LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

TĪMEKĻA LIETOTŅU KLIENTA PUSES OPTIMIZĀCIJA

BAKALAURA DARBS

Autors: **Romāns Kolduns**
Studenta apliecības numurs: rk13025
Vadītājs: M. dat., vecākais programmētājs, Pēteris Prokofjevs

RĪGA 2017

ANOTĀCIJA

Bakalaura darba mērķis ir izpētīt, kā interneta pārlūkprogrammas algoritmiski nodrošina tīmekļa vietņu attēlošanu un CSS stilu likumu piemērošanu un aplūkot šo algoritmu optimizācijas iespējas.

Darba 1. nodaļa sniedz teorētisku ieskatu mūsdienu interneta pārlūkprogrammu darbības pamatprincipos un tīmekļa lapu ielādes procesā, ar mērķi noskaidrot potenciālos veiktspējas krituma cēloņus. 2. nodaļa veltīta tipiskām veiktspējas problēmām, kuras ir aktuālas, izstrādājot efektīvas tīmekļa lietotnes, kā arī ir apskatīti iespējamie šo problēmu risinājumi. Savukārt 3. nodaļā aprakstīti praktiski eksperimenti un reālu problēmu risināšanas piemēri.

Darbā izvirzītais uzdevums ir izpētīt tīmekļa lapu ielādes un attēlošanas optimizācijas iespējas un pielietot tās praktiskos risinājumos.

Atslēgas vārdi: veiktspēja, DOM, Javascript, CSS

ABSTRACT

PERFORMANCE OPTIMIZATION OF CLIENT-SIDE WEB APPLICATIONS

The purpose of this paper is to find out how web browsers render web pages and apply CSS style rules, and provide optimizations to web applications according to these findings.

The first chapter covers theoretical insight in modern web browser work principles and explains about web page loading process in order to determine potential performance issues. The second chapter underlines performance issues that can appear when working with web applcations development. Possible solutions are also explained. The third chapter contains information on practical approaches to real-word problems.

The main goal of this paper is to research ways how to optimize web application load and rendering performance and use them in real web applications.

Key words: performance, DOM, Javascript, CSS

SATURA RĀDĪTĀJS

APZĪMĒJUMU SARAKSTS

API (Application Programming Interface) jeb lietojumprogrammas saskarne ir gatavu klašu un funkciju kopums, kuru var izmantot ārējas programmas.

BNF (Bekusa-Naura) forma – sintakses aprakstīšanas formālā sistēma, kurā vienas sintaktiskās kategorijas var noteikt ar citu kategoriju palīdzību.

CSS (Cascading Style Sheets) – formāla valoda, ko izmanto HTML un XML dokumentu elementu vizuālā noformējuma definēšanai.

CSS selektors – pieraksts, kurš definē nosacījumus, lai atrastu vēlamos DOM elementus.

CSSOM (CSS Object Model) – kokveida struktūra, kurā CSS selektoriem atbilst noteikti CSS likumi.

DOM (Document Object Model) – kokveida struktūra, ar kuras palīdzību Javascript var gūt piekļuvi HTML dokumentu saturam un rediģēt tos. Tā ir tīmekļa lapas struktūras attēlojums.

DTD (Document Type Definition) – iezīmēšanas deklarāciju kopums, kas definē HTML vai XML dokumenta tipu.

HTML – standarta iezīmēšanas valoda, ko izmanto tīmekļa vietņu saskarnes definēšanai.

Javascript – skriptu valoda, ko lieto tīmekļa lietotņu izstrādē.

Leksēma (lexeme, token) – simbolu virkne, kas veido sintaktisko vienību jeb "vārdu", piemēram, HTML birkas vai CSS īpašības nosaukumu.

Renderēšana – process, kurā pārlūkprogramma no dota HTML koda grafiski attēlo tajā definētos elementus uz ekrāna.

Renderēšanas koks – kokveida struktūra, kas parāda, kā DOM koka mezgli tiks rādīti uz ekrāna.

Steku konteksts – HTML elementu trīsdimensiju attēlojums pa iedomātu Z-asi, attiecībā pret lietotāju, kurš redzēs tīmekļa lapas saturu.

WebKit – tīmekļa vietņu attēlošanas dzinējs, kuru izmanto Safari, Opera un Google Chrome pārlūks.

XML (eXtensible Markup Language) – iezīmēšanas valoda, ko mēdz izmantot tīmekļa vietņu saskarnes definēšanai.

IEVADS

Līdz ar mūsdienu straujo informācijas tehnoloģiju attīstību, cilvēki biežāk izmanto internetu, lai veiktu darbības, ko iepriekš veica bez datora. Parādās arvien jaunas iespējas, kuras agrāk nebija pieejamas. Tīmekļa tehnoloģijas kļūst daudzveidīgākas un sarežģītākas. Palielinās uzdevumu daudzums, ko var paveikt, izmantojot interneta pārlūkprogrammu, neuzstādot datorā papildu programmatūru.

Par ierastu parādību kļūst ne tikai iespēja pārlūkot statiskas interneta vietnes, bet arī iespēja strādāt ar dinamiskām tīmekļa lietotnēm. Lietotāji sagaida, ka tīmekļa lietotnes spēj piedāvāt tikpat daudzveidīgu funkcionalitāti un bagātīgu lietošanas pieredzi kā tradicionālās datorprogrammas un pat aizstāt tās.

Tomēr līdz ar straujo informācijas tehnoloģiju attīstību, tīmekļa lietotnes kļūst arvien komplicētākas. Palielinās internetā pārsūtāmo un apstrādājamo datu apjomi, kā arī paaugstinās lietotāju prasības pret lietotņu veiktspēju. Lietotāji strādā ar tīmekļa lietotni, izmantojot tās grafisko saskarni. Tieši no saskarnes lielā mērā ir atkarīga visas tīmekļa lietotnes veiktspēja pēc tam, kad ir lejupielādēti visi vajadzīgie pirmkoda faili un dati.

Cilvēkiem strādājot ar tīmekļa lietotnēm, tiek izdarītas daudzas dinamiskas izmaiņas, kuras prasa atsevišķu ekrāna daļu satura pārzīmēšanu vai jebkādu citādu pārveidošanu. Tomēr pārzīmēšana var būt visai laikietilpīgs process, kurš turklāt notiek ar pieprasītā satura aizkavēm. Tāpēc šī darba ietvaros tiek pētīts, kā interneta pārlūkprogrammas algoritmiski nodrošina lietotāju mijiedarbību ar grafisko saskarni, lai būtu iespējams veikt attiecīgus ātrdarbības uzlabojumus. Tīmekļa lietotņu saskarnes veiktspēja ir viens no būtiskākajiem lietotāju apmierinātības faktoriem.

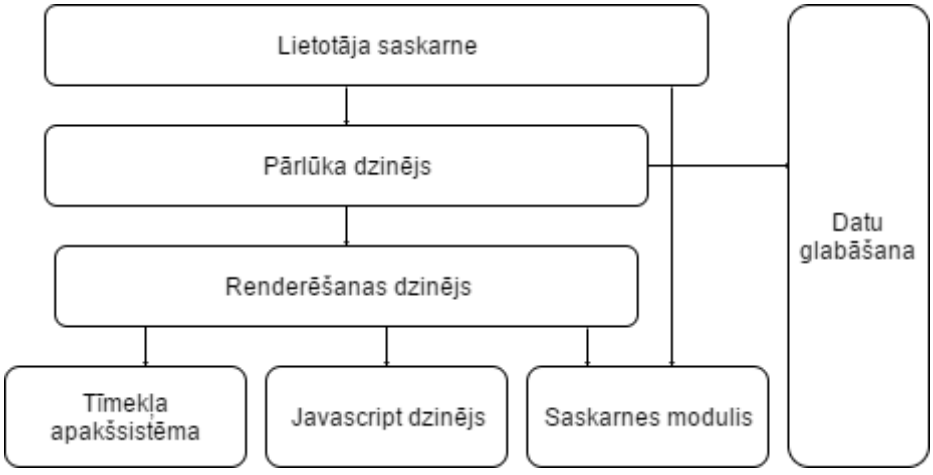
Darbā tiek salīdzināta divu pārlūkprogrammu dzinēju: WebKit (izmantots Google Chrome un Safari pārlūkos) un Gecko (izmantots Mozilla Firefox) darbība.

1. TĪMEKĻA VIETŅU ATTĒLOŠANA PĀRLŪKPROGRAMMĀS

Šajā nodaļā tiek vispārīgi aprakstīti pārlūkprogrammas darbības pamatprincipi. Ir nepieciešams izprast, kā darbojas katra tīmekļa pārlūkprogrammas komponente, ar mērķi izdomāt, kā izmantot pārlūku darbības īpatnības, lai izstrādātu efektīvākas tīmekļa lietotnes.

Interneta pārlūki sastāv no vairākām komponentēm [1]. Nozīmīgākās no tām ir:

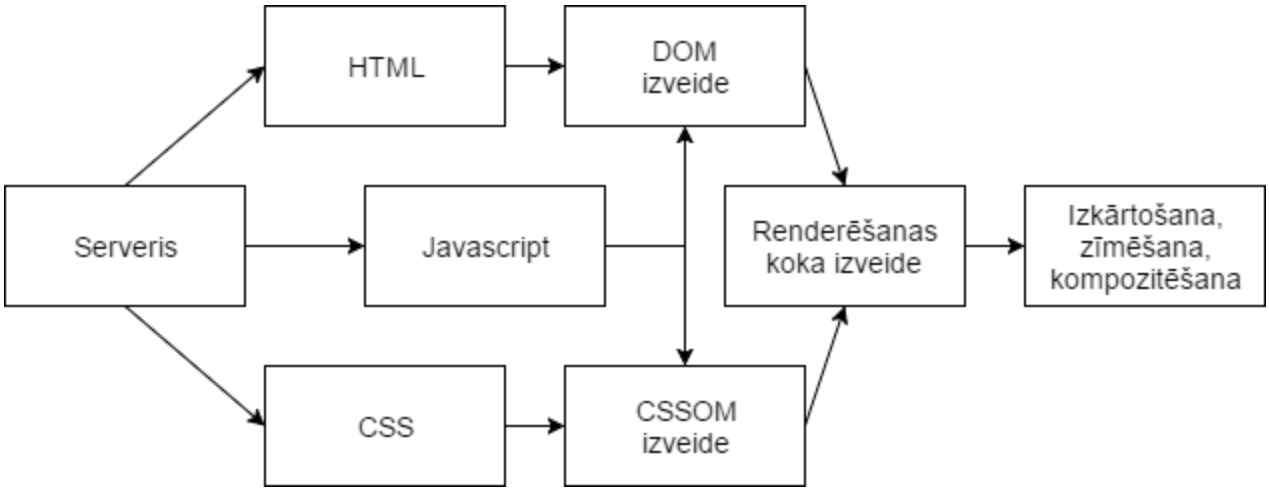
- Lietotāja saskarne. Visa vizuāli redzamā pārlūka daļa, izņemot laukumu, kurā tiek attēlots HTML dokumentu saturs. Pie saskarnes pieder, piemēram, adreses lauks, cilnes, navigācijas pogas, izvēlne u.c.
- Pārlūka dzinējs – komponente, kas nodrošina iespēju mijiedarboties ar renderēšanas dzinēju.
- Renderēšanas dzinējs – pārlūka daļa, kas tīmekļa lapu HTML dokumentu saturu pārveido par vizuālu attēlojumu uz ekrāna. Atsevišķi pārlūki, piemēram, Google Chrome, var uzturēt vairākas dzinēja instances, katru savā cilnē [2]. Piemēri: WebKit, Gecko, Blink, Presto.
- Tīmekļa apakšsistēma – tiek izmantota tīmekļa pieprasījumu izpildei.
- Javascript dzinējs – programma, kas interpretē un izpilda Javascript kodu. Piemēram, V8, SpiderMonkey, Rhino, Nitro.
- XML parseris – apakšprogramma, kas konvertē tekstu XML formātā, kurā tiek veidots DOM koks.
- Saskarnes izpildāmais modulis – nodrošina iespēju zīmēt lietotāja saskarnes elementus, neatkarīgi no platformas tipa.
- Datu glabāšanas apakšsistēma – nodrošina iespēju veikt pārlūka datu uzglabāšanu lietotāja datorā. Piemēram, var tikt uzglabāts kešatmiņas saturs, mājaslapu apmeklējumu vēsture u.c.



1.1. att. **Interneta pārlūka pamata sastāvdaļu vispārīga shēma**

1.1. Tīmekļa vietnes sākotnējais ielādes process

Lai pārlūks spētu parādīt lietotāja pieprasīto saturu, tam jāveic virkne darbību.



1.1.1. att. **Tīmekļa lapu lietotāja saskarnes ielādes posmi**

Satura atveidošanas moduļa darbība sākas ar HTML koda parsēšanu un DOM un CSSOM koka izveidi. No HTML dokumenta pārlūkprogramma izveido DOM, bet no CSS – CSSOM koku. Tomēr arī Javascript kodu mēdz izmantot, lai veiktu dinamiskas izmaiņas saskarnes vizuālajā attēlojumā [3]. Bez tam arī CSS likumi var kļūt par izmaiņu iniciatoru. Pirms pārlūks var sākt rādīt uz ekrāna tīmekļa vietnes saturu, tam jāizveido DOM un CSSOM koki. Tāpēc ir būtiski, ka HTML un CSS faili tiek lejupielādēti pēc iespējas ātrāk. Gan HTML, gan CSS failu parsēšana notiek pēc vienāda pamatprincipa: faili sastāv no baitiem, baiti tiek pārveidoti par simboliem, no simboliem veido leksēmas, no leksēmām veido koka mezglu un no mezgliem saliek DOM vai CSSOM koku:

Baiti → Simboli → Leksēmas → Mezgli → Objektu modelis (DOM vai CSSOM)

1.1.2. att. **HTML un CSS dokumentu apstrādes secība**

Pēc DOM koka izveides tiek veikti stilu likumu prioritāšu aprēķini. Šajā procesā DOM elementiem tiek piekārtoti atbilstošie CSS likumi. Informācija par stiliem tiek iegūta gan no CSS failiem, gan no stilu definīcijām HTML kodā – *style* atribūta vērtībām un no citiem HTML vizuālajiem atribūtiem [4]. Iegūtā informācija tiek izmantota, lai izveidotu renderēšanas koku (*render tree*). Renderēšanas koks satur elementus ar vizuālajiem atribūtiem. Koka struktūra definē DOM elementu secību, kādā tiem jābūt attēlotiem uz ekrāna.

Pēc renderēšanas koka izveides tiek iniciēts izkārtošanas process. Tā kā pārlūkam ir zināmi visi CSS likumi katram elementam, ir iespējams izrēķināt, cik daudz vietas aizņems elementa vizuālais attēlojums un kurā pozīcijā tas atradīsies uz ekrāna. Ir vērts atcerēties, ka

elementi savā starpā ietekmē viens otru, piemēram, vecāka elementa izmērs tiešā veidā ietekmē bērnu maksimālos izmērus.

