平成20年度 修士論文

ポインティング操作が
キーボードのみで可能な
Webブラウジングインタフェースの
設計と実装

電気通信大学 大学院情報システム学研究科

情報システム基盤学専攻

0753018 豊田 義純

指導教員 多田 好克 教授
小宮 常康 准教授
大森 匡 准教授

提出日 平成21年1月29日

目 次

図 目 次

表 目 次

第 1 章

はじめに

現在，GUI 操作が可能な OS が広く普及している．ここで，GUI 操作に最もよく用いられる入力デバイスとしてマウスが挙げられる．しかし計算機の操作は GUI 操作だけではなく，文字入力操作を含んでいる．文字入力操作によく用いられる入力デバイスはキーボードである．つまり，一般に計算機の操作を行う際には，複数の入力デバイスを持ち替えながら作業を行う必要がある．

特に Web ブラウザの操作では，この持ち替えが頻繁に行われる．検索エンジンを利用するときには，フォームへ検索キーワードを入力しなければならない．一方リンクやプルダウンといった，Web コンテンツの選択にはグラフィクスカーソルの操作が欠かせない．Web ブラウザの操作はユーザが日常的に経験する操作であるため，入力デバイスを持ち替える回数が増加すると，作業効率の低下を招くと考えられる．

Card らの調査結果では，キーボードからマウスへ手を移動する操作一回につき 0.4 秒かかるとしている [1]．Web ブラウザの操作において，文字入力操作の後にグラフィクスカーソルを用いて Web コンテンツを選択し，再度文字入力を行うと，入力デバイスの持ち替えだけで約 1 秒かかると言える．作業効率の向上を考えると，この時間の蓄積は無視できないという報告 [6] もある．

ここで，単一の入力デバイスのみを用いれば，この持ち替えの手間は発生しない．しかし通常キーボードで行う操作の全てを，マウスのみによって行うのは煩雑である．通常，文字の入力操作にはキーボードが用いられる．この操作をマウ

スのみによって行わなければならない．たとえば，ソフトウェアキーボードによって，マウスのみでも文字入力は可能である．しかし，文字の入力を行うたびにグラフィクスカーソルを操作し選択する方法では，キーボードで行う文字入力と比べ，操作に必要となる時間は甚大である．

そこで本研究は，Web ブラウザの操作にキーボードのみを用いる．通常マウスで行っている，グラフィクスカーソルによるポインティング操作は，全てキーボードで行う．また，グラフィクスカーソルをキーボードで操作することにより，操作効率の向上を図る．

テキストの範囲選択を考慮しなければ，クリック操作が行われる可能性がある Web コンテンツは限定される．ユーザは，リンクやプルダウンなどクリッカブルな Web コンテンツに対してクリック操作を行う．そこで本研究は，それらの Web コンテンツの表示領域を考慮する．クリッカブルな Web コンテンツのみに対してクリック操作が行われると仮定し，それらの表示領域上だけをグラフィクスカーソルが移動する操作を可能にした．

本研究は，一般的な Web ブラウザとして Mozilla Firefox 3.0 を対象とする．Firefox は GUI 操作が可能な Web ブラウザであるため，通常マウスとキーボードの両方が操作に用いられる．そこで Firefox を対象とし，その操作を可能とするような UI を設計し，実装する．また，Firefox の拡張機能として実装することにより，導入コストが低い UI を実現する．

第 2 章

背景

Web ブラウザの操作に必要となる，マウスとキーボードとの持ち替えは手間である．Web ブラウザに表示されたリンクを順に選択していく操作が主であれば，キーボードを用いる必要はない．しかしフォームへの文字入力と，リンクを選択していく操作を繰り返し行う場合には，頻繁にこれらを持ち替える必要がある．この例として，検索キーワードを変更しながら何度も Web 検索エンジンを利用する場合が挙げられる．

Web 検索エンジンでのキーワード検索を補助するよう，Web ブラウザが機能を提供していることがある．それらの機能の多くは，Web ブラウザのウィンドウの一部に設けられた専用のフォームへの文字入力と，Web 検索エンジンでのキーワード検索とを関連付けるものである．この機能を用いれば，キーボードのみでもキーワード検索を簡単に行うことができる．

しかし，現在閲覧している Web ページ内に，ある特定の語句が含まれているかどうか検索を行うことがある．この語句の入力にはキーボードが用いられることが極めて多い．ユーザはいつもその検索を行うわけではなく，閲覧している Web ページの内容によって検索するかどうかを決定している．所望のページが表示されたと思い，マウスに持ち替えて Web ページの閲覧を始めたユーザが，ここでキーボードへの持ち替えを余儀なくされることも多い．

一方，ユーザの操作によって表示が動的に変更されるような Web コンテンツが増加している．それらの中には，音声コンテンツや動画コンテンツなどが含まれ

る．一般的な Web ブラウザはそれらの表示が可能であるため，多くのユーザはそれらの閲覧が可能である．

マウスとキーボードとの持ち替えを嫌い，キーボードのみで操作可能な Web ブラウザをユーザが利用したとする．しかし，その Web ブラウザに表示できない Web コンテンツがあるとき，それらのコンテンツを閲覧した経験のあるユーザはストレスを感じうる．たとえば w3m[13] や Lynx[14] というテキストブラウザは，全ての操作がキーボードのみで可能であるが，それらには表示できない Web コンテンツがある．つまりそういったユーザには，キーボードのみで操作可能であり，かつ一般的な Web コンテンツが表示可能であるような Web ブラウザが必要である．

そこで本研究は，一般的な Web ブラウザとして Mozilla Firefox を対象とした．Firefox はマウスによる GUI 操作が可能な Web ブラウザである．また，Firefox は音声コンテンツや動画コンテンツを含む，一般的な Web コンテンツの表示が可能である．本研究は，Firefox の操作をキーボードのみで行う時に必要となる機能を UI として設計し，実装を行った．また，UI は Firefox の拡張機能として実装したため，Firefox を利用しているユーザであれば誰でも利用が可能なものである．

第 3 章

UIに対する要求事項

キーボードのみによる Web ブラウザの操作を，UI を用いることで可能にする必要がある．また，その操作効率が向上することが望ましい．その実現のために求められる要求事項として，以下の三点が挙げられる．

1. キーボードによる Web ブラウザの操作が効率的に行える必要がある．
2. フォーム等に対して行う文字入力操作とキーボードによって行う Web ブラウザの操作を分けて考える必要がある．
3. マウスによる操作がキーボードのみでも実行可能であり，かつユーザに対し視覚的なフィードバックを与える必要がある．

これら三点について，それぞれ次節以降に述べる．また，以降 “マウス” を “マウスを含む，ポインティングデバイスの総称” と定義し，述べる．

3.1 キーボード操作の効率化

キーボードのみで Web ブラウザの操作を可能とするよう，UI の設計を行う必要がある．そのために通常キーボードで行っている計算機操作を考える．それと似た操作方法によって Web ブラウザの操作が行えれば，その操作効率の向上が見込める．

エディタの操作はキーボードと深い関係を持つ．またエディタの操作は基本的な計算機操作である．加えて，エディタの操作に慣れているユーザは多い．そのため，エディタと似た操作方法で Web ブラウザの操作が行えれば，多くのユーザの作業効率が向上する可能性がある．

3.2 文字入力操作と Web ブラウザ操作の切り分け

通常 GUI 操作が可能な Web ブラウザの機能は，Web ブラウザが提供する階層的なメニューを順次選択していくことによって実行が可能である．そのうちの機能のいくつかは，ショートカットコマンドを用いても実行が可能である．ショートカットコマンドとは，Ctrl キーや Shift キーのような修飾キーと他のキーを同時に入力することによって，Web ブラウザの操作を簡単に行うためのものである．たとえば，現在閲覧しているウィンドウとは別に，新たなウィンドウで Web ブラウザを開く操作は Ctrl キーと N キーを同時に入力することで行えることが多い．

フォームに対して，文字を入力することだけを考えれば Ctrl キーや Alt キーを用いる必要はない．ショートカットコマンドが必ずそれらのキーを利用するものだとして設計を行えば，たとえ文字の入力操作中であっても Web ブラウザが提供する機能の実行は可能である．

修飾キーを利用するとき，それを含む複数のキーを同時に入力しなければならない．つまり，修飾キーの利用は手間となりうる．そこで修飾キーを利用せずに，ショートカットコマンドの実行を可能にすれば，Web ブラウザの操作効率の向上が見込める．

修飾キーを利用することなく Web ブラウザが提供する機能の実行を実現するためには，ショートカットコマンドを変更するだけではいけない．その実現のためには，ユーザによるキー入力の対象までもを考慮し，それらの違いを扱う必要がある．たとえばフォームに対する文字の入力操作時における，キー入力の対象は

フォーム，つまり特定の Web コンテンツである．一方ショートカットコマンドの入力時における，キー入力の対象は Web ブラウザである．

3.3 マウス操作の置き換えと，視覚的なフィードバック

通常ユーザは，OS が提供するグラフィクスカーソルをマウスをによって操作し，それによって Web コンテンツの選択を行っている．キーボードのみによる Web ブラウザの操作を行う時は，キーボードのみで Web コンテンツの選択が可能である必要がある．

マウスには物理的な操作が必要である．それらの操作が行われると，グラフィクスカーソルは表示位置の変更等，視覚的なフィードバックをユーザへ与える．キーボードのみで Web コンテンツを選択する手法は複数挙げられるが，そのいずれもグラフィクスカーソルを用いるものではない．つまり，それらの手法を用いると，グラフィクスカーソルによる視覚的フィードバックをユーザに与えることができない．

Web コンテンツの選択にグラフィクスカーソルを用いれば，ユーザは視覚的フィードバックを得ることができる．その実現のためには，キーボードのみによってグラフィクスカーソルの操作が可能である必要がある．

キーボードのみによって，Web コンテンツを選択する手法として代表的なものを次節以降に挙げる．

Tab キーや Enter キー等の利用

通常 Web ブラウザは，クリッカブルな Web コンテンツを Tab キー等で順にフォーカスしながら巡る機能や，フォーカスされた Web コンテンツを Enter キー等によって選択するという機能を提供している．この機能を利用すれば，キーボードのみで Web コンテンツの選択が可能である．しかしこの手法では，グラフィクスカー

ソルを用いない．そのため，通常の視覚的フィードバックをユーザに与えることができない．

Web コンテンツのラベリング

クリッカブルな Web コンテンツに対し，一意にラベリングを行う手法 [16][18] がある．これらの手法では，Web コンテンツの選択と，その Web コンテンツに対応付けられたラベルのキータイプとを等価であるものとして定義している．この定義により，キーボードのみで Web コンテンツの選択は可能である．しかしこれらの手法でも，グラフィクスカーソルを用いない．そのため，通常の視覚的フィードバックをユーザに与えることができない．

第 4 章

UIの設計

まず，UIが対象とするWebブラウザは，一般的なWebコンテンツを表示できる必要がある．ここで言う一般的なWebコンテンツとは，ユーザ数が多いWebブラウザで表示可能なWebコンテンツを指す．その一例として，音声コンテンツや動画コンテンツなどが挙げられる．これらのコンテンツを閲覧した経験のあるユーザにとっては，UIの利用によって閲覧できるコンテンツが制限されてしまうことにストレスを感じうる．そこで本研究はそのようなWebブラウザの一般例として，Mozilla Firefoxを対象とするUIを設計する．

次に，前章で述べた，UIに対する要求事項を満たすような設計を行う必要がある．

まず，キーボードのみによって行うWebブラウザの操作効率の向上を図るため，通常キーボードを用いて行う計算機操作を考える．その操作と同じ方法によってWebブラウザの操作が行えれば，操作効率の向上が見込める．ここでエディタの操作はキーボードと深い関係を持つ．そこで，エディタと似た操作方法によってWebブラウザの操作を可能にするよう，UIの設計を行う．

次にキー入力の対象の違いによって操作を分けるよう，設計を行う．Webブラウザに対して行われるキー入力は，Webコンテンツを対象とするものとWebブラウザを対象とするものとに大別できる．Webコンテンツを対象とするキー入力は，操作に通常用いられる入力デバイスの違いによって二つに大別する．Webブラウザを対象とするキー入力については，表示するWebページを変更する操作であるかどうかという点を考慮する．

またキーボードによって行う，グラフィクスカーソルの操作を考える．グラフィクスカーソルの位置の変更における，キーボードの操作に意味を持たせるよう，設計を行う．加えてWebコンテンツ選択の手間の軽減を図るため，Webブラウザのウィンドウ内にあるクリッカブルなWebコンテンツだけを選択対象とするよう，設計を行う．

OSがマウスキーという機能を提供していることがある．マウスキーとは，OSが提供するグラフィクスカーソルをキーボードによって操作する機能である．WindowsやMachintoshといった一般的なOSはマウスキー機能を備えている[11][12]．マウスキーは，ピクセル単位でグラフィクスカーソルの移動操作を行うものである．しかしユーザは，Webコンテンツの表示領域に応じて，グラフィクスカーソルを操作する．そこで，Webコンテンツの選択効率の向上のため，その選択専用のグラフィクスカーソルを設計する．ここで設計するグラフィクスカーソルは，Webブラウザのウィンドウ内にあるWebコンテンツの表示領域を考慮するものである．また，これはOSが提供するものとは異なるため“仮想カーソル”と呼び区別する．

これらについて次節以降に述べる．

4.1 対象とするWebブラウザ

Webブラウザは多数存在する．現在，一般的なWebブラウザはGUI操作が可能なものである．ここでGUI操作とは，グラフィクスカーソルによって項目の選択を行う操作を指す．これらには，視覚化された階層的なメニューの操作や，同時に開かれている複数のWebページをタブによって切り替える操作，ウィンドウ内に描画されたボタンの押下によってWebブラウザの機能を実行する操作等が含まれる．GUI操作が可能なWebブラウザとして，Internet ExplorerやSafari，Operaなどが挙げられる．

GUI操作はマウスによってなされることが多く，マウスの操作は直感的で分か

りやすい．そのため，広く普及している OS の多くは，Web ブラウザのデフォルトを，GUI 操作が可能な Web ブラウザとして設定している．

一方，キーボードのみで操作可能な Web ブラウザとして w3m[13] や Lynx[14] のようなテキストブラウザが挙げられる．しかし，それらでは一部の Web コンテンツが表示できない場合がある．たとえば Flash や Java アプレット等，マルチメディアに関連のある Web コンテンツは一般的になりつつあるが，テキストブラウザでは，これらを表示できない．そのためキーボードのみで Web ブラウザを操作し，一般的な Web コンテンツを閲覧したいと思う時，テキストブラウザの利用では問題が解決しない．

そこで本研究は，GUI 操作が可能であり，かつ一般的な Web ブラウザの例として Mozilla Firefox 3.0 を対象とした．Mozilla Firefox は，Windows，Mac OS X，Linux のそれぞれに対し，正式なサポートがなされている．この点より，Mozilla Firefox は一般的であると言える．

なお以降，特に断りがなければ “Mozilla Firefox 3.0” を “Firefox” と表記する．

4.2 キーボードによる Web ブラウザの操作の効率化

本研究は，キーボードのみで Web ブラウザの操作を行うことを目的としている．また，それにより Web ブラウザの操作の効率向上が望ましい．そこでまず，通常キーボードで行っている計算機操作を考える．それと似た操作方法を UI に持たせるよう，設計する．

エディタの操作はキーボードと深い関係を持つ．またエディタの操作は基本的な計算機操作である．それゆえ，エディタの操作に慣れているユーザは極めて多い．そこでエディタと似た操作方法によって Web ブラウザの操作を可能にするよう，UI の設計を行う．

エディタは多数存在する．その中でも，ホームポジションからあまり手を離さず

に利用できるものは，その操作効率が高い．本研究はそのようなエディタの中から vim を取り上げた．UI には vim と似た操作性を持たせる．これにより，Web ブラウザの操作効率の向上を図る．

vim は，挿入モードや編集モードを適宜切り替えることによって，効率的な文書編集を可能にするエディタである．また，vim は Ctrl キーや Alt キーをあまり用いずに利用できる．3.2 節では，修飾キーの入力が手間となることもあると述べた．UI の操作を vim と似た方法によって行うことにより，この問題を避けることができる．

また 3.2 節ではあわせて，ユーザからのキー入力の対象の違いを扱う必要があると述べた．そこで本研究では，これらの対象の違いを，vim におけるモードの違いと対応付ける．またユーザがキー入力の対象を変更し操作を行う場合，その対象の変更に伴うモードの切り替えを必要とするよう設計する．

4.3 Web ブラウザの操作の種別分け

本節では 3.2 節に述べた，キー入力の対象には複数の種類があることを問題にする．キーボードのみで Web ブラウザの操作を行う時，ユーザが期待する操作が異なれば，キー入力の対象は異なる．そこでその対象の違いを利用し，Web ブラウザ操作の種別分けを行う．

4.3.1 Web コンテンツを対象とする操作

ユーザが Web コンテンツを対象として操作を行うとき，それらの操作には，通常キーボードとマウスとが用いられる．たとえば，フォームを対象として行われるキー入力は文字入力の操作を意味する．また，リンクを対象として行われるクリック操作はリンク先の Web ページの表示操作を意味する．

ここで文字入力の操作は両手をキーボードの上に乗せて行うものと考える．す

ると，Web コンテンツに対してユーザが行う全ての操作は，キーボードを用いるものなのか，マウスを用いるものなのか，そのいずれか一方である．

入力デバイスの持ち替えには時間がかかる．ユーザはその時間をかけてまで，操作の対象となる Web コンテンツを切り替えていると言える．そこで Web コンテンツを対象とする操作を，それに用いられる入力デバイスによって以下の二つに大別する．

- 通常キーボードを用いる操作
- 通常マウスを用いる操作

上記について，そのそれぞれの設計を次節以降に述べる．

文字入力の操作

通常キーボードを用いて行っている操作として，フォームへの文字入力が挙げられる．また文字入力が行われる時は，常にフォームがフォーカスされている．そこで，フォームがフォーカスされている時は文字入力が行われるものとして設計した．

また，UI には vim と似た操作性を持たせると述べた．UI の操作方法は統一する必要がある．つまり，フォームへの文字入力の操作も vim と同じように行えるよう，設計する必要がある．

フォームに対して行う文字入力の操作は，一般の文書編集の操作と比べ，規模が非常に小さい．簡単な文字入力の操作を行うために，UI には以下に挙げる vim の操作のみを可能にするよう，設計を行った．ここでは，文字の入力や文書の編集の位置を示すカーソルを “編集カーソル” と呼び，グラフィクスカーソルと区別する．

- 挿入モードにおける，文字入力の操作
- 編集モードにおける，文書編集の操作

- 挿入モードと編集モードの切り替え操作
- 編集モードにおける，以下の操作
  - 編集カーソルの移動操作
  - 文字単位での編集操作
  - 行単位，単語単位での編集操作

なお編集モード中にあるときは，編集カーソルを表示する必要がある．そのため，このときはフォームがフォーカスされているよう，設計した．

グラフィクスカーソルの操作

通常マウスを用いて行っている操作として，Web コンテンツを選択する操作が挙げられる．この操作は，文字入力を行っていない時は常に行われる可能性がある．そこで，簡単にグラフィクスカーソルの操作が行えるよう，UI の設計を行った．

グラフィクスカーソルの操作と文字入力の操作とは，分けて設計している．そこで，グラフィクスカーソルの操作へ移行するためのコマンドを設計した．このコマンドは “通常マウスを用いて行う操作への移行” という意味を込め，表 4.1 に示すような設計を行った．またグラフィクスカーソルの操作への移行は，フォーカスされている Web コンテンツがない場合のみ可能であるよう，設計した．前述文字入力の操作における編集モードにおいては，フォームがフォーカスされている．よって，編集モードにあるときはグラフィクスカーソルの操作へ移行できない．そこで

表 4.1: グラフィクスカーソル操作への移行操作の方法

コマンド	実行される操作
m	グラフィクスカーソル操作への移行

編集モードにあるとき，そのフォームからフォーカスを外し，グラフィクスカーソルの操作へ移行可能な状態へ遷移するためのコマンドを設計した．この操作方法を表 4.2 に示す．また，グラフィクスカーソルの操作と文字入力の操作とを行う状態の遷移を，編集モード，挿入モードとあわせて状態遷移図として図 4.1 に示す．

