Fachhochschule Frankfurt am Main
University of Applied Sciences

Bachelor Thesis

Daniel Baulig,
eingereicht am 18.04.2011

High Performance ECMAScript und HTML5 Canvas

Einsatz Moderner Webtechnologien für Onlinespiele

Betreuer

Prof. Dr. Jörg Schäfer,
Fachhochschule Frankfurt am Main

Prof. Dr. Matthias Schubert,
Fachhochschule Frankfurt am Main

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst habe und keine anderen als die angegebenen Hilfsmittel verwendet habe. Die Arbeit wurde in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde zur Erlangung eines akademischen Grades vorgelegt.

Gelnhausen, den 18.04.2011

_____

Daniel Baulig

Zusammenfassung

Diese Bachelor Thesis befasst sich mit der Leistungsfähigkeit modernen ECMAScript und HTML5 Canvas Implementationen. Anhand eigenständig erhobener Messdaten werden die Leistung einzelner Komponenten und Herangehensweisen bestimmt und verglichen. Anschließend werden Handlungsempfehlungen erarbeitet und abschließend untersucht inwiefern HTML5 Canvas und ECMAScript geeignet sind um anspruchsvolle Anwendungen wie Computerspiele damit zu entwickeln.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

1 Einführung

Als Tim Berners-Lee 1992 seine Idee eines World Wide Web formulierte[1], beschrieb er einen Traum. Den Traum von untereinander vernetzten Dokumenten. Von überall aus zugänglich – unabhängig von Ort, Zeit und Plattform. Berners-Lee konnte nicht ahnen, wie sein Traum die Welt verändern würde - und auch nicht, wie die Welt seinen Traum verändern würde. Seit der Entwicklung von Javascript durch Netscape 1996[2] und spätestens seit der Entdeckung von AJAX durch James Garret 2005[3] ist das WWW nicht mehr nur eine Plattform für vernetzte Dokumente, sondern eine Plattform für vernetzte Applikationen.

Zu den kommerziell erfolgreichsten Anwendungen im Web gehören Social- und andere Browsergames[4]. Doch die bisher mangelnde Unterstützung von Multimedia- und dynamischen Inhalten durch die Browserplattform nötigte die Hersteller stets für die Entwicklung solcher Spiele auf Plugin-Middleware wie Adobe Flash oder Microsoft Silverlite zurückzugreifen. Durch diesen Mangel der Browserplattform konnte vor allem Flash eine marktdominierende Verbreitung erreichen[5].

Doch Flash steht in vielen Bereichen Berners-Lees Idee einer offenen, vernetzten Webplattform konträr gegenüber. So ist Flash kein Teil des offenen Web-Standards und wird als proprietäre Technologie durch Adobe vertrieben, es integriert sich nicht ubiquitär in die vernetzte Struktur des Webs, sondern kapselt sich strikt von anderen Webinhalten ab und wirkt dadurch wie ein Fremdkörper. Zusätzlich ist Flash auf vielen Plattformen nicht verfügbar und schränkt somit die universelle Verfügbarkeit von Webinhalten ein.

Der kommende Webstandard HTML5 versucht diese und weitere Schwächen der Webplattform auszubessern und bringt nun viele Features, für die zuvor auf Flash oder andere Plugin-Lösungen zurückgegriffen werden musste, direkt in den Browser. So können beispielsweise Audio- und Video-Elemente direkt eingebettet und angesteuert werden. Websockets erlauben den Datenaustausch mit dem Server über nicht HTTP-basierte Netzwerkverbindungen. Und das Canvas-Element ermöglicht es auf einer Bitmapfläche zu zeichnen und Bilder pixelgenau auszulesen und zu manipulieren.[6] Dies ermöglicht es in der Zukunft Browsergames unabhängig von Drittanbieter-Plugins und ausschließlich mit offenen Webtechnologien zu entwickeln.

Ziel dieser Arbeit ist es die Leistungsfähigkeit von HTML5 für den Einsatz in Browsergames zu untersuchen. Dabei konzentriere ich mich auf die Analyse der Leistung von ECMAScript und der Canvas-Element Implementierungen und untersuche mögliche Ansätze zur Laufzeit- und Leistungsoptimierung. Die Zahlen werden mithilfe der offenen Plattform jsPerf generiert, welche einfach vergleichbare und statistisch relevante Werte liefert[7]. Alle Testfälle wurden mindestens dreimal ausgeführt und können auf jsPerf eingesehen und wiederholt werden.

Alle Messungen wurden unter Windows 7 Professional (64 Bit) in den aktuellen Versionen der verbreitetsten Browser vorgenommen. Die eingesetzten Browser sind Google Chrome 10

1 [Berners-Lee, 1992]
2 [Zakas, 2010], S. xi
3 [Garret, 2005]; es wird von Entdeckung und nicht Erfindung gesprochen, da alle zu AJAX gehörenden Technologien und Methoden bereits zuvor bekannt waren und eingesetzt wurden. Lediglich die Zusammenstellung und formelle Definition wurden durch Garret ergänzt. Siehe auch [Crockford, 2010a], 1:14:40 bis 1:17:09.
4 [SPON, 2010]
5 [Adobe, 2010]
6 [WHATWG, 2011], Abschnitte 4.8.6, 4.8.7, 4.8.11 und [W3C, 2011]
7 [JSPERF], Abschnitt "Which benchmarking engine is being used?"; siehe auch [Bynens, 2010]

(10.0.648.204), Mozilla Firefox 4.0 und Internet Explorer 9 (9.0.8112.16421)[8]. Es wurden ausschließlich die 32 Bit Versionen der Browser eingesetzt, sofern die Browser in mehreren Varianten verfügbar sind.[9] An den Browsereinstellungen wurden keine Änderungen vorgenommen. Dies hat besondere Bedeutung für die Canvas Messungen, denn Firefox 4 und Internet Explorer 9 verfügen über Hardwarebeschleunigung für das Canvas Element. Chrome verfügt zwar ebenfalls über die Option einer Hardwarebeschleunigung für Canvas, hat diese aber in der aktuellen Version nicht standardmäßig aktiviert.

2 ECMAScript

ECMAScript, besser bekannt als JavaScipt, ist eine dynamische, schwach typisierte und objektorientierte Programmiersprache. ECMAScript[10] wurde 1995 im Auftrag von Netscape durch Brendan Eich entwickelt. Eich ließ sich dabei stark von den Programmiersprachen Self, Scheme und Java beeinflussen. Von Self erbte ECMAScript die klassenlose, auf Prototypen basierte Objektorientierung, von Scheme verschiedene Funktionale Eigenschaften wie anonyme Funktionen und Closures, und von Java die Syntax und einige wenige Bibliotheken – inklusive ihrer Bugs.[11]

Die Entwicklung immer aufwendiger und gößerer webbasierter Applikationen beflügelte die Leistungsfähigkeit von ECMAScript in den letzten Jahren enorm. Die neueste Browsergeneration implementiert hochgradig optimierende JIT Compiler, Hardwarebeschleunigung und viele weitere Features, mit dem Ziel die Grenzen der Browserperformance weiter zu verschieben. ECMAScript ist inzwischen eine der performantesten dynamischen Sprachen und kommt in manchen Bereichen sogar erstaunlich nahe an die Performance vollständig typisierter und kompilierten Sprachen wie Pascal oder C heran[12].

2.1 Eigenschaften

Einige der ECMAScript Eigenschaften und Konzepte können einen signifikanten Beitrag zur Performance einer Anwendung beitragen. Das Verständniss dieser Eigenschaften und Konzepte und der damit verbundenen Auswirkungen ist daher von grundlegender Bedeutung für die Interpretation der folgenden Performance-Tests und möglicherweise auftretenden Engpässe. Darum werde ich zunächst einige der grundlegenden Konzepte und deren Implementation vorstellen.[13]

2.1.1 Dynamische und schwache Typisierung

ECMAScript ist eine dynamische und schwach typisierte Programmiersprache. Dies hat verschiedene Auswirkungen für die Programmierung in ECMAScript. Zunächst bedeutet das, dass die Sprache zwar das Konzept verschiedener Datentypen kennt, diese aber beim deklarieren einer Variable nicht explizit angegeben werden müssen und erst zur Laufzeit bestimmt wird. Weiter ist es in ECMAScript möglich, dass sich der Typ einer Variablen zur Laufzeit durch Zuweisung ändert. ECMAScript versucht außerdem implizit alle Argumente einer Operation in passende Datentypen

8 Die Angabe auf der jsPerf Webseite ist **falsch**. Es wurde die hier angegebene Version des Internet Explorers verwendet.

9 ua. um die Vergleichbarkeit zu erhöhen, aber auch weil der Entwicklungsstand mancher 64 Bit Varianten stark hinterher hinkt (zB. kein JIT Compiling in IE9 64 Bit), siehe auch [Law, 2009], Abschnitt "Why does 64bit IE9 get faster JavaScript benchmark scores than IE8 but slower scores than 32bit IE9?"

10 Zur der Zeit LiveScript bzw. JavaScript

11 [Powers, 2009], S. xi; [ECMA, 2009], S. 2; [Eich, 2008]; siehe auch [Crockford, 2010b], 0:04:47 bis 0:06:40.

12 [Fulgham, 2011]

13 Für eine ausführlichere Einführung in ECMAScript empfehle ich [Crockford, 2008]

umzuwandeln, sollten die gegebenen Datentypen nicht den erwarteten Datentypen entsprechen.

```
1  // Eine Angabe des Typs ist nicht notwendig
2  var i = 23,
3      s = "Hello, World!",
4      f = 1.337;
5
6  // Der Typ lässt sich zur Laufzeit ändern
7  i = 1.337;
8  s = 23;
9  f = "Hello, World!";
10
11 // "Type Coercion" wandelt Typen implizit um
12 i = s + f;
```

Quelltext 1: Dynamische und schwache Typisierung

2.1.2 Array- und Objectliterals

In ECMAScript ist es möglich Felder und Objekte mitsamt ihrem Inhalt, Methoden und Eigenschaften direkt zu notieren. Man spricht dabei von Array- bzw. Objectiterals. Während Erstere auch in vielen anderen Programmiersprachen bekannt sind, so ist Letzteres außergewöhnlich und nur in wenigen Programmiersprachen möglich.

```
1  // Ein Array mit den Werten 1, 2, 3
2  var a = [1, 2, 3];
3
4  // Ein Objekt mit der Eigenschaft foo
5  // und der Methode bar
6  var o = {
7      foo: 23,
8      bar: function () {
9          return this.foo;
10     }
11 };
```

Quelltext 2: Array und Objectliterals

2.1.3 Prototypenbasierte Objektorientierung

ECMAScript verfügt wie die meisten modernen Programmiersprachen über das Konzept der Objektorientierung. Im Gegensatz zu den meisten objektorientierten Sprachen kennt ECMAScript allerdings das Konzept einer Klasse nicht. Ihre Eigenschaften und Methoden erhalten Objekte in ECMAScript nicht von Klassen, sondern von anderen Objekten, ihren sog. Prototypen. Jedes Objekt hat genau einen Prototypen. Wird auf eine Methode oder eine Eigenschaft eines Objektes lesend zugegriffen, über welche das Objekt nicht verfügt, so wird versucht diese Eigenschaft oder Methode beim Prototypen des Objektes zu finden. Dies wird rekursiv wiederholt bis entweder die Eigenschaft oder Methode gefunden wurde oder sichergestellt wurde, dass kein Objekt in der Prototypen-Kette über die gesuchte Eigenschaft oder Methode verfügt. Schreibzugriffe erfolgen immer auf das Objekt selbst und beeinflussen die Prototypen-Kette nicht.

Eigenschaften und Methoden werden einem Objekt hinzugefügt indem sie bei der Erzeugung des Objektes angegeben oder nachträglich hinzugefügt werden. Außerdem können Eigenschaften und Methoden aus Objekten wieder entfernt werden. Änderungen an einem Prototypen können das

Verhalten und die Eigenschaften von auf diesen Prototypen basierenden Objekten ändern. Objekte sind somit in ECMAScript, genau wie Variablen, äußerst dynamische Konstrukte.

```
1  // Erzuege ein Objekt mit Object als Prototyp
2  var o = Object.create(Object);
3
4  // Füge dem Objekt eine Eigenschaft und eine Methode hinzu
5  o.foo = 23;
6  o.bar = function () {
7      return this.foo;
8  };
9
10 // Erzeuge ein Objekt mit dem Prototypen o
11 var o2 = Object.create(o);
12
13 if (o2.bar() == o2.foo) {
14     // o2 verfügt, durch den Prototypen o, über die
15     // Methode bar und die Eigenschaft foo
16 }
17
18 // Schreibe o2.foo und erzeuge diese Eigenschaft
19 // somit auf o2 selbst!
20 o2.foo = 42;
21
22 if (o2.foo == 42 && o.foo == 23) {
23     // Der Prototyp bleibt davon unberührt!
24 }
25
26 o.bar = function () {
27     return 0;
28 };
29
30 if (o2.bar() == 0) {
31     // Die Änderung des Prototypen hingegen hat auch
32     // auf o2 Auswirkungen!
33 }
34
35 // entferne die Eigenschaft foo ust dem o2 Objekt
36 delete o2.foo;
37
38 if (o2.foo == o.foo) {
39     // Da o2 nun nicht mehr über die eigene Eigenschaft
40     // foo verfügt, wird stattdessen für den Lesezugriff
41     // die Eigenschaft foo des Prototypen gefunden.
42 }
```

Quelltext 3: Prototypenbasierte Objektorientierung

Der Prototyp eines Objektes muss zum Zeitpunkt der Erzeugung angegeben werden und lässt sich nachträglich nicht mehr ändern. Wird kein Prototyp angegeben, so wird das Standardobjekt `Object` automatisch Prototyp des Objektes. Alle Prototypen-Ketten in ECMAScript enden mit `Object`. Somit „erben“ alle Objekte von `Object`. `Object` ist das einzige Objekt in ECMAScript ohne einen eigenen Prototypen. Beim erzeugen eines Objektes mithilfe eines Objectliterals ist es nicht möglich einen Prototypen anzugeben, stattdessen muss die Methode `Object.create` oder eine Konstruktorfunktion benutzt werden.

2.1.4 First-Class Functions

Funktionen sind in ECMAScript lediglich besondere Objekte. Besonders dahingehend, dass sie als Prototypen das Standardobjekt `Function` haben. `Function` selbst hat als Prototypen das Objekt `Object`. Funktionen sind somit nichts weiteres als Objekte und können auch so behandelt werden. Funktionen können der Wert einer Variable oder einer Eigenschaft sein und Funktionen können von einer Variablen oder Eigenschaft in eine andere kopiert werden.

Funktionen können, ganz wie andere Objekte, selber über Eigenschaften und Methoden verfügen. Dabei sind Methoden nichts weiter als Funktionen, die in der Eigenschaft eines Objektes gespeichert sind. Das Schlüsselwort `this`, das in Methoden verfügbar ist, referenziert das Objekt dem die Eigenschaft gehört, in dem die Funktion/Methode gespeichert ist. Ist eine Funktion in einer einfachen Variablen gespeichert, so referenziert `this` stattdessen das globale Objekt[14].

14 Im Browser ist das globale Objekt das `window` Objekt. In anderen Laufzeitumgebungen kann dies ein anderes Objekt sein (z.B. das `process` Objekt in node.js). Es gibt Pläne dies zu ändern und stattdessen `this` außerhalb des Kontext eines Objektes undefined oder eine Exception werfen zu lassen.

```
// Definiere die Funktion func
function func () {
    return 23;
}

// Weise der Variablen f die Funktion func zu
var f = func;

if (f() == func()) {
    // f und func referenzieren die selbe Funktion
}

// Erzeuge ein Objekt mit der Eigenschaft method.
var o = {
    method: 42
};

// Weise der Eigenschaft method den Wert der Variablen f zu
o.method = f;

if (f() == o.method()) {
    // f und o.method referenzieren nun die selbe Funktion
}

// weise o.method eine neue Funktion zu
o.methode = function () {
    return this;
};

if (o.method() === o) {
    // this in o.method referenziert o
}

f = o.method;

if (f() === window) {
    // this in f referenziert das globale Objekt window
}

// Weise der Funktion func die Methode callMe zu
// welche this als Funktion aufruft.
func.callMe = function () {
    return this();
};

if (func() == func.callMe()) {
    // thisreferenziert in func.callMe und ruft somit
    // die Funktion func auf.
}
```

Quelltext 4: First-Class Functions

2.1.5 Anonymous Functions

Eine weitere Besonderheit von ECMAScript ist die Möglichkeit Funktionen ohne Namen zu definieren. Man nennt diese anonyme Funktionen oder zu Englisch anonymous Functions. In den zuvor gezeigten Quelltext-Beispielen wurden sie bereits mehrfach eingesetzt. Eine anonyme Funktion wird definiert, indem nach dem `function` Schlüsselwort der Bezeichner der Funktion ausgelassen wird. Anonyme Funktionen können wie andere Funktionen in Variablen oder Eigenschaften gespeichert werden und selbstverständlich ebenfalls aufgerufen werden.

Die Besonderheit von anonymen Funktionen ist, dass sie über keinen Bezeichner verfügen und somit auch ohne zuvor in einer Variable gespeichert worden zu sein aufgerufen oder als Parameter an eine weitere Funktion gegeben werden können.

```
// Eine anonyme Funktion wird erzeugt und in der Variablen f
// gespeichert. Sie erwartet als Parameter eine weitere
// Funktion, welche von ihr aufgerufen wird.
var f = function (func) {
    return func();
};

// die Funktion f wird aufgerufen und ihr wird als Parameter
// eine anonyme Funktion übergeben, welche den Wert 23 zurück
// liefert.
f(function () {
    return 23;
});

if (f(function () {
    return 42;
}) == 42) {
    // Erneut wird f aufgerufen und diesesmal eine Funktion übergeben
    // die den Wert 42 zurück liefert. f wird die übergebene Funktion
    // aufrufen und ihren Rückgabewert zurück geben.
}

if ((function () {
    return 23;
})() == 23) {
    // Eine anonyme Funktion wird definiert und direkt aufgerufen.
    // Sie liefert den Wert 23 zurück.
}

if ((function (value) {
    return value;
})(42) == 42) {
    // Eine anonyme Funktion die einen Parameter erwartet wird
    // definiert und sofort aufgerufen. Als Parameter wird ihr
    // der Wert 42 übergeben. Die Funktion liefert den übergebenen
    // Parameter zuück.
}
```

Quelltext 5: Anonymous Functions

2.2 Scope, Closures und Identifier Resolution

Der Gültigkeitsbereich für Variablen und Bezeichner,zu Englisch schlicht Scope, ist in ECMAScript im Gegensatz zu den meisten anderen Programmiersprachen mit C-ähnlicher Syntax nicht an Blöcke gebunden. ECMAScript hat stattdessen Function-Scope. Variablen und andere Bezeichner sind innerhalb der Funktion in der sie deklariert wurden gültig. Globale Variablen sind überall gültig. Wird eine Variable nicht am Anfang einer Funktion deklariert sondern erst später, so wird der ECMAScript Interpreter dafür sorgen, dass die Deklaration der Variable intern an den Anfang der Funktion verschoben wird. Dieses „anheben“ der Variablen wird zu Englisch auch hoisting genannt.

