všech hodnot, které CSS podporuje. Jedná se například o:

- color,
- percentage,
- number,
- lenght,

- url,
- string.

Na obrázcích můžeme vidět dva způsoby deklarace proměnné. První způsob je deklarace v javascriptovém souboru. Druhý způsob je vytvoření proměnné přímo v CSS souboru.

```
CSS.registerProperty({
    name:'--rotater',
    syntax: '<angle>',
    initialValue: '0deg',
    inherits: false
});
```

```
@property --color-a {
    syntax: '<color>';
    inherits: false;
    initial-value: red;
}
```

Obrázek 26: Ukázka deklarace CSS proměnné.

Popišme si jednotlivé atributy.

- name
    - Atribut specifikuje jméno proměnné, pod kterým budeme proměnnou provolávat. Všimněme si, že proměnná vždy začíná dvojitým symbolem „-“.
- syntax
    - Atribut definuje, o jaký datový typ se jedná. Podle toho bude prohlížeč k proměnné přistupovat. Pokud bychom například vkládali proměnou se syntaxí angel, která může nabývat hodnot pro rotaci (stupně, otáčky, radiány). Do CSS atributu width můžeme očekávat error, ale to pouze v případě, že tuto operaci budeme provádět na straně javascriptu. Pokud bychom na straně CSS vložili proměnou do CSS atributu, kde proměnná nemá požadovaný datový typ, žádnou změnu bychom neregistrovali.
- initialValue
    - Je atribut, který specifikuje, jaké hodnoty proměnná nabývá od svého deklarování, dokud nebude přepsána.
- inherits
    - Jedná se o atribut typu boolean. V případě, že je true, proměnná bude dědit hodnotu od rodičovského elementu. V momentě, kdy bude nabývat hodnoty false, se dědičnost neprovede.

Po deklarování proměnných máme možnost je používat, a to jak v javascriptovém souboru, tak v CSS souboru. Ukažme si příklad, ve kterém pomocí těchto proměnných můžeme měnit gradientní pozadí elementů, které při přejetí myší mění hodnotu definované proměnné.

```
.circle1,.circle2{
    --color-hover: rgba(255,255,255,1);
    background: linear-gradient(to right bottom, var( --color-hover), rgba(255,255,255,0.5));
    transition:  --color-hover 0.5s;
    border-radius: 10rem;
    height: 15rem;
    width: 15rem;
    position: absolute;
    z-index: 3;
}

.circle1{
    top: 5%;
    left: 10%;
}

.circle1:hover{
    --color-hover: skyblue;
}

.circle2{
    bottom: 5%;
    right: 7%;
}

.circle2:hover{
    --color-hover: deeppink;
}
```

Obrázek 27: Ukázka použití CSS proměnné v gradientních přechodech.

Na obrázku je viditelná část CSS kódu, v níž definujeme vlastnosti dvěma CSS třídám. Obě třídy jsou při normálním stavu totožné. Obě mají definovanou proměnnou --color-hover, jejíž hodnoty voláme metodou var() jako součást gradientního pozadí elementů. Všimněme si, že třídy se od sebe navzájem liší chováním ve chvíli, kdy se na ně najede myší. Každá třída má jinak definovanou proměnnou --color-hover. Právě tady se děje to, co bychom bez těchto proměnných nemohli dříve realizovat – změna hodnot v gradientních přechodech v rámci pseudotříd.

Je dobré si pamatovat, že konkrétní proměnná je platná pouze v rozsahu, v jakém je definována. To znamená, že pokud proměnnou definujeme pouze pro třídu circle1, bude platná pouze v třídě circle1 a ve všech potomcích třídy circle1. V případě, že bychom chtěli proměnnou definovat pro celý dokument, je poté zapotřebí ji definovat do kořenové

pseudotřídy :root. Dědičnost v Custom Properties and Values API je závislá na tom, zda jsme nastavili v definici proměnné atribut inherits na true. Pokud jsme atribut nastavili na false, všechny „child“ elementy požívající proměnnou budou z proměnné dostávat hodnotu, která je definována jako initialValue, bez ohledu na skutečnost, jakou hodnotu té stejné proměnné má rodičovský element.

Nyní bychom se mohli podívat, jak si toto API umí poradit s chybami, které mohou v rámci dynamiky systému vznikat. Předpokládejme, že máme aplikaci, která je napsána tak, aby měla dynamické stylování v závislosti na okolních vjemech (noční režim, interakce uživatele, …). V tomto scénáři se může stát chyba ze strany vývojáře a do proměnné, která by měla nabývat konkrétních hodnot (například barev), by se dostala hodnota v pixelech. Logicky tato hodnota je pro nastavení barvy neplatná, proto bychom nezaznamenali žádnou změnu. Pro tyto případy je zde záložní řešení, které se nazývá fallback values. Jak už název napovídá, jde o záložní hodnotu. V úvodu kapitoly jsme tvrdili, že metoda var() má pouze jeden vstupní argument. To ovšem nebyla tak docela pravda. Právě pro tento případ lze do metody var() dát i druhý argument pro případ, že první argument nebude validní. Takže zápis by potom mohl vypadat následovně:

```
background: linear-gradient(to right bottom, var( --color-hover, rgba(255,255,255,1)), rgba(255,255,255,0.5));
```

Obrázek 28: Ukázka aplikace fallback hodnoty.

Tento přístup je možné aplikovat v několika vrstvách. Potom bychom mohli mít fallback hodnotu v rámci další fallback hodnoty.

Je také vhodné zmínit, jak funguje spolupráce javascriptu a Custom Properties and values.

Pomocí javascriptu lze elementy stylovat oběma způsoby, jak pomocí takzvaného inline zápisu, tak i prostřednictvím dalšího API z rodiny CSS Houdini, Typed Object Model API.

Na obrázku níže máme ukázku nastavování vlastností proměnné použitím javascriptu:

```
function rotateInline(){
    box.style.setProperty( property: '--rotate',  value: '15deg');
}
function rotateCPV(){
    box.attributeStyleMap.set('--rotater', '15deg');
    console.log(box.computedStyleMap().get('--rotater'))
}
```

Obrázek 29: Ukázka změny hodnoty proměnné pomocí javascriptu.

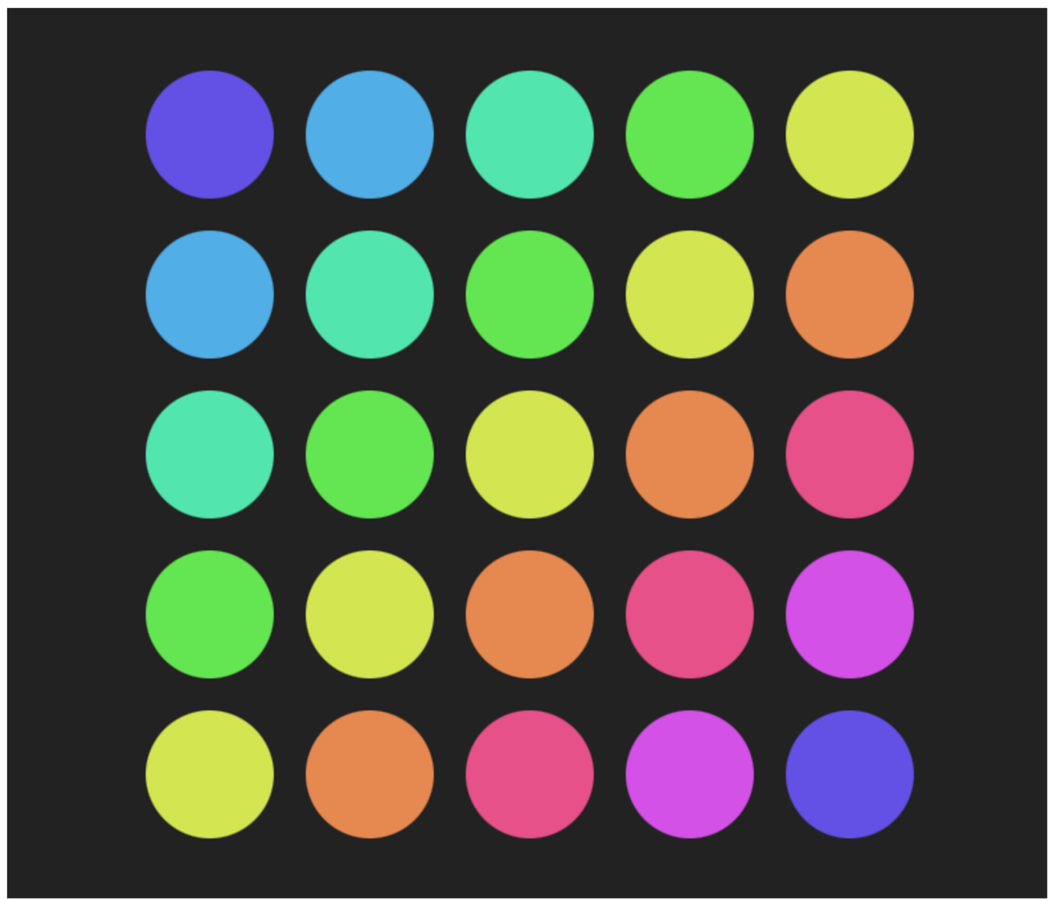
Zde máme výstup z konzole:

```
▼CSSUnitValue ℹ                                          typedOM.js:76
   unit: "deg"
   value: 15
 ▶__proto__: CSSUnitValue
```

Obrázek 30: Výstup z konzole z kódu na obrázku číslo 29.

