LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

TEKSTA IEGŪŠANA NO HTML5 CANVAS ELEMENTIEM

BAKALAURA DARBS

Autors: **Vilnis Laipnieks**
Studenta apliecības Nr.: vl12030
Darba vadītāja: docente Dr. dat. Darja Solodovņikova

RĪGA 2016

ANOTĀCIJA

HTML5 *canvas* elements un visa tajā atveidotā informācija pēc būtības ir attēls, līdz ar to, no lietojamības viedokļa, grūti pieejams lietotājiem, it īpaši redzes invalīdiem, kuri paļaujas uz ekrāna lasīšanas programmatūru.

Darbā pētītas vairākas dažādas metodes HTML5 *canvas* elementos zīmētās teksta informācijas iegūšanai no lietotāja puses, ļaujot tekstu iezīmēt, kopēt un padarot to pieejamu ekrāna lasīšanas programmatūrai, tādējādi uzlabojot tīmekļa vides lietojamību, it īpaši cilvēkiem ar redzes problēmām.

Darba rezultātā veiksmīgi izveidots *JavaScript* spraudnis, kuru tīmekļa lietotņu izstrādātāji var izmantot, lai bez piepūles zīmētu iezīmējamu, kopējamu un ekrāna lasītājiem pieejamu tekstu, to joprojām saglabājot kā daļu no *canvas* zīmētā attēla.

Atslēgvārdi: canvas, HTML, ekrāna lasītāji, JavaScript, spraudnis, lietojamība, pieejamība.

ABSTRACT

TEXT RETRIEVAL FROM HTML5 CANVAS ELEMENTS

The HTML5 canvas element and all information represented within, is, by definition, an image, which is poorly accessible by users, especially visually handicapped ones, who rely on screen reading software.

The work investigates multiple different approaches to retrieval of text, which is drawn within an HTML5 canvas element, as a user, by letting the user select and copy the text, and making it accessible to screen reading software, thus improving the usability of the web, especially to the visually handicapped.

A successful result of the work is a JavaScript plugin, which can be used by web application developers to effortlessly draw selectable, copiable and screen reader accessible text within the native canvas.

Keywords: canvas, HTML, screen readers, JavaScript, plugin, usability, accessibility.

SATURS

APZĪMĒJUMU SARAKSTS 6
IEVADS 7
1. PĒTĀMĀ PROBLĒMA 8
1.1. Problēma ar teksta pieejamību *canvas* elementos 8
1.2. Problēmas risinājuma prasības 9
2. ZINĀMIE RISINĀJUMI 11
2.1. Teksta glabāšana atsevišķos DOM elementos 11
2.2. Slēpto DOM elementu izmantošana 12
2.3. OCR izmantošana 14
2.4. Iezīmēšana ar *JavaScript* palīdzību 16
2.4.1. CanvasInput 16
2.4.2. Textor 17
2.4.3. Canvas Text Editor 18
2.4.4. Esošo *JavaScript* risinājumu salīdzinājums 19
3. JAVASCRIPT SPRAUDŅA TEKSTA IEGUVEI DARBĪBAS PRINCIPI 20
3.1. Prasības pret datiem par teksta elementu 20
3.2. Prasības teksta pieejamībai ekrāna lasītājiem 22
3.3. Pamata funkciju darbības principi un prasības 24
3.3.1. Teksta zīmēšana 24
3.3.2. Teksta elementa noteikšana 24
3.3.3. Konkrēta simbola noteikšana 24
3.3.4. Simbolu iezīmēšana 25
3.3.5. Vārda tūlītēja iezīmēšana 25
3.3.6. Vairāku līniju un teksta elementu iezīmēšana 26
3.3.7. Iezīmēto elementu glabāšana vai izvade pareizā secībā 26
3.3.8. Iezīmējuma iekrāsošana 26

3.3.9. Teksta elementa jaunināšana ... 28
3.3.10. Pielāgošanās līdzinājuma un bāzlīnijas izmaiņām ... 28
3.3.11. Kopēšana ... 29
3.3.12. Sagatavošana pieejamībai ... 29
3.3.13. Dublikātu atmešana ... 29
3.3.14. Izveidoto elementu iznīcināšana ... 29
4. IZVEIDOTĀ PROGRAMMATŪRA ... 30
4.1. CanvasTextAccess darbība ... 31
4.1.1. Teksta izveidošana ... 31
4.1.2. Teksta jaunināšana ... 34
4.1.3. Teksta iezīmēšana ... 35
4.1.4. Teksta iekrāsošana ... 39
4.1.5. Teksta kopēšana ... 40
4.1.6. Teksta redzamība ekrāna lasītājiem ... 42
4.2. CanvasTextAccess veiktspēja ... 44
4.3. CanvasTextAccess spraudņa kopsavilkums ... 46
SECINĀJUMI ... 47
IZMANTOTĀ LITERATŪRA UN AVOTI ... 48
PIELIKUMI ... 50
1. PIELIKUMS. "CANVASTEXTACCESS" IZVĒLĒTU METOŽU PIRMKODS 50
1.1. TextAccessCanvas konstruktora funkcija ... 50
1.2. TextAccessElement konstruktora funkcija ... 51
1.3. Metode textAccessCanvas.selectMissing ... 53
1.4. Metode textAccessCanvas.selectWholeElement ... 54
1.5. Metode textAccessElement.getLineHeight ... 55
1.6. Metode textAccessElement.updateText ... 56
1.7. Metodes bāzlīnijas un līdzinājuma kompensēšanai ... 57

APZĪMĒJUMU SARAKSTS

CSS – kaskādisku stilu saraksts. Apraksts tam, kā jāizskatās noteiktiem HTML dokumenta elementiem.

DOM – dokumentu objektu modelis. W3C izveidota specifikācija HTML, XHTML un XML dokumentu reprezentācijai, kurā dokumenta elementi ir izkārtoti koka struktūrā.

HTML – hiperteksta marķēšanas valoda. Specifikācija tam, kā pierakstīt dažādus elementus tīmekļa dokumentā, lai norādītu, kā pārlūkprogrammai tos atveidot.

OCR - no angļu valodas „Optical character recognition” jeb optiskā simbolu atpazīšana, ir elektroniska attēlos ievietotā teksta pārveidošana mašīnām saprotamā kodējumā.

W3C - Vispasaules Tīmekļa konsorcijs ir konsorcijs, kas veido standartus jeb "rekomendācijas" vispasaules tīmeklim.

WAI-ARIA - no angļu valodas „Web Accessibility Initiative - Accessible Rich Internet Applications” ir tehniskā specifikācija, kura apraksta veidus dinamisku tīmekļa vietņu pieejamības palielināšanai.

IEVADS

Mūsdienās tīmekļa vide ieņem arvien lielāku lomu cilvēku dzīvēs. Tīmekļa vietnes un lietotnes ar to sniegtajām, plašajām multivides iespējām ļauj ātri, ērti un vienkārši ievietot un uzzināt milzīgus informācijas apjomus.

Viens no populārākajiem tīmekļa vietņu elementiem, kurš popularitāti un atbalstu guvis līdz ar HTML5 specifikācijas izveidi, ir *canvas* elements. Šis multivides elements ļauj veidot košas grafiskās lietotnes, tai skaitā spēles, iespējas zīmēt grafikus un vēl. Tomēr šis elements neļauj vienkāršā veidā iegūt tajā attēloto informāciju – tajā attēlotais teksts nav lietotājam pieejams ne iezīmēšanai, ne kopēšanai, kā arī *canvas* elements neatveido tajā attēloto informāciju pieejamu cilvēkiem ar īpašām vajadzībām, to slēpjot no speciālajām programmatūrām ekrāna lasīšanai.

Lai tīmekļa vides populācija nebūtu spiesta katru reizi, kad vēlas saglabāt *canvas* attēloto informāciju, ar roku norakstīt tekstu uz lapas vai atsevišķā teksta redaktorā un neierobežotu tīmekļa pārlūkošanas iespējas cilvēkiem ar redzes problēmām, ir jāmeklē uzticams risinājums, kuru būtu vienkārši iekļaut tīmekļa lietotnē tās izstrādes laikā un kurš ļautu iegūt tekstu no HTML5 *canvas* elementiem lietotājiem intuitīvi saprotamā veidā.

Šajā darbā tiek pētīta iespēja izveidot *JavaScript* spraudni, kurš padara *canvas* zīmēto tekstu pieejamu lietotājiem. Darba struktūra:

1. Pētāmās problēmas un vēlamā risinājuma apraksts;
2. Tīmekļa vidē atrasto potenciālo risinājumu apskats un novērtējums;
3. Darbības principu un prasību uzskaitījums jauna *JavaScript* spraudņa izveidei;
4. Izveidotā rezultāta apskats.

1. PĒTĀMĀ PROBLĒMA

1.1. Problēma ar teksta pieejamību *canvas* elementos

Viens no būtiskākajiem jauninājumiem HTML specifikācijas 5. versijā ir *canvas* elements. Tā pirmsākumi atrodami 2004. gadā, kad kompānija "Apple" šo elementu laida klajā kopā ar savas operētājsistēmas "OS X", par *WebKit* dēvēto tīmekļa izvietojuma komponenti. Kopš tā laika, *canvas* ir ieviesa arī tādas kompānijas kā "Mozilla" un "Opera", līdz W3C nolēma to ieviest kā daļu no jaunās HTML specifikācijas [1].

Canvas ir tīri vizuālas dabas elements, un tā galvenā funkcija ir veidot ar *JavaScript* kontrolētus bitkartes zīmējumus. *Canvas* pielieto t.s. "izšauj un aizmirsti" zīmēšanas metodoloģiju – brīdī, kad *JavaScript* kodam tiek padots zīmējamais elements un tas parādās *canvas* reģionā, tas pārtop par daļu no *canvas* bitkartes un vairs nav pieejams lietotājam, jo nav iekšējās DOM struktūras. Lai zīmēto bitkarti pārveidotu, viss zīmējums jādzēš nost un no atkal jāzīmē jaunā, pārveidotā zīmējuma versija. Vienkāršots salīdzinājums būtu ar to, kā strādā grafisko elementu ievietošana attēlos ar MS Paint programmas palīdzību, proti, neko vairs nevar iesākt no brīža, kad elements kļūst par daļu no zīmējuma. Toties iekšējās DOM struktūras trūkums un tiešā kontrole ar *JavaScript* nozīmē ļoti augstu zīmēšanas ātrumu.

Uz šī elementa ir iespējams zīmēt figūras, vilkt līnijas, novietot jau esošus grafiskos elementus, izmantot kā izeju gan vektoru, gan rastera grafikai, kā arī zīmēt tekstu.

Problēma ir tajā, ka arī teksts, kurš ir ierakstīts *canvas* elementā, kļūst par daļu no zīmējuma un tas nozīmē to, ka tas nav pieejams ekrāna lasīšanas programmatūrai un uz *canvas* balstītās *JavaScript* tīmekļa lietotnes nav pieejamas redzes invalīdiem vai vājredzīgiem cilvēkiem, jo ekrāna lasīšanas programmatūra spēj piekļūt tikai tādiem HTML elementiem kā *<div>, <p>, <span>* u.c. elementiem, kuri satur simbolu virknes. Teksts arī nav pieejams iezīmēšanai un kopēšanai – tā ir problēma, kura īpaši jūtama kļūst mobilo ierīču kontekstā, kad atvērt citu programmu teksta norakstīšanai ir nepraktiski, tomēr var būt vajadzība to darīt, piemēram, svarīga teksta kopēšanai no spēles saskarnes vai ar *canvas* palīdzību izveidota tīmekļa vietnes teksta kopēšanai.

1.2. Problēmas risinājuma prasības

Vienkāršākais veids, kā apiet problēmas gan ar teksta iezīmēšanu, gan ekrāna lasīšanu, ir teksta elementus ievietot atsevišķos HTML elementos un ar CSS palīdzību novietot tos īstajās vietās virs *canvas* elementa, lai tie izskatītos kā daļa no lietotnes saskarnes, taču tas prasa papildu elementu apstrādi un ir gadījumi, kuros hibrīdie risinājumi (HTML, CSS un *canvas*) nestrādā un vajadzīgs tīrs *canvas* risinājums, vismaz no ārpuses.

Lietojot metodi ar *canvas* pārklāšanu ar citiem DOM elementiem, izstrādātājam arī vairāk jāpiedomā pie teksta animēšanas, izmantojot CSS vai tādas bibliotēkas kā *jQuery,* kas var radīt liekas problēmas risinājumos, kuriem jāatbalsta vairākas, dažādas pārlūkprogrammas. Teksts, kurš zīmēts *canvas* izskatīsies vienlīdzīgi visos pārlūkos (kuri atbalsta HTML5).

Arī citi potenciālie risinājumi *canvas* teksta iegūšanai ir problemātiski un nepilnīgi. Teksta slēpšana ar CSS palīdzību tikai padara tekstu pieejamu ekrāna lasīšanas programmatūrai, nevis iezīmēšanai un optiskā simbolu noteikšana (angliski - "optical character recognition", jeb OCR) arī neatbalsta teksta iezīmēšanu, taču var panākt ekrāna lasītāju atbalstu, ja *canvas* attēlotais teksts ir skaidri salasāms un bez raiba fona.

Darba mērķis ir izveidot *JavaScript* spraudni*,* ar kura palīdzību, uz *canvas* zīmētais teksts būtu:

- Iezīmējams, kopējams un ielīmējams – tekstu var iezīmēt, turot nospiestu peles pogu un pārvietojot kursoru. Šo tekstu, pēc tam, iespējams nokopēt ar CTRL+C taustiņu kombināciju un ielīmēt jebkur citur;
- Iezīmējams arī tad, ja dalīts vairākās rindās – ja sāk iezīmēt vienu teksta rindiņu, taču turpina iezīmēt arī citu, tad teksts joprojām tiek iezīmēts;
- Pieejams ekrāna lasīšanas programmatūrai – *canvas* ierakstītais teksts parasti nav pieejams "no ārpuses", tai skaitā arī programmatūrām, kuras no ekrāna nolasa tekstu, lai atvieglotu dzīvi cilvēkiem ar redzes traucējumiem. Spraudnim vajag spēt padarīt šo tekstu pieejamu šāda veida programmatūrai;
- Maināms – pēc sākotnējās teksta elementa pievienošanas, konkrētajam elementam iespējams mainīt teksta saturu, stilu, izmēru u.c. īpašības, tomēr tas joprojām saglabā iezīmējamību un pieejamību ekrāna lasītājiem;
- Animējams vai pārvietojams – tekstu pēc pievienošanas iespējams pārvietot pa *canvas* elementu un tas saglabā iezīmējamību un pieejamību ekrāna lasītājiem;

- Plašu teksta stilu klāstu atbalstošs – iespējams iezīmēt tekstu, kurš ievadīts gan ar vienplatuma fontu, kuriem visu rakstzīmju platums ir vienāds (Courier New, Consolas u.c.), gan sarežģītākiem fontiem, kuriem katras rakstzīmes platums un augstums būtiski atšķiras, kā arī ņem vērā rakstzīmju platuma un augstuma atšķirības, kuras rodas, izmantojot slīprakstu vai treknsrakstu;
- Iezīmējams ar mobilajām ierīcēm – tekstu iespējams iezīmēt un kopēt, izmantojot skāriensaskarni un mobilo ierīču pārlūkprogrammas.

Papildus prasības:

- Tīrs *JavaScript* – mērķis izveidot šo spraudni tādu, lai tas darbotos bez ārējām atkarībām vai bibliotēkām kā *jQuery*, jo šīs ārējās bibliotēkas ar laiku mainās un izmaiņas var padarīt šo teksta pieejas spraudni nelietojamu;
- Atbalsta visas modernās pārlūkprogrammas – rakstīšanas brīdī jaunākās Google "Chrome", Mozilla "Firefox", "Safari" un "Opera" pārlūkprogrammu versijas, kā arī "Internet Explorer", sākot no 10. versijas;

2. ZINĀMIE RISINĀJUMI

Problēma ar teksta pieejamību *canvas* elementos nav jaunums tīmekļa lietotņu izstrādātāju apvidū, un ir pieejami arī vairāki risinājumi, kuri vismaz daļēji tiek galā ar pētītajām problēmām – it īpaši ar teksta iezīmēšanu un ievadi, taču pilnīgu risinājumu atrast nav izdevies.

Iepriekš pētīts, ka risinājumus var iedalīt četrās kategorijās: teksta glabāšana atsevišķos DOM elementos, slēpto DOM elementu izmantošana, OCR izmantošana un teksta iezīmēšana ar *JavaScript* palīdzību [2].

