Develop High Performance Sites and Modern Apps with JavaScript and HTML5

Doris Chen Ph.D.
Developer Evangelist
Microsoft
doris.chen@microsoft.com
http://blogs.msdn.com/dorischen/
@doristchen

Who am I?

- Developer Evangelist at Microsoft based in **Silicon Valley, CA**
  - Blog: http://blogs.msdn.com/b/dorischen/
  - Twitter @doristchen
  - Email: doris.chen@microsoft.com
- Has over 15 years of experience in the software industry focusing on web technologies
- Spoke at O'Reilly OSCON, Fluent, HTML5 DevConf, JavaOne, Developer Week, WebVisions and worldwide meetups
- Doris received her Ph.D. from the University of California at Los Angeles (UCLA)

Agenda

- General Best Practices for JavaScript Development
- Game: Make it Run Faster
  - 5 principles
- Resources

General Best Practices for JavaScript Development

Tips & tricks that still work

- For safe dynamic content, use **innerHTML** to create your DOM
- Link CSS stylesheets at the top of the page, not at the bottom
- **Avoid inline** JavaScript and inline CSS styles
- Don’t parse **JSON** by hand, use JSON.parse
- Remove Duplicate, limit your library
- Build Session“50 performance tricks to make your HTML5 apps and sites faster”
  - http://channel9.msdn.com/Events/Build/2012/3-132

User innerHTML to Create your DOM

Use DOM Efficiently

```
function InsertUsername()
{
      document.getElementById('user').innerHTML =
userName;
}
```

Avoid Inline JavaScript

Efficiently Structure Markup

```
<html>
  <head>
      <script type="text/javascript">
           function helloWorld() {
                alert('Hello World!') ;
           }
      </script>
  </head>
  <body>
      …
   </body>
</html>
```

JSON Always Faster than XML for Data

XML Representation

```
<!DOCTYPE glossary PUBLIC "DocBook V3.1">
    <glossary><title>example glossary</title>
       <GlossDiv><title>S</title>
          <GlossList>
           <GlossEntry ID="SGML" SortAs="SGML">
              <GlossTerm>Markup Language</GlossTerm>
              <Acronym>SGML</Acronym>
              <Abbrev>ISO 8879:1986</Abbrev>
              <GlossDef>
                 <para>meta-markup language</para>
                 <GlossSeeAlso OtherTerm="GML">
                 <GlossSeeAlso OtherTerm="XML">
              </GlossDef>
            <GlossSee OtherTerm="markup">
        </GlossEntry>
     </GlossList>
    </GlossDiv>
</glossary>
```

JSON Representation

```
"glossary":{
    "title": "example glossary", "GlossDiv":{
        "title": "S", "GlossList": {
            "GlossEntry": {
                "ID": "SGML",
                "SortAs": "SGML",
                "GlossTerm": "Markup Language",
                "Acronym": "SGML",
                "Abbrev": "ISO 8879:1986",
                "GlossDef": {
                    "para": "meta-markup language",
                    "GlossSeeAlso": ["GML", "XML"] },
                "GlossSee": "markup" }
            }
        }
    }
```

Use Native JSON Methods

Write Fast JavaScript

Native JSON Methods

```
var jsObjStringParsed = JSON.parse(jsObjString);
var jsObjStringBack = JSON.stringify(jsObjStringParsed);
```

Remove Duplicate Code

Efficiently Structure Markup

```
<html>
   <head>
      <title>Test</title>
   </head>
   <body>
      …
      <script src="jquery.js" … ></script>
      <script src="myscript.js" … ></script>
      <script src="navigation.js" … ></script>
      <script src="jquery.js" … ></script>
   </body>
</html>
```

Remove Duplicate Code

Efficiently Structure Markup

52%

of the pages on the web
have duplicate code

Standardize on a Single Framework

Efficiently Structure Markup

```
<script src="jquery.js" … ></script>
<script src="prototype.js" … ></script>
<script src="dojo.js" … ></script>
<script src="animater.js" … ></script>
<script src="extjs.js" … ></script>
<script src="yahooui.js" … ></script>
<script src="mochikit.js" … ></script>
<script src="lightbox.js" … ></script>
<script src="jslibs.js" … ></script>
<script src="gsel.js" … ></script>
…
```

Don’t Include Script To Be Cool

Efficiently Structure Markup

```
<script src="facebookconnect.js" … ></script>
<script src="facebooklike.js" … ></script>
<script src="facebookstats.js" … ></script>
<script src="tweetmeme.js" … ></script>
<script src="tweeter.js" … ></script>
<script src="tweetingly.js" … ></script>
<script src="googleanalytics.js" … ></script>
<script src="doubleclick.js" … ></script>
<script src="monitor.js" … ></script>
<script src="digg.js" … ></script>
…
```