Tagad ir iespējams sākt attēlot uz ekrāna visus renderēšanas koka elementus tādā kārtībā, kādā tie atrodas šajā kokā. Šo procesu sauc par zīmēšanu jeb rasterizēšanu. Lai uzzīmētu elementus, tiek izmantots saskarnes izpildāmais modulis. Pati zīmēšana izpaužas kā ekrāna pikseļu aizpildīšana ar krāsām. Tā tiek uzzīmēts teksts, krāsas, ēnas, attēli un saskarnes elementu vizuālās daļas. Zīmēšana mēdz notikt vairākās divdimensiju plaknēs, ko sauc par slāņiem. Viens slānis var atrasties uz cita slāņa, jo bērnu elementi var tikt pilnībā izvietoti uz saviem vecākiem. Pārlūki cenšas attēlot lietotāja saskarni pēc iespējas īsākā laikā, tāpēc, piemēram, DOM koka un renderēšanas koka izveide var sākties vēl pirms HTML koda lejupielādes vai parsēšanas pabeigšanas [4]. Tas nozīmē, ka vienu elementu HTML kods vēl tikai tiek lejupielādēts, bet paralēli jau notiek citu elementu zīmēšana.

Visbeidzot pēdējais process, pirms saskarnes attēlošana ir pilnībā pabeigta, ir kompozitēšana. Tas vajadzīgs, lai saliktu slāņus pareizajās vietās un pareizajā secībā. Kompozitēšana ietekmē elementus, kuri savā starpā pārklājas, piemēram, lai nepieļautu, ka kāds elements bez vajadzības aizsedz citu. Elementu slāņi, kas bija iegūti izkārtošanas laikā, tiek kombinēti kā gala attēlojums, ko lietotājs redzēs uz ekrāna. Kompozitēšana tipiski izpildās, izmantojot videokarti un nepatērējot procesora resursus.

Katrs no minētajiem saskarnes atveidošanas posmiem ir spējīgs izraisīt satura raustīšanos vai tīmekļa lietotnes sastingšanu, tāpēc ir svarīgi zināt, kuru posmu izpildi izsauc kāds noteikts Javascript koda fragments.

1.1.1. Izkārtošanas (layouting) posms

Tīmekļa vietnes grafiskās attēlošanas pamatā ir izkārtošanas process, kurš ietver zīmēšanas un kompozitēšanas procesu.

Ja tiek mainīta kāda stilu definīcija, kas izsauc izkārtošanas (*layouting*) proces, tad tas nozīmē, ka neizbēgami notiks arī visi nākamie saskarnes attēlošanas posmi, kas seko aiz izkārtošanas – pārzīmēšana un kompozitēšana. Ar izkārtojumu saistītas tās īpašības, kuras ietekmē elementa izmērus vai atrašanās vietu, piemēram, platums, garums, pozīcija no kreisās malas utml. Šajā gadījumā ir jāveic izmaiņas renderēšanas kokā un jāpārzīmē visi tie elementi, kuru grafiskais attēlojums tika ietekmēts pārzīmēšanas procesā. Papildus tam, ir jāpārrēķina kaimiņu elementu pozīcijas, jo tās var tikt ietekmētas un mainītas. Jāņem vērā, ka pārzīmēšanai (*reflow*) var tikt pakļauti vēl arī tie elementi, kuri atrodas blakus pārzīmējamajam elementam. Plašāks izkārtošanas procesa apraksts skatāms 1.6. nodaļā "Izkārtošanas process un tā optimizācija".

1.1.2. Zīmēšanas (painting) posms

Ja tiek mainīta kāda ar zīmēšanu (*painting*) saistīta īpašība, piemēram, fona attēls, teksta krāsa utml., kas neietekmē elementu izmērus vai izvietojumu, tad pārlūks izlaiž izkārtošanas procesu un iniciē tikai pārzīmēšanu un kompozitēšanu. Zīmēšanas algoritma apraksts atrodams 1.7. nodaļā "Zīmēšanas process".

1.1.3. Kompozitēšanas (compositing) posms

Ja mainīta tāda īpašība, kas neattiecas ne uz izkārtošanu, ne uz zīmēšanu, tad pārlūks, lai pielietotu vajadzīgās izmaiņas, izpilda vien kompozitēšanu. Ir skaidrs, ka šādā gadījumā izmaiņas saskarnes attēlojumā notiks ievērojami ātrāk, nekā, ja pirms tam vēl notiktu pārkārtošana un elementu pārzīmēšana. Izmaiņas, kuras ir saistītas tikai ar kompozitēšanu, ir īpaši kritiskas animācijām un satura plūdenai ritināšanai. CSS likumu piemēri, kas attiecas tikai uz kompozitēšanu [5,6]:

- *transform: translate* (elementa pozīcijas maiņa),
- *transform: scale* (elementa izmēra maiņa),
- *transform: rotate* (elementa pagriešana par noteiktu leņķi),
- *opacity* (elementa puscaurspīdīgums),
- *cursor,*
- *z-index,*
- *translateZ,*
- *translate3D.*

1.2. Koda sintaktiskā analīze

Koda sintaktiskā analīze jeb parsēšana ir viena no pirmajām darbībām, ko veic pārlūkprogramma, lai tā varētu grafiski parādīt pieprasīto saturu [4]. Dokumenta parsēšana nozīmē arī tā satura pārveidošanu par tādu datu struktūru, ko pārlūks var izmantot turpmāk. Rezultātā tiek radīts mezglu koks, kurš līdzinās sākotnējā dokumenta struktūrai. Šo koku sauc par sintakses koku.

Parsēšana iedalās divos procesos: leksiskā analīze un sintaktiskā analīze. Leksiskā analīze ir informācijas sadalīšana atsevišķās vienībās jeb leksēmās. Dokumenta analīzi izdara divi komponenti:

- Leksiskais analizators, kurš kodu kā ieejošo simbolu virkni sadala pieturzīmēs un vārdos jeb leksēmās,

- Sintakses analizators, kurš analizē dokumenta struktūru atbilstoši konkrētās valodas sintakses likumiem un izveido sintakses koku. Analizators ignorē vienīgi tukšumzīmes, atstarpes un rindu pārtraukumus.

Sintaktiskā analīze ir iteratīvs process. Sintakses analizators pieprasa leksiskajam analizatoram kārtējo leksēmu un pārbauda, vai tas atbilst kādam no sintakses likumiem. Ja atbilstība ir atrasta, tad leksēmai tiek izveidots attiecīgs sintakses koka mezgls un analizators pieprasa nākamo leksēmu.

Ja leksēma neatbilst nevienam likumam, tad sintakses analizators to ignorē un pārbauda nākamās leksēmas. Vēlāk analizators gan cenšas piemeklēt likumus, kuriem atbilstu visas ignorētās leksēmas. Ja tas tomēr neizdodas, tad analizators to uzskata par izņēmuma situāciju un var atgriezt kļūdas ziņojumu. Tas nozīmē, ka dokuments satur sintakses kļūdas un nevar tikt veiksmīgi apstrādāts tālāk.

Sintakses koks ne vienmēr ir uzskatāms par parsēšanas gala rezultātu. Sintaktiskā analīze var tikt izmantota, lai tulkotu ieejas dokumentu vajadzīgajā formātā, piemēram, kompilācijas procesā. Kompilators, kurš pirmkodu pārveido mašīnkodā, vispirms izveido no tā sintakses koku un tikai tad no šī koka veido dokumentu ar mašīnkodu.

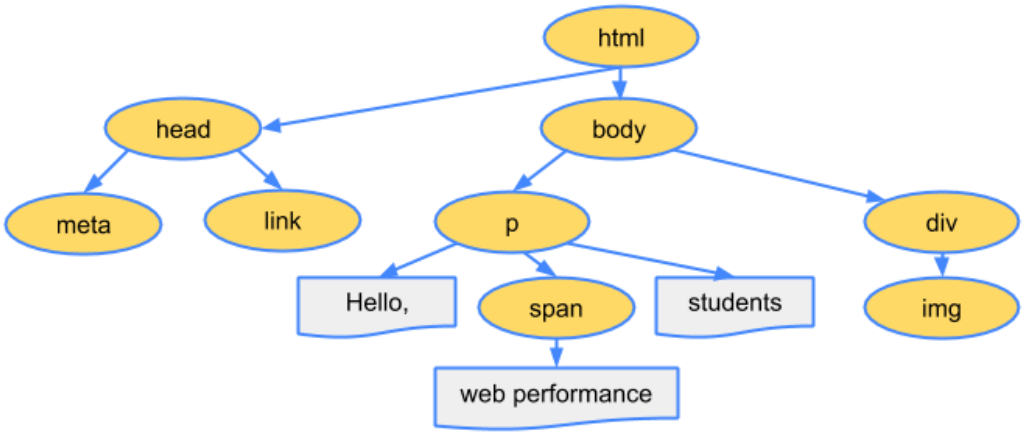
WebKit izmanto divas programmas, kas ļauj veidot sintaktiskos analizatorus: Flex (saukts arī par Lex), lai veidotu leksisko analizatoru un Bison (Yacc), lai veidotu sintakses analizatoru. Flex gadījumā nepieciešams ielādēt failu ar leksēmu definīcijām regulāro izteiksmju formā, bet Bison vajadzīgi valodas sintakses likumi BNF formā [7].

1.2.1. Objektu modeļu konstruēšana

Pārlūkprogramma konstruē objektu modeļus (DOM, CSSOM), pārveidojot HTML un CSS kodu par leksēmām, no kurām veido mezglu elementus, lai no tiem saliktu objektu modeļus. Apskatīsim HTML kodu [3]:

```
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link href="style.css" rel="stylesheet">
    <title>Critical Path</title>
  </head>
  <body>
    <p>Hello <span>web performance</span> students!</p>
    <div><img src="awesome-photo.jpg"></div>
  </body>
</html>
```

HTML parseris sadala doto kodu birkās, atribūtos un atribūtu vērtībās. Rezultātā tiek iegūts atbilstošs DOM koks [3]:



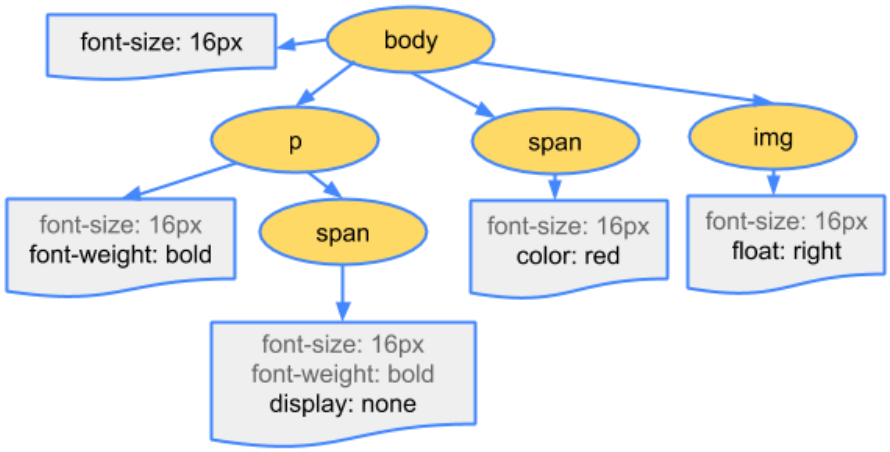
1.2.1.1. att. **DOM koka struktūras paraugs**

HTML parsēšanas rezultātā tiek radīts DOM koks, kuru pārlūks turpmāk izmanto kā tīmekļa vietnes lietotāja saskarnes struktūras attēlojumu. DOM parāda, kā savā starpā ir saistīti saskarnes elementi, bet tajā nav atrodama informācija par šo elementu ārējo izskatu.

CSS faila parsēšanas rezultāts ir CSSOM koks, kura struktūra atbilst iepriekš definētu CSS likumu un selektoru struktūrai. Piemēram, dots CSS kods:

```
body { font-size: 16px }
p { font-weight: bold }
span { color: red }
p span { display: none }
img { float: right }
```

Dotais kods satur 5 selektorus un katram no tiem ir pa vienam CSS likumam ar vienu stila deklarāciju. Ir redzams, ka ir viens selektors (`p span`), kurš satur pēcnācējus, tāpēc tādam selektoram arī CSSOM kokā parādīsies bērnu mezgls, kuram var būt definēti vēl savi stili. Rezultātā no dotā CSS koda tiek iegūts CSSOM koks [3]:

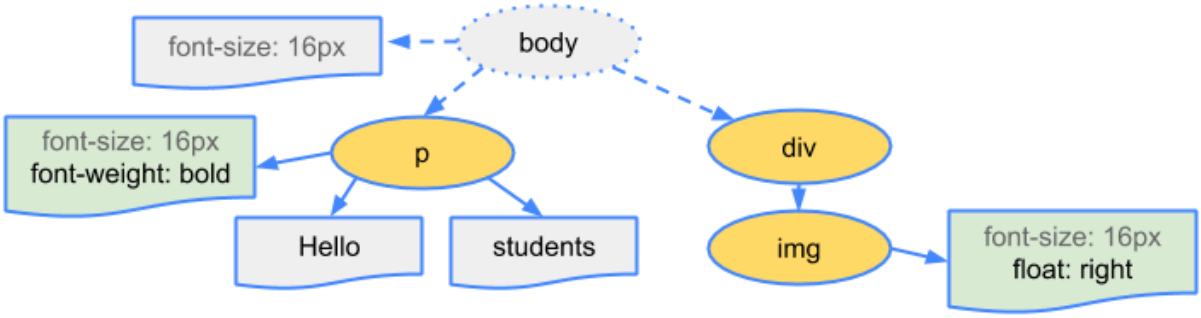


1.2.1.2. att. **CSSOM koka struktūras paraugs**

CSSOM ir kokveida struktūra, jo nosakot katra objekta stilu kopu, pārlūks sāk ar pašu vispārīgāko CSS likumu, kas ir derīgs kārtējam objektam (piemēram, ja tas ir *body* objekta bērns, tad tam piemīt arī *body* objekta stili), un tālāk rekursīvi piemēro arvien specifiskākus CSS likumus, līdz nonāk pie CSS likuma, kas tiek piemērots tikai pašreizējam objektam.

Tomēr, lai sāktu zīmēt objektus uz ekrāna, nepietiek tikai ar DOM un CSSOM struktūrām. DOM satur objektus, CSSOM satur stilus, taču nav kokveida struktūras, kura to visu apvienotu. Tāpēc vajadzīgs renderēšanas koks.