Abychom viděli, jaké nevšední a impozantní efekty se dají vytvářet pouze za kooperace javascriptu a CSS proměnných poskytnutých API, odkazuji na výtvory dvou vývojářů, kteří výsledek své práce prezentují na stránce codepen.io.

Prvním příkladem je ukázka šachovnice skládající se z několika kruhových objektů, které mají gradientní přechod v rozmezí duhových barev. Autorka Ana Tudor svůj výtvor pojmenovala „moon-grid with clip-path“[15].



Obrázek 31: Ukázka práce Any Tudor s názvem "moon-grid with clip-path".

V případě druhého příkladu odkazujeme přímo na webovou stránku, kde si můžete efekt prohlédnout. Jde o animaci, která mění hodnoty elementů. Konkrétně se jedná o mřížku složenou z několika hexagonů, jež v pravidelném intervalu mění svůj tvar. Autorem je Dan Wilson. Svůj výtvor pojmenoval „Dance of the Tiled Houdini Hexagons“. Na tomto odkazu je

možné si jeho práci prohlédnout [19]. Tento konkrétní příklad je založen na odrážkovém seznamu, který je upravený tak, aby vytvořil několik šestiúhelníků měnících během animace barvu a velikost.

Chris Coyier ve svém článku na téma Custom Propertis and State zmiňuje myšlenku přímého vytváření CSS proměnných pomocí Reactu a Vue. Jeho idea si pohrává s myšlenkou vytvořit metodu pro práci se stavem. Konkrétně zmiňuje React hook se vstupním argumentem, který by byl zpracován jako CSS proměnná. Tímto způsobem by se mohl snadno měnit přímý přístup k CSS atributům skrze změny stavů v javascriptových knihovnách a rámcích [4].

f) Paint API

Jak již bylo zmíněno v úvodu, CSS Houdini a jeho API je stále ve vývoji a není plně podporován. Aby se předešlo situacím, kdy by se na starší verzi prohlížeče nic nevykreslilo, do projektu se musí naimportovat takzvaný polyfill. Polyfill je balíček starající se o podporu nových rozšiřujících technologií v prohlížeči. Bez polyfillů bychom nebyli schopni využívat funkcí, které přináší Paint API. K naimportování máme 2 možné způsoby.

1. Vložení odkazu na balíček vhtml tagu <script> -
   <script src="https://unpkg.com/CSS-paint-polyfill"></script>
2. Importování pomocí javascriptu. Tuto metodu bychom využili například v knihovně react - import „https://unpkg.com/CSS-paint-polyfill“;

Paint API dovoluje vývojářům pomocí javascriptu vykreslovat přímo pozadí elementu, ohraničení nebo obsah pomocí 2D Rendering Context (podmnožina HTML Canvas API). K vykreslování se používá Paint worklet[1].

Po definování workletu se musí worklet provolat v HTML následujícím způsobem: *CSS.paintWorklet.addModule*(). Metoda addModule() disponuje jedním argumentem nesoucím umístění javascriptového souboru. V javascriptovém souboru máme uložený samotný vykreslovací proces.

[1] Na javascriptu nezávislé moduly, které se spouští během vykreslovaní. Spouštějí se pouze na stránkách využívajících HTTPS nebo na lokálním prostředí.

Ukažme si nyní, jak bychom správně měli definovat, registrovat a zavolat worklet, aby správně fungoval.

- Musíme registrovat Paint worklet pomocí registerPaint().
  - Atribut registerPaint se skládá z několika částí:
    - inputProperties() – sledované závislosti, potřebné k vykreslení,
    - inputArguments() – vstupní argumenty, které jsou předávány z CSS,
    - contextOptions() – může povolit nebo zakázat průhlednost(opacity) elementu, nabývá hodnot true/false,
    - paint() – primární vykreslovací funkce.

```
/*
Copyright 2016 Google, Inc. All Rights Reserved.
Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at
    http://www.apache.org/licenses/LICENSE-2.0
Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
*/

registerPaint( targetElement: 'ripple', class {
    static get inputProperties() { return ['background-color', '--ripple-color',
        '--animation-tick', '--ripple-x', '--ripple-y']; }
    paint(ctx, geom, properties) {
        const bgColor = properties.get('background-color').toString();
        const rippleColor = properties.get('--ripple-color').toString();
        const x = parseFloat(properties.get('--ripple-x').toString());
        const y = parseFloat(properties.get('--ripple-y').toString());
        let tick = parseFloat(properties.get('--animation-tick').toString());
        if(tick < 0)
            tick = 0;
        if(tick > 1000)
            tick = 1000;

        ctx.fillStyle = bgColor;
        ctx.fillRect( x: 0,  y: 0, geom.width, geom.height);
        ctx.fillRect( x: 0,  y: 0, geom.width, geom.height);

        ctx.fillStyle = rippleColor;
        ctx.globalAlpha = 1 - tick/1000;
        ctx.arc(
            x, y, // center
            radius: geom.width * tick/1000, // radius
            startAngle: 0, // startAngle
            endAngle: 2 * Math.PI //endAngle
        );
        ctx.fill();
    }
});
```

Obrázek 32: Vytvoření workeltu pro vykreslování.

Popišme si, co se v ukázce kódu děje:

- V samotném těle funkce paint() definujeme vykreslování. Popišme si jednotlivé vstupní argumenty:
    - ctx – vykreslovací kontext, je striktně definován jako podmnožina HTML 5 Canvas API,
    - geom – my jsme použili objekt geom, který v sobě nese informace o výšce a šířce elementu, ale můžeme zde použít jakýkoliv libovolný CSS atribut, se kterým bychom chtěli pracovat (size, color, …),
    - properties – pod argumentem properties můžeme volat a dále pracovat s argumenty, které jsme definovali jako inputProperties v rámci funkce registerPaint(),
    - args – argument args nese data registrovaná ve funkci registerPaint(), konkrétně se jedná o data, která jsme dostali zavoláním funkce inputArguments v těle funkce registerPaint().
- Přidáváme worklet do html dokumentu.
    - Worklet přidáme do html dokumentu pomocí zavolání metody CSS.addModule() v těle párového tagu <script> pro javascriptový kód.
      Metoda addModule() disponuje pouze jedním argumentem, kterým je cesta k souboru nesoucí worklet.

Nyní, po vytvoření a přidání workletu do html souboru, už pouze zbývá zavolat vykreslovací wokrlet v CSS. Ten zavoláme ke konkrétnímu CSS atributu. Například: *background-image: paint(imageWidth)*

Ukázali jsme si, jak můžeme napsat vlastní vykreslovací worklet. Tento způsob je velmi tvůrčí a umožňuje posouvat hranice toho, co lze vytvořit. Ovšem ne každý je schopen využít tohoto potenciálu naplno. Spíše by to pro někoho mohlo být přítěží. Proto je možnost importovat již vytvořené worklety z knihoven, které vytvořil někdo jiný a umožnil jejich využívání i ostatním vývojářům. Na webové stránce [8] můžeme vidět konkrétní ukázky těchto knihoven.

Importování workletu z html souboru:

```
<script src='https://unpkg.com/extra.css/sparkles.js'></script>
```

Obrázek 33: Importování workletu z externí knihovny.