Jede Funktionsobjekt in ECMAScript verfügt intern über eine sogenannte Scope-Chain. Diese Scope-Chain enthält mindestens das globale Objekt. Wird eine Funktion aufgerufen, so wird ein sogenannter Execution Context für die Funktion erstellt, welcher über eine eigene Scope-Chain

verfügt. Diese enthält die selben Elemente wie die Scope-Chain des Funktionsobjektes. Im nächsten Schritt wird ein Activation Object erzeugt, dass alle lokalen Variablen, Objekte und Funktionen der aufgerufenen Funktion instanziert und anschließend an die erste Stelle der Scope-Chain des Execution Context gesetzt wird. Der Execution Context und damit die in dessen Activation Object enthaltenen lokalen Instanzen, kann auch gültig bleiben, wenn der Funktionsaufruf, durch welchen er erzeugt wurde, bereits beendet ist. Dies hat zur Folge, dass Funktionen, die von anderen Funktionen eingeschlossen sind, auch noch Zugriff auf Variablen der einschließenden Funktion haben, wenn die einschließende Funktion bereits verlassen wurde. Man nennt diesen Effekt Closure.[15]

15 Für eine ausführlichere Einführung in das Konzept von Scope-Chains in ECMAScript empfehle ich [Zakas, 2010], S. 16 – 26.

```
 1
 2 function scope () {
 3     {
 4         var i = 23;
 5     }
 6     // Obwohl i innerhalb eines eigenständigen Blocks definiert wurde
 7     // Kann auf den Wert innerhalb der gesamten Funktion zugegriffen
 8     // werden
 9     return i;
10 }
11
12 // Varibalendefinitionen ohne das var Schlüsselwort führen
13 // zu einer globalen Variable.
14 globalVar = 42;
15
16 function outer () {
17     var o = 23;
18     function inner() {
19         // inner hat Zugriff auf die Variablen von outer
20         var i = o;
21     }
22     // i ist außerhalb von inner nicht mehr gültig
23     return i != 23;
24 }
25
26 function createFunc() {
27     var c = 42;
28     // createFunc liefert eine anonyme Funktion als Rückgabewert
29     return function () {
30         // die anonyme Funktion hat Zugriff auf die Variablen von
31         // createFunc
32         return c;
33     }
34 }
35
36 // rufe createFunc auf und speichere die zurück gegebene
37 // Funktion in f
38 var f = createFunc();
39
40
41 if (f() == 42) {
42       // die von createFunczurückgegebene Funktion hat noch immer
43       // Zugriff auf die Variablen von createFunc, obwohl
44       // createFunc bereits verlassen wurde.
45 }
```

Quelltext 6: Scope, Closures und Identifier Resolution

Trifft der ECMAScript Interpreter auf einen Bezeichner, so wird versucht diesen Bezeichner aufzulösen und den Wert des Bezeichners zu bestimmen. Dazu wird zunächst der lokale, aktuell gültige Scope nach dem Bezeichner durchsucht. Wird kein entsprechender Wert gefunden, so wird die Referenz auf den vorhergehenden Scope genutzt um den umschließenden Scope nach dem Bezeichner zu durchsuchen. Dies wird wiederholt bis der Bezeichner gefunden wurde oder aber der globale Scope erreicht und der Bezeichner nicht aufgelöst werden konnte.

2.3 Performance-Tests

Um Anwendungen mit hohen Performance-Ansprüchen zu implementieren ist es wichtig die Stärken und die Schwächen der ECMAScript Sprache zu kennen. Dazu werden im Folgenden verschiedene gängige Methoden und Konzepte der ECMAScript Programmierung wie das Iterieren mithilfe von Schleifen, das Aufrufen von Funktionen oder der Zugriff auf Variablen vorgestellt und untersucht. Anschließend werden die Ergebnisse interpretiert und eine Handlungsempfehlung für die Leistungsoptimierung ausgesprochen.

Grundlegend sollte aber beachtet werden, dass Performance-Optimierungen immer eine Abwägung mit anderen Faktoren sind. Performance-Optimierungen gehen oft auf Kosten der Lesbarkeit des Quelltextes und erfordern immer Tests und somit Zeit der Entwickler. Außerdem sollten Performance-Optimierungen immer am Ende eines Entwicklungszyklus stehen. Bereits implementierte Optimierungen können sehr schnell durch Änderungen am Quelltext ihre Wirkung verlieren oder sich gar negativ auswirken. Nicht zuletzt auch deswegen sollte man von Optimierungen „ins Blaue hinein“ absehen. Erst wenn tatsächlich Performance-Probleme auftreten sollte von verschiedenen Optimierungsmöglichkeiten Gebrauch gemacht werden.

2.3.1 In- und Dekrementierung

Eine der häufigsten Operationen, vor allem in stark beanspruchten Quelltextsequenzen, sogenannten „Hotspots“, ist das in- und dekrementieren von Zählern. ECMAScript bietet dazu insgesamt 4 verschiedene spezialisierte Operatoren zur Verfügung:

- Post-Increment: `i++`
- Post Decrement: `i--`
- Pre-Increment: `++i`
- Pre-Decremet: `--i`

Außerdem ist es natürlich möglich eine Variable durch bloße Add- und Subtraktion von 1 zu in- und dekrementieren:

- Addition: `i += 1`
- Subtraktion: `i -= 1`

Für jede Variante wurde ein separater Testfall angelegt. Es wird keine Initialisierung außerhalb der einzelnen Testfälle für die Testreihe benötigt. Es folgt der Quelltext für die einzelnen Testfälle.

2.3.1.1 Ergebnisse

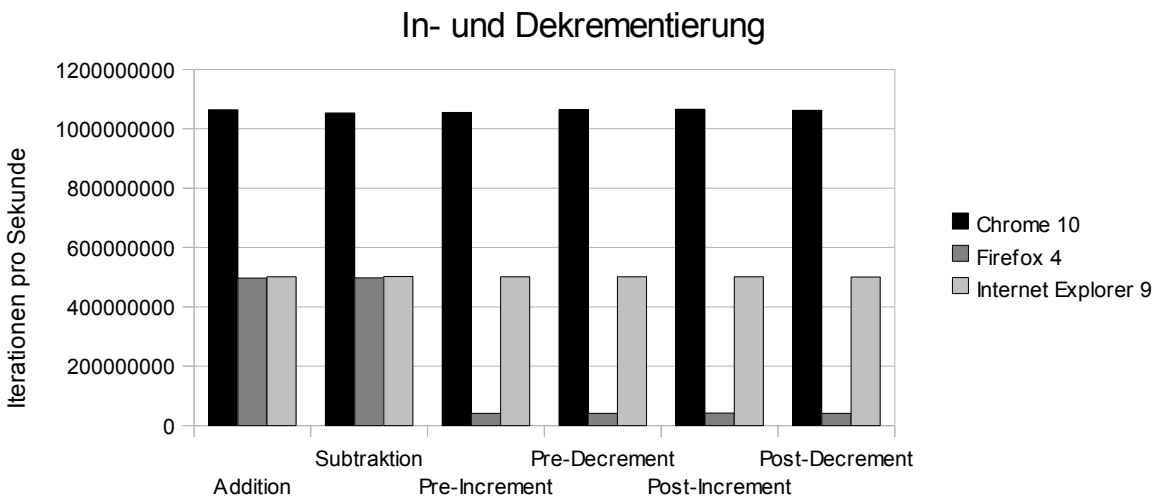


Abbildung 1: In- und Dekrementierung

Testfall	Chrome 10	Firefox 4	Internet Explorer 9
Addition	1.063.526.207	496.558.708	501.044.256
Subtraktion	1.053.522.692	497.722.234	502.069.368
Pre-Increment	1.055.522.282	41.204.862	501.514.022
Pre-Decrement	1.064.826.916	41.137.108	500.920.804
Post-Increment	1.065.619.532	41.634.326	501.150.221
Post-Decrement	1.062.306.177	41.186.922	500.180.322

Tabelle 1: In- und Dekrementierung

Chrome ist den beiden anderen Browsern eindeutig voraus und kann über eine Milliarde Durchläufe pro Sekunde abarbeiten. In den meisten Fällen gibt es keinen signifikanten Unterschied zwischen den verschiedenen Operatoren. Eine Ausnahme jedoch stellen die Increment und Decrement Operatoren im Firefox dar, welche entgegen jeder Erwartung im Vergleich zu einer einfachen Addition oder Subtraktion deutlich langsamer arbeiten. Es empfiehlt sich somit auf die Increment und Decrement Operatoren zu verzichten und stattdessen mithilfe einer einfach Addition oder Subtraktion zu in- bzw. zu dekrementieren.

2.3.2 Schleifen

Schleifen bilden das Rückgrat einer jeden Anwendung. Besonders in Anwendungen in denen viele Objekte verwaltet werden, die sich schnell und oft ändern, wie das in Spielen der Fall ist, ist das performante Abarbeiten von Listen und Arrays in Schleifen von besonderer Bedeutung.

ECMAScript bietet wie die meisten modernen Programmiersprachen drei verschiedene Schleifen.

Eine kopfgesteuerte while-Schleife, eine fußgesteuerte while-Schleife und eine for-Schleife.

Alle drei Schleifen können steigend und fallend durchlaufen werden.

Für alle 6 Varianten wurde ein Testfall angelegt. Die Testreihe wird mit einer konstanten Variable c initialisiert, die die Anzahl der Iterationen für die einzelnen Testfälle auf 10.000 festlegt.

2.3.2.1 Ergebnisse

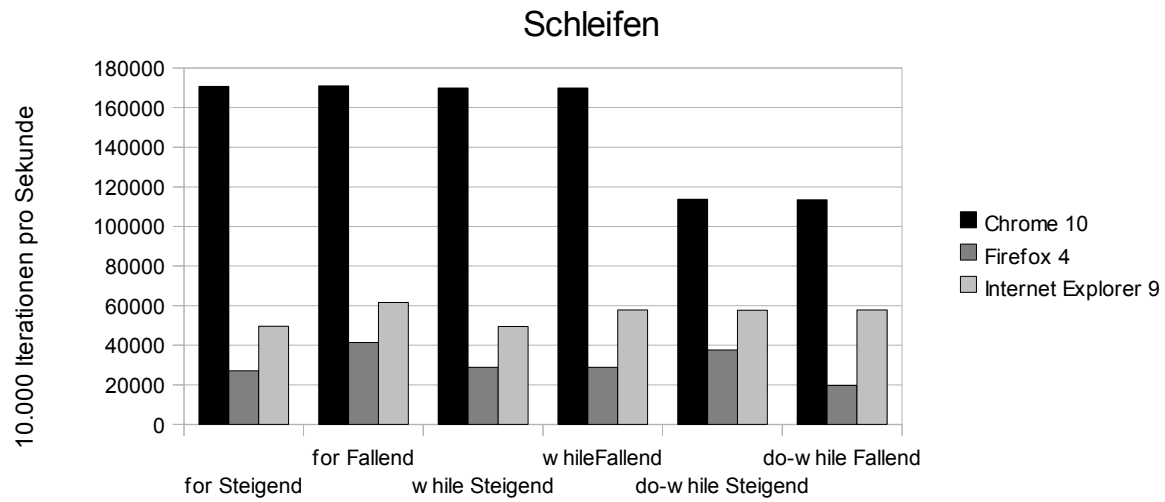


Abbildung 2: Schleifen

Testfall	Chrome 10	Firefox 4	Internet Explorer 9
for Steigend	170.691	27.178	49.556
for Fallend	171.029	41.431	61.570
while Steigend	169.877	28.894	49.481
whileFallend	169.877	28.894	57.791
do-while Steigend	113.773	37.591	57.666
do-while Fallend	113.518	19.696	57.860

Tabelle 2: Schleifen

Zunächst sei festgehalten, dass alle getesteten Browser mehrere hundertmillionen Iterationen pro Sekunde schaffen und damit äußerst performant ablaufen. Auffallend ist wieder, dass Chrome die anderen Browser in den Schatten stellt, dafür aber eine etwas schlechtere Perfomance bei der fußgesteuerten while-Schleife an den Tag legt. Die fallende for-Schleife scheint im Schnitt das beste Ergebnis zu liefern. Alles in allem Unterscheidet sich die Performance der verschiedenen Schleifen aber nicht signifikant. Wo es möglich ist und die Lesbarkeit des Quelltextes nicht signifikant beeinträchtigt sollte auf die fallende for-Schleife zurückgegriffen werden. Ansonsten ist der Einsatz jedes anderen Schleifentyps in den meisten Fällen ebenfalls vertretbar.

Sollten jedoch besonders hohe Ansprüche an die Schleife gestellt werden, so kann die Ausführungszeit von Schleifen durch das Ausrollen der Schleife weiter gesteigert werden. Dabei werden die in der Schleife ausgeführten Routinen je Iteration mehrfach ausgeführt um somit die Gesamtzahl der Iterationen zu verringern. Man spricht dabei auch von Loop Unfolding oder Loop Unrolling.

```
 1 var i;
 2 // Reguläre Schleife
 3 for (i = 0; i < 10000; i += 1) {
 4     func(i);
 5 }
 6
 7 // Ausgerollte Schleife
 8 for (i = 0; i < 10000; i += 5) {
 9     func(i);
10     func(i+1);
11     func(i+2);
12     func(i+3);
13     func(i+4);
14 }
```

Quelltext 7: Loop Unfolding

Diese Technik reduziert die Anzahl der Iterationen und somit den Aufwand, der durch die Iterationen entsteht. Der Effekt kann signifikant sein[16], vor allem, wenn der Schleifenkörper an sich klein ist und nur geringen Aufwand erfordert. Da diese Technik allerdings mit diversen Problemen[17] verbunden ist, rate ich grundsätzlich vom Einsatz dieser Technik ab, sofern die gewonnene Performance nicht absolut notwendig ist.

2.3.3 Scope Resolution

Das Auflösen von Bezeichnern entlang der Scope-Chain kann aufwendig sein, vor allem, wenn Bezeichner tief in der Scope-Chain sitzen oder die einzelnen Scopes viele Bezeichner enthalten. Da die allermeisten Operationen das Auflösen von Bezeichnern erfordern, ist es wichtig dieses so effizient wie möglich zu gestalten.

Grundsätzlich muss zwischen 3 verschiedenen Fällen unterschieden werden:

- Lokaler Bezeichner
- Globaler Bezeichner
- Andere Bezeichner

Bei der dritten Kategorie handelt es sich um eine starke Verallgemeinerung. Selbstverständlich können sich die Zahlen hier stark unterscheiden, abhängig davon wie tief ein Bezeichner in der Scope-Chain sitzt. Um einen Anhaltspunkt zu erhalten soll ein Bezeichner dienen, der um eine Ebene tieferals der loakel Scope in der Scope-Chain sitzt.

Für diese Testreihe haben wir also drei verschiedene Fälle. Da wir lediglich die Auflösungsperformance messen möchten wird der Quelltext zum erzeugen des Closures und der Variablen in allen drei Testfällen wiederholt, auch wenn er dort nicht zum ausführen des jeweiligen Testfalls notwendig wäre. Dadurch wird erreicht, dass die Messungen nicht durch zusätzliche Funktionsaufrufe oder durch das Erzeugen von Variablen verfälscht werden. Wir erzeugen die

16 Einfache Tests haben eine bis zu 10-fach erhöhte Performance für ein Unfolding von 5 Iterationen gezeigt. Siehe dazu auch den Testfall http://jsperf.com/loops-optimized

17 Ist die Anzahl der ausgerollten Schritte kein Divisor der Gesamtzahl der Iterationen, so entsteht eine Restmenge, die separat behandelt werden muss. Dies erfordert eine zusätzliche Schleife mit zusätzlichem Aufwand und macht den Quelltext noch unübersichtlicher. Ansätz dieses Problem effizienter zu lösen verringern die Lesbarkeit des Quelltextes nur noch weiter; siehe dazu auch [Duff, 1988]

Funktionen für die Testfälle im Initialisierungsblock um einen stärkeren Effekt durch die Unterschiedlichen Scope Auflösungen zu erzielen und das Ergebnis nicht mit dem Erzeugen der Funktionen in den einzelnen Testfällen zu verwässern.

2.3.3.1 Ergebnisse

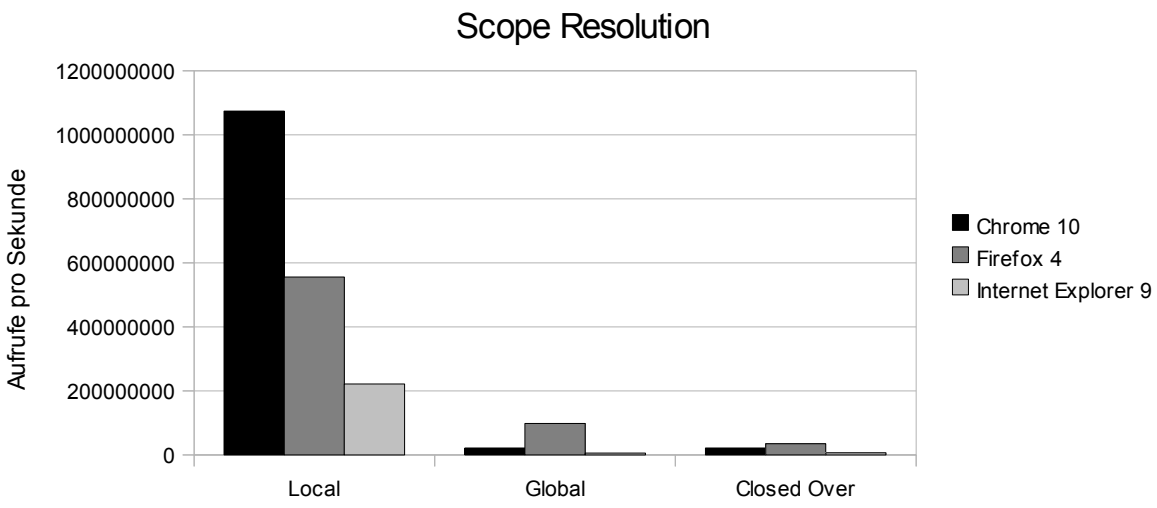


Abbildung 3: Scope Resolution

Testfall	Chrome 10	Firefox 4	Internet Explorer 9
Local	1.074.039.487	555.309.773	221.925.216
Global	21.911.367	98.300.793	6.318.738
Closed Over	21.938.021	35.315.517	6.518.895

Tabelle 3: Scope Resolution

Alle drei Browser zeigen wie erwartet eine deutlich höhere Performance für die Auflösung lokaler Bezeichner. Verwunderlich hingegen ist die geringe Performance von Closures im Vergleich zu globalen Variablen im Firefox Browser.[18]

Aufgrund der überragenden Performance lokaler Variablen empfiehlt es sich, wenn eine nicht-lokale Variable mehr als einmal innerhalb eines Scopes genutzt wird, eine lokale Kopie der Variablen anzulegen. Dies ermöglicht es die Variable mit der Performance des lokalen Scope aufzulösen und erhöht somit die Gesamtperformance.

2.3.4 Prototype Resolution

Moderne objektorientierte Anwendungen werden in ECMAScript mithilfe von Prototypen entwickelt. Um Eigenschaften und Methoden eines Objektes zu bestimmen ist es dabei notwendig stets auch die Prototypen-Kette des Objektes zu durchlaufen. Wie sich diese Operationen auf die Laufzeit auswirken kann von entscheidender Bedeutung für viele Design und

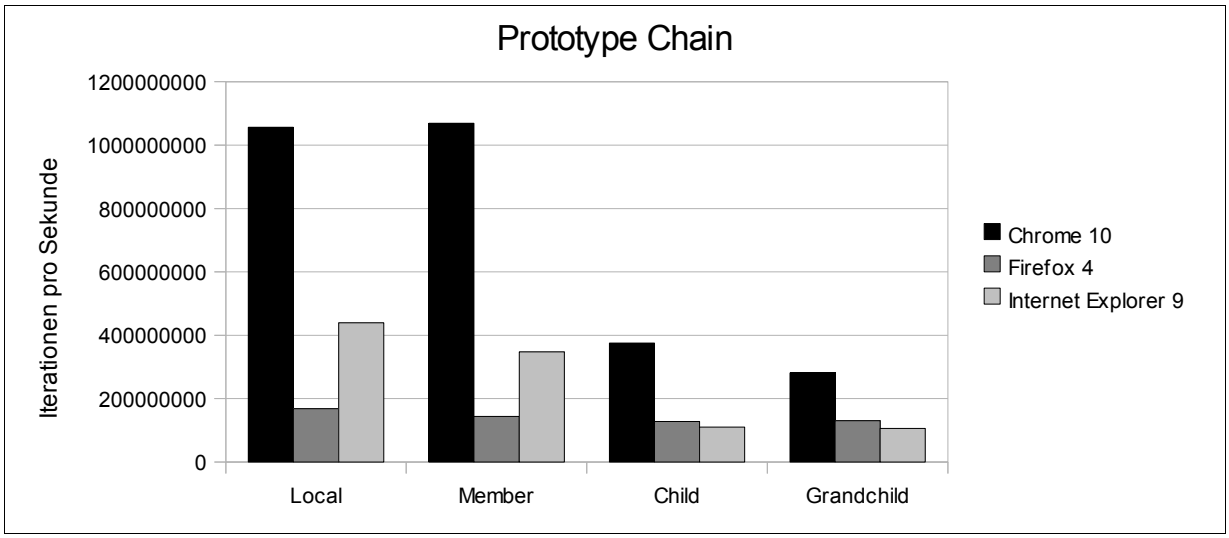
18 Verschiedene Versuche mit anderen Testfällen und Testreihen reproduzierten dieses Verhalten kontinuierlich. Es konnte auch keine Quelle gefunden werden, die dieses Phänomen erklärt. Es empfiehlt sich dieses Paradox zu einem späteren Zeitpunkt genauer zu untersuchen.

Implementierungsentscheidungen sein.

Die folgende Testreihe vergleicht den Zugriff auf eine einfache, lokale Variable und Funktion, den Zugriff auf eine eigene Eigenschaft und Methode eines Objektes, den Zugriff eine Eigenschaft und Methode eines direkten Prototypen und den Zugriff auf eine Eigenschaft und Methode eines indirekten Prototypen.

Im Initialisierungsblock werden alle Objekte und Variablen erstellt und vorbereitet.