Power Consumption

Let it rest! Power efficiency can drain your users' battery and decrease satisfaction with your application

Game: How to Make it Run Faster

-Five principles to improve JavaScript performance of your app

High Five Yeah - Overview

Single Player

Single Page

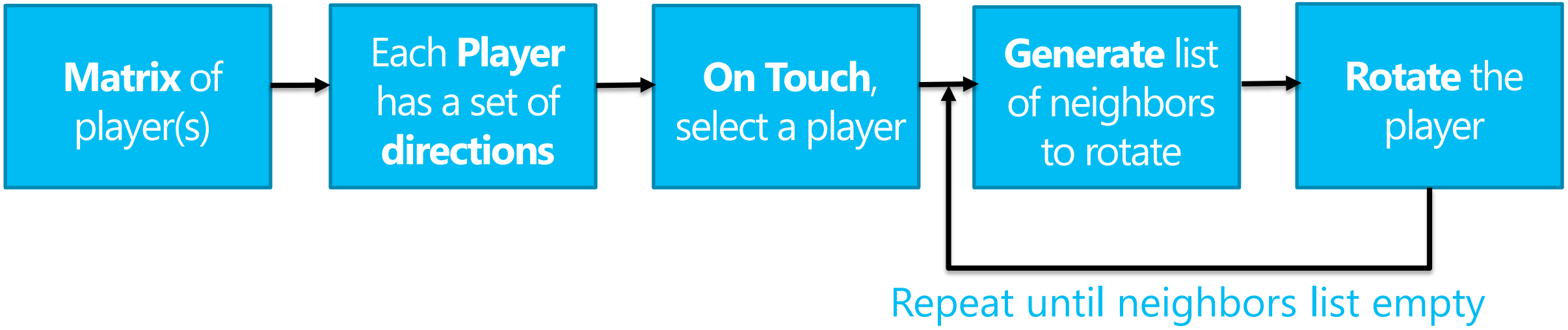
Casual Game

Raw JS code: http://aka.ms/FastJS

Demo

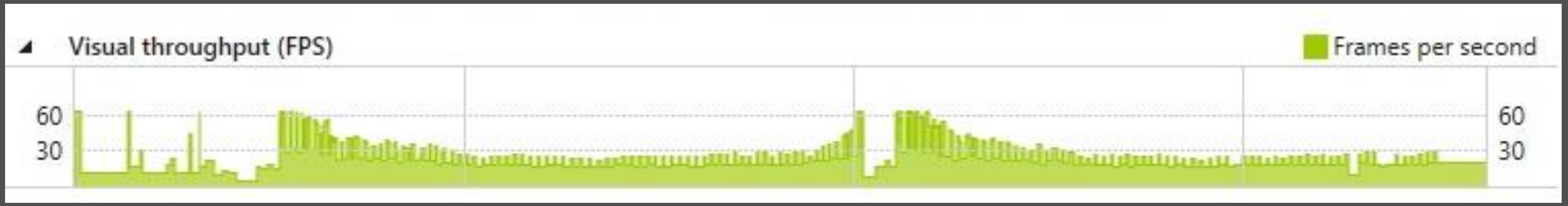
High Five

Components and control flow

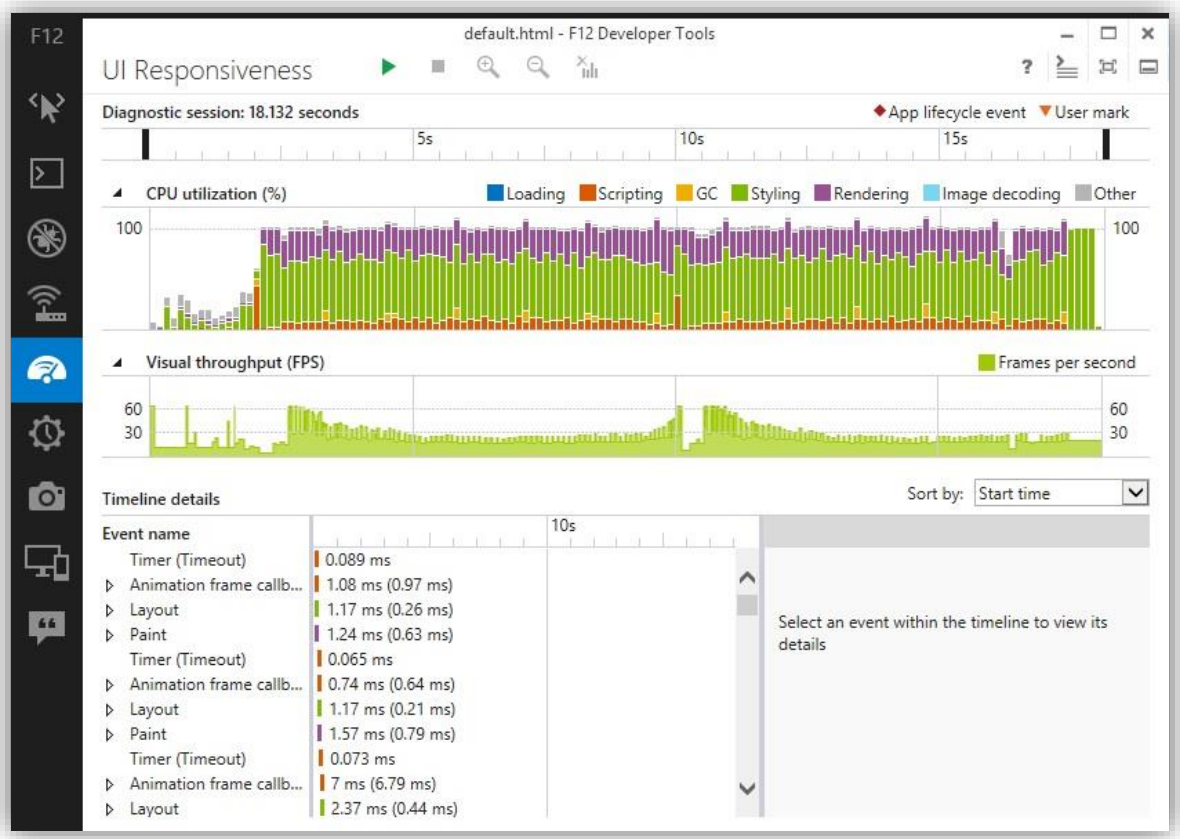


Arrays	**Objects and properties**	**Numbers**	**Animation loop**
Memory allocations			