表 4.2: グラフィクスカーソルへ移行可能な状態へ遷移するための操作方法

コマンド	実行される操作
Esc	フォームからフォーカスを外す

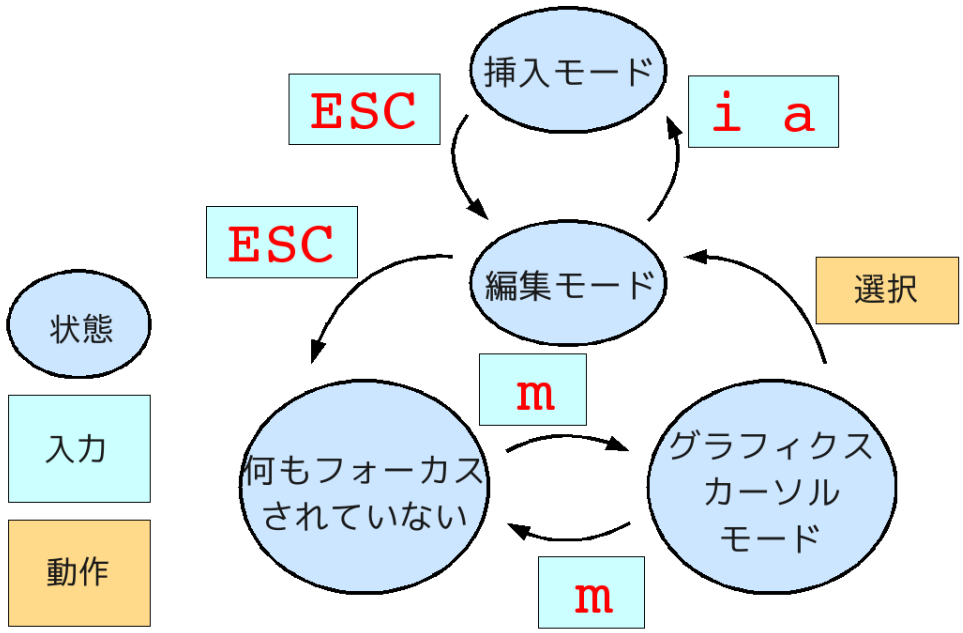

図 4.1: グラフィクスカーソルの操作と文字入力の操作における状態遷移図

4.3.2 Web コンテンツを対象としない操作

Web コンテンツがフォーカスされていない状態におけるユーザのキー入力は，Web ブラウザが提供する機能の実行を意味する．この操作にはマウスが用いられることが多いが，Web ブラウザが提供するショートカットコマンドを用いることで実行が可能なこともある．そのため，用いられる入力デバイスによって操作を分けることはできない．

ここでは，それらの操作について，以下の二つの概念を導入する．

- ナビゲーション操作
- メニュー操作

また，そのそれぞれを以下のように定義する．

- ナビゲーション操作
  - 閲覧する Web ページを変更する操作
    - 閲覧履歴を “戻る”，“進む” 操作
    - 表示している Web ページを，Web ブラウザに再度読み込ませる操作
    - アドレスバーをフォーカスする操作
    - 検索バーをフォーカスする操作
  - Web ページの閲覧箇所を変更する操作
    - 表示している Web ページのスクロール操作
  - タブの操作
    - 新たなタブを表示する操作
    - 最前面に表示するタブを切り替える操作
    - 最前面に表示されているタブを閉じる操作

＊ 全てのタブを閉じ，Web ブラウザを終了する操作

- メニュー操作
  - Web ブラウザが提供するメニューバーから操作可能である操作

Web ページのスクロール操作を除けば，全てのナビゲーション操作はメニューバーを用いても実行が可能である．すなわち，ナビゲーション操作はほぼメニューバー操作に含まれている．この様子を図 4.2 に示す．

多くのナビゲーション操作は，ショートカットコマンドによる実行が可能である．そこで，ナビゲーション操作に対してあらかじめ設定されているショートカットコマンドは，そのまま利用可能とするよう，UI を設計した．

また頻繁に用いられるナビゲーション操作に対しては，新たなショートカットコマンドを設計した．その対象とする操作を以下に挙げる．

- 閲覧履歴を “戻る”，“進む” 操作
- 表示している Web ページのスクロール操作
- 最前面に表示するタブを切り替える操作

上記の操作に対するショートカットコマンドの設計を次節以降に述べる．

閲覧履歴を戻る/進む操作に対するショートカットコマンドの設計

vim には，編集を行ってきたファイルの履歴を保存する機能がある．これを用いると，直前に編集していたファイルを開く操作が行える．vim における，その操作方法を表 4.3 に示す．

文字入力の操作中でなければ，それらに用いられるコマンドは別の用途に利用可能である．そこで “戻る”，“進む” 操作には，編集カーソルを左右へ移動する操作に対応するショートカットコマンドを割り当てるよう，設計した．これにより，

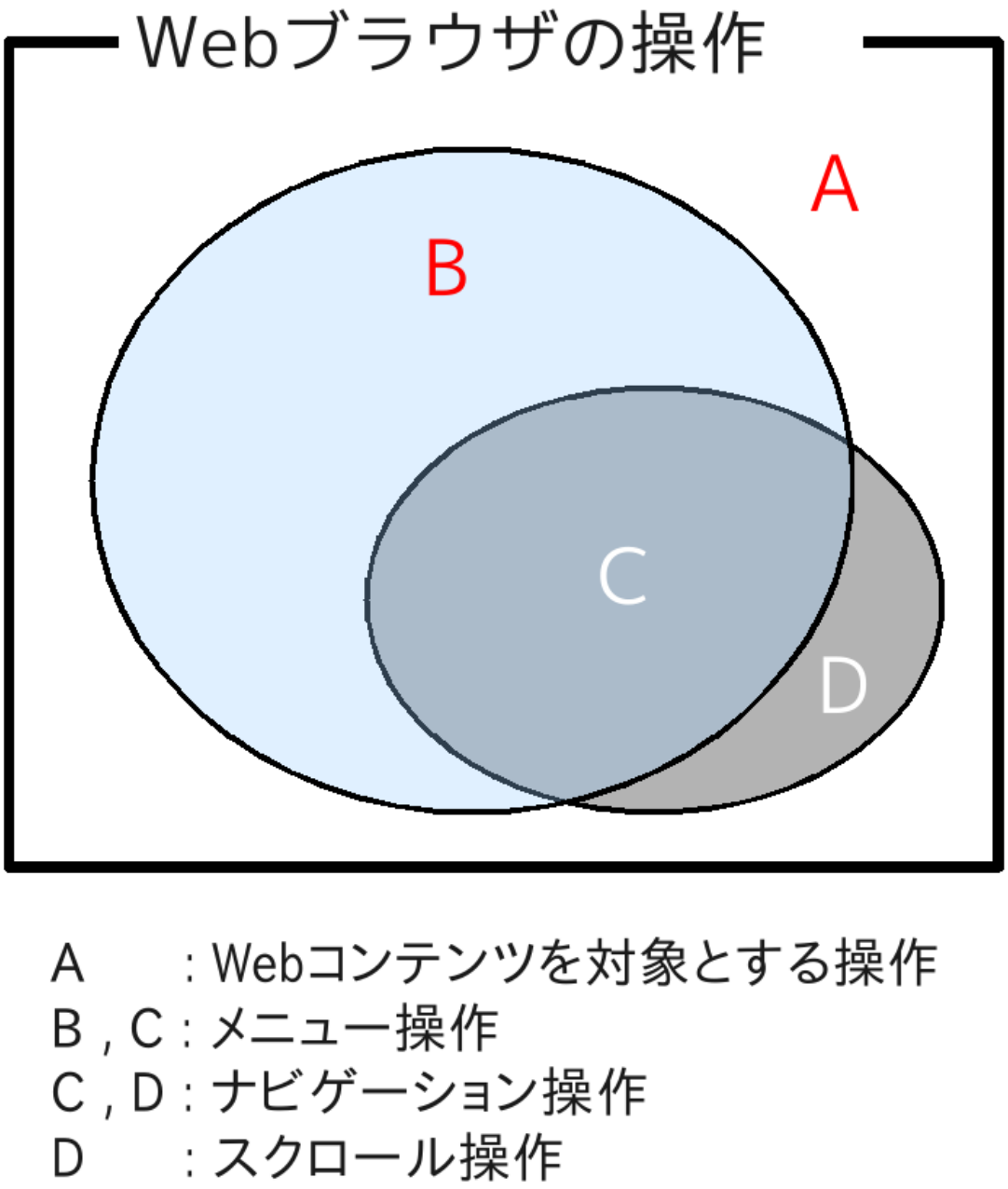

図 4.2: Web ブラウザの操作における，メニュー操作とナビゲーション操作および Web コンテンツを対象とする操作の関係

表 4.3: vim の編集モードにおける，履歴に基づき編集するファイルを変更するための操作方法

コマンド	実行される操作
:bp	編集履歴を一つだけ戻る
:bn	編集履歴を一つだけ進む

表 4.3 に示すよりも少ない打鍵数によって，操作の実行が可能になる．ここで行ったショートカットコマンドの設計を表 4.4 に示す．

表 4.4: UI における，閲覧履歴を “戻る”，“進む” 操作の方法の提案

コマンド	実行される操作
h	閲覧履歴を一つだけ戻る
l	閲覧履歴を一つだけ進む

スクロール操作に対するショートカットコマンドの設計

vim は画面のスクロール操作を行うための機能を備えている．この操作方法を表 4.5 に示す．

一般的なページャである less は，そのスクロール操作の方法が vim と似ている．画面のスクロール操作においては，より少ない打鍵数によって vim と同等の操作が可能である．そこで，この操作を less の操作方法と対応付けるよう，設計を行った．ここで行ったショートカットコマンドの設計を表 4.6 に示す．

表 4.5: vim の編集モードにおける，画面のスクロールを行うための操作方法

コマンド	実行される操作
Ctrl-e	画面を一行だけ下にスクロールする
Ctrl-y	画面を一行だけ上にスクロールする
Ctrl-f	一画面分だけ，画面を下にスクロールする
Ctrl-b	一画面分だけ，画面を上にスクロールする
G	ファイルの末尾の行が表示されるように，画面を下にスクロールする
gg	ファイルの先頭の行が表示されるように，画面を上にスクロールする

表 4.6: UI における，Web ページのスクロール操作の方法の提案

コマンド	実行される操作
j	画面を一行だけ下にスクロールする
k	画面を一行だけ上にスクロールする
f	一画面分だけ，画面を下にスクロールする
b	一画面分だけ，画面を上にスクロールする
G	ページの末尾が表示されるように，下方向にスクロールする
g	ページの先頭が表示されるように，上方向にスクロールする

タブの切り替え操作に対するショートカットコマンドの設計

Firefox は，複数の Web ページを同時に開くことができる．このとき開かれる Web ページは，それぞれ異なる複数のタブによって保持される．しかし一度に選択できるタブは一つであり，選択されているタブが保持している Web ページしか表示できない．そのため，同時に開かれている他の Web ページを閲覧するためには，選択されているタブを切り替える操作が必要である．

選択されているタブの切り替え操作を行う時，現在選択されているものの隣にあるタブを選択できればよい．そこで，vim の編集カーソルの操作と似た方法によって切り替え操作が可能になるよう，ショートカットコマンドを設計した．この設計を表 4.7 に示す．

表 4.7: UI における，タブの切り替え操作の方法の提案

コマンド	実行される操作
H	現在表示されているタブの左隣のタブを開く
L	現在表示されているタブの右隣のタブを開く

4.4 グラフィクスカーソルの操作の設計

通常ユーザはマウスを用いてグラフィクスカーソルを操作する．また，それによって Web ブラウザの操作を行う．そこでキーボードのみによる，グラフィクスカーソルの操作を設計する．

UI には vim と似た操作方法を与える．vim における編集カーソルの基本的な操作として，上下左右の四方向への移動操作が挙げられる．そこでグラフィクスカーソルの操作は，これと対応付けて行うよう，設計した．

vim の編集カーソルの操作と似た操作方法を持つものとして，Rogue が挙げられる．Rogue はその全ての操作がキーボードのみで可能な，キャラクタベースのゲームである．またその目的は様々なアイテムを駆使しながら敵を倒し，進んでいくことにある．

vim の編集カーソルの移動操作は上下左右の四方向のみ可能である．一方 Rogue における，主人公の位置を変更する操作は vim の編集カーソルの移動できる四方向に加え，斜め四方向への移動もできる．一度に移動できる方向が増えれば，操作効率が向上する可能性がある．そこでグラフィクスカーソルが Rogue と同じように操作できるよう，設計した．ここで述べた移動操作の方法については，4.4.1 節で具体的に述べる．

また，通常マウスによって行われる操作のように，グラフィクスカーソルの表示座標を動的に変更するよう設計した．これによって，ユーザに視覚的なフィードバックを与えることができる．加えて，グラフィクスカーソルの表示座標に対するクリック操作が行えるよう，設計した．この操作が簡単に実行できるよう，表 4.8 に示すようなコマンドを設計した．

表 4.8: グラフィクスカーソルのクリック操作の方法

コマンド	実行される操作
スペース	グラフィクスカーソルによるクリック

ユーザが選択する Web コンテンツとして，リンクやフォーム，ラジオボタンやプルダウンなど，マウスのクリック操作等によって状態が変化するものが挙げられる．ここではそれらをクリッカブルな Web コンテンツと呼ぶことにする．

Web コンテンツの範囲選択，特に Web ページ内のテキストの範囲選択を考えなければ，選択される可能性がある Web コンテンツはクリッカブルなものに限定さ

れる．そこで，クリッカブルなWebコンテンツのみを選択対象とすることにより，選択効率の向上を図る．また，Webブラウザのウィンドウ内に表示されているWebコンテンツのうち，クリッカブルなもののみを選択の対象とするよう，グラフィクスカーソルの操作性を設計した．

ピクセル単位でしかグラフィクスカーソルを操作できない状況よりも，操作の粒度を適宜変更しながらWebコンテンツの選択を行った方が作業効率は高くなる[4]．そこでグラフィクスカーソルの移動が以下の三種類の粒度を持つよう，設計した．

粒度：小 ピクセル単位での移動

粒度：中 クリッカブルなWebコンテンツの表示領域を考慮した移動

粒度：大 Webブラウザのウィンドウの端への移動

上記，ピクセル単位での移動とは，一回の移動量が，あらかじめ決められたピクセル幅である移動のことを指す．このピクセル幅は非常に小さいものであるため，グラフィクスカーソルの位置の微調整に用いることができる．

また上記，**粒度：中**，**粒度：大**である二つの操作は，グラフィクスカーソルの移動方向を示すコマンドと修飾キーとが同時に入力された時になされるものとして設計した．言い換えれば，修飾キーの入力によって移動操作の粒度の指定が可能であるよう，設計した．なお，ここで述べた移動操作の方法については，4.4.1節で具体的に述べる．

ピクセル単位での移動操作と比べると，他二種類の操作におけるグラフィクスカーソルの移動距離は長い．そのため一度移動操作を誤ってしまうと，その位置が目標から大きく外れる可能性がある．そこでグラフィクスカーソルが一度に長い距離を移動する時，その移動を中断する操作も準備した．これによって操作を誤った時の目標とのずれを最小限に抑えることができる．

移動を中断する操作は，ホームポジションから手を離さずに入力可能であることを目的とした．この設計を表 4.9 に示す．

表 4.9: グラフィクスカーソルの移動を中断する操作の方法

コマンド	実行される操作
i	移動中のグラフィクスカーソルを停止させる

ユーザが右利きであれば，マウスの操作も右手で行うことが多い．そこでグラフィクスカーソルの操作を右手のみで行えるよう，設計した．あわせて，グラフィクスカーソルによる選択の可能性がある Web コンテンツを絞り込むことにより，選択効率の向上を図る設計を行った．

そのそれぞれの設計について，次節以降に述べる．

4.4.1 グラフィクスカーソルの移動

以降，Rogue の主人公の動きとグラフィクスカーソルの動きとを対応付けて述べる．

粒度：小　ピクセル単位での移動

Rogue における，主人公の移動操作には，移動方向に対応するコマンドの入力が必要である．あるコマンドが入力された時，Rogue の主人公は対応する方向へ一マス移動する．そこで，このコマンドはグラフィクスカーソルをピクセル単位で動かす操作と対応付けるよう，設計を行った．

グラフィクスカーソルの移動方向と，それに対応する操作の設計を図 4.3 に示す．

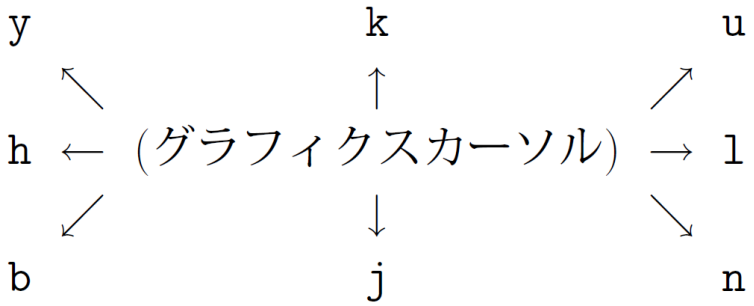

図 4.3: Rogue の操作と対応付ける，グラフィクスカーソルの移動操作の方法

粒度：中　クリッカブルな Web コンテンツの表示位置を考慮した移動

テキストの範囲選択操作を考えなければ，ユーザがクリック操作を行うのは，クリッカブルな Web コンテンツのみに限定される．そこで，このクリッカブルな Web コンテンツのみを，グラフィクスカーソルの選択の対象とするような移動操作を設計する．

Rogue には，主人公がアイテムのすぐそばを通るか，あるいは壁や敵やアイテムに衝突するまで一方向に進み続けるという操作がある．そこでアイテムをクリッカブルな Web コンテンツと対応付ける．つまり，仮想カーソルがクリッカブルな Web コンテンツのすぐそばを通るとき，その Web コンテンツの表示領域上に止まる操作を設計した．また図 4.3 中に示した方向を示すキーと Ctrl キーとの組み合わせによって，この操作がなされるよう，設計した．

粒度：大　Web ブラウザのウィンドウの端への移動

グラフィクスカーソルの現在位置と，選択したい Web コンテンツとが非常に離れている場合がある．この場合，クリッカブルな Web コンテンツを順に巡るだけでは選択に手間がかかる．

Rogue には，主人公が何かに衝突するまで一方向に進み続けるという操作がある．ここで衝突先を Web ブラウザのウィンドウの端と対応付ける．つまり，グラ

フィクスカーソルがWebブラウザのウィンドウの端まで移動する操作を設計した．また図4.3中に示した方向を示すキーとShiftキーとの組み合わせによって，この操作がなされるよう，設計した．

4.4.2 右手のみによる，グラフィクスカーソルの移動

右手側にCtrlキーやShiftキーを備えていないキーボードが存在する．また，人差し指，中指，薬指はグラフィクスカーソルの移動操作を担うことを考慮する．そこで小指の利用によって粒度の異なるグラフィクスカーソルの移動操作を可能とするよう，設計した．

右手のみによって行うグラフィクスカーソルの移動の粒度は，それぞれを状態として設計した．小指はそれらを切り替えるために用いる．

ユーザが利用しているキーボードの配列によって，右手の小指付近にあるキーは異なる．ユーザが日本語配列のキーボードを利用している場合を例に挙げ，その際の小指の操作と粒度の変化を状態遷移図として図4.4に示す．日本語配列のキーボードを利用している場合，’;’キーや’:’キーが押された時，以降行われる移動操作は，それぞれCtrlキー，Shiftキーが同時に入力されたものと見なす．つまり’;’キーや’:’キーは，グラフィクスカーソルの移動操作の粒度に対し，ある状態を定義する．

方向キーがCtrlキーやShiftキーと同時に入力されることもある．その際，状態として定義されている粒度を一時的に無視し，修飾キーによって指定される移動操作の粒度を優先させるよう，設計した．またこの後の移動操作は，状態として定義されている粒度にしたがう．