2.1. Teksta glabāšana atsevišķos DOM elementos

Teksta glabāšana atsevišķos DOM elementos un vēlāka *canvas* elementa pārklāšana ar tiem, ir vienkāršākais veids HTML5 *canvas* balstītu tīmekļa lietotņu pieejamības problēmas risināšanā. Lai arī tehniski tā nav manipulācija ar *canvas* iezīmēto tekstu, pareizi izstrādājot lietotni ar šo metodi, lietotājam rodas ilūzija par to, ka teksts strukturāli ir daļa no lietotnes saskarnes (skatīt attēlu 2.1.1).

Attēls 2.1.1 **Ar <div> elementu pārklāts canvas elements**

HTML elementus, ar kuriem pārklāj *canvas*, parasti veido dinamiski, ar *JavaScript* izsaucot jauna elementa pievienošanas funkciju un pēc tam, izmantojot HTML un CSS tehnoloģijas, novietojot elementu virs *canvas*. Parasti tiek lietoti *<div>, <p>, <span>, <input>* un *<button>* elementi. Izstrādātāja ziņā ir rūpēties par to, lai elements tiek novietots īstajā pozīcijā un pielāgojas dažādiem ekrānu tipiem un to izšķirtspējām, un jānodrošina tas, lai, vajadzības gadījumā, šiem pārklājošajiem elementiem var "izklikšķināt cauri", kā arī jāpielāgojas WAI-ARIA specifikācijai, norādot laukus, kuri tiks mainīti, lai tādas ekrāna

lasītāja programmatūras kā “JAWS” vai “NVDA” zina, ka šie elementi ir papildus jāuzrauga [3].

Canvas elementa pārklāšana ar teksta elementiem risina problēmu ar teksta pieejamību pašā tās saknē, proti, teksts nemaz netiek rakstīts *canvas*, bet gan ārpus tā. Vairums HTML elementu ir brīvi pieejami ekrāna lasītāju programmatūrām un lietotāja manipulācijām, turklāt ir plašas teksta stila maiņas iespējas, pateicoties CSS. Turklāt, ir iespējams bez problēmām tekstu iezīmēt arī ar mobilajām ierīcēm, kurām ir skārienjūtīgie ekrāni.

Lai arī *canvas* elementa pārklāšana ar citiem HTML elementiem ir no izstrādes sarežģītības viedokļa visvienkāršākais risinājums teksta pieejamības risināšanā, tomēr, izmantojot šādu metodi, ir trūkumi un, lai izveidotu funkcionēt spējīgu tīmekļa lietotni, izstrādātājam ir jāņem vērā daudz nianšu. Piemēram, liels skaits dinamisku HTML DOM elementu var palēnināt tīmekļa vietnes ātrdarbību, it īpaši, ja šie elementi regulāri tiek pievienoti un noņemti no DOM koka vai arī mainīts to saturs, jo, vairumā gadījumu, pārlūkprogramma no jauna aprēķina visas lapas elementu izmērus, atrašanās vietas un citus parametrus. Lai arī ir vairākas *JavaScript* metodes teksta satura maiņai (`.html()`, `.innerHtml()`, `.append()` un `.textContent()`), tomēr to ātrdarbība ir krasi atšķirīga starp pārlūkprogrammām un to versijām [4].

Izstrādātājam arī jābūt uzmanīgam ar elementu izvietošanu, jo pat nelielas izmaiņas lapas struktūrā var būt katastrofālas pārklājošajiem elementiem. Īpaša uzmanība jāpievērš vietņu versijām priekš mobilajām ierīcēm, kuru ekrānu izmēri un satura mērogošana var veidot novirzes izkārtojumā. Turklāt jāņem vērā, ka vairumam pārlūkprogrammu ir atšķirības tajā, kā tiek attēlots saturs, kā arī, šī metode ir darboties nespējīga, ja lietotājs savā pārlūkprogrammā ir atslēdzis CSS funkcionalitāti.

2.2. Slēpto DOM elementu izmantošana

Ja variants ar teksta glabāšanu virs *canvas* novietotos DOM elementos neder lietotnes tehniskās specifikācijas dēļ, taču, kā minimums, nepieciešama teksta pieejamība ekrāna lasīšanas programmatūrai, tad ir iespēja izmantot metodi ar HTML DOM elementu slēpšanu no lietotāja.

HTML elementu slēpšana, lai tie joprojām būtu pieejami ekrāna lasītājiem, gan nav tik vienkārša kā CSS vērtību uzstādīšana uz `visibility:hidden`, `display:none`, vai `width:0px` un `height:0px`, jo šajos gadījumos, lai arī lietotājs šos elementus neredz un tie tehniski joprojām atrodas tīmekļa vietnes elementu kokā, pārlūkprogrammas un vairums ekrāna lasītāju būtībā ignorē elementus ar šādām vērtībām.

Populārākais risinājums HTML elementu slēpšanā, lai tie joprojām būtu pieejami ekrāna lasītājiem, ir to novietošana ārpus tīmekļa lapas robežām, tādējādi padarot to neredzamu lietotājam, taču saglabājot elementa funkcionalitāti dokumenta plūsmā. Ieteicamais risinājums ir konkrētajiem HTML elementiem definēt sekojošas CSS vērtības [5]:

- `position:absolute` – izņem elementu no normālās dokumenta plūsmas un ļauj to patvaļīgi novietot;
- `left:-10000px` – pārvieto saturu 10'000 pikseļus pa kreisi. Nav obligāti pārvietot tik tālu, jo galvenais ir pārvietot elementu ārpus lapas robežām;
- `top:auto` – liek pārlūkprogrammai vertikāli pozicionēt saturu tā sākotnējā atrašanās vietā. Ar dažām pārlūkprogrammām šī atribūta neizmantošanas rezultātā var gadīties, ka `left` atribūts tiek ignorēts;
- `width:1px; height:1px; overflow:hidden` – padara elementu 1x1 pikseļa izmērā un liek pārlūkprogrammai paslēpt visu, kas neietilpst šajos izmēros.

Izstrādātājs nedrīkst arī aizmirst par pareizu WAI-ARIA atribūtu lietojumu, lai slēptajos elementos esošais teksts un tā izmaiņas būtu pieejamas ekrāna lasītājiem.

Galvenais ieguvums slēpto DOM elementu lietošanā ir tajā, ka, no parastā lietotāja skatapunkta, *canvas* un attiecīgie slēptie elementi viens otram netraucē, taču cilvēki, kuriem nepieciešams ekrāna lasītāju atbalsts, varēs pilnvērtīgāk pārlūkot tīmekļa vietni. Šajos elementos arī var ievietot ne tikai *canvas* redzamo tekstu, bet gan arī kontekstam atbilstošus skaidrojumus attēlā redzamajam, piemēram, apzīmējumus grafikā. Turklāt, ja vien nav citi CSS stili, kas pārraksta jau esošās, slēpšanai paredzētās, vērtības, tad šie paslēptie elementi arī nekādi neietekmē lapas izskatu. Visbeidzot, šis risinājums arī ir piemērots ne tikai statiskām tīmekļa lietotnēm, bet arī spēlēm vai citām animētām lietotnēm, kurās ar tekstu tiek veiktas vizuālas manipulācijas, jo, atšķirībā no HTML elementiem, kuri pārklāj *canvas*, šeit teksta kustināšanā nav nepieciešamas manuālas izmaiņas elementa stilā, kuras var sabojāt tīmekļa vietnes izskatu – tekstu kontrolē *JavaScript* un *canvas*.

Trūkumi ir tajā, ka, ja pārlūkprogrammā ir pilnībā atspējota CSS darbība, tad viss paslēptais kļūst redzams un var būtiski sabojāt tīmekļa lietotnes izskatu. Lietotnei arī, visticamāk, būs nepieciešamība dinamiski izveidot HTML elementus, kurus slēpj, vai arī mainīt to tekstuālo saturu, un šīs darbības tomēr var negatīvi ietekmēt tīmekļa vietnes ātrdarbību. Tomēr galvenais trūkums ir tajā, ka teksts ir pieejams tikai ekrāna lasītājiem un to nav iespējams iezīmēt un kopēt.

2.3. OCR izmantošana

OCR tehnoloģija, ļauj atpazīt dažādas rakstzīmes attēlos, PDF failos, kameras straumējumā un citos medijos, un šīs rakstzīmes pārveidot mašīnām saprotamā un rediģējamā kodējumā. Tā kā arī *canvas* elements ir uzskatāms par attēlu, jo ir bitkartes izejas elements, tad arī tur attēloto informāciju var analizēt ar OCR algoritmiem.

Iepriekšējā pētījuma [2] autors izmēģināja šo metodi, lietojot Ocrad.js [6] spraudni, un pārbaudot funkcionalitāti pēc šāda principa:

1. Uzzīmē tekstu *canvas* elementā;
2. Palaiž teksta atpazīšanas funkciju;
3. Simbolu virkni ievieto slēptā HTML elementā;
4. Ekrāna lasītājs nolasa HTML ierakstīto tekstu.

Tika izmēģināti divi dažādi elementu slēpšanas veidi:

1. *Canvas* slēptais DOM – HTML elementu ar atpazīto tekstu ievieto <*canvas*> elementa iekšienē – vietā, kas paredzēta kā atkāpšanās iespēja gadījumiem, kad pārlūkprogramma neatbalsta *canvas* elementu – lai šo tekstu varētu nolasīt ar ekrāna lasīšanas programmatūru;
2. Ar CSS slēptie elementi – elementu ar atpazīto tekstu slēpj ar dažādām, CSS piedāvātajām iespējām.

No abiem izmēģinātajiem elementu slēpšanas veidiem, strādāja tikai slēpšana ar CSS. Elementu ievietošana *canvas* DOM struktūrā padarīja šos elementus nepieejamus ekrāna lasītājiem, neatkarīgi no uzstādītajiem WAI-ARIA atribūtiem. Otra metode darbojās un teorētiski šo risinājumu var apkopot *JavaScript* bibliotēkā un to automātiski palaist jebkurai tīmekļa vietnei ar *canvas* elementiem, izmantojot pārlūkprogrammu papildinājumus, kuri ļauj palaist ārējos skriptus, tādējādi nodrošinot iespēju nolasīt tekstu no *canvas* elementiem.

Testēšanas laikā arī tika novērots, ka OCR ievērojami paildzina programmas izpildi, tomēr autors pieminēja, ka, ka ikdienas lietošanas gadījumos sevišķa aizkave nebūtu jūtama, jo *canvas* tiek apstrādāts tikai noteiktos gadījumos, nevis nepārtraukti. Spēlēs OCR metode pilnīgi noteikti būtu nelietojama.

Tomēr šai metodei, bez iztrūkstošās iespējas iezīmēt tekstu, ir vēl viens ievērojams trūkums – teksta atpazīšana atpazīšana ir atkarīga no izmantotā fonta. Ar atpazīšanas problēmām var cīnīties, izmantojot citu OCR bibliotēku – tādu, kura spēj pati apgūt simbolus mašīnmācīšanās ceļā, taču Ocrad.js bija teju vienīgais tīra *JavaScript* risinājums, laikā, kad autors rakstīja darbu.

Teksta atpazīšana arī strādā tikai noteiktos apstākļos – ja fonā nav citu grafisko elementu un ja ir pietiekami augsts kontrasts starp fonu un tekstu, kā tas ir ar melniem burtiem uz balta fona (skatīt attēlu 2.3.1). Attēla labajā pusē, iezīmētajā laukumā, var redzēt HTML elementā ievietoto tekstu, kurš iegūts, pielietojot simbolu atpazīšanas algoritmu.

Attēls 2.3.1 **OCR metodes darbība tam ideālos apstākļos**

Tomēr, ja attēlā parādās vēl kāds grafiskais elements (skatīt attēlu 2.3.2)., tad tas tiek tulkots nepareizi. Attēlā redzamā dzīvnieka galva tika tulkota kā "_" simbols.

Attēls 2.3.2 **OCR ar grafiskajiem elementiem**

Autors secināja, ka lai arī teorētiski un, noteiktos apstākļos, arī praktiski, OCR lietošana palīdz iegūt *canvas* ierakstīto tekstu, tomēr dzīvē šāds risinājums nav derīgs, jo piemērots tikai tīri tekstuāla rakstura tīmekļa lietotnēm. Turklāt mobilajām ierīcēm – viedtālruņiem un planšetdatoriem – ekrānlasīšanas risinājumu tikpat kā nav.

2.4. Iezīmēšana ar *JavaScript* palīdzību

Šajā sadaļā apskatītie risinājumi fokusējas uz *JavaScript* izmantošanu teksta pieejamības uzlabošanā, taču vietām tiek izmantota arī dinamiski veidoti slēptie DOM elementi.

2.4.1. CanvasInput

Goldfire Studios izstrādātais "CanvasInput" *JavaScript* spraudnis nodrošina teksta ievadi caur *canvas*, imitējot vienkāršu, vienas rindas HTML teksta ievades (jeb <input>)elementu. Šis spraudnis ļauj arī iezīmēt šajā ievades laukā ierakstīto tekstu, to kopēt un ielīmēt, turklāt atbalstot CSS līmeņa teksta stila mainīšanas iespējas [7].

Šis spraudnis tika veidots kompānijas radītas tīmekļa spēles vajadzībām, tomēr tagad ir brīvi pieejams publikai. Tas ir ievērības cienīgs ar to, ka ļauj brīvi mainīt teksta stilu, kā arī ierakstīt tekstu un padot to tālāk tīmekļa lietotnei, piemēram, lai ievadītu lietotāja vārdu vai tērzētu. Izstrādātāji apgalvo, ka šis spraudnis ir "Tuvākā DOM ievades lauka imitācija *canvas*" [7] līdz šim, un patiešām – nezinātājs atšķirību, visticamāk, nemanītu. Vēl viena izcila īpašība ir ārējo bibliotēku neizmantošana – izveidē lietots tikai tīrs *JavaScript*.

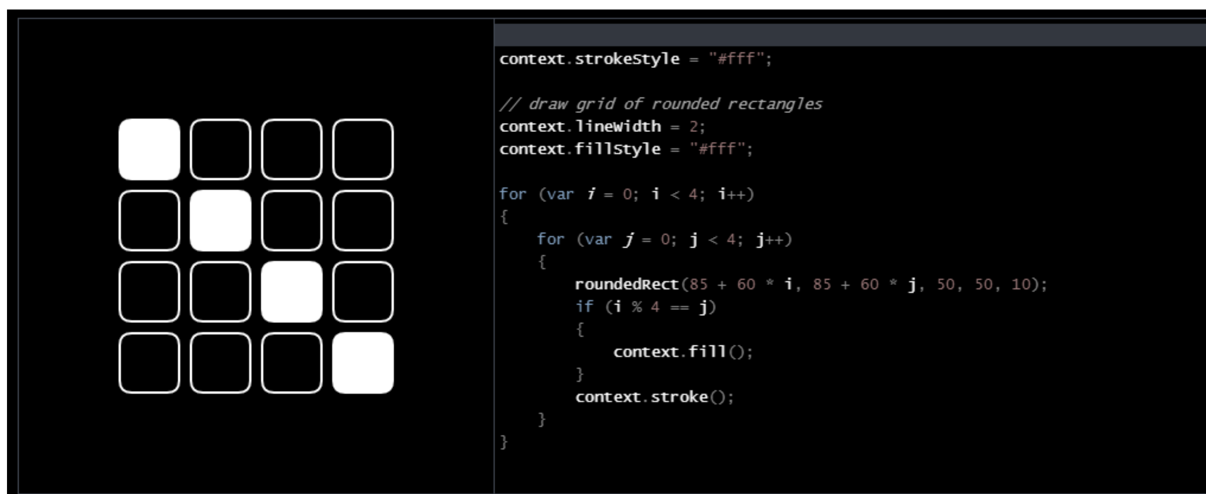
Teksta ievade tiek papildināta, izmantojot slēptus HTML ievades (*<input>*) elementus. Šie elementi palīdz noteikt iezīmēto teksta apgabalu un padara tekstu pieejamu ekrāna lasīšanas programmatūrai, pēc katra klikšķa uz *canvas* ievades reģiona, fokusējot pārlūkprogrammu uz slēpto ievades elementu.