Renderēšanas koks satur tikai tos elementus, kuri ir jāuzzīmē uz ekrāna un tas tiek padots kā izkārtošanas un zīmēšanas procesu ieejas dati. Augstāk dotajiem HTML un CSS paraugiem atbilstošs renderēšanas koks dots 1.2.1.3. attēlā [3]:



1.2.1.3. att. **Renderēšanas koka struktūras paraugs**

Renderēšanas koka mezgli tiek veidoti pēc šāda algoritma:

- Sākot ar DOM koka sakni, rekursīvi tiek apstaigāts katrs tā mezgls. Vizuāli neattēlojamās birkas, piemēram, *script*, *meta* u.c., tiek ignorētas, jo tām nav vizuālas reprezentācijas. Mezgli, kuri, izmantojot CSS likumus, ir iestatīti kā neredzami, arī tiek izlaisti. Tas attiecas uz gadījumiem, kad, piemēram, objekts padarīts neredzams, izmantojot `display:none`.
- Katram redzamajam DOM mezglam tiek atrasti un pielietoti atbilstoši CSS likumi no CSSOM koka.
- Vajadzības gadījumā veic skaitļošanas aprēķinus ar stilu likumu vērtībām (piemēram, 40% platumu pārvērš no procentiem uz pikseļiem) un atgriež gatavo rezultātu kā kārtējo renderēšanas koka mezglu.

1.3. HTML koda parsēšana un izpilde

HTML koda sintakses analizatora uzdevums ir pārveidot HTML kodu par sintakses koku. Programmēšanas valodu sintakses likumus var definēt, piemēram, BNF formā. Taču sintakses analizatori nav derīgi HTML valodai [8], jo HTML nav iespējams definēt kā bezkonteksta

gramatiku (skat. 1.4. nodaļu), ar kurām strādā sintakses analizatori. HTML formālā standarta (DTD) gramatika nav bezkonteksta gramatika.

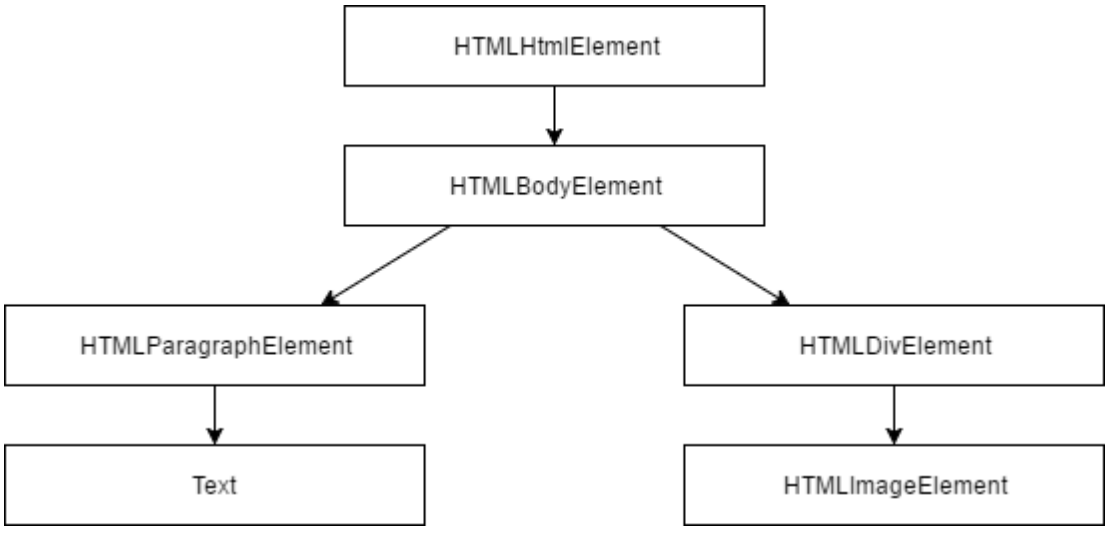
HTML valodas sintakse ir līdzīga XML valodai, kurai eksistē sintakses analizatori. Tomēr HTML standarts atšķiras no XML standarta ar to, ka tas pieļauj sintakses kļūdas HTML kodā, piemēram, var tikt izlaistas atsevišķas atverošās vai aizverošās birkas, kamēr XML sintakse ir stingri noteikta. Tādējādi var teikt, ka HTML valoda "piedod" programmētāju kļūdas, bet šī īpašība samazina iespējas definēt formālu valodas gramatiku. No tā var secināt, ka HTML nav bezkonteksta valoda un attiecīgi to nevar analizēt ne ar standarta, ne ar XML analizatoriem.

HTML dokumenta parsēšanas rezultātā tiek iegūts sintakses koks, kas sastāv no DOM elementiem un atribūtu mezgliem. DOM koka struktūra atbilst attiecīgā HTML faila struktūrai.

Piemēram, dots HTML kods:

```
<html>
  <body>
    <p>
      Hello World
    </p>
    <div> <img src="example.png"/></div>
  </body>
</html>
```

Dotajam kodam atbilstošs DOM koks izskatītos šādi:



1.3.1. att. **DOM koka paraugs dotam HTML pirmkodam**

Tīmekļa dokumentu apstrādes algoritmi balstās uz sinhrono izpildes modeli. Paredzams, ka <script> birkās esošie skripti tiks izpildīti nekavējoties, uzreiz pēc "script" birkas apstrādes. Līdz ar to dokumenta sintakses analīze tiek atlikta līdz skripta izpildes beigām. Ja ir norādīta saite uz ārēju skriptu, tad arī šāda skripta iegūšana notiek sinhroni, tāpēc tālākā koda izpilde nenotiek, kamēr nav lejupielādēts pieprasītais kods. Tomēr ir iespējams apzīmēt skriptu ar

“defer” birku, lai HTML dokumenta analīzi varētu pabeigt vēl pirms skripta izpildes [9]. HTML5 ir iespējams atzīmēt skriptu kā asinhronu, lai tas izpildītos atsevišķi [10].

Tikmēr CSS stilu tabulas jeb CSS faili tiek apstrādāti uz citas pieejas pamata. Tā kā stilu likumi neveic izmaiņas DOM koka struktūrā, nav jēgas apturēt HTML dokumenta analīzi pirms CSS parsēšanas beigām. Tomēr skripti var pieprasīt informāciju par stiliem jau dokumenta parsēšanas laikā. Ja stili vēl nav ielādēti un apstrādāti, skripts var iegūt nepatiesu informāciju. Tas attiecīgi novestu pie citām problēmām ar saskarnes korektu attēlošanu. Ja Mozilla Firefox pārlūks parsēšanas laikā konstatē CSS stilu tabulas deklarāciju, kas vēl nav ielādēta un parsēta, tad visu skriptu izpilde uz laiku tiek pārtraukta. Tikmēr WebKit gadījumā skripti pārtrauc darbu tikai tajā gadījumā, ja tie mēģina iegūt stilu īpašības, kuras varētu tikt definētas neielādētajās CSS stilu tabulās.

1.4. Koda sintakses analīze interneta pārlūkos

Jau minēts, ka HTML koda parsēšanu nevar veikt ar standarta top-down vai bottom-up parseriem. Tam ir savi iemesli:

- HTML valodā ir pieļaujamas koda kļūdas,
- Interneta pārlūkos ir iebūvēti apstrādes un kļūdu korekcijas mehānismi biežāk sastopamajām kļūdām HTML kodā,
- HTML parsēšanas process ir atgriezenisks. Parasti ieejas dokumenta saturs nemainās parsēšanas laikā, tomēr HTML valodai ir iespējami gadījumi, kad parsēšanas laikā var veidoties jauni elementi. Tāpēc sākotnējais kods beigās var izmainīties. Tas iespējams arī tad, ja HTML dokuments satur <*script*> birkas, kurās ar Javascript palīdzību tiek izveidoti jauni saskarnes elementi.

Tā kā standarta parseri nav derīgi HTML valodai, pārlūkos eksistē savi koda analizatori. Tā, piemēram, Google Chrome kā pamatu izmanto *libxml* un *libxslt* parserus. To var pārbaudīt, pārlūka adreses laukā ierakstot “*chrome://credits/*”. Savukārt Mozilla Firefox izmanto *Expat* un to iespējams redzēt, adreses laukā ierakstot “*about:license*” un atrodot tur *Expat License*.

Sintakses analīzes algoritms ir aprakstīts HTML5 specifikācijas 8. nodaļā “Parsing HTML Documents” [10]. Tajā ir divi posmi: leksiskā analīze un sintakses koka konstruēšana.

Leksiskās analīzes gaitā ieejošā simbolu virkne tiek sadalīta leksēmās. HTML valodā par leksēmām sauc atverošās un aizverošās birkas, kā arī to atribūtu nosaukumus un vērtības. Leksiskais analizators atrod leksēmu, padod to sintakses koka konstruktoram un HTML kodā pāriet pie nākamā simbola, lai meklētu nākamās leksēmas. Tā turpinās līdz ir sasniegtas HTML koda beigas.

1.4.1. HTML leksiskās analīzes algoritms

HTML leksiskās analīzes algoritma darbības rezultāts ir HTML leksēma. Šo algoritmu var raksturot kā automātu, kurā katrs stāvoklis apstrādā vienu vai vairākus ieejas simbolus, no kuriem tiek noteikts nākamais stāvoklis. Stāvoklis ir atkarīgs no leksiskās analīzes pašreizējā etapa un no sintakses koka izveides statusa. Tas nozīmē, ka viena un tā paša simbola apstrāde, atkarībā no pašreizējā stāvokļa, var novest pie atšķirīgiem turpmākajiem stāvokļiem.

1.4.2. Sintakses koka izveides algoritms

DOM koka sakne ir *Document* objekts. Tālāk, koka izveides laikā saknes elementam tiek pievienoti jauni bērnu elementi. Koka konstruktors apstrādā katru mezglu, ko ir ģenerējis leksiskais analizators. Katrai leksēmai tiek veidots savs DOM koka mezgls. Tomēr šie elementi tiek pievienoti ne tikai DOM kokam, bet arī atvērto elementu stekam, kurš kalpo nepareizi ievietoto vai neaizvērto birku vēlākai izlabošanai. Sintakses koka izveides algoritms arī ir izsakāms kā automāts ar galīgu stāvokļu skaitu, kur stāvokļus sauc par "ievietošanas veidiem" (*insertion modes*).

Insertion mode ir mainīgais, kurš nosaka DOM koka izveides primāro operāciju [10]. "Ievietošanas veids" ietekmē leksēmu apstrādes procesu. Sākumā "ievietošanas veids" ir *initial*. Parsēšanas gaitā tas var iegūt šādas vērtības:

- "*before html*", "*before head*",
- "*in head*", "*in head noscript*", "*in body*", "*in table*", "*in table text*", "*in caption*", "*in column group*", "*in table body*", "*in row*", "*in cell*", "*in select*", "*in select in table*", "*in template*", "*in frameset*"
- "*after head*", "*after body*", "*after frameset*", "*after after body*", vai "*after after frameset*"
- "*text*".

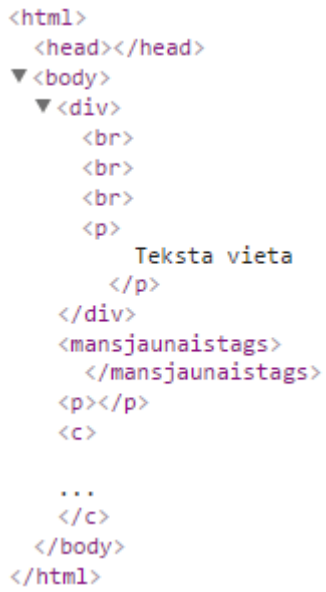
1.4.3. HTML koda kļūdu apstrāde tīmekļa pārlūkos

Pārlūkprogrammās ir iebūvēta funckionalitāte, kas nodrošina sintaktisko kļūdu korekciju, nepārtraucot darbu un neizvadot tādus kļūdas paziņojumus kā, piemēram, "Nepareiza HTML sintakse". Apskatīsim HTML kodu:

```
<html>
  <div>
  <br><br/></br>
  <p>
    Teksta vieta
  </div>
  <mansjaunaistags>
  </mansjaunaistags>
```

```
  </p>
  <C>
</html>
...
```

Šajā kodā pavisam ir 8 kļūdas, piemēram, nepareizi lietotas *br* birkas, nepareizi ievietotas atverošās un aizverošās *p* un *div* birkas, ieviestas HTML valodā neeksistējošas birkas, piemēram, *C* un ievietots teksts pēc aizverošās *html* birkas. Tomēr šāds kods darbojas un to ir iespējams izpildīt pārlūkā, nesaņemot nekādus kļūdu ziņojumus pārlūka konsolē. Dotajam kodam atbilstoša DOM struktūra, kādu to ir izveidojis Google Chrome, parādīta 1.4.3.1. attēlā.



1.4.3.1. att. **DOM koka paraugs Google Chrome pārlūkā**

Aplūkojot, piemēram, uz WebKit HTML parsera pirmkodu, ir redzams, ka lielāko daļu aizņem tieši koda kļūdu apstrāde [11]. Jāmin, ka HTML standarta formālajā specifikācijā nav atrodama informācija par HTML dokumentu kļūdu apstrādi. Šādi HTML kļūdu korekcijas mehānismi radās pakāpeniski, laika gaitā, un tie ir specifiski katrai pārlūkprogrammai. Eksistē valodas konstrukcijas, kas HTML standartā nav pieļaujamas, taču tik un tā ir atrodamas dažādās mājaslapās, tāpēc pārlūki šīs problēmsituācijas mēģina labot, ieviešot kļūdaino gadījumu apstrādi. HTML koda atbilstību standartam ir iespējams pārbaudīt W3C validatorā, https://validator.w3.org/ .

WebKit *HTMLParser.cpp* kodā atrodams komentārs: "*The parser parses tokenized input into the document, building up the document tree. If the document is well-formed, parsing it is straightforward. Unfortunately, we have to handle many HTML documents that are not well-formed, so the parser has to be tolerant about errors.*"

Piemēram, mājaslapās var sastapt gadījumus, kad *
* vietā tiek lietota *</br>* birka. Šāds birkas lietojums neatbilst HTML standartam un WebKit to labo šādā veidā:

```
// Apparently some sites use </br> instead of <br>. Be
compatible with IE and Firefox and treat this like <br>.
```

```
if (t->isCloseTag(brTag) && m_document->inCompatMode()) {
    reportError(MalformedBRError);
    t->beginTag = true;
}
```

Kļūdu apstrādes mehānisms ir iebūvēts pārlūkā un programmētājs neredz kļūdu ziņojumus. Cits gadījums ir, piemēram, "head" birkas obligāta ievietošana pirms "body" birkas:

```
// If the body does not exist yet, then the <head> should be
pushed as the current block.
if (m_head && !body) {
    pushBlock(m_head->localName(), m_head->tagPriority());
    setCurrent(m_head.get());
}
```

Pārlūka kļūdu apstrāde aizņem laiku, tāpēc, skatoties no veiktspējas viedokļa, ir nepieciešams rakstīt HTML standartam atbilstošu kodu, lai nepieļautu izņēmuma gadījumu iestāšanos, kuru apstrāde prasa papildu aparatūras resursus.