4.4.3 選択される可能性のあるWebコンテンツの絞り込み

クリッカブルなWebコンテンツが多数含まれるWebページを閲覧している時，ユーザが選択したいWebコンテンツが含まれる範囲を徐々に絞り込むことによっ

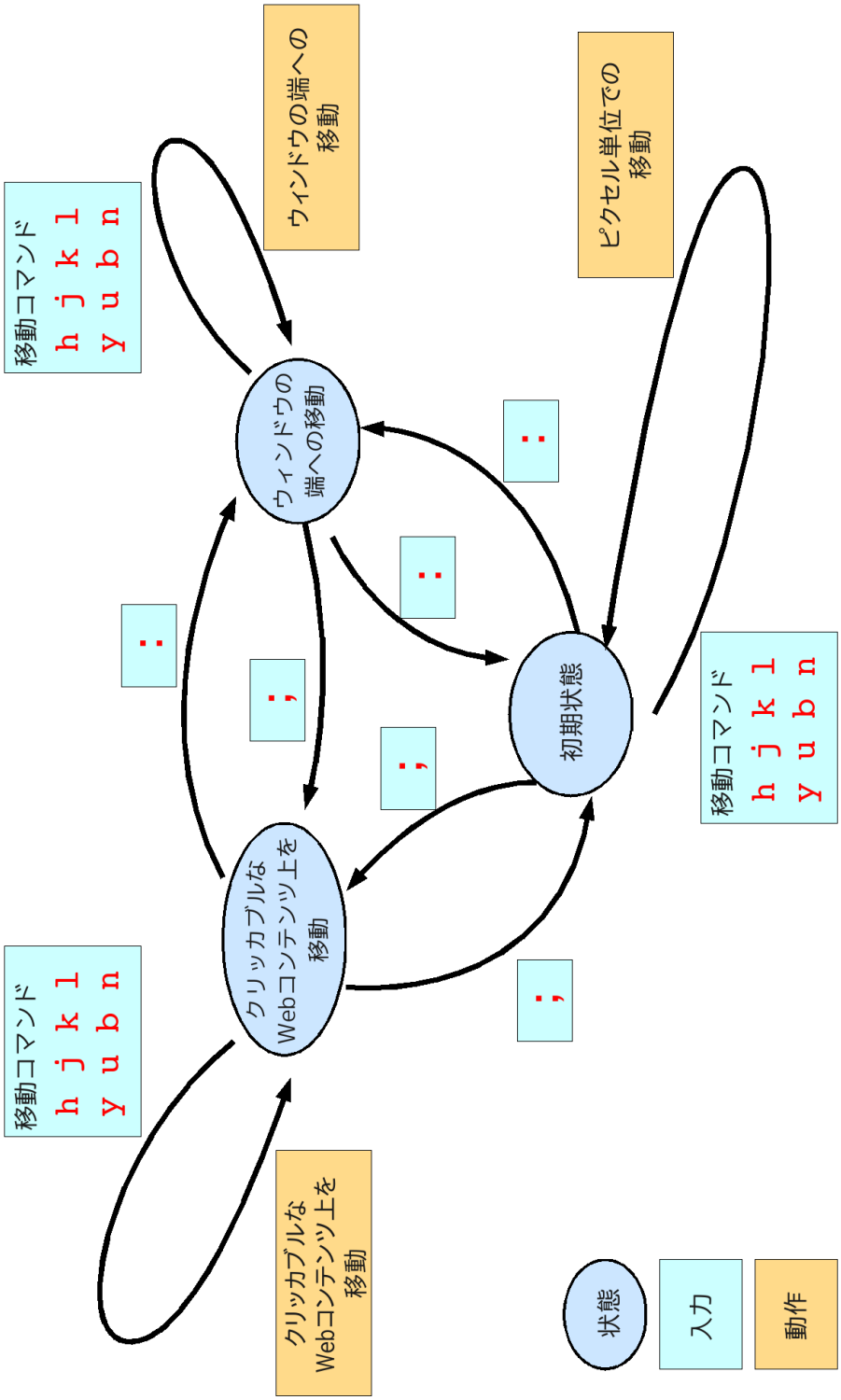

図 4.4: グラフィクスカーソルの移動操作における状態遷移図

て，効率的に選択が行える可能性がある．ここで OS がマウスキー機能を提供していることがある．マウスキー機能とは，キーボードによって OS が提供するグラフィクスカーソルを操作するための機能である．テンキーでマウスキー機能を扱う手法に関する先行研究においては，テンキーの物理的配置を画面上にマッピングし，入力されたキーに対応する画面領域を再帰的に分割していくことにより，所望のコンテンツを選択するまでに必要となる時間を大幅に短縮できると報告されている [3].

そこでそれと似た手法によって画面領域の再帰的な分割ができるよう，設計を行った．まず画面は縦方向，横方向のそれぞれを三分割した．分割されたそれぞれの領域へは，キーボードの右手側にあるキーの物理的配置をマッピングした．この様子を図 4.5 に示す.

u	i	o
j	k	l
m	,	.

図 4.5: キーボードの右手側にあるキーの物理的配置と，分割された画面領域の対応

また前節と同じ理由により，右手のみによってその操作を可能にする．グラフィクスカーソルの移動操作に用いない小指によって，分割操作を行う状態を表現するよう，設計を行った．加えて，分割された領域を再帰的に選択する際，選択された領域内に表示されているクリッカブルな Web コンテンツの数によって，その処

理を分けるよう設計を行った．

領域内にWebコンテンツがなければ，選択された領域をリセットする．領域内にただ一つのWebコンテンツがあれば，それを選択する．領域内に二つ以上のWebコンテンツがある場合のみ，再帰的な領域分割を行うよう，設計した．つまり，Webブラウザのウィンドウが領域に分割されるとき，分割された領域を適切に選び続けるだけで，Webコンテンツの選択が行えるよう，設計した．この操作とグラフィクスカーソルの状態の変化を状態遷移図として図4.6に示す．

4.5 仮想カーソルの設計

4.4節では，ユーザはWebコンテンツの表示領域に応じてWebコンテンツの選択操作を行っていると述べた．また，それに適応すべく，グラフィクスカーソルの移動操作における粒度の設計を行った．しかしマウスキーは，ピクセル単位でしかグラフィクスカーソルの操作が行えない．つまりマウスキーはグラフィクスカーソルの移動操作に必要となる，粒度の違いを扱うことができない．

グラフィクスカーソルの移動操作の粒度の違いを扱えれば，Webコンテンツの選択を効率的に行える可能性がある．そこで，Webコンテンツの選択専用のグラフィクスカーソルを設計する．また，これはOSが提供するものとは異なるため，“仮想カーソル” と呼ぶことにする．

通常OSが提供するグラフィクスカーソルを用いて行うWebコンテンツの選択操作は，仮想カーソルが全て担う．そのためWebページの閲覧においては，OSが提供するものと似た振る舞いを持つよう，仮想カーソルを設計する．以下，仮想カーソルに必要となる基本的な振る舞いを挙げる．

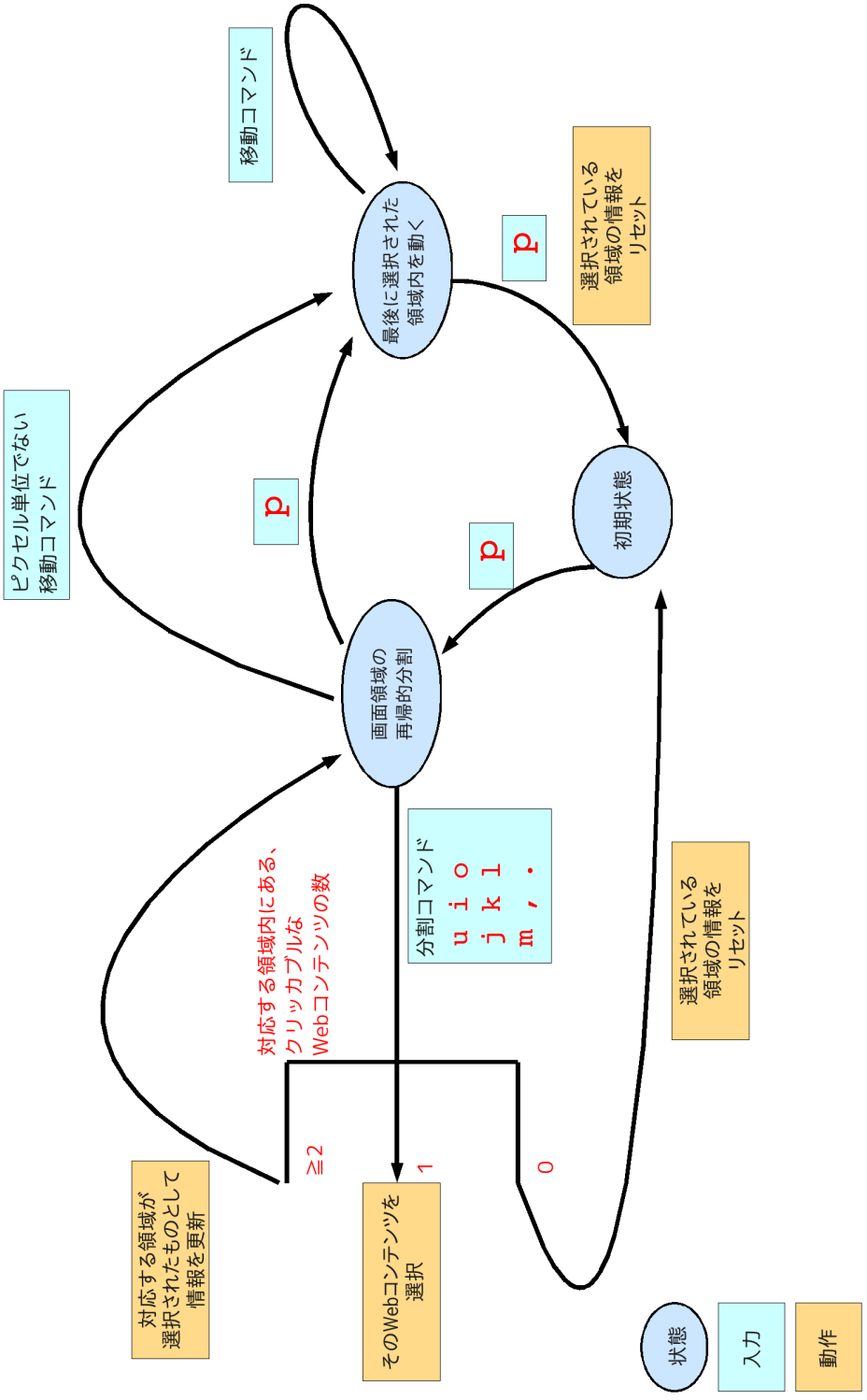

図 4.6: 領域の再帰的な分割における状態遷移図

- 閲覧している Web ページの表示に対し，大きな変更を加えない必要がある．
- Web コンテンツよりも手前側にあるように表示される必要がある．
- 表示座標が変更される時，ユーザに対して視覚的なフィードバックを与える必要がある．
- 閲覧している Web ページがスクロールされても，そのウィンドウ内の表示座標は変更されない必要がある．
- 表示座標に対するクリック操作が可能である必要がある．

第 5 章

UIの実装

本研究は，Firefoxの拡張機能としてUIを実装した．UIは約5500行のJavaScriptとXULによって構成されている．またUIは，キー入力に対する操作の割り当てが記述されたコマンドディスパッチテーブルと，ユーザからのキー入力を受け取り，行うべき操作を決定するディスパッチャによって構成されている．

本章ではまず，Firefoxの拡張機能ならびに，各処理を実行するプログラム群について述べる．拡張機能の分類を述べることにより，UIの実装形式を明確にする．次に拡張機能の構成要素について述べ，UIを実装する上で重要であるDOMについて述べる．また，UIを構成するコマンドディスパッチテーブルとディスパッチャの実装について述べ，仮想カーソルの実装について述べる．

5.1 Firefoxの拡張機能

拡張機能をFirefoxに導入することにより，ユーザは新たな操作性や機能性を得ることができる．またFirefoxは，複数の拡張機能を同時に利用することができる．

UIはFirefoxの“拡張”として実装した．Firefoxへ UIを導入することにより，キーボードのみによるWebブラウザ操作を効率的に行うことができる．

本節ではFirefoxにおけるUIの位置付けを明確に行った後，拡張機能の構成要素について述べる．またUIにとって重要である，DOMについて述べる．

5.1.1 拡張機能の分類

Firefox の拡張機能は以下のように大きく二つに分類できる．

1. プラグイン（Plug-in）

2. 拡張（Extension）

上記 1 のプラグインは，Web ページに含まれる全ての情報を表示するために用いられる拡張機能である．ここで言う全ての情報とは，全ての Web コンテンツを意味する．たとえば Windows Media Player や Apple Quick Time のプラグインを利用すれば，Firefox によって音楽コンテンツや動画コンテンツの再生が可能になる．

上記 2 の拡張もまた，Firefox に新たな機能を追加するものである．しかし，拡張は新たな Web コンテンツの表示ではなく，新たな操作性や機能性の追加のために用いられるものである．

本 UI は，拡張機能の中の “拡張” を利用した．なお以降 “拡張機能” と “拡張” とを異なるものとして述べる．

5.1.2 Firefox の拡張機能の構成要素

Firefox の拡張機能には多くの技術が含まれる．それらを以下に示す．

- CSS（Cascading Style Sheet）
- XUL（XML User Interface Language）
- JavaScript
- XPCOM（Cross Platform Component Object Model）

拡張機能における構成要素の役割を，それぞれ以下に簡潔に述べる．

CSS

CSS[8] は W3C によって開発された仕様である．これを用いることにより，HTML 文書や XML 文書に含まれる要素に対し，視覚的な装飾を加えられるようになる．

また CSS は，拡張機能の外観を定義するものである．Firefox の拡張機能の構成要素には XML 文書が含まれる．この XML 文書の内容の視覚的装飾に CSS が用いられる．

XUL

XUL[9] は Mozilla が開発した言語である．拡張機能において，XUL は Firefox の GUI を記述する．Firefox の GUI とは，階層的なメニューやウィンドウやダイアログボックス等を指す．

XUL を用いることにより，Firefox に対する新たなショートカットコマンドの追加も可能である．しかし Firefox が提供するショートカットコマンドが，XUL を用いて追加したものと衝突した場合には，前者が優先される．

JavaScript

JavaScript は数値，文字列，論理値を基本データ型として持つインタプリタ型の言語である．また JavaScript は，Web ブラウザ上で動作することも多い．拡張機能においては，JavaScript はその大まかな動作を決定するために用いられる．

Firefox が備える JavaScript エンジンは，DOM に準拠している．そのため拡張機能を構成する JavaScript は，DOM の API を利用することができる．DOM を用いることにより，HTML 文書の構造の動的な変更が可能になる．DOM については 5.1.3 節に後述する．

XPCOM

XPCOM は OS に依存しないコンポーネントの集合である．拡張機能において，XPCOM は JavaScript では実現できないような高度な操作を行う．たとえばファイルの操作やメモリの管理等の操作は，XPCOM の利用によって可能となる．

Firefox は多数の XPCOM コンポーネントを含んでいる．たとえば閲覧履歴に関する操作やタブに関する操作，クリップボードの操作等は，XPCOM コンポーネントによって実現されている．XPConnect（Cross Platform Connect）を用いることにより，拡張機能に含まれる JavaScript で XPCOM コンポーネントを扱うことができる．

5.1.3 DOM

DOM（Document Object Model）[10] とは，W3C（World Wide Web Consortium）によって勧告がなされている，XML 文書や HTML 文書を扱うための API（Application Programming Interface）である．また，Firefox が備える JavaScript エンジンは DOM に準拠している．そのため，拡張機能を構成する JavaScript は DOM の API を利用することができる．

UI を実装する上で，DOM は欠かすことができない重要な役割を担っている．そこで本節では DOM について述べる．

DOM によるツリー構造化

DOM は HTML 文書や XML 文書を対象とし，その文書の論理的構造を階層的なツリー構造として表現する．入れ子構造を持つ要素には親子関係を設け，並列構造である要素には，共通の親の下にある並列な子という関係を設ける．この様子を HTML 文書を例にしながら以下に述べる．

図 5.1 の左側に示した HTML 文書は，DOM によって図 5.1 の右側のようなツリー構造化が行われる．ここで図 5.1 の左側にある HTML 文書の，<HEAD>から</HEAD>

までの部分と<BODY>から</BODY>までの部分とは，<HTML>から</HTML>までの部分のすぐ内側にある．このときDOMはこのHTML文書を，<HTML>要素が<HEAD>要素と<BODY>要素を子として持つものとして表現する．同様にして，HTML文書内に現れる全ての要素は，いずれかの要素と親子関係を持つものとして表現される．このように表現されたツリー構造をDOMツリーと呼ぶ．

一般に<HTML>要素の親要素はDocumentオブジェクトではない．しかしDocumentオブジェクトは，その子要素の一部として<HTML>要素を含む．そのため，便宜上Documentオブジェクトをルートとして，図5.1にDOMツリーを示した．また以降も便宜上，同じ方法で示すことにする．

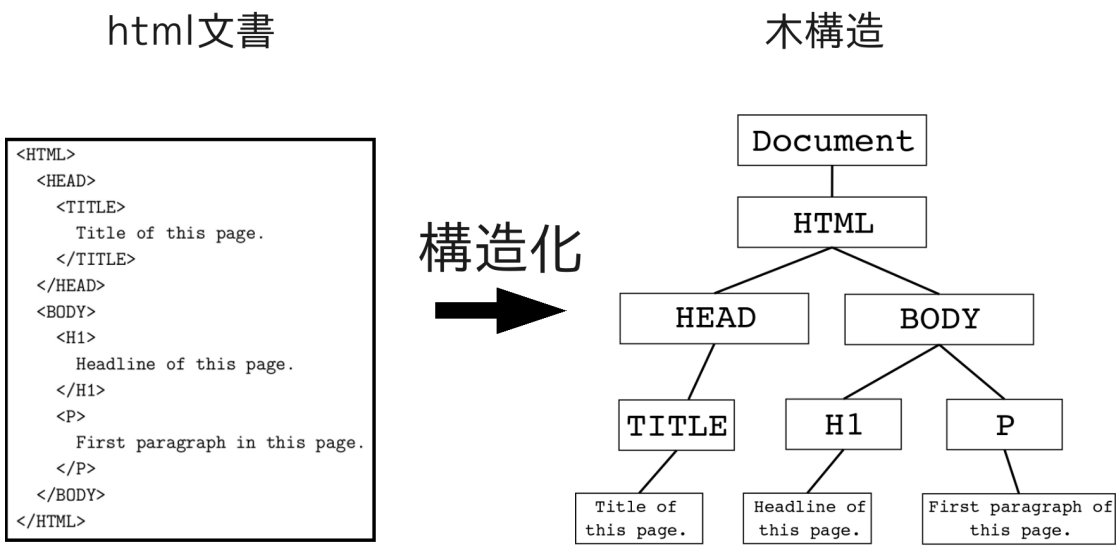

図 5.1: DOMによる，HTML文書のツリー構造化

DOMによる，文書構造の変更

HTML文書やXML文書より得られるDOMツリーには，その文書内に現れる全ての要素が含まれる．DOMツリーのルートから，その子要素を順にたどることによって，DOMツリーに含まれる全ての要素を参照することができる．

DOMツリーに含まれる要素が一意のIDを持つ場合がある．その場合，DOMは

その ID を持つ要素だけを抽出することができる．また，HTML 文書における要素名（タグ名）を元にし，その要素名を持つ要素だけを抽出し，参照することもできる．この方法を用いれば，ある要素を参照する時に DOM ツリーのルートから子要素を順にたどる必要はない．

また参照する要素に対しては，子要素の追加，その要素の削除，その要素の内容の変更等の操作が可能である．たとえば HTML 文書においては<BODY>要素の内部に記述されている要素が，Web コンテンツとして Web ブラウザ上に表示されることが多い．DOM を用いてその HTML 文書内の<BODY>要素に新たな子要素を追加すれば，元々の HTML 文書には存在しえない新たな Web コンテンツを Web ブラウザ上に表示することも可能である．

DOM によるイベント処理

Web ブラウザに対してユーザがキー入力を行う時，必ずキーイベントが発行される．キーイベントは，押下されたキー情報を含む．このキーイベントは DOM によって定義されている．

キーイベントを含む全てのイベントは，イベントリスナの利用によってキャプチャが可能である．イベントリスナには，キャプチャするイベントと，そのキャプチャ時に実行する処理の記述が可能である．