2.3.4.1 Ergebnisse



Testfall	Chrome 10	Firefox 4	Internet Explorer 9
Local	1.056.425.422	168.425.831	439.291.079
Member	1.069.040.049	143.683.713	347.295.796
Child	375.682.599	127.979.496	110.091.473
Grandchild	280.672.652	130.520.751	106.121.910

Tabelle 4: Prototype Resolution

Lokale Variablen scheinen einen leichten Vorsprung zu Objekt Eigenschaften zu haben. Die Werte im Chrome Browser schwankten leicht. Mal schienen die Objekt Eigenschaften und mal die lokalen Variablen leicht vorne zu liegen, woraus geschlossen wird, dass sie in etwa gleichauf liegen.

Der erste Schritt in die Prototype Chain scheint sehr aufwendig zu sein. Alle Zugriffe auf die Prototype Chain sind deutlich langsamer als der Zugriff auf objekteigene Eigenschaften. Weitere Schritte auf der Prototype Chain scheinen nicht mehr ganz so teuer zu sein, lassen sich aber dennoch deutlich in den Zahlen ablesen.

Der erste Schritt auf der Prototype Chain ist teuer und sollte darum wohl überlegt sein. Ansonsten sind weitere Schritte auf der Prototype Chain in Maßen vertretbar. Unnötig tiefe Vererbungshierarchien sollten aber aus Performancegründen vermieden werden.

2.3.5 Funktionsaufrufe

Eine der grundlegendsten Konzepte in Programmiersprachen sind Unterprogramme bzw. Funktionen. Gerade in ECMAScript mit seinen First-Class Funktionen und funktionalen Eigenschaften sind Funktionen überaus wichtig. Darum ist es essentiell zu verstehen wie die verschiedenen Muster zum erzeugen und aufrufen von Funktionen gegeneinander in ihrer Performance aufwiegen.

Es stehen grundsätzlich 4 verschiedene Methoden zur Verfügung einen Funktion aufzurufen. Zunächst gibt es die Möglichkeit den Quelltext gar nicht erst in eine Funktion zu kapseln sondern jedes mal auszuschreiben. Man spricht dabei im Englischen auch von „inlining. Weiter besteht die Möglichkeit eine Funktion in einer Variablen oder aber in einer Objekt Eigenschaft zu speichern. Die letzte Möglichkeit zum Funktionsaufruf ist es eine anonyme Funktion direkt aufzurufen bzw. als Parameter an eine weitere Funktion zu übergeben, wie es für Rückruffunktionen äußerst beliebt ist.

Im Initialisierungsblock erzeugen wir ein Objekt und eine Funktion. Die Funktion dient zusätzlich als Methode des Objektes. Diese benötigen wir für zwei der Testfälle.

2.3.5.1 Ergebnisse

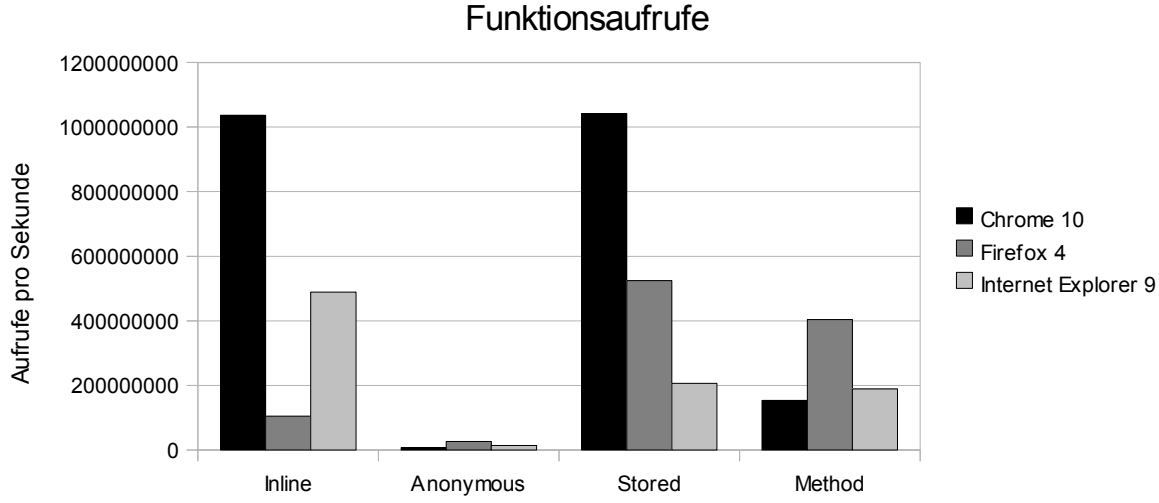


Abbildung 4: Funktionsaufrufe

Testfall	Chrome 10	Firefox 4	Internet Explorer 9
Inline	1.036.795.773	104.730.998	488.615.016
Anonymous	7.468.733	25.787.780	13.900.518
Stored	1.041.354.203	524.233.104	206.010.556
Method	153.370.351	403.485.909	188.881.493

Tabelle 5: Funktionsaufrufe

Hier sind einige Interessante Beobachtungen zu machen. Zunächst einmal ist sehr eindeutig zu sehen, dass alle Browser einen enormen Performanceverlust bei anonymen Funktionen zu verzeichnen haben. Das ist nicht weiter verwunderlich, denn das erzeugen einer anonymen Funktion

ist aufwendig und erzeugt eine signifikanten Mehrlast.

Weiter lässt sich am Beispiel des Firefox Browser sehr schön der Effekt von JIT Compilern erkennen. Der Inline Testfall läuft im Firefox besonders langsam, da er den JIT Compiler der ECMAScript Engine nicht ausgelöst wird. Die in Variablen gespeicherten Funktionen hingegen werden deutlich effizienter ausgeführt, da sie vom JIT Compiler richtigerweise als Hotspots erkannt werden können.

Um den Vorteil der JIT Compiler voll zu nutzen empfiehlt es sich auch bzw. gerade viel genutzte Segmente in Funktionen auszulagern. Vorsicht jedoch sei bei Methoden geboten. Sie sind deutlich weniger performant als reguläre Funktionen. Auf den Einsatz von anonymen Funktionen sollte man innerhalb kritischer Stellen am besten ganz verzichten. Besser ist es die Funktion an einer weniger beanspruchten Stelle zu erzeugen und vorzuhalten bis die kritische Stelle erreicht wird welche die Funktion benötigt – sofern dies möglich ist.

2.3.6 Funktionsparameter

Ebenso wichtig wie Funktionen sind Parameter, die den Funktionen übergeben werden. Die folgende Testreihe untersucht inwiefern sich Parameter auf die Performance von Funktionsaufrufen auswirken. Gemessen werden Funktionsaufrufe ohne Parameter, mit einem , mit zwei und mit drei Parametern. Zusätzlich wird der Funktionsaufruf mithilfe eines Parameterobjektes untersucht. Letzteres ist eine beliebte Methode von in ECMAScript implementierten Bibliotheken einfach optionale Parameter und erweiterbare Parameterlisten mit Abwärtskompatibilität zu implementieren.

Im Initialisierungsblock werden die aufzurufenden Funktionen definiert. Die fünf Testfälle rufen diese Funktionen mit unterschiedlichen Parameterzahlen auf.

2.3.6.1 Ergebnisse

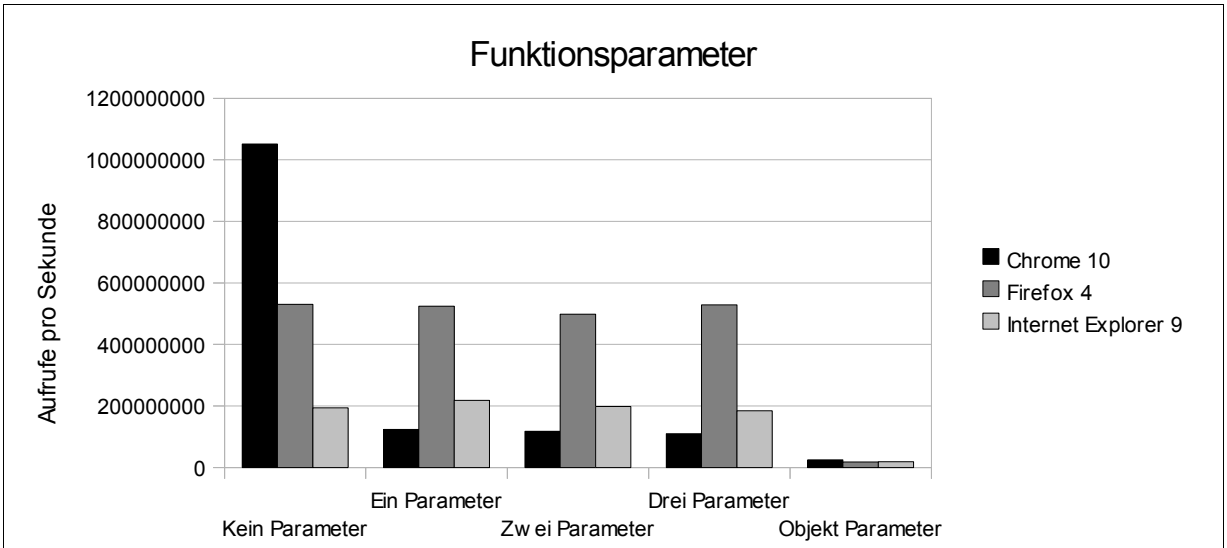


Abbildung 5: Funktionsparameter

Testfall	Chrome 10	Firefox 4	Internet Explorer 9
Kein Parameter	1.051.550.648	529.876.570	193.386.647
Ein Parameter	123.908.206	524.292.452	218.268.970
Zwei Parameter	117.067.960	498.507.421	198.485.184
Drei Parameter	109.319.663	528.302.493	184.027.657
Objekt Parameter	24.280.495	17.886.637	18.734.790

Tabelle 6: Funktionsparameter

Die Anzahl der Parameter scheint keinen nennenswerten Einfluss auf die Performance von Funktionsaufrufen zu haben. Einzig Chrome scheint Aufrufe ohne Parameter besonders zu optimieren und kann diese mit bis zu 10-facher Performance aufrufen. Der Einsatz von Parameterobjekten wirkt sich, wie die Zahlen sehr deutlich zeigen, äußerst negativ auf die Performance aus. Dies zeigt erneut, wie aufwendig das Erzeugen von Objekten im Vergleich zu anderen, primitiveren Operationen ist.

Die Anzahl der Parameter ist für das Ausführen von Funktionen also im Grunde unerheblich. Man sollte allerdings darauf verzichten Objekte im Rahmen eines Funktionsaufrufes zu erzeugen oder häufig benötigte Parameterobjekte vorhalten und wieder verwenden.

2.3.7 Funktionsrückgabewerte

Viele Funktionen liefern ein Ergebnis zurück. Die folgende Testreihe untersucht wie sich verschiedene Typen von Rückgabewerten auf die Performance eines Funktionsaufrufes auswirken. Dazu wird zwischen keinem `return` Statement, einem leeren `return` Statement, einem `return` Statement mit primitivem Datentyp und einem `return` Statement mit einem Objekt unterschieden.

Im Initialisierungsblock werden die Funktionen mit den unterschiedlichen Rückgabewerten erzeugt.

2.3.7.1 Ergebnisse

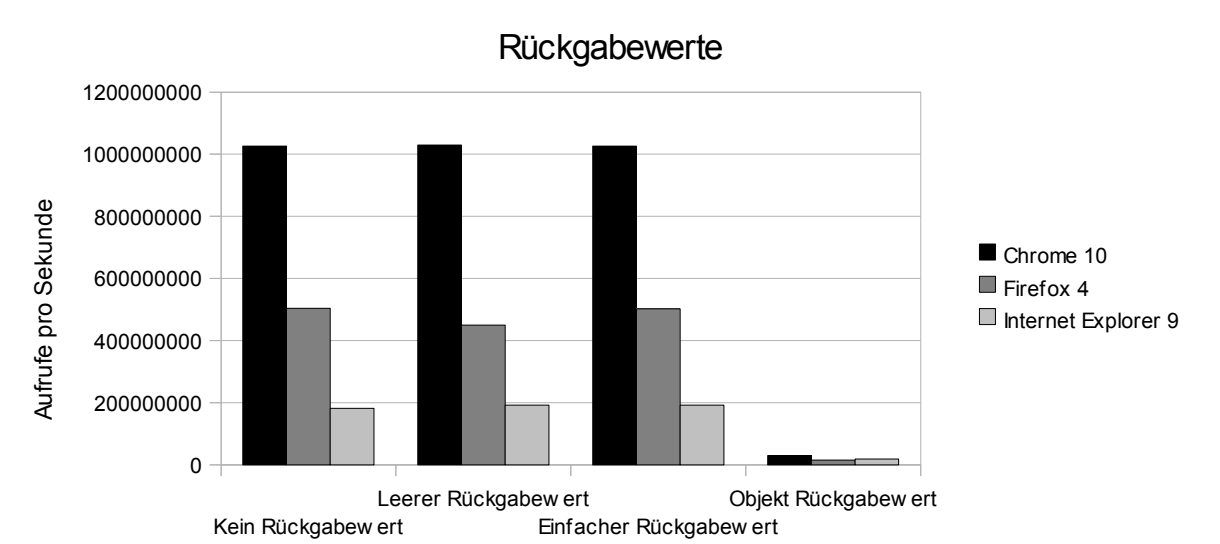


Abbildung 6: Rückgabewerte

Testfall	Chrome 10	Firefox 4	Internet Explorer 9
Kein Rückgabewert	1025896807	504196724	181587073
Leerer Rückgabewert	1029677169	449655037	192211039
Einfacher Rückgabewert	1026346396	502000082	191970906
Objekt Rückgabewert	29679983	14661463	18742955

Tabelle 7: Rückgabewerte

Für die meisten Typen von Rückgabewerten unterscheidet sich die Performance nicht nennenswert. Es ist jedoch erneut zu sehen, dass sich das Erzeugen eines Objektes signifikant auf die Performance auswirkt. Es wird darum dringend davon abgeraten Objekte in viel aufgerufenen Funktionen zu erzeugen.

2.3.8 Member Resolution

Der Zugriff auf die Eigenschaften eines Objektes funktioniert im Grunde ähnlich wie das Auflösen eines Bezeichners in der Scope-Chain oder das finden einer Objekt Eigenschaft in der Protoype-Chain, wurde selbst bisher aber noch nicht genauer beleuchtet.

In der folgenden Testreihe werden die Performance für das auflösen von Eigenschaften in Objekten untersucht. Verglichen werden der Zugriff auf eine reguläre Variable, der Zugriff auf eine Objekt Eigenschaft und der Zugriff auf eine Eigenschaft eines Objektes, das selbst die Eigenschaft eines weiteren Objektes ist. Da sich bereits bei der Scope Resolution gezeigt hat, dass viel Performance gewonnen werden kann, indem man einen einmal aufgelösten nicht-lokalen Bezeichner in einem lokalen Cache vorhält, werden wir diese Testreihe gleich um einen solchen Fall erweitern.

Der Initialisierungsblock wird die Variablen und verschachtelten Objekte erzeugen, die für die Testreihe notwendig sind.

2.3.8.1 Ergenisse

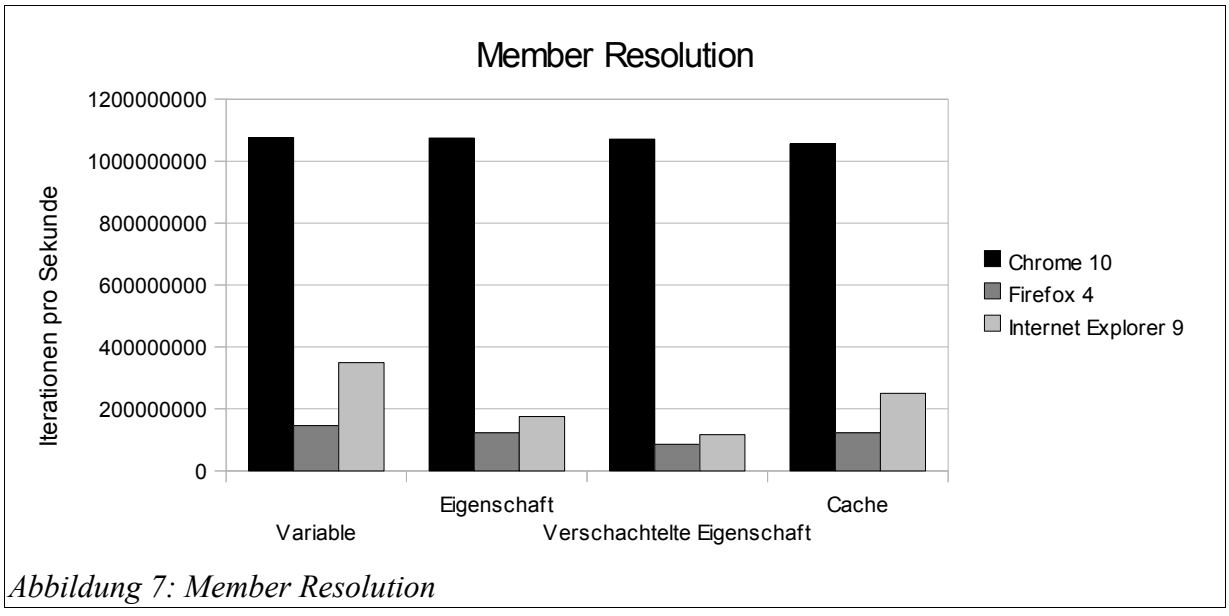


Abbildung 7: Member Resolution

Testfall	Chrome 10	Firefox 4	Internet Explorer 9
Variable	1.075.521.207	146.071.263	349.527.061
Eigenschaft	1.074.400.504	122.707.821	175.261.157
Verschachtelte Eigenschaft	1.071.304.439	85.302.422	116.929.964
Cache	1.055.352.257	122.987.495	250.597.825

Tabelle 8: Member Resolution

Wie zu erwarten, sinkt die Performance mit zunehmender Verschachtelung. Nur Chrome ist weitestgehend unbeeindruckt und kann in allen Fällen eine äußerst hohe Anzahl an Iterationen erreichen. Besonders interessant allerdings ist, dass der Testfall mit Cache in den langsameren Browsern einen enormen Performancegewinn erzielen kann und sogar den einfachen Testfall ohne Vorschachtelung übertrumpft.

Es ist also in jedem Fall zu empfehlen einmal aufgelöste Eigenschaften eines Objektes in einer lokalen Variable vorzuhalten, sollte die selbe Eigenschaft mehrfach benötigt werden. Dies kann sich signifikant auf die Performance auswirken, vor allem, wenn stark verschachtelte Objekte im Einsatz sind. Zu beachten ist allerdings, dass das kopieren einer Methode in eine lokale Variable den Wert des `this` Schlüsselwortes für die Funktion ändert und somit nicht zu empfehlen ist!

2.4 Zusammenfassung

ECMAScript ist eine Leistungsstarke dynamische Sprache. Vor allem der Chrome Browser und dessen V8 Engine haben in vielen Bereichen neue Maßstäbe gesetzt. Zwar können die anderen Browserhersteller durch die letzte Browsergeneration inzwischen in vielen Bereichen nachziehen, doch der Chrome Browser hat insgesamt noch immer das bessere Leistungsbild.

Im Grunde sind zwei Aspekte während der Untersuchungen immer wieder zum Vorschein getreten, welche die Laufzeit einer Anwendung stark beeinflussen. Den ersten Aspekt haben wohl die meisten Programmiersprachen gemein: Das allozieren von Speicher. Das erzeugen von Objekten und Funktionen in ECMAScript ist eine teure Operation und sollte in stark beanspruchten Abschnitten des Programmablaufes vermieden werden Wie wir mehrfach gesehen haben kann es zu signifikanten Performanceeinbußen führen.

Der zweite Aspekt, der beachtet werden sollte, ist der Umstand, dass das auflösen von Bezeichnern, vor allem nicht-lokalen Bezeichnern in ECMAScript eine im Vergleich zu anderen Programmiersprachen aufwendige Operation ist. Anstelle einen nicht-lokalen Bezeichner mehrfach aufzulösen, sollte der Wert in einer lokalen Variable vorgehalten werden. Dies gilt sowohl für verschachtelte Objekte, als auch für Variablen außerhalb des lokalen Scope.

Die Zahlen der Testläufe, die beinahe alle in die Milliarden Iterationen pro Sekunde gehen, sprechen eine deutliche Sprache: ECMAScript bietet ein sehr hohes Maß an Peformance. Alle hier vorgestellten Optimierungsansätze sind grundsätzlich nicht notwendig um schnelle Anwendungen in ECMAScript zu entwickeln, sondern dienen lediglich dazu, dass letzte bisschen aus der Sprache „heraus zu kitzeln“.