Použití workletu v CSS souboru:

```css
@supports (background: paint(something)) {
    .primary {
        /*
          Optionally set property values.
          This can be done at root or local scope
        */
        --extra-sparkleNumber: 90; /* defaults to 30 */
        --extra-sparkleHue: 180; /* Defaults to 60 (yellow) */
        --extra-sparkleHeightVariance: 12; /* Defaults to 9 */
        --extra-sparkleWidthVariance: 15; /* Defaults to 12 */
        --extra-sparkleWeightVariance: 3; /* Defaults to 2 */


        background: paint(extra-sparkles);
    }
    .secondary{
        --extra-sparkleNumber: 50; /* defaults to 30 */
        --extra-sparkleHue: 40; /* Defaults to 60 (yellow) */
        --extra-sparkleHeightVariance: 9; /* Defaults to 9 */
        --extra-sparkleWeightVariance: 3; /* Defaults to 2 */
```

Obrázek 34: Volání metody paint() v CSS souboru.

```css
    }
}
```

Zde už máme konečný výstup:

The Paint API allows developers to use JavaScript functions to draw directly into an element's background, border, or content using 2D Rendering Context, which is a subset of the HTML5 Canvas API. Paint API uses Paint Worklet to draw an image that dynamically responds to changes in CSS (changes in CSS variables, for example). Anyone familiar with Canvas API will feel right at home with Houdini's Paint API.

On the background of the text you can se that animation API can do more then animate. Try to refresh page to see change.

Obrázek 35: Efekt, který byl vytvořen pomocí importování knihovny.

Příklad na obrázku 37 je pouze ukázkou toho, čeho můžeme pomocí Paint API dosáhnout. Je zde spousta zdrojů, kde jsou k vidění opravdu až dechberoucí efekty. V jednom z dostupných efektů autor použitím Paint API vytvořil QR kód generátor, který se mění po každém zadání dat do vstupního argumentu.

Dále za zmínku stojí také vytvoření takzvaného „ripple efektu". Jde o efekt aplikující se na element, který se spustí kliknutím. Po něm můžeme zaznamenat, jak se spustí řetězová reakce, která se vzdaluje od bodu a vytváří vlny **Chyba! Nenalezen zdroj odkazů.**[7]. Spoluautorem příkladu je vývojář známý jako Surma. Na jeho git uložišti můžeme vidět mnohem více z tématiky Paint API. Jsou zde k dispozici další příklady toho, jak může být Paint API využito a jak s ním lze posouvat hranice stylování.

My si ukážeme další dva příklady, které vytvořili vývojáři z GoogleChromeLabs. Prvním příkladem je několikabarevná hranice elementu. Tvorbu tohoto efektu si těžko umíme představit bez Paint API [6].

Obrázek 36: Několikabarevná hranice elementu.

Třetím efektem, který bychom si mohli ukázat je takzvaný „border radius reverse“. Jak název napovídá, jedná se o obrácené rohy hranice elementu [12].

Altere os valores do css abaixo :)

```
.radius-normal {
   border-radius: 10px;
}
.radius-reverse {
   background-color: transparent;
   --background-color: #ff6a00;
   background-image: paint(border-radius-reverse);
   --border-radius-reverse: 10;
}
```

Obrázek 37: Efekt obrácených rohů u hranice elementu.

g) Animation API

Druhou sadou z vysokoúrovňových API, které představíme, je animation API. Jak název prozrazuje, animation API bude mít za úkol obsluhovat animace. Podle [13] *„Animation Worklet umožňuje psát imperativní animace, které běží s nativní snímkovou frekvencí.“*

Animation API nelze brát jako zcela nový způsob vytváření animací, ale spíše jako rozšíření stávajících řešení. Nyní máme 3 možnosti provádění animací pomocí javascriptu. První možností je třída CSSTransition pro jednoduché přechody. Druhou možností je třída CSSAnimation, která umožňuje cyklit určitý typ děje, tím se vytváří animace. Třetí možností je asi nejkomplexnější Web Animation API pro pokročilejší animace.

Tyto tři možnosti mají společné, že jsou řízené časově a neuchovávají stav (state). Na tento limit narazíme při tvorbě sofistikovanějších animací, které jsou závislé na uživatelské interakci. Příkladem je třeba animace závislá na skrolování. Pomocí animačního workletu jsme schopni tento stav uchovávat a dále s ním pracovat. Tento přístup lze dobře využít například při vytváření takzvaných „parallax“ stránek, kdy využitím animačního workletu dosáhneme výkonnostního progresu. Animační worklet lze efektivně využít k parallax efektu, u kterého je

jinak obtížné dosáhnout plynulosti [13]. Příklad použití animation API pro realizaci parallaxového efektu naleznete v tomto zdroji [16].

Bohužel toto vysokoúrovňové API se nepovedlo zprovoznit. Konzole opět vypsala chybu u registrování workletu, který jsme registrovali podle instrukcí z dokumentace.

API umožňuje animacím využívat posluchače různých událostí (například: klik, „scroll“, …). Dále také zvyšuje výkon tím, že animace běží na svém vlastním přiřazeném vlákně používajícím animation worklet. Stejně jako v Paint API, i zde se musí worklet registrovat. Po registraci musí být přidán do hlavního javascriptového souboru.

1) Registrování workletu
    - Jak můžeme vidět na obrázku 39, registrování workletu probíhá trochu jinak než u Paint API. U Paint API jsme do těla metody pro registraci psali celý vykreslovací kód. Zde ve workletu definujeme jméno animace, konstruktor a metodu animate().

```
registerAnimator(
    "gaussian",
    class {
        constructor(options) {
            this.duration = options.duration;
            this.distance = options.distance;
        }

        animate(currentTime, effect) {
            const sigma = this.duration / 10;
            const mi = this.duration / 2;

            const x = currentTime % this.duration;

            const e = -0.5 * Math.pow( x: (x - mi) / sigma,  y: 2);
            const f = 1 / (sigma * Math.sqrt( x: 2 * Math.PI));

            const localTime = this.distance * this.duration * f * Math.exp(e);

            effect.localTime = localTime;
        }
    }
);
```

Obrázek 38: Registrování workletu pro animace.

2) Přidání workletu do hlavního javascriptového souboru

- Dalším krokem je vytvořený worklet přidat do hlavního javascriptového souboru a dodefinovat chování animace jako jsou trvání animace a atributy.
- Právě tato vlastnost, kombinování několika animací, je velkou předností animation API.
- Ve druhé části animace, na obrázku 40, jsme animaci dokončili. Je velmi důležité, jak animaci poskládáme. Prvním elementem s identifikátorem rotation se bude rotovat po ose Z. Druhý element reprezentován identifikátorem translation se bude pohybovat po ose X. Pokud bychom prohodili pořadí animací, které jsou přiděleny k jednotlivým elementům, dostali bychom jinou výslednou animaci. Z toho se dá odvodit, že animace fungují na principu matic, které nejsou kumulativní.

```
async function init() {
    await CSS.animationWorklet.addModule( moduleURL: "./gaussian.js");

    new WorkletAnimation(
        "gaussian",
        new KeyframeEffect(
            document.querySelector( selectors: "#rotation"),
             keyframes: [
                {
                    transform: "rotateZ(0deg) "
                },
                {
                    transform: "rotateZ(-280deg)"
                }
            ],
             options: {
                duration: 3000,
                iterations: Number.POSITIVE_INFINITY
            }
        ),
        document.timeline,
        {
            duration: 3000,
            distance: 750
        }
    ).play();
```

Obrázek 39: Definování první části animace.

```
    new WorkletAnimation(
        "gaussian",
        new KeyframeEffect(
            document.querySelector( selectors: "#translation"),
            keyframes: [
                {
                    transform: "translateX(0) "
                },
                {
                    transform: "translateX(750px)"
                }
            ],
            options: {
                duration: 3000,
                iterations: Number.POSITIVE_INFINITY
            }
        ),
        document.timeline,
        {
            duration: 3000,
            distance: 750
        }
    ).play();
}
init();
```

Obrázek 40: Definování druhé části animace.

- Jak si můžeme všimnout, vše je zabaleno ve funkci s názvem init(). Ta se spouští ihned po její deklaraci. Z toho plyne, že tyto funkce nesoucí animace můžeme spouštět na libovolných událostech dostupných v rámci DOM. Mluvíme o událostech jako jsou „onClick“, „onMouseOver“, …
- Dalším krokem po vytvoření funkce je přidání definovaného workletu. Poté už můžeme vytvářet samotné animace pomocí konstruktoru objektu WorkletAnimation. Konstruktor třídy WorkletAnimation přijímá 2 argumenty. Prvním je jméno workletu, které vkládáme ve formě textového řetězce. Druhým argumentem je instance třídy KeyframeEffect. Tato třída přijímá 3 argumenty.

Prvním argumentem je element, který bude animován. Druhý argument je javascriptové pole nesoucí informace o animaci typu keyframes. Keyframes obsahují CSS atributy a jejich vlastnosti. Třetí argument je opět pole, které nese informace o animaci, jako je doba trvání animace, počet opakování, … Celý konstruktor třídy KeyframeEffect je po definici animace spouštěn metodou play().

Výše uvedené ukázky kódů byly převzaty z článku, jejichž autorem Adrian Bece [1].