1.5. CSS parsēšana

Atšķirībā no HTML, CSS valodā izmanto bezkonteksta gramatiku, tāpēc tās parsēsanai ir derīgi standarta līdzekļi. CSS gramatika tiek uzdota kā regulārā izteiksme katrai leksēmai [12]:

```
comment   \/\*[^*]*\*+([^/*][^*]*\*+)*\/
num   [0-9]+|[0-9]*"."[0-9]+
nonascii  [\200-\377]
nmstart   [_a-z]|{nonascii}|{escape}
nmchar    [_a-z0-9-]|{nonascii}|{escape}
name    {nmchar}+
ident   {nmstart}{nmchar}*
```

CSS sintakses likumi aprakstīti BNF (Bekusa-Naura) formā. CSS selektoru formālā struktūra tiek definēta šādi:

```
ruleset
  : selector [ ',' S* selector ]*
    '{' S* declaration [ ';' S* declaration ]* '}' S*
  ;
```

Tas nozīmē, ka CSS likumi darbojas kā viens vai vairāki selektori, kas savā starpā atdalīti ar komatiem vai atstarpēm. CSS likums ir ietverts figūriekavās.

CSS likumiem ir dažādas prioritātes. Likumi ar augstāku prioritāti ir svarīgāki un tie tiks pielietoti gadījumos, kad vienam un tam pašam elementam kāda stila īpašība tika iestatīta vairākas reizes. Prioritāte ietekmē selektoru veiktspēju; selektori ar augstāku prioritāti darbojas efektīvāk [16]. CSS selektoru prioritātes, sākot no augstākās līdz zemākajai, ir:

- ID selektori ar `!important` norādi,
- Klases ar `!important` norādi,
- Elementi ar `!important` norādi,

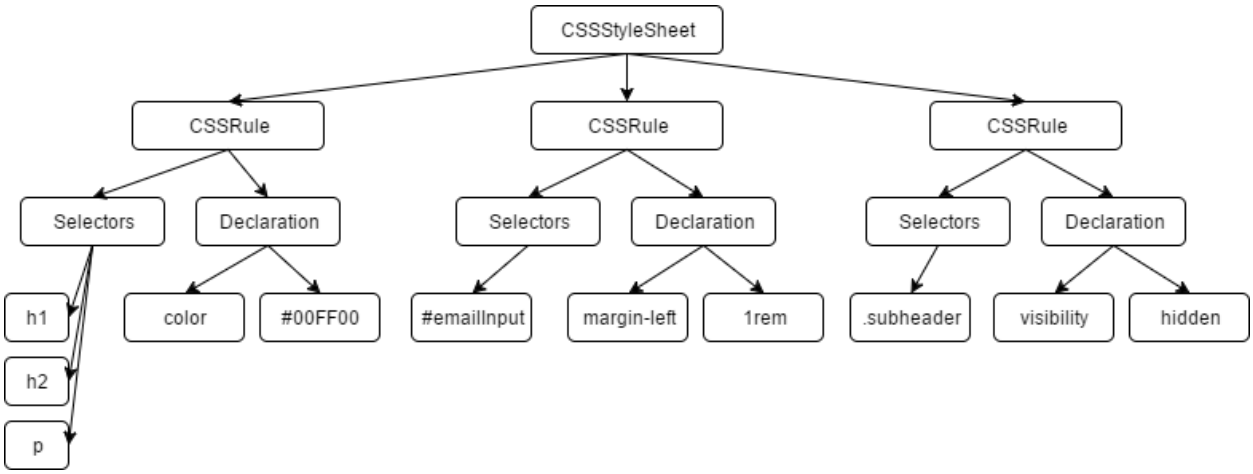
- *Inline* stili, piemēram, HTML *style* atribūts,
- ID selektori, piemēram, `#header`,
- Klašu selektori, piemēram, `.promo`,
- Elementu selektori, piemēram, `div`,
- Blakus esošie kaimiņu elementi, piemēram, `h2 + p`,
- Bērnu elementi, piemēram, `li > ul`,
- Pēcnācēji, piemēram, `ul a`,
- Universālie selektori, piemēram, `*`,
- Atribūti, piemēram, `[type="text"]`,
- Pseidoelementi, piemēram, `a:hover`.

1.5.1. WebKit CSS parseris

Jau minēts, ka WebKit savu parseru automātiskai veidošanai izmanto Flex un Bison ģeneratorus. Turpretī Mozilla Firefox izmanto Mozilla izstrādātais lejupejošais parseris. Gan vienā, gan otrā gadījumā CSS fails tiek sadalīts *StyleSheet* objektos, kuri satur CSS likumus. CSS likuma objekts satur selektoru un likuma definīciju. Piemēram, dots CSS kods:

```
h1, h2, p {
   color: #00FF00;
}
#emailInput {
   margin-left: 1rem;
}
.subheader {
   visibility: hidden;
}
```

Šim kodam atbilstošs CSS parsēšanas rezultāts būtu attēlojams kā kokveida struktūra:



1.5.1.1. att. **CSS sintakses analīzes rezultāts**

Redzams, ka beigās katrs CSS likums sadalās selektoros un deklarācijās. Ir iespējami gan vairāki selektori, gan vairākas stilu īpašības viena CSS likuma ietvaros.

1.5.2. Atveidošanas koka izveide

DOM koka veidošanas laikā pārlūks veido vēl vienu struktūru – atveidošanas koku (*render tree*). Tajā vizuālie elementi izvietojas tajā secībā, kādā tie ir jāizvada uz ekrāna. Tas ir HTML dokumenta vizuālais attēlojums. Atveidošanas koks kalpo tam, lai dokumenta satura zīmēšana notiktu pareizā secībā.

Mozilla Firefox pārlūkā atveidošanas elementu sauc par rāmi (*frame*). WebKit tiek lietots termins "renderēšanas objekts" (*render object*). Katram renderēšanas objektam ir dati par paša objekta un visu savu bērnu elementu attēlošanu. WebKit par objektu attēlošanu atbild *RenderObject* klase. WebKit kodā šī klase atrodama pēc ceļa `/WebCore/rendering/RenderObject.cpp`. Fragments no klases definīcijas:

```
class RenderObject{
  virtual void layout();
  virtual void paint(PaintInfo);
  virtual void rect repaintRect();
  Node* node; // the DOM node
  RenderStyle* style; // the computed style
  RenderLayer* containgLayer; //the containing z-index layer
}
```

Katrs atveidošanas objekts sastāv no taisnstūrveida apgabala, kas atbilst CSS kastes modeļa rāmim. Tas satur elementa ģeometriskos datus, piemēram, garumu, platumu un atrašanās vietu. Elementa tips ir atkarīgs no atribūta vērtības.

Atveidošanas koks lielā mērā līdzinās DOM kokam, taču tas nesatur grafiski neattēlojamos HTML elementus, piemēram, *head*. Bez tam, atveidošanas kokā netiek iekļauti elementi, kuriem *display* atribūta vērtība ir *none*, kamēr elementi ar vērtību *hidden*, šajā kokā ir iekļauti.

Eksistē tādi DOM elementi, kuriem atbilst vairāki vizuālie objekti vienlaikus. Parasti tie ir elementi ar sarežģītu struktūru un tos nevar aprakstīt tikai ar vienu taisnstūri. Piemēram, HTML *select* elementam atbilst 3 vizuālie objekti: viens paredzēts atveidošanas apgabalam, viens domāts sarakstam, ko var atvērt, nospiežot uz pirmā objekta, un trešais domāts pogai. Bez tam, ja teksts neievietojas vienā rindā un sadalās fragmentos, tad jaunas rindas tiek definētas kā neatkarīgi attēlošanas objekti. Šis process ir iespējams arī, mainot *select* elementa izmēru.

Mozilla Firefox vizuālā attēlošana izpaužas kā DOM izmaiņu pārtveršana. Rāmju izveidošanu veic *FrameConstructor* konstruktors, kurš pirms rāmja izveides nosaka stilus.

WebKit "attēlošanas objekta" un stila noteikšanas procesu sauc par pievienošanu (*attachment*). Katram DOM mezglam ir *attach* metode. Pievienošana notiek sinhroni; pievienojot DOM kokam jaunu mezglu, tiek izsaukta *attach* metode.

Html un *body* birku apstrādes rezultātā izveidojas atveidošanas koka saknes objekts. CSS specifikācijā to sauc par konteineru – augšējā līmeņa bloku, kurš satur visus pārējos blokus. Tā izmēri veido pārlūka loga daļu, kurā var rādīt vēlamo saturu. Firefox gadījumā to sauc par *ViewPortFrame*, bet WebKit gadījumā – *RenderView* (kods: `/WebCore/rendering/RenderView.cpp`). Šis ir atveidošanas objekts, uz kuru norāda dokuments. Pārējais koks tiek veidots, ievietojot tajā DOM mezglus.

1.5.3. CSS stilu skaitļošana

Lai pilnībā izveidotu atveidošanas koku, nepieciešams vispirms izskaitļot katra objekta vizuālās īpašības. Stilus nolasa no CSS failiem, no HTML *style* elementa, vai no vizuālo atribūtu vērtībām HTML kodā, piemēram, *bgcolor*. Taču tādus atribūtus vēl tulko par CSS likumiem un šis process prasa papildu laiku. Ar stilu skaitļošanu ir saistīti vairāki sarežģījumi:

- Stilu dati var būt visai apjomīgi un saturēt daudz īpašību, kas noved pie palielināta atmiņas patēriņa.
- Katram elementam atbilstošu likumu meklēšana var palēnināt attēlošanu, ja kods nav optimizēts. Piemēram, ja katram elementam ir jāatrod visi vajadzīgie stili pēc kārtas, tad tas ievērojami pasliktina veiktspēju, jo atsevišķiem selektoriem var būt sarežģīta struktūra.

Sarežģīta selektora piemērs:

```
div div div div div {
    …
}
```

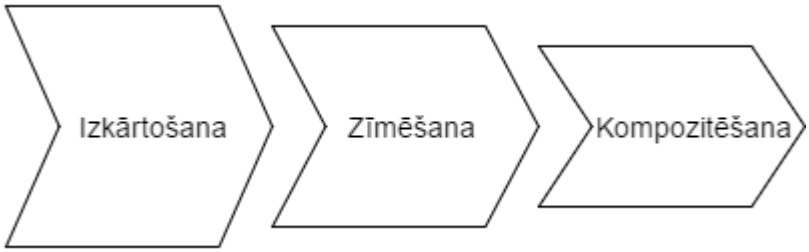
Šajā gadījumā CSS likumu vajag piemērot <*div*> elementam, kurš pats ir četru citu *div* elementu pēctecis. Pieņemsim, ka mums jāpārbauda, vai noteiktais likums ir derīgs dotajam <*div*> elementam. Mēs izvēlamies kokā noteiktu pārbaužu secību. Var izrādīties, ka mēs būsim izstaigājuši gandrīz visu koku un beigās pamanīsim ka ir tikai trīs <*div*> elementi un, attiecīgi, likums nav piemērojams. Tad jāizmēģina cita secība, un tā tālāk.

1.6. Izkārtošanas process un tā optimizācija

Kad jaunizveidotais objekts tiek iekļauts DOM kokā, tam vēl nav ne izmēra, ne atrašanās vietas grafiskajā attēlojumā. Dimensiju aprēķināšana notiek izkārtošanas (*layouting* vai *reflow*). procesa rezultātā.

Grafiskās attēlošanas procesi notiek secīgi. Izkārtošanas process ir virsprocess zīmēšanas procesam, kas savukārt ir virsprocess kompozitēšanai. Skatoties 1.6.1. attēlā, procesi izpildās tikai un vienīgi virzienā no kreisās uz labo pusi. Tas nozīmē, ka, piemēram, ja tiek izsaukts

izkārtošanas process, izpildīsies arī zīmēšana un kompozitēšana, taču, ja kādas stila īpašības maiņa izsauc zīmēšanas procesu, tad izkārtošana nenotiek.



1.6.1.att. **Izkārtošanas procesa un turpmāko procesu secības shēma**

HTML valodā izmanto plūsmas veida izkārtošanu, tas ir, vairumā gadījumu visus ģeometriskos datus var izskaitļot vienā reizē. Elementi, kas plūsmā ir nākamie, neietekmē jau apstrādāto elementu īpašības, tāpēc izkārtošanu var izpildīt no kreisās uz labo pusi un no augšas uz leju. Tiesa gan, pastāv izņēmumi, piemēram, tabulu izkārtošanai var būt vajadzīgs vairāk nekā viens cikls [13].

Koordinātas izskaitļo, balstoties uz pārlūkprogrammas rāmi (*root frame*). Tiek izmantotas koordinātas no augšas un no kreisās puses.

Izkārtošana notiek vairākos ciklos. Tā sākas ar saknes atveidošanas elementu, kurš atbilst HTML dokumenta <*html*> birkai. Tālāk tiek apstrādāta rāmju hierarhija vai atsevišķas tās daļas, un tiem atveidošanas objektiem, kuriem nepieciešams, tiek izskaitļota arī ģeometriskā informācija. Saknes atveidošanas objekta koordinātas ir (0; 0) un tā izmēri atbilst pārlūkprogrammas loga satura redzamās daļas izmēram. Jebkurš atveidošanas objekts var vajadzības gadījumā izsaukt *layout* metodi saviem bērnu elementiem.

Lai nebūtu jāveic izkārtošana pēc katras izmaiņas, pārlūki izmanto tā saucamo "netīro bitu" sistēmu. Mainītais atveidošanas objekts un tā bērni tiek atzīmēti kā "netīri", t.i., tādi, kuriem vajadzīga atkārtota izkārtošana (*reflow*). Tiek izmantoti 2 karogi: *dirty* un *children are dirty*. Karogs *children are dirty* nozīmē, ka pārkārtošana vajadzīga nevis pašam objektam, bet vismaz vienam no tā bērniem.

Ja izkārtošana tiek izpildīta visam atveidošanas kokam, tad tādu izkārtošanu sauc par globālu. To var izraisīt 2 notikumi:

- Globālas izmaiņas tādos stilos, kuri tiek izmantoti visus objektos, piemēram, fonta lieluma maiņa,
- Pārlūka loga izmēra maiņa.

Ja tiek pārkārtoti tikai "netīrie" elementi, tad tādu pārkārtošanu sauc par inkrementālu. Inkrementāla izkārtošana izpildās asinhroni un tā sākas pēc "netīro" atveidošanas objektu noteikšanas. Mozilla Firefox inkrementālās izkārtošanas komandas tiek ievietotas rindas

struktūrā, bet pēc tam plānotājs izpilda tās visas uzreiz. Arī WebKit atliek inkrementālās izkārtošanas izpildi uz vēlāku laiku, lai apstrādātu visu koku vienā ciklā un pārkārtotu visus "netīros" atveidošanas objektus.