3 HTML5 Canvas

Das Schlagwort HTML5 bezeichnet eine ganze Reihe verschiedener Web-Technologien. Viele von ihnen sind nicht Teil des offiziellen HTML5 Standards, werden aufgrund ihrer konzeptionellen

und zeitlichen Nähe allerdings oft zusammenfassend als „HTML5“ bezeichnet.

Viele der als HTML5 bezeichneten Technologien haben als Ziel den Browser mit mehr Multimedia Fähigkeiten zur Verfügung zu stellen und Webapplikationen mit Features auszurüsten, die bisher Desktopanwendungen vorbehalten waren. Zu diesen Technologien gehören CSS3[19], Web Sockets[20], Web Worker[21], Web Storage[22], Geolocation[23], Video / Audio und Canvas.

Viele dieser Technologien bieten besondere Vorteile auch für den Einsatz in Browserspielen. Im Rahmen dieser Arbeit möchte ich mich aber auf den Einsatz von Canvas für Browserspiele konzentrieren und die anderen Technologien nicht betrachten.

Es folgt eine kurze Einführung in das HTML5 Canvas Element und dessen API.[24]

3.1 Eigenschaften

HTML5 Canvas stellt, wie der Name vielleicht bereits suggeriert, eine Bitmap Zeichenfläche dar. Mithilfe einer umfangreichen API lassen sich beliebige Inhalte auf die Fläche zeichnen. Die API bietet dabei grundlegende Operationen, die sich kombinieren lassen um komplexere Zusammenstellungen zu erzeugen. Die API lässt sich in 4 logische Abschnitte unterteilen:

- die Image API bietet Operationen zum Zeichnen von kompletten Bildern auf das Canvas
- die Path API ermöglicht es primitive Objekte Linien, Rechtecke und Kreise zu zeichnen. Außerdem bietet sie die Möglichkeit quadratische und Bezier-Kurven zu zeichnen
- die Text API ermöglicht es Text zu zeichnen
- die Pixel API ermöglicht es einzelne Pixel auszulesen und zu setzen.

Alle Operationen auf die Canvas Fläche wirken kumulativ. Änderungen überzeichnen somit bereits gezeichnete Inhalte[25]. Möchte man beispielsweise die Position eines gezeichneten Objektes ändern, so muss die Zeichenfläche zunächst gelöscht und dann das Objekt an der angepassten Position neu gezeichnet werden. Dies ist ein gängiges Konzept in vielen Grafik APIs, wie OpenGL oder DirectX.

Klassische Desktop Spiele nutzen diese Eigenschaft und zeichnen die Bildfläche möglichst oft neu. Nach jeder Iteration aktualisiert das Spiel seinen internen Status der Spielwelt. Dazu gehören Beispielsweise die Position von Objekten und Gegnern, Animationen und das Interface. Nachdem

19 Cascading Style Sheets 3. CSS3 ist der Nachfolger von CSS 2.1 und dient als Sprache um HTML Elemente zu Layouten und ihre visuelle Repreäsentation zu variieren. CSS3 bietet Features wie Transformationen (Rotationen, Spiegelungen, Verzerrungen), Schatten und Animationen. Die meisten Browser greifen zur Darstellung dieser Effekt auf eventuell vorhandene dedizierte Grafikhardware zurück.

20 Web Sockets bieten Anwendungen die Möglichkeit Netzwerkverbindungen auch ohne den Overhead von HTTP zu erstellen. Außerdem eignen sie sich aufgrund der bidirektionalen, persistenten Verbindung hervorragend für Server-Push und somit als Comet Ersatz.

21 Web Worker bieten Anwendungen eine Möglichkeit der ECMAScript Eventloop zu entfliehen und aufwendige Operationen und Berechnungen in einen eigenen Thread, den Web Workern, auszulagern.

22 Web Storage bezeichnet die Möglichkeit Daten in einer lokalen, im Browser integrierten Datenbank zu speichern, was vor allem für Offline Webanwendungen äußerst interessant ist.

23 Geolocatino ist eine API, die es ermöglicht die Position eines Endgerätes z.B. via GPS festzustellen. Dies ist vor allem auf mobilen Endgeräten wie Smartphones oder Tablet PCs eine interessante Neuerung.

24 Leider gibt es aktuell noch keine dedizierte Literatur zum Canvas Element. Details zum Canvas Element und dessen API lassen sich aber hervorragend aus der technischen Spezifikation für HTML5 entnehmen: [WHATWG, 2011]. Vielversprechend scheint das im Mai beim O'Reilly Verlag erscheinende Buch „HTML5 Canvas“ von Jeff Fulton, ISBN: 1-4493-9390-X

25 Dieser Standard Operationsmodus lässt sich über die Eigenschaft `globalCompositeOperation` ändern.

der interne Status des Spiels aktualisiert wurde, wird die Bildfläche, auf Basis des neuen Spielstatus, erneut gezeichnet. Dies sorgt, abhängig von der Leistung des ausführenden Computers, für einen möglichst flüssigen Ablauf des Spiels. Man nennt dieses iterative Konzept auch Game-Loop[26].

Da das komplette Neuzeichnen einer Szene aufwendig ist, haben sich verschiedene Optimierungstechniken etabliert, die es ermöglichen den Aufwand zum aktualisieren der Bildfläche zu reduzieren. Ein paar Grundlegende möchte ich hier vorstellen um sie später auf ihre Effizienz zu untersuchen.

3.1.1 Clipping

Clipping bezeichnet eine Technik die es erlaubt den Zeichenbereich einzuschränken und die in der Regel von der Grafik API zur Verfügung gestellt wird. Auch die Canvas API ermöglicht es Clipping einzusetzen. Clipping erlaubt es bestimmte Bereiche der Zeichenfläche für alle folgenden Zeichenoperationen zu deaktivieren. Alle Flächen welche außerhalb der sog. Clip-Region liegen werden durch folgende Zeichenoperationen nicht verändert.

Dies ermöglicht es Bereiche vor Veränderung zu schützen und reduziert gleichzeitig die Menge der tatsächlich veränderten Pixel, was in der Regel zu einer Performancesteigerung führt. Wenn wir das Beispiel des sich bewegenden Objektes erneut aufgreifen, so ist es möglich mithilfe von Clipping nur die Bereiche der Zeichenfläche neu zu zeichnen an denen sich das Objekt zuvor befand und an der es sich nun befindet. Alle weiteren Bereiche bleiben von der veränderten Position des Objektes unberührt.

Clipping wird für das Canvas Element von der Path API zur Verfügung gestellt. Alle mithilfe der Path API konstruierbaren Objekte lassen sich auch als Clip-Region einsetzen indem anstelle der Zeichenoperationen `fill()` und `stroke()` die Operation `clip()` aufgerufen wird.

3.1.2 Culling

Culling ist eine Technik mit deren Hilfe nicht sichtbare Objekte bestimmt und in Folge nicht an die Grafik API zum rendern übergeben werden. In vielen Spielen bekommt ein Spieler zu einem gegebenen Zeitpunkt lediglich einen beschränkten Bereich der Spielwelt zu sehen. Alle Objekte, die sich außerhalb des Sichtfeldes des Spielers befinden müssen darum nicht gerendert werden.

Culling muss vom Programmierer einer Anwendung implementiert werden. Bevor ein Objekt mithilfe der durch die API zur Verfügung gestellten Operationen gezeichnet wird, muss geprüft werden, ob das Objekt tatsächlich sichtbar ist. Dies kann für eine besonders große Anzahl an Objekten durchaus aufwendig sein. Üblicherweise werden verschiedene Baumstrukturen eingesetzt um diese Überprüfungen zu beschleunigen. Anstelle alle Objekte einzeln zu prüfen genügt es nun zu überprüfen ob ein bestimmter Zweig des Baums sichtbar ist. Ist das nicht der Fall werden alle Objekte dieses Zweiges verworfen und nicht an die Grafik API zum rendern gegeben.

3.1.3 Buffering

Buffering bezeichnet die Möglichkeit eine bereits gezeichnete Szene zwischen zu speichern um sie den Puffer später zu nutzen um erneute Zeichenoperationen zu beschleunigen. Oft kann man Objekte in einer Szene in zwei Grundlegende Klassen unterteilen: statische und dynamische

26 [Barron, 2001], S. 40 ff

Objekte. Statische Objekte sind Objekte, die sich nicht verändern und in jedem gezeichnetem Frame an der selben Stelle erscheinen. Dies ist oft der Fall für Hintergründe und feste, unbewegliche Elemente der Spielwelt. Dynamische Objekte hingegen sind Objekte, die sich sehr oft verändern indem sie ihre Position oder andere optische Eigenschaften verändern.

Wird nun eine Szene neu gezeichnet so ist es meist notwendig auch die statischen Objekte neu zu zeichnen, da sie als Hintergrund für die dynamischen Objekte dienen. Gibt es jedoch viele statische Objekte und sind diese wohl möglich sogar aus komplexen Elementkompositionen zusammengesetzt, so kann das Zeichnen enorm aufwendig sein.

Da sich statische Objekte jedoch meist im Hintergrund der dynamischen Objekte befinden ist es möglich zunächst alle statischen Objekte zu zeichnen und noch bevor die dynamischen Objekte darauf gezeichnet werden, die Zeichenfläche, samt gezeichneter statischer Objekte, in einem Puffer zu sichern. Anstelle das nächste mal alle statischen Objekte neu zu zeichnen, kann nun einfach der Puffer auf die Zeichenfläche geschrieben werden. Dies kann, je nach Komplexität der statischen Objekte, zu enormem Einsparungen führen.

3.2 Performance-Tests

Moderne Web-Anwendungen wie Browsergames sind auf eine Leistungsfähige Grafikanbindung angewiesen. Vor allem wenn das Canvas Element in Konkurrenz zum bisherigen „Platzhirsch“ Flash treten soll, muss es die nötige Leistungsfähigkeit bieten.

In den folgenden Testreihen wird diese Leistungsfähigkeit des Canvas Elements untersucht. Es wird untersucht wie die verschiedenen APIs mit verschiedenen Datenmengen umgehen können, ob es Schwächen in den APIs und deren Implementationen in den Browsern gibt und ob und inwiefern die zuvor vorgestellten Optimierungstechniken die Performance beeinflussen können.

Besonders interessant dürfte es sein inwiefern sich die mangelnde Hardware-Beschleunigung des Chrome Browsers bemerkbar machen wird. In seiner ECMAScript Performance konnte der Browser überzeugen und seine Konkurrenten knapp hinter sich lassen. Wie wird sich die fehlende Hardwarebeschleunigung für den Browser auswirken?

3.2.1 Clearing

Eine der wichtigsten Operationen beim Zeichnen auf eine Zeichenfläche ist immer das löschen des bisherigen Inhaltes um aktualisierten Inhalt auf die Zeichenfläche zu bringen. Das Canvas Element bietet dazu zwei verschiedene Möglichkeiten:

- `clearRect(x, y, w, h)`
- Neu-Setzen der Canvas Dimensionen `width` und `height`

In der folgenden Testreihe wird untersucht welcher der beiden Möglichkeiten effizienter ist und inwiefern die Größe der Canvas Zeichenfläche dies beeinflusst. Insgesamt werden drei verschieden große Canvas Flächen untersucht, nämlich

- 800x600px
- 200x150px
- 1680x1050px

Der Initialisierungsblock enthält die drei Canvas Elemente und initialisiert Referenzen auf ihren

Zeichenkontext.

3.2.1.1 Ergebnisse

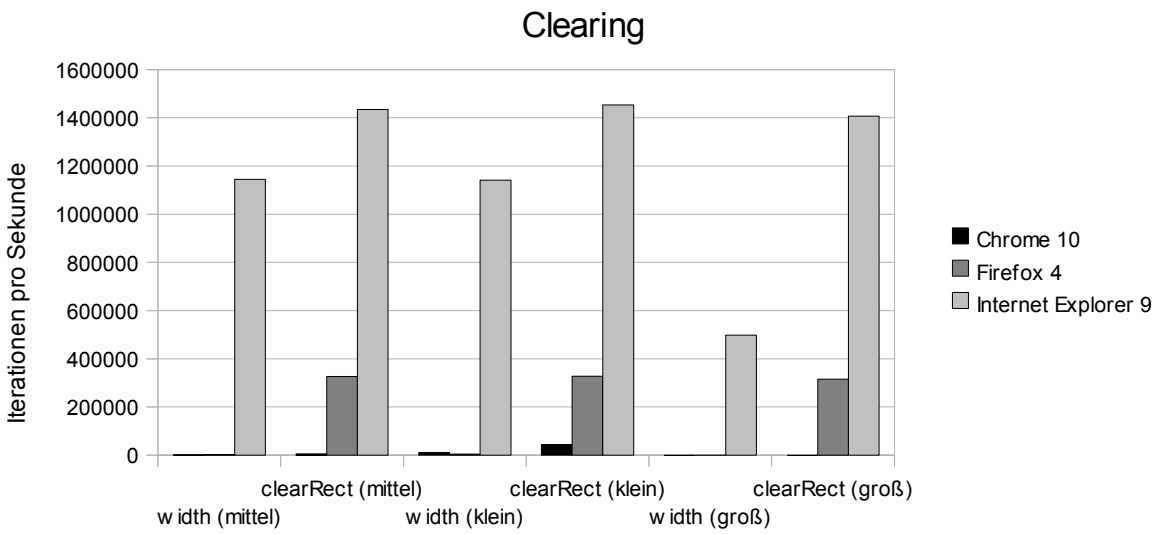


Abbildung 8: Clearing

Testfall	Chrome 10	Firefox 4	Internet Explorer 9
width (mittel)	868	1.745	1.144.139
clearRect (mittel)	5.728	327.009	1.434.419
width (klein)	11.474	4.783	1.141.463
clearRect (klein)	43.749	327.288	1.453.449
width (groß)	179	511	498.300
clearRect (groß)	468	315.158	1.406.754

Tabelle 9: Clearing

Der Chrome Browser kann hier nicht mithalten. Die anderen beiden Browser sind deutlich effizienter im Zeichenfläche Zurücksetzen. Außerdem zeigt sich deutlich, dass `clearRect` die effizientere Methode ist. Erstaunlich ist, dass, bis auf Chrome, alle Browser problemlos auch große Zeichenflächen löschen können.

Es empfiehlt sich in jedem Fall der Einsatz von clearRect zum löschen der Zeichenfläche.

3.2.2 Path API 1

Die Path API stellt die grundlegende Geometrische Zeichenoperationen zur Verfügung. Mithilfe dieser primitiven Geometrischen Figuren lassen sich komplexe Figuren und Objekte konstruieren. In der folgenden Testreihe wird untersucht wie performant sich aus einfachen Linien komplexere Objekte zeichnen lassen.

Es werden mithilfe von einfachen Linien Operationen folgende Objekte gezeichnet:

- Fallende Gerade
- Dreieck
- Rechteck
- (unregelmäßiges) Polygon

Nachträglich wurde noch ein weiterer Testfall hinzugefügt um eine mögliche Interpretation der Ergebnisse zu bestätigen:

- Horizontale Gerade

Im Initialisierungsblock wird das Canvas Element erstellt und eine Referenz auf den Zeichenkontext gesichert. Außerdem werden die Höhe und die Breite des Canvas Element in Variablen gespeichert.

3.2.2.1 Ergebnisse

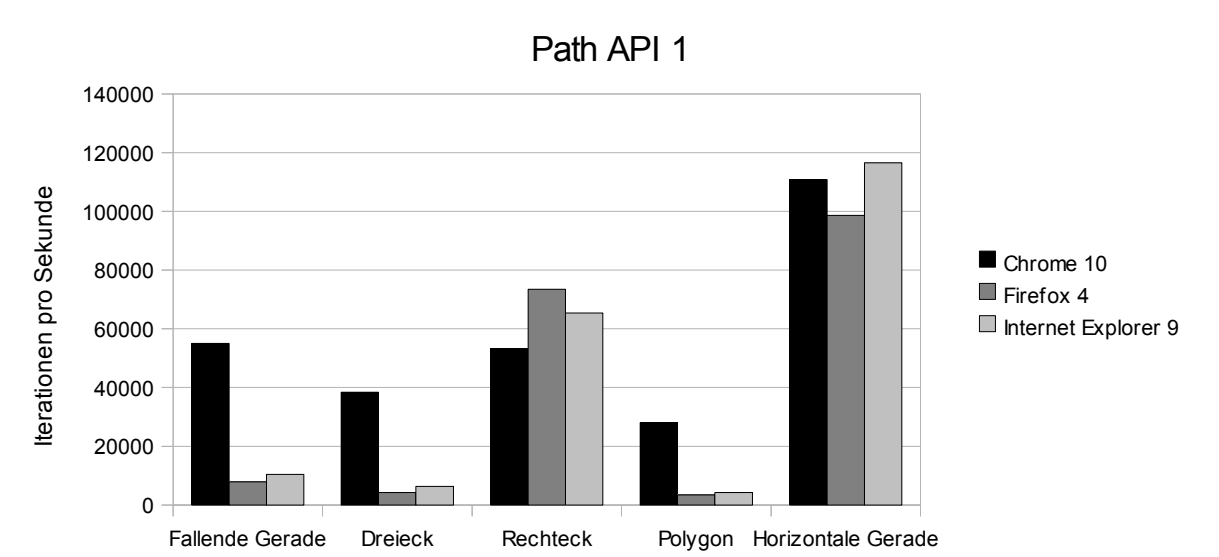


Abbildung 9: Path API 1

Testfall	Chrome 10	Firefox 4	Internet Explorer 9
Fallende Gerade	55.053	7.881	10.405
Dreieck	38.388	4.283	6.354
Rechteck	53.226	73.513	65.348
Polygon	28.085	3.455	4.198
Horizontale Gerade	110.810	98.643	116.565

Tabelle 10: Path API 1

Zunächst ist auffällig, dass trotz fehlender Hardwarebeschleunigung Chrome die anderen Browser im Schnitt klar abhängen kann. Besonders interessant ist aber vor allen Dingen, dass das Zeichnen des Rechteckes deutlich performanter als das Zeichnen auch von deutlich simpleren

Figuren von statten geht.

Da das Rechteck die einzige Figur ohne Schrägen war, lag die Vermutung nahe, die verringerte Performance bei den anderen Figuren könnte an den Schrägen liegen. Um Kanten und Schrägen weicher erscheinen zu lassen, wird eine Technik mit dem Namen Anti-Aliasing eingesetzt. Diese erfordert jedoch zusätzlichen Rechenaufwand und könnte die Ursache für die verringerte Performance sein. Dieser Verdacht wurde durch den Nachträglich hinzugefügten fünften Testfall bestätigt. Die horizontal verlaufende Gerade kann deutlich performanter gezeichnet werden als die schräg verlaufende Gerade, welche Anti-Aliasing erfordert.

Alles in allem bietet die Path API nur eine relativ schwache Leistung. Es empfiehlt sich auf die Path API nur mit Bedacht zurück zu greifen da zu viele Path Operationen die Anzahl der gezeichneten Bilder pro Sekunde schnell stark reduzieren können.

3.2.3 Path Size

Da sich die Anzahl der gezeichneten Kanten, wie die vorherige Testreihe gezeigt hat, stark nachteilig auf die Performance der Path API auswirken kann, stellt sich die Frage ob auch die Größe der gezeichneten Elemente negativ auf die Performance wirkt.

Dazu wird die folgen Testreihe drei Testfälle untersuchen. In jedem Testfall wird ein Dreieck gezeichnet. Die Dreiecke unterscheiden sich lediglich in ihrer Dimension.

Der Initialisierungsblock wird wieder das Canvas Element vorbereiten.

3.2.3.1 Ergebnisse

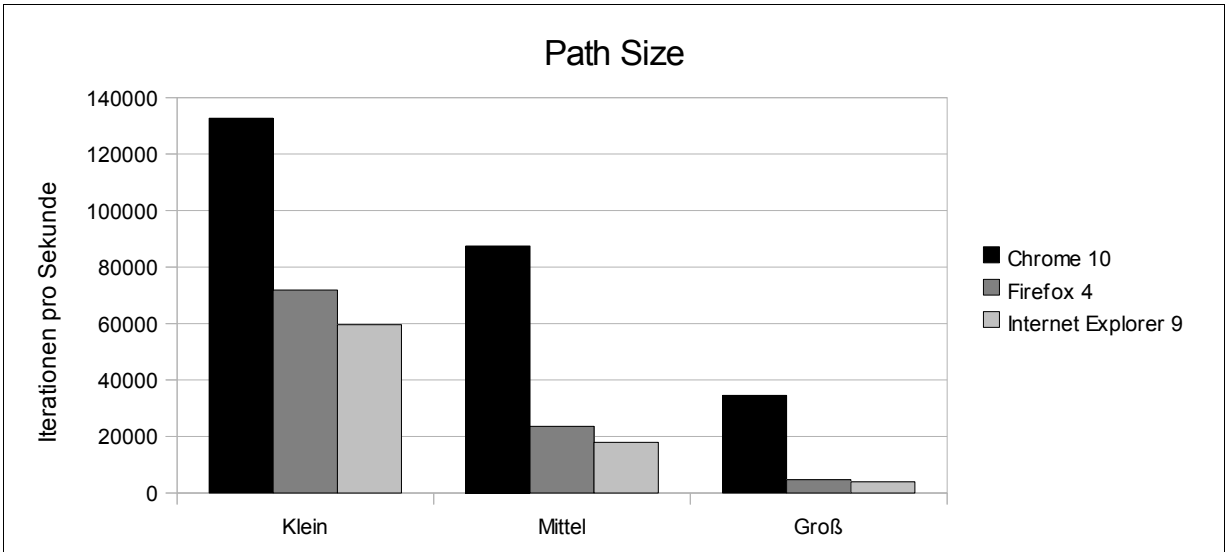


Abbildung 10: Path Size

Testfall	Chrome 10	Firefox 4	Internet Explorer 9
Klein	132.685	71.847	59.591
Mittel	87.443	23.613	17.960
Groß	34.540	4.713	3.948

Tabelle 11: Path Size

Eindeutig zu erkennen ist, dass die Performance für größere Objekte stetig abnimmt, auch wenn die Anzahl der Kanten konstant bleibt. Erneut kann Chrome die anderen beiden Browser trotz fehlender Hardwarebeschleunigung weit hinter sich lassen.