一度発行されたイベントは，DOM ツリー内を伝播する．この伝播には二つのフェーズがある．またイベントリスナが処理を行うフェーズは，そのうちのいずれか一方のみである．

イベントの伝播の流れを以下に述べる．

0. イベントの発生

1. キャプチャ・フェーズ（capture phase）

   DOM ツリーのルートは，イベントの発生元である要素を特定できない．

そこで DOM ツリーのルートから子要素に向かって，イベントの発生元を探索する．イベントの発生元を特定した後，一度 DOM ツリーのルートから発生元に向かって要素を辿る．その時，処理を行うべきイベントリスナを持つ要素があれば，そのイベントリスナの処理を行う．

2. バブリング・フェーズ（bubbling phase）

   キャプチャ・フェーズで処理されたイベントは，発生元から親要素へ向かって順に伝播する．ここでも，処理を行うべきイベントリスナを持つ要素があれば，そのイベントリスナの処理を行う．

イベントの伝播の流れを，フォームに対する入力操作を例に挙げながら示す．

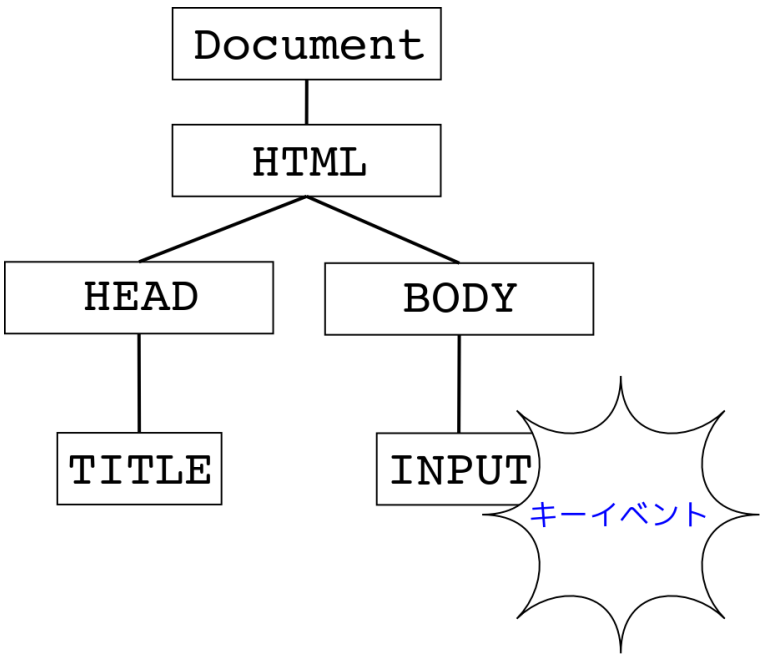

図 5.2: フォームを対象とするイベントの発生

フォームは HTML 文書において<INPUT>要素として記述される．そのためフォームに対するキー入力は，<INPUT>要素を対象とするキーイベントである（図 5.2）．

DOM ツリーのルートを始点とし，イベントの発生元に至るまで子要素をたどる．この時，ある要素がキャプチャ・フェーズで処理されるべきイベントリスナを持つ

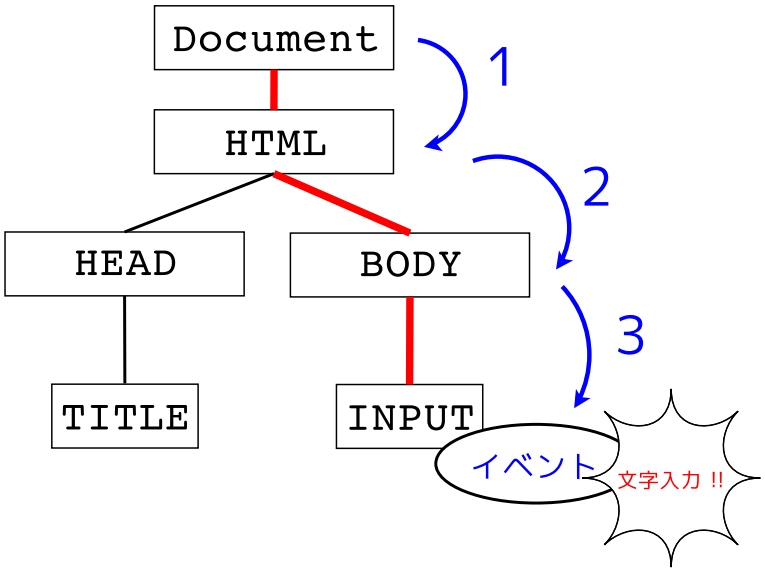

図 5.3: キャプチャ・フェーズにおける，ルートから子要素方向へのイベントリスナ探索

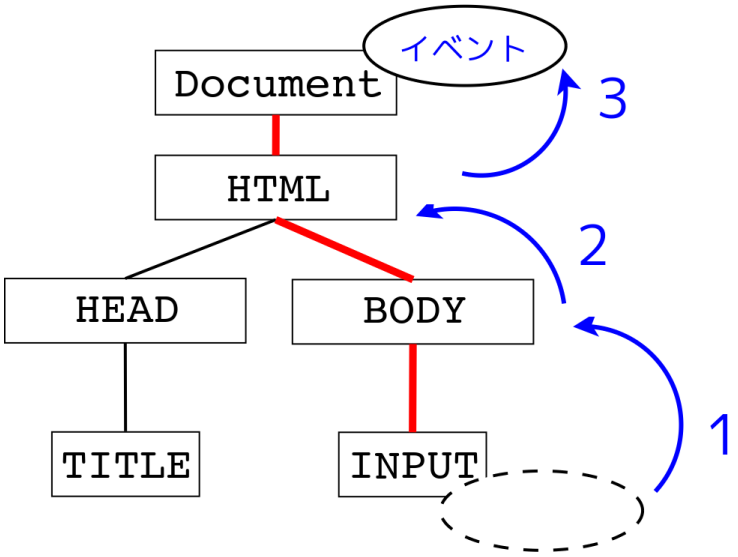

図 5.4: バブリング・フェーズにおける，子要素からルート方向へのイベントリスナの探索

ならば，その要素に注目するのと同時にそのイベントリスナの処理を行う．DOMツリーのルートから<INPUT>要素に至る経路上に，キャプチャ・フェーズで処理されるイベントリスナを持つ要素が存在しない場合，<INPUT>要素に対してキーイベントが持つ，デフォルトの操作が実行される．この例におけるデフォルトの操作は文字入力である（図5.3）．

イベントの発生元である要素での処理を終えると，イベントは，その親要素を順に伝播していく．この時，ある要素がバブリング・フェーズで処理されるべきイベントリスナを持つならば，その要素に注目するのと同時にそのイベントリスナの処理を行う．DOMツリーのルートに至るまでに，バブリング・フェーズで処理されるイベントリスナを持つ要素が存在しない場合，伝播はDOMツリーのルートまで続く（図5.4）．またDOMツリーのルートまでイベントが伝播すると，イベントは消滅する．

5.2 UIの構成

FirefoxのUIを実装した．UIは複数のコマンドディスパッチテーブルと，ディスパッチャによって構成される．コマンドディスパッチテーブルには，特定のキー入力に対するWebブラウザ操作の割り当てが記述されている．ディスパッチャはユーザからのキー入力を元にし，コマンドディスパッチテーブルの探索を行う．キー入力にある操作が割り当てられていた場合，その操作を実行する．

Firefoxに対して行われる操作の種別ごとにモードを設定した．Webブラウザの全ての操作は，いずれかのモードに必ず属している．また操作の種別の変更を，モードの切り替えと対応付けて実装した．

エディタの操作はキーボードと深い関係を持つ．そのため，エディタの操作と似た方法によってWebブラウザの操作が行えれば，その操作効率の向上が見込める．そこで本研究は，vimと似た方法によってWebブラウザの操作を可能にするよう，

UIを実装した．vimの操作には複数のモードの切り替え操作が含まれる．Webブラウザ操作に対するモードを切り替える必要があるとしても，操作方法をvimに似たものとして統一すれば不自然ではない．

UIは，フォームに対して行う文字入力の操作を，vimの操作方法と似たものとして提供する．つまり，フォームがフォーカスされている時であっても，文字の入力がなされる前にUIの処理を行う必要がある．そこで，ユーザによるキー入力が行われるときに発行される`keypress`イベントのキャプチャを行った．UIはDOMツリーのルートに対してイベントリスナを追加し，その`keypress`イベントをキャプチャする．またイベントのキャプチャを，キャプチャ・フェーズにおいて行った．

UIにはWebコンテンツ選択専用の仮想カーソルを実装した．このカーソルはOSが提供するものとは異なる．また，クリッカブルなWebコンテンツの表示領域を考慮した操作が行えるよう，仮想カーソルを実装した．

これらの詳細を次節以降に述べる．

5.2.1 コマンドディスパッチテーブルの実装

コマンドディスパッチテーブルには，ユーザからのキー入力に対する操作の割り当てを記述する．UIは複数のコマンドディスパッチテーブルを持ち，ユーザからの入力に応じてそれらの切り替えを行う．ここでは，ある一つのコマンドディスパッチテーブルを例に挙げ，その実装について述べる．

コマンドディスパッチテーブルは，JavaScriptのオブジェクトとして実装した．UIへの入力コマンドは，そのオブジェクトのプロパティとした．また入力コマンドに割り当てる操作は，そのプロパティの値とした．

以降の説明のため，数種類の語句を以下のように定義する．

- キー入力
  - ユーザがキーボード操作によって行うものである．
- キーイベント
  - ユーザによるキー入力を，DOMツリーから見たときの表現である．
- 入力コマンド
  - DOMツリー内を伝播するキーイベントを，UIから見たときの表現である．

ユーザによるキー入力を，UIは入力コマンドとして捉える．コマンドディスパッチテーブルは，入力コマンドに対する操作の割り当てを保持している．

入力コマンドに割り当てる全ての操作は，関数として実装した．コマンドディスパッチテーブルを表すオブジェクトが持つプロパティの値は，その関数名を文字列化したものである．二つの入力コマンド `a`，`b` に対して，それぞれ**操作** `A`，**操作** `B` を割り当てる時のコマンドディスパッチテーブルの例を図5.5に示す．

5.2.2 操作モードとコマンドディスパッチテーブルの対応

Webブラウザの操作には種別があり，その種別ごとに求められる操作は大きく異なる．たとえば文字入力を行っている時，グラフィクスカーソルの操作は必要ない．そこで，それらの種別ごとにモードを設定した．

モードには，それぞれに対応するコマンドディスパッチテーブルを実装した．モードの切り替え操作には，コマンドディスパッチテーブルの切り替えを対応付ける．またディスパッチャは，UIのモードに対応するコマンドディスパッチテーブル内

```
1   CommandDispatchTable = {
2     a : 'Func_A()'  ,
3     b : 'Func_B()'  ,
4   }
5
6   function Func_A(){
7     操作 A;
8     return;
9   }
10
11  function Func_B(){
12    操作 B;
13    return;
14  }
```

図 5.5: コマンドディスパッチテーブルの実装

に，入力コマンドに対する操作の割り当てが記述されているかどうか探索を行う．この様子を図 5.6 に示す．

Web コンテンツを対象とする操作には，それに通常用いられる入力デバイスに対応する二つのモードを実装した．これは，通常キーボードを用いて行う操作のためのモードと，通常マウスを用いて行う操作のためのモードである．また Web ブラウザを対象とする操作には一つのモードのみを実装した．

Web ブラウザ操作の種別ごとに実装したモードを図 5.7 に示す．また実装したそれぞれのモードについて，次節以降に述べる．

文字入力モード

文字入力モードは，通常キーボードを用いて行う操作のためのモードである．Web ブラウザを操作するとき，キーボードは主にフォームへの文字入力を行うために用いられる．フォームへの文字入力が行われるとき，必ずフォームがフォーカ

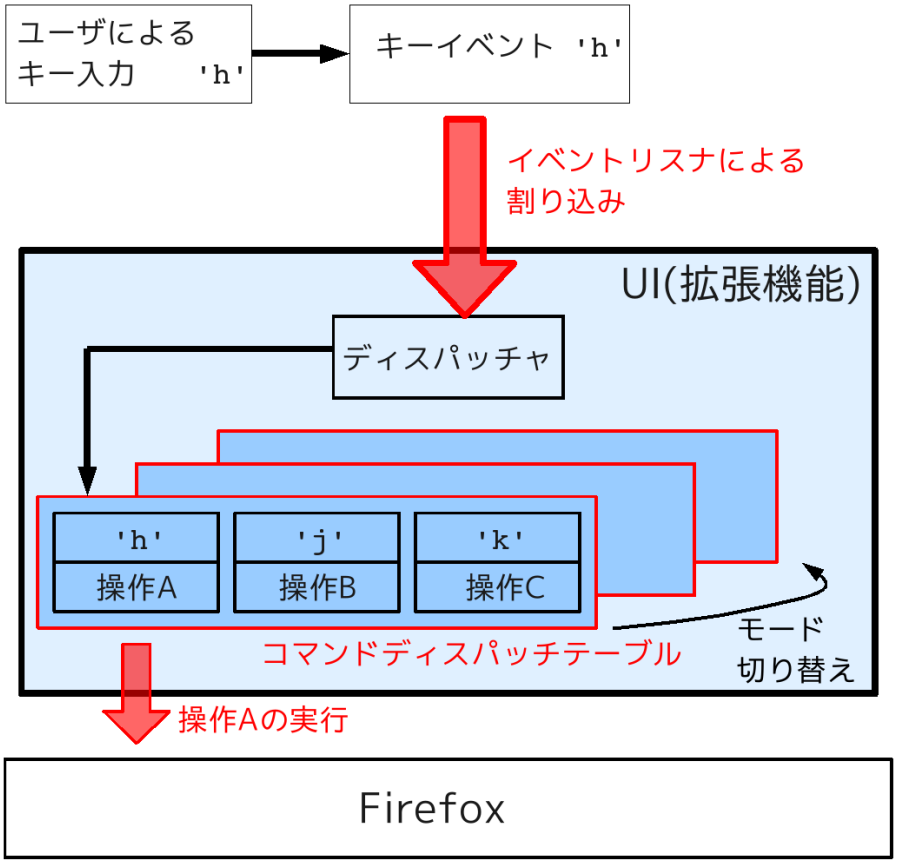

図 5.6: 入力コマンドに割り当てられた操作の実行

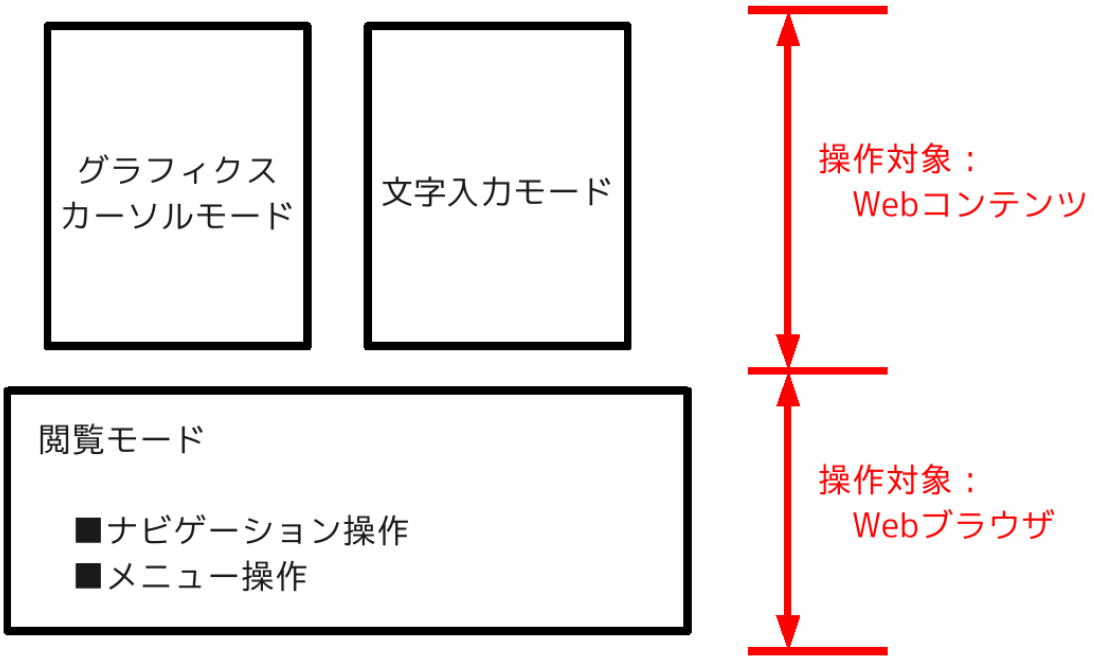

図 5.7: 種別分けされた，Web ブラウザの操作に対応するモード

スされている．つまりフォームがフォーカスされている時，UI は文字入力モードで動作する．

文字入力モードには，フォームを対象とする文字入力の操作と文書編集の操作を可能とするよう，vim に必要となる最低限の機能を実装した．それらの機能のため，文字入力モードは複数のモードの集合として実装した．

vim と同じように，文字入力モードには挿入モードと編集モードという二つのモードを実装した．さらに編集モードには，削除モードと変更モードという，補助的なモードを実装した．vim と同じく，ある特定のコマンドの入力によって，これらのモードの切り替えが可能である．なお，これらは UI の文字入力モードにおける操作であるため，全て同一のコマンドディスパッチテーブルに記述した．UI は，文字入力モードにおける，その内部のモードの違いを状態の違いとして扱う．

文字入力モードとして実装した操作を，状態遷移図として図 5.8 に示す．

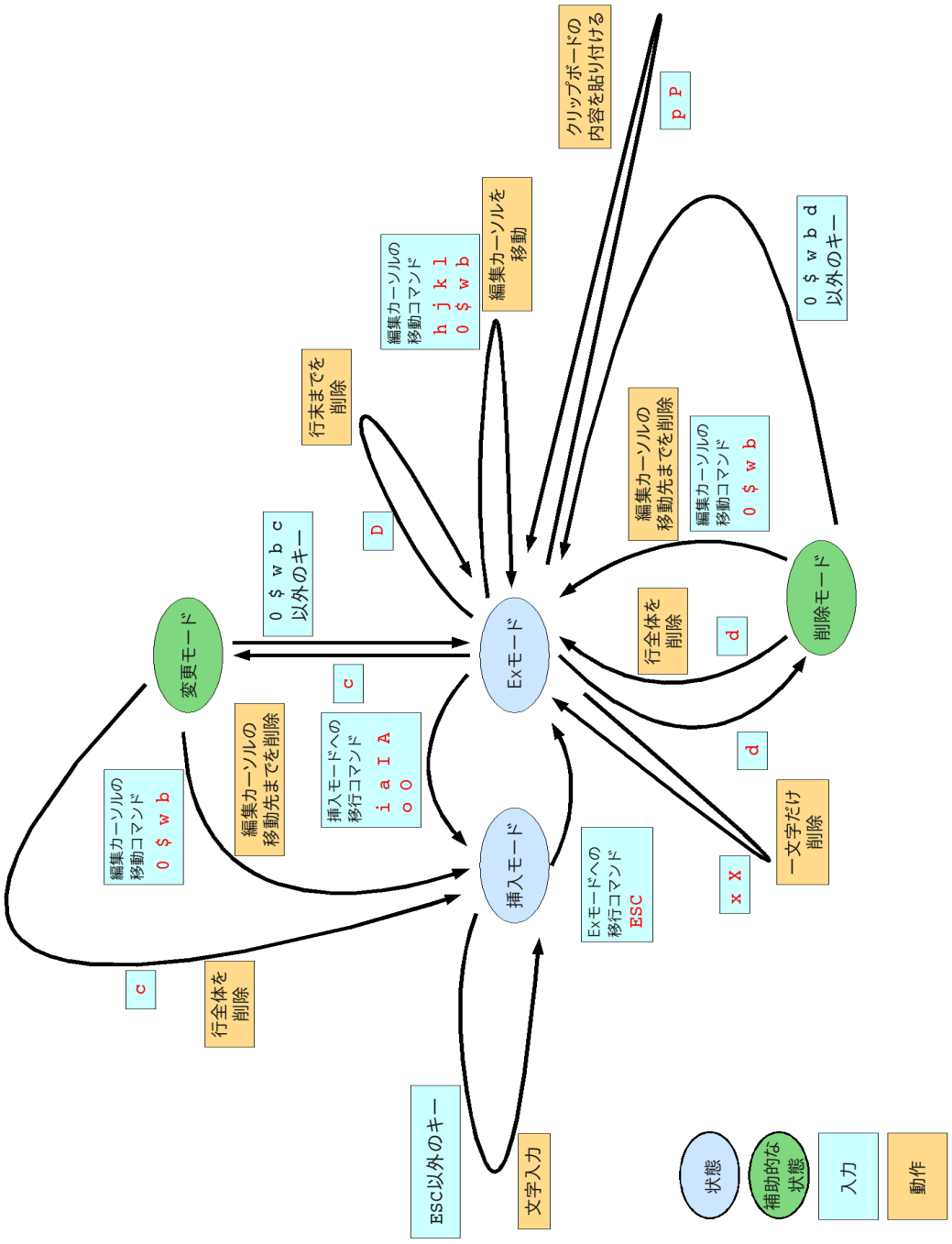

図 5.8: 文字入力モードの操作と，その状態遷移図

グラフィクスカーソルモード

グラフィクスカーソルモードは，通常マウスを用いて行う操作のためのモードである．Webブラウザを操作するとき，マウスは主にWebコンテンツの選択を行うために用いられる．