Skripti, kuri pieprasa datus par stiliem, piemēram, offsetHeight, var novest pie inkrementālās izkārtošanas sinhronās izpildes. Dažreiz izkārtošana izpildās atgriezeniskā (*callback*) izsaukumā pēc pirmreizējās izkārtošanas, jo mainās atsevišķu atribūtu vērtības, piemēram, notiek lapas ritināšanas pozīcijas maiņa.

Ja izkārtošanu izraisīja *resize* notikums vai attēlošanas objekta atrašanās vietas (bet ne izmēra) maiņa, tad objekta izmēri tiek nolasīti no kešatmiņas un netiek pārrēķināti no jauna. Ja mainās tikai daļa no atveidošanas koka, tad visa koka pārkārtošana nav vajadzīga. Tā notiek, ja izmaiņas ir lokālas un tās neietekmē apkārtējos objektus, piemēram, teksta laukā ievadot tekstu. Citos gadījumos katra simbola ievadīšana izraisa visa koka pārkārtošanu.

Ja izkārtošanas procesā atveidošanas objektam ir konstatēts, ka vajadzīgs rindas pārnesums, izkārtošana tiek pārtraukta un vecāka elementam tiek padots rindas pārneses pieprasījums. Vecāka elements izveido papildu atveidošanas objektus un izpilda to izkārtošanu.

1.6.1. Izkārtošanas procesa algoritma apraksts

Izkārtošanas process notiek pēc šāda algoritma:

1. Bērnu elementu dimensijas tiek rēķinātas, balstoties uz vecākobjektu dimensiju informāciju.
2. Vecāks objekts apstrādā savus bērnu elementus:
    1. Nosaka bērna attēlošanas objekta pozīciju uz ekrāna (iestata tam x un y koordinātas);
    2. Izsauc bērna elementa izkārtošanas procesu, ja vien tas ir atzīmēts kā "netīrs" vai ja notiek globāla pārkārtošana utml. Rezultātā tiek izskaitļots tā augstums.
3. Zinot visu bērnu elementu un to atstarpju (*margin, padding*) summāro augstumu, tiek izskaitļots pašreizējā attēlošanas objekta augstums. Tas ir vajadzīgs arī šī objekta vecākam.
4. Iestata objekta "netīros bitus" kā *false*, t.i., objekts vairs nav "netīrs".

Mozilla Firefox pārlūkā kā izkārtošanas parametrs tiek izmantots *nsHTMLReflowState* objekts. Tas nosaka, piemēram, vecāka elementa platumu. Izkārtošanas rezultātā Firefox pārlūkā izveidojas *nsHTMLReflowMetrics* objekts, kurš satur atveidošanas objekta augstumu un citu ģeometrisko informāciju.

1.6.2. Objekta platuma aprēķināšana

Atveidošanas objekta platums tiek aprēķināts, balstoties uz dotā objekta konteinera platumu, atveidošanas objekta *width* vērtību un atstarpju (*margin, padding*) izmēru. Apskatīsim, kā notiek elementa platuma aprēķināšana:

```
<div style="width:30%"/>
```

WebKit gadījumā platuma aprēķināšanu veiks RenderBox klases *calcWidth* metode:

- Konteinera platums ir lielākā no *availableWidth* vērtībām un 0. *availableWidth* vērtība ir vienāda ar *contentWidth* vērtību, kuru aprēķina pēc formulas: `clientWidth() - paddingLeft() - paddingRight()`. *ClientWidth* un *clientHeight* vērtības atbilst objekta iekšējiem izmēriem, izņemot robežu (*border*) un ritināšanas joslu (*scrollbar*).
- Elementu platumu nosaka pēc *style* objekta *width* atribūta vērtības. Tā absolūto vērtību aprēķina pēc procentuālās daļas no konteinera platuma.
- Tiek pieliktas horizontālās atstarpes un robežas (*borders, padding*).

Ja aprēķinātais platums pārsniedz maksimālo pieļaujamo objekta platumu, tad tiek izmantots maksimālais platums, bet ja tas ir mazāks par minimālo, tad izmanto minimālo atļauto platuma vērtību, kuru var nolasīt, piemēram, no *min-width* atribūta. Šie dati tiek uz laiku pieglabāti atmiņā gadījumam, ja būs vajadzīga izkārtošana bez platuma maiņas.

1.7. Zīmēšanas process

Katram atveidošanas objektam, kuru nepieciešams zīmēt vai pārzīmēt, tiek izsaukta *paint* metode un objekts tiek parādīts uz ekrāna. Zīmēšanai tiek izmantots lietotāja saskarnes izpildāmais modulis.

Globālās pārzīmēšanas laikā tiek pārzīmēts viss atveidošanas koks, bet inkrementālās pārzīmēšanas laikā – tikai atsevišķi atveidošanas objekti, kuri neietekmē pārējās koka daļas. Izmainītais atveidošanas objekts atzīmē savu taisnstūri kā spēkā neesošu. Pārlūkprogramma to uzskata par "netīru" apgabalu un izsauc *paint* metodi. Šajā laikā apvienojas visi vajadzīgie apgabali, lai pārzīmēšanu varētu veikt vienā reizē visiem objektiem. Google Chrome pārlūkā pārzīmēšanas process ir sarežģītāks, jo atveidošanas objekts atrodas ārpus galvenā izpildes procesa; Chrome pats zināmā mērā imitē operētājsistēmas darbību, kur katra cilne darbojas savā procesā [2]. Vizuālās attēlošanas komponente pārtver šos notikumus un deleģē ziņojumu saknes atveidošanas objektam. Visi atveidošanas koka objekti pēc kārtas tiek pārbaudīti, līdz tiek atrasts vajadzīgais objekts. Kad meklējamais objekts ir atrasts, tas tiek pārzīmēts kopā ar visiem saviem bērnu elementiem.

Zīmēšanas secību nosaka CSS specifikācija. Faktiski tā atbilst elementu ievietošanas kārtībai steku kontekstā [14]. Zīmēšanas secībai ir nozīme, jo steki tiek zīmēti no beigām uz sākumu. Bloka elementu pievienošana stekā notiek pēc šādas kārtības:

1. Fona krāsa,
2. Fona attēls,
3. Robeža (*border*),
4. Bērnu objekti,
5. Ārējās robežas (*outline*).

Mozilla Firefox vidē uz attēlošanas koka analīzes pamata tiek izveidots zīmējamo taisnstūru attēlošanas saraksts. Saraksts satur doto taisnstūru attēlošanas objektus, kas izvietoti nepieciešamajā secībā (vispirms fons, tad robeža utt.). Pateicoties tam, atkārtotai fona, fona attēlu, robežu u.c. pārzīmēšanai pietiek izstaigāt visu atveidošanas koku tikai vienu reizi. Firefox šis process ir optimizēts tā, ka elementi, kuri būs paslēpti, piemēram, zem necaurspīdīgiem elementiem, netiek pievienoti.

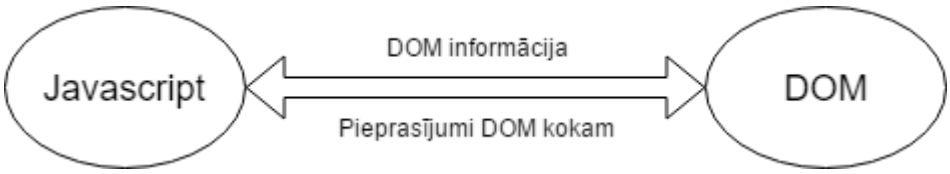
Salīdzinājumā ar Mozilla Firefox lietoto pārlūka dzinēju, WebKit tas pats process notiek citādi. WebKit gadījumā, pirms atkārtotas pārzīmēšanas, pārzīmējamā objekta taisnstūris tiek saglabāts kā rastru attēls un pēc tam tiek pārzīmētas tikai tās vietas, kur ir vizuālas atšķirības starp veco un jauno taisnstūri.

Dinamisko izmaiņu gadījumā pārlūki vienmēr cenšas izdarīt pēc iespējas mazāk darba. Piemēram, mainot kāda objekta krāsu, pārējie elementi netiek pārzīmēti no jauna. Mainot elementa pozīciju, notiek šī elementa, bērnu elementu un, iespējams, arī tā paša līmeņa kaimiņu izkārtošana un pārzīmēšana. Pievienojot DOM elementu, tiek izpildīta tā izkārtošana un uzzīmēšana. Nopietnas izmaiņas, kā, piemēram, *html* elementa fonta izmēra maiņa, izraisa visas ar izkārtošanu un zīmēšanu saistītās atmiņas attīrīšanu un atkārtotu visa attēlošanas koka izkārtošanu un pārzīmēšanu.

2. TĪMEKĻA VIETŅU VEIKTSPĒJAS OPTIMIZĀCIJAS PIEEJAS

Interneta pārlūku izstrādātāji pieņem, ka jebkādas DOM izmaiņas varētu potenciāli būt vizuālas, tas ir, pēc izmaiņām mainīsies ekrāna saturs. Līdz ar to, pēc DOM manipulāciju veikšanas, pārlūks no jauna veic objektu izkārtošanu, pārzīmēšanu un kompozitēšanu.

Neskatoties uz to, ka DOM ir no valodas neatkarīga lietojumprogrammatūras saskarne, kas paredzēta galvenokārt darbam ar HTML un XML dokumentiem, pārlūki pamatā strādā ar Javascript kodu [7]. Pārlūku DOM un Javascript implementācijas ir neatkarīgas viena no otras un atdalītas savā starpā. Piemēram, Google Chrome mājaslapu renderēšanai izmanto WebKit, bet kā Javascript dzinējs tiek izmantots V8. Ja runā par DOM un Javascript savstarpēju mijiedarbību, var iztēloties, ka abas puses ir savienotas ar tiltu, kuram ir ierobežota caurlaidība. Tāpēc no veiktspējas viedokļa, var ieteikt pēc iespējas vairāk strādāt ar Javascript un mazāk vērsties pie DOM, jo pat tikai piekļūšana DOM elementiem prasa laiku.



2.1. att. **Javascript un DOM koka mijiedarbības ilustrācija**

Skatoties no interneta pārlūku puses, ir jēga atdalīt DOM no Javascript, jo tīmekļa lietotņu Javascript kodam var nebūt nepieciešamības kaut vienu reizi piekļūt DOM kokam [15]. Turklāt citas valodas, piemēram VBScript, arī var strādāt ar to pašu DOM koku.

Apskatīsim Javascript koda fragmentu:

```
var testDiv = {innerHTML: ''};
for (var count = 0; count < 100000; count++) {
 testDiv.innerHTML = 'DOM: ' + count.toString();
}
```

Šajā piemērā objektam *testDiv* tiek 100 000 reizes pēc kārtas mainīta *innerHTML* īpašība. Tagad aplūkosim citu fragmentu:

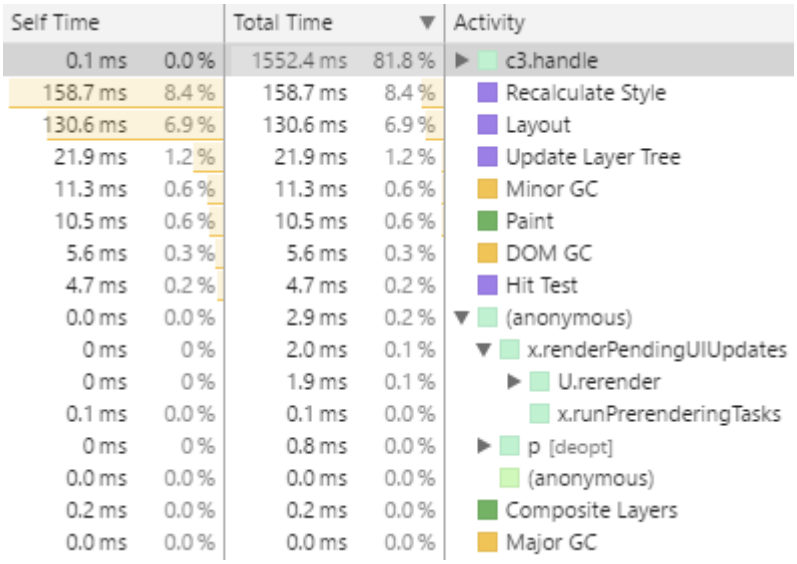
```
for (var count = 0; count < 100000; count++) {
 document.getElementById('testDiv').innerHTML = 'DOM: ' +
count.toString();
}
```

Šis piemērs izskatās līdzīgs iepriekš dotajam, proti, atkal 100 000 reizes tiek mainīta *innerHTML* īpašības vērtība. Varētu domāt, ka šis kods neatšķiras no iepriekšējā fragmenta un abu piemēru izpildes laiks būs aptuveni vienāds. Tomēr otrajā piemērā katrā cikla iterācijā divas reizes pēc kārtas notiek piekļūšana DOM kokam – vienu reizi, lai nolasītu elementa *innerHTML* īpašības vērtību, bet otru reizi, lai mainītu šo vērtību. Rezultātā cikls veic 200 000 darbības ar

DOM koku. Atšķirība parādās tajā, ka, Javascript kodā mainot kādu no DOM elementa īpašībām, piemēram, `testDiv.innerHTML = 'Sveika, pasaule!'`, ir darīšana nevis ar mainīgo, kurā tiek saglabāta vērtība, bet ar īpašiem objektiem, kuri tālāk izmaina pārlūka stāvokli un izsauc veselu virkni citu darbību, kuras dotajā Javascript piemērā nav redzamas. Līdz ar to šādas `testDiv.innerHTML` izmaiņas izraisa negatīvas sekas. Neskaitot objekta īpašības vērtības maiņu, mūsu apskatītajā piemērā notiek vēl šādas darbības:

- Simbolu virkne 'Sveika, pasaule!' tiek parsēta kā HTML kods,
- Ja pārlūkam eksistē paplašinājumi, tad pārlūks tiem pieprasa atļauju veikt tālākās DOM manipulācijas,
- Tiek iznīcināti visi *testDiv* bērnu mezgli,
- *TestDiv* bērnu mezgli tiek izveidoti no jauna,
- Tiek pārrēķināti *testDiv* stili,
- Tiek pārrēķināti objekta fiziskie izmēri un tā atrašanās vieta,
- Pārlūka paplašinājumiem tiek padots ziņojums par veiktajām izmaiņām,
- Tiek atjauninātas Javascript mainīgo vērtības, kuras ir kā saikne ar DOM koka mezgliem.

Minētās darbības tiek izpildītas, izmantojot API, kas savieno Javascript ar DOM. Mūsdienās Javascript kods tiek interpretēts citā valodā vai kompilēts mašīnkodā, kas attiecīgi var paātrināt Javascript izpildi, taču attiecībā uz DOM API, nekas tāds nav iespējams.