Ausreichend kleine Objekte können in allen Browsern mit einer ansehnlichen Geschwindigkeit gezeichnet werden. Sowohl das kleine als auch das mittlere Dreieck erreicht auf allen Browsern eine fünfstellige Anzahl an Iterationen. Größere Objekte hingegen drücken die Performance signifikant.

Wie auch schon in der vorherigen Testreihe ist zu empfehlen, mit der Path API vorsichtig umzugehen. Vor allem großflächige Objekten sollten nur zurückhaltend eingesetzt werden.

3.2.4 Path API 2

Die Canvas Path API bietet nicht nur die Möglichkeit primitive Objekte mithilfe von Linien zu zeichnen, sondern bietet auch eine Reihe weiterer Funktionen zum zeichnen komplexerer Objekte. Die folgende Testreihe untersucht die Performance der verschiedenen Figuren und wie sie sich voneinander unterscheiden. Zum Vergleich zur einfachen Path API ist der erste Testfall erneut ein Dreieck aus einzelnen Linien.

Es werden die Funktionen `rect`, `arcTo`, `arc`, `bezierCurveTo` und `quadraticCurveTo` untersucht. Der Initialisierungsblock erzeugt erneut das Canvas Element.

3.2.4.1 Ergebisse

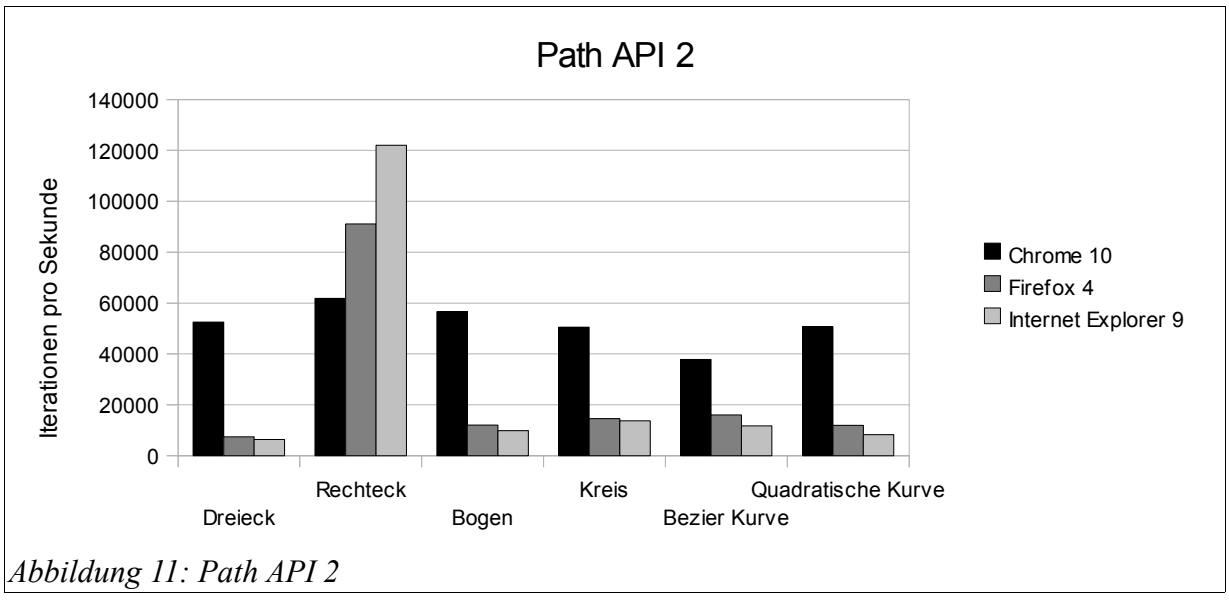


Abbildung 11: Path API 2

Testfall	Chrome 10	Firefox 4	Internet Explorer 9
Dreieck	52.479	7.387	6.375
Rechteck	61.856	91.128	122.009
Bogen	56.616	12.095	9.853
Kreis	50.499	14.556	13.748
Bezier Kurve	37.777	16.043	11.709
Quadratische Kurve	50.706	11.915	8.310

Tabelle 12: Path API 2

Die verschiedenen Figuren scheinen nur unbedeutende Performanceunterschiede vorzuweisen, vor allem wenn man den leichten Größenunterschied der einzelnen Elemente mit in Betracht zieht. Eine Ausnahme stellt allerdings erneut das Rechteck dar, das vermutlich aufgrund seiner nicht vorhandenen Schrägen eine deutliche höhere Performance als die restlichen Figuren zeigt.

Interessant ist auch hier wieder die deutliche Dominanz des Chrome Browsers, trotz der fehlenden Hardwarebeschleunigung. Die höhere Performance der beiden anderen Browser beim Zeichnen des Rechteckes lässt jedoch vermuten, dass auch hier die Schrägen und damit das Anti-Aliasing für den enormen Performanceverlust verantwortlich sind.

Erneut muss empfohlen werden nicht übermäßig Gebrauch von der Path API zu machen. Zwar können alle Browser eine beachtliche Zahl von Iterationen pro Sekunde hinlegen, doch überaus komplexe Szenen, wie sie in modernen Desktop Spielen üblich sind, lassen sich damit nicht realisieren. Einfachere Szenen mit wenigen hundert Objekten stellen allerdings kein Problem dar.

3.2.5 Stroke und Fill

Die Path API erlaubt es Objekte grundsätzlich auf zwei verschiedene Varianten zu zeichnen. In allen bisher gezeigten Testreihen wurde ausschließlich die `stroke` Methode genutzt. `stroke` zeichnet ein Objekt mit einer Konturlinie. Die alternative zu `stroke` ist `fill`. `fill` zeichnet ein Objekt indem es den von ihm eingeschlossenen Innenbereich mit einer Farbe, einem Farbverlauf oder einem Muster füllt.

Die beiden Zeichenmethoden haben vermutlich unterschiedliche Perfomanceeigenschaften. Die folgende Testreihe soll dies untersuchen. Dabei wird wieder beachtet wie sich die Performance für unterschiedlich große Objekte verhält. Erneut werden drei unterschiedlich große Dreiecke gezeichnet., jedes jeweils einmal mithilfe von `stroke` und einmal mithilfe von `fill`.

Der Initialisierungsblock erzeugt das Canvas Element.

3.2.5.1 Ergebnisse

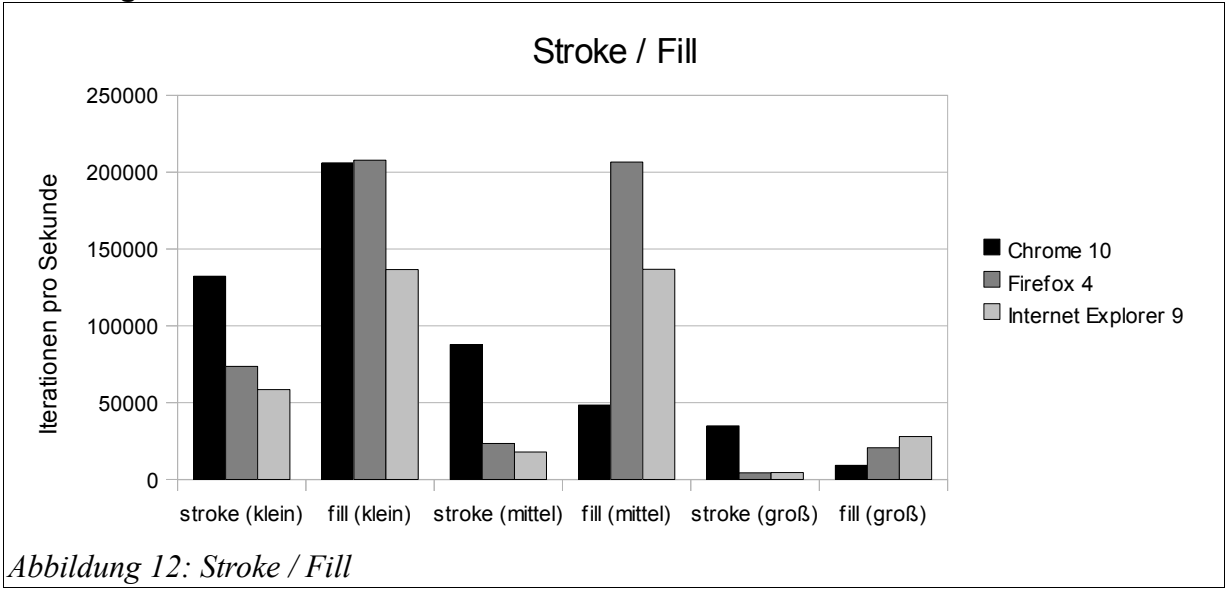


Abbildung 12: Stroke / Fill

Testfall	Chrome 10	Firefox 4	Internet Explorer 9
stroke (klein)	132.192	73.525	58.571
fill (klein)	205.953	207.779	136.645
stroke (mittel)	87.869	23.386	17.874
fill (mittel)	48.493	206.516	136.839
stroke (groß)	34.873	4.342	4.602
fill (groß)	9.253	20.577	27.990

Tabelle 13: Stroke / Fill

Wie sich herausstellt ist `fill` in beinahe allen untersuchten Fällen die Effizientere Lösung. Nur bei größeren Flächen im Chrome Browser ist `fill` langsamer als `stroke`. Die Zahl der Iterationen für `fill` liegt in einem deutlich höheren Bereich und erlaubt selbst große Flächen mit einer hohen Wiederholrate zu zeichnen.

Interessant ist, dass `fill` im Verhältnis zu `stroke` im Chrome Browser mit ansteigender Objektgröße stets schlechter abschneidet. Der Effekt ist bei den anderen beiden Browser nicht oder nur deutlich weniger stark zu beobachten. Allerdings wiegen die überragende Performance von `fill` in den anderen Browsern und das generell gute Abschneiden des Chrome Browsers diesen Nachteil leicht wieder auf.

Steht die Wahl zwischen `fill` und `stroke` offe, so sollte das im Schnitt schnellere `fill` in jedem Fall vorgezogen werden.

3.2.6 Multi- und Singlepath

Die Canvas Path API ermöglicht es wie nun bereits mehrfach demonstriert zusammenhängende Objekte entlang eines Pfades zu zeichnen. Dabei ist es möglich mithilfe des Befehls `moveTo` einen Unterpfad zu beginnen und somit Lücken im Pfad zu belassen. Im Grunde ist es somit möglich zwei

visuell voneinander unabhängige Objekte mithilfe eines einzigen Pfades zu zeichnen.

Als Beispiel dienen soll eine Reihe von Rechtecken, die, der Einfachheit halber, alle übereinander gezeichnet werden. Es besteht die Möglichkeit für jedes Rechteck einen neuen Pfad zu beginnen und am Ende der Zeichenoperationen für jedes Rechteck separat `stroke` aufzurufen. Oder aber es werden die Rechtecke entlang eines einzigen Pfades gezeichnet indem man `moveTo` nutzt um Unterpfade zu erzeugen. Sobald alle Rechtecke mithilfe der Path API konstruiert sind, wird ein einziges mal `stroke` aufgerufen um alle Rechtecke gleichzeitig zu zeichnen.

Dadurch wird die Gesamtzahl der Zeichenoperationen reduziert, was höchst warscheinlich zu einem Performancegewinn führen kann. Dies würde der Path API helfen ihre Effizienz zu steigern und es damit erlauben komplexere Szenen zu zeichnen.

In der folgenden Testreihe werden beide Szenarien miteinander verglichen. Es werden 100 Rechtecke der Reihe nach übereinander gezeichnet. Einmal erhält jedes Rechteck einen eigenen Pfad und im anderen Fall werden alle Rechtecke entlang eines einzigen Pfades gezeichnet. Der Initialisierungsblock erzeugt das Canvas Element und initialisiert alle nötigen Variablen.

3.2.6.1 Ergebnisse

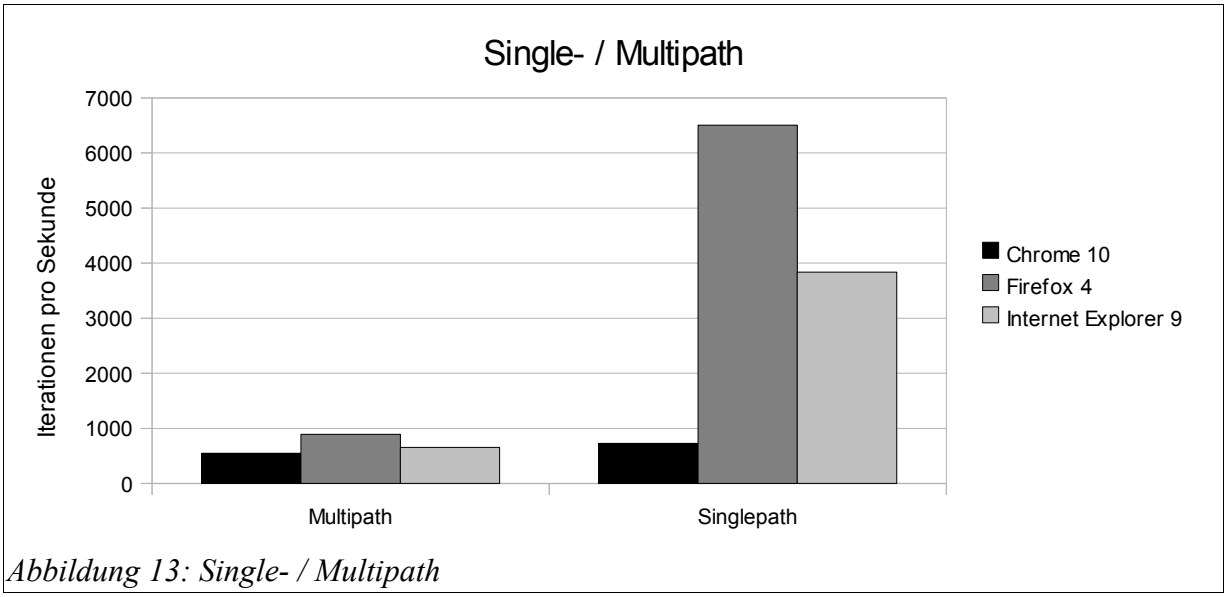


Abbildung 13: Single- / Multipath

Testfall	Chrome 10	Firefox 4	Internet Explorer 9
Multipath	549	893	653
Singlepath	726	6.507	3.835

Tabelle 14: Single- / Multipath

Der Effekt ist eindeutig zu sehen. Die Rechtecke alle in einem einzigen Pfad zu zeichnen ist in allen Browsern deutlich effektiver. In Chrome ist der Effekt zwar weniger stark, aber dennoch nachvollziehbar.

Diese Optimierungsmöglichkeit gut eingesetzt könnte die Anzahl der mit der Path API renderbaren Objekte leicht vervielfachen. Aus diesem Grund sollte von dem Singlepath Ansatz möglichst oft Gebrauch gemacht werden.

3.2.7 Path Clipping

Oft ist es beim Neuzeichnen einer Szene nicht notwendig alle Bereiche tatsächlich neu zu zeichnen. Die Technik des Clipping wurde bereits vorgestellt um dies zu ermöglichen. Da Canvas ebenfalls Clipping durch die `clip` Funktion unterstützt liegt es nahe diese Optimierungstechnik auf ihre Effizienz für die Path API zu untersuchen.

Die folgende Testreihe wird dies untersuchen. Dazu werden insgesamt 6 verschiedene Fälle unterschieden.

- Kein Clipping
- Teilweises Clipping
- Deckendes Clipping (Objekt liegt komplett in der Clip Region)
- Nur Clipping (Objekt wird nicht gezeichnet, nur Clip Region gesetzt)
- Leeres Clipping (Objekt wird gezeichnet, liegt aber gänzlich außerhalb der Clip Region)
- Nur Leeres Clipping (Objekt wird nicht gezeichnet, läge aber gänzlich außerhalb der Clip Region)

Als Objekt dient uns ein Dreieck. Die beiden Testfälle, in denen das Objekt nicht gezeichnet und lediglich die Clip Region gesetzt wird, dienen lediglich als Referenz, um sehen zu können wie aufwendig das setzen einer Clip Region im Vergleich zur eigentlichen Zeichenoperation ist.

Der Initialisierungsblock erstellt das Canvas Element.

3.2.7.1 Ergebnisse

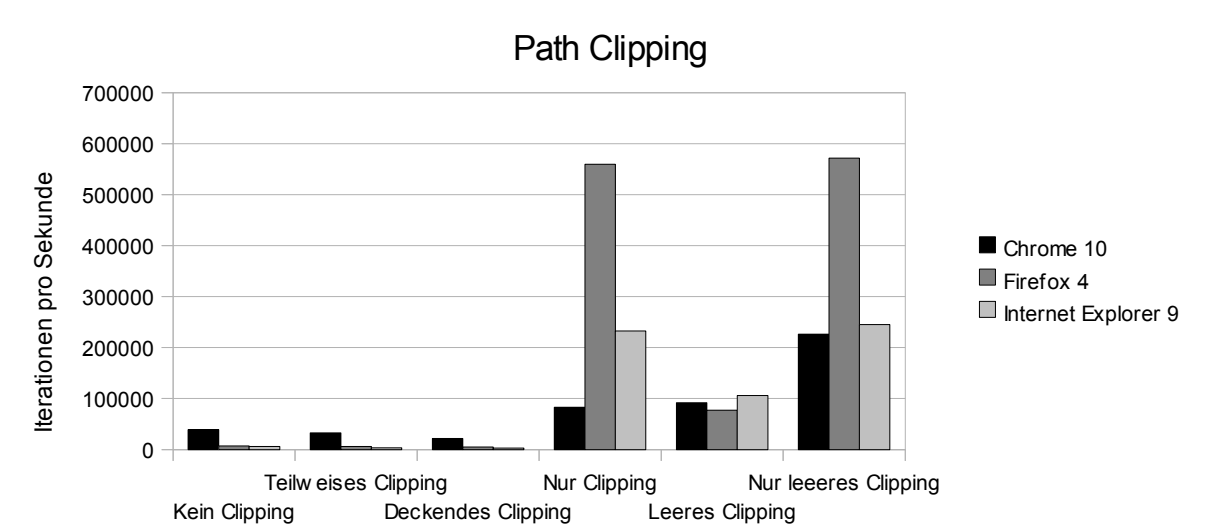


Abbildung 14: Path Clipping

Testfall	Chrome 10	Firefox 4	Internet Explorer 9
Kein Clipping	39.068	7.346	5.832
Teilweises Clipping	32.698	6.159	3.661
Deckendes Clipping	21.587	4.927	2.890
Nur Clipping	83.154	559.839	232.780
Leeres Clipping	91.873	77.061	106.108
Nur leeeres Clipping	226.266	571.550	245.241

Tabelle 15: Path Clipping

Das Clipping scheint sich auf den ersten Blick negativ auf die Leistung der Path API auszuwirken. Beachtet man allerdings das Messergebnis für das leere Clipping, so fällt auf, dass in den anderen Testfällen der Inhalt des geclippten Bereiches lediglich nicht komplex genug ist um einen Netto-Performancegewinn zu erzielen.

Komplexere Szenen sollten einen sichtbaren Performancegewinn durch Clipping erzielen können. Darum sollte ab einer ausreichend komplexen Szene von der Möglichkeit des Clippings Gebrauch gemacht werden. Was eine ausreichend komplexe Szene ist, muss allerdings für den Einzelfall entschieden werden.