Animation worklet se vytvoří pomocí funkce registerAnimator(). Tato funkce se skládá ze dvou funkcí:

- constructor() – Je volán tehdy, když je vytvořena nová instance. Přijímá 2 vstupní argumenty. Prvním jsou možnosti (options), druhým argumentem je stav (state).
- animate()
  - Jedná se o hlavní funkci, která obsahuje logiku animace.
  - Vkládáme do ní tyto vstupní argumenty:
    - currentTime,
    - effect.

U workletu v tomto API je důležité vyvarovat se používání konstant. Aby animace fungovaly, je potřeba vkládat pouze vstupní argumenty. Jde tu hlavně o dynamiku, například – když uživatel bude skrolovat jinou rychlostí, která ovlivňuje rychlost animace, budeme vkládat proměnnou rychlost skrolování. V tom spočívá kouzlo CSS Houdini. Vše je dynamické a výkonné.

Ukažme si, jak by mohla vypadat animace, která by pracovala se stavem a na jeho základě by ovlivňovala více elementů. Zároveň si na této animaci ukážeme, jak lze skrolováním jednoho elementu ovlivnit animaci druhého elementu. Ukázka byla převzata z tohoto článku [13].

```
registerAnimator(
    "passthrough",
    class {
        constructor(options :{} = {}, state :{} = {}) {
            this.direction = state.direction;
        }

        animate(currentTime, effect) {
            effect.localTime = 2000 + this.direction * (currentTime % 2000);
        }
        destroy() {
            return {
                direction: this.direction
            };
        }
    }
);
```

Obrázek 41: Worklet pro animaci, která pracuje se stavem.

```
async function init() {
    await CSS.animationWorklet.addModule( moduleURL: "./passthrough-aw.js");
    new WorkletAnimation(
        'passthrough',
        new KeyframeEffect(
            document.querySelector( selectors: '#a'),
             keyframes: [
                {
                    transform: 'translateX(0)'
                },
                {
                    transform: 'translateX(500px)'
                }
            ],
             options: {
                duration: 2000,
                fill: 'both'
            }
        ),
        new ScrollTimeline({
            scrollSource: document.querySelector( selectors: 'main'),
            orientation: "vertical",
            timeRange: 2000
        })
    ).play();
}
init();
```

Obrázek 42: Animování elementu, který využívá worklet pracující se stavem. Zároveň je animován na základě skrolování v druhém elementu.

h) Layout API

Poslední API z rodiny CSS Houdini je Layout API. Dalo by se říci, že z hlediska složitosti a komplexnosti se jedná o jedno z nejsložitějších vysokoúrovňových API z CSS Houdini. API umožňuje vývojářům definovat nové modely rozvržení, které mohou být využity v atributu display v CSS. Podobně jako v předchozích dvou API, i zde máme pro pomoc worklet. I v tomto případě musí být nejdříve worklet registrován. Jako v každém API, kde jsme doposud používali worklety, i zde bude třeba modul přidat v html souboru. Worklet přidáme tímto způsobem: *CSS.layoutWorklet.addModule(„cesta k workletu");*. Opět musíme konkrétní worklet zavolat v CSS pod jeho jménem *display: layout(„jméno funkce pro vykreslení");*.

Jak jsme si už jistě mohli všimnout, API ze sady CSS Houdini neslouží primárně k tvorbě běžných věcí, kterých lze dosáhnout použitím standartních CSS. U Layout API tomu není jinak. Dnes můžeme pomocí běžného stylování v CSS dosáhnout zajímavých rozvržení. Máme pro to mnoho možností v rámci atributu display, ať se jedná o grid, flex, inline-block, block a další. Jedno mají však rozvržení společné, jsou striktně daná a jakékoli složitější rozvržení, kterým bychom chtěli naši stránku odlišit od těch ostatních, přináší další výkonnostní propad. Dobrým příkladem takového rozvržení je v poslední době oblíbený „masonry layout". Právě tento příklad ve svém článku použil Philip Walton, když uváděl příklad pro layout API. *"I když jsou tato rozložení působivá, bohužel trpí problémy s výkonem, zejména na méně výkonných zařízeních."* [18].

Opět zde máme různé funkce, které jsou specifické pro daný worklet. Jako každý předchozí worklet zde zavoláme metodu pro jeho registraci registerLayout(), která opět přijímá 2 argumenty. První atribut je název workletu. Druhým atributem je samotná funkce pro definování rozvržení. V těle funkce pro rozvržení máme, podobně jako v předchozích vysokoúrovňových API, statické metody pro definování:

- InputProperties() – pole hodnot obsahující rodičovský element, který volá tento layout.
- ChildrenInputProperties() – pole hodnot, které patří do potomka rodičovského elementu.
- LayoutOptions() – definuje následující vlastnosti rozvržení:
    - childDisplay – rozhoduje, zda bude zobrazení rovinné nebo v boxech. Nabývá hodnot block nebo normal,

- sizing – říká prohlížeči, zda má přepočítat nebo nepřepočítat velikost. Nabývá hodnot manual nebo block-like.

- IntrinsicSizes() – definuje, jak bude obsah zarovnáván do nového rozložení. Přijímá tyto argumenty:
  - children – dědí se od rodičovského elementu,
  - edges – určuje okraje layoutu,
  - styleMap – atribut pro práci s Typed Obejct Modelem.
- Layout() – hlavní funkce, která vypočítá rozvržení. Funkce disponuje těmito argumenty:
  - children – potomci rodičovského elementu,
  - edges – okraje rozvržení rodičovského elementu,
  - constraints – omezení rodičovského elementu,
  - styleMap – atribut pro práci s Typed Obejct Modelem u rodičovského elementu,
  - breakToken – používá se pro obnovení rozložení v případě stránkování nebo tisku.

Příklad užití si ukážeme na ukázce kódu, opět se jedná o ukázku z git uložiště [9].

```
/**
 * Copyright 2018 Google Inc. All Rights Reserved.
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *     http://www.apache.org/licenses/LICENSE-2.0
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
registerLayout('masonry', class {
  static get inputProperties() {
    return [ '--padding', '--columns' ];
  }

  async intrinsicSizes() { /* TODO implement :) */ }
  async layout(children, edges, constraints, styleMap) {
    const inlineSize = constraints.fixedInlineSize;

    const padding = parseInt(styleMap.get('--padding').toString());
    const columnValue = styleMap.get('--columns').toString();

    // We also accept 'auto', which will select the BEST number of columns.
    let columns = parseInt(columnValue);
    if (columnValue == 'auto' || !columns) {
      columns = Math.ceil(inlineSize / 350); // MAGIC NUMBER \o/.
    }
```

Obrázek 43:Ukázka Layout API, první část.

```
    // Layout all children with simply their column size.
    const childInlineSize = (inlineSize - ((columns + 1) * padding)) / columns;
    const childFragments = await Promise.all(children.map((child) => {
      return child.layoutNextFragment({fixedInlineSize: childInlineSize});
    }));

    let autoBlockSize = 0;
    const columnOffsets = Array(columns).fill(0);
    for (let childFragment of childFragments) {
      // Select the column with the least amount of stuff in it.
      const min = columnOffsets.reduce((acc, val, idx) => {
        if (!acc || val < acc.val) {
          return {idx, val};
        }

        return acc;
      }, {val: +Infinity, idx: -1});

      childFragment.inlineOffset = padding + (childInlineSize + padding) * min.idx;
      childFragment.blockOffset = padding + min.val;

      columnOffsets[min.idx] = childFragment.blockOffset + childFragment.blockSize;
      autoBlockSize = Math.max(autoBlockSize, columnOffsets[min.idx] + padding);
    }

    return {autoBlockSize, childFragments};
  }
});
```

Obrázek 44:Ukázka Layout API, druhá část.

i) Font Metrics API

CSS Houdini je stále ve vývoji. Oficiálně se píše podrobná specifikace. Právě Font Metrics API je jeden z modulů, které nejsou dokončené, ale jak již název avizuje, jeho úkolem bude pokročilá práce s fonty a s textem.

j) Parse API

Bohužel ani toto API není momentálně funkční. V tuto chvíli je ve stejném stavu jako Font Metrics API.

Zatím není známo, jakou úlohu toto API bude mít, šíří se ovšem informace o tom, že by mohlo rozšiřovat některé stávající CSS atributy o nová pravidla. Tuto myšlenku zmiňuje Philip Walton ve svém článku [18].

k) Analýza implementace CSS Houdini

Z toho, co již o probírané technologii víme, můžeme určit, kde bude skutečným přínosem a v jakých případech bude spíše na obtíž.