Mobilo ierīču atbalsts ir labs. Testējot Microsoft Lumia 640 telefonā ar Windows 10 operētājsistēmu, pārlūkprogrammā "Edge", Nokia Lumia 625 telefonā ar Windows 8.1 operētājsistēmu un "Internet Explorer 11" pārlūkprogrammu, kā arī iPad planšetdatorā, gan ar "Safari", gan "Google Chrome" pārlūkprogrammām, teksts tika ierakstīts tam paredzētajos lauciņos, tomēr darbības notika jūtami lēnāk, nekā izmantojot tradicionālos teksta ievades lauku elementus. Nebija arī iespējas tekstu iezīmēt un kopēt.

CanvasInput gan atrisina tikai dažas no pētāmajām problēmām, proti, teksta iezīmēšanu un kopēšanu, tiesa, bez skārienekrānu atbalsta, ekrāna lasīšanas atbalstu un plašu stila iespēju atbalstu. Trūkst vairāku teksta rindu attēlošanas iespēja, kā arī, teksts nav animējams (ja atskaita iespēju ar CSS pārvietot *canvas* elementu pa tīmekļa dokumentu). Teksta ierakstīšana arī nav viena no pētāmajām problēmām, līdz ar to šī funkcionalitāte, lai arī labi strādā, nav nepieciešama šī darba kontekstā.

2.4.2. Textor

"Textor" [8] ir kompānijas Microsoft izstrādātāja Lutza Rēdera patvaļīgi izstrādāta tīmekļa lietotne, kas darbojas kā teksta redaktors ar *JavaScript*, HTML un CSS sintakses izcelšanas funkcionalitāti, automātiski pielāgojot teksta krāsu vai transformāciju, atkarībā no konteksta. "Textor" pirmkods arī ir brīvi pieejams, taču bez dokumentācijas vai paskaidrojumiem par programmas darbību. Tīmeklī esošajā piemērā redzams redaktorā ierakstīts koda fragments, kurš izpilda zīmēšanu blakus esošajā *canvas* reģionā. Turklāt kodu pamainot, reāllaikā tiek izmainīts arī attēlotais zīmējums (skatīt attēlu 2.4.2.1).



Attēls 2.4.2.1 **Textor lietotne**

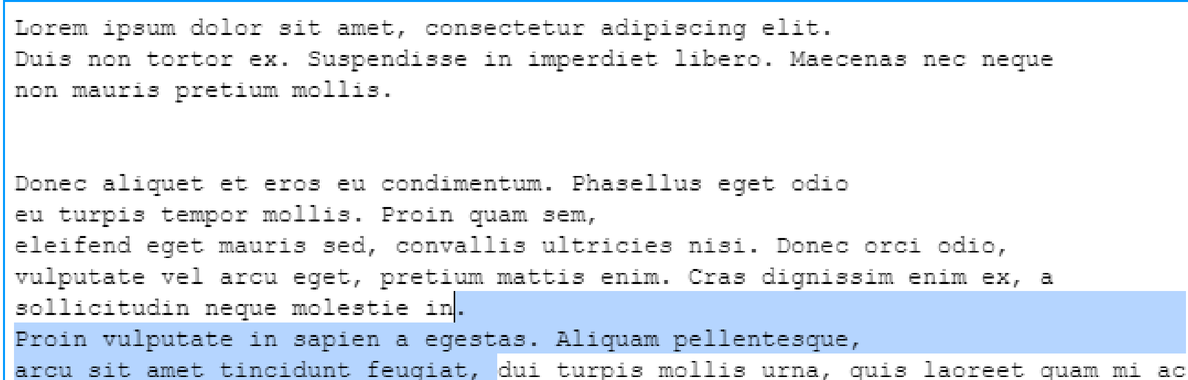
Atšķirībā no "CanvasInput", "Textor" atbalsta teksta rakstīšanu vairākās rindiņās, pieļaujot pat atkāpju ievietošanu. Tekstu var iezīmēt, kopēt, griezt un ielīmēt turklāt, kopējot teksta fragmentu ar atkāpēm un to pēc tam ielīmējot citur, piemēram, programmā "Notepad", atkāpes saglabājas. Turklāt arī šeit iespējams tekstu ievadīt un dzēst, bet ne iezīmēt, ar visām testētajām ierīcēm un pārlūkprogrammām (Lumia 625 ar IE 11, Lumia 640 ar "Edge", iPad ar "Chrome" un "Safari").

Iztrūkst vairākas būtiskas īpašības, kuras nepieciešamas esošās problēmas risināšanai. Teksts nav pieejams ekrāna lasīšanas programmatūrai, nav animējams (vismaz par šādu funkcionalitāti nekas nav minēts) un lietotne atbalsta tikai vienplatuma fontus, jeb fontus, kuriem visas rakstzīmes ir vienlīdz platas, piemēram, Courier, Courier New, Monaco, Consolas. Šis piemērs arī ir tīmekļa lietotnes, kas nav veidota ar tīru *JavaScript*, bet gan kopā ar *TypeScript* paplašinājumu, formātā, nevis kā spraudnis.

2.4.3. Canvas Text Editor

"Canvas Text Editor" [9], pēc būtības, ir vienkāršs teksta redaktors, kurš veidots ar *JavaScript* un *canvas* starpniecību. Lai arī redaktors, pamatā, ir pamācība tam, kā izveidot teksta redaktoru *canvas* vidē, šī pamācība ir noderīga arī šeit pētāmās problēmas risināšanā. Pamācībā ir aprakstīts, kā uzprogrammēt funkcionalitāti, kuru parasts tehnoloģiju lietotājs sen uzskata par pašsaprotamu, proti: no kādām datu struktūrām sastāv teksts, fontu metrikas, teksta iezīmēšana (tiesa, ne ar peles kursoru), teksta dalīšana vairākās līnijās, kopēšana un ielīmēšana, teksta ritināšana un vēl.

Esošais demonstrācijas risinājums, protams, ir ļoti vienkāršots un bez paplašinātām iespējām (skatīt attēlu 2.4.3.1). Piemēram, nedarbojas peles rullītis, nevar iezīmēt ar peles kursoru, netiek atbalstīti dažādu tipu fonti, dalīšana līnijās nenotiek automātiski un vēl; tomēr kā piemērs, šī lietotne (un pamācība) ārkārtīgi uzskatāmi parāda to, ka veikt manipulācijas ar tekstu *canvas* vidē ir iespējams – tas tikai prasa papildus pūliņus, kuri, iespējams, nav vajadzīgi teksta redaktoru kontekstā, jo esošie bezsaistes risinājumi ir pietiekamā skaitā.



Attēls 2.4.3.1 **Canvas Text Editor lietotne**

No darbā aprakstītās problēmas risinājuma kritērijiem, šī lietotne izpilda vien dažus no vajadzīgajiem kritērijiem: ir rudimentāra teksta iezīmēšana (arī vairākās rindās), risinājums

bez ārējo bibliotēku izmantošanas, tekstu iespējams kopēt un ielīmēt, kā arī mainīt teksta saturu. Tomēr nav iespējas tekstu animēt, vai iezīmēt ar peles kursora palīdzību. Nav arī atbalsta ekrāna lasīšanas programmatūrai, mobilajām ierīcēm, kā arī lietotne atbalsta tikai vienplatuma fontus.

2.4.4. Esošo *JavaScript* risinājumu salīdzinājums

Iepriekš aprakstītie bija vienīgie vērā ņemamie piemēri *canvas* zīmētā teksta pieejamības risināšanā ar *JavaScript* palīdzību. Lai arī neviens no tiem nerisina esošo problēmu ar teksta pieejamību iezīmēšanai, kopēšanai un ekrāna nolasīšanai vienkāršās tīmekļa lietotnēs, tomēr katrs no risinājumiem sniedz vismaz daļēju ieskatu problēmas risinājuma variantos (skatīt tabulu 2.4.4.1).

Tabula 2.4.4.1.

***Canvas* teksta apstrādes risinājumu salīdzinājums**

Kritērijs	CanvasInput	Textor	Canvas Text Editor
Ir spraudnis	Jā	Nē	Nē
Tīrs *JavaScript*	Jā	Nē	Jā
Peles atbalsts	Jā	Jā	Nē
Var iezīmēt	Jā	Jā	Jā
Var kopēt	Jā	Jā	Jā
Var ielīmēt	Jā	Jā	Jā
Dalīšana līnijās	Nē	Jā	Jā
Maināms saturs	Jā	Jā	Jā
Animējams	Nē	Nē	Nē
Citu stilu atbalsts	Jā	Nē	Nē
Ekrānlasāms	Jā	Nē	Nē
Mobilo ierīču atbalsts	Daļējs	Jā	Nē

No esošajiem risinājumiem vislielākais vajadzīgo īpašību atbalsts ir “CanvasInput” spraudnim, tomēr tas pēc būtības pilda citu funkciju – teksta ievadi – nevis tā izgūšanu no *canvas* elementa.

3. JAVASCRIPT SPRAUDŅA TEKSTA IEGUVEI DARBĪBAS PRINCIPI

Lai izveidotu darbojošos spraudni, kurš spējīgs tekstu iezīmēt, kopēt, pārvietot, kā arī padarīt pieejamu ekrāna lasīšanas programmatūrai un ierīcēm ar skārienekrānu saskarni, nepieciešams ievērot vairākas būtiskas prasības un darbības principus.

Darbības principi sadalīti prasībās par teksta elementu, prasībās par teksta pieejamību ekrāna lasītājiem un pamata funkciju darbības principos.

3.1. Prasības pret datiem par teksta elementu

Teksts, kurš uzzīmēts *canvas* elementā, kļūst nepieejams no ārpuses, un faktiski tiek attēlots kā bitkartes attēls, tāpēc visa informācija par zīmēto tekstu ir atsevišķi jāglabā kā datu objekts, pretējā gadījumā, pat ja zīmē tekstu, to ievietojot mainīgajā, mainīgais nekādu informāciju neuzglabā (skatīt attēlu 3.1.1).

```
> var canvas = document.getElementById("canvas");
  var ctx = canvas.getContext("2d");
  var helloWorld = ctx.fillText("Hello, world!", 50, 50);
<· undefined
> helloWorld
<· undefined
>
```

Attēls 3.1.1 **Mēģinājums tekstu glabāt mainīgajā (Opera 37.0 izstrādātāja režīms)**

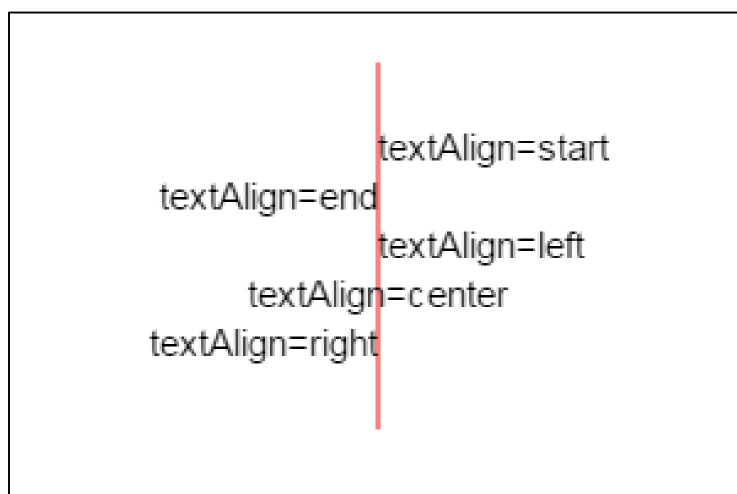
Lai informācija par zīmēto tekstu tiktu saglabāta, nepieciešams to piešķirt vienam vai vairākiem mainīgajiem, piemēram, bet ne tikai, izveidojot *JavaScript* objektu (skatīt attēlu 3.1.2).

```
> var canvas = document.getElementById("canvas");
  var ctx = canvas.getContext("2d");
  var helloWorld = {string: "Hello, world!", x:50, y:50};
  ctx.fillText(helloWorld.string, helloWorld.x, helloWorld.y);
<· undefined
> helloWorld
<· Object {string: "Hello, world!", x: 50, y: 50}
>
```

Attēls 3.1.2 **Datu objekts ar teksta parametriem (Opera 37.0 izstrādātāja režīms)**

Svarīgākā prasība, lai *canvas* zīmēto tekstu varētu iegūt, ir spēja saglabāt visu teksta zīmēšanai svarīgo informāciju un parametrus pārējam skriptam pieejamos mainīgajos. Zīmējot pieejamu tekstu, svarīgi ir šādi parametri, vairumu no kuriem lietotājs padod konstruktora funkcijai:

- Teksts – simbolu virkne, kura tiks zīmēta. Piemēram, "Sveika, pasaule!";
- Atrašanās vietas koordinātas – skaitļu pāris, kurš apzīmē X un Y koordinātas attiecībā pret *canvas* elementa augšējo, kreiso stūri. X koordināta ir teksta līdzināšanas atskaites punkts un Y koordināta ir teksta bāzlīnijas atrašanās punkts;
- Teksta līdzinājums [10] – maina teksta elementa X koordinātas nozīmi, padarot to vai nu par teksta sākuma punktu, beigu punktu vai viduspunktu, faktiski mainot teksta pozīciju *canvas* elementā (skatīt attēlu 3.1.3). Pēc būtības līdzīgs CSS "text-align" parametram;



Attēls 3.1.3 **Teksta līdzinājums *canvas*. Līnija apzīmē teksta X koordinātu**

- Teksta bāzlīnija [11] – teksta elementa pamatne. Atkarībā no uzstādītās vērtības, maina teksta Y koordinātas nozīmi – var būt teksta apakša, augša, viduspunkts un vēl (skatīt attēlu 3.1.4).

Attēls 3.1.4 ***Teksta bāzlīnijas veidi***

- Fonta izmērs – skaitlis pikseļos, kurš apzīmē fonta izmēru. Piemēram "30px";

- Fontu saime – apzīmējums izmantojamajai fontu saimei CSS tipa pieraksta veidā. Piemēram “Arial”, “Courier New” vai “Comic Sans MS”;
- Fonta stils – slīpraksts (“italic”), ieslīps (“oblique”) vai normāls (“normal” teksts);
- Fonta variants – normāls (“normal”) vai mazi lielie burti (“small-caps”);
- Fonta biezums – teksta biezums, atbilstoši CSS definīcijām. Piemēram “normal”, “bolder”, “700”;
- Teksta platums – zīmētā teksta platums pikseļos atbilstoši tam, kā tas tiek uzzīmēts, ņemot vērā teksta garumu un visus piemērotos stila parametrus, kuri ietekmē teksta izskatu (treknums, fonta izmērs utt.). Iegūst, ar *canvas* `measureText()` metodi;
- Līnijas augstums – vienas teksta rindas augstums ar konkrētajiem fonta izmēra un saimes parametriem;

Šie parametri ir nepieciešami, lai varētu noteikt teksta elementa robežas un noteikt, vai kursors atrodas virs šī elementa vai pat konkrēta simbola.

Papildus teksta izmēru mainošajiem parametriem, nepieciešams glabāt arī teksta aizpildījumu, kuram ir pilnībā kosmētiska nozīme. Tas var būt krāsa, kura definēta kā krāsas nosaukums (piemēram “blue” vai “red”), heksadecimālais kods (piem. “#E5E5D4” gaiši pelēkai krāsai), vai RGB (piem. “rgb(0,255,0)” zaļai krāsai. Var būt arī lineārs vai radiāls gradienta objekts vai arī iepriekš izveidots raksta objekts jeb attēls.

3.2. Prasības teksta pieejamībai ekrāna lasītājiem

Ekrāna lasītāji darbojas pēc principa, ka pārlūko nevis redzamo tīmekļa vietnes daļu, bet gan nolasa lapas DOM koka struktūru, lai iegūtu informāciju par visiem lapā attēlotajiem elementiem. Arī *canvas* ir šāds elements un ekrāna lasītāji tam spēj piekļūt, taču nav iespējas nolasīt to, kas attēlots *canvas* elementā, jo tas ir tikai zīmējums.

Lai varētu risināt pieejamības problēmu, ir vajadzīgs:

- Glabāt datus par zīmēto objektu;
- Padarīt zīmētos objektus pieejamus atlasīšanai ar klaviatūru;
- Fokusējoties uz objektu, tos speciāli iezīmēt (apvilkt kontūru);

WAI-ARIA specifikācija [3] nosaka to, kā padarīt tīmekļa vidi pieejamu cilvēkiem ar īpašām vajadzībām, kas ietver gan redzes invalīdus, gan cilvēkus, kuri nav spējīgi pārlūkot

tīmekļa vietnes, pilnvērtīgi izmantojot tradicionālās ievadierīces un paļaujas uz, piemēram, tikai klaviatūras izmantošanu pārlūkošanā.