3.2.8 Image API

Die Canvas API bietet nicht nur die Möglichkeit primitive Objekte zu zeichnen, sondern stellt auch Funktionen zum zeichnen von Bildern zur Verfügung. Dies ist vor allem für Browsergames von entscheidender Bedeutung, denn ein Großteil der in Browsergames eingesetzten Objekte sind schlicht Bilder und zu Animationen zusammengesetzte Reihen von Bildern.

Die Canvas Image API ist sehr simpel. Im Grunde besteht sie aus einer Funktion, die für die verschiedenen Anwendungsfälle überladen ist: `drawImage`. Eine hohe Performance dieser Funktion ist Grundvoraussetzung für den Einsatz von Canvas für Browsergames.

Die folgende Testreihe untersucht die Image API auf ihre Leistungsfähigkeit und vergleicht dabei wie sich die API beim Zeichnen von unterschiedlich großen Bildern verhält.

Der Initialisierungsblock erstellt erneut das Canvas Element und erzeugt drei verschieden große Image Objekte, welche später auf durch die Testfälle auf das Canvas gezeichnet werden.

3.2.8.1 Ergebnisse

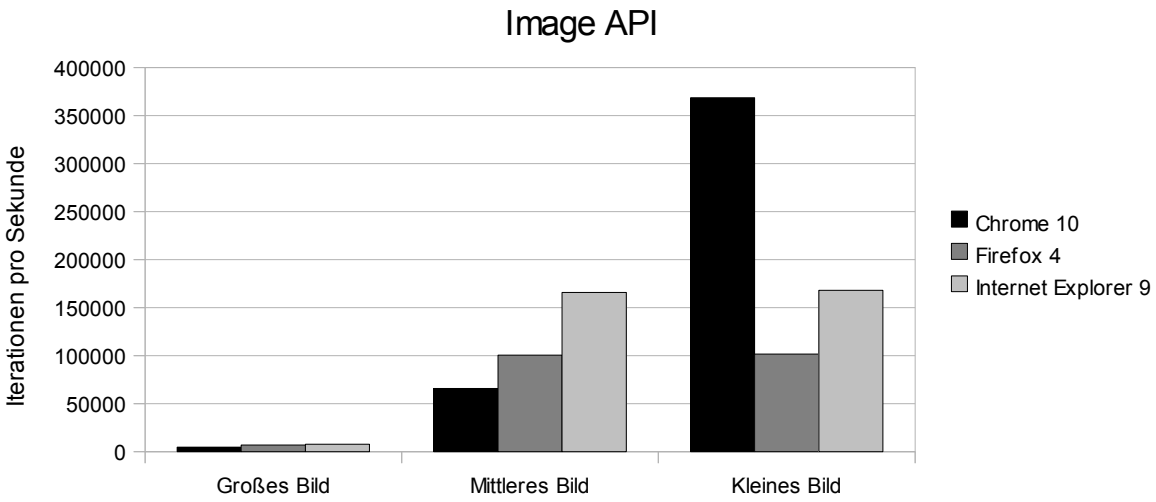


Abbildung 15: Image API

Testfall	Chrome 10	Firefox 4	Internet Explorer 9
Großes Bild	4.848	6.816	7.801
Mittleres Bild	65.821	100.609	165.941
Kleines Bild	368.653	101.787	168.152

Tabelle 16: Image API

Wie sehr schön an den Messdaten zu sehen ist können kleine und mittelgroße Bilder äußerst performant gezeichnet werden. Und selbst große Bilder erreichen noch durchaus akzeptable Iterationszahlen.

Kleine und mittelgroße Bilder können beinahe ohne Bedenken genutzt werden. In einem normalen Rahmen eingesetzt dürften sie keine Performanceprobleme erzeugen. Da Bilder von besonderer Wichtigkeit sind, sind diese Ergebnisse äußerst positiv zu bewerten.

3.2.9 Image Clipping

Auch für Bilder gilt, dass nicht alle Bereiche stets neu gezeichnet werden müssen. Mithilfe von Clipping ist es auch hier möglich den neu zu zeichnenden Bereich einzuschränken und so wohl möglich Leistungsgewinne zu erzielen.

Die Canvas API macht bezüglich der Clipping Region keinen Unterschied zwischen Path oder Image Operationen[27] und somit unterscheidet sich auch die Anwendung nicht, mit der Ausnahme, dass kein Path Objekt sondern ein Bild gezeichnet wird.

Auch in der folgenden Testreihe wurden wieder verschiedene Clipping Szenarien durchgespielt. Diesmal allerdings wurde darauf geachtet, dass die Clip Region auch im leeren Fall die selbe Größe aufweist wie in den anderen Szenarien, um die Vergleichbarkeit untereinander etwas zu erhöhen.

27 Tatsächlich macht die Canvas API garkeine unterschiede beim Clipping. Alle Zeichenoperationen, mit der Ausnahme der Pixeloperationen, werden durch die Clipping Region beschränkt. Sogar das Clipping selbst.

Damit fällt auch die Notwendigkeit für ein nur leeres Clipping zur besseren Vergleichbarkeit weg.

Der Initialisierungsblock erzeugt das Canvas Element und ein Bild Objekt, dass von den Testfällen auf das Canvas gezeichnet wird.

3.2.9.1 Ergebnisse

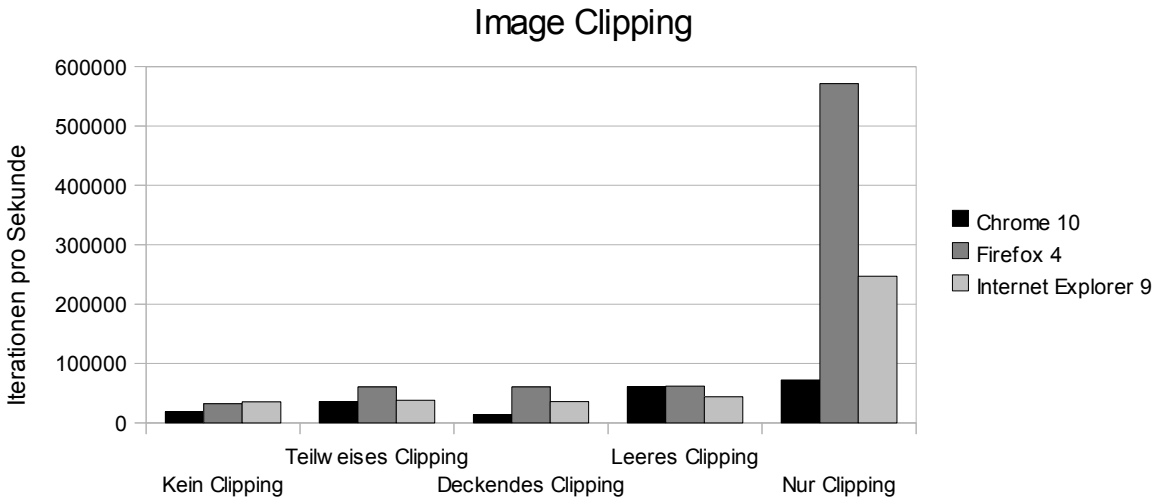


Abbildung 16: Image Clipping

Testfall	Chrome 10	Firefox 4	Internet Explorer 9
Kein Clipping	18.623	32.396	35.317
Teilweises Clipping	35.573	60.497	38.089
Deckendes Clipping	13.891	60.308	35.871
Leeres Clipping	60.832	61.727	44.000
Nur Clipping	72.146	571.390	246.734

Tabelle 17: Image Clipping

Das teilweise geclippte Bild wird von allen Browser schneller gezeichnet als das nicht geclippte Bild. Damit ist auch hier eine Performancesteigerung durch den Einsatz von Clipping zu erzielen.

Eigenartig jedoch ist die erhöhte Performance des deckend geclippten Bildes im Firefox Browser. Eine genauere Betrachtung zeigt, dass die hier angegebene Zahl für diesen einen konkreten Testlauf *nur dann* auftreten, wenn der Testfall im Rahmen der gesamten Testreihe ausgeführt wird. Wird der Testfall einzeln ausgeführt erreicht er lediglich eine Laufzeit die etwa der des kein Clipping Testfall entspricht, aber noch immer leicht höher liegt. Dies sollte nicht auftreten. Leider konnte bis zum Abschluss der Arbeit keine Erklärung für diesen Effekt gefunden werden.

Generell gilt aber dennoch: durch Clipping kann das Zeichnen von Bildern beschleunigt werden und somit sollte davon Gebrauch gemacht werden. Insbesondere, wenn mehrere oder besonders große Bilder gezeichnet werden und man eine relativ kleine Clip Region nutzen kann.

3.2.10 Text API

Die Darstellung von Text ist ein wichtiges Feature auch für Browsergames. Statusmeldungen, Beschriftungen von Interface Elementen oder Eigenschaften von Ausrüstungsgegenständen. Alles benötigt die Ausgabe von Text.

Auch das Canvas Element verfügt über eine Text API. Obwohl Texte in der Regel nicht die dominanten Elemente in einem Browsergame sind, so sind sie dennoch von enormer Wichtigkeit und kommen regelmäßig vor.

Wie sehr sich die Ausgabe von Text auf dem Canvas Element auf die Performance auswirkt soll in der folgenden Testreihe untersucht werden. Dazu wird zwischen drei verschiedenen Textausgaben unterschiedenen: einem kurzen Text, einem langen Text und einem besonders großflächigen Text. Alle drei Texte werden sowohl einmal mit `stroke` als auch einmal mit `fill` auf die Canvas Fläche gezeichnet.

Der Initialisierungsblock erzeugt das Canvas Element.

3.2.10.1 Ergebnisse

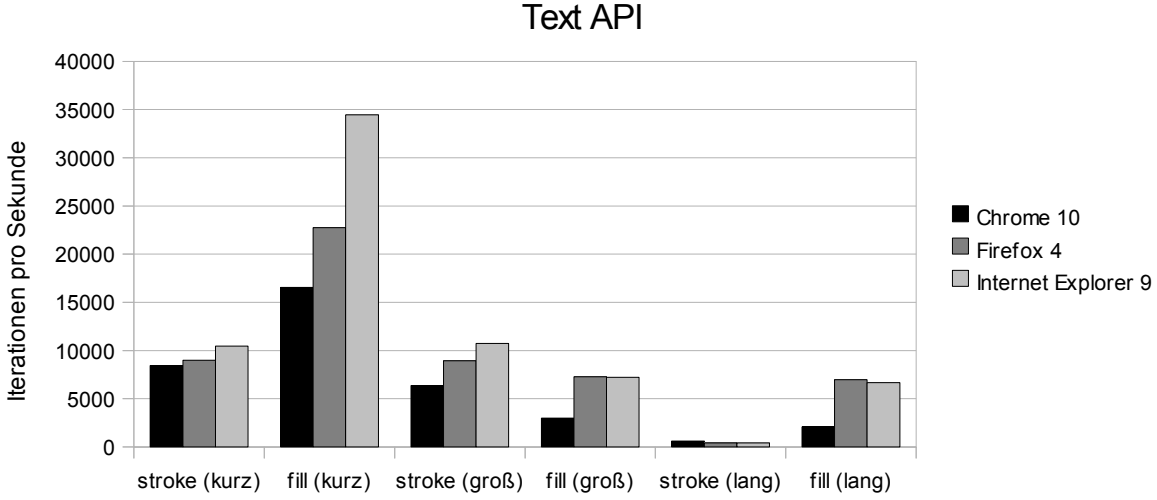


Abbildung 17: Text API

Testfall	Chrome 10	Firefox 4	Internet Explorer 9
stroke (kurz)	8.443	8.997	10.451
fill (kurz)	16.543	22.751	34.456
stroke (groß)	6.367	8.943	10.753
fill (groß)	2.993	7.276	7.229
stroke (lang)	606	412	425
fill (lang)	2.094	6.972	6.670

Tabelle 18: Text API

Kleiner Text, und zwar grundsätzlich unabhängig davon, wie viel Text ausgegeben wird, scheint auf allen Browser besser abzuschneiden, wenn er mithilfe von `fill` gezeichnet wird. Große

Mengen an Text mit `stroke` gezeichnet haben sogar geradezu katastrophale Itereationszahlen. Großflächiger Text hingegen kann ein wenig effizienter mit `stroke` als mit `fill` gezeichnet werden. Die Unterschiede sind allerdings eher geringfügig.

Es empfiehlt sich kleinen Text möglichst mit `fill` zu zeichnen. Die Geschwindigkeit ist für kleinere Mengen Text sehr gut. Für größere Mengen Text sollte auf keinen Fall auf `stroke` zurück gegriffen werden. Die Performance für `stroke` ist mit größeren Mengen Text nicht tragbar, zumindest sofern der Text wiederholt gezeichnet werden muss.

3.2.11 Culling

Objekte, die nach einer Zeichenoperation ohnehin nicht sichtbar sind, müssen erst gar nicht an die Grafik API zum zeichnen übergeben werden. Das Aussortieren solcher nicht sichtbaren Objekte bezeichnet man als Culling. Gerade in komplexen Szenen sollte sich mit einer solchen Technik viel Performance gewinnen lassen.

In der folgenden Testreihe soll diese Technik auf ihre Effektivität im Einsatz mit Canvas überprüft werden. Dazu wird eine komplexe Szene bestehend aus 100 zufällig verteilten, blauen, großflächigen Kreisen generiert und gezeichnet. Anschließend wird in jedem Testdurchlauf einer dieser Kreise zufällig bestimmt und dessen Farbe auf rot geändert. Anschließend wird die Repräsentation der Szene auf dem Canvas aktualisiert.

Es werden drei verschiedene Optimierungsgrade genutzt, die jeweils in einem eigenen Testfall implementiert sind. Der erste, unoprimierte Fall wird einfach die gesamte Szene mit allen 100 Kreisen neu zeichnen. Der zweite wird mithilfe von Clipping den zu zeichnenden Bereich eingrenzen, aber dennoch alle 100 Kreise an die Grafik API übergeben. Der letzte Optimierungsschritt wird nun letzten Endes Clipping und Culling einsetzen und nur die Kreise, die in der Bounding Box[28] des sich geänderten Kreises liegen, neu zeichnen. Alle anderen Kreise werden einfach verworfen und nicht neu gezeichnet.

Im Initialisierungsblock wird das Canvas Element erstellt, das Array mit den Kreisen angelegt, eine Funktion zum zeichnen eines beliebigen Kreise definiert und das Canvas erstmalig mit 100 blauen Kreisen gezeichnet.

28 zu deutsch: minimal umgebendes Rechteck.

3.2.11.1 Ergebnisse

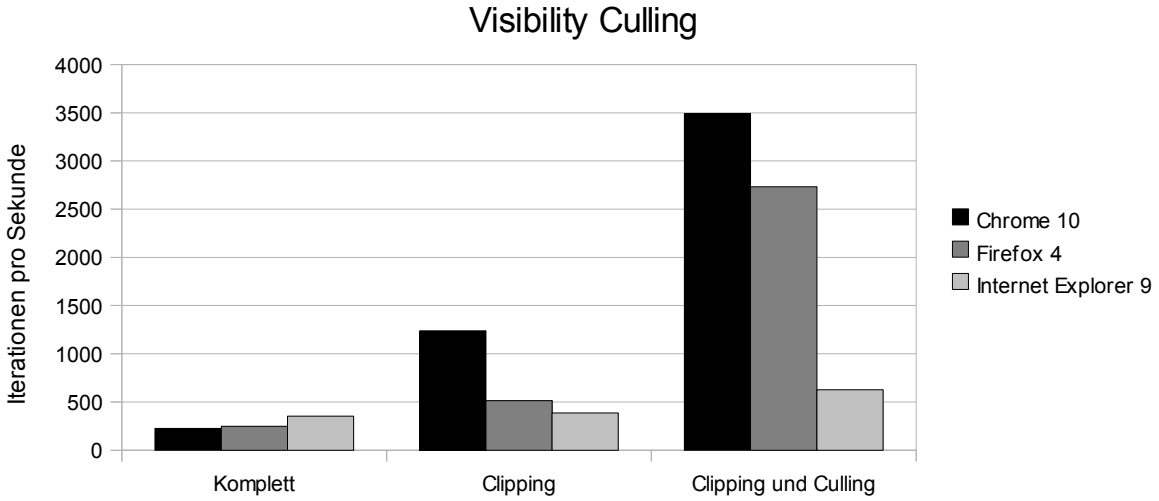


Abbildung 18: Culling

Testfall	Chrome 10	Firefox 4	Internet Explorer 9
Komplett	225	247	353
Clipping	1237	514	385
Clipping und Culling	3490	2731	626

Tabelle 19: Culling

Die Performance der recht komplexen Szene mit 100 Kreisen kann durch den Einsatz von Culling in den Chrome und Firefox Browsern um mehr als das 10fache gesteigert werden. Der Internet Explorer profitiert leider nicht ganz so stark, aber auch seine Leistung steigt um beinahe 80%.

Der eingesetzte Testfall könnte sogar noch weiter optimiert werden. So könnten die Koordinaten der Clip Region in einer lokalen Variable gecached werden und die Kreise in eine Baumstruktur einsortiert werden, um ein effizienteres Culling zu ermöglichen.

Culling sollte in jeder Anwendung ab einer gewissen Komplexität vorhanden sein. Natürlich erfordert ein Culling Mechanismus einen gewissen Mehraufwand durch die Entwickler, die Resultate sollten aber überzeugen. Culling ist von grundlegender Bedeutung für die Implementierung komplexer Browsergames mit großen Spielwelten.

3.2.12 Buffering

Mithilfe von Buffering lässt sich der aktuelle Zustand des Canvas sichern um ihn zu einem späteren Zeitpunkt wieder herstellen zu können. Dies ist vor allem nützlich um komplexe Hintergründe nicht wiederholt neu zeichnen zu müssen.

Man stelle sich vor, man wolle die Szene aus dem letzten Beispiel als einen Hintergrund nutzen und davor einen weiteren Kreis sich ständig bewegen lassen. Dies würde es erfordern den komplexen Hintergrund nach jeder Bewegung des Kreises im Vordergrund zumindest teilweise neu

zu zeichnen.

Nutzt man jedoch einen Buffer, so ist es möglich den komplexen Hintergrund nur einmal zu zeichnen und im Buffer zu speichern. Möchte man dann einen bestimmten Bereich des Hintergrundes neu zeichnen, weil sich der Kreis im Vordergrund darüber hinweg bewegt hat, so kann man einfach diesen Bereich des Buffers an die entsprechende Stelle des Canvas zeichnen.

Damit dies effektiv möglich ist, ist es vor allem notwendig, dass das Zeichnen vom Buffer auf die Canvas Fläche schneller, als das Neuzeichnen der Canvas Fläche ist. Ansonsten kann kein Performancegewinn durch den Einsatz eines Buffers erzielt werden.

Nicht ganz so kritisch aber dennoch von Interesse ist, wie schnell sich ein Buffer erstellen lässt. Ist das erzeugen eines Buffers besonders langsam so verliert der Buffer zur Performancesteigerung an Attraktivität. Dies gilt insbesondere wenn der Buffer gelegentlich selbst neu gezeichnet werden muss, z.B. für den Fall dass sich der Hintergrund ändert.

Die folgende Testreihe untersucht darum wie schnell sich ein Buffer erzeugen lässt und wie schnell er wieder auf die Canvas Fläche gezeichnet werden kann. Zum Vergleich wird zusätzlich ein Testfall eingerichtet, welcher die Fläche komplett neu zeichnet.

Die Canvas API stellt insgesamt drei Möglichkeiten zur Verfügung einen Buffer zu erzeugen und anschließend wieder auf die Canvas Fläche zu zeichnen: Die erste Variante liest mithilfe der Pixel API Funktion `getImageData` die Pixelinformationen des Canvas aus und sichert diese. Die Daten können anschließend mit `putImageData` wieder auf das Canvas gezeichnet werden.

Die Zweite Variante nutzt die Funktion `toDataURL` des Canvas Objektes, um eine Base64[29] kodierten String Repräsentation des Bildes zu erhalten. Dieser String kann anschließend als `src` eines `Image` Objektes gesetzt werden. Das `Image` Objekt enthält nun den selben Inhalt wie die Canvas Fläche und kann zu einem späteren Zeitpunkt mithilfe von `drawImage` wieder auf die Canvas Fläche gezeichnet werden.

Die letzte Möglichkeit den Zustand eines Canvas zu sichern ist ein weiteres Canvas Objekt mithilfe von `document.createElement` zu erzeugen. Da die Methode `drawImage` als ersten Parameter anstelle eines Image Objektes auch ein anderes Canvas erhalten kann, ist es nun möglich den Zustand des original Canvas mithilfe von `drawImage` in das Buffer Canvas zu zeichnen. Anschließend kann die selbe Methode benutzt werden um das Buffer Canvas wieder in das original Canvas zu zeichnen.

Der Initialisierungsblock erzeugt das Canvas und zeichnet 100 zufällig verteilte Kreise darauf. Außerdem werden die verschiedenen Buffer vorbereitet.