Použití CSS Houdini má skutečný význam tam, kde se budou muset hodnoty CSS atributů měnit na základě uživatelské interakce, času nebo okolního prostředí. Tyto atributy reprezentovány jako javascriptové objekty, které vytváříme pomocí nízkoúrovňových API prvků, lze poté používat v částech vysokoúrovňových API prvků. Nyní si rozebereme přínos jednotlivých API v oblastech, kde je smysluplná jejich implementace.

1. Typed Obejct Model

   Typed Object Model ve své podstatě nepřináší nové možnosti práce javascriptu s CSS, pouze dává jiný způsob, který může být efektivnější, pokud se s tímto přístupem naučíme zacházet. Při jeho správném požívání můžeme pak poznat výkonnostní progres.

2. Custom Properties and Values

   Custom Properties and Values už přináší poměrně zásadní změnu. Jedná se o možnost vytvářet CSS na straně javascriptu. Již dříve jsme mohli v CSS vytvářet proměnné, ale nemohli jsme na straně CSS definovat určité aspekty jako je dědičnost, datový typ nebo výchozí hodnotu.

3. Paint API

   Paint API dovoluje vykreslovat některé až neobyčejné efekty. Tyto efekty by se bez Paint API přístupu vytvářely velmi složitě. Vytváření vlastních provedení ohraničení, ať se jedná o tvar nebo barevnou škálu, bylo do doby před Paint API nemožné nebo velmi omezené. Je zřejmé, že ne každý vývojář bude chtít do svého

projektu vytvářet nové a obtížné efekty, které nejsou potřeba. Proto jsou zde již vytvořené knihovny, které stačí importovat a správně používat. Pokud bychom chtěli do svých projektů použít jiná, již hotová řešení, můžeme vždy použít příklady z git uložiště, které jsou při dodržení licenčních podmínek dostupné.

4. Animation API

   Animation API je druhým API z vysokoúrovňových API. I zde můžeme říct, že přináší nové a zásadní změny. Jak již bylo zmíněno v kapitole o animation API, tímto API můžeme docílit zvýšení výkonu při tvorbě parallaxových stránek. Tak docílíme vyššího snímkování a lepšího pocitu plynulosti. Kromě výkonnostních vylepšení API přináší možnost kombinování a vytváření několika animací najednou. To bylo možné i za použití běžného CSS, ale pouze v omezeném rozsahu. Animation API umožňuje určit konkrétní rychlost, plynulost, časování a průběh animace. Všechny aspekty si můžeme sami nadefinovat do posledního detailu. Pokud k tomu přidáme fakt, že animace můžeme kombinovat a přiřazovat ke konkrétním událostem („onClick“, „onMouseOver“ a další), máme zde neuvěřitelnou tvůrčí svobodu, v níž nás limituje pouze vlastní kreativita.

5. Layout API

   Layout API přináší možnosti, které tu doposud nebyly. Dříve jsme nemohli vytvářet vlastní rozvržení, jenž layout API dovoluje. Máme zde velkou variabilitu možností pro jejich vytváření, protože nám API umožňuje definovat si každý „child“ element ve vztahu k rodičovského elementu zvlášť. A tak to lze provést i naopak. Je samozřejmé, že nebudeme každý den vytvářet nová rozložení, ale v čem nám layout API opravdu pomáhá je optimalizace při vykreslování rozložení.

E. Analýza přímého přístupu k výrazům CSS

Po zanalyzování, jak nové nativní API CSS Houdini funguje a pracuje s CSS, bude pro realizaci praktické části nejvhodnější psaní neměnných atributů CSS pomocí standardního CSS. Proto se budeme v této kapitole zabývat tím, jak bychom mohli příklady vysokoúrovňových API replikovat pomocí standartního CSS.

Toto rozhodnutí je opodstatněno tím, že je nepraktické definovat pro každý CSS atribut novou metodu z Properties and Values API. Tento přístup je vhodný pouze v případě, že se daná vlastnost bude dynamicky měnit na základě interakce uživatele, času nebo prostředí.

a) Layout

Pojďme si ukázat, jak by se dal pomocí CSS vypracovat layout, který jsme viděli v ukázce v Layout API. Jedná se o rozvržení, kterému se přezdívá „masonry". Tohoto efektu docílíme pomocí javascriptu a CSS. Dnes zatím není možné vytvořit „masonry" rozložení pouze s použitím CSS, které by bylo kompatibilní se všemi prohlížeči. Momentální situace ohledně tohoto rozložení je taková, že se chystá její nativní podpora v CSS. Nyní je implementace „masonry" rozložení ve stavu, kdy je podporována pouze v Nightly verzi Firefoxu. Nightly Firefox prohlížeč je určen pouze pro vývojářské účely, proto v něm lze tento druh rozložení použít. Implementace „masonry" rozvržení v CSS se aplikuje následujícím způsobem.

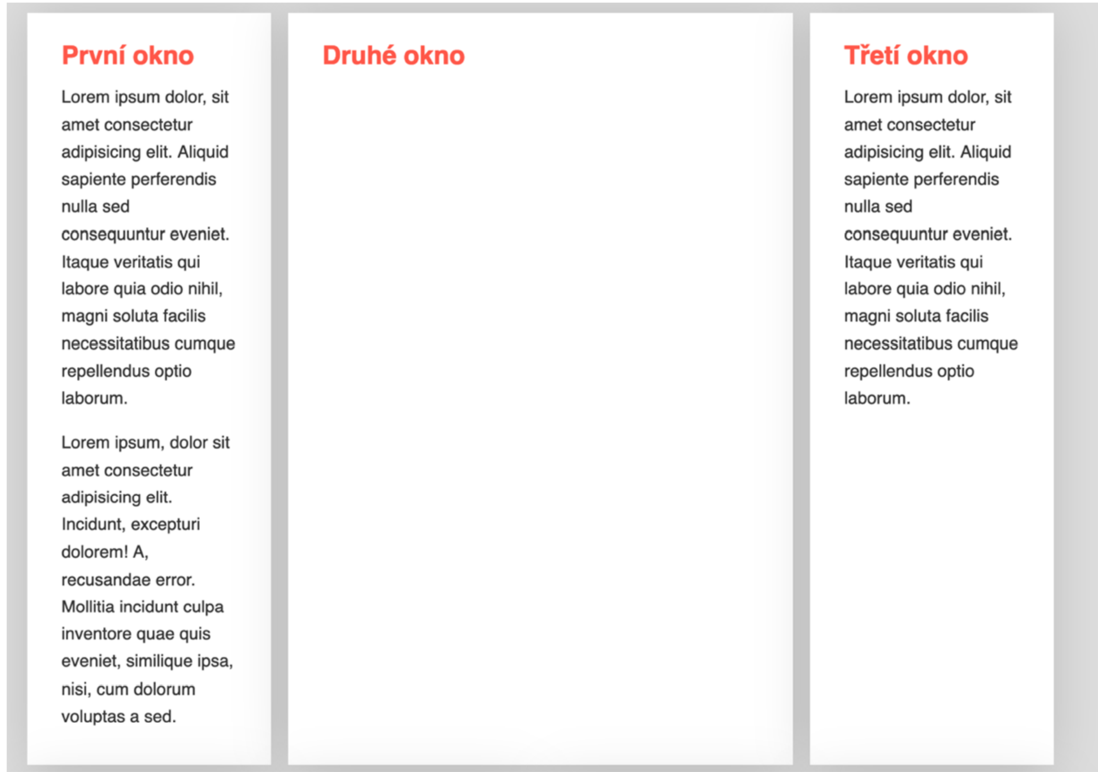
```
.grid {
    --gap: 1em;
    --columns: 4;
    max-width: 60rem;
    margin: 0 auto;
    columns: var(--columns);
    gap: var(--gap);
    display: grid;
    grid-template-columns: repeat(var(--columns), 1fr);
    grid-template-rows: masonry;
    grid-auto-flow: dense;
  }
```

Obrázek 45: Implementace "masonry" rozložení s použitím pouze CSS.

Klíčovým prvkem je zde atribut grid-template-rows nabývající hodnoty masonry. Právě tento prvek má být zahrnut ve třetí verzi pro mřížku, která je ve vývoji. Tento atribut se samozřejmě musí uplatnit v zobrazení mřížky. Všechny prvky ve třídě grid budou uspořádány do „masonry“ rozložení. Výstup v podporovaném prohlížeči pak vypadá následovně.

Obrázek 46: Výstup z podporovaného prohlížeče v závislosti na kódu ze 46. obrázku.

Takovýto bude výstup ze všech dnešních běžně používaných prohlížečů (Google Chrome, Mozila Firefox, Opera, Safari, …).



Obrázek 47: Výstup z nepodporovaného prohlížeče v závislosti na kódu ze 46. obrázku.