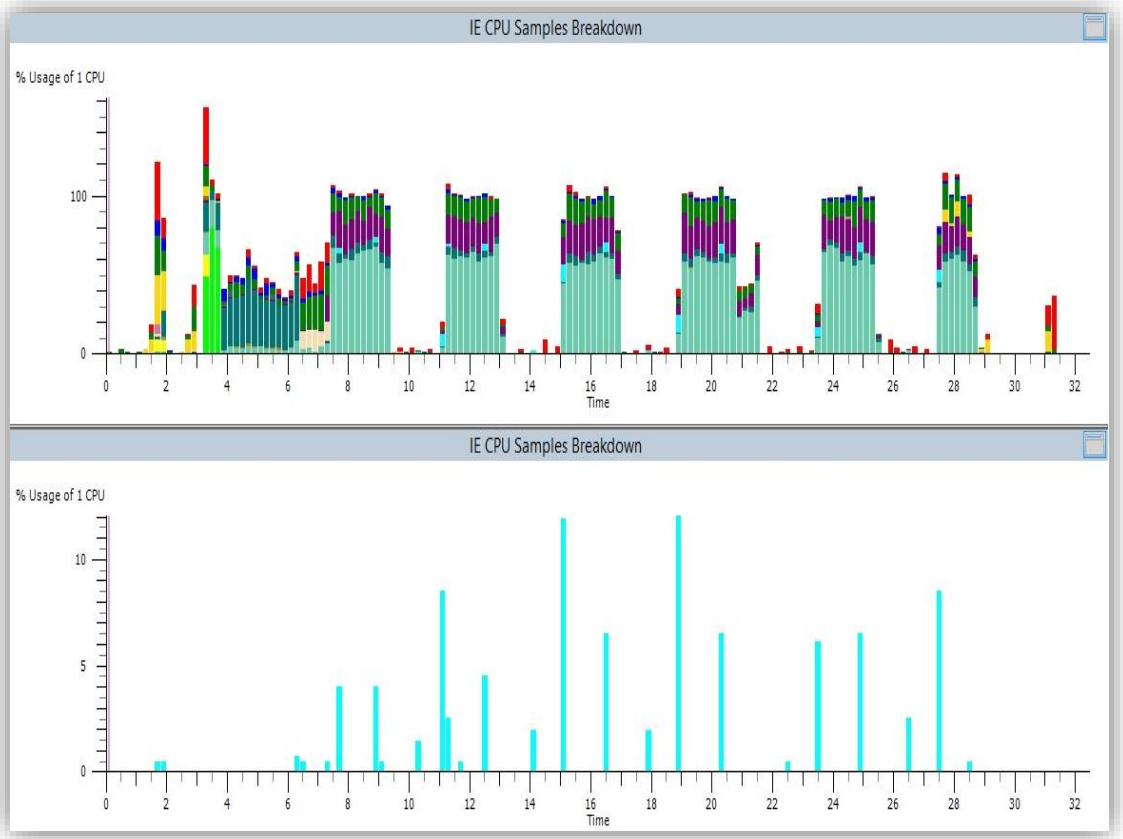
We've got a problem…



Always start with a good profiler

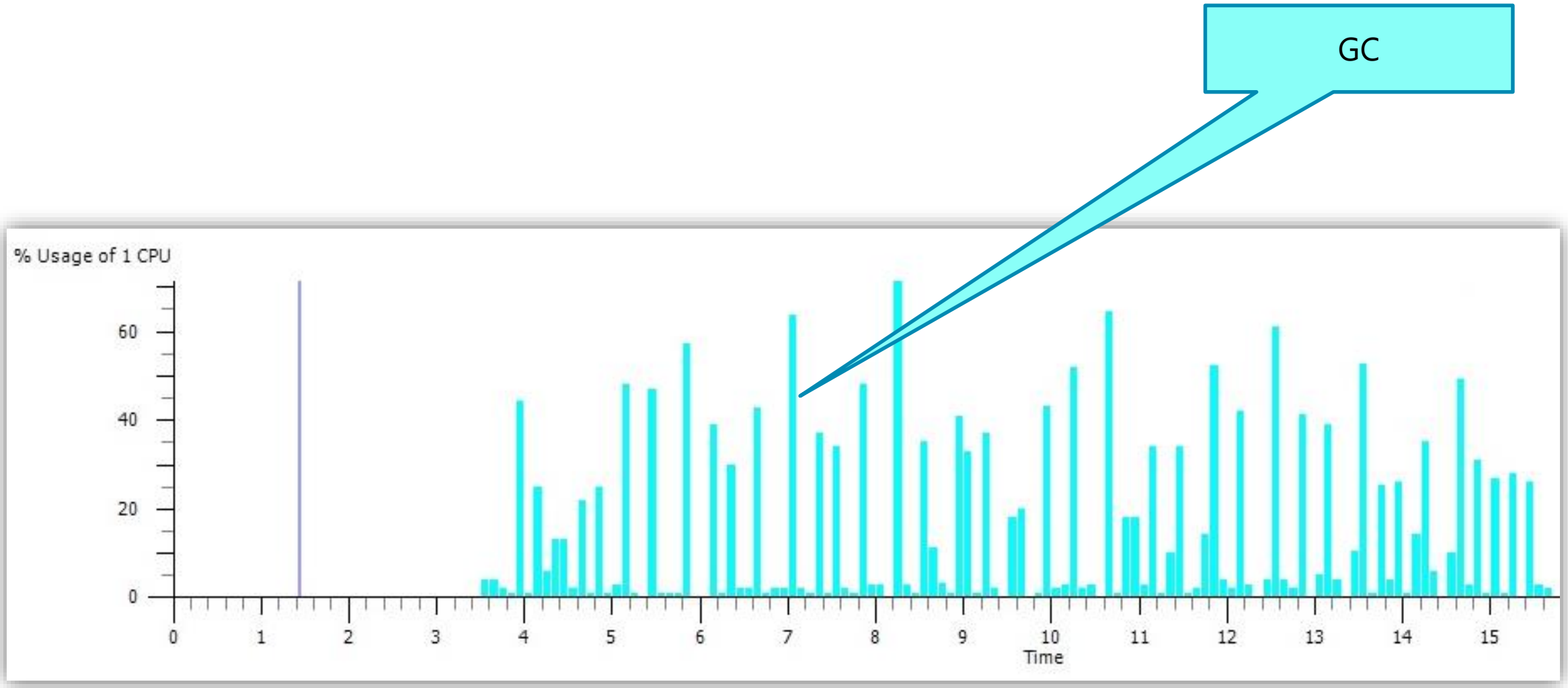


F12 UI Responsiveness Tool



Windows Performance Toolkit
http://aka.ms/WinPerfKit

Do we expect so much of GC to happen?



Principle #1:

Memory usage: Stay lean

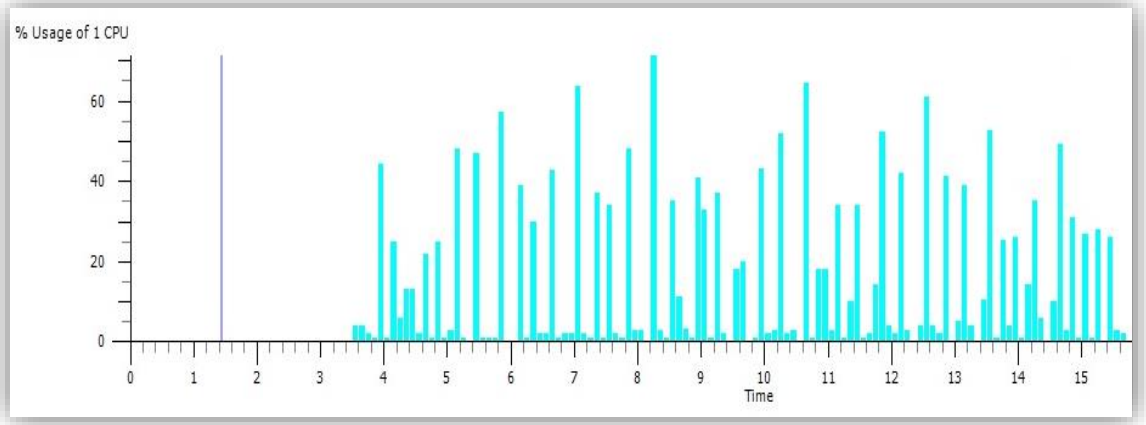
What triggers a garbage collection?

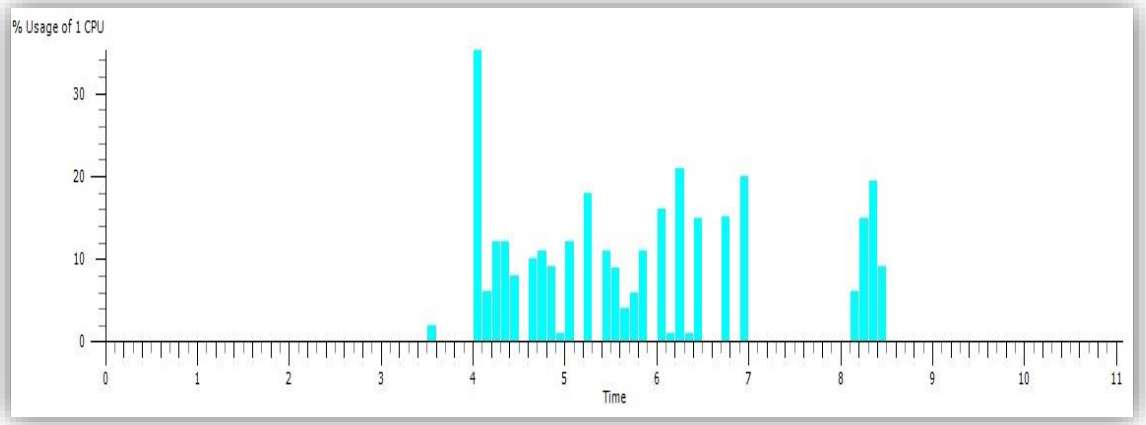
- Every call to `new` or implicit memory allocation reserves GC memory
  - Allocations are cheap until current pool is exhausted
- When the pool is exhausted, engines force a collection
  - Collections cause program pauses
  - Pauses could take milliseconds
- Be careful with object allocation patterns in your apps
  - Every allocation brings you closer to a GC pause

demo

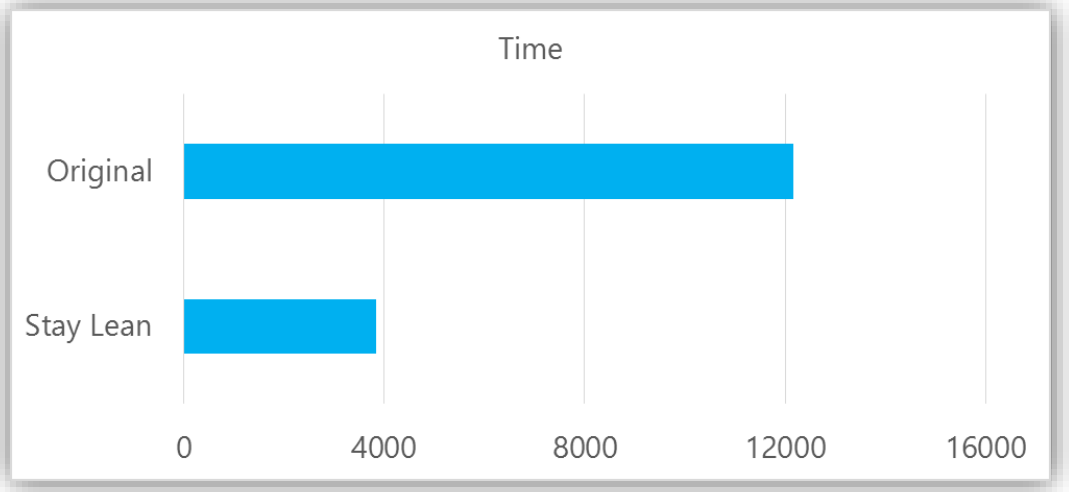
Fix It!