Lai *canvas* zīmētais teksts būtu pieejams, ir obligāti jāveido hibrīdais risinājums starp *canvas* un citiem HTML elementiem, katram *canvas* zīmētajam tekstam izveidojot HTML elementu (piemēram, *<span> vai <textarea>*), kurš glabā to pašu tekstuālo saturu, kāds rādās zīmēts *canvas*. Šos HTML teksta elementus var veidot vai nu ārpus *canvas* elementa un tos slēpt ar CSS palīdzību, vai arī ievietot *canvas* elementa iekšējā apakškokā (angliski dēvēts par "sub-DOM").

Lai arī nu jau vairums pārlūkprogrammu pilnībā atbalsta *canvas* iespējas, HTML5 standarta veidotāji ir rēķinājušies ar vecākām pārlūkprogrammām, kuras to neatbalsta un izveidojuši atkāpšanās sistēmu (fallback), kura pārsvarā tiek izmantota, lai ziņotu lietotājam par pārlūkprogrammas iespēju trūkumu, gluži kā alternatīvais teksts attēliem gadījumos, ja attēls netiek ielādēts.

Pārlūkos, kuri atbalsta *canvas*, lietotājs redzētu ar *JavaScript* izveidotos attēlus/animācijas, pretējā gadījumā *<canvas>* elements tiek ignorēts un tā vietā parāda saturu, kurš ievietots elementa iekšienē, piemēram, `<canvas><h1>Hello, world!</h1></canvas>`, attēlos tekstu "Hello, world!", ja *canvas* elementu neatbalsta.

Šāds rezerves saturs, kurš var iekļaut arī attēlus, sarakstus un teju jebkādu citu HTML elementu, parasti tiek slēpts, ja pārlūkprogramma atbalsta *canvas*, taču tas joprojām ir pieejams no ārpuses, līdz ar to arī nolasāms tādām palīgprogrammām kā ekrāna lasītājiem.

Izveidotajiem elementiem, neatkarīgi no to atrašanās vietas lapas DOM, jāiestata "tabindex" atribūts, kurš ļauj pārslēgties starp elementiem, izmantojot klaviatūru, un uz tiem fokusēt no programmas izpildes puses. Šis atribūts arī nosaka pārslēgšanas un atlasīšanas prioritāti starp elementiem [12].

Tāpat nepieciešams uzstādīt WAI-ARIA atribūtus. No tiem nozīmīgākie, šā spraudņa kontekstā, ir:

- aria-hidden – būla vērtības atribūts, kurš nosaka to, vai konkrētais elements ir redzams ekrāna lasīšanas programmatūrai vai nē;
- aria-live – atribūts, kurš nosaka teksta nolasīšanas prioritāti. Pēc noklusējuma ir atslēgts, taču var iestatīt uz "polite" – izsaka tekstu pie pirmās pieklājīgās iespējas, piemēram, kad lietotājs beidzis rakstīt tekstu vai nolasīts iepriekšējais teksts, un "assertive" – nolasa tekstu tūlīt pēc izmaiņu veikšanas tajā, pārtraucot iepriekšējo lasījumu;
- aria-atomic – būla atribūts, kurš nosaka to, vai teksta izmaiņu gadījumā nolasa visu teksta elementā esošo saturu vai tikai tā mainīto daļu.

3.3. Pamata funkciju darbības principi un prasības

Sakarā ar to, ka *canvas* ir grafiskais elements bez teksta iebūvētām teksta atlasīšanas, ekrāna lasīšanas un kopēšanas iespējām, ar to saistītās funkcijas ir jāveido no nulles. Šī funkcionalitātes apraksts ir svarīgs spraudņa darbības izprašanā. Ar vārdu "kursors" turpmākajā aprakstā jāsaprot arī skāriena punkts skārienekrāna saskarnes kontekstā.

3.3.1. Teksta zīmēšana

Teksta zīmēšanu vajag realizēt, nevis tiešā veidā izsaucot `context.fillText()` metodi, kura zīmē tekstu, bet gan caur teksta elementa objekta konstruktoru, kurš saglabā datus par konkrēto elementu, sagatavo to pieejamībai un tikai tad zīmē uz *canvas* ar `context.fillText()` metodi.

3.3.2. Teksta elementa noteikšana

Tā kā *canvas* elementā ievietotajiem objektiem nav iespējams pievienot atsevišķus notikumu uztvērējus, nepieciešams pievienot vienu kursora kustības notikuma uztvērēju visam *canvas* elementam un, vadoties pēc kursora koordinātām, noteikt, vai un kura teksta elementa robežās kursors atrodas. To var noteikt, pēc katras kursora kustības veicot pilnu *canvas* pievienoto teksta elementu pārlasi un pārbaudot, vai kursora koordinātas atrodas *canvas* apgabalā sākot no teksta sākuma koordinātām līdz beigu koordinātām gan uz X, gan Y asīm.

3.3.3. Konkrēta simbola noteikšana

Lai noteiktu simbolu, virs kura atrodas kursors, pirmkārt nepieciešams zināt teksta elementa platumu pikseļos un tā atrašanās vietu. Pēc tam, kad noteikts, ka kursors atrodas virs šī teksta elementa, jāaprēķina kursora atrašanās vietas koordinātas teksta elementa iekšienē, kur koordinātu X ass nulles punkts ir teksta elementa sākums. Kad tas izdarīts, ņem teksta pirmo elementu un izmēra tā platumu ar iebūvēto `context.measureText()` metodi. Ja kursora X koordinātas vērtība ir mazāka vai vienāda par simbola platumu, tad kursors atrodas virs šī simbola. Ja kursora X koordināta joprojām lielāka, tad mērāmajai teksta daļai pievieno nākamo simbolu un nosaka abu šo simbolu kopējo platumu. Ja šoreiz kursora koordināta

iekļaujas simbolu platumā, tad ņem šo pēdējo mērāmajai daļai pievienoto simbolu. Ciklu veic, līdz atrasts īstais simbols.

Tā kā tikai vienplatuma fontiem ir savstarpēji vienādi rakstzīmju platumi, un pārējos fontos, atkarībā no stila un fonta izmēra, savstarpējie simbolu platumi var būt ļoti dažādi, drošākais veids, kā atrast simbolu, virs kura novietots kursors, ir veikt simbolu virknes pilno pārlasi, sākot no nulles punkta jeb virknes pirmā simbola, nevis mēģināt veidot algoritmu, kurš aptuveni nosaka simbolu, balstoties uz vidējo simbola platumu tekstā un kursora atrašanās vietu. Šāda sistēma, visticamāk, ļoti bieži sāktu meklēt no simbola, kurš atrodas jau aiz kursora.

Līdz ar iegūto simbolu jāpievieno arī metadati par tā atrašanās vietu teksta elementā, teksta elementa piederību utt.

3.3.4. Simbolu iezīmēšana

Iezīmēšana notiek tikai tad, ja tiek turēta peles poga vai turēts skāriens uz ekrāna, vai veikts dubultklikšķis. Pēc tam, kad nosaka, virs kura simbola atrodas kursors, to pievieno speciālam masīvam, kurš glabā iezīmētos simbolus.

Turpmākajiem simboliem, kurus iezīmē, skatās, vai tas atrodas blakus masīvā jau esošajam simbolam – tādā gadījumā iezīmē arī šo simbolu. Ja simbols, virs kura veic kustību ar nospiestu peles pogu, jau atrodas iezīmēto simbolu masīvā, to vajag ignorēt un dzēst tur jau esošo simbolu, jo šāda kustība nozīmē iezīmēšanas atcelšanu.

Ja nākamais simbols, kuru mēģina iezīmēt, ir no tā paša teksta elementa, taču ir izlaisti elementi starp šo un pēdējo iezīmēto simbolu, tad tas nozīmē to, ka, iezīmēšanas laikā, kursors ir izgājis no teksta elementa robežām. Nepieciešams noteikt trūkstošos simbolus un pievienot to iezīmēto simbolu masīvam.

3.3.5. Vārda tūlītēja iezīmēšana

Vārda tūlītēju iezīmēšanu nepieciešams veikt, ja tiek konstatēts dubultklikšķa notikums virs kāda no teksta elementiem. Šajā gadījumā, konkrētā teksta elementa simboli tiek viens pēc otra ievietoti iezīmēto simbolu masīvā.

3.3.6. Vairāku līniju un teksta elementu iezīmēšana

Gadījumos, kad kursors, iezīmēšanas laikā, tiek pārvietots no viena teksta elementa robežām uz cita, nepieciešams iezīmēt visu iepriekšējā teksta elementa saturu, jeb secīgi ievietot simbolus to glabāšanai paredzētajā masīvā.

Jānodrošina arī iezīmēšana tekstā, uz kuru pāriets, iezīmējot visus simbolus no teksta elementa sākuma līdz kursora atrašanās vietai.

3.3.7. Iezīmēto elementu glabāšana vai izvade pareizā secībā

Tā kā pastāv iespēja iezīmēt tekstu arī no labās uz kreiso pusi, tad simboli to glabāšanas masīvā arī tiktu ievadīti spoguļattēlā. Nepieciešams nodrošināt mehānismu masīva kārtošanai vai nu iezīmēšanas brīdī vai pēc tās beigām, vai mirklī, kad teksts tiek kopēts.

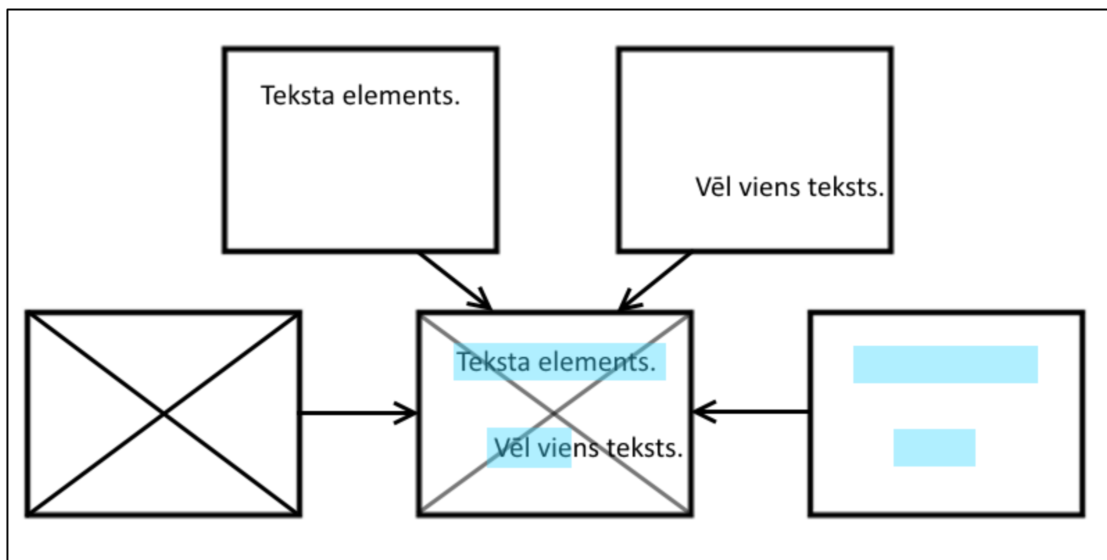
3.3.8. Iezīmējuma iekrāsošana

Iezīmēto elementu iekrāsošanai jāizmanto *canvas* `context.fillRect()` metode, kura zīmē iekrāsotus, daļēji caurspīdīgus taisnstūrus. Elementu iekrāsošana jāveic katram atsevišķajam simbolam, no simbola metadatiem iegūstot nepieciešamo taisnstūra platumu un augstumu, kā arī, iekrāsošanas funkcija jāizsauc katru reizi, kad simbolu glabāšanas masīvs tiek papildināts.

Tomēr ar iekrāsošanu ir viena būtiska problēma – katru reizi, kad tiek izsaukta iekrāsošanas funkcija, nepieciešams arī nodzēst jau esošo krāsojumu. Tā kā *canvas* zīmējumos, dzēšot daļu no attēlotā, šajā gadījumā – krāsojuma - tiek dzēsts arī tas, kas atrodas fonā, tad pie katras iezīmēšanas funkcijas izsaukšanas vajadzētu pilnībā pārzīmēt arī visus pārējos *canvas* attēlotos objektus, kas var būt neērti no izstrādātāja puses un var ciest lietotnes veiktspēja. Ir divas iespējas, kā šo problēmu risināt:

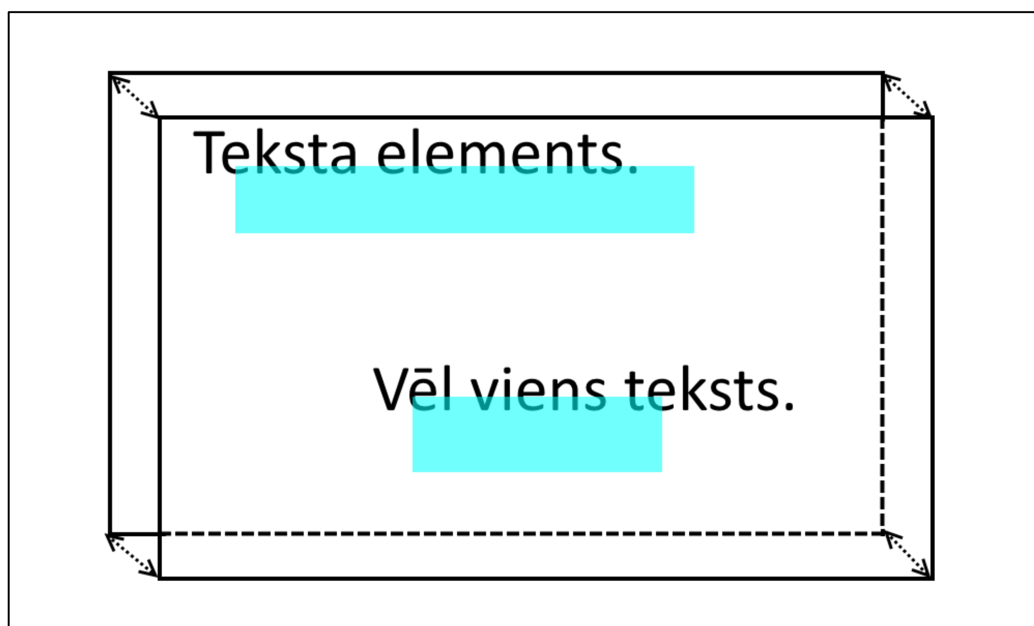
1. Izmantojot vairākus *canvas* elementus, kuri atrodas ārpus DOM [13], lai veidotu vienotu kompozīciju (skatīt attēlu 3.3.8.1). Šī metode, arī zināma kā ārpus ekrāna atveidošana (no angļu valodas “offscreen rendering”) strādā tā, ka galvenā lietotnes vizuālā daļa, katrs teksta elements un iezīmējuma reģions tiek katrs zīmēti uz sava, atsevišķa *canvas* elementa, taču to dara atmiņā, nevis uz ekrāna, jo šie *canvas* elementi netiek pievienoti tīmekļa vietnes DOM. Pēc tam, izmantojot `context.getImageData()` metodi, iespējams katra neredzamā *canvas* atveidoto attēlu kopēt un secīgi ielīmēt centrālajā *canvas* elementā ar

`context.putImageData()` metodi. Tādēji izvairās no tā, ka no jauna jāzīmē pārējie objekti bez iezīmējuma reģiona, jo var no jauna kopēt jau esošos. Šāda pieeja arī nekaitē lietotnes veiktspējai vai pat to uzlabo.



Attēls 3.3.8.1 **Attēla salikšana no vairākiem ārpus DOM *canvas* elementiem**

2. Var galveno *canvas* elementu, kurā notiek lietotnes un, tai skaitā, arī pieejamībai gatavā teksta, attēlošana un to pārklāt ar vēl vienu *canvas* elementu, kurā iekrāso iezīmētos reģionus (skatīt attēlu 3.3.8.2).



Attēls 3.3.8.2 ***Canvas* elementa pārklāšana ar vēl vienu *canvas* elementu**

Šādā gadījumā, iekrāsojot un dzēšot iezīmējuma laukumus, galvenais *canvas* elements vispār netiek aizskarts. Protams, ir savlaicīgi jāpadomā par to, lai elements vienmēr pārklātos arī tad, ja tiek mainīts pārlūkprogrammas loga izmērs un elementi

tiek nobīdīti. Vēl viena redzama *canvas* elementa zīmēšana arī var radīt negatīvas sekas uz ātrdarbību, it īpaši mazāk jaudīgos datoros un/vai telefonos.