Nyní se pojďme podívat, jak dnes můžeme „masonry“ rozložení aplikovat na všechny prohlížeče, a to bez použití Layout API. Aby bylo možné docílit tohoto efektu, bude zapotřebí si v javascriptu vytvořit instanci třídy Masonry. V prvním argumentu vkládáme do konstruktoru třídy Masonry element, na kterém má být rozložení aplikováno. V druhém argumentu už definujeme funkci, která určuje parametry rozložení. Nesmíme ani zapomenout na importování modulu, který zajišťuje podporu rozložení. Poslední věcí, na kterou je třeba dát si pozor, je rozložení html elementů a jaké třídy elementům přiřazujeme. Všechny podrobné informace, včetně ukázky, kterou si ukážeme, naleznete na odkazu níže [5].

Nyní se už podívejme na samotnou implementaci.

```
<div class="grid">
  <div class="grid-item"></div>
  <div class="grid-item grid-item--width2 grid-item--height2"></div>
  <div class="grid-item grid-item--height3"></div>
  <div class="grid-item grid-item--height2"></div>
  <div class="grid-item grid-item--width3"></div>
  <div class="grid-item"></div>
  <div class="grid-item"></div>
  <div class="grid-item grid-item--height2"></div>
  <div class="grid-item grid-item--width2 grid-item--height3"></div>
  <div class="grid-item"></div>
  <div class="grid-item grid-item--height2"></div>
  <div class="grid-item"></div>
  <div class="grid-item grid-item--width2 grid-item--height2"></div>
  <div class="grid-item grid-item--width2"></div>
  <div class="grid-item"></div>
  <div class="grid-item grid-item--height2"></div>
  <div class="grid-item"></div>
  <div class="grid-item"></div>
  <div class="grid-item grid-item--height3"></div>
  <div class="grid-item grid-item--height2"></div>
  <div class="grid-item"></div>
  <div class="grid-item"></div>
  <div class="grid-item grid-item--height2"></div>
</div>

<script src="https://unpkg.com/masonry-layout@4/dist/masonry.pkgd.min.js"></script>
<script>
    let elem = document.querySelector('.grid');
    let msnry = new Masonry( elem, {
    itemSelector: '.grid-item',
    columnWidth: 100
    });
</script>
```

Obrázek 48: Implementace "masonry" rozložení pro všechny prohlížeče.

b) Animace

Podobně jako v předchozí ukázce zkusíme zreprodukovat příklad z ukázky z kapitoly Animation API, kde jsme viděli, jak pomocí CSS Houdini můžeme vytvořit takzvaný parallaxový efekt.

```css
body, html{
  height:100%;
  margin:0;
  font-size:18px;
  font-family: 'Trebuchet MS', serif;
  font-weight:500;
  line-height:2rem;
  color: #282e34;
}

.part1, .part2, .part3{
  position:relative;
  background-position:center;
  background-size:cover;
  background-repeat:no-repeat;
  background-attachment:fixed;
}

.part1{
  background-image:url('../img/flowers.jpg');
  min-height:100%;
}

.part2{
  background-image:url('../img/chair.jpg');
  min-height:600px;
}

.part3{
  background-image:url('../img/sunset.jpg');
  min-height:600px;
}

.section{
  text-align:center;
  padding:60px 70px;
}
```

Obrázek 49: Parallaxový efekt s použitím pouze CSS, první část.

```css
.section-dark{
  background-color:#282e34;
  color:#dddd;
}

.ptext{
  position:absolute;
  top:50%;
  width:100%;
  text-align:center;
  color:#282e34;
  font-size:30px;
  letter-spacing:8px;
  text-transform:uppercase;
}

.ptext .border{
  background-color:#282e34;
  color:#fff;
  padding:25px;
}

.ptext .border_trans{
  color:#fff;
  background-color:transparent;
}

@media(max-width:568px){
  .part1, .part2, .part3{
    background-attachment:scroll;
  }
}
```

Obrázek 50: Parallaxový efekt s použitím pouze CSS, druhá část.

Na obrázcích vidíme CSS kód. Všimněme si, že ačkoli replikujeme ukázku z kapitoly o animation API, při použití standartních CSS žádnou animaci nepoužíváme. Je zřejmé, že stránku využívající parallaxový efekt lze vytvořit pouze pomocí CSS bez nutnosti implementace javascriptu. Jak jsme však mohli číst v kapitole Animation API, kde se odkazujeme na článek, který řeší problematiku s parallaxovým efektem, tak už víme, že při použití animačního workletu lze dosáhnout většího výkonu a díky tomu při skrolování i vyšší snímkovací frekvenci.

c) Vykreslování

Abychom byli schopni napodobit efekt, který je vytvořen pomocí Paint API, je vhodné vybrat efekt, který se dynamicky nemění. Proto vytvoříme takzvaný „ripple efekt", který se bude spouštět při kliknutí na tlačítko.

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">

    <link rel="stylesheet" href="style.css">
  </head>
  <body>
        <button href="#">Click me</button>
<script>
    const buttons = document.querySelectorAll("button");
    buttons.forEach((button) => {
      button.onclick = function(e){
        let x = e.clientX - e.target.offsetLeft;
        let y = e.clientY - e.target.offsetTop;
        let ripple = document.createElement("span");
        ripple.style.left = `${x}px`;
        ripple.style.top = `${y}px`;
        this.appendChild(ripple);
        setTimeout(function(){
          ripple.remove();
        }, 1000);
      }
    });

    </script>

  </body>
</html>
```

Obrázek 51: Ripple efekt pomocí javascriptu a CSS, javascriptový soubor.

```css
button{
    border: none;
    position: relative;
    display: inline-block;
    padding: 12px 36px;
    margin: 10px 0;
    text-decoration: none;
    font-size: 20px;
    letter-spacing: 1px;
    border-radius: 4px;
    background: linear-gradient(60deg,#ff00d4, #00ddff );
    overflow: hidden;
}
button:focus {outline:0;}

span{
    position: absolute;
    background: white;
    transform: translate(-50%, -50%);
    pointer-events: none;
    border-radius: 50%;
    animation: animate 1s linear infinite;
}