Webコンテンツの選択のため，UIには仮想カーソルを実装した．仮想カーソルは表示/非表示という二つの状態を持つ．このうち仮想カーソルがFirefoxのウィンドウ内に表示されている時，UIはグラフィクスカーソルモードで動作する．

グラフィクスカーソルモードには，仮想カーソルの操作を実装した．この操作には，仮想カーソルの表示座標の変更操作や，その表示座標へのクリック操作などが挙げられる．またFirefoxのウィンドウ内に表示されている，クリッカブルなWebコンテンツの表示領域を考慮した選択操作も可能である．

閲覧モード

閲覧モードは4.3.2節で述べた，ナビゲーション操作とメニュー操作を行うためのモードである．フォーカスされているWebコンテンツがなく，かつ仮想カーソルが表示されていない時，UIは閲覧モードで動作する．

閲覧モードに求められるナビゲーション操作とメニュー操作の実装を，以下に述べる．

ナビゲーション操作　ナビゲーション操作は，XPCOMやDOMツリーにおける要素のID等を利用することで実装した．以下，実装の手法別にそれらを簡潔に述べる．

- XPCOMを用いて実装した操作
  - Firefoxがその内部で宣言しているグローバル変数が，操作に必要なXPCOMの機能を有していた．また，XPCOMを利用する上でショートカッ

トとなる関数をFirefoxが定義していることがあった．そこで，それらを用いて操作の実装を行った．以下に挙げる操作の実装を，図5.9に示す．

  - 閲覧履歴を “戻る”，“進む” 操作
  - タブの操作
  - 現在表示されているWebページのスクロール操作

- DOMツリーにおける，要素のIDを用いて実装した操作
  - 操作の対象となる要素が，DOMツリー内で一意のIDを持っていた．そこで，そのIDを持つ要素を参照し，操作の実装を行った．以下に挙げる操作の実装を，図5.10に示す．
    - アドレスバーのフォーカス操作
    - 検索バーのフォーカス操作
- DOMツリーにおける，要素のIDを用いずに実装した操作
  - 以下に挙げる操作の実装を，図5.11に示す．
    - Webページの再読み込み操作

メニュー操作　通常メニュー操作はマウスを用い，OSが提供するグラフィクスカーソルを操作することによって行う．UIはその代わりに仮想カーソルを用いる．これにより，Webコンテンツを選択することができる．仮想カーソルの実装の詳細は次節に述べるが，これはFirefoxが読み込むHTML文書の<BODY>要素に追加されるものである．すなわち仮想カーソルは，Firefoxのクライアント領域に表示されるWebコンテンツである．そのため，仮想カーソルではメニュー操作が行えない．

そこでメニュー操作は，Altキーと英字キーの組み合わせによって操作できるよう，実装した．メニューを階層的にたどるためには矢印キーを用い，フォーカスされているメニュー項目の選択のためにはEnterキーを用いるよう，実装した．

```
1  //-----------------------------------------
2  // グローバル変数 gBrowser を用いれば,
3  // 閲覧履歴に基づく操作とタブ操作が可能である.
4  //-----------------------------------------
5
6  // 閲覧履歴を ‘‘戻る’’ 操作
7  function Func_GoBack(){
8    gBrowser.goBack();
9    return;
10 }
11
12 // 現在閲覧しているタブの左側にあるタブを選択する操作
13 function Func_NextTab(){
14   gBrowser.mTabContainer.advanceSelectedTab(-1,true);
15  return;
16 }
17
18 // 現在閲覧しているタブを閉じる操作
19 function Func_CloseTab(){
20   gBrowser.removeCurrentTab();
21   return;
22 }
23
24 //-----------------------------------------
25 // 関数 goDoCommand() は Firefox が定義した,
26 // XPCOM の利用のためのショートカット関数である.
27 //-----------------------------------------
28
29 // 現在閲覧している Web ページを
30 // 一行分だけ下方向にスクロールする操作
31 function Func_ScrollLineDown(){
32   goDoCommand('cmd_scrollLineDown');
33   return;
34 }
```

図 5.9: Firefox のグローバル変数や，ショートカット関数を用いて実装したナビゲーション操作

```
1  //-----------------------------------------
2  // アドレスバー，検索バーは DOM ツリー内で
3  // 'urlbar', 'searchbar' という
4  // 一意の ID をそれぞれ持つ.
5  //-----------------------------------------
6
7  // アドレスバーをフォーカスする操作
8  function Func_FocusAddressBar(){
9    window.document.getElementById('urlbar').focus();
10   return;
11 }
12
13 // 検索バーをフォーカスする操作
14 function Func_FocusSearchBar(){
15   window.document.getElementById('searchbar').focus();
16   return;
17 }
```

図 5.10: DOM ツリーにおける，要素の ID を用いて実装したナビゲーション操作

```
1  // 閲覧している Web ページの再読み込み操作
2  function Func_Reload(){
3    window.location.reload();
4    return;
5  }
```

図 5.11: DOM ツリーにおける，要素の ID を用いずに実装したナビゲーション操作

メニュー操作は常に行われる可能性がある．そこで，メニュー操作のために用いる入力コマンドを，UIのモードによらず常に探索されるような，特別なコマンドディスパッチテーブルに記述した．

5.2.3 仮想カーソルの実装

仮想カーソルはWebコンテンツとして実装した．UIは，現在閲覧しているWebページに対し，画像コンテンツとして仮想カーソルを埋め込む．つまりUIは，Firefoxが読み込むHTML文書に対し，新たな画像要素を含ませるような修正を行う．このとき仮想カーソルは，HTML文書が持つ<BODY>要素の子要素として追加される．

OSが提供するグラフィクスカーソルと仮想カーソルとは互いに異なる．つまり仮想カーソルが表示されている時，ディスプレイには二種類のグラフィクスカーソルが表示される．この様子を図5.12に示す．

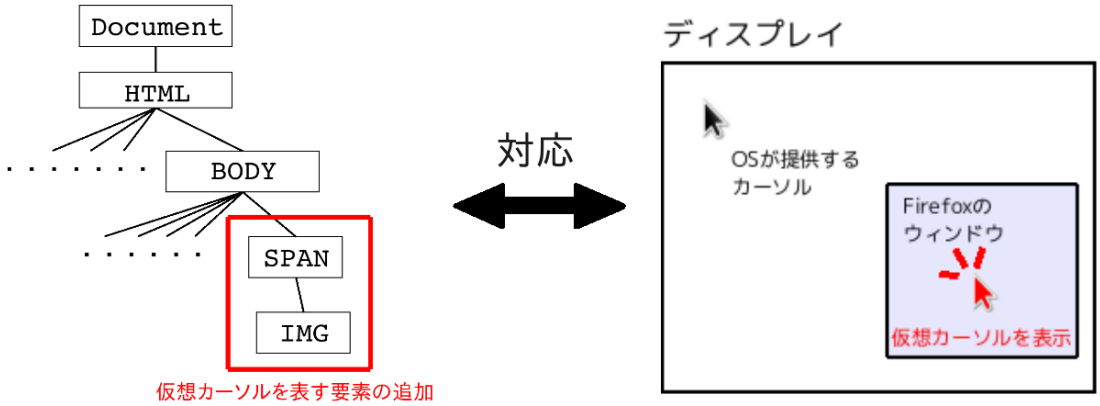

図 5.12: DOMツリーへの要素の追加による，仮想カーソルの実現

仮想カーソルはHTMLの<SPAN>要素として実装した．また，その子要素として<IMG>要素を持つ．<IMG>要素はWebブラウザ上で画像コンテンツして表示される．このため，仮想カーソルはグラフィクスカーソルのように表示される．

Webコンテンツの選択操作は，全て仮想カーソルを用いて行う．通常その選択

操作には，OSが提供するグラフィクスカーソルが用いられる．そのため仮想カーソルは，OSが提供するものと似た振る舞いを持つ必要がある．ここでは4.5節で述べた，仮想カーソルに求められる振る舞いの実装について述べる．

Webページのレイアウトを崩さないための実装

<IMG>要素はHTMLにおけるブロック要素である．ブロック要素がWebブラウザ上で表示される時，その前後に改行が含まれる可能性がある．そのため<IMG>要素として仮想カーソルを実装すると，閲覧しているWebページのレイアウトが崩れる可能性がある．一方<SPAN>要素はインライン要素である．インライン要素がWebブラウザ上で表示される時，その前後に改行が含まれない．そのため<SPAN>要素の子要素として<IMG>要素を持たせても，Webページのレイアウトが崩れることはない．

Webコンテンツよりも手前側に表示するための実装

仮想カーソルを，Webコンテンツよりもディスプレイの手前側にあるように表示するために，仮想カーソルを表す<SPAN>要素のCSSの指定を行った．

CSSにはz-indexというプロパティが存在する．ディスプレイの横方向と縦方向をそれぞれX軸，Y軸とするとき，z-indexプロパティは，ディスプレイを基準とし，ユーザに向かう方向を正とするZ軸上の座標を定義するものである．

仮想カーソルのCSSが持つz-indexプロパティの値を非常に大きな数値とすることにより，仮想カーソルが常にWebコンテンツの手前側に表示されるよう，実装した．この様子を図5.13に示す．

表示座標の変更を可視化するための実装

仮想カーソルを表す<SPAN>要素の表示座標を動的に変更することによって，仮想カーソルの移動を実装した．Webブラウザのウィンドウ内におけるWebコンテ

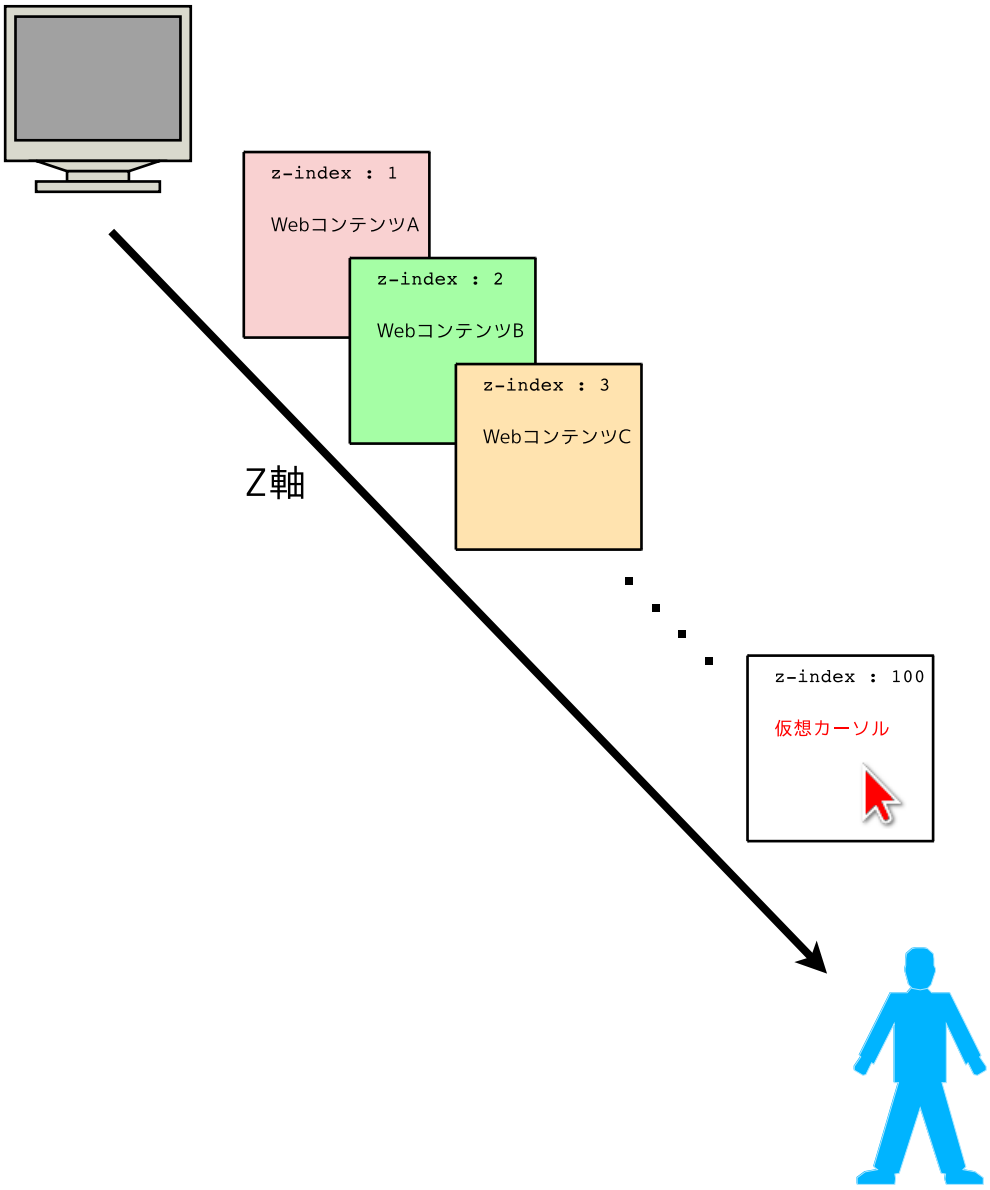

図 5.13: 仮想カーソルに対する Z 軸座標の指定

ンツの表示座標は，その Web コンテンツが持つ CSS の top プロパティと left プロパティによって定義可能である．そこで，仮想カーソルを表す<SPAN>要素の CSS を動的に変更することにより，仮想カーソルの表示座標の変更を実装した．

Web コンテンツが持つ CSS に加える変更は，即座に Web ブラウザ上の表示として反映される．これにより仮想カーソルの表示座標の変更を，視覚的なフィードバックとしてユーザに与えることができる．この様子を図 5.14 に示す．

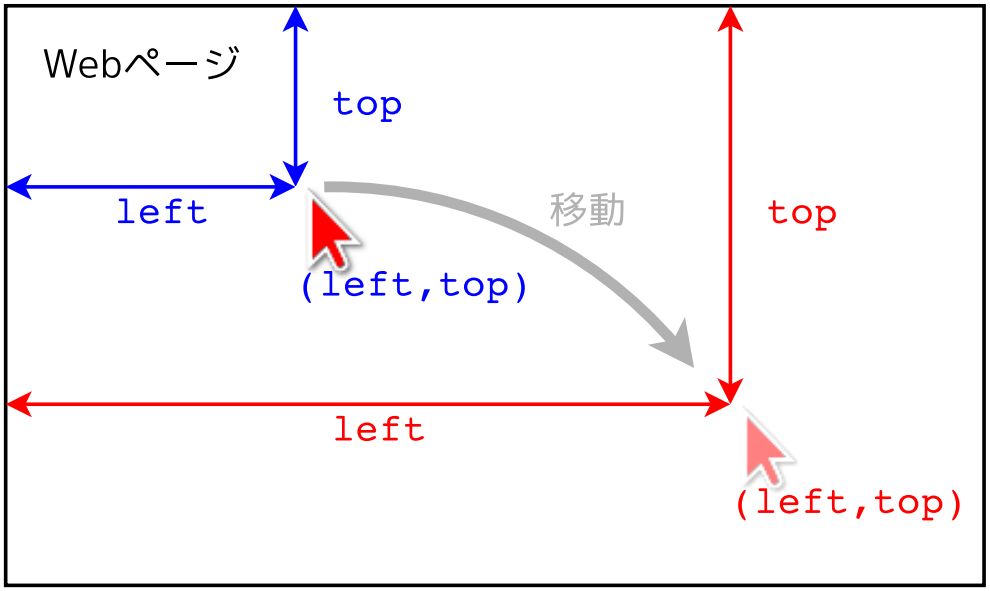

図 5.14: top，left プロパティの変更による，仮想カーソルの移動

Web ページのスクロールに追従するための実装

一般に Web コンテンツは Web ページのスクロールに追従しない．しかし Web ブラウザ上に表示される仮想カーソルは，OS が提供するグラフィクスカーソルと似た振る舞いを持つ必要がある．つまり仮想カーソルは Web コンテンツであるが，Web ページのスクロールに追従する必要がある．

Web ページの表示がその先頭から離れているとき，Web ページはスクロールされている．Web ページがどれだけスクロールされているのかという量は，DOM に

よって取得可能である．Webページのスクロールに追従するような，仮想カーソルの振る舞いの実装にはこのスクロール量を用いた．この振る舞いの実装について，以下に述べる．

0. 一定時間ごとに起動するよう，JavaScriptのタイマを設定する．
1. タイマは，その起動のたびに，Webブラウザが現在表示しているWebページのスクロール量を調べる．
2. 直近の起動時に得られたスクロール量と，今回の起動時に調べたスクロール量とを比較する．
3. それらが互いに異なる値であった時，その差分だけ仮想カーソルの表示座標を修正する．

クリック操作のための実装

仮想カーソルには，その表示座標に対するクリック操作を実装した．この実装には，Firefoxが提供するXPCOMを利用する．ここで利用するXPCOMは，Firefoxのウィンドウ内の任意の点に対し，マウスイベントの生成と送信の機能を提供するものである．マウスの左ボタンをクリックするイベントを，仮想カーソルの表示座標に対して生成し送信することにより，クリック操作をシミュレートするよう，実装した．

5.2.4 仮想カーソルに対する，Webコンテンツの選択操作の効率化を図るための操作の実装

Webページ内のテキストに対する範囲選択操作を考えなければ，選択される可能性があるWebコンテンツはクリッカブルなものに限定される．そこで仮想カーソ

ルに，クリッカブルなWebコンテンツのみを選択対象とする操作を実装した．これにより，Webコンテンツの効率的な選択が行える．

また仮想カーソルの移動操作が持つ，三種類の粒度を実装した．これによりピクセル単位での細かな座標修正をしたり，クリッカブルなWebコンテンツの表示領域上を順に巡ったり，ウィンドウの端への移動が可能になる．

また，仮想カーソルはOSが提供するグラフィクスカーソルと似た振る舞いを持つ必要がある．そこで，仮想カーソルが長い距離を移動するときは移動の軌跡を表示するよう，実装した．

次にWebコンテンツを，その表示領域によって絞り込む操作を実装した．ウィンドウに対する領域分割を再帰的に行い，選択された領域内に表示されているWebコンテンツだけを選択の対象とすれば，選択効率の向上が見込める．

仮想カーソルの移動操作に持たせる粒度の実装

仮想カーソルの移動操作には三種類の粒度を持たせた．本節ではその粒度が小さい順に実装を述べる．

また仮想カーソルは，OSが提供するグラフィクスカーソルに似た振る舞いを持つ必要がある．そこで仮想カーソルの移動を，Firefoxのウィンドウの表示領域内だけにするよう，実装した．この実装には，ウィンドウの表示領域の大きさに関する情報を用いた．この情報はDOMによって定義されているものである．

粒度：小　ピクセル単位での移動　仮想カーソルの移動に持たせる最小粒度は，ピクセル単位での移動である．ピクセル単位での移動を意味する入力コマンドがあるとき，事前に定めておいた移動幅を用いて移動先の座標を求める．求められた座標への移動は，5.2.3節で述べた，CSSの`top`，`left`プロパティが持つ値の変更によって実装した．