29 Base64 ist ein ein Verfahren um beliebige Binärdaten im 7bit ASCII Zeichensatz zu kodieren

3.2.12.1 Ergebnisse

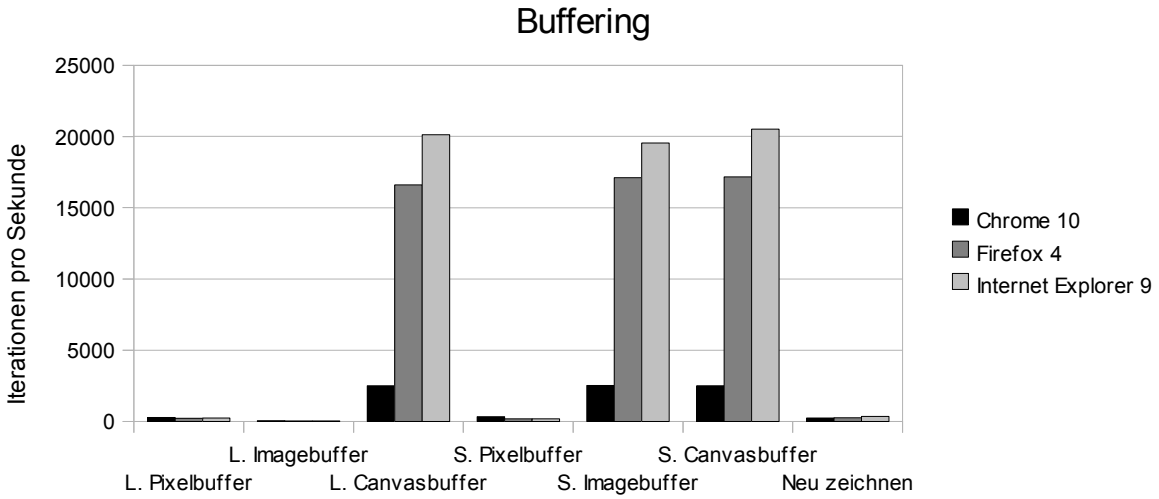


Abbildung 19: Buffering

Testfall	Chrome 10	Firefox 4	Internet Explorer 9
L. Pixelbuffer	276	212	239
L. Imagebuffer	52	29	28
L. Canvasbuffer	2509	16610	20132
S. Pixelbuffer	337	184	172
S. Imagebuffer	2516	17100	19545
S. Canvasbuffer	2509	17167	20516
Neu zeichnen	229	264	346

Tabelle 20: Buffering

Wie die Zahlen deutlich zeigen ist die Pixel API absolut ungeeignet um den gewünschten Effekt zu erzielen. Die meisten Browser brauchen länger die Pixeldaten wieder in das Canvas zu schreiben als benötigt wird um das Bild einfach neu zu zeichnen.

Das schreiben des Imagebuffers und des Canvasbuffers weisen in etwa die selbe Performance vor, was aufgrund der Tatsache, dass die selbe Funktion genutzt wird, auch nicht weiter verwunderlich ist. Die Performance ist in jedem Fall hervorragend , vor allem, wenn man sich vor Augen hält, dass es sich hierbei um ein 800x600 Pixel großes Bild handelt.

Allerdings ist das erzeugen des Imagebuffer mit abstand der langsamste Buffer in der Erzeugung. Selbst wenn er nicht gelegentlich erneuert werden müsste, wäre er aufgrund der schlechten Erzeugungsperformance absolut ungeeignet für die Aufgabe.

Da der Canvasbuffer sowohl zum sichern als auch zum wieder herstellen die selben Mechanismen nutzt, kann er genauso schnell erzeugt, wie er in das Canvas geschrieben werden kann und bietet somit ideale Voraussetzung um als Buffer eingesetzt zu werden.

3.3 Zusammenfassung

Das Canvas Element ist junge Technologie. Dies ist an einigen Stellen bemerkbar. Die Performance ist nicht in allen Bereichen optimal. Gerade im freien Zeichnen mithilfe der Path API und beim zeichnen von Text könnte die Performance etwas besser sein.

Aber die Performance genügt um einfache Browsergames wie sie üblich sind zu implementieren. Mit ein paar wenigen Kniffen und Optimierungstechniken, die heute bereits in Desktopspielen Anwendung finden, sind auch deutlich komplexere Szenen und Spiele denk- und machbar. Bereits einfache Optimierungstechniken wie Culling oder Buffering können wirklich erstaunliche Frameraten erzielen.

4 Schluss

4.1 Fazit

HTML5 und das Canvas Element als Teil davon helfen dem Web wieder einen Schritt in Richtung der Ideale, die Berners-Lee einmal als Grundsteine gelegt hat. Dies ist wichtig für das Web, das seine Innovationskraft über all die Jahre stets aus diesen Idealen gezogen hat. Ein Web ohne diese Offenheit und Interoperabilität würde seine treibende Kraft verlieren. Schon allein deshalb ist HTML5 ein Erfolg und wichtig für das Web.

Die Zahlen und Daten, die im Rahmen dieser Arbeit gesammelt wurden zeigen, dass ECMAScript und das HTML5 Canvas Element bereit dafür sind diese Offenheit und Innovationskraft zurück in den Browserspielemarkt zu bringen. Mit nur wenigen Tricks ist es möglich beachtliche Leistung aus ECMAScript und dem Canvas Element zu ziehen. Und die Entwicklung der Browser in diesem Bereich steht gerade erst am Anfang.

Doch ECMAScript und vor allem dem Canvas Element liegt ein großer Stolperstein gleich mehrfach im Weg: Verbreitung. Viele Spielehersteller setzen heutzutage auf proprietäre Lösungen wie Flash und haben dort ihr Know-How und ihre Expertise. Von einer Technologie, die man jahrelang eingesetzt hat weg zu migrieren benötigt einen gewichtigen Grund. Canvas und HTML5 im allgemeinen müssen erst noch beweisen, ob sie den Entwicklern diese nötigen Gründe liefern können.

Das Problem der Verbreitung endet allerdings nicht bei den Entwicklern. Viel gravierender ist die zur Zeit noch mangelnde Verbreitung von HTML5 Canvas fähigen Browsern. Nichteinmal 30% der heute eingesetzten Browser[30] verfügen über die hier vorgestellten Leistungsfähigen ECMAScript und Canvas Implementationen. Die restlichen und älteren Browser können entweder lange nicht mit der nötigen Performance dienen oder haben überhaupt keine Canvas Implementation.

Doch diese Probleme wird die Zeit lösen. Die Browserhersteller haben bereits angekündigt noch dieses Jahr neue Versionen ihre Browser zu veröffentlichen. Die Innovationszyklen im Browsermarkt werden kürzer und dadurch der offene Webstack konkurrenzfähiger und weniger träge.

Dem Einsatz von Canvas zur Entwicklung von Browsergames steht – zumindest aus technischer Sicht – nichts entgegen. Ob sich der Umstieg von Flash als Plattform aus wirtschaftlicher Sicht lohnt muss wohl jedes Unternehmen individuell abwägen.

30 [W3Schools, 2011]

4.2 Ein- und Ausblick

Bereits heute setzen viele Spieleentwickler auf Canvas und HTML5 und veröffentlichen erfolgreich Browserspiele unter Einsatz dieser jungen Technologie. So gibt es eine Canvas basierte Game Engine mit dem Namen IMPACT[31], welche bereits aktiv eingesetzt wird um Spiele zu entwickeln und die Isogenic Engine[32], ebenfalls auf Canvas basierend, befindet sich zur Zeit in einer geschlossenen Beta Phase. Auch ich selbst arbeite zur Zeit daran meine Erkenntnisse aus dieser Arbeit in eine Canvas Bibliothek zu kapseln[33].

Und die nahe Zukunft bietet noch mehr. Mit wachsender Popularität konnte ECMAScript mithilfe von node.js[34] einen Fuß als Sprache für die Entwicklung von Backendlösungen auf Serverseite fassen und die höchst skalierbaren NoSQL Document Stores wie CoucheDB[35] und MongoDB[36] nutzen ECMAScript als ihre Abfragesprache. So entsteht um ECMAScript und HTML5 herum ein Ecosystem modernen Hochleistungstechnologien, die hervorragend ineinander greifen und interagieren.

Und die Browserhersteller stehen ebenfalls nicht still. So kann das Canvas Element in Chrome bereits eine hardwarebeschleunigte 3D Zeichenfläche auf Basis von WebGL[37] zur Verfügung stellen und die anderen Browserhersteller haben angekündigt bald nachziehen zu wollen. So werden in naher Zukunft auch 3D Anwendungen im Browser realisierbar sein.

Die Nahe Zukunft bietet viele großartige Entwicklungen im Web Bereich, vor allem auch für Speieleentwickler. Wer nicht rechtzeitig auf diesen Zug aufspringt, verpasst ihn vielleicht und bleibt am Ende zurück.

31 Webseite: http://impactjs.com/
32 Webseite: http://www.isogenicengine.com/
33 Git Repository: git clone http://git.danielbaulig.de/fla.js.git, Demo: http://danielbaulig.de/fla.js/examples/stars.html
34 node.js ist eine Laufzeitumgebung inklusive asnychroner I/O für ECMAScript. Webseite: http://nodejs.org/
35 Webseite: http://couchdb.apache.org/
36 Webseite: http://www.mongodb.org/
37 Technische Spezifikation: http://www.khronos.org/registry/webgl/specs/latest/

Quellen- und Literaturverzeichnis

Berners-Lee, 1992: Tim Berners-Lee, Robert Cailliau, Jean-François Groff, Bernd Pollermann, World-wide web: the information universe, 1992
Garret, 2005: Jesse James Garrett, Ajax: A New Approach to Web Applications, 2005
Zakas, 2010: Nicholas C. Zakas, High Performance Javascript, 2010
Crockford, 2010a: Douglas Crockford, Crockford on JavaScript IV: The Metamorphosis of Ajax, 2010, http://developer.yahoo.com/yui/theater/video.php?v=crockonjs-4 (Abgerufen am 22.03.2011)
Adobe, 2010: ohne Autor, Flash Player Version Penetration, 2010, http://www.adobe.com/products/player_census/flashplayer/version_penetration.html (Abgerufen am 22.03.2011)
SPON, 2010: Stephan Freundorfer, Die kleinen Könige der Spielebranche, 2010, http://www.spiegel.de/netzwelt/games/0,1518,680100,00.html (Abgerufen am 22.03.2011)
WHATWG, 2011: WHATWG (Web Hypertext Application Technology Working Group), HTML - Living Standard, 2011, http://www.whatwg.org/specs/web-apps/current-work/ (Abgerufen am 23.03.2011)
W3C, 2011: W3C (World Wide Web Consortium), The WebSocket API, 2011, http://dev.w3.org/html5/websockets/ (Abgerufen am 23.03.2011)
JSPERF: ohen Autor, jsPerf - Frequently asked questions, ohne Jahr, http://jsperf.com/faq (Abgerufen am 23.03.2011)
Bynens, 2010: Mathias Bynens, John-David Dalton, Bulletproof Javascript benchmarks, 2010, http://calendar.perfplanet.com/2010/bulletproof-javascript-benchmarks/ (Abgerufen am 23.03.2011)
Law, 2009: Eric Law, Q&A: 64-Bit Internet Explorer, 2009, http://blogs.msdn.com/b/ieinternals/archive/2009/05/29/q-a-64-bit-internet-explorer.aspx (Abgerufen am 14.04.2011)
Powers, 2009: Shelly Powers, Learning Javascript, Second Edition, 2009
ECMA, 2009: ECMA (Ecma International), Standard ECMA-262 5th Edition / December 2009: ECMAScript Language Specification, 2009, http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf (Abgerufen am 23.03.2011)
Eich, 2008: Brendan Eich, Popularity, 2008, http://weblogs.mozillazine.org/roadmap/archives/2008/04/popularity.html (Abgerufen am 27.03.2011)
Crockford, 2010b: Douglas Crockford, Crockford on JavaScript II: And Then There Was JavaScript, 2010, http://developer.yahoo.com/yui/theater/video.php?v=crockonjs-2 (Aberufen am 27.03.2011)
Fulgham, 2011: Brent Fulgham, Computer Language Benchmarks Game, 2011, http://shootout.alioth.debian.org/u32/benchmark.php (Abgerufen am 14.04.2011)
Crockford, 2008: Douglas Crockford, JavaScript: The Good Parts, 2008
Duff, 1988: Tom Duff, Re: Explanation, please!, 1988, http://www.lysator.liu.se/c/duffs-device.html (Abgerufen am 15.04.2011)
Barron, 2001: Todd Barron, Multiplayer Game Programming, 2001
W3Schools, 2011: W3Schools, Browser Statistics, 2011, http://www.w3schools.com/browsers/browsers_stats.asp (Abgerufen am 17.04.2011)

Anhang

Es folgt einer Auflistung aller eingesetzten Testreihen. Jeweils mit angegeben ist der Link zur jsPerf Testreihe.

1 In- und Dekrementierung

Diese Testreihe kann unter der URL http://jsperf.com/incrementing-decrementing abgerufen, wiederholt, angepasst und erweitert werden.

1.1 Testfälle

Post-Increment

```
 1 var i = 0;
 2 i++;
 3 i++;
 4 i++;
 5 i++;
 6 i++;
 7 i++;
 8 i++;
 9 i++;
10 i++;
11 i++;
```

Pre-Increment

```
 1 var i = 0;
 2 ++i;
 3 ++i;
 4 ++i;
 5 ++i;
 6 ++i;
 7 ++i;
 8 ++i;
 9 ++i;
10 ++i;
11 ++i;
```

Addition

```
 1 var i = 0;
 2 i += 1;
 3 i += 1;
 4 i += 1;
 5 i += 1;
 6 i += 1;
 7 i += 1;
 8 i += 1;
 9 i += 1;
10 i += 1;
11 i += 1;
```

Post-Decrement

```
 1 var i = 0;
 2 i--;
 3 i--;
 4 i--;
 5 i--;
 6 i--;
 7 i--;
 8 i--;
 9 i--;
10 i--;
11 i--;
```

Pre-Decrement

```
 1 var i = 0;
 2 --i;
 3 --i;
 4 --i;
 5 --i;
 6 --i;
 7 --i;
 8 --i;
 9 --i;
10 --i;
11 --i;
```

Subtraktion

```
 1 var i = 0;
 2 i -= 1;
 3 i -= 1;
 4 i -= 1;
 5 i -= 1;
 6 i -= 1;
 7 i -= 1;
 8 i -= 1;
 9 i -= 1;
10 i -= 1;
11 i -= 1;
```

2 Schleifen

Diese Testreihe kann unter der URL http://jsperf.com/dba-loops abgerufen, wiederholt, angepasst und erweitert werden.

2.1 Initialisierung

```
<script>
 var c = 10000;
</script>
```

2.2 Testfälle

for Steigend

```
var i;
for (i = 0; i < c; i += 1) {
 i;
}
```

for Fallend

```
var i;
for (i = c; i > 0; i -= 1) {
 i;
}
```

while Steigend

```
var i = 0;
while (i < c) {
 i += 1;
}
```

while Fallend

```
var i = c;
while (i) {
 i -= 1;
}
```

do-while Steigend

```
var i = 0;
do {
 i;
} while ((i += 1) < c);
```

do-while Fallend

```
var i = c;
do {
 i;
} while (i -= 1);
```

3 Scope Resolution

Diese Testreihe kann unter der URL http://jsperf.com/scope-resolution abgerufen, wiederholt, angepasst und erweitert werden.

3.1 Initialisierung

```
window.globalVar = 0;

var closedFunc = (function() {
 var closedOver = 0;

 return (function() {
  var localVar = 0;
  closedOver += 1;
  closedOver += 1;
  closedOver += 1;
  closedOver += 1;
  closedOver += 1;
  closedOver += 1;
  closedOver += 1;
  closedOver += 1;
  closedOver += 1;
  closedOver += 1;
 });
})();

var localFunc = (function() {
 var closedOver = 0;
 return (function() {
  var localVar = 0;
  localVar += 1;
  localVar += 1;
  localVar += 1;
  localVar += 1;
  localVar += 1;
  localVar += 1;
  localVar += 1;
  localVar += 1;
  localVar += 1;
  localVar += 1;
 });
})();

var globalFunc = (function() {
 var closedOver = 0;

 return (function() {
  var localVar = 0;
  globalVar += 1;
  globalVar += 1;
  globalVar += 1;
  globalVar += 1;
  globalVar += 1;
  globalVar += 1;
  globalVar += 1;
  globalVar += 1;
  globalVar += 1;
  globalVar += 1;
 });
})();
```

3.2 Testfälle

Local

```
1 localFunc();
```

Global

```
1 globalFunc();
```

Closed Over

```
1 closedFunc();
```

4 Prototype Resolution

Diese Testreihe kann unter der URL http://jsperf.com/fixed-prototype-resolution abgerufen, wiederholt, angepasst und erweitert werden.

4.1 Initialisierung

```
1  var o = Object.create(Object);
2  var child = Object.create(o);
3  var grandChild = Object.create(child);
4
5  o.member = 0;
```

4.2 Testfälle

Local

```
1 o;
```

Member

```
1 o.member;
```

Child

```
1 child.member;
```

Grandchild

```
1 grandChild.member;
```

5 Funktionsaufrufe

Diese Testreihe kann unter der URL http://jsperf.com/dba-function-calling abgerufen, wiederholt, angepasst und erweitert werden.

5.1 Initialisierung

```
1 var func = function() {
2   var foo = 10;
3   var bar = 10;
4   var result = foo * bar;
5  };
6
7  var obj = {
8   method: func
9  };
```

5.2 Testfälle

Inline

```
1 var foo = 10;
2 var bar = 10;
3 var result = foo * bar;
```

Anonymous

```
1 (function() {
2  var foo = 10;
3  var bar = 10;
4  var result = foo * bar;
5 })();
```

Stored

```
1 func();
```

Member

```
1 obj.method();
```

6 Funktionsparameter

Diese Testreihe kann unter der URL http://jsperf.com/parameter abgerufen, wiederholt, angepasst und erweitert werden.

6.1 Initialisierung

```
1 var func = function() {
2   return 0;
3  };
4
5  var obj_func = function() {
6   return 0;
7  }
```

6.2 Testfälle

Kein Parameter

```
1 func();
```

Ein Parameter

```
1 func(0);
```

Zwei Parameter

```
1 func(0, 0);
```

Drei Parameter

```
1 func(0, 0, 0);
```

Objekt Parameter

```
1 obj_func({
2  one: 0,
3  two: 0,
4  three: 0
5 });
```

7 Funktionsrückgabewerte

Diese Testreihe kann unter der URL http://jsperf.com/return abgerufen, wiederholt, angepasst und erweitert werden.

7.1 Initialisierung

```
1  var no_return = function() {};
2  var return_empty = function() {
3   return;
4  };
5  var return_primitive = function() {
6   return 0;
7  };
8  var return_object = function() {
9   return {
10   value: 0
11  };
12 };
```

7.2 Testfälle

Kein Rückgabewert

```
1 no_return();
```

Leerer Rückgabewert

```
1 return_empty();
```

Einfacher Rückgabewert

```
1 return_primitive();
```

Objekt Rückgabewert

```
1 return_object();
```

8 Member Resolution

Diese Testreihe kann unter der URL http://jsperf.com/dba-identifier-resolution abgerufen, wiederholt, angepasst und erweitert werden.

8.1 Initialisierung

```
1 var variable = 0;
2  var object = {
3   member: 0
4  };
5
6  var container = {
7   element: object
8  };
```

8.2 Testfälle

Variable

```
1 variable;
2 variable;
```

Eigenschaft

```
1 object.member;
2 object.member;
```

Verschachtelte Eigenschaft

```
1 container.element.member;
2 container.element.member;
```

Cache

```
1 var cache = container.element.member;
2 cache;
3 cache;
```

9 Clearing

Diese Testreihe kann unter der URL http://jsperf.com/canvas-clearing abgerufen, wiederholt, angepasst und erweitert werden.

9.1 Initialisierung

```
1 <canvas width="800" height="600" id="canvas">
2 </canvas>
3 <canvas width="200" height="150" id="canvas_small">
4 </canvas>
5 <canvas width="1680" height="1050" id="canvas_large">
6 </canvas>
7 <script>
8  var canvas = document.getElementById('canvas');
9  var context = canvas.getContext('2d');
10  var canvas_small = document.getElementById('canvas_small');
11  var context_small = canvas_small.getContext('2d');
12  var canvas_large = document.getElementById('canvas_large');
13  var context_large = canvas_large.getContext('2d');
14 </script>
```

9.2 Testfälle

width (klein)

```
1 canvas_small.width = canvas_small.width;
```

clearRect(klein)

```
1 var width = canvas_small.width;
2 var height = canvas_small.height;
3 context_small.clearRect(0, 0, width, height);
```

width (mittel)

```
1 canvas.width = canvas.width;
```

clearRect (mittel)

```
1 var width = canvas.width;
2 var height = canvas.height;
3 context.clearRect(0, 0, width, height);
```

width (groß)

```
1 canvas_large.width = canvas_large.width;
```

clearRect (groß)

```
1 var width = canvas_large.width;
2 var height = canvas_large.height;
3 context_large.clearRect(0, 0, width, height);
```

10 Path API 1

Diese Testreihe kann unter der URL http://jsperf.com/canvas-polygon abgerufen, wiederholt, angepasst und erweitert werden.