@keyframes animate{
    0%{
        width: 0px;
        height: 0px;
        opacity: 0.5;
    }
    100%{
        width: 500px;
        height: 500px;
        opacity: 0;
    }
}
```

Obrázek 52:Ripple efekt pomocí javascriptu a CSS, CSS soubor.

Je očividné, že tento efekt nelze provést pouze se standartním CSS, proto jsme si museli vypomoci javascriptem. Na příkladu můžeme vidět, že k provedení tohoto efektu nebudeme muset použít Paint API, lze se obejít i bez jeho implementace. Lisi Linhart ve svém článku porovnává výkonnostní rozdíl mezi Paint API a standartním vykreslováním CSS. Výsledkem testu bylo, že vykreslovací proces je pomocí Paint API znatelně rychlejší. Musíme však uvážit, že k provedení efektu je zapotřebí velkého množství skriptování a konečný čas je ve výsledku téměř stejný [10].

F. Závěr

Jak jsme mohli vidět, CSS Houdini přináší nové možnosti, jakým způsobem prostřednictvím javascriptu pracovat s CSS a postupně posouvat limity stylování. Je ovšem škoda, že CSS Houdini stále není kompletně hotový. To samozřejmě souvisí s podporou prohlížečů a připraveností samotného API. Jako příklad si uveďme Font Metrics API a Parse API, která nejsou stále hotová. Na obrázku číslo 54 můžeme vidět aktuální podporu CSS Houdini napříč prohlížeči [14].

	Google Chrome	Microsoft Edge	Opera	Samsung Internet	Mozilla Firefox	Apple Safari	Spec
Engine	Blink				Gecko	WebKit	-
Paint API (Explainer Demos Article)	Shipped (Chrome 65) Details	Shipped (Edge 79) Details	Shipped (Opera 52) Details	Shipped (Internet 9.2) Details	Under consideration Details	In Development Details	Candidate Recommendation Spec
Properties & Values API (Demos Article)	Shipped (Chrome 78) Details	Shipped (Edge 79) Details	Shipped (Opera 65) Details	Shipped (Internet 12.0) Details	Under consideration Details	Partial support (Safari TP 67) Details	Working Draft Spec
Typed OM (Explainer Article)	Shipped (Chrome 66) Details	Shipped (Edge 79) Details	Shipped (Opera 53) Details	Shipped (Internet 9.2) Details	Under consideration Details	In Development Details	Working Draft Spec
Layout API (Explainer Demos)	Partial support (Canary) Details	Partial support (Canary) Details	Partial support (Developer) Details	No signal	Under consideration Details	Under consideration Details	First Public Working Draft Spec
AnimationWorklet (Explainer Demos Article)	Partial support (Chrome 71) Details	Partial support (Edge 79)	Partial support (Opera 58)	Partial support (Internet 10.2)	No signal	Under consideration Details	First Public Working Draft Spec
Parser API (Explainer)	No signal	No signal	No signal	No signal	No signal	No signal	Proposal Spec
Font Metrics API (Explainer)	No signal	No signal	No signal	No signal	No signal	No signal	Proposal Spec

Obrázek 53: Podpora CSS Houdini napříč prohlížeči.

V bakalářské práci se povedlo předvést, jak je CSS Houdini schopen pracovat ve vykreslovacím procesu mnohem rychleji, než toho jsou prohlížeče schopné docílit dnes. Tento fakt jsme si mohli vyzkoušet. V případech, kdy jsme nebyli schopni výkonnost vykreslovacího procesu ověřit, mohli jsme se o ní dočíst v článcích, kde autoři popsali výsledky svých testování.

Dalším důležitým poznatkem, na který jsme v bakalářské práci poukázali, byla jednotlivá použití konkrétních druhů API. V nízkoúrovňových API jsme demonstrovali, jak CSS Houdini přináší nový způsob přístupu k CSS prostřednictvím javascriptu. Tento nový přístup zprostředkovává Typed Object Model (TOM), který je nepatrně výkonnější než CSSOM. Typed Object Model umí pracovat s již existujícími třídami využívajícími javascript pro práci s CSS prvky. TOM zprostředkovává přístup k objektům, které jsou vytvořené

konstruktory metod pomocí getterů a setterů. Settery nám umožňují zřetelněji zapisovat hodnoty. Tento přístup jsme si ukázali na příkladu výpočtu délky elementu na základě uživatelského vstupu. V příkladu jsme použitím getterů a setterů získávali a nastavovali hodnoty pro přesný výpočet procent a přesnou délku elementů. Druhým API, které jsme si ukázali, bylo Custom Properties and Values rozšiřující CSS proměnné o typování, dědičnost a výchozí hodnotu. Tato nízkoúrovňová API rozšiřují již dostupné funkce. Běžně se CSS proměnná deklarovala v CSS souboru. S Custom Properties and Values můžeme tyto proměnné definovat v javascrptu, kde jim definujeme dědičnost, datový typ, jméno a počáteční hodnotu. Custom Propertes and Values má svůj největší přínos při vytváření gradientních přechodů, kterými lze měnit hodnoty na základě uživatelské interakce. Konkrétně jsme si předvedli, jak se plynule mění pouze jedna barva z gradientního přechodu u přejetí nad elementem myši. Tohoto efektu jsme docílili pouze pomocí CSS, javascript byl použit pouze pro definování proměnné.

Druhou podmnožinou CSS Houdini jsou vysokoúrovňová API, která už jsou komplikovanější a komplexnější než nízkoúrovňová API. U každého vysokoúrovňového API jsme našli případy, kdy je výhodnější a snazší použít alternativní metodiku k vytvoření chtěného efektu. Naopak zde byly i případy, kdy bychom bez CSS Houdini těchto efektů nemohli dosáhnout nebo zde byl znatelný výkonnostní rozdíl. Mezi vysokoúrovňová API přinášející nové možnosti nebo znatelný rozdíl lze řadit Animation API a Layout API. U Layout API jsme si představili vytvoření nového masonry rozložení. Tento druh rozložení můžeme vytvořit i bez CSS Houdini, ale pokud si budeme chtít vytvořit jakékoli jiné rozložení, které není v rámci CSS ani javascriptu dostupné, s Layout API toho je možné dosáhnout. Animation API worklet umožňuje kombinování několika po sobě jdoucích animací s uchováváním stavu, a to je u CSS něco zcela nového. Uplatnění tohoto API jsme si ukázali na parallaxové stránce, kde bylo dosaženo významného výkonnostního progresu, který se projevil na snímkování. Posledním nezmíněným API je Paint API. Jedná se o API nabízející velkou řadu možností. Vytvořili jsme si hned několik efektů, které lze s API vytvořit. Vytvoření několikabarevného ohraničení elementu s proměnlivou tloušťkou hranice nebo „border-radius-reverse“ efektu by nebylo možné dosáhnout bez použití Paint API. Při výkonnostním testu bylo zjištěno, že API má podobný výkon jako při běžném přístupu. Konkrétně bylo API výkonnější při samotném vykreslování, ale tento benefit byl vykompenzován velkou mírou skriptování.

V závěru můžeme říct, že CSS Houdini v některých případech přináší opravdu nové možnosti, které budou schopní vývojáři moci využít, například vytváření různých variant ohraničení, vlastních rozložení, nebo rozsáhlých a komplikovaných animací. Tam kde CSS Houdini nepřináší žádné nové možnosti, umožňuje dosáhnout rychlejšího vykreslovacího procesu.

G. Možnosti dalšího pokračování

CSS Houdini má potenciál být mezi vývojáři velmi populární. Důvodem je neomezená možnost vytváření nových efektů. Bohužel, jak je nám z praxe známo, implementace změn na poli CSS bývá zdlouhavým procesem. Jakmile však bude CSS Houdini plně podporován napříč všemi prohlížeči, bude mít své místo u vývojářů jisté.

V návaznosti na tuto bakalářkou práci by bylo zajímavé pozorovat implementaci CSS Houdini společně s některým z frontendových rámců, jako je React nebo Vue. Tato kombinace by byla rozhodně pozoruhodná z hlediska sledování výkonnostního progresu. Pokud je CSS Houdini schopný zprostředkovat vyšší vykreslovací výkon, pak by spolupráce s Reactem a jeho state managementem pro optimalizaci vykreslování znamenala docílení dalšího ušetření času potřebného pro vykreslení.

H. Seznam použitých zdrojů a literatury

[1] BECE Adrian, A practical overview of CSS Houdini [online],19.4 .2020, dostupné z: https://www.smashingmagazine.com/2020/03/practical-overview-css-houdini/, přístup ze dne: 15. 4. 2021

[2] BENGTSSON Peter, BERSHANSKIY Anton a SCHONNING Nick, Worklet [online], 8 .2 .2021, dostupné z: https://developer.mozilla.org/en-US/docs/Web/API/Worklet, přístup ze dne: 15. 4. 2021

[3] BENGTSSON Peter, BERSHANSKIY Anton a SCHONNING Nick, CSSUnparsedValue [online], 19.2 .2021, dostupné z: https://developer.mozilla.org/en-US/docs/Web/API/CSSUnparsedValue, přístup ze dne: 15. 4. 2021