なお，この移動幅の値を設定するにあたり，試行錯誤を繰り返した．移動幅が1

ピクセルでは移動が遅すぎると感じた．移動幅が10ピクセルでは移動が速すぎると感じた．そこで，移動幅の値は5ピクセルとして設定した．

粒度：中　クリッカブルなWebコンテンツのみを対象とする移動　仮想カーソルがこの移動を行うとき，その移動方向にクリッカブルなWebコンテンツの表示領域があるかどうかを調べる必要がある．移動方向にその表示領域がなければ，後述する**粒度：大**の操作と同じく，Firefoxのウィンドウ端へ移動するよう，実装した．また移動方向に表示領域があれば，現在の仮想カーソルの表示座標からの距離が最短となるWebコンテンツの表示領域上に移動するよう，実装した．

この移動操作の実現手順を以下に述べる．

step1 クリッカブルなWebコンテンツの抽出

- XPath（XML Path Language）を用い，閲覧しているWebページに含まれる，クリッカブルなWebコンテンツを抽出する．XPathはHTML文書やXML文書のような，構造化された文書を対象にし，その文書に含まれる特定の一部分を問い合わせることができる．

  DOMは要素名をもとにした抽出ができるのに対し，XPathは要素が持つ属性をもとにした抽出が可能である．ここでは，`onclick`属性および`href`属性を持つ要素を問い合わせることにより，クリッカブルなWebコンテンツの抽出を行う．

step2 クリッカブルなWebコンテンツの表示領域に対する近似

- *step1*で抽出した要素は，表示座標や表示領域の情報を含んでいない．そこで，その要素が対応するWebコンテンツの表示領域を矩形で近似する．近似にはDOMで定義されているメソッドを用いる．このメソッドを用いることにより，Webコンテンツの矩形領域の四隅の点の座標情報

が得られる．また，Firefox のウィンドウ内に表示されている要素のみを対象とする．

step3 仮想カーソルと，クリッカブルな Web コンテンツの表示領域との衝突判定

- 仮想カーソルがある一方向へ進むと考え，その移動の軌跡上にクリッカブルな Web コンテンツの表示領域があるかどうかを調べる．軌跡上に表示領域を持つような Web コンテンツのみを，仮想カーソルの移動先の候補とする．

step4 仮想カーソルの移動先の決定

- *step3* で得られた全ての Web コンテンツについて，仮想カーソルの現在位置からの距離を算出する．その距離が最短となる Web コンテンツの座標を，仮想カーソルの新しい座標として決定する．もし *step3* までに該当する Web コンテンツがなければ，仮想カーソルの移動先をウィンドウの端として決定する．

以下，仮想カーソルが現在の表示座標から画面上方向に向かって，クリッカブルな Web コンテンツだけを選択対象にしながら移動する場合を例に挙げ，この手順を示す．

まず *step1* では XPath を用いる．現在 Firefox が表示している Web ページ内に存在する全てのクリッカブルな Web コンテンツを，XPath によって抽出する．Firefox が図 5.15 に示す Web ページを表示している時，合計五個のリンクが抽出される．

step2 では，*step1* で得られた Web コンテンツのそれぞれの表示領域を，矩形として表示されるものとして，近似する．この近似には，DOM で定義されているメソッド `Element.getBoundingClientRect()` を用いる．これにより *step1* で得られた全ての Web コンテンツに対し，Web ページの先頭の左上の点を基準とする，その Web コンテンツの座標情報を得ることができる．ここで得られる座標情報は以

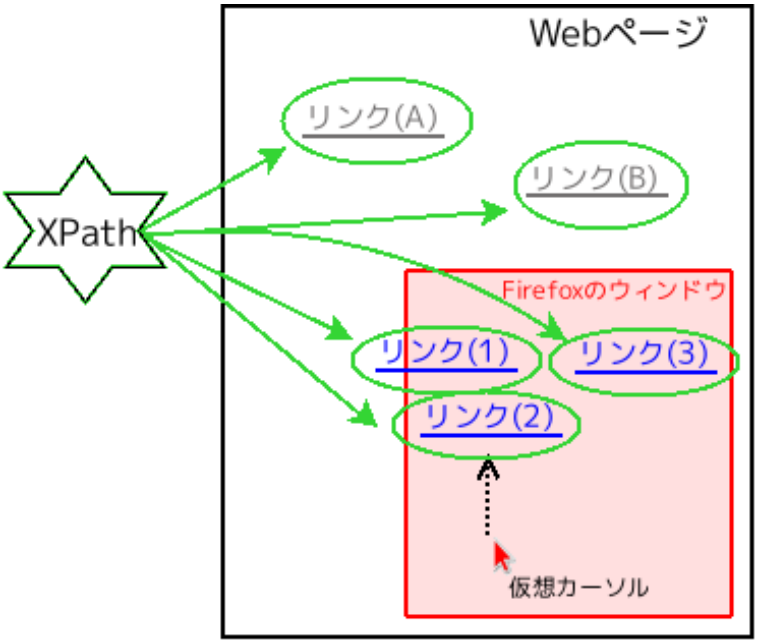

図 5.15: XPath を用いた，クリッカブルな Web コンテンツの抽出

下の四項目である．

- `top`
  - その Web コンテンツの上端の Y 座標
- `bottom`
  - その Web コンテンツの下端の Y 座標
- `left`
  - その Web コンテンツの左端の X 座標
- `right`
  - その Web コンテンツの右端の X 座標

これらの四項目について，図 5.16 に示す．

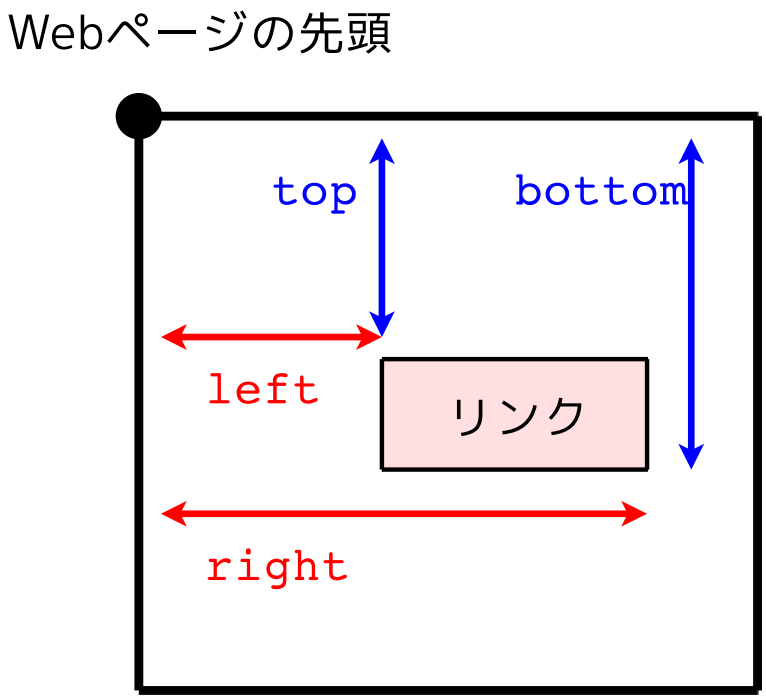

図 5.16: 矩形領域として近似された Web コンテンツが持つ座標情報

得られた座標情報をもとにし，表示領域全体が Firefox のウィンドウ外にあるような Web コンテンツを，仮想カーソルの移動先の候補から除外する．図 5.17 の場合，リンク (A) およびリンク (B) は，その表示領域全体が Firefox のウィンドウの外にあるため，仮想カーソルの移動先の候補から除外される．

step3 では，仮想カーソルの現在位置から上方向に移動するとき，仮想カーソルと表示位置が重なるような Web コンテンツがあるかどうか調べる．この処理のため，まず仮想カーソルの軌跡を直線の方程式として表現する．次に各 Web コンテンツの表示領域をなす四辺のいずれかと，その軌跡とが交点を持つかどうかを判定する．交点を持つならば，仮想カーソルの移動中にその Web コンテンツと表示位置が重なることになる．そこでこの Web コンテンツを，仮想カーソルの移動先の候補の集合に加える．仮想カーソルの軌跡と，ある一辺とが交点を持つかどうか判定を行う様子を図 5.18 に示す．

step4 では，現在の仮想カーソルの表示座標から最も近い位置にある Web コン

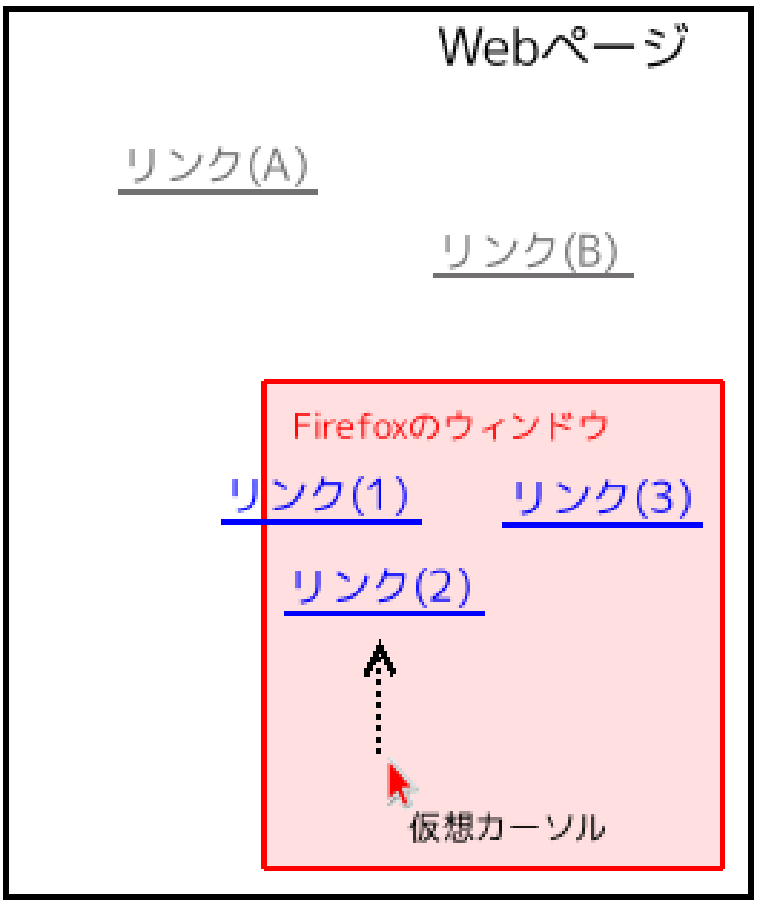

図 5.17: ウィンドウに表示されている Web コンテンツのみを選択対象対象とする，仮想カーソルの移動

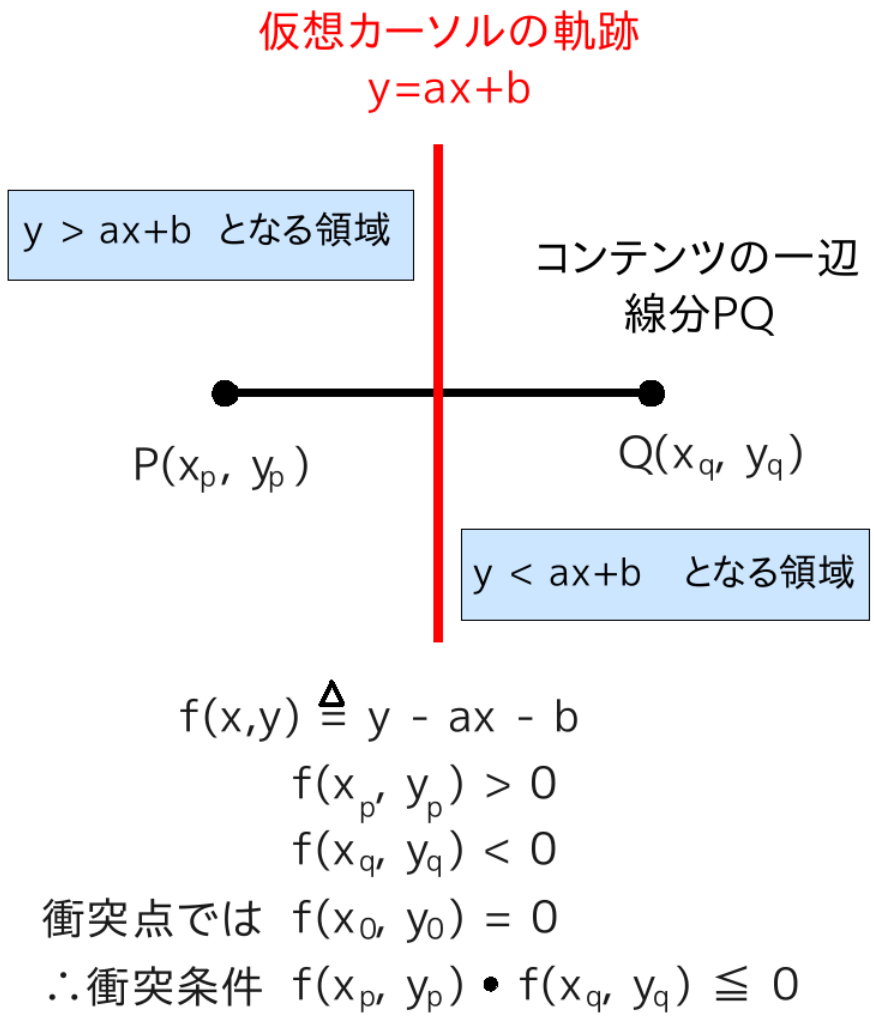

図 5.18: ある辺と仮想カーソルの移動軸との衝突判定

テンツを，*step3* で得た候補の集合の中から探す．図 5.17 の場合，*step3* において，リンク (1) とリンク (2) とが移動先の候補として残るが，現在の仮想カーソルの表示座標により近いリンク (2) を，仮想カーソルの移動先として決定する．

また，クリッカブルな Web コンテンツの表示領域を矩形として近似するだけではいけない．仮想カーソルは，それらの Web コンテンツの表示領域と重なるときだけ，それらを移動先の候補に加える．つまり，Web コンテンツの表示領域と仮想カーソルの移動軌跡とが重ならない場合，仮想カーソルはその Web コンテンツの表示領域上に止まることはない．

Grossman らは，ある二次元平面上に複数存在するコンテンツの選択のためにバブルカーソル [2] を開発した．バブルカーソルとは，コンテンツを母形状とする二次元一般図形ボロノイ図の概念を用いるものであり，現在のグラフィクスカーソルの座標から最も近い場所にあるコンテンツを，常に一つだけ選択し続けるという振る舞いを持つものである．図 5.19 にその概観を示す．バブルカーソルが移動すれば，バブルカーソルと他のコンテンツとの距離も変わるため，この移動を適切に行えば，コンテンツの選択操作を効率的に行うことができる．

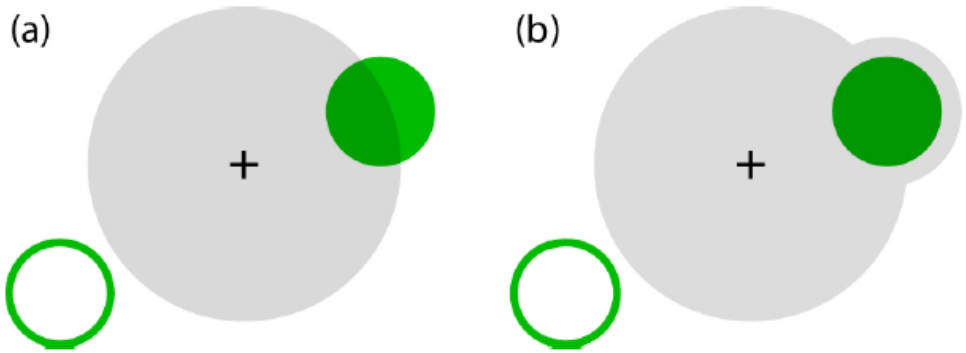

図 5.19: バブルカーソルの概観．(a) カーソルは常に一つのコンテンツだけを選択するよう，その半径を動的に調節する．(b) 選択するコンテンツがカーソルの領域に完全には含まれない場合，カーソルの領域はそのコンテンツを含むように拡げられる．(文献 [2]pp.283 の図 4 を引用)

しかし，一般にボロノイ図の計算量は大きい．仮想カーソルの移動のためにこの計算を行うと，UI の反応速度が著しく低下する可能性がある．そこで本研究は，事前に定めておいたある一定の幅だけ Web コンテンツの表示領域を上下左右の四方向へ拡大するアプローチをとることにした．この様子を図 5.20 に示す．この拡大

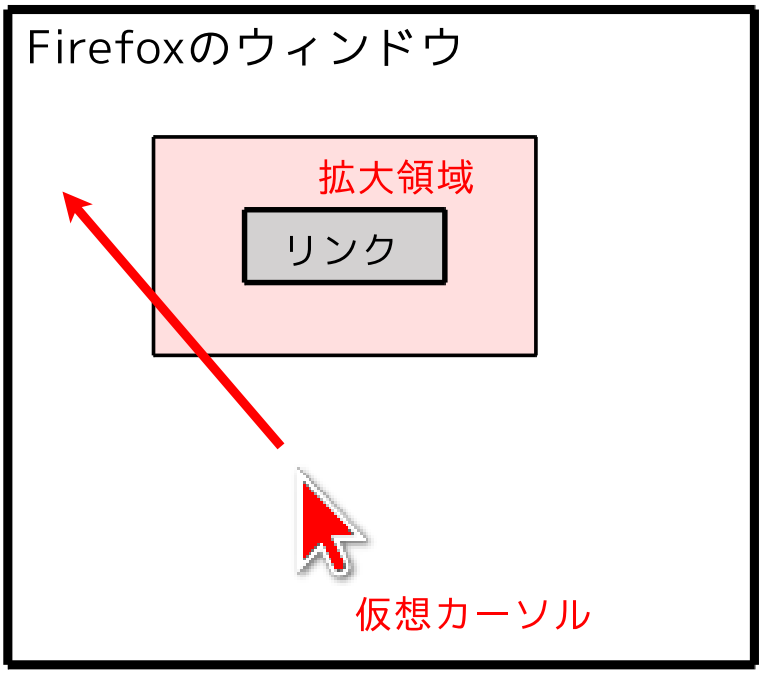

図 5.20: ある一定幅による，Web コンテンツの表示領域の拡大

に用いた幅を 10 ピクセルとしたとき，仮想カーソルが直感に反するような方向へ移動することが多かった．そこで，拡大に用いる幅を 5 ピクセルとして設定した．

粒度：大　Firefox のウィンドウの端への移動　DOM を用いることにより，Firefox のウィンドウの表示領域の大きさに関する情報が取得可能である．ウィンドウの端への移動の実装には，この情報を用いた．またこの移動操作が行われる時も，仮想カーソルの移動の軌跡を表示した．

仮想カーソルの移動軌跡の視覚化

前節で述べた，**粒度：中**，**粒度：大**である移動中，仮想カーソルの軌跡を視覚的に表示するよう，実装した．これにより仮想カーソルに対し，OS が提供するグラフィクスカーソルにさらに似た振る舞いを持たせることができる．

この実装には JavaScript のタイマを用いた．あるタイムスライスごとに仮想カーソルの表示座標を変更するよう，タイマを設定することによって仮想カーソルの軌跡を近似した．Card らの調査では，マウスを用いて行う一回のポインティング操作にかかる時間を，約 1.1 秒程度として見積もっている [1]．そこで本研究は，このタイムスライスと，移動における単位距離をそれぞれ 20ms，20 ピクセルとして設定した．値をこのように設定すると，1600x1200 のディスプレイを用いたときに，仮想カーソルが水平方向へ移動するときに最大で 1.6 秒，垂直方向へ移動するときに最大で 1.2 秒かかることになる．

JavaScript のタイマを用いて行う全ての操作は，一意のタイマ ID を持つ．仮想カーソルの移動の中断操作のために，このタイマ ID を用いた．この実装については次節で述べる．