Results





- Overall FG GC time reduced to 1/3rd
- Raw JavaScript perf improved ~3x



Best practices for staying lean

- Avoid unnecessary object creation
- Use object pools, when possible
- Be aware of allocation patterns
  - Setting closures to event handlers
  - Dynamically creating DOM (sub) trees
  - Implicit allocations in the engine

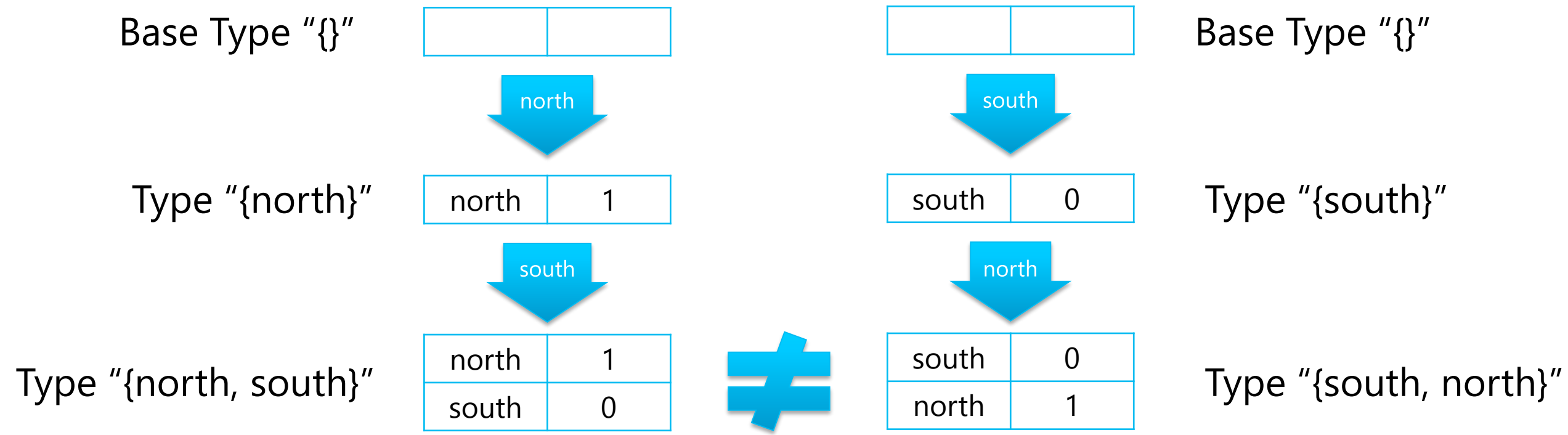
Principle #2

Use fast objects and manipulations

Internal Type System: Fast Object Types

```
var p1;
p1.north = 1;
p1.south = 0;
```

```
var p2;
p2.south = 0;
p2.north = 1;
```



Create fast types and avoid type mismatches

Don't add properties conditionally

```
function Player(direction) {
    if (direction = “NE”) {
        this.n = 1;
        this.e = 1;
    }
    else if (direction = “ES”) {
        this.e = 1;
        this.s = 1;
    }
    ...
}

var p1 = new Player(“NE”);      // p1 type {n,e}
var p2 = new Player(“ES”);      // p2 type {e,s}
```

p1.type != p2.type

```
function Player(north,east,south,west) {
    this.n = north;
    this.e = east;
    this.s = south;
    this.w = west;
}

var p1 = new Player(1,1,0,0);//p1 type {n,e,s,w}
var p2 = new Player(0,0,1,1);//p2 type {n,e,s,w}
```

p1.type == p2.type

Create fast types and avoid type mismatches

Don't default properties on prototypes

```
function Player(name) {
    ...
};

Player.prototype.n = null;
Player.prototype.e = null;
Player.prototype.s = null;
Player.prototype.w = null;

var p1 = new Player("Jodi");      //p1 type{}
var p2 = new Player("Mia");       //p2 type{}
var p3 = new Player("Jodi");      //p3 type{}

p1.n = 1;                         //p1 type {n}
p2.e = 1;                         //p2 type {e}
```

p1.type != p2.type != p3.type

```
function Player(name) {
  this.n = null;
  this.e = null;
  this.s = null;
  this.w = null;
  ...
}


var p1 = new Player("Jodi");  //p1 type{n,e,s,w}
var p2 = new Player("Mia");   //p2 type{n,e,s,w}
var p3 = new Player("Jodi");  //p3 type{n,e,s,w}

p1.n = 1;                     //p1 type{n,e,s,w}
p2.e = 1;                     //p2 type{n,e,s,w}
```

p1.type == p2.type == p3.type

Avoid conversion from fast type to slower property bags

Deleting properties forces conversion

```
function Player(north,east,south,west) {
  this.n = north;
  this.e = east;
  this.s = south;
  this.w = west;
}
var p1 = new Player();

delete p1.n;
```