10.1 Initialisierung

```
1 <canvas width="800" height="600" id="canvas">
2 </canvas>
3 <script>
4  var canvas = document.getElementById('canvas');
5  var context = canvas.getContext('2d');
6
7  var
8  w = canvas.width,
9      h = canvas.height;
10 </script>
```

10.2 Testfälle

Fallende Gerade

```
1 context.beginPath();
2 context.moveTo(w / 4, h / 4);
3 context.lineTo(w / 4 * 3, h / 4 * 3);
4 context.stroke();
```

Dreieck

```
1 context.beginPath();
2 context.moveTo(w / 4, h / 4);
3 context.lineTo(w / 4 * 3, h / 4);
4 context.lineTo(w / 2, h / 4 * 3);
5 context.closePath();
6 context.stroke();
```

Rechteck

```
1 context.beginPath();
2 context.moveTo(w / 4, h / 4);
3 context.lineTo(w / 4 * 3, h / 4);
4 context.lineTo(w / 4 * 3, h / 4 * 3);
5 context.lineTo(w / 4, h / 4 * 3);
6 context.closePath();
7 context.stroke();
```

Polygon

```
1 context.beginPath();
2 context.moveTo(w / 4, h / 4);
3 context.lineTo(w / 2, h / 4 - h / 8);
4 context.lineTo(w / 4 * 3, h / 4);
5 context.lineTo(w / 4 * 3 + w / 8, h / 2);
6 context.lineTo(w / 4 * 3, h / 4 * 3);
7 context.lineTo(w / 4, h / 4 * 3);
8 context.closePath();
9 context.stroke();
```

Horizontale Gerade

```
1 context.beginPath();
2 context.moveTo(w / 4, h / 4);
3 context.lineTo(w / 4 * 3, h / 4);
4 context.stroke();
```

11 Path Size

Diese Testreihe kann unter der URL http://jsperf.com/path-size abgerufen, wiederholt, angepasst und erweitert werden.

11.1 Initialisierung

```
<canvas width="800" height="600" id="canvas"></canvas>
<script>
 var canvas = document.getElementById('canvas');
 var context = canvas.getContext('2d');
 var w = canvas.width,
     h = canvas.height;
</script>
```

11.2 Testfälle

Klein

```
context.beginPath();
context.moveTo(0, 0);
context.lineTo(0, 10);
context.lineTo(10, 10);
context.closePath();
context.stroke();
```

Mittel

```
context.beginPath();
context.moveTo(0, 0);
context.lineTo(0, 100);
context.lineTo(100, 100);
context.closePath();
context.stroke();
```

Groß

```
context.beginPath();
context.moveTo(0, 0);
context.lineTo(0, 500);
context.lineTo(500, 500);
context.closePath();
context.stroke();
```

12 Path API 2

Diese Testreihe kann unter der URL http://jsperf.com/canvas-shapes abgerufen, wiederholt, angepasst und erweitert werden.

12.1 Initialisierung

```
<canvas width="800" height="600" id="canvas"></canvas>
<script>
 var canvas = document.getElementById('canvas');
 var context = canvas.getContext('2d');
 var w = canvas.width,
     h = canvas.height;
</script>
```

12.2 Testfälle

Dreieck

```
context.beginPath();
context.moveTo(100, 100);
context.lineTo(400, 100);
context.lineTo(400, 400);
context.closePath();
context.stroke();
```

Rechteck

```
context.beginPath();
context.rect(100, 100, 300, 300);
context.stroke();
```

Bogen

```
context.beginPath();
context.moveTo(100, 100);
context.arcTo(400, 400, 100, 400, 40);
context.stroke();
```

Kreis

```
context.beginPath();
context.arc(250, 250, 100, 0, Math.PI);
context.stroke();
```

Bezier Kurve

```
context.beginPath();
context.moveTo(100, 100);
context.bezierCurveTo(100, 200, 300, 100, 300, 300);
context.stroke();
```

Quadratische Kurve

```
1 context.beginPath();
2 context.moveTo(100, 100);
3 context.quadraticCurveTo(200, 400, 300, 100);
4 context.stroke();
```

13 Stroke und Fill

Diese Testreihe kann unter der URL http://jsperf.com/canvas-stroke-vs-fill abgerufen, wiederholt, angepasst und erweitert werden.

13.1 Initialisierung

```
1 <canvas width="800" height="600" id="canvas"></canvas>
2 <script>
3  var canvas = document.getElementById('canvas');
4  var context = canvas.getContext('2d');
5  var w = canvas.width,
6      h = canvas.height;
7 </script>
```

13.2 Testfälle

stroke (klein)

```
1 context.beginPath();
2 context.moveTo(0, 0);
3 context.lineTo(0, 10);
4 context.lineTo(10, 10);
5 context.closePath();
6 context.stroke();
```

fill (klein)

```
1 context.beginPath();
2 context.moveTo(0, 0);
3 context.lineTo(0, 10);
4 context.lineTo(10, 10);
5 context.closePath();
6 context.fill();
```

stroke (mittel)

```
1 context.beginPath();
2 context.moveTo(0, 0);
3 context.lineTo(0, 100);
4 context.lineTo(100, 100);
5 context.closePath();
6 context.stroke();
```

fill (mittel)

```
1 context.beginPath();
2 context.moveTo(0, 0);
3 context.lineTo(0, 100);
4 context.lineTo(100, 100);
5 context.closePath();
6 context.fill();
```

stroke (groß)

```
1 context.beginPath();
2 context.moveTo(0, 0);
3 context.lineTo(0, 500);
4 context.lineTo(500, 500);
5 context.closePath();
6 context.stroke();
```

fill (groß)

```
1 context.beginPath();
2 context.moveTo(0, 0);
3 context.lineTo(0, 500);
4 context.lineTo(500, 500);
5 context.closePath();
6 context.fill();
```

14 Single- und Multipath

Diese Testreihe kann unter der URL http://jsperf.com/canvas-multi-singlepath-comparision abgerufen, wiederholt, angepasst und erweitert werden.

14.1 Initialisierung

```
1 <canvas width="800" height="600" id="canvas"></canvas>
2 <script>
3  var canvas = document.getElementById('canvas');
4  var context = canvas.getContext('2d');
5  var loops = 100,
6      w = canvas.width,
7      h = canvas.height;
8 </script>
```

14.2 Testfälle

Multipath

```
var i = loops;
while (i -= 1) {
 context.beginPath();
 context.moveTo(w / 4, h / 4);
 context.lineTo(w / 4, h / 4 * 3);
 context.lineTo(w / 4 * 3, h / 4 * 3);
 context.lineTo(w / 4 * 3, h / 4);
 context.closePath();
 context.stroke();
}
```

Singlepath

```
var
i = loops;
context.beginPath();
while (i -= 1) {
 context.moveTo(w / 4, h / 4);
 context.lineTo(w / 4, h / 4 * 3);
 context.lineTo(w / 4 * 3, h / 4 * 3);
 context.lineTo(w / 4 * 3, h / 4);
 context.closePath();
}
context.stroke();
```

15 Path Clipping

Diese Testreihe kann unter der URL http://jsperf.com/canvas-path-clipping abgerufen, wiederholt, angepasst und erweitert werden.

15.1 Initialisierung

```
<canvas width="800" height="600" id="canvas"></canvas>
<script>
 var canvas = document.getElementById('canvas');
 var context = canvas.getContext('2d');
 var w = canvas.width,
     h = canvas.height;
</script>
```

15.2 Testfälle

Kein Clipping

```
1 context.beginPath();
2 context.moveTo(w / 4, h / 4);
3 context.lineTo(w / 4 * 3, h / 4);
4 context.lineTo(w / 2, h / 4 * 3);
5 context.closePath();
6 context.stroke();
```

Teilweises Clipping

```
 1 context.save();
 2
 3 context.beginPath();
 4 context.moveTo(w / 3, h / 3);
 5 context.lineTo(w / 3, h / 3 * 2);
 6 context.lineTo(w / 3 * 2, h / 3 * 2);
 7 context.lineTo(w / 3 * 2, h / 3);
 8 context.closePath();
 9 context.clip();
10
11 context.beginPath();
12 context.moveTo(w / 4, h / 4);
13 context.lineTo(w / 4 * 3, h / 4);
14 context.lineTo(w / 2, h / 4 * 3);
15 context.closePath();
16 context.stroke();
17
18 context.restore();
```

Deckendes Clipping

```
 1 context.save();
 2
 3 context.beginPath();
 4 context.moveTo(w / 4, h / 4);
 5 context.lineTo(w / 4, h / 4 * 3);
 6 context.lineTo(w / 4 * 3, h / 4 * 3);
 7 context.lineTo(w / 4 * 3, h / 4);
 8 context.closePath();
 9 context.clip();
10
11 context.beginPath();
12 context.moveTo(w / 4, h / 4);
13 context.lineTo(w / 4 * 3, h / 4);
14 context.lineTo(w / 2, h / 4 * 3);
15 context.closePath();
16 context.stroke();
17
18 context.restore();
```

Nur Clipping

```
 1 context.save();
 2
 3 context.beginPath();
 4 context.moveTo(w / 3, h / 3);
 5 context.lineTo(w / 3, h / 3 * 2);
 6 context.lineTo(w / 3 * 2, h / 3 * 2);
 7 context.lineTo(w / 3 * 2, h / 3);
 8 context.closePath();
 9 context.clip();
10
11 context.restore();
```

Leeres Clipping

```
 1 context.save();
 2
 3 context.beginPath();
 4 context.moveTo(0, 0);
 5 context.lineTo(1, 0);
 6 context.lineTo(1, 1);
 7 context.lineTo(0, 1);
 8 context.closePath();
 9 context.clip();
10
11 context.beginPath();
12 context.moveTo(w / 4, h / 4);
13 context.lineTo(w / 4 * 3, h / 4);
14 context.lineTo(w / 2, h / 4 * 3);
15 context.closePath();
16 context.stroke();
17
18 context.restore();
```

Nur leeres Clipping

```
 1 context.save();
 2
 3 context.beginPath();
 4 context.moveTo(0, 0);
 5 context.lineTo(1, 0);
 6 context.lineTo(1, 1);
 7 context.lineTo(0, 1);
 8 context.closePath();
 9 context.clip();
10
11 context.restore();
```

16 Image API

Diese Testreihe kann unter der URL http://jsperf.com/canvas-image abgerufen, wiederholt, angepasst und erweitert werden.

16.1 Intialisierung

```
<canvas width="800" height="600" id="canvas"></canvas>
<script>
 var canvas = document.getElementById('canvas');
 var context = canvas.getContext('2d');
 var images = [new Image(), new Image(), new Image(), new Image()];
 var w = canvas.width,
     h = canvas.height;

 images[0].src =
'http://www.w3.org/html/logo/downloads/HTML5_Badge_512.png';
 images[1].src =
'http://www.w3.org/html/logo/downloads/HTML5_Badge_128.png';
 images[2].src =
'http://www.w3.org/html/logo/downloads/HTML5_Badge_32.png';
</script>
```

16.2 Testfälle

Kleines Bild

```
context.drawImage(images[2], 0, 0);
```

Mittleres Bild

```
context.drawImage(images[1], 0, 0);
```

Großes Bild

```
context.drawImage(images[0], 0, 0);
```

17 Image Clipping

Diese Testreihe kann unter der URL http://jsperf.com/image-clipping abgerufen, wiederholt, angepasst und erweitert werden.

17.1 Initialisierung

```
<canvas width="800" height="600" id="canvas"></canvas>
<script>
 var canvas = document.getElementById('canvas');
 var context = canvas.getContext('2d');
 var w = canvas.width,
     h = canvas.height;
 var image = new Image();
 image.src =
'http://www.w3.org/html/logo/downloads/HTML5_Badge_256.png';
</script>
```

17.2 Testfälle

Kein Clipping

```
1 context.drawImage(image, 0, 0);
```

Teilweises Clipping

```
 1 context.save();
 2
 3 context.beginPath();
 4 context.moveTo(0, 0);
 5 context.lineTo(127, 0);
 6 context.lineTo(127, 127);
 7 context.lineTo(0, 127);
 8 context.closePath();
 9 context.clip();
10
11 context.drawImage(image, 0, 0);
12
13 context.restore();
```

Deckendes Clipping

```
 1 context.save();
 2
 3 context.beginPath();
 4 context.moveTo(0, 0);
 5 context.lineTo(255, 0);
 6 context.lineTo(255, 255);
 7 context.lineTo(0, 255);
 8 context.closePath();
 9 context.clip();
10
11 context.drawImage(image, 0, 0);
12
13 context.restore();
```

Leeres Clipping

```
context.save();

context.beginPath();
context.moveTo(256, 0);
context.lineTo(512, 0);
context.lineTo(512, 256);
context.lineTo(256, 256);
context.closePath();
context.clip();

context.drawImage(image, 0, 0);

context.restore();
```

Nur Clipping

```
context.save();

context.beginPath();
context.moveTo(256, 0);
context.lineTo(512, 0);
context.lineTo(512, 256);
context.lineTo(256, 256);
context.closePath();
context.clip();

context.restore();
```

18 Text API

Diese Testreihe kann unter der URL http://jsperf.com/canvas-text abgerufen, wiederholt, angepasst und erweitert werden.

18.1 Initialisierung

```
<canvas width="800" height="600" id="canvas"></canvas>
<script>
 var canvas = document.getElementById('canvas');
 var context = canvas.getContext('2d');
 var w = canvas.width,
     h = canvas.height;
</script>
```

18.2 Testfälle

stroke (kurz)

```
1 context.save();
2 context.font = '12px serif';
3 context.strokeText('Hello, World!', 100, 100);
4 context.restore();
```

fill (kurz)

```
1 context.save();
2 context.font = '12px serif';
3 context.fillText('Hello, World!', 100, 100);
4 context.restore();
```

stroke (groß)

```
1 context.save();
2 context.font = '60px serif';
3 context.strokeText('Hello, World!', 100, 100);
4 context.restore();
```

fill (groß)

```
1 context.save();
2 context.font = '60px serif';
3 context.fillText('Hello, World!', 100, 100);
4 context.restore();
```

stroke (lang)

```
1 context.save();
2 context.font = '12px serif';
3 context.strokeText('Lorem ipsum dolor sit amet, consectetur adipisici
elit, sed eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut
enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut
aliquid ex ea commodi consequat. Quis aute iure reprehenderit in voluptate
velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint obcaecat
cupiditat non proident, sunt in culpa qui officia deserunt mollit anim id
est laborum.', 100, 100);
4 context.restore();
```

fill (lang)

```
1 context.save();
2 context.font = '12px serif';
3 context.fillText('Lorem ipsum dolor sit amet, consectetur adipisici
elit, sed eiusmod tempor incidunt ut labore et dolore magna aliqua. Ut
enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut
aliquid ex ea commodi consequat. Quis aute iure reprehenderit in voluptate
velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint obcaecat
cupiditat non proident, sunt in culpa qui officia deserunt mollit anim id
est laborum.', 100, 100);
4 context.restore();
```

19 Culling

Diese Testreihe kann unter der URL http://jsperf.com/canvas-partial-redrawing abgerufen, wiederholt, angepasst und erweitert werden.

19.1 Initialisierung

```
<canvas width="800" height="600" id="canvas"></canvas>

<script>
 var canvas = document.getElementById('canvas');
 var context = canvas.getContext('2d');
 var w = canvas.width,
     h = canvas.height,
     i, c = 100;

 var circles = [],
     circle;

 for (i = 0; i < c; i++) {
  circle = {};
  circle.r = 50;
  circle.x = Math.random() * (w - 2 * circle.r) + circle.r;
  circle.y = Math.random() * (h - 2 * circle.r) + circle.r;
  circle.color = 'blue';
  circles.push(circle);
 }

 function drawCircle(circle) {
  context.save();
  context.beginPath();
  context.arc(circle.x, circle.y, circle.r, 0, 4 * Math.PI);
  context.fillStyle = circle.color;
  context.fill();
  context.restore();
 }

 // Setup Canvas by initially drawing scene
 for (i = 0; i < c; i++) {
  drawCircle(circles[i]);
 }
</script>
```

19.2 Testfälle

Komplett

```
i = Math.floor(Math.random() * c);
circle = circles[i];
circle.color = 'red';
4
context.clearRect(0, 0, w, h);
for (i = 0; i < c; i++) {
 drawCircle(circles[i]);
}
```

Clipping

```
var clipping = {};

i = Math.floor(Math.random() * c);
circle = circles[i];
circle.color = 'red';
clipping.x = circle.x - circle.r;
clipping.y = circle.y - circle.r;
clipping.w = clipping.h = circle.r * 2;

context.save();
context.beginPath();
context.moveTo(clipping.x, clipping.y);
context.lineTo(clipping.x, clipping.y + clipping.h);
context.lineTo(clipping.x + clipping.w, clipping.y + clipping.h);
context.lineTo(clipping.x + clipping.w, clipping.y);
context.clip();

context.clearRect(0, 0, w, h);
for (i = 0; i < c; i++) {
 drawCircle(circles[i]);
}
context.restore();
```

Culling und Clipping

```
var clipping = {};

i = Math.floor(Math.random() * c);
circle = circles[i];
circle.color = 'red';
clipping.x = circle.x - circle.r;
clipping.y = circle.y - circle.r;
clipping.w = clipping.h = circle.r * 2;

context.save();
context.beginPath();
context.moveTo(clipping.x, clipping.y);
context.lineTo(clipping.x, clipping.y + clipping.h);
context.lineTo(clipping.x + clipping.w, clipping.y + clipping.h);
context.lineTo(clipping.x + clipping.w, clipping.y);
context.clip();

context.clearRect(0, 0, w, h);
for (i = 0; i < c; i++) {
 circle = circles[i];
 var cx = circle.x - circle.r,
     cy = circle.y - circle.r,
     cw = circle.r * 2,
     ch = cw;
 if (clipping.x < (cx + cw) && (clipping.x + clipping.w) > cx
    && clipping.y < (cy + ch) && (clipping.y + clipping.h) > cy) {
  drawCircle(circle);
 }
}
context.restore();
```

20 Buffering

Diese Testreihe kann unter der URL http://jsperf.com/buffering abgerufen, wiederholt, angepasst und erweitert werden.

20.1 Initialisierung

```
<canvas width="800" height="600" id="canvas"></canvas>
<script>
 var canvas = document.getElementById('canvas');
 var context = canvas.getContext('2d');
 var w = canvas.width,
     h = canvas.height,
     i, c = 100;

 var circles = [],
     circle;

 for (i = 0; i < c; i++) {
  circle = {};
  circle.r = 50;
  circle.x = Math.random() * (w - 2 * circle.r) + circle.r;
  circle.y = Math.random() * (h - 2 * circle.r) + circle.r;
  circle.color = 'blue';
  circles.push(circle);
 }

 function drawCircle(circle) {
  context.save();
  context.beginPath();
  context.arc(circle.x, circle.y, circle.r, 0, 4 * Math.PI);
  context.fillStyle = circle.color;
  context.fill();
  context.restore();
 }

 // Setup Canvas by initially drawing scene
 for (i = 0; i < c; i++) {
  drawCircle(circles[i]);
 }

 var bufferCanvas = document.createElement('canvas'),
     bufferContext = bufferCanvas.getContext('2d'),
     pixelBuffer, imageBuffer = new Image(w, h);

 bufferCanvas.width = w;
 bufferCanvas.height = h;
</script>
```

20.2 Testfälle

Lese Pixelbuffer

```
1 pixelBuffer = context.getImageData(0, 0, w, h);
```

Lese Imagebuffer

```
1 imageBuffer.src = canvas.toDataURL();
```

Lese Canvasbuffer

```
1 bufferContext.drawImage(canvas, 0, 0, w, h);
```

Schreibe Pixelbuffer

```
1 context.putImageData(pixelBuffer, 0, 0);
```

Schreibe Imagebuffer

```
1 context.drawImage(imageBuffer, 0, 0);
```

Schreibe Canvasbuffer

```
1 context.drawImage(bufferCanvas, 0, 0);
```

Neu zeichnen

```
1 context.clearRect(0, 0, w, h);
2 for (i = 0; i < c; i++) {
3  drawCircle(circles[i]);
4 }
```