2.2. att. **Google Chrome darbības ilustrācija pēc DOM izmaiņu veikšanas**

2.1. Daudzkārtēja piekļūšana DOM kokam

Kad tiek izpildīts skripts, kura darbība izraisa atkārtotu elementu izkārtošanu, pārlūki cenšas šīs darbības uz laiku atlikt un izpildīt vēlāk visas reizē [7]. Tas notiek tāpēc, ka elementu izkārtošana ir laikietilpīgs process. Atliktās darbības tiek saglabātas rindas veida struktūrā. Tomēr eksistē darbības, kuru izpildes laikā nepieciešams vispirms atbrīvot šo atlikto izkārtošanas darbību rindu. Izkārtošanas rinda tiks izpildīta un attīrīta, ja pārlūks saņems pieprasījumu saņemt aktuālos saskarnes elementu izmēru datus vai citu ar izkārtošanu saistītu informāciju. Ir vairākas Javascript funkcijas, kuras izsaucot, vispirms izpildīsies visas atliktās izkārtošanas darbības (skat. 2. pielikumu):

- offsetTop, offsetLeft, offsetWidth, offsetHeight,
- scrollTop, scrollLeft, scrollWidth, scrollHeight,
- clientTop, clientLeft, clientWidth, clientHeight,
- getComputedStyle(),
- WebKit: scrollBy(), scrollTo(), scrollX, scrollY.

Izkārtojuma informācijai, ko atgriež šīs īpašības un funkcijas, ir jābūt aktuālai, tāpēc pārlūkam vispirms jāizpilda atliktās izmaiņas, kuras glabājas izkārtošanas rindā un jāveic attiecīgās izkārtošanas darbības. No tā var secināt, ka laikā, kad notiek stilu mainīšana vai citas darbības ar DOM koku, vēlams neizmantot nevienu no minētajām īpašībām, jo to izmantošana izraisa izkārtošanas rindas attīrīšanu pat tad, ja tiks pieprasīti tāda izkārtojumu dati, kas iepriekš netika mainīti.

Ja tomēr ir nepieciešamas vairākas izmaiņas DOM kokā, ir iespējams samazināt izkārtošanas un pārzīmēšanas darbību skaitu. To var panākt, izpildot attiecīgās izmaiņas ārpus paša DOM koka. Eksistē 3 veidi, kā to izdarīt:

- Izmantojot `display:none`, paslēpt elementu, veikt visas vajadzīgās izmaiņas un beigās atzīmēt elementu kā redzamu,
- Izmantojot dokumenta fragmentu, kurā var izveidot savu apakškoku un beigās ievietot to vajadzīgajā vietā DOM kokā,
- Kopēt vēlamo elementu, rediģēt kopiju un beigās aizstāt oriģinālo DOM koka mezglu ar šo pārveidoto kopiju.

Efektīvs veids, kā izvairīties no liekajām pārzīmēšanas darbībām, ir dokumenta fragmenta izmantošana. Tādā veidā ir iespējams panākt, ka Javascript kods tikai vienu reizi piekļūst DOM kokam un izraisa izkārtošanas procesu tikai vienu reizi.

2.2. Tīmekļa vietnes dinamiskās attēlošanas veiktspējas optimizācijas iespējas

Izstrādājot tīmekļa lietotnes, var rasties vajadzība piekļūt DOM koka saturam un veikt izmaiņas tajā. Tas ir izdarāms ar dažādiem paņēmieniem. Tomēr to darbība un līdz ar to arī veiktspēja, ir atšķirīga.

2.2.1. Jaunu elementu pievienošana

Eksistē vairāki veidi, kā pievienot vienu vai vairākus jaunus DOM elementus. Apskatīsim katru no tiem.

1) HTML koda papildināšana innerHtml īpašībai

Pievienot DOM elementus var, vecāka objekta `innerHtml` esošajam HTML kodam galā pierakstot papildu kodu. Piemēram:

```
for (var i = 0; i < iterations; i++) {
 div.innerHTML += '<li>List item ' + i + '</li>';
}
```

Šajā gadījumā iepriekš definētam *div* elementam tiks pievienoti *li* elementi, papildinot esošo `innerHtml` īpašības vērtību ar vēlamo HTML kodu. Katrā cikla iterācijā šis kods divas reizes piekļūst DOM kokam – vienreiz, lai nolasītu `div.innerHtml` vērtību, bet otrreiz, lai to izmainītu. Katrā cikla iterācijā vajadzētu notikt izkārtošanas un līdz ar to arī pārzīmēšanas procesam, tomēr interneta pārlūki to optimizē, atliekot pārzīmēšanu uz vēlāku laiku. Faktiski izkārtošana notiek tikai pēc cikla izpildes, ja vien cikls nesatur tādas komandas, kuras izsauc pārzīmēšanas procesu katrā cikla iterācijā. Piespiedu izkārtošana notiks, nolasot kaut vienu tādu stila īpašību vai izpildot Javascript funkciju, kura automātiski izsauc izkārtošanu. Piemēram:

```
for (var i = 0; i < iterations; i++) {
 div.innerHTML += '<li>List item ' + i + '</li>';
 var w = div.offsetWidth;
}
```

Šis kods atšķiras no iepriekšējā tikai ar to, ka katrā cikla iterācijā tiek nolasīta `div.offsetWidth` vērtība. Pat neskatoties uz to, ka šī vērtība netiek izmantota, ar to ir pietiekami, lai tiktu izsaukts izkārtošanas process. Tas saistīts ar to, ka mainot `innerHtml` vērtību, dotajā piemērā mainās *div* elementa bērnu skaits, kas savukārt nozīmē, ka var mainīties šī elementa izmēri, tai skaitā platums. Tāpēc, lai katrā iterācijā atgrieztu aktuālo platumu, tas ir jāpārrēķina un faktiski ir jāizpilda *li* elementa ievietošana tādā veidā, kā tas ir definēts kodā. Pat, ja pirms `offsetWidth` nolasīšanas, *div* elementu skaits nebūtu mainījies, šīs īpašības

nolasīšana jebkurā gadījumā izraisa stilu pārrēķināšanu. Vēl jāpiemin, ka jebkura `innerHtml` izmaiņa automātiski izraisa visu vecāka elementa bērnu iznīcināšanu un izveidošanu no jauna, tai skaitā arī tad, ja bija vajadzīgs tikai pievienot jaunu bērnu elementu. Iznīcinot bērnus, tiek pazaudēti tiem piesaistītie notikumi, tāpēc tos vēlāk nāksies pielikt no jauna. Tāpēc, lai saglabātu visus esošos bērnu elementus ar tiem piesaistītajiem notikumiem, ieteicams izmantot specifiskās DOM funkcijas, piemēram, `appendChild`.

Runājot par algoritmisko sarežģītību, nav būtiski, vai pievienojot jaunus elementus notiek vai nenotiek piespiedu izkārtošana. Izkārtošanas process tikai paildzina koda izpildi. Jebkurā gadījumā pie katras `innerHtml` izmaiņas tiks zaudēti un atjaunoti visi tajā iepriekš definētie elementi, tāpēc jo vairāk elementu šī īpašība satur, jo garāks ir izpildes laiks.

2) Specifiskās DOM funkcijas

Cits veids, kā pievienot jaunus objektus, ir `document.createElement` un `appendChild` metožu izmantošana. Šis paņēmiens atšķiras ar to, ka vairs nav jānolasa `innerHtml` un jāparsē jaunais HTML kods, kurš tiek pievienots šai īpašībai. Pievienojot jaunu elementu, netiek iznīcināti esošie. Koda piemērs:

```
for (var i = 0; i < iterations; i++) {
 var el = document.createElement('li');
 el.innerText = 'List item ' + i;
 div.appendChild(el);
}
```

Šis kods izdara to pašu, ko iepriekš dotais piemērs ar `innerHtml`. Tomēr tas izpildās ātrāk, jo tiek izdarīts tieši tas, kas ir rakstīts kodā – katrā cikla iterācijā tiek pievienots jauns elements, pārējiem bērnu elementiem paliekot neskartiem. Sarežģītības kārta vērtējama kā O(n).

3) DOM fragmentu izmantošana

Lai veiktu pēc iespējas mazāku izmaiņu skaitu DOM kokā, var izmantot DOM fragmentus. Tādā gadījumā ir iespējams elementu pievienošanu izpildīt fragmentā un beigās šo fragmentu pievienot DOM kokam vēlamajā vietā. Tādā veidā DOM struktūra tiks mainīta tikai vienu reizi. Vēl viena DOM fragmentu priekšrocība ir iespēja nolasīt tādas stilu īpašības, kuras izraisa izkārtošanu. Šajā gadījumā izkārtošana nenotiks, jo netiek skarts reālais DOM koks. Tas nozīmē, ka attiecīgo īpašību izmantošana gandrīz neatstāj iespaidu uz koda izpildes laiku.

2.3. Veiktspējas analīzes un optimizācijas rīki

Google Chrome pārlūkā ir iespējams veikt mērījumus un testēt pārzīmēšanas procesus. To iespējams darīt, izstrādātāju rīku izvēlnē izvēloties *More Tools* → *Rendering*. Parādās jauna cilne, kurā atzīmējot Paint Flashing, tiek rādīti ekrāna laukumi, kuros notiek pārzīmēšanas process. Šie ekrāna apgabali tiek iezīmēti zaļā krāsā, tā atvieglojot iespēju pārbaudīt konkrētas tīmekļa vietnes efektivitāti attiecībā uz DOM manipulācijām. 2.2.1. attēlā parādīts piemērs, kad uzbraucot ar peli virs kādas tabulas rindas, šī rinda tiek pārzīmēta:

Description	HierarchyLevel
> 1	0
> 2	0
> 3	0
> 1.1	1

2.2.1. att. **Pārzīmēšanai pakļauto ekrāna apgabalu attēlošana Google Chrome**

Arī Mozilla Firefox ir iespējams redzēt pārzīmējamos ekrāna apgabalus. To var izdarīt, atverot izstrādātāju rīkus un nospiežot "Toolbox Options". Tālāk pie *Available Toolbox Options* jāatzīmē "Highlight painted area". Lai pārzīmējamos apgabalus varētu sākt redzēt, starp izstrādātāju rīku rīkjoslas pogām jāatrod un jānospiež "Highlight painted area" ikona. Lai vairs nerādītu pārzīmējamos apgabalus, šī ikona jānospiež vēlreiz.

3. TĪMEKĻA LIETOTŅU VEIKTSPĒJAS PROBLĒMGADĪJUMU ANALĪZE

Darba ietvaros tika veikti praktiski mēģinājumi pielietot iegūtās zināšanas reālu tīmekļa lietotņu, piemēram, *SAP Screen Personas* un *SAP Pricing*, veiktspējas uzlabošanai.

Darba vide:

Operētājsistēma: Windows Server 2012 R2, 64 bitu versija

Procesors: Intel Xeon CPU E7-4830 v3 2.10 GHz (4 kodoli)

RAM: 8 GB

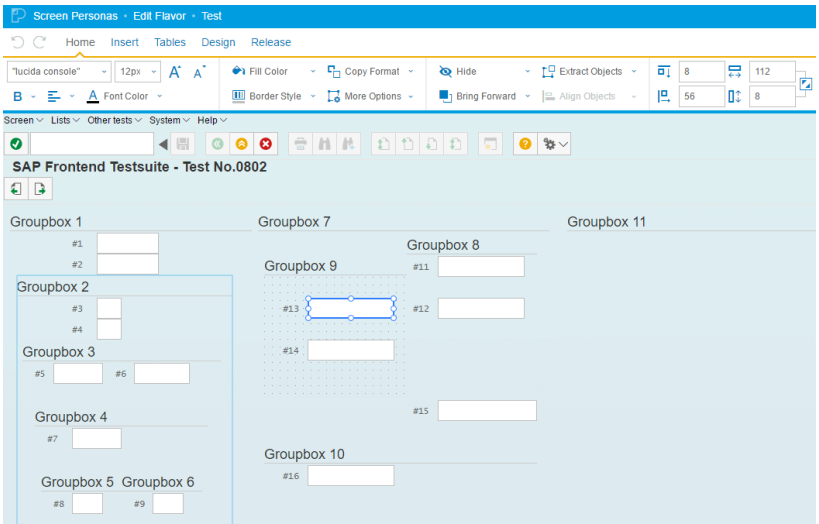
Pārlūki: Google Chrome 57.0.2978.98 (64 bit) un Mozilla Firefox 52.0 (32 bit)

3.1. Neapmierinoša sākotnējās ielādes veiktspēja

3.1.1 Problēmas apraksts

Tika meklēts veids, kā uzlabot veiktspēju situācijās, kad kādam no lietotāja saskarnes elementiem tika mainīts relatīvais izmērs, piemēram, garums.

Ja daudziem DOM koka elementiem vienlaikus izmēri ir izteikti relatīvās vienībās attiecībā pret to vecākelementiem, tad, mainot viena konkrēta elementa izmēru, tiek nolasīts vecākelementu izmērs, kā arī pārrēķināts un mainīts visu bērnu elementu izmērs. Mainoties vecāka izmēram, mainās arī izmērs nākamā līmeņa elementam. Process turpinās, līdz tiek sasniegts kāds elements, kura izmērs ir absolūts lielums. Šādi elementi parasti atrodas tuvāk saknes līmenim, tāpēc to sasniegšanai nepieciešams vispirms veikt pārrēķināšanu visiem bērnu elementiem. Turklāt, mainot izmērus, tiek ietekmēti arī kaimiņu elementi, kas iedarbina analoģisku procesu arī uz tiem. Rezultātā rodas veiktspējas samazināšanās un saskarnes elementu izmēra maiņa var notikt ar raustīšanos.



3.1.1.1. att. **SAP Personas ekrāns**

3.1.2. Problēmas analīze

Pētot problēmas cēloni, tika izmērīts, ka vēlamās tīmekļa lietotnes pārlādēšana aizņem apmēram 9 sekundes, no kurām 7 sekundes aizņem izkārtošanas procesi. Šajā laikā tiek nelietderīgi izmantoti procesora resursi un uz ekrāna redzamais saturs netiek atjaunināts, bet pārlūks nereaģē uz lietotāja darbībām.

Lietotnes pirmkodā tika atrasta konkrēta Javascript funkcija, kurā izpildās kods, kas izraisa izkārtošanas darbības. Šajā funkcijā bija atrodams kods:

```
oScrollContentRootRef.style.width = '100%';
oScrollContentRootRef.style.height = '100%';
```

3.1.3. Piedāvātais risinājums

Platuma izmērs, kas izteikts kā "100%", nozīmē, ka tiks iegūts arī vecāka elementa izmērs, lai no tā izskaitļotu vajadzīgā elementa platumu. Bet tā kā vecāka elementa izmērs arī ir izteikts procentos, izmēra noteikšanai jāskatās nākamā līmeņa vecāks. Rezultātā eksistējošais kods tika labots:

```
oScrollContentRootRef.style.width = oScrollDomRef.scrollWidth+'px';
oScrollContentRootRef.style.height = oScrollDomRef.scrollHeight+'px';
```

Šis kods atšķiras no sākotnējā ar to, ka objekta izmēru noteikšanai tagad pietiek vien zināt vecāka objekta izmērus. Līdz ar to objekta izmēru var noteikt konstantā laikā. Konkrētajā lietotnē šāds risinājums ir pieļaujams un tas nemaina lietotnes darbību, salīdzinājumā ar sākotnējo versiju.