SLOW

```
function Player(north,east,south,west) {
  this.n = north;
  this.e = east;
  this.s = south;
  this.w = west;
}
var p1 = new Player();

p1.n = 0;              // or undefined
```

FAST

Avoid creating slower property bags

Add properties in constructor, restrict total properties

```
function Player() {
  this.prop01 = 1;
  this.prop02 = 2;
  ...
  this.prop256 = 256;
  ...              // Do you need an array?
}

var p1 = new Player();
```

SLOW

```
function Player(north,east,south,west) {
    this.n = north;
    this.e = east;
    this.s = south;
    this.w = west;
    ...    // Restrict to few if possible
}

var p1 = new Player();
```

FAST

Avoid creating slower property bags

Restrict using getters, setters and property descriptors in perf critical paths

```
function Player(north, east, south, west) {
    Object.defineProperty(this, "n", {
        get : function() { return nVal; },
        set : function(value) { nVal=value; },
        enumerable: true, configurable: true
    });
    Object.defineProperty(this, "e", {
        get : function() { return eVal; },
        set : function(value) { eVal=value; },
        enumerable: true, configurable: true
    });
    ...
}
var p = new Player(1,1,0,0);
var n = p.n;
p.n = 0;
...
```

SLOW

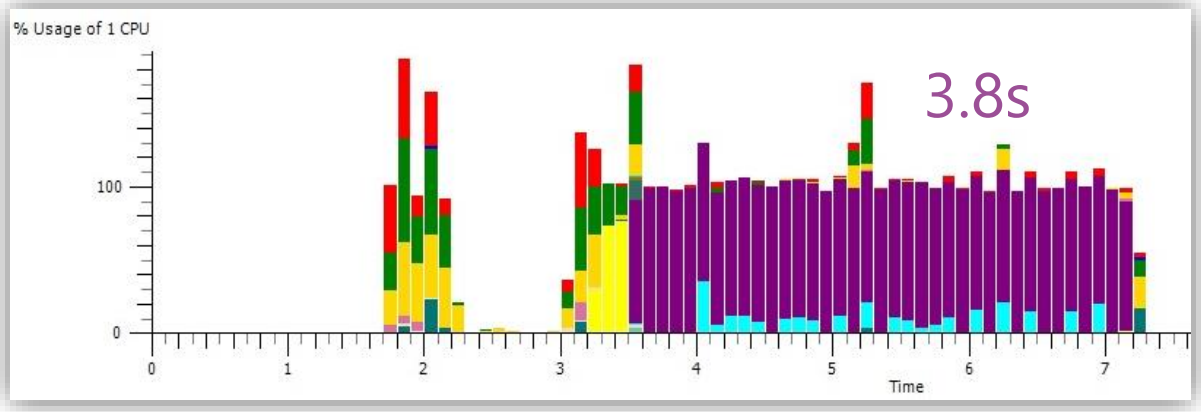
```
function Player(north, east, south, west) {
  this.n = north;
  this.e = east;
  this.s = south;
  this.w = west;
  ...
}

var p = new Player(1,1,0,0);
var n = p.n;
p.n = 0;
...
```