3.3.9. Teksta elementa jaunināšana

Jāvar caur atsevišķu funkciju padot izmaiņas zīmētajā tekstā, piemēram, zīmējamos simbolus, fonta izmēru, saimi, bāzlīniju utt. Jauninot elementu, tas tiktu uzzīmēts tā jaunajā versijā līdz ar nākamo reizi, kad tiks izsaukta zīmēšanas metode.

3.3.10. Pielāgošanās līdzinājuma un bāzlīnijas izmaiņām

Teksta līdzinājums un bāzlīnija maina objekta attēlojuma pozīciju, taču programmas atmiņā koordinātas paliek tāda, kādas tiek padotas sākumā (skatīt attēlu 3.3.10.1). Piemēram, tekstam, kura X koordināta ir 50, mainot līdzinājumu uz tādu, ka teksts tiek zīmēts sākot no 30. pikseļa uz X ass, piemēram, tekstu centrējot, teksta objekta mainīgais X koordinātei joprojām saturētu vērtību "50".

Attēls 3.3.10.1 **Teksta līdzināšanas ietekme uz attēlojumu pret atmiņā glabāto**

Atkāpe no līdzinājuma un bāzlīnijas noklusējuma vērtībām rada situāciju, kad teksts ir zīmēts vienā vietā, taču visi notikumu uztvērēji un iezīmēšana joprojām uzskatītu, ka teksts atrodas atmiņā esošajā pozīcijā, līdz ar to nevarētu iezīmēt tekstu tur, kur tas zīmēts, bet gan *canvas* reģionos, kuri izskatās tukši.

Lai šo problēmu risinātu, vajadzīgas funkcijas, kuras ņem lietotāja padotās bāzlīnijas vai līdzinājuma vērtības un, tā vietā, lai šīs vērtības piemērotu zīmējamajam tekstam, vienkārši

maina teksta X un Y koordinātu vērtības, lai novietojums būtu tāds pats, kā tad, ja ļautu mainīt līdzināšanas un bāzlīnijas uzstādījumus.

3.3.11. Kopēšana

Lai iezīmēto tekstu varētu kopēt, jāpārlasa visi teksta elementa simbolu objekti un jāizveido viena simbolu virkne.

Iezīmētā teksta kopēšanu uz starpliktuvi jāvar veikt ar CTRL+C un CMD+C (MAC datoriem) taustiņu kombinācijām, kā arī jāizveido uzlecoša kontekstuālā poga, kuru noklikšķinot, teksts arī tiek kopēts. Poga vajadzīga, lai lietotājs saprastu, ka tekstu iespējams kopēt, kā arī, lai varētu kopēt lietotāji, kuri izmanto skāriensaskarni, jo viņi gluži vienkārši nevar nospiest CTRL+C vai CMD+C taustiņu kombināciju un operētājsistēmas iebūvētā kopēšanas konteksta poga nerādītos, jo *canvas* nav teksta elements.

3.3.12. Sagatavošana pieejamībai

Katram jaunam teksta elementam nepieciešams izveidot jaunu HTML elementu, kurš glabā šo pašu tekstu, un elements jāievieto *canvas* iekšējā DOM. Tāpat jānodrošina, lai ekrāna lasītājs piekļūst iezīmētajam tekstam brīdī, kad iezīmēšana pabeigta vai arī teksts nokopēts.

Izveidotajiem HTML elementiem jādefinē "tabindex" un attiecīgie WAI-ARIA atribūti.

3.3.13. Dublikātu atmešana

Lai lietotājs nevarētu vairākas reizes iezīmēt vienu un to pašu vārdu vai simbolu, piemēram, starp diviem teksta elementiem virzot kursoru, tos iezīmējot un gribot atsaukt iezīmēšanu, nepieciešama arī pārbaude pret dublikātiem iezīmēto simbolu glabātuvē. Šo pārbaudi vajadzētu veikt pēc katras simbola pievienošanas glabātuves masīvā, lai varētu arī korekti iekrāsot iezīmētos simbolus.

3.3.14. Izveidoto elementu iznīcināšana

Izsaucot pieejamā teksta iznīcināšanas funkciju, jādzēš visi ar tekstu saistītie elementi – ārpus ekrāna esošie *canvas* elementi, saistītie datu objekti un to vērtības jāiestata uz *null*, jānoņem notikumu uztvērēji un teksta elementa mainīgais jāatzīmē kā pieejams pārlūkprogrammas atmiņas atbrīvošanas procedūrām.

4. IZVEIDOTĀ PROGRAMMATŪRA

Darba rezultātā ir izdevies radīt "CanvasTextAccess" [14] - *JavaScript* spraudni, ar kura palīdzību *canvas* zīmētais teksts ir pieejams iezīmēšanai gan ar peli, gan skārieniem, turklāt tekstu padara pieejamu ekrāna lasīšanas programmatūrai. Diemžēl, padarīt pieejamu tekstu, kurš ir uzzīmēts jau iepriekš ar citu spraudņu vai iebūvēto `canvas.fillText()` funkcionalitāti, nav iespējams tādēļ, ka dati par zīmēto tekstu nesaglabājas programmas atmiņā.

Spraudnis testēts sekojošajās operētājsistēmas Windows 7 pārlūkprogrammās:

- Opera 37;
- Google Chrome 50;
- Mozilla Firefox 45;
- Internet Explorer 11.

Spraudnis testēts uz sekojošajām mobilajām ierīcēm un pārlūkprogrammām:

- Samsung Galaxy S3 viedtālrunis (OS versija – Android 5.0.1, pārlūkprogrammas versija – Chrome 50);
- Nokia Lumia 625 viedtālrunis (OS versija – Windows phone 8.1, pārlūkprogrammas versija – Internet Explorer 11);
- Microsoft Lumia 640 viedtālrunis (OS versija – Windows phone 10, pārlūkprogrammas versija – Microsoft Edge 25);
- Apple iPad planšetdators (OS versija – iOS 8.1.1, pārlūkprogrammas versija – Chrome 40).

Ekrāna lasīšanas funkcionalitāte pārbaudīta ar programmatūru "NVDA" [15] un "ChromeVox" [16] spraudni Google Chrome pārlūkprogrammai

Izstrādes laikā konstatētas dažas atšķirības funkcionalitātē starp dažādām sistēmām:

- Internet Explorer un Mozilla Firefox pārlūkprogrammās nedarbojas dubultklikšķa notikums;
- Safari pārlūki uz MAC un iOS ierīcēm nekopē tekstu;
- CSS "pointer-events" parametrs, kurš ļauj "klikšķināt cauri" *canvas* pārklājam, netiek atbalstīts Internet Explorer versijās, kuras vecākas par IE 11.

Nepietiekamu resursu dēļ nebija iespējams testēt spraudni uz plašāka ierīču klāsta.

4.1. CanvasTextAccess darbība

Spraudnis neizmanto jebkādas papildus bibliotēkas un rakstīts, izmantojot tīru *JavaScript.* Tas sastāv no divām klasēm:

- `textAccessCanvas()` – sagatavo HTML esošo *canvas* elementu darbam. Šī klase satur visu iezīmēšanas, krāsošanas, notikumu uztvērēju un kopēšanas funkcijas, kā arī glabā informāciju par visiem šim *canvas* elementam pievienotajiem tekstiem;
- `textAccessElement()` – zīmē jauno tekstu, saglabājot visus nepieciešamos tā parametrus, lai nodrošinātu vēlamo stila attēlojumu un nodrošinātu korektu iezīmēšanu.

4.1.1. Teksta izveidošana

Lai izveidotu jaunu, pieejamu teksta elementu, vispirms nepieciešams HTML kodā jau esošs *canvas* elements (skatīt attēlu 4.1.1.1).

```
<canvas id="accessCanvas2" width="400" height="250"
style="border:1px solid #d3d3d3;"></canvas>
<script>
var canvas = document.getElementById("accessCanvas");
var ctx = canvas.getContext("2d");
var img = document.getElementById("bg");
ctx.drawImage(img, 0, 0, 400, 250);
</script>
```

Attēls 4.1.1.1 **Piemēra kods HTML *canvas* ar uzzīmētu fonu izveidei**

Pēc tam var izsaukt `textAccessCanvas()` konstruktoru un tad, padodot tikko izveidoto objektu, var izveidot jaunu teksta elementu, kurš pieejams iezīmēšanai, kopēšanai un ekrāna lasītājiem, ar `textAccessElement()` konstruktoru (skatīt attēlus. 4.1.1.2 un 4.1.1.3).

```
var access = new textAccessCanvas("accessCanvas", "universal", true,
"red");
var text = new textAccessElement(access, "Sveika, pasaule!", 200, 125,
"32pt", "Courier New", "italic", "small-caps", "bold", "#BB1188",
"center", "middle");
```

Attēls 4.1.1.2 **CanvasTextAccess konstruktoru izsaukšana**

Parametri `textAccessCanvas()` konstruktoram ir sekojoši:

- Norāde uz *canvas* elementu. Var būt gan elementa identifikators, gan *canvas* konteksts. Ja vērtība netiek padota, konstruktors atrod pirmo *canvas* elementu dokumentā un izmantos to;
- Ekrāna lasīšanas iestatījums. Iespējamās vērtības:
  - null/false – atspējo ekrāna lasīšanas atbalstu;
  - "local" – nolasa *canvas* attēloto tekstu tikai pēc tieša klikšķa uz tā un tikai konkrēto tekstu;
  - "global" – pēc klikšķa *canvas* reģionā, nolasa visu tur izveidoto tekstu;
  - "universal" – "local" un "global" apvienojums. Nolasa klakšķināto tekstu un pēc tam arī visu pārējo *canvas* saturu.
- Būla vērtība skāriensaskarnes iespējošanai vai atspējošanai;
- Simbolu virkne - iezīmētā reģiona iekrāsošanas krāsas iestatīšana. Pieņem krāsu gan kā tās nosauku, gan heksadecimālu vērtību, gan krāsu pēc RGB formāta. Var arī izveidot gradientu vai izmantot attēla objektu. Noklusējuma iekrāsojums ir gaiši zils.

Parametri `textAccessElement()` konstruktoram ir sekojoši:

- Norāde uz iepriekš izveidoto `textAccessCanvas()` elementu;
- Simbolu virkne – zīmējamais teksts;
- Skaitlis – teksta novietojuma X koordināta;
- Skaitlis – teksta novietojuma Y koordināta;
- Simbolu virkne – teksta fonta izmērs. Pieņem gan "px", gan "pt" vērtības.
- Simbolu virkne – teksta fonta saime;
- Simbolu virkne – fonta stils. Iespējamās vērtības:
  - "normal"/false/null – normāls teksts;
  - "italic" – slīpraksts;
  - "oblique" – ieslīps teksts.
- Simbolu virkne – fonta variants. Iespējamās vērtības:
  - "normal"/false/null – normāls teksts;
  - "small-caps" – visi burti, arī mazie, tiek zīmēti kā lielie burti.
- Simbolu virkne – teksta treknums. Iespējamās vērtības:
  - "normal"/false/null – normāls teksts;
  - "bold" – treknraksts;
  - "bolder" – vēl treknāks teksts;

- “lighter” – plānāks teksts par normālo;
- “100” līdz “900” – treknuma vērtība.

- Simbolu virkne – teksta krāsa. Pieņem krāsu gan kā tās nosauku, gan heksadecimālu vērtību, gan krāsu pēc RGB formāta. Var arī padot gradientu vai izmantot attēla objektu kā teksta aizpildījumu. Noklusējuma krāsa ir melna.
- Simbolu virkne – teksta līdzinājums. Iespējamās vērtības:
  - “start”/”left”/false/null – teksts sākas dotajā teksta X koordinātā;
  - “end”/”left” – teksts beidzas dotajā teksta X koordinātā;
  - “center” – teksta centrs atrodas dotajā teksta X koordinātā.
- Simbolu virkne – teksta bāzlīnija. Iespējamās vērtības:
  - “alphabetic”/false/null – noklusējuma bāzlīnija;
  - “top” – bāzlīnija pašā teksta elementa augšā (augstāk par simboliem);
  - “hanging” – bāzlīnija simbolu augšā;
  - “middle” – bāzlīnija teksta vidū;
  - “ideographic” – ideogrāfiskā bāzlīnija. Paredzēta nestandarta rakstzīmēm kā ķīniešu hieroglifiem;
  - “bottom” – bāzlīnija pašā teksta elementa apakšā.

Attēls 4.1.1.3 **Ar `textAccessElement()` konstruktoru uzzīmētais teksts**

Ar 4.1.1.2. attēlā ilustrētajiem konstruktoriem, tiek panākta violeta, gan vertikāli, gan horizontāli centrēta, trekna un ieslīpa teksta izveide.

Jāpiebilst, ka par teksta elementu tiek uzskatīts viss ar konkrēto konstruktoru izveidotais teksta objekts. Ja ir vajadzība spēt mainīt atsevišķus vārdus vai pat simbolus, tad katram jāveido savs teksta elements. *Canvas* arī nepiedāvā iebūvētu funkcionalitāti teksta pārnešanai

jaunā rindiņā, tāpēc, lai veidotu jaunu rindiņu, ir jāveido jauns teksta objekts ar citām Y koordinātām.

4.1.2. Teksta jaunināšana

Jebkādas izmaiņas jau uzzīmētajā – pozīciju, teksta saturu, stila izmaiņas, līdzināšanu un bāzlīniju - teksta elementā veic ar `textAccessElement.updateText()` metodi (skatīt attēlu 4.1.2.1).

```
text.updateText(true, "Sveika vēlreiz, pasaule!", 200, 125, "28pt",
"Calibri", null, null, "bolder", "black");
```

Attēls 4.1.2.1 **Teksta jaunināšanas metodes izsaukums**

Jaunināšanas metode ieejā saņem tos pašus parametrus, kādus saņem teksta elementa konstruktora funkcija. Vienīgais izņēmums ir pirmais parametrs, kas vairs nav norāde uz `textAccessCanvas` objektu, bet gan būla vērtība, kura nosaka to, vai teksta elements tiks pārzīmēts uzreiz vai arī tiks sagaidīta lietotnes nākamā kadra zīmēšana, kad no jauna tiks pārzīmēti pilnīgi visi *canvas* zīmētie elementi.

Teksta tūlītēja pārzīmēšana nozīmē iepriekšējā teksta nodzēšanu, kas, savukārt, nozīmē arī tā fona reģiona, kurš atrodas aiz teksta, dzēšanu (skatīt attēlu 4.1.2.2). Tūlītējā pārzīmēšana piemērota gadījumos, kad teksts atrodas uz balta fona.

Attēls 4.1.2.2 **Ar textAccessElement.updateText() metodi jauninātais teksts**

Jaunināšanas metode maina tikai tos parametrus, kuri tiek padoti ieejā. Ja kāds no parametriem netiek iekļauts vai arī tā vērtība ir `null` vai `false`, tad tiek saglabāta tā vērtība, kāda tika padota teksta elementa konstruktora funkcijai.

4.1.3. Teksta iezīmēšana

Teksta iezīmēšanu veic, vai nu velkot kursoru pāri tekstam, kamēr tiek turēta peles kreisā poga, ar ko var iezīmēt daļu no teksta, vai arī veicot dubultklikšķi virs teksta elementa, kas iezīmē visu teksta elementu. Ja tiek turēta CTRL vai CMD (MAC datoriem) poga, var iezīmēt vairākus elementus arī ja to starpā tiek atlaista peles poga.