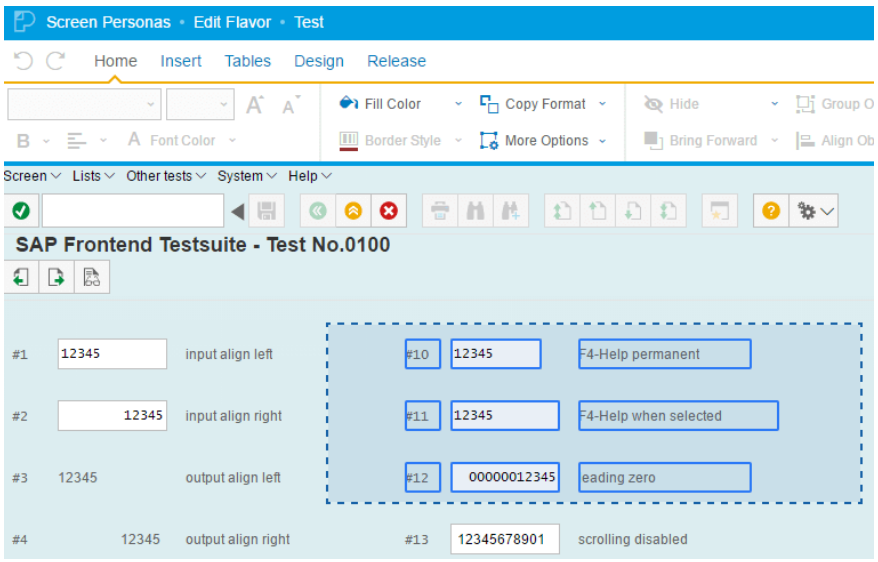
3.1.4 Risinājuma efektivitātes novērtējums

Sākotnējais risinājums paredzēja vecāku elementu izmēra nolasīšanu vairākos līmeņos, kamēr uzlabotais risinājums piekļūst tikai katra objekta nākamā līmeņa vecākam. Tāpēc risinājuma efektivitāte jeb starpība starp sākotnējo un laboto versiju ir atkarīga no tā, cik daudzos līmeņos objekta vecāku izmērs ir noteikts procentos.

3.2. Neapmierinoša dinamiskās attēlošanas veiktspēja

3.2.1 Problēmas apraksts

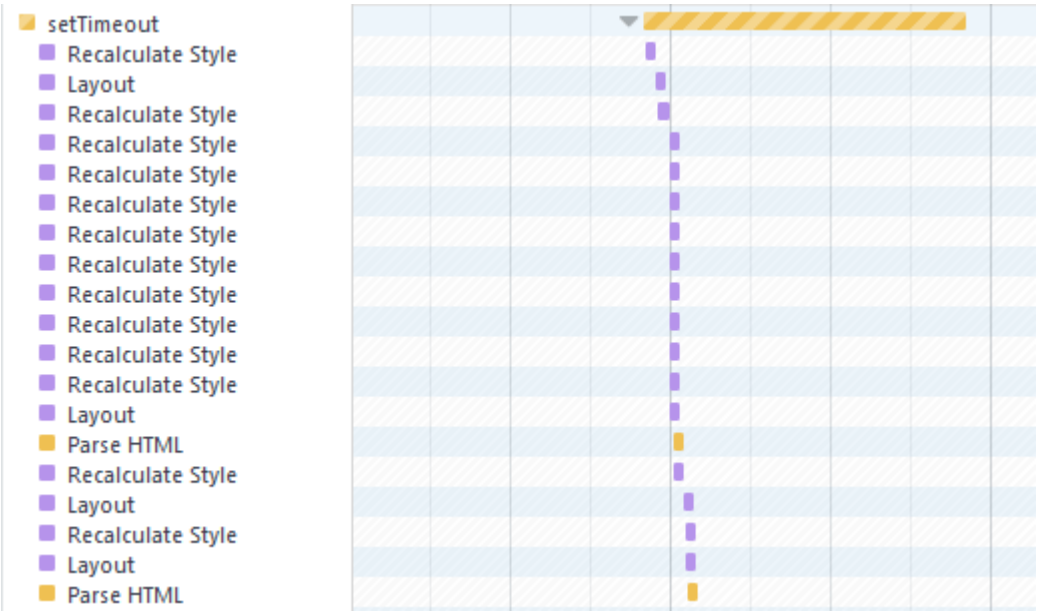
Tika apskatīts gadījums, kad vairāki vizuālie saskarnes elementi tiek iezīmēti ar peli. Iezīmēšana notika, turot nospiestu *Shift* taustiņu un velkot ar peli. Iezīmējot vairākus vizuālos elementus vienlaikus, tika ievērots, ka laikā, kad, velkot ar peli, mainās iezīmēto elementu skaits, iezīmēšanas process notiek ar aizturi. Pretējā gadījumā iezīmēšana notiek gandrīz bez aizķeršanās.



3.2.1.1. att. **Vairāku elementu iezīmēšanas procesa ilustrācija**

3.2.2. Problēmas analīze

Aplūkojot notiekošo, izmantojot pārlūka profilēšanas rīkus, izrādījās, ka pēc katra peles kustības notikuma iestāšanās notiek CSS izkārtojuma pārrēķināšana. Tas apskatāms attēlā:



3.2.2.1. att. **Pārlūka *Google Chrome* veiktās darbības, iezīmējot saskarnes elementus**

Mērot *mousemove* notikuma izpildes laiku, atklājās, ka, jo vairāk saskarnes elementu tika no jauna iezīmēts vai noņemts, jo lielāks ir izpildes laiks.

```
Multi select 28: 2.69ms
Multi select 28: 3.51ms
Multi select 28: 2.58ms
Multi select 32: 122ms
Multi select 33: 16.8ms
Multi select 34: 19.7ms
Multi select 36: 57.9ms
Multi select 36: 3.66ms
Multi select 36: 4.74ms
```

3.2.2.2. att. ***Google Chrome*** **konsoles izvads, iezīmējot saskarnes elementus**

Pārbaudot lietotnes izstrādātāju definēto `getReferenceRectangle` funkciju, tika ievērots, ka tā divas reizes izmanto *getBoundingClientRect* funkciju, kuras izpilde izraisa izkārtošanas procesu. Starp abiem *getBoundingClientRect* funkcijas izsaukumiem atrodams šāds kods:

```
mReturnRect.scrollTop = oScrollbar.scrollTop;
mReturnRect.scrollLeft = oScrollbar.scrollLeft;
```

Gan *scrollTop*, gan *scrollLeft* īpašību nolasīšana un mainīšana izraisa piespiedu izkārtošanu, t.i., pārlūks obligāti izpildīs visas tās ar izkārtošanu saistītās darbības, kuras iepriekš bija atliktas uz vēlāku laiku. Izkārtošanas process izpildās sinhroni, tāpēc pārlūks nevar reaģēt uz lietotāja darbībām. Tādējādi, lietotājam turpinot iezīmēt saskarnes elementus, iezīmēšana notiek ar raustīšanos.

3.2.3. Piedāvātais risinājums

Ir ievērots, ka viena no lietotnē iepriekš definētajām funkcijām izmanto `getBoundingClientRect`, kura savukārt izraisa izkārtošanas procesu. Diemžēl no šīs funkcijas izmantošanas nevar atteikties pavisam, tomēr ir iespējams samazināt tās izsaukumu skaitu līdz minimāli iespējamajam.

Papildus tam, ir iespējams optimizēt objektu iezīmēšanu. Proti, tad, kad iezīmēšanas laikā objekts, kurš agrāk bija iezīmēts, vairs tāds nav, tad nebūtu jādzēš atsauce uz papildu objektu, kas nodrošina iezīmējuma grafisko attēlošanu. Tā vietā šo objektu var pasludināt par neredzamu un nākamajā iezīmēšanas reizē viss, ko vajadzēs izdarīt, ir tikai objekta padarīšana par redzamu.

3.2.4 Risinājuma efektivitātes novērtējums

Risinājuma efektivitāte galvenokārt izpaužas kā atbrīvošanās no nelietderīgi izmantotajiem funkciju izsaukumiem un neefektīvu funkciju aizstāšana ar tādām, kuras neizraisa izkārtošanu.

3.3. Nepietiekama datu apstrādes veiktspēja

3.3.1. Problēmas apraksts

Novērots, ka pēc visu tīmekļa lietotnei nepieciešamo JSON formāta datu saņemšanas, to apstrāde notiek pārāk lēni – tā aizņem vismaz 1 sekundi. Šajā laikā lietotne nereaģē uz lietotāja darbībām.

3.3.2. Problēmas analīze

Noskaidrots, ka apstrādājamo datu apjoms ir aptuveni 11 MB. Datu apstrāde notiek sinhroni, apturot visas lietotnes darbību.

3.3.3. Piedāvātais risinājums

Problēmu iespējams risināt, optimizējot datu apstrādes Javascript pirmkodu – veicot tādas izmaiņas kodā, lai tā izpildes laiks samazinātos. Tomēr ņemot vērā, ka datu apstrāde aptur pārējo lietotnes daļu darbību, šo procesu var paveikt citā pavedienā ārpus galvenā pavediena, kurā izpildās Javascript kods. To iespējams izdarīt, izmantojot tīmekļa strādni (*web worker*).

Tīmekļa strādņi ļauj asinhroni izpildīt Javascript skriptus atsevišķā izolētā pavedienā. Tie nespēj piekļūt DOM kokam un *window* objektam, tomēr tiem ir iespējams padot apstrādājamos JSON datus [17].

Šajā gadījumā ir iespējams definēt jaunu strādni, kuram tiktu padota saite uz JSON datiem. Strādnī var veikt datu apstrādi un kad tas izdarīts, strādnis paziņo tīmekļa lietotnei par darba beigšanu. Datu kopēšana nav vajadzīga ne pirms, ne pēc apstrādes.

3.3.4. Risinājuma efektivitātes novērtējums

Risinājums pārvirza JSON datu apstrādes procesu no galvenā pavediena, tāpēc lietotājs var nemaz nepamanīt, ka kaut kas tiek darīts. Šajā gadījumā patērētais aparatūras resursu apjoms neatšķiras no sākotnējās implementācijas; atšķirība ir tikai aparatūras resursu sadalīšanā.

REZULTĀTI

Darba ietvaros tika izpētīti pārlūkprogrammu un to komponenšu darbības principi. Balstoties uz iegūtajām zināšanām, tika izdarīti secinājumi par atsevišķiem potenciālajiem veiktspējas problēmu cēloņiem, kuri ir tiešā veidā saistīti ar konkrētu pārlūku implementācijām.

Lai pārbaudītu dažādu specifisko DOM funkciju ātrdarbību, tika izveidoti testpiemēri, kuros var mērīt funkciju izpildes laiku pie noteiktas slodzes un redzēt pārlūka veiktās darbības šo funkciju izpildes laikā (skat. 1. pielikumu).

Ir izstrādāta lietotne, kurā iespējams ievadīt lietotāja izvēlētās īpašības un dažādos pārlūkos pārbaudīt, kuras Javascript īpašības izraisa izkārtošanu un kuras to neizraisa (skat. 3. pielikumu).

Tika atrasti risinājumi vairākiem veiktspējas problēmgadījumiem reālās tīmekļa lietotnēs. Visos gadījumos ir pamanāma veiktspējas uzlabošanās gan ar, gan bez speciālu mērījumu veikšanas.

SECINĀJUMI

Mūsdienu interneta pārlūku darbība lielā mērā balstās uz sinhrono izpildes modeli, tomēr pārlūkos ir iebūvēti daudzi mehānismi, kas ļauj optimizēt tīmekļa vietņu ielādes procesu. Var secināt, ka mainot darbību prioritātes un izdarot tikai tik daudz darba, cik ir nepieciešams, var būtiski uzlabot grafiskās saskarnes veiktspēju. Papildus tam, pārlūkos ir iebūvēta funkcionalitāte, kas nodrošina HTML standartam neatbilstoša koda izpildi.

Var secināt, ka tīmekļa lietotņu izstrādātājiem ir pieejamas iespējas veidot tādas lietotnes, kas izmanto ne tikai procesora, bet arī videokartes jaudu. Modernas tīmekļa lietotnes var tikt darbinātas vairāku procesoru sistēmās, dažādos izpildes pavedienos, tādējādi efektīvāk izmantojot pieejamos aparatūras resursus.

Dažādu stilu īpašību piemērošana izpildās visai atšķirīgā veidā, ar atšķirīgu darbību daudzumu, kas jāveic, lai vēlamās īpašības tiktu dinamiski nomainītas un stātos spēkā. Eksistē gan tādas īpašības, kuru izmaiņa izsauc veselu ekrāna apgabalu pārzīmēšanu, gan tādas, kuru izpilde prasa relatīvi minimālus procesora resursus. Tātad ir būtiski, kādas CSS stilu īpašības izvēlēties, lai izstrādātu vēlamo lietotāja saskarnes izskatu.

Veicot praktiskus eksperimentus un izdarot reālu veiktspējas problēmgadījumu analīzi, tika secināts, ka pat divas Javascript koda rindas spēj ievērojami paildzināt tīmekļa lietotņu ielādes laiku un izsaukt citus negatīvus blakusefektus. Tātad vērts zināt, kā interneta pārlūki izpilda tīmekļa lietotņu programmas kodu, lai nodrošinātu maksimāli efektīvu grafiskās saskarnes veiktspēju.