[4] COYIER Chris, Custom properties as state [online], 5 .1 .2021, dostupné z: https://CSS-tricks.com/custom-properties-as-state/, přístup ze dne: 15. 4. 2021

[5] DESANDRO David, Masonry layout [online], datum neuvedeno, dostupné z: https://masonry.desandro.com/options.html, přístup ze dne: 15. 4. 2021

[6] DINIZ Adam Arthur, HOSPODARETS Serg, KILPATRICK Ian, LUNDIN Portus, SURMA, Border-color [online], 5 .4 .2018, dostupné z: https://github.com/GoogleChromeLabs/houdini-samples/tree/master/paint-worklet/border-color, přístup ze dne: 15. 4. 2021

[7] DINIZ Adam Arthur, HOSPODARETS Serg, KILPATRICK Ian, LUNDIN Portus, SURMA, Ripple [online], 5 .4 .2018, dostupné z: https://github.com/GoogleChromeLabs/houdini-samples/tree/master/paint-worklet/ripple, přístup ze dne: 15. 4. 2021

[8] KRAVETS Una, A CSS Houdini library for making your site a little more #extra.[online] 11.11.2020, dostupné z: https://extra-CSS.netlify.app/ přístup ze dne: 15.04.2021

[9] KRAVETS Una, SURMA, TOCCHINI Dan, Layout-worklet-masonry [online], 22.9 .2019, dostupné z: https://github.com/GoogleChromeLabs/houdini-samples/blob/master/layout-worklet/masonry/masonry.js, přístup ze dne: 15. 4. 2021

[10] LINHART Lisa, Performance – CSSPainting versus CSS Houdini Paint API [online], 1 .7 .2020, dostupné z: https://lisilinhart.info/posts/CSS-houdini-performance/, přístup ze dne: 15. 4. 2021

[11] MEDLEY Joe, SMITH Michael, WEYL Estelle, CSSPositionValue[online], 4 .1 .2021, dostupné z: https://developer.mozilla.org/en-US/docs/Web/API/CSSPositionValue, přístupný ze dne: 15. 4. 2021

[12] SOUTO Mario, Border-radius-reverse [online], 24.8 .2018, dostupné z: https://github.com/omariosouto/border-radius-reverse, přístup ze dne: 15. 4. 2021

[13] SURMA, Animation API [online], 24.7 .2020, dostupné z: https://developers.google.com/web/updates/2018/10/animation-worklet, přístup ze dne: 15. 4. 2021

[14] SURMA, Is houdini ready yet? [online], 26.10.2020, dostupné z: https://ishoudinireadyyet.com, přístup ze dne: 15. 4. 2021

[15] TUDOR Ana, Moon grid with clip-path [online], 17.7 .2018, dostupné z: https://codepen.io/thebabydino/pen/zLqbxX, přístup ze dne: 15. 4. 2021

[16] VALIPOUR Majid, Parallax-scrolling [online], 31.7 .2018, dostupné z: https://github.com/GoogleChromeLabs/houdini-samples/tree/master/animation-worklet/parallax-scrolling, přístup ze dne: 15. 4. 2021

[17] W3.org, CSS-TAG Houdini Editor Drafts [online], 8 .2 .2021, dostupné z: https://drafts.CSS-houdini.org, přístup ze dne: 15. 4. 2021

[18] WALTON Phillip, Houdini_ Maybe the most exciting development in CSS you've never heard of [online], 24.3 .2016, dostupné z: https://www.smashingmagazine.com/2016/03/houdini-maybe-the-most-exciting-development-in-CSS-youve-never-heard-of/, přístup ze dne: 15. 4. 2021

[19] WILSON Dan, Dance of the Tiled Houdini Hexagons [online], 10.9 .2018, dostupné z: https://codepen.io/danwilson/pen/ddNYeV, přístup ze dne: 15. 4. 2021

I. Seznam použitých obrázků

Obrázek 1: Ukázka práce CSSOM s HTML elementy. 11
Obrázek 2: Ukázka práce CSSObject Model s atributy CSS. 11
Obrázek 3: Ukázka výstupu práce CSSOM s textem. 11
Obrázek 4: Ukázka výstupu práce CSSOM do konzole. 12
Obrázek 5: Ukázka práce pomocí StylePropertyMap s parametry CSS. 14
Obrázek 6: Ukázka práce s CSS parametry pomocí takzvaného „inline“ zápisu. 14
Obrázek 7: Ukázka práce CSSKeywordValueSerialization. 15
Obrázek 8: Ukázka dopočítávání délky elementu na základě rodičovského elementu pomocí CSS Houdini. 16
Obrázek 9: Ukázka změny barvy textu pomocí CSS Houdini. 16
Obrázek 10: Ukázka výstupů změny barvy textu pomocí přístupů CSSOM a TOM. 17
Obrázek 11: Demonstrace metod delete() a append(). 17
Obrázek 12: Ukázka konstruktoru CSSUnparsedValue, který používáme v ukázce kalkulace šířky elementu. 18
Obrázek 13: Ukázka metody parse() u třídy CSSTransformValue. 19
Obrázek 14: Ukázka výpisu konzole pro metodu parse() použitou u třídy CSSTransformValue. 19
Obrázek 15: Ukázka metody parseAll u třídy CSSTransformValue. 20
Obrázek 16: Ukázka výpisu konzole pro metodu parseAll() pro třídu CSSTransformValue. 20
Obrázek 17: Kód ukazující metody pro sčítání hodnot a převádění jednotek v rámci třídy CSSNumericValues. 22
Obrázek 18: Ukázka kódu, ukazující pokročilejší metody třídy CSSNumericValues. 22
Obrázek 19: Výstup do konzole z ukázek na 19. obrázku. 22
Obrázek 21: Výpis do konzole z obrázku číslo 18. 23
Obrázek 20: Výpis do konzole s metodou toString() z 18. obrázku. 23
Obrázek 22: Ukázka použití CSSPositionValue z webové stránky https://developer.mozilla.org. 25
Obrázek 23: Ukázka práce uživatelské interakce pomocí CSS Houdini. 27
Obrázek 24: Výsledek výkonnostního testu mezi Typed Object Model a CSSObject Model – Typed Object Model. 28

Obrázek 25: Výsledek výkonnostního testu mezi Typed Object Model a CSSObject Model – CSS Object Model. 28
Obrázek 26: Ukázka deklarace CSS proměnné. 29
Obrázek 27: Ukázka použití CSS proměnné v gradientních přechodech. 30
Obrázek 28: Ukázka aplikace fallback hodnoty. 31
Obrázek 29: Ukázka změny hodnoty proměnné pomocí javascriptu. 31
Obrázek 30: Výstup z konzole z kódu na obrázku číslo 29. 32
Obrázek 31: Ukázka práce Any Tudor s názvem "moon-grid with clip-path". 32
Obrázek 32: Vytvoření workeltu pro vykreslování. 34
Obrázek 33: Importování workletu z externí knihovny. 35
Obrázek 34: Volání metody paint() v CSS souboru. 36
Obrázek 35: Efekt, který byl vytvořen pomocí importování knihovny. 37
Obrázek 36: Několikabarevná hranice elementu. 37
Obrázek 37: Efekt obrácených rohů u hranice elementu. 38
Obrázek 38: Registrování workletu pro animace. 39
Obrázek 39: Definování první části animace. 40
Obrázek 40: Definování druhé části animace. 41
Obrázek 41: Worklet pro animaci, která pracuje se stavem. 43
Obrázek 42: Animování elementu, který využívá worklet pracující se stavem. Zároveň je animován na základě skrolování v druhém elementu. 43
Obrázek 43:Ukázka Layout API, první část. 45
Obrázek 44:Ukázka Layout API, druhá část. 46
Obrázek 45: Implementace "masonry" rozložení s použitím pouze CSS. 49
Obrázek 46: Výstup z podporovaného prohlížeče v závislosti na kódu ze 46. obrázku. 50
Obrázek 47: Výstup z nepodporovaného prohlížeče v závislosti na kódu ze 46. obrázku. 51
Obrázek 48: Implementace "masonry" rozložení pro všechny prohlížeče. 52
Obrázek 49: Parallaxový efekt s použitím pouze CSS, první část. 53
Obrázek 50: Parallaxový efekt s použitím pouze CSS, druhá část. 54
Obrázek 51: Ripple efekt pomocí javascriptu a CSS, javascriptový soubor. 55
Obrázek 52:Ripple efekt pomocí javascriptu a CSS, CSS soubor. 56
Obrázek 53: Podpora CSS Houdini napříč prohlížeči. 57

J. Přílohy

Příloha č.1

UNIVERZITA HRADEC KRÁLOVÉ
Fakulta informatiky a managementu
Akademický rok: 2019/2020

Studijní program: Aplikovaná informatika
Forma studia: Prezenční
Obor/kombinace: Aplikovaná informatika (ai3-p)

Podklad pro zadání BAKALÁŘSKÉ práce studenta

Jméno a příjmení: Jakub Tichý
Osobní číslo: I1900772
Adresa: Na Drahách 399, Rtyně v Podkrkonoší, 54233 Rtyně v Podkrkonoší, Česká republika

Téma práce: CSS Houdini: programové rozhraní k vlastnostem a hodnotám jazyka CSS
Téma práce anglicky: CSS Houdini: CSS Properties and Values API

Vedoucí práce: Mgr. Daniela Ponce, Ph.D.
Katedra informačních technologií

Zásady pro vypracování:

Cíl: představit API jazyka CSS, ukázat jeho možnosti, přínosy a omezení.
Osnova

1. Úvod
2. Cíl práce
3. Analýza přímého přístupu k výrazům CSS
4. Analýza přístupu k výrazům CSS prostřednictvím API
5. Praktické ukázky implementace obou přístupů
6. Porovnávání a vyhodnocení obou přístupů
7. Výsledky práce
8. Shrnutí
9. Použitá literatura

Seznam doporučené literatury:

1. Jeremy Keith, Jeffrey Sambells: DOM Scripting: Web Design with JavaScript and the Document Object Model.
2. Tab Atkins-Bittner, Daniel Glazman, Alan Stearns, Greg Whitworth: CSS Properties and Values API Level 1 (https://github.com/w3c/css-houdini-drafts/wiki)
3. Sam Richard: CSS Houdini(https://houdini.glitch.me/)
4. Adrian Bece: A practical overview of CSS Houdini(https://www.smashingmagazine.com/2020/03/practical-overview-css-houdini/)

Podpis studenta: Datum: 27. 4. 2021

Podpis vedoucího práce: Datum:

© IS/STAG, Portál – Podklad kvalifikační práce , tichyja1, 27. dubna 2021 16:08