仮想カーソルの移動を中断する操作の実装

ユーザが仮想カーソルの移動操作を誤ることを考慮し，仮想カーソルがピクセル単位でない移動を行うとき，これを中断できるよう，実装した．

前節で述べたように，仮想カーソルが**粒度：中**，**粒度：大**である移動を行うとき，その軌跡が表示される．軌跡の表示には JavaScript のタイマが用いられる．軌跡の表示に対して割り当てられたタイマ ID を参照し，その ID に対応する移動軌跡の表示を取り消すことにより，仮想カーソルの移動を中断する操作を実装した．

領域分割による，Web コンテンツの絞り込み操作の実装

Firefox のウィンドウの表示領域の分割操作は，その表示領域の大きさに関する情報を用いて実装した．4.4.3 節で述べたように，ある領域が選択されるごとに，その領域の大きさを縦方向，横方向ともに三分割するよう，実装した．本節ではその実装について述べる．

まず領域の分割を視覚的に表示するために，領域の境界を表示するよう，実装した．領域の境界は HTML の<SPAN>要素であり，これを Firefox が読み込む HTML 文書が持つ<BODY>要素の子要素として追加した．縦方向の三分割のために必要となる<SPAN>要素は四つであるため，横方向の三分割のために必要となるものと合計すると，一度の分割につき八つの<SPAN>要素を追加する必要がある．また，それらの<SPAN>要素にはそれぞれ一意の ID を設定した．ID を設定する理由は後述する．この様子を図 5.21 に示す．

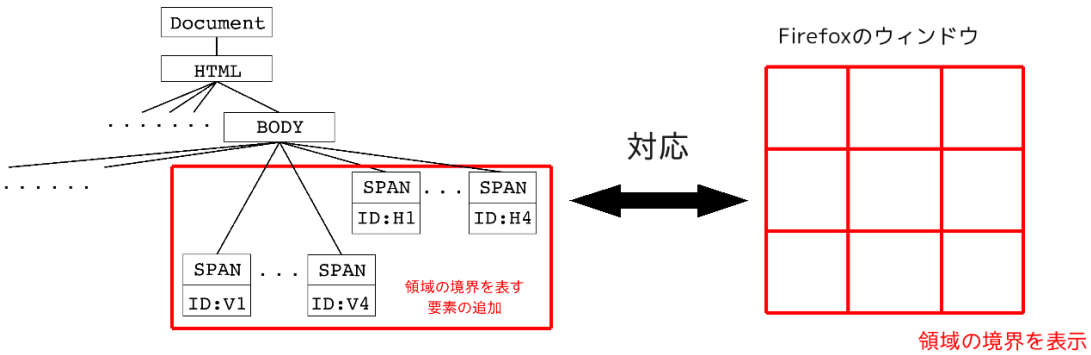

図 5.21: DOM ツリーへの<SPAN>要素の追加による，領域の境界表示

元々のディスプレイの表示領域の大きさを起点とし，領域分割が行われるたびに表示領域の縦幅と横幅をそれぞれ三分の一に設定していく．前述，仮想カーソルの移動はその選択領域内のみに制限するよう，実装した．また，ある領域が選択されているとき，その領域内に表示されているクリッカブルな Web コンテンツのみを対象とする選択を実装した．これらのいずれも，前述，仮想カーソルの移動範囲

を変更しただけのものである．

ある領域が選択されるとき，その領域内にあるクリッカブルなWebコンテンツの数によって行う操作を変えるよう，実装した．領域内に一つもWebコンテンツがない場合，全ての領域の境界を取り除き，領域分割の操作を初期化する．領域内に一つだけWebコンテンツがある場合，そのWebコンテンツの選択を行う．それ以外の場合は，領域分割を引き続き行う．

領域分割を行うたびに，既存の領域の境界を一度全て消去する必要がある．その消去のために，境界を表す<SPAN>要素のIDを用いる．このIDは一意であるため，DOMツリーに対し，IDを参照するだけで所望の<SPAN>要素を得ることができる．

5.2.5 ディスパッチャの実装

UIが入力コマンドを受け取るごとに，その入力コマンドに対して操作が割り当てられているかどうか，ディスパッチャはコマンドディスパッチテーブルの探索を行う．ユーザが入力するコマンドは多様である．そこで任意の入力コマンドに対し，ディスパッチャによる探索が可能であるように実装した．

ディスパッチャは，UIのモードによって探索するコマンドディスパッチテーブルを変更するよう実装した．たとえばフォームがフォーカスされている場合は，文書入力モードに対応するコマンドディスパッチテーブルを探索の対象とする．また仮想カーソルが表示されている場合は，グラフィクスカーソルモードに対応するものを対象とする．

以降，それらの実装について述べる．

入力コマンドの文字列表現

コマンドディスパッチテーブルの実体はJavaScriptのオブジェクトである．ある入力コマンドに操作を割り当てるとき，入力コマンドをオブジェクトのプロパ

ティに記述し，割り当てるべき操作をそのプロパティの値に記述する．

ユーザからの入力コマンドが必ず英字キーであるとは限らない．たとえ英字キーのみに操作を割り当てたとしても，存在しうる全ての入力コマンドによって，コマンドディスパッチテーブルが探索できなければならない．そのためディスパッチャは，入力コマンドを，一意に決まる文字列として表現する．ここで得られる文字列は，JavaScript のオブジェクトのプロパティとして存在しうるものである．

全てのキーは，印字が可能であるものと不可能であるものとに大別できる．印字が可能であるものは，その印字を文字列化して用いた．印字が不可能であるものには印字可能な一意のラベルを付け，それを用いた．また印字が可能であるかどうかは，キーイベントが持つ文字コード情報やキーコード情報を参照することで判別を行った．そのうち，キーイベントが持ちうるキーコード情報は，DOM によって全て定義されている．

ここで JavaScript における識別子は，以下の条件を満たす必要がある．

- 識別子の先頭の文字は以下のいずれかに限られる．
  - Unicode 文字
  - アンダースコア (’_’)
  - ドル記号 (’$’)
- 識別子の先頭以外の文字は以下のいずれかに限られる．
  - Unicode 文字
  - 数字
  - アンダースコア (’_’)
  - ドル記号 (’$’)

JavaScript のオブジェクトが持つプロパティも，この条件を満たす必要がある．そ

こで修飾キーを伴う入力コマンドは，接頭辞にその情報を持つような文字列として表現した．

接頭辞は，修飾キーを表現する一文字の英字の後にアンダースコアを付けたものである．ただし印字が可能であるキーがShiftキーを伴って入力された場合，この接頭辞を用いずに表現する．Shiftキーを伴って英字キーが入力された場合は，大文字英字として表現した．

英字キーでないキーがShiftキーを伴って入力される時，JavaScriptでは，実際にどのキーが押されたのか判別できない．そのためこれらのキーの表現には，Shiftキーとの同時押下によって印字される特殊文字を用いた．しかし，特殊文字はJavaScriptの識別子として利用不可能である．そこで一意のラベルを付けることによって，これらを表現した．

修飾キーを伴わない，数字キーの入力に対しては，アンダースコアを接頭辞とし，その後ろに数字を添えることによって表現した．

また複数個の入力コマンドの組み合わせも表現可能である．この場合，それぞれの入力コマンドをドル記号で連結させて用いる．

プロパティとして存在しうる形式で表現された入力コマンドの例を，表5.1に示す．また，入力コマンドを文字列として表現する過程を図5.22，5.23に示す．

入力コマンドによる，コマンドディスパッチテーブルの探索

JavaScriptでは，あるオブジェクト内に特定のプロパティが存在するかどうか，容易に確認できる．つまり，ある入力コマンドがあったとき，それがコマンドディスパッチテーブルに記述されているかどうかは容易に調べることができる．この様子を図5.24に示す．

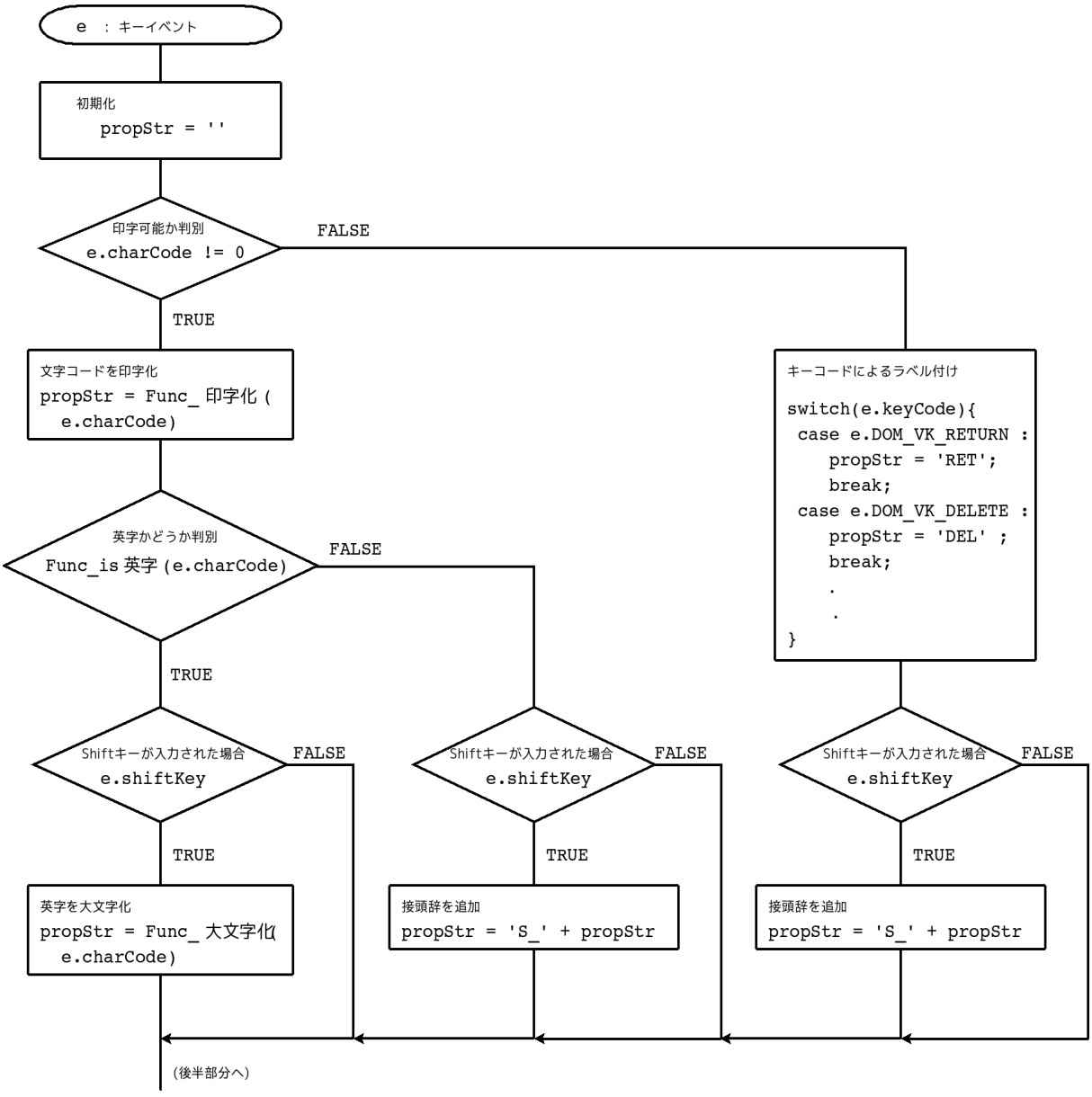

図 5.22: 入力コマンドを文字列表現する過程の前半部分

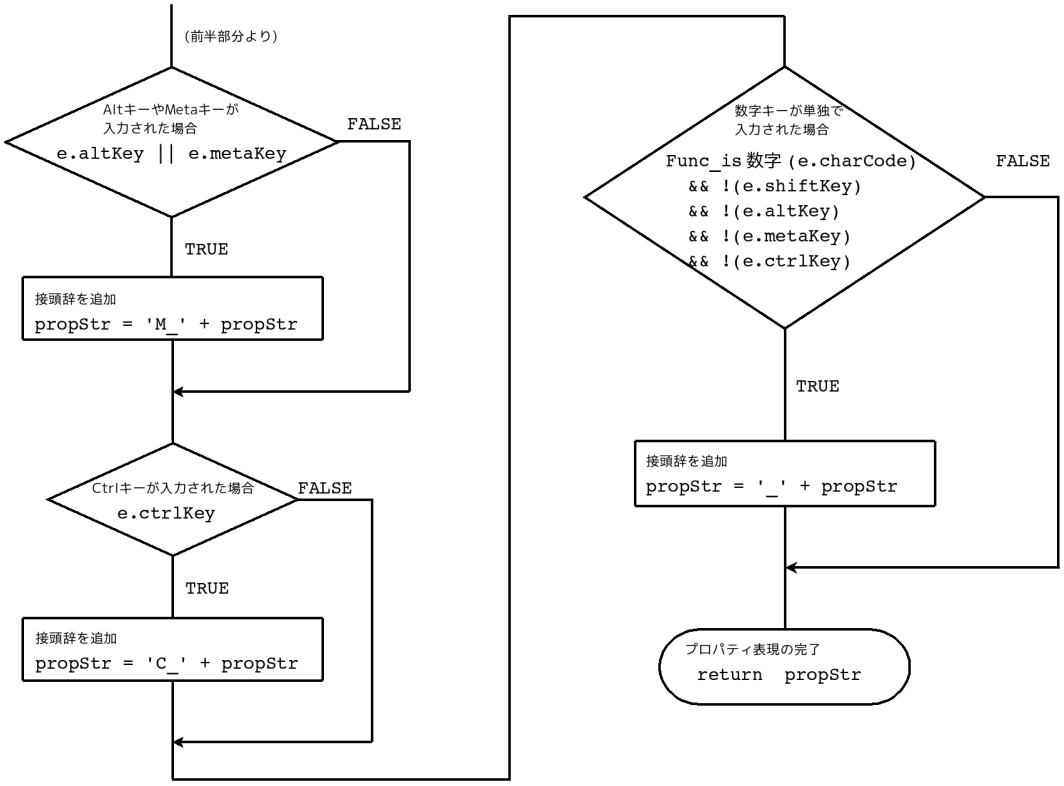

図 5.23: 入力コマンドを文字列表現する過程の後半部分

```
1   // 英字キー 'a' が単独で入力された場合,
2   // 入力コマンドを文字表現しても 'a' のままである.
3   InputCommand = 'a';
4
5   // コマンドディスパッチテーブルの探索を行う.
6   var operation = SearchTable(InputCommand);
7
8   //-----------------------------------------
9   // コマンドディスパッチテーブルに,
10  // 入力コマンドが登録されているかどうか探索を行う.
11  //-----------------------------------------
12  function SearchTable(input){
13    var operation = '';
14    // for/in ループによって, オブジェクトが持つプロパティを
15    // 全て探索する.
16    for (var ptr in CommandDispatchTable){
17      if (ptr == input){
18        // input で与えられるプロパティを持つ場合は,
19        // そのプロパティの値を返す.
20        operation = CommandDispatchTable[ptr];
21      }
22    }
23    return operation;
24  }
25
26  CommandDispatchTable = {
27    a : 'Func_A()'  ,
28    b : 'Func_B()'  ,
29  }
```

図 5.24: ディスパッチャによる，コマンドディスパッチテーブル探索の実装

表 5.1: 日本語配列キーボード利用時における，入力コマンドのプロパティ表現

キー入力	入力される修飾キー			
	なし	Shift	Alt	Shift + Alt
i	'i'	'I'	'M_i'	'M_I'
_	N/A	'UdrScore'	N/A	'M_UdrScore'
$	N/A	'DOL'	N/A	'M_DOL'
1	'_1'	'EXCLM'	'M_1'	'M_EXCLM'
gg	'g$g'	N/A	N/A	N/A

操作の実行

ある入力コマンドがコマンドディスパッチテーブルに登録されているとき，コマンドディスパッチテーブルの探索の結果として，行うべき操作が実装された関数名が得られる．ディスパッチャはこの関数を実行する．

第 6 章

実験および結果

本研究は，キーボードとマウスとの持ち替えに時間がかかることを問題にし，UIの設計および実装を行った．そこでUIの利用によって，Webブラウザの操作にかかる時間が短縮できるのかどうか実験を行った．

実験には，入力デバイスの持ち替えが含まれる必要がある．そこで複数のフォームを持つWebページを作成し，これを実験に用いた．またそれぞれのフォームに対し，決められた語句の入力操作をタスクとして設定した．なお入力操作の完了は，Webページ内に表示するボタンの押下によって行った．

また比較すべき問題を，持ち替えの手間だけに限定する必要がある．そこで入力すべき語句は，計算機の扱いに慣れたユーザにとって馴染み深いものを選択した．実験に用いた入力語句を以下に挙げる．

- ls
- cd
- rm
- cat

実験に用いたWebページを図6.1に示す．

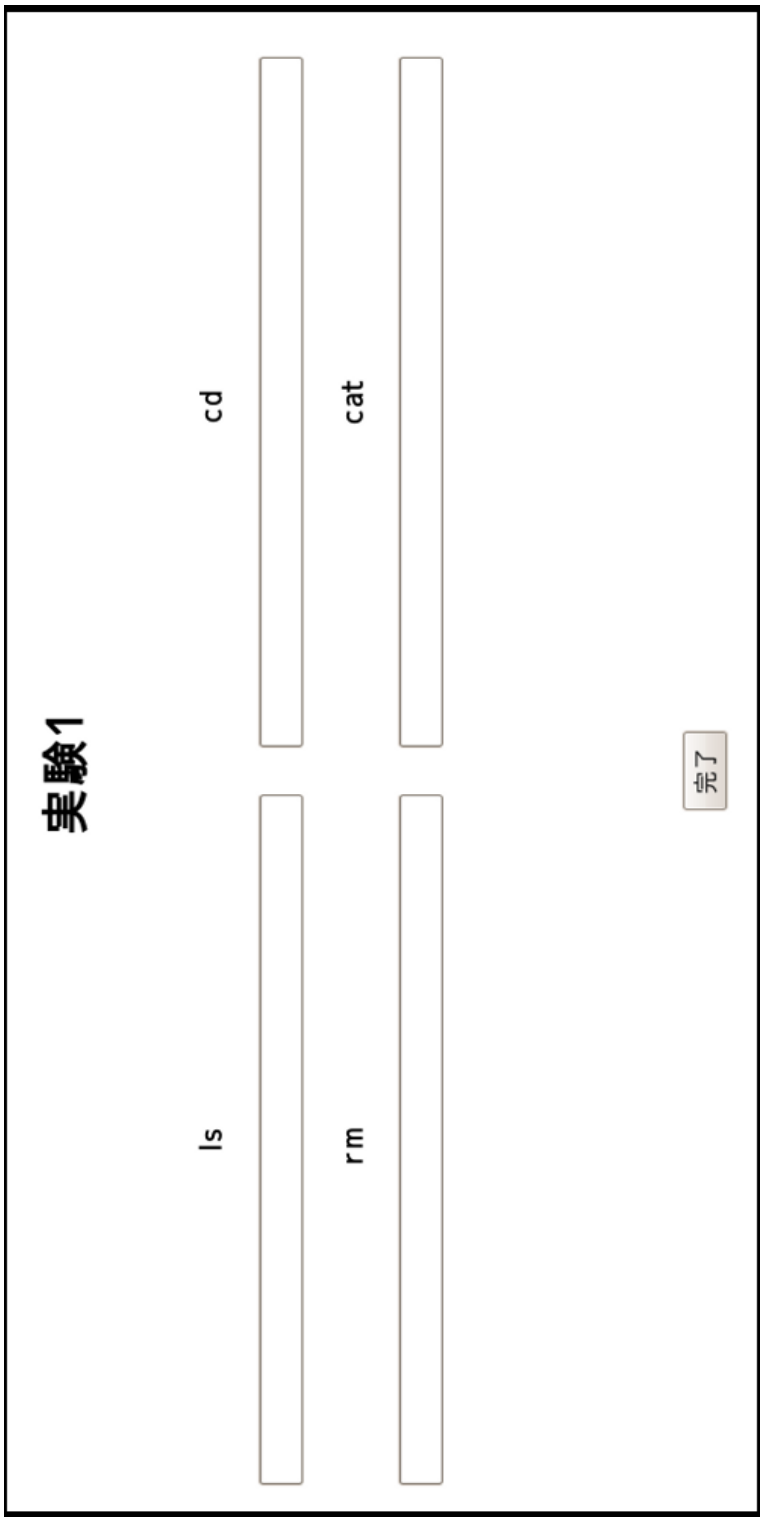

図 6.1: 実験に用いた Web ページ

タスクとして設定した入力操作を UI を利用して行うときと，キーボードとマウスとを持ち替えながら行うときとで，その完了までにかかる時間を比較した．時間の計測には，実験に用いた Web ページを対象とするキーイベントとクリックイベントのログを利用した．また実験に用いた Web ページが，Firefox 上に表示された瞬間に発行されるイベントのログも利用した．一つのイベントに対するログは表 6.1 に示す書式にしたがう．また，実験より得られたログの一部を図 6.2 に示す．