IZMANTOTĀ LITERATŪRA UN AVOTI

1. Grosskurth, Alan, *A Reference Architecture for Web Browsers*, [tiešsaiste]. [Atsauce 07.03.2017]. Pieejams: http://grosskurth.ca/papers/browser-refarch.pdf
2. *Google Chrome (Vikipēdijas raksts),* [tiešsaiste]. [Atsauce 07.03.2017]. Pieejams: https://en.wikipedia.org/wiki/Google_Chrome
3. Web Fundamentals. Rendering Performance, [tiešsaiste]. [Atsauce 08.03.2017]. Pieejams: https://developers.google.com/web/fundamentals/performance/rendering/
4. How Browsers Work, [tiešsaiste]. [Atsauce 10.03.2017]. Pieejams: https://www.html5rocks.com/en/tutorials/internals/howbrowserswork/
5. CSS Triggers, [tiešsaiste]. [Atsauce 10.03.2017]. Pieejams: https://csstriggers.com/
6. CSS Properties By Style Operation Required, [tiešsaiste]. [Atsauce 10.03.2017]. Pieejams: https://docs.google.com/spreadsheets/d/1Hvi0nu2wG3oQ51XRHtMv-A_ZlidnwUYwgQsPQUg1R2s/pub?single=true&gid=0&output=html
7. Nicholas C. Zakas, *High Performance JavaScript*, O'Reilly, 2010
8. Lukas Rychnovsky, *Parsing of Context Sensitive Languages*, 2007
9. David Flanagan, *JavaScript: The Definitive Guide: Activate Your Web Pages*, O'Reilly, 2011 318.-320. lpp.
10. HTML5 specifikācija, [tiešsaiste]. [Atsauce 17.03.2017]. Pieejams: https://www.w3.org/TR/html5/
11. HTML parsera pirmkods (WebKit), [tiešsaiste]. [Atsauce 03.04.2017]. Pieejams: https://opensource.apple.com/source/WebCore/WebCore-955.66/html/HTMLParser.cpp.auto.html
12. CSS3 formālā specifikācija, [tiešsaiste]. [Atsauce 03.04.2017]. Pieejams: https://www.w3.org/TR/css3-selectors/
13. Chris Waterson, *Notes on HTML Reflow*, [tiešsaiste]. [Atsauce 07.04.2017]. Pieejams: https://www-archive.mozilla.org/newlayout/doc/reflow.html
14. The Stacking Context, [tiešsaiste]. [Atsauce 12.04.2017]. Pieejams: https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Positioning/Understanding_z_index/The_stacking_context
15. Stoyan Stefanov, *Javascript Patterns*, O'Reilly, 2010
16. Writing Efficient CSS Selectors, [tiešsaiste]. [Atsauce 04.05.2017]. Pieejams: https://csswizardry.com/2011/09/writing-efficient-css-selectors/
17. Eric Bidelman, *The Basics of Web Workers*, [tiešsaiste]. [Atsauce 15.05.2017]. Pieejams: https://www.html5rocks.com/en/tutorials/workers/basics/#toc-inlineworkers
18. Paul Irish, What Forces Layout / Reflow, [tiešsaiste]. [Atsauce 18.05.2017]. Pieejams: https://gist.github.com/paulirish/5d52fb081b3570c81e3a

PIELIKUMI

1. pielikums.

DOM funkciju veiktspējas testu HTML pirmkods

```
<!DOCTYPE HTML>
<html>
<head>
<title>DOM manipulation test</title>
<style>
.btn {
 width: 5rem;
}
.captions {
 width: 12rem;
 display: inline-block;
}
</style>
</head>
<body>
<script type="text/javascript">
 function runAppendWithReflow(iterations) {
       var div = document.createElement('div');
       var w = 0;
       document.body.appendChild(div);
       console.time("Div - append with reflow " + iterations + "x");
       for (var i = 0; i < iterations; i++) {
             div.innerHTML += '<li>List item ' + i + '</li>';
             w = div.offsetWidth;
       }
       console.timeEnd("Div - append with reflow " + iterations + "x");
 }

 function runAppend(iterations) {
       var div = document.createElement('div');
       document.body.appendChild(div);
       console.time("Div - append " + iterations + "x");
       for (var i = 0; i < iterations; i++) {
             div.innerHTML += '<li>List item ' + i + '</li>';
       }
       console.timeEnd("Div - append " + iterations + "x");
 }

 function runAppendChild(iterations) {
       var div = document.createElement('div');
       document.body.appendChild(div);
       console.time("Div - appendChild " + iterations + "x");
       for (var i = 0; i < iterations; i++) {
             var el = document.createElement('li');
             el.innerText = 'List item ' + i;
             div.appendChild(el);
       }
       console.timeEnd("Div - appendChild " + iterations + "x");
 }

 function runAppendFragment(iterations) {
       var div = document.createElement('div');
       var fragment = document.createDocumentFragment();
       document.body.appendChild(div);
       console.time("Div - append fragment " + iterations + "x");
```

```
            for (var i = 0; i < iterations; i++) {
                  var el = document.createElement('li');
                  el.innerText = 'List append fragment ' + i;
                  fragment.appendChild(el);
            }
            div.appendChild(fragment);
            console.timeEnd("Div - append fragment " + iterations + "x");
      }

      function runAppendJoin(iterations) {
            var div = document.createElement('div');
            var html = '';
            document.body.appendChild(div);
            console.time("Div - append join " + iterations + "x");
            for (var i = 0; i < iterations; i++) {
                  html += '<li>List item ' + i + '</li>';
            }
            div.innerHTML = html;
            console.timeEnd("Div - append join " + iterations + "x");
      }
    </script>
    <span class="captions">Append list elements, reflow </span><button
class="btn" onclick="runAppendWithReflow(100)">100x</button>
    <button class="btn" onclick="runAppendWithReflow(200)">200x</button>
    <button class="btn" onclick="runAppendWithReflow(500)">500x</button>
    <br/><br/>
    <span class="captions">Append list elements </span><button class="btn"
onclick="runAppend(100)">100x</button>
    <button class="btn" onclick="runAppend(200)">200x</button>
    <button class="btn" onclick="runAppend(500)">500x</button>
    <button class="btn" onclick="runAppend(1000)">1000x</button>
    <button class="btn" onclick="runAppend(2000)">2000x</button>
    <br/><br/>
    <span class="captions">Append (appendChild) </span><button class="btn"
onclick="runAppendChild(100)">100x</button>
    <button class="btn" onclick="runAppendChild(200)">200x</button>
    <button class="btn" onclick="runAppendChild(500)">500x</button>
    <button class="btn" onclick="runAppendChild(1000)">1000x</button>
    <button class="btn" onclick="runAppendChild(2000)">2000x</button>
    <button class="btn" onclick="runAppendChild(10000)">10000x</button>
    <button class="btn" onclick="runAppendChild(20000)">20000x</button>
    <br/><br/>
    <span class="captions">Append (with fragment) </span><button class="btn"
onclick="runAppendFragment(100)">100x</button>
    <button class="btn" onclick="runAppendFragment(200)">200x</button>
    <button class="btn" onclick="runAppendFragment(500)">500x</button>
    <button class="btn" onclick="runAppendFragment(1000)">1000x</button>
    <button class="btn" onclick="runAppendFragment(2000)">2000x</button>
    <button class="btn" onclick="runAppendFragment(10000)">10000x</button>
    <button class="btn" onclick="runAppendFragment(20000)">20000x</button>
    <br/><br/>
    <span class="captions">Append list elements join </span><button
class="btn" onclick="runAppendJoin(100)">100x</button>
    <button class="btn" onclick="runAppendJoin(200)">200x</button>
    <button class="btn" onclick="runAppendJoin(500)">500x</button>
    <button class="btn" onclick="runAppendJoin(1000)">1000x</button>
    <button class="btn" onclick="runAppendJoin(2000)">2000x</button>
    <button class="btn" onclick="runAppendJoin(10000)">10000x</button>
    <button class="btn" onclick="runAppendJoin(20000)">20000x</button>
    <br/><br/>
    <div id="testDiv"></div>
    </body>
    </html>
```

- `elem.focus()`
- `elem.computedRole, elem.computedName`
- `elem.innerText`

Ģeometriskie dati:

- `elem.offsetLeft, elem.offsetTop,`
- `elem.offsetWidth, elem.offsetHeight, elem.offsetParent`
- `elem.clientLeft, elem.clientTop,`
- `elem.clientWidth, elem.clientHeight`
- `elem.getClientRects(), elem.getBoundingClientRect()`

Ritināšana:

- `elem.scrollBy(), elem.scrollTo()`
- `elem.scrollIntoView(), elem.scrollIntoViewIfNeeded()`
- `elem.scrollWidth, elem.scrollHeight`
- `elem.scrollLeft, elem.scrollTop`

Window objekts:

- `window.getComputedStyle()`
- `window.scrollX, window.scrollY`
- `window.innerHeight, window.innerWidth`

Formas [18]:

- `input.focus()`
- `input.select(), textarea.select()`

Peles notikumi [18]:

- `mouseEvent.layerX, mouseEvent.layerY,`
- `mouseEvent.offsetX, mouseEvent.offsetY`

Document objekts [18]:

- `doc.scrollingElement`

Apgabals:

- `range.getClientRects(), range.getBoundingClientRect()`

Piespiedu izkārtošanas testu HTML pirmkods

```
<!DOCTYPE HTML>
<html>
<head>
<title>Reflow test</title>
</head>
<body>
<script type="text/javascript">
      function runTest() {
            var properties = ["accessKey", "align", "className",
"clientHeight", "clientLeft", "clientTop", "clientWidth",
"contentEditable", "getBoundingClientRect()", "getClientRects()", "hidden",
"innerHTML", "innerText", "localName", "nodeName", "nodeType",
"offsetHeight", "offsetLeft", "offsetParent", "offsetTop", "offsetWidth",
"outerHTML", "outerText", "scrollHeight", "scrollLeft", "scrollTop",
"scrollWidth", "textContent"]; // default properties for testing
            var div = document.createElement('div');
            var results = [];
            var propertyExists;
            var value = 0; // variable for reflow testing
            var i, j, start, end, minTime, maxTime, firstTestTime;
            var propertiesInput =
document.getElementById("propertiesInput");
            if (propertiesInput.value) {
                  properties = propertiesInput.value.split(",");
            }
            for (i = 0; i < properties.length; i++) {
                  properties[i] = properties[i].trim();
            }
            var iterations =
document.getElementById("iterationCount").value || 100;
            document.body.appendChild(div);
            minTime = 0;
            maxTime = 0;

            // first DOM manipulations
            for (j = 0; j < iterations; j++) {
                  div.innerHTML += '<li>List item ' + j + '</li>';
            }
            div.innerHTML = '';

            // first DOM empty test (no reflow)
            start = Date.now();
            for (j = 0; j < iterations; j++) {
                  div.innerHTML += '<li>List item ' + j + '</li>';
            }
            firstTestTime = Date.now() - start;
            div.innerHTML = '';

            // actual DOM test
            for (i = 0; i < properties.length; i++) {
                  propertyExists = false;
                  if (properties[i].substr(properties[i].length - 2) ===
"()") {
                        if (div[properties[i].substring(0,
properties[i].length - 2)]) {
                              propertyExists = true;
                        }
                  } else {
```

```
                        if (div[properties[i]] !== undefined) {
                              propertyExists = true;
                        }
                  }
                  if (propertyExists) {
                        start = Date.now();
                        for (j = 0; j < iterations; j++) {
                              div.innerHTML += '<li>List item ' + j +
'</li>';
                              if (properties[i].substr(properties[i].length
- 2) === "()") {
                                    value = eval("div." + properties[i]);
                              } else {
                                    value = div[properties[i]];
                              }
                        }
                        end = Date.now();
                        results[i] = end - start;
                        if (i === 0) {
                              minTime = results[i];
                              maxTime = results[i];
                        } else {
                              if ((results[i] < minTime) && (results[i] >
0)) {
                                    minTime = results[i];
                              }
                              if ((results[i] > maxTime) && (results[i] >
0)) {
                                    maxTime = results[i];
                              }
                        }
                        div.innerHTML = '';
                  } else {
                        results[i] = -1;
                  }
            }

            // summary
            var summaryDiv = document.getElementById("outputSummaryDiv");
            summaryDiv.innerHTML = '';
            var reflowProperties = [];
            var reflowFreeProperties = [];
            var unknownProperties = [];
            if (minTime < 2 || maxTime < 2) {
                  summaryDiv.innerHTML = 'Not enough iterations to test
reflow. Please choose bigger number.';
                  document.getElementById("outputDiv").innerHTML = '';
                  return;
            }
            for (i = 0; i < results.length; i++) {
                  if (results[i] === -1) {
                        unknownProperties.push(properties[i]);
                        continue;
                  }
                  if (maxTime / minTime > 3) { // there were some reflow
and reflow-free properties
                        if (results[i] / minTime <= 2) {
                              reflowFreeProperties.push(properties[i]);
                        }
                        if (maxTime / results[i] <= 2) {
                              reflowProperties.push(properties[i]);
                        }
                        if (!(results[i] / minTime <= 2) && !(maxTime /
```

```
results[i] <= 2)) {
                              unknownProperties.push(properties[i]);
                        }
                  } else { // only reflow or reflow-free
                        if ((results[i] / firstTestTime <= 2) &&
(firstTestTime / results[i] <= 2)) {
                              reflowFreeProperties.push(properties[i]);
                        }
                        if (results[i] / firstTestTime > 2) {
                              reflowProperties.push(properties[i]);
                        }
                        if (!((results[i] / firstTestTime <= 2) &&
(firstTestTime / results[i] <= 2)) && !(results[i] / firstTestTime > 2)) {
                              unknownProperties.push(properties[i]);
                        }
                  }
            }

            summaryDiv.innerHTML += 'Reflow:<br/>' +
reflowProperties.join(", ") + '<br/><br/>';
            summaryDiv.innerHTML += 'No reflow:<br/>' +
reflowFreeProperties.join(", ") + '<br/><br/>';
            if (unknownProperties.length) {
                  summaryDiv.innerHTML += 'Unknown:<br/>' +
unknownProperties.join(", ") + '<br/><br/>';
            }

            // output to table
            var resultTable = '<table><tr><th>Property</th><th>Duration,
ms</th></tr>';
            for (i = 0; i < properties.length; i++) {
                  if (results[i] === -1) {
                        resultTable += '<tr><td>'+ properties[i] +
'</td><td>N/A</td></tr>';
                  } else {
                        resultTable += '<tr><td>'+ properties[i] +
'</td><td>' + results[i].toString() + '</td></tr>';
                  }
            }
            resultTable += '</table>';
            document.getElementById("outputDiv").innerHTML = resultTable;
      }
</script>
Enter HTML element properties for reflow testing (comma-separated), for
example <i>clientWidth, clientHeight, getClientRects()</i>:<br/>
<textarea id="propertiesInput" rows="4" cols="120"></textarea><br/>
Number of DOM event iterations for each property:
<select id="iterationCount">
      <option value="50">50</option>
      <option value="75">75</option>
      <option value="100" selected="selected">100</option>
      <option value="200">200</option>
      <option value="500">500</option>
</select><br/>
<button id="testBtn" onclick="runTest()">Test</button>
<br/><br/>
<div id="outputSummaryDiv"></div>
<br/><br/>
<div id="outputDiv"></div>
<br/><br/>
<div id="testDiv"></div>
</body>
</html>
```

REĢISTRĀCIJAS LAPA

Bakalaura darbs „Tīmekļa lietotņu klienta puses optimizācija” izstrādāts LU Datorikas fakultātē.

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai.

Autors: _____ Romāns Kolduns

Rekomendēju/nerekomendēju darbu aizstāvēšanai
Vadītājs: M. dat., Pēteris Prokofjevs _____ ___.05.2017.

Recenzents: asociētais profesors, Dr. dat. Darja Solodovņikova

Darbs iesniegts Datorikas fakultātē ___.05.2017.
Dekāna pilnvarotā persona:
vecākā metodiķe Ārija Sproģe _____

Darbs aizstāvēts bakalaura gala pārbaudījuma komisijas sēdē

_____

Komisijas sekretāre: _____