Ar katru kursora kustību virs *canvas* elementa, tiek pārbaudīts, vai tas atrodas virs kāda no *canvas* zīmētajiem tekstiem. To veic peles kustības uztvērēja atzvanīšanas funkcija (skatīt attēlu 4.1.3.1).

```
mousemoveEvent: function(e) {
	var self = this;
	var rect = self.canvas.canvas.getBoundingClientRect();
	//Iegūst peles/skāriena koordinātas
	if (self.touch && e.type === "touchmove") {
		var x = e.touches[0].clientX - rect.left;
		var y = e.touches[0].clientY - rect.top;
		self.lastMove = e;
	} else {
		var x = e.clientX - rect.left;
		var y = e.clientY - rect.top;
	}
	//Nosaka, vai kursors/skāriens atrodas virs teksta elementa
	for (var i = 0, len = self.textElements.length; i < len; i++) {
		if (x >= self.textElements[i].x - 5 && (x <=
self.textElements[i].x + self.textElements[i].textWidth + 5) && (y >=
self.textElements[i].y - self.textElements[i].lineHeight) && y <=
self.textElements[i].y) {
			var text = self.textElements[i];
			//Nomaina kursoru, ja atrodas virs teksta
			self.changeCursor(true);
	//Ja nospiesta peles poga/tiek turēts pirksts, uzsāk iezīmēšanu
			if (self.mouseDown) {
				//Iegūst pirmo rakstzīmi un ievieto masīvā
				self.currGlyph = self.getGlyph(text, x - text.x);
				if (!self.selection.length) {
					self.selection.push(self.currGlyph);
					self.colourSelection(text);
				}
				//Iegūst nākamo rakstzīmi
				if (self.currGlyph) {
					self.addGlyph(text);
				}
			}
			break;
		} else self.changeCursor(false);
	}
}
```

Attēls 4.1.3.1 **Peles kustības atzvanīšanas funkcija, kura nosaka, vai kursors atrodas teksta elementa robežās**

Šī metode vispirms nosaka kursora koordinātas *canvas* elementa kontekstā. Pēc tam ciklā pārlasa visus šim `textAccessCanvas` objektam pievienotos `textAccessElement`

objektus, uz kuriem atsaucies tiek turētas `textAccessCanvas.textElements` masīvā. Tiek pārbaudīts, vai kursors atrodas kāda no šo elementu robežās. Ja atrodas, tad maina *canvas* elementa kursora CSS stilu, lai tiktu rādīts teksta kursors, kāds parasti parādās, virzot kursoru pāri jebkādam tekstam. Šāda funkcionalitāte nepieciešama, lai lietotājs saprastu, ka tekstu iespējams iezīmēt.

Pēc kursora maiņas tiek izsaukta `textAccessCanvas.getGlyph()` metode (skatīt attēlu 4.1.3.2) aktuālā teksta simbola noteikšanai, kura ieejā saņem teksta elementu, virs kura atrodas kursors un kursora X ass koordinātu elementa kontekstā. Ja pagaidu masīvs iezīmēto simbolu glabāšanai, `textAccessCanvas.selection`, ir tukšs, simbolu tūlīt tam arī pievieno. Ja tajā ir kaut viens simbols, tad veic tālāku apstrādi un simbola pievienošanu glabātuvei ar a`textAccessCanvas.addGlyph()` metodi (skatīt attēlu 4.1.3.3).

```
getGlyph: function(string, mouseX) {
	var self = string;
	if (mouseX > string.textWidth) {
		return null; //Ja kursors nav virs teksta, atgriež null
	}
	//Iestata fontu slēptajam canvas
	self.offCanvas.ctx.font = self.styleString;
	//Iet cauri textAccessElement objekta teksta saturam
	//Ņem pirmo simbolu un mēra tā platumu pikseļos;
	//Ja kursors atrodas platuma robežās, tad atgriež elementu;
	//Ja kursors neatrodas simbola robežās, ņem nākamo simbolu.
	for (var i = 1, len = self.string.length; i <= len; i++) {
		var substr = self.string.substring(0, i);
		if (i === len)
		var substrWdt = self.offCanvas.ctx.measureText(substr).width - 1;
		else var substrWdt = self.offCanvas.ctx.measureText(substr).width;

		if (mouseX <= substrWdt - 1) {
			var glyph = self.string.substring(i - 1, i);
			var letterWdt =self.offCanvas.ctx.measureText(glyph).width;
			return {
				start: Math.ceil(substrWdt - letterWdt),
				end: Math.ceil(substrWdt),
				yPos: string.rectY,
				hgt: string.lineHeight,
				stringPos: string.x,
				id: string.id,
				glyph: glyph
			};
		}
	}
}
```

Attēls 4.1.3.2 **textAccessCanvas.`getGlyph()` metodes pirmkods**

Simbola noteikšanas metode darbojas tā, ka, sākot ar teksta pirmo simbolu, pēc kārtas ņem klāt katru nākamo simbolu un mērā izveidotās virknes platumu, līdz tiek sasniegta cikla iterācija, kurā kursora X koordināta ir mazāka vai vienāda ar virknes platumu – tas nozīmē, ka

pēdējais virknei pievienotais simbols ir tas, virs kura atrodas kursors un šis simbols tiek tajā brīdī iezīmēts. Metode atgriež atrastā simbola objektu, kurš satur simbola sākuma un beigu koordinātas teksta elementa kontekstā, simbola Y koordinātu *canvas* kontekstā, teksta elementa augstumu, teksta elementa atrašanās vietas X koordinātu *canvas* kontekstā, teksta elementa, kuram pieder simbols, identifikatoru, kā arī pašu simbolu.

```
addGlyph: function(string) {
	var self = this;
	var len = self.selection.length;
	var lastGlyph = self.selection[len - 1];
	//Ja tas pats simbols, pārtrauc izpildi
	if (lastGlyph.id === self.currGlyph.id &&
		lastGlyph.start === self.currGlyph.start) {
		return;
	}
      //Ja nolasīts iepriekšējais simbols, tas nozīmē pretējo iezīmēšanu
(atzīmēšanu) un izņem iepriekšējo elementu no glabātuves
	else if (len >= 2 &&
		self.selection[len - 2].id === self.currGlyph.id &&
		self.selection[len - 2].start === self.currGlyph.start) {
		self.selection.pop();
		self.colourSelection();
	}
      //Pretējā gadījumā, simbolu ievieto glabātuvē
	else {
	//Ja kursors iezīmēšanas laikā izgājis no teksta reģiona vai pārgājis
uz citu teksta elementu, iezīmē trūkstošo daļu
		if (lastGlyph.id === self.currGlyph.id &&
			lastGlyph.end < self.currGlyph.start &&
			lastGlyph.end < self.currGlyph.end) {
			self.selectMissing(string, false, self.currGlyph);
			return;
		} else if (lastGlyph.id === self.currGlyph.id &&
			lastGlyph.start > self.currGlyph.end &&
			lastGlyph.start > self.currGlyph.start) {
			self.selectMissing(string, false, self.currGlyph);
			return;
		}

		if (lastGlyph.id !== self.currGlyph.id) {
			self.selectMissing(string, true, self.currGlyph);
			return;
		}
		self.selection.push(self.currGlyph);
		self.colourSelection();
	}
}
```

Attēls 4.1.3.3 **textAccessCanvas.addGlyph() metodes pirmkods**

Simbola pievienošanas metode `textAccessCanvas.addGlyph()` nosaka to, vai iepriekš iegūtais simbols tiks pievienots atlasīto simbolu masīvam. Simbolu unikalitāti nosaka pēc to teksta elementa identifikatora un simbola sākuma koordinātām teksta kontekstā – ja abas vērtības uz iezīmēšanu pretendējošajam simbolam ir vienādas ar pēdējā iezīmētā simbola vērtībām, tad tas nozīmē, ka simbols ir tas pats un metode tiek izbeigta. Ja identifikators un

sākuma koordināta ir vienādas pretendējošajam un pirmspēdējam simbolam, tad tas nozīmē, ka lietotājs atceļ iezīmēšanu, virzot kursoru pretējā virzienā un pēdējo simbolu no glabātuves dzēš.

Ja simbolu sākuma koordinātas nesakrīt, taču identifikators ir tas pats, tad tas nozīmē, ka iezīmēšanas laikā kursors ir izgājis no teksta elementa robežām un pēc tam atgriezies tajā pašā teksta elementā. Šajā gadījumā tiek izsaukta `textAccessCanvas.selectMissing()` metode, kura pievieno simbolu glabātuves masīvam iztrūkstošos simbolu objektus.

Ja tiek konstatēts, ka simbolu identifikatori nesakrīt, tad tas nozīmē kursora pāreju pie cita teksta elementa, kas nozīmē arī pāreju jaunā līnijā. Šajā gadījumā tiek pilnībā iezīmēts iepriekšējais teksta elements un jaunā elementa daļa līdz kursora atrašanās vietai (skatīt attēlu 4.1.3.4).

Attēls 4.1.3.4 **Teksta iezīmēšana, tās laikā pārvelkot kursoru no apakšējās uz augšējo rindiņu**

Ja neviens no speciālgadījumu nosacījumiem neizpildās, simbols tiek pievienots glabātuves masīvam (skatīt attēlu 4.1.3.5).

Attēls 4.1.3.5 **Teksta iezīmēšana, neizpildoties speciālgadījumiem**

4.1.4. Teksta iekrāsošana

Katru reizi, kad simbolu glabātuves masīvam tiek pievienots jauns simbola objekts, tiek izsaukta arī `textAccessCanvas.colourSelection()` - iezīmētā reģiona iekrāsošanas metode (skatīt attēlu 4.1.4.1).

```
colourSelection: function() {
        var self = this;

        if (self.overlay) {
                self.clearOverlay(); //Notīra pārklāju
                self.removeDups(); //Iznīcina dublikātus
                self.overlay.globalAlpha = 0.15; //Uzstāda pārklāja caurspīdību

                //Cikls iekrāso taisnstūri ap katru iezīmēto simbolu
                for (var i = 0, len = self.selection.length; i < len; i++) {
                        var glyph = self.selection[i];
                        self.overlay.fillRect(glyph.stringPos + glyph.start,
                                glyph.yPos,
                                glyph.end - glyph.start,
                                glyph.hgt);
                }
        } else return;
}
```

Attēls 4.1.4.1 **textAccessCanvas.colourSelection() metode teksta iekrāsošanai**

Šī metode iekrāso taisnstūri ap katru simbola objektu, kurš atrodas simbolu pagaidu glabātuvē. Lai negadītos nejauši dublikāti, tad katru reizi, kad tiek iekrāsots teksts, tiek izsaukta arī dublikātu dzēšanas metode `removeDups()`.

Teksta iekrāsošana tiek veikta uz iepriekš izveidota *canvas* elementa, kurš pozicionēts tā, lai pārklātu galveno *canvas* elementu, uz kura tiek zīmēts teksts un citi lietotnes objekti.

4.1.5. Teksta kopēšana

Teksta kopēšana uz starpliktuvi no *JavaScript* koda ir iespējama tikai tad, ja lietotājs šo darbību ir uzsācis. Ja kods varētu patvaļīgi piekļūt sistēmas starpliktuvei, tas būtu nopietns drošības risks vai, kā minimums, varētu radīt neērtības lietotājiem, tāpēc lietotājam ir kaut kas jāizdara – tas var būt peles klikšķis, taustiņu kombinācija, HTML pogas nospiešana – lai izsauktu galveno kopēšanas funkciju - `document.execCommand('copy')` [17].

Lai būtu iespējams tekstu nokopēt, tam vispirms jābūt sagatavotam kā simbolu virknei. Līdz šim, ar "CanvasTextAccess" iezīmētais teksts glabājas objektu masīvā, kur katrs masīva elements ir simbola objekts ar ne tikai pašu rakstzīmi, bet gan arī pavadošajiem metadatiem.

Tā ar šo spraudni zīmētie teksta elementi ir savstarpēji neatkarīgi, tad katru no tiem ir iespējams iezīmēt atšķirīgos virzienos. Respektīvi – vienu vārdu var iezīmēt no kreisās uz labo pusi, taču citu vārdu no labās uz kreiso pusi. Tā kā simboli pagaidu glabāšanas masīvā tiek ievietoti to iezīmēšanas secībā, var rasties situācija, ka, ciklā apstrādājot masīvu, lai veidotu simbolu virkni, gala rezultātā viens vārds būs iezīmēts pareizi, taču cits – spoguļrakstā. Lai no tā izvairītos, ikreiz, kad tiek atlaista peles poga vai skāriens, masīvs tiek kārtots (skatīt attēlu 4.1.5.1).

```
for (var i = 0, len = self.selection.length; i < len; i++) {
	if (i + 1 === len) {
		break;
	}
	if (self.selection[i].id === self.selection[i + 1].id &&
		self.selection[i].start < self.selection[i + 1].start) {
		self.readyString += self.selection[i].glyph;
	} else if (self.selection[i].id === self.selection[i + 1].id &&
		self.selection[i].start > self.selection[i + 1].start) {
		while (self.selection[i].id === self.selection[i + 1].id) {
			tempArr.push(self.selection[i].glyph);
			i++;
		}
		if (self.selection[i + 1].id === 0) {
			tempArr.push(self.selection[i].glyph);
		}
		while (tempArr.length) {
			self.readyString += tempArr.pop();
		}
	} else {
		self.readyString += self.selection[i].glyph;
	}
}
```

Attēls 4.1.5.1 **Simbolu kārtošanas cikls**

Cikls pēc kārtas salīdzina simbolu teksta elementu identifikatorus un to sākuma koordinātas. Ja secīgi simboli ir ar to pašu identifikatoru un nākamā simbola koordināta ir lielāka par iepriekšējā, tad tas nozīmē, ka iezīmēšana notikusi no kreisās uz labo pusi un simbolu pievieno izejas virknei.

Ja nākamā simbola koordināta ir mazāka par iepriekšējā, tad simbolu ievieto pagaidu masīvā. Kad sasniegts simbols ar citu identifikatoru (cits teksta elements) – no pagaidu masīva tiek atpakaļgaitā pievienoti simboli izejas virknei, tādējādi nodrošinot simbolu pareizu secību. Starp atsevišķiem teksta elementiem, iezīmēšanas laikā, tiek automātiski pievienotas atstarpes. Teksts, kurš *canvas* attēlots kā izvietots pa vairākām līnijām, kopējot būs vienā līnijā, jo teksta zīmēšanas koordinātas *canvas* lietotājs izvēlas patvaļīgi, līdz ar to nav iespējams noteikt, vai teksts, kurš atrodas citā elementā, ir domāts kā jaunā rindiņā.

Kopēšanas funkciju (skatīt attēlu 4.1.5.2) iespējams izsaukt divos veidos:

1. Veicot taustiņu kombināciju CTRL+C vai CMD+C;
2. Klikšķinot uz konteksta pogas (skatīt attēlu 4.1.3.4 vai 4.1.3.5), kura parādās ikreiz pēc iezīmēšanas beigām un peles pogas vai pirksta atlaišanas. Šī poga ir obligāta, lai mobilo ierīču lietotāji varētu kopēt tekstu, jo mobilajās ierīcēs nav iespējams veikt klaviatūras taustiņu kombinācijas.

Kopēšana, izmantojot klaviatūrās vai datorpelēs iebūvēto kopēšanas funkcionalitāti, nestrādā, jo *JavaScript* to neuzskata par tiešu darbību ar tīmekļa vietni, pat ja izveidots CTRL+C makro.

```
copy: function() {
	var self = this;
	var text = self.readyString;
	//Izveidot textArea HTML elementu, pievieno dokumentam
un ievieto tajā gatavo tekstu
	var tempCopy = document.createElement("textArea");
	document.body.appendChild(tempCopy);
	tempCopy.value = text;
	tempCopy.select(); //Atlasa teksta elementu
	//Mēģina veikt kopēšanu
	try {
		document.execCommand('copy');
	} catch (err) {
		console.log('Copy error');
	}
	//Pēc kopēšanas dzēš textArea elementu
	tempCopy.parentNode.removeChild(tempCopy);
}
```

Attēls 4.1.5.2 **textAccessCanvas.copy() metode**

Metode ņem kopēšanai sagatavoto simbolu virkni `readyString` un ievieto to tikko izveidotā HTML teksta laukuma elementā, jo ar select() funkciju atlasīt var tikai teksta lauka un ievades (*<input>*) elementus. Kad elements atlasīts, mēģina veikt kopēšanu un, ja tā

neizdodas, pārlūkprogrammas konsolē tiek izvadīts kļūdas paziņojums. Pēc kopēšanas izpildes teksta laukuma elements tiek dzēsts no DOM.

4.1.6. Teksta redzamība ekrāna lasītājiem

Lai padarītu tekstu redzamu ekrāna lasīšanas programmatūrai, vienīgais risinājums ir teksta elementu saturu dublēt atsevišķos HTML elementos un ievietot tos galvenā *canvas* elementa slēptajā DOM un arī ārpus tā.