表 6.1: イベント別に設定した，ログの書式

イベント	書式
キーイベント	“KEY : ”+ (入力コマンドを表現する文字列) + (イベントの発行時間)
クリックイベント	“CLICK : ”+ (クリックされた座標) + (イベントの発行時間)
Web ページ表示イベント	“PAGESHOW : ”+ (Web ページのタイトル) + (イベントの発行時間)

```
 1   PAGESHOW  :  title:,time:38m,13sec,722msec
 2   PAGESHOW  :  title:,time:38m,13sec,726msec
 3   PAGESHOW  :  title:,time:38m,13sec,740msec
 4   PAGESHOW  :  title:,time:38m,13sec,945msec
 5   PAGESHOW  :  title:,time:38m,14sec,80msec
 6   PAGESHOW  :  title:,time:38m,14sec,94msec
 7   KEY  :  C_k,time:38m,15sec,360msec
 8   KEY  :  RET,time:38m,15sec,697msec
 9   PAGESHOW  :  title:Mozilla Firefox スタートページ,time:38m,16sec,706msec
10   CLICK  :  X:522,Y:20,time:38m,18sec,271msec
11   CLICK  :  X:541,Y:109,time:38m,19sec,590msec
```

図 6.2: 実験における，各種イベントのログ

実験は四人の被験者に対して行った．実験の開始前に，被験者には UI の操作方法を簡単に伝える内容のチュートリアルを受けてもらった．また被験者が UI の操

作に慣れてから実験を行う必要があるため，実験開始のタイミングは全て被験者に任せた．

実験を行った計算機の CPU は Intel(R) Pentium(R) D 3.20GHz であり，メモリを 4GB 搭載していた．また OS として Ubuntu 8.10，Web ブラウザとして Mozilla Firefox 3.0.5 を用いた．実験に用いたキーボードは PFU 製の Happy Hacking Keyboard Lite 2 であり，マウスには DELL 製のスクロールマウス MO56UOA を用い，ディスプレイには EIZO 製の FlexScan S2000 を用いた．

被験者より得られた結果を表 6.2 に示す．

表 6.2: 実験結果

被験者	打鍵数 [回]		クリック数 [回]		タスク完了までの時間 [秒]	
	通常	UI	通常	UI	通常	UI
A	9	41	8	5	10.737	64.551
B	9	157	5	6	8.713	67.960
C	9	220	5	5	8.650	62.052
D	9	46	5	5	9.992	48.145

表 6.2 より，UI の使用により打鍵数および操作完了までの時間が増加することが分かる．打鍵数については最大で約 25 倍まで増加し，操作完了までの時間は最大で約 8.5 倍まで増加した．なお実験に際し，UI の操作は十分に速く，UI 上の問題はない．つまり上記の結果は，UI が提供する機能が有用でないことを示すものである．

そこで実験後，被験者全員に対してアンケートを実施した．そのアンケートの設問と，それに対する被験者からの回答を順に挙げる．

設問 1. これからもこの UI を使ってみたいと思いましたか.
設問 2. UI の利用による，キータイプ数の増加は気になりましたか.
設問 3. 率直に言って，マウスと比べて使いやすいと思いましたか.

設問 1. に対しては，“まったく思わない” から “使ってみたい” までの四項目を設けた．設問 2. に対しては，“非常に気になる” から “気にならない” までの四項目を設けた．設問 3. に対しては，“使いにくい” から “使いやすい” までの四項目を設けた．

また上記の設問が持つ四項目について，そのそれぞれに対して 1 点から 4 点のスコアを与えた．その後，被験者より得られた回答の平均点を算出した．その結果を以下に示す．

- 設問 1. より得られた平均点：1.750 点
- 設問 2. より得られた平均点：2.750 点
- 設問 3. より得られた平均点：2.000 点

設問 2. の結果より，打鍵数の増加はそれほどユーザの操作感に影響を与えていないことが分かる．しかし設問 1. や設問 3. の結果からは，UI がユーザにとって使いづらいものであることが分かる．

設問 4. UI の操作は覚えやすいと思いましたか.

設問 4. に対しては，“覚えにくい” から “覚えやすい” までの四項目を設けた．また，これらのそれぞれに対して 1 点から 4 点のスコアを与え，被験者より得られた回答の平均点を算出した．その結果を以下に示す．

- 設問 4. より得られた平均点：3.000 点

この結果は，UI の操作方法を全て vim として統一し，設計を行ったことを反映していると考えられる．また，操作方法の統一を図ったことにより，打鍵数の増加が被験者の操作感にさほど影響しなかった可能性がある．

設問 5. 日常の計算機操作において，もっとも頻繁に使うエディタは何ですか.

設問 5. に対しては，被験者には自由に記述してもらった．その結果を以下に示す.

- Emacs ： 2 名
- notepad : 2 名

この結果と設問 4. の結果とを考慮すると，普段から vim を利用していない場合であっても，UI の操作方法は比較的覚えやすいものであると言える．この結果も，操作方法を統一したことが反映されていると考えられる.

設問 6. UI が採用した，クリッカブルな Web コンテンツのみを対象とする操作の使用感について，感想をお願いします.

設問 6. に対しては，被験者には自由に記述してもらった．その中で得られた代表的な回答を，以下に挙げる.

- グラフィクスカーソルの操作の効率化のためには，慣れがかなり必要である.
- 今のままではやや使いづらい.
- 改良を行えば，より使いやすいものになる可能性がある.

仮想カーソルに対しては，クリッカブルな Web コンテンツを対象とする機能を持たせた．設問 6. の結果より，その機能は有用であると言える．しかし全ての被験者より，その機能は改良の余地があるとの回答が得られた．また，仮想カーソルの操作性および仮想カーソルの操作に対する慣れについては次章にて述べる.

設問7. UIの使用感についてお気に召さない点，あるいは改良点等についての感想をお願いします.

設問7. に対しては，被験者には自由に記述してもらった．その中で得られた代表的な回答を以下に挙げる.

- 仮想カーソルの移動速度が速すぎる.
- 仮想カーソルの移動を中断する操作が難しい.
- フォームに入力するだけのために，vim の利用を覚えるのは面倒である.

仮想カーソルの移動は，i というコマンドによって中断が可能であるように設計した．しかし被験者が仮想カーソルの移動を中断しようと思っても，移動速度が速すぎるため，被験者が意図する位置に仮想カーソルを止められないという回答が多かった．これより仮想カーソルの移動速度や操作方法に関しては，再考の余地があると言える.

また，フォームへの入力も vim と同じ方法で行えるよう，UI の設計を行った．仮想カーソルによってフォームを選択した直後のモードは，フォームへの入力が行えない編集モードであるよう，UI の設計を行った．被験者からは，フォームを選択した直後に文字の入力を行えれば，UI における文字入力の操作性が向上する可能性があるとの回答が得られた.

これらより，ユーザが求めるフォームへの入力方法は様々であり，これに柔軟に対応することが必要であると分かった．UI が持つべき操作の柔軟性については次章にて述べる.

第 7 章

考察

実験では，UI の有用性を示す結果が得られなかった．また，被験者に対して行ったアンケートからは，慣れが必要であるとの回答が得られた．つまり，実験を開始する前に被験者に対して行ったチュートリアルだけでは，不十分であるという結果が得られた．そこで参考までに，UI に最も慣れていると考えられる，設計者みずからが被験者と全く同じ状況で実験を行った．その結果を表 7.1 に示す．

表 7.1: 設計者みずからが被験者と同じ実験を行った結果

被験者	打鍵数 [回]		クリック数 [回]		操作完了までの時間 [秒]	
	通常	本 UI	通常	本 UI	通常	本 UI
設計者	9	38	5	5	10.440	10.190

表 7.1 より，UI の利用によって操作完了までにかかる時間は減少したことが分かる．しかし，その減少の幅は小さい．つまり，UI に対して非常に高い習熟度を持つ場合であっても，それほど操作の効率が向上しないと言える．

また前章に述べたアンケートの結果からは，仮想カーソルの移動速度や，その操作性が問題であると分かった．Card ら [1] は，マウスによってグラフィクスカーソルを操作し，それによって行うポインティングに要する時間を見積もっている．本研究においては，仮想カーソルの移動速度を Card らの手法における見積り時間

をモデルとして実装した．しかし，その実装ではUIの有用性を示すことができなかった．

Cardらは通常行われる計算機操作に対し，その操作にかかる時間の見積りについて調査した．つまりCardらは，あくまでマウスを用いるポインティング操作にかかる時間の見積りを行っただけである．本研究は，Webコンテンツの選択までにかかる時間という観点でCardらの手法をモデル化した．しかし，それらの間には大きな差異があったため，UIの有用性を示すことができなかったと考える．

またアンケートからは，仮想カーソルが持つ現在の移動操作だけでは操作性が悪いという回答が得られた．これらの回答の多くは，ユーザが移動を指示した方向と，ユーザが期待する仮想カーソルの移動方向との間に差異があることについて言及していた．この問題は，バブルカーソル [2] と同様の機能を仮想カーソルに実装することで劇的に改善する可能性がある．

しかしバブルカーソルの実装に必要となる，二次元一般図形ボロノイ図の計算量は大きい．UIには応答性が求められるため，この実装には計算量を抑えることが求められる．

OSが提供するグラフィクスカーソルには，加速度を設定する機能がある．しかし仮想カーソルは，加速度を持たない．仮想カーソルの移動に加速度を持たせることにより，UIを使用することでWebコンテンツ選択を効率的に行えるようになる可能性がある．

計算機の扱いに慣れており，キーボード操作をいとわないユーザを想定し，実験を行った．実験は，それらのユーザにとって馴染み深い単語をフォームへ入力するものである．そのような限定された状況においてもWebブラウザの操作効率は向上しなかった．一般にWebブラウザの操作を行うとき，ユーザがフォームへ入力する単語はそれらだけではない．そのとき，vimの操作に慣れていないユーザの操作効率はさらに低下する可能性が高い．

UIは操作方法をvimに固定したものであるが，アンケートからはvimでなくて

もよいという意見が得られた．エディタと同じ方法で Web ブラウザを操作することを考案するとしても，数種類のエディタを想定し，それぞれと同じ方法によって Web ブラウザの操作を可能にする必要がある．

第 8 章

今後の課題

Web ブラウザの操作を行う際にマウスとキーボードとの持ち替えが発生していることを問題とし，本研究はキーボードのみで Web ブラウザの操作を可能とする UI を実装した．また，UI には Web コンテンツの選択専用である，仮想カーソルを導入した．仮想カーソルは，Firefox のウィンドウ内に表示されているクリッカブルな Web コンテンツのみを選択対象とすることができる．

仮想カーソルは HTML 文書の<BODY>要素の子要素として実装したため，<BODY>要素を持たない HTML 文書に適合できない．<BODY>要素を持たない HTML 文書として，フレームによって構成されている Web ページが挙げられる．一般にフレーム内には，それぞれ異なる Web ページが表示されることが多い．

フレームは，<FRAME>要素として HTML 文書内に記述される．フレーム内にはそれぞれの Web ページが表示される．つまり全ての<FRAME>要素は，その子要素として<BODY>要素を持つ．フレームが持つ<BODY>要素へは，仮想カーソルの追加が可能である．これを利用することにより，本 UI が利用できる状況は増えると考えられる．

仮想カーソルによる選択ができない Web コンテンツが存在する．その例として，Flash が挙げられる．Flash の表示領域に仮想カーソルが重なると，仮想カーソルは Flash に隠れて見えなくなってしまう．Web コンテンツよりも手前側に見えるよう，仮想カーソルが持つ CSS のプロパティの値を設定したが，この方法では Flash を選択することができない．

また，クリッカブルであるとして抽出できないWebコンテンツが存在する．本研究ではXPathを用いることで，Webコンテンツがクリッカブルであるかどうかを判断した．XPathは，HTML文書内に存在する全ての要素に対して問い合わせが可能である．しかしXPathは<EMBED>要素によって埋め込まれる，Flashなどのオブジェクトの内容まで関知できない．そのため，たとえそのオブジェクトがクリッカブルであったとしても，XPathでは抽出できない．

それらを全て扱えるようになれば，仮想カーソルの利用価値，ひいては本UIの利用価値は今よりも高くなると考えられる．

第 9 章

関連研究

マウスとキーボードとの持ち替えの手間をなくす試みとして，まず TrackPoint が挙げられる．TrackPoint は ThinkPad に内蔵されたポインティングスティックである．ThinkPad が持つキーボードのホームポジションと近い場所に TrackPoint は存在する．これを用いることにより，キーボードとマウスとを持ち替えることなく，OS が提供するグラフィクスカーソルの操作を行うことができる．しかし Web ブラウザの操作を行うためだけに，専用の筐体を導入するコストは高いという問題がある．一方，Web ブラウザに追加可能な UI として導入すれば，筐体を選ばずに利用することができる．

ユーザの足を用いることでグラフィクスカーソルの制御が可能な入力デバイス [5] やユーザの視線を検知する入力デバイス [7] が存在する．これらを用いることによっても，マウスとキーボードとの持ち替えは必要なくなる．しかしこれらにも上述同様のことが言える．

w3m[13] や Lynx[14] はテキストブラウザである．これらの操作は全てキーボードのみで行えるよう，設計されている．しかしテキストブラウザであるがゆえ，表示できない Web コンテンツが多数存在する．キーボードとマウスとの持ち替えをなくす目的のために，閲覧できる Web コンテンツが減少するコストは高い．

Opera[15] は空間ナビゲーションという機能を備えている．この機能には Shift キーと矢印キーを用いる．現在フォーカスされている Web コンテンツを起点とし，矢印キーが示す方向にある Web コンテンツへフォーカスを移す機能である．これ

により，Web コンテンツの選択が効率的に行える．起点からある方向に向かってフォーカスを移すという点では本研究と似ているが，この選択操作にはグラフィクスカーソルを用いない．その点が本研究と異なる．

Firefox を対象とした UI の中にも，持ち替えの手間をなくす複数の試みが見られる．それらの一つとして，Hit-a-Hint[18] が挙げられる．Hit-a-Hint は，クリッカブルな Web コンテンツに一意のラベルを付ける．Hit-a-Hint そのラベルのキータイプと，ラベルに対応付けられた Web コンテンツの選択操作とを同義として定義している．これにより，Web コンテンツの選択がキーボードのみによって効率的に行える．しかし，Hit-a-Hint は Web コンテンツに対するラベルこそ視覚化するものの，グラフィクスカーソルを用いるものではない．その点が本研究と異なる．

また Firefox をエディタと同じように操作可能にする UI として，vimperator[16] と Firemacs[17] が挙げられる．それぞれ，vim，Emacs と同じ操作方法によって，Firefox の操作を可能にするものである．これらを用いれば，Firefox の操作は効率的に行えるようになる．しかしそのいずれも，クリッカブルな Web コンテンツの選択の際にはグラフィクスカーソルを用いない．その点が本研究と異なる．

第 10 章

おわりに

Web ブラウザの操作にはマウスとキーボードとの持ち替えが含まれる．そこで本研究はその持ち替えの手間をなくすべく，キーボードのみで Web ブラウザの操作が可能となるような UI を設計し，実装した．また操作効率の向上を図るため，通常キーボードを用いて行っている，エディタの操作と同じ操作方法によって Web ブラウザの操作を可能にするよう，UI を設計した．

まず通常マウスを用いて行う操作は，すべて OS が提供しているグラフィクスカーソルの操作であることに注目した．そこで本研究はキーボードのみで操作ができるような，グラフィクスカーソルを実装した．本研究が実装したグラフィクスカーソルは，OS が提供するものとは異なるものである．

次に，通常マウスのクリック操作はリンクやプルダウン等，クリッカブルな Web コンテンツに対してのみ行われると仮定した．クリック操作がそれらに対してだけ行われるならば，グラフィクスカーソルは，それらの表示領域の上だけを順にたどればよい．そこで本研究が実装したグラフィクスカーソルには，その操作を可能にするような機能を持たせた．

数人の被験者に対して行った実験からは，UI を利用しても Web ブラウザの操作効率は向上しないという結果が得られた．しかし，本研究が実装したグラフィクスカーソルの操作性を改良することにより，Web ブラウザの操作効率は向上する可能性があることを見出した．

謝辞

本研究を遂行するにあたっては，いろいろな方々にお世話になりました．

まず，指導教員の多田好克先生と佐藤喬助教には日頃から熱心なご指導，そしてご鞭撻を賜わりました．また，ご多忙中にもかかわらず，論文の草稿を丁寧に読んで下さった上，大変貴重なご助言を頂きました．ここに厚く御礼申し上げます．

加えて，本研究が行えたことは，研究方針や方法論について日頃より活発に議論をし，共に充実した研究生活を送ってきた多田研，小宮研，村山研，水野研の学生諸氏のおかげでもあります．これらの学生諸氏にも深く感謝致します．

また，研究活動に起因する心身の疲労の解消には，電気通信大学フォークソング部の皆さんとのお付き合いが欠かせませんでした．皆さんにも深く感謝致します．

様々な条件付きではありましたが，研究活動について何も言わず，ただ見守って頂いた両親には格別の念をもって感謝致します．

最後に，これら全ての方に深く感謝致します．

参考文献

[1] Stuart K. Card, *et al.* ： The Keystroke - Level Model for User Performance Time with Interactive Systems, Comm. ACM. Vol. 23, No. 7, pp. 396–410 (1980).

[2] Tovi Grossman, and Ravin Balakrishnan : The Bubble Cursor : Enhancing Target Acquisition by Dynamic Resizing of the Cursor's Activation Area, ACM Proc. of the SIGCHI conference on Human factors in computing systems, pp.281–290 (2005).

[3] 遠藤光, 伊藤久祥, 伊藤憲三 : 再帰的画面分割を用いた肢体不自由者向けポインティング補助手法の開発と評価，電子情報通信学会技術研究報告，WIT, Vol. 107，No.555，pp. 73–78 (2008).

[4] 辰巳勇臣，野田尚志，旭敏之：限定された入出力環境でのポインティング手法の提案，電子情報通信学会総合大会講演論文集，Vol. 2003 年_基礎・境界, pp. 281 (2003).

[5] 久米祐一郎、井上啓：両足操作型ポインティングデバイスの検討、映像情報メディア学会誌、Vol. 54, No. 6, pp. 871–874 (2000).

[6] 角田博保，久野靖：流れて行かない Unix 環境の評価，情報処理学会 第 29 回冬のプログラミングシンポジウム報告集，pp. 13–22 (1988).

[7] 久野悦章，八木透，藤井一幸，古賀一男，内川嘉樹：EOG を用いた視線入力インタフェースの開発，情報処理学会論文誌 Vol.39, No.5, pp.1455–1462 (1998).

[8] Cascading Sytle Sheets, `http://www.w3.org/Style/CSS/`

[9] XML User Interface Languate (XUL) Project,
http://www.mozilla.org/projects/xul/

[10] W3C Document Object Model, http://www.w3.org/DOM/

[11] ユーザー補助でマウス操作をテンキーでする方法，Microsoft サポートオンライン，文書番号 879728，リビジョン 1.3,
http://support.microsoft.com/kb/879728/ja

[12] Mac OS X: キーボードを操作して機能を実行する方法，記事 HT2840,
http://support.apple.com/kb/HT2840?viewlocale=ja_JP

[13] w3m, http://w3m.sourceforge.net/index.en.html

[14] Lynx, http://lynx.browser.org

[15] Opera, http://js.opera.com

[16] vimperator, http://vimperator.mozdev.org/

[17] Firemacs, http://www.mew.org/~kazu/proj/firemacs/

[18] Hit-a-Hint, http://addons.mozilla.org/ja/firefox/addon/1341