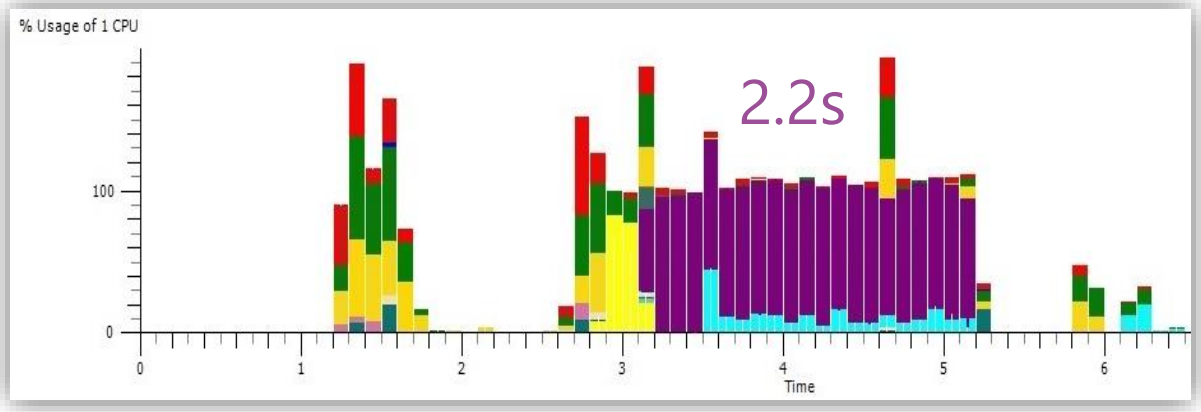
FAST

demo

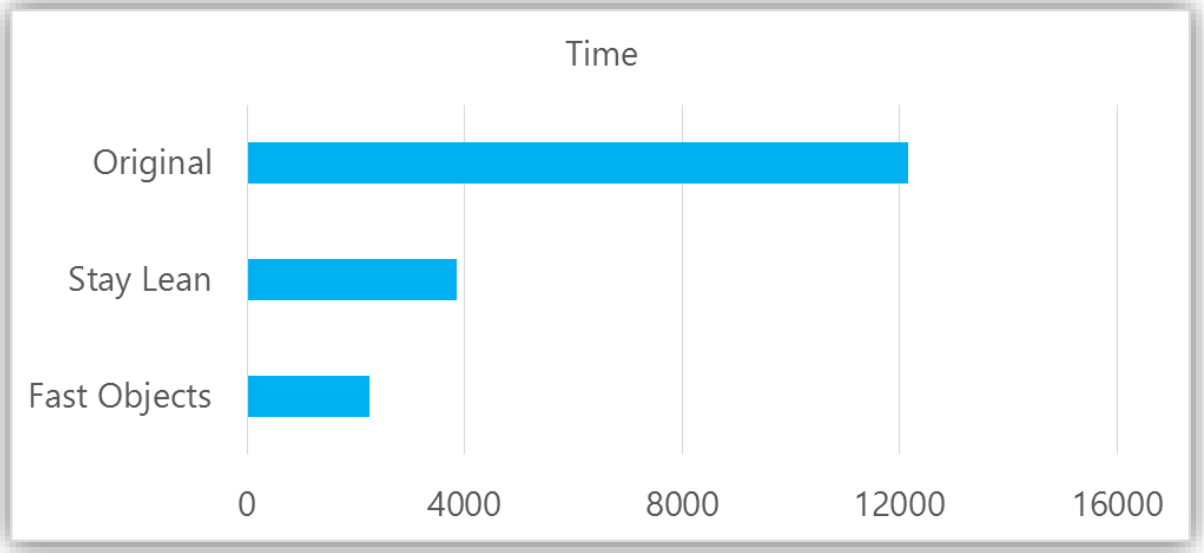
Fix It!

Results





- Time in script execution reduced ~30%
- Raw JS performance improved ~30%



Best practices for fast objects and manipulations

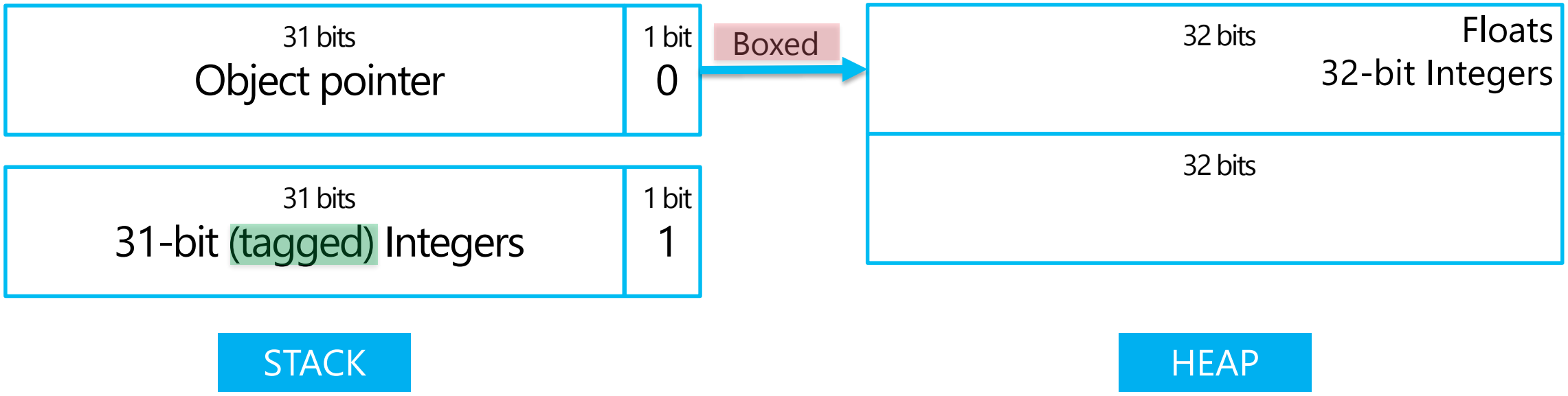
- Create and use fast types
- Keep shapes of objects consistent
- Avoid type mismatches for fast types

Principle #3

Use fast arithmetic