Ja `textAccessCanvas` konstruktorā padota nenegatīva ekrāna lasīšanas atbalsta vērtība, tad visiem pēc tam pievienotajiem `textAccessElement` objektiem tiek izveidots speciāls <*span*> elements, kuru ievieto *canvas* slēptajā DOM. Atšķirība starp režīmiem (“local”, “global” un “universal”) ir tajā, ka ar ”local” ekrāna lasīšanas iestatījumu, šiem izveidotajiem <*span*> elementie tiek uzstādīts “aria-hidden = true” atribūts, kurš slēpj tekstu no ekrāna lasītājiem. Elements kļūst īslaicīgi pieejams ekrāna lasītājiem tikai tad, kad uz to tiek fokusēts ar TAB taustiņa palīdzību. Ar TAB taustiņu vēl iespējams fokusēt, ja iestatīta “universal” ekrāna lasīšana (skatīt attēlu 4.1.6.1).

```
generateAccessibleText: function() {
	var self = this;
	this.readableElement = document.createElement("span");

	if (self.accessCanvas.screenReader === "local" ||
		self.accessCanvas.screenReader === "universal") {
		this.readableElement.tabIndex = 1;
		this.readableElement.addEventListener("focus", function() {
			if (self.accessCanvas.screenReader === "local") {
			self.readableElement.setAttribute("aria-hidden", false);
			setTimeout(function() {
			self.readableElement.setAttribute("aria-hidden", true);
			}, 100);
		}
			self.accessCanvas.overlay.strokeRect(self.x - 2,
self.rectY - 2, self.textWidth + 4, self.lineHeight + 4);
		});
		this.readableElement.addEventListener("blur", function() {
			self.accessCanvas.overlay.clearRect(0, 0,
self.accessCanvas.rectCoords.width, self.accessCanvas.rectCoords.height);
		});
	}
	this.readableElement.textContent = self.string;

	if (self.accessCanvas.screenReader === "local") {
		this.readableElement.setAttribute("aria-hidden", true);
	}

	self.accessCanvas.canvas.canvas.appendChild(this.readableElement);
}
```

Attēls 4.1.6.1 **canvasTextElement.generateAccessibleText() metode**

Fokusējot uz elementu ar TAB taustiņu, arī tiek apvilkta līnija ap fokusēto tekstu, to zīmējot uz *canvas* elementa pārklāja (skatīt attēlu 4.1.6.2).

Attēls 4.1.6.2 **Ap ar TAB taustiņu fokusēto elementu apvilkta kontūra**

Tā kā “aria-hidden = true” tiek uzstādīts tikai elementiem ar “local” ekrāna lasīšanas parametra vērtību, tad gan elementiem ar “global”, gan “universal”, tiek nolasīti visi izveidotie teksta elementi pēc klikšķa uz *canvas* elementa. Jāņem vērā, ka elementi tiek nolasīti to izveidošanas secībā, nevis secībā, kādā tie zīmēti *canvas*.

Teksta elementu individuāla nolasīšana vēl iespējama, ja uzstādīts “local” vai “universal” ekrāna lasīšanas parametrs. Pēc katra peles klikšķa vai pirksta spiediena nosaka, vai trāpīts uz teksta elementa un ja jā, tad īslaicīgi izveido HTML teksta lauka elementu, kurš satur nolasāmo tekstu un to slēpj ar CSS palīdzību, tomēr ļaujot ekrāna lasītājam to reģistrēt un izrunāt tekstu. Pēc neilga laika teksta lauka elementu dzēš, lai netraucētu tālākā pārlūkošanā (skatīt attēlu 4.1.6.3).

```
var tmpElem = document.createElement("textArea");

tmpElem.style.position = "absolute";
tmpElem.style.left = "-9999px";
tmpElem.style.top = rect.top + "px";

tmpElem.setAttribute("aria-live", "assertive");
tmpElem.value = text.string;
document.body.appendChild(tmpElem);
self.canvas.canvas.blur();

setTimeout(function() {
	tmpElem.parentNode.removeChild(tmpElem);
}, 100);
```

Attēls 4.1.6.3 **Teksta padarīšana par ekrānlasāmu pēc klikšķa uz teksta elementa**

Tā kā arī kopēšanas laikā tiek īslaicīgi izveidots HTML teksta lauka elements, kuram var piekļūt ekrāna lasītāji (skatīt attēlu 6.1.5.2), tad kopējot tiek nolasīts starpliktuvē ievietotais teksts.

4.2. CanvasTextAccess veiktspēja

Sagaidāms, ka "CanvasTextAccess" spraudnis negatīvi ietekmēs teksta izveides veiktspēju. Apakšcikla esamība dublikātu pārbaudes un izmešanas metodē, kura tiek izsaukta katru reizi, kad tiek iezīmēts jauns simbols, nozīmē $O(N^2)$ algoritma sarežģītību. Turklāt katru reizi, kad tiek pārvietots kursors un noteikts, vai kursors atrodas virs kāda no izveidotajiem teksta elementiem, meklēts teksta elementa simbols vai iekrāsots iezīmētais reģions, tiek veikta attiecīgo masīvu pilnā pārlase.

Tika testētas *canvas* iebūvētā teksta zīmēšanas metode `fillText()` teksta elementa izveide ar `textAccessElement()` konstruktoru un iznīcināšana ar un teksta pārzīmēšana ar `textAccessElement.redrawAll()` metodi, kura neveido elementus no jauna, taču tos tikai pārzīmē. Testi veikti, zīmējot simts teksta elementus ar vārdu "Test" ciklā 1000 reizes. Teksts zīmēts ar noklusējuma stilu – 10pt Helvetica. Diemžēl, nav iespēja objektīvi testēt teksta iezīmēšanu un veiktspējas zudumu pie lieliem teksta elementu un tajos esošā teksta satura apjomiem, jo cilvēciskais faktors, velkot kursoru pāri tekstam, atstātu pārāk lielu ietekmi uz rezultātiem.

Katrs tests veikts piecas reizes, galā aprēķinot vidējo aritmētisko rezultātu. Testi veikti ar pārlūkprogrammu Opera 37, Google Chrome 50 un Internet Explorer 11 procesora noslodzes profilēšanas rīkiem. Mozilla Firefox izstrādātāju rīki ir nepiemēroti automātiskai testēšanai. Rezultāti uzrādīti milisekundēs (ms) (skatīt tabulu 4.2.1).

Tabula 4.2.1

Veiktspējas testu rezultāti

Tests	Opera 37	Chrome 50	IE 11
fillText() metode	1224,2	1230,8	1700,7
	1124,6	1243,7	1643,5
	1142,0	1230,5	1576,0
	1141,3	1235,2	1697,4
	1135,9	1256,4	1643,8
Vidējais laiks	**1153,6**	1239,3	1652,3
Jauna elementa izveide	212939,5	137715,2	224602,8
	198331,3	147083,8	225342,9
	221961,7	164277,1	221282,7
	217821,5	184176,3	230303,2
	222288,4	159735,8	250094,3
Vidējais laiks	214668,5	**158597,6**	230325,2
redrawAll() metode	5591,3	5844,3	5842,3
	5422,8	5760,9	6270,4
	5518,8	5977,7	6115,3
	5564,6	5722,6	7272,4
	5438,0	6056,1	6172,4
Vidējais laiks	**5507,1**	5872,3	6334,6

No veiktspējas testu rezultātiem var secināt, ka, animētas lietotnes gadījumā, kad katra kadra sākumā ir no jauna jāzīmē visi attēlojamie objekti, *canvas* iebūvētā zīmēšanas metode izpildās visātrāk, taču ar to zīmētais teksts nav pieejams lietotājam. Starp "CanvasTextAccess" spraudņa izmantošanas iespējām, visefektīvāk ir teksta elementus vienkārši zīmēt no jauna katra animācijas kadra sākumā, nevis tos iznīcināt un veidot pilnīgi jaunus objektus. Kopumā, teksta pārzīmēšanas ar spraudņa palīdzību veiktspēja, salīdzinot ar *canvas* iebūvēto funkcionalitāti, ir labāk nekā sagaidīta.

Mazliet pārsteidzoši, ka, lai arī gan Opera, gan Google Chrome pārlūkprogrammas ir veidotas uz vienas un tās pašas *WebKit* bāzes, Opera tomēr ir ātrāka manipulācijās ar *canvas* elementiem. Chrome uzvar jauna elementa izveides testā, jo ātrāk izveido jaunus HTML elementus un pievieno tos DOM.

4.3. CanvasTextAccess spraudņa kopsavilkums

Ir izdevies izveidot spraudni, kurš apmierina visas sākotnējās prasības:

- Tekstu iespējams iezīmēt, kopēt un ielīmēt gan ar klaviatūras taustiņu kombinācijas, gan konteksta pogas palīdzību;
- Lai arī *canvas* nepastāv iespēja dalīt tekstu vairākās rindiņās, ir iespējams iezīmēt vairākus teksta elementus, kuri izvietoti viens virs otra vai blakus;
- Teksts pieejams ekrāna lasītājiem, turklāt ar vairākiem pieejamības režīmiem;
- Tekstam iespējams piemērot visus *canvas* atbalstītos teksta stila veidus;
- Izveidotajam tekstam iespējams mainīt gan saturu, gan stilu;
- Tekstu iespējams pārvietot pa *canvas* elementu, mainot tā koordinātas katra jauna kadra iterācijā;
- Teksts iezīmējams un kopējams arī ar skāriena saskarnēm;
- Spraudņa kods rakstīts ar tīru *JavaScript*, bez citu bibliotēku palīdzības.

Nebija iespējams spraudni veidot tā, lai netiktu izmantoti dinamiski veidoti un slēpti HTML elementi, jo tie nepieciešami, lai varētu kopēt un darīt tekstu pieejamu ekrāna lasītājiem.

Spraudņa demonstrāciju un pilnu kodu ar komentāriem iespējams aplūkot šeit: http://codepen.io/Villene/pen/wGVNJq/.

Izvēlētu metožu pirmkodu ar komentāriem iespējams apskatīt 1. pielikumā.

SECINĀJUMI

Bakalaura darba izstrādes laikā ir veiksmīgi izdevies izveidot "CanvasTextAccess" - *JavaScript* spraudni, kurš padara *canvas* zīmēto tekstu pieejamu tīmekļa lietotājiem, ļaujot to iezīmēt, kopēt un padarot tekstu redzamu ekrāna lasīšanas programmatūrai.

Lielākās problēmas šāda spraudņa izstrādē sagādā tas, ka nepieciešams no jauna izveidot teksta iezīmēšanas mehānismu – funkcionalitāti, pie kuras tehnoloģiju lietotāji ir tik ļoti pieraduši, ka pat neiedomājas par to, kā tam jānotiek, lai teksts tiktu iezīmēts korekti.

Salīdzinot izveidoto *JavaScript* spraudņa risinājumu ar citiem, jāsecina, ka vienkāršākā metode teksta pieejamības problēmas risināšanā, ir tekstu nevis zīmēt *canvas*, bet gan *canvas* elementu pārklāt ar citiem HTML teksta elementiem – šādi ar minimālu piepūli no izstrādātāja puses tiek nodrošināta gan teksta iezīmēšana, gan pieejamība ekrāna lasītājiem, izstrādātajam nedomājot par tādām niansēm kā teksta zīmēšana katra animācijas kadra sākumā, tomēr, arī šo metodi būtu iespējams atvieglot vēl vairāk, izveidojot specializētu *JavaScript* spraudni tieši *canvas* elementa pārklāšanai ar citiem HTML teksta elementiem. Arī izveidotais spraudnis daļēji izmanto citas apskatītās pieejamības problēmas risinājuma metodes – *canvas* elementa pārklāšanu un teksta elementu slēpšanu ar CSS. Tomēr izveidotais spraudnis ir laba alternatīva gadījumos, kad izstrādātājs nevēlas lieki nopūlēties pats ar sava risinājuma izveidi. Tādā gadījumā atliek tikai iekļaut "CanvasTextAccess" spraudņa kodu savas lietojumprogrammas kodā un ar dažām koda rindiņām iespējams izveidot lietotājiem, tajā skaitā arī redzes invalīdiem, pieejamu tekstu.

Izveidotais spraudnis gan nerisina problēmu ar teksta iegūšanu no jau izveidotām tīmekļa lietotnēm – šāda funkcionalitāte iespējama tikai ar zīmējamā teksta priekšapstrādi.

Kopumā secināts, ka, teksta iegūšana no HTML5 *canvas* elementiem ir iespējama. Spraudni tālāk iespējams attīstīt, ļaujot arī teksta ievadīšanu un uzlabojot teksta iezīmēšanas nianses, lai iezīmēšana vēl vairāk līdzinātos tai, kāda tā ir ierīces operētājsistēmas līmenī.

IZMANTOTĀ LITERATŪRA UN AVOTI

[1] RGraph, «The HTML5 canvas tag - what is it?,» RGraph, [Tiešsaiste]. Pieejams: http://www.rgraph.net/html5-canvas#history. [Piekļūts 17.05.2016.].

[2] V. Laipnieks, «Teksta iegūšana no HTML5 canvas elementiem», kursa darbs. Latvijas Universitāte, Rīga, 2015.

[3] W3C, «Accessible Rich Internet Applications (WAI-ARIA) 1.1,» W3C, [Tiešsaiste]. Pieejams: https://www.w3.org/TR/2016/WD-wai-aria-1.1-20160317/. [Piekļūts 21.05.2016.].

[4] A. Hedges, «Speed test: innerHTML versus DOM manipulation,» [Tiešsaiste]. Pieejams: http://andrew.hedges.name/experiments/innerhtml/. [Piekļūts 17.05.2016.].

[5] WebAIM, «CSS in Action - Invisible Content Just for Screen Reader Users,» WebAIM, [Tiešsaiste]. Pieejams: http://webaim.org/techniques/css/invisiblecontent/#techniques. [Piekļūts 17.05.2016.].

[6] K. Kwok, «Ocrad.js - Optical Character Recognition in Javascript,» [Tiešsaiste]. Pieejams: http://antimatter15.com/ocrad.js/demo.html. [Piekļūts 17.05.2016.].

[7] Goldfire Studios, «CanvasInput - HTML5 Canvas Text Input,» [Tiešsaiste]. Pieejams: http://goldfirestudios.com/blog/108/CanvasInput-HTML5-Canvas-Text-Input. [Piekļūts 10.05.2016.].

[8] L. Roeder, «Textor HTML Sample,» [Tiešsaiste]. Pieejams: http://www.lutzroeder.com/html5/textor/. [Piekļūts 10.05.2016.].

[9] D. Kubiškins, «Everything Frontend,» [Tiešsaiste]. Pieejams: https://www.everythingfrontend.com/pages/canvas-text-editor-tutorial.html. [Piekļūts 11.05.2016.].

[10] W3schools, «HTML canvas textAlign Property,» [Tiešsaiste]. Pieejams: http://www.w3schools.com/tags/canvas_textalign.asp. [Piekļūts 22.05.2016.].

[11] W3schools, «HTML canvas textBaseline Property,» [Tiešsaiste]. Pieejams: http://www.w3schools.com/tags/canvas_textbaseline.asp. [Piekļūts 22.05.2016.].

[12] W3schools, «HTML tabindex Attribute,» [Tiešsaiste]. Pieejams: http://www.w3schools.com/tags/att_global_tabindex.asp. [Piekļūts 22.05.2016.].

[13] O. Hagers, «Devbutze: HTML5 Canvas: Offscreen Rendering,» [Tiešsaiste]. Pieejams: http://devbutze.blogspot.com/2014/02/html5-canvas-offscreen-rendering.html. [Piekļūts 23.05.2016.].

[14] V. Laipnieks, «CanvasTextAccess,» [Tiešsaiste]. Pieejams: http://codepen.io/Villene/pen/wGVNJq. [Piekļūts 25.05.2016.].

[15] NV Access, «NV Access,» [Tiešsaiste]. Pieejams: http://www.nvaccess.org. [Piekļūts 24.05.2016.].

[16] Google, «ChromeVox,» [Tiešsaiste]. Pieejams: http://www.chromevox.com/. [Piekļūts 24.05.2016.].

[17] C. Buckler, «Roll Your Own Copy to Clipboard Feature in 20 Lines of JavaScript,» [Tiešsaiste]. Pieejams: https://www.sitepoint.com/javascript-copy-to-clipboard/. [Piekļūts 25.05.2016.].

PIELIKUMI

1. PIELIKUMS. “CANVASTEXTACCESS” IZVĒLĒTU METOŽU PIRMKODS

1.1. TextAccessCanvas konstruktora funkcija

```
//Konstruktors, kurš sagatavo lapā esošo canvas tālākai specializētā teksta
ievietošanai.
//Parametri:
//canvas - norāda canvas elementa ID, canvas kontekstu vai atstāj tukšu. Ja atstāj
tukšu, tad konstruktors ņem pirmo lapā esošo elementu.
//screenreader - false/null (ja nevēlas ekrāna lasītāju atbalstu);
//              "local" - string (tiek nolasīts teksta elements, ja uz tā
uzklikšķina;
//              "global" - string (nolasa visu canvas esošo tekstu pēc klikšķa
canvas reģionā;
//              "universal" - string ("local" un "global" apvienojums)
//touch - true/false (skārienekrānu atbalsta iespējošana)
//selectionColour - simbolu virkne (krāsa, kādā iekrāsos iezīmēto reģionu)
var textAccessCanvas = function(canvas, screenreader, touch, selectionColour) {
	//atrod canvas elementu un iegūst kontekstu
	if (typeof canvas === 'string') {
		var c = document.getElementById(canvas);
		this.rectCoords = c.getBoundingClientRect();
		ctx = c.getContext("2d");
	} else if (canvas === null) {
		var c = document.getElementsByTagName("canvas")[0];
		this.rectCoords = c.getBoundingClientRect();
		ctx = c.getContext("2d");
	} else {
		if (typeof canvas.canvas === 'undefined') {
			this.rectCoords = canvas.getBoundingClientRect();
			var c = canvas;
			var ctx = canvas.getContext("2d");
		} else {
			this.rectCoords = canvas.canvas.getBoundingClientRect();
			var c = canvas.canvas;
			ctx = canvas;
		}
	}

	c.tabIndex = 1; //nodrošina iespēju fokusēt uz canvas ar TAB taustiņu
	c.style.zIndex = 2; //ļauj novietot elementus zem canvas
	this.canvas = ctx;
	this.textElements = []; //glabās visus šim elementam pievienotos teksta
elementus
	this.selection = []; //glabās iezīmētos simbolus
	this.readyString = ""; //glabās kopēšanai sagatavoto simb. virkni
	this.mouseDown = false; //peles turēšanas karogs
	this.selectionColour = selectionColour;
	this.overlay = this.generateOverlay(c); //izveido pārklāju
```

```
        this.screenReader = screenreader; //ekrāna lasītāja karogs
        this.touch = touch; //skāriensaskarnes karogs

        //ja skāriensaskarne iespējota, iestata papildu mainīgos
        if (touch) {
                //          this.lastTap = 0;
                this.lastMove; //pēdējās pirksta atrašanās vietas koordinātas
        }

        //ievieto notikumu uztvērējus mainīgajā, lai vēlāk varētu tos dzēst
        this.eventReferences = this.createListeners();

        //pagaidu glabātava patreiz iezīmētajam simbolam
        this.currGlyph = null;

        //izveido ikonu kopēšanas pogai
        this.copyIcon = new Image();
        this.copyIcon.src = 'http://icons.iconarchive.com/icons/ampeross/qetto-
2/256/copy-icon.png';
        this.copyIconDimension = 48; //izmērs (noklusējuma 48x48 px)
        this.copyIconRect = null; //ikonas robežu objekts
};
```

1.2. TextAccessElement konstruktora funkcija

```
//Konstruktors, kurš izveido textAccessCanvas zīmējamo tekstu
//Parametri:
//accessCanvas - textAccessCanvas objekts;
//string - simbolu virkne (zīmējamā simbolu virkne);
//x - skaitlis (zīmēšanas x koordināta, kur 0 ir kreisā canvas mala);
//y - skaitlis (zīmēšanas y koordināta, kur 0 ir canvas augšējā mala);
//fontSize - simbolu virkne (zīmējamā teksta izmērs. Atbalsta gan pt, gan px
tipus);
//fontFamily - simbolu virkne (zīmējamā teksta fonts, atbilstoši CSS tipa fontu
pierakstam);
//fontStyle - simbolu virkne (teksta stils - normal, italic, oblique);
//fontVariant - simbolu virkne (fonta variants - normal, small-caps);
//fontWeight - simbolu virkne (fonta biezums, atbilstoši CSS tipa pierakstam);
//colour - simbolu virkne (teksta krāsa);
//textAlign - simbolu virkne (teksta pielīdzināšana kādai no malām);
//baseline - simbolu virkne (bāzlīnijas nobīde)
var textAccessElement = function(accessCanvas, string, x, y, fontSize, fontFamily,
fontStyle, fontVariant, fontWeight, colour, textAlign, baseline) {
        this.accessCanvas = accessCanvas;
        //Uzstāda id kā pēdējā elementa id+1
        if (this.accessCanvas.textElements.length) {
                this.id =
this.accessCanvas.textElements[this.accessCanvas.textElements.length - 1].id + 1;
        } else this.id = 1;

        //Iestata mainīgo sākuma vērtības
        this.string = string;
        this.x = x;
        this.y = y;
        this.fontSize = fontSize;
        this.fontFamily = fontFamily;
        this.fontStyle = fontStyle;
        this.fontVariant = fontVariant;
        this.fontWeight = fontWeight;
```

```
	this.colour = colour;

	//Sagatavo stila parametrus canvas saprotamā formātā
	this.styleString = this.getStyle(fontSize, fontFamily, fontStyle,
fontVariant, fontWeight, colour);

	//Izveidot slēpto canvas elementu teksta mērījumu veikšanai bez zīmēšanas
uz galvenā canvas elementa
	this.offCanvas = document.createElement('canvas');
	this.offCanvas.setAttribute('width', this.textWidth);
	this.offCanvas.setAttribute('height', this.lineHeight);
	this.offCanvas.ctx = this.offCanvas.getContext('2d');

	this.offCanvas.ctx.font = this.styleString;
	this.textWidth =
Math.ceil(this.offCanvas.ctx.measureText(this.string).width); //Izmēra teksta
platumu

	//Ja lietotājs noteicis teksta pielīdzināšanu, veic kompensāciju teksta x
koordinātai
	this.textAlign = textAlign;
	if (textAlign) {

		this.compensateAlignment(textAlign);
	}

	//Iegūst teksta augstumu
	this.lineHeight = this.getLineHeight().height;
	this.baseline = this.getLineHeight().baseline;

	//Ja definēta cita bāzlīnija, pielāgo y koordinātu
	this.baselineType = baseline;
	if (baseline) {
		this.compensateBaseline(baseline);
	}
	this.rectY = this.y - this.baseline; //nosaka iezīmēšanas taisnstūra
augstumu

	this.accessCanvas.textElements.push(this); //Piesaista izveidoto teksta
elementu textAccessCanvas objektam
	this.drawText(); //Zīmē tekstu

	//Ja ekrāna lasīšanas iestatīta, padara tekstu redzamu ekrānlasītājiem pēc
klikšķa uz canvas elementa
	if (accessCanvas.screenReader) {
		this.generateAccessibleText();
	}
};
```

1.3. Metode textAccessCanvas.selectMissing

```
//Trūkstošās daļas iezīmēšana
//Parametri:
//string - textAccessElement objekts;
//newLine - true/false (atzīmē, vai notikusi pāreja jaunā līnijā/uz citu teksta
elementu;
//currGlyph - patreiz iezīmētais simbols
selectMissing: function(string, newLine, currGlyph) {
	var self = this;
	var tmpString = [];
	var startX;
	var endX;
	var rightSelection; //karogs apzīmē iezīmēšanas virzienu
	var selLen = self.selection.length;
	//Ja pāreja jaunā līnijā
	if (newLine) {
		var prevID = self.selection[selLen - 1].id;
		//Nosaka, no kura elementa pāriets un to elementu iezīmē pilnībā
		for (var i = 0, len = self.textElements.length; i < len; i++) {
			if (prevID === self.textElements[i].id) {
				self.selectWholeElement(self.textElements[i]);
			}
		}
		rightSelection = true;

		//No simbolu glabātuves dzēš visus jau esošos tā elementa simbola
elementus, uz kuru pāriet
		for (var i = 0; i < self.selection.length; i++) {
			if (self.selection[i].id === string.id) {
				self.selection.splice(i, 1);
				i--;
			}
		}
		//Iestata sākuma un beigu koordinātas trūkstošo elementu iezīmēšanai
		startX = 0;
		endX = currGlyph.end;
	} else {
		//Nosaka iezīmēšanas virzienu un, atkarībā no tā, iezīmē sākuma un
beigu koordinātas iezīmēšanai
		if (self.selection[selLen - 1].end < currGlyph.start) {
			startX = self.selection[selLen - 1].end;
			endX = currGlyph.end;
			rightSelection = true;
		} else {
			startX = self.selection[selLen - 1].start;
			endX = currGlyph.end;
			rightSelection = false;
		}
	}

	//Kamēr nav sasniegtas beigu koodrinātas, iezīmē trūkstošos simbolus
	while (startX !== endX) {
		if (rightSelection) {
			self.selection.push(self.getGlyph(string, startX));
			startX = self.selection[self.selection.length - 1].end;
		} else {
			tmpString.push(self.getGlyph(string, endX));
			endX = tmpString[tmpString.length - 1].end;
		}
```

```
        }
        if (!rightSelection) {
                while (tmpString.length) {
                        self.selection.push(tmpString.pop());
                }
                self.selection.push(currGlyph);
        }
}
```

1.4. Metode textAccessCanvas.selectWholeElement

```
//Vesela elementa iezīmēšana.
//Parametri:
//string - textAccessElement objekts, kuru iezīmēt
selectWholeElement: function(string) {
        var self = this;

        //Dzēš visus jau iezīmētos tā objekta simbolus
        for (var i = 0; i < self.selection.length; i++) {
                if (self.selection[i].id === string.id) {
                        self.selection.splice(i, 1);
                        i--;
                }
        }

        var lastElem = self.getGlyph(string, 0); //Iegūst elementa pirmo simbolu
        self.selection.push(lastElem);

        var strWdt = string.textWidth - 1; //Iegūst virknes garumu pikseļos

        //Ievieto visus elementa simbolus glabātuvē
        while (lastElem.end !== strWdt) {
                lastElem = self.getGlyph(string, lastElem.end);
                self.selection.push(lastElem);

        }
        //Vārda/teikuma beigās ievieto tukšumu(atstarpe)
        self.selection[self.selection.length - 1].glyph += " ";

        //Iekrāso iezīmētos reģionu
        self.colourSelection();
},
```

1.5. Metode textAccessElement.getLineHeight

```
getLineHeight: function() {
	var self = this;

	//Ja nav iestatīts fonta izmērs, iestata noklusējuma vērtību
	if (self.fontSize) {
		var size = self.fontSize + " ";
	} else {
		var size = "10px ";
	}

	//Izveido pagaidu elementu teksta ievietošanai un to paslēpj
	var line = document.createElement('div');
	var body = document.body;
	line.style.position = 'absolute';
	line.style.whiteSpace = 'nowrap';
	line.style.font = size + self.fontFamily;
	body.appendChild(line);

	//Ievieto 10 "m" simbolus iestatītajā fontā, precizitātei
	//"m" sniedz ticamu rezultātu vairumam fontu
	var text = 'mmmmmmmmmm';
	line.textContent = text;

	//Izveido 1px lielu elementu, kuru pielīdzina teksta bāzlīnijai, lai
aprēķinātu bāzlīnijas nobīdi
	var span = document.createElement('span');
	span.setAttribute("aria-hidden", true);
	span.style.display = 'inline-block';
	span.style.overflow = 'hidden';
	span.style.width = '1px';
	span.style.height = '1px';
	line.appendChild(span);

	//Iegūst teksta augstumu un bāzlīniju
	var baseline = span.offsetTop + span.offsetHeight;
	var height = line.offsetHeight;
	document.body.removeChild(line);
	//Atgriež augstumu un bāzlīniju
	return {
		baseline: baseline,
		height: height
	};
}
```

1.6. Metode textAccessElement.updateText

```
//Atjaunina teksta parametrus.
//Parametri:
//redraw - true/false (vai tekstu vajag pārzīmēt uzreiz);
//string - simbolu virkne (zīmējamā simbolu virkne);
//x - skaitlis (zīmēšanas x koordināta, kur 0 ir kreisā canvas mala);
//y - skaitlis (zīmēšanas y koordināta, kur 0 ir canvas augšējā mala);
//fontSize - simbolu virkne (zīmējamā teksta izmērs. Atbalsta gan pt, gan px
tipus);
//fontFamily - simbolu virkne (zīmējamā teksta fonts, atbilstoši CSS tipa fontu
pierakstam);
//fontStyle - simbolu virkne (teksta stils - normal, italic, bold, oblique);
//fontVariant - simbolu virkne (fonta variants - normal, small-caps);
//fontWeight - simbolu virkne (fonta biezums, atbilstoši CSS tipa pierakstam);
//colour - simbolu virkne (teksta krāsa);
//textAlign - simbolu virkne (teksta pielīdzināšana kādai no malām);
//baseline - simbolue virkne (teksta bāzlīnija)
updateText: function(redraw, string, x, y, fontSize, fontFamily, fontStyle,
fontVariant, fontWeight, colour, textAlign, baseline) {
	var self = this;

	if (redraw) {
		self.clearText();
	}
	if (string) {
		this.string = string;
	}
	if (x) {
		this.x = x;
	}
	if (y) {
		this.y = y;
	}
	if (fontSize) {
		this.fontSize = fontSize;
	} else {
		fontSize = this.fontSize;
	}
	if (fontFamily) {
		this.fontFamily = fontFamily;
	} else {
		fontFamily = this.fontFamily;
	}
	if (fontStyle) {
		this.fontStyle = fontStyle;
	}
	if (fontVariant) {
		this.fontVariant = fontVariant;
	}
	if (fontWeight) {
		this.fontWeight = fontWeight;
	}
	if (colour) {
		this.colour = colour;
	}

	this.styleString = this.getStyle(fontSize, fontFamily, fontStyle,
fontVariant, fontWeight, colour);
	this.offCanvas.ctx.font = this.styleString;
```

```
	this.textWidth =
Math.ceil(this.offCanvas.ctx.measureText(this.string).width);
	if (textAlign) {
		self.compensateAlignment(textAlign);
	} else this.compensateAlignment(this.textAlign);

	this.lineHeight = this.getLineHeight().height;
	this.baseline = this.getLineHeight().baseline;

	if (baseline) {
		this.compensateBaseline(baseline);
	} else this.compensateBaseline(this.baselineType);
	this.rectY = this.y - this.baseline;

	if (this.accessCanvas.screenReader) {
		self.readableElement.textContent = this.string;
	}

	if (redraw) {
		self.drawText();
	}
}
```

1.7. Metodes bāzlīnijas un līdzinājuma kompensēšanai

```
//Funkcija, kura, atkarībā no teksta pielīdzināšanas parametra vērtības, pārvieto
teksta x koordinātu
compensateAlignment: function(textAlign) {
	switch (textAlign) {
		case "start":
			break;
		case "center":
			this.x = this.x - this.textWidth / 2;
			break;
		case "end":
			this.x = this.x - this.textWidth;
			break;
		case "left":
			break;
		case "right":
			this.x = this.x - this.textWidth;
			break;
	}
},

//Bāzlīnijas kompensācijas funkcija - pielāgo teksta y koordinātu
compensateBaseline: function(baseline) {
	switch (baseline) {
		case "alphabetic":
			break;
		case "bottom":
			this.y = this.y - (this.lineHeight - this.baseline);
			break;
		case "hanging":
			this.y = this.y + (this.lineHeight - this.baseline) * 2.5;
			break;
		case "ideographic":
			this.y = this.y - (this.lineHeight - this.baseline);
			break;
```

```
            case "middle":
                this.y = this.y + (this.lineHeight - (this.lineHeight - 
this.baseline) * 2) / 2;
                break;
            case "top":
                this.y = this.y + (this.lineHeight - (this.lineHeight - 
this.baseline));
                break;
        }
}
```

Bakalaura darbs „Teksta iegūšana no HTML5 canvas elementiem” izstrādāts Latvijas Universitātes Datorikas fakultātē.

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai.
Autors: Vilnis Laipnieks ____________ ___.05.2016.

Rekomendēju/nerekomendēju darbu aizstāvēšanai
Vadītāja: docente Dr. dat. Darja Solodovņikova ____________ ___.05.2016.

Recenzents: ______________________

Darbs iesniegts Datorikas fakultātē ___.05.2016.
Dekāna pilnvarotā persona: metodiķe Ārija Sproģe ____________

Darbs aizstāvēts bakalaura gala pārbaudījuma komisijas sēdē
___.06.2016. prot. Nr. ___
Komisijas sekretārs: ____________________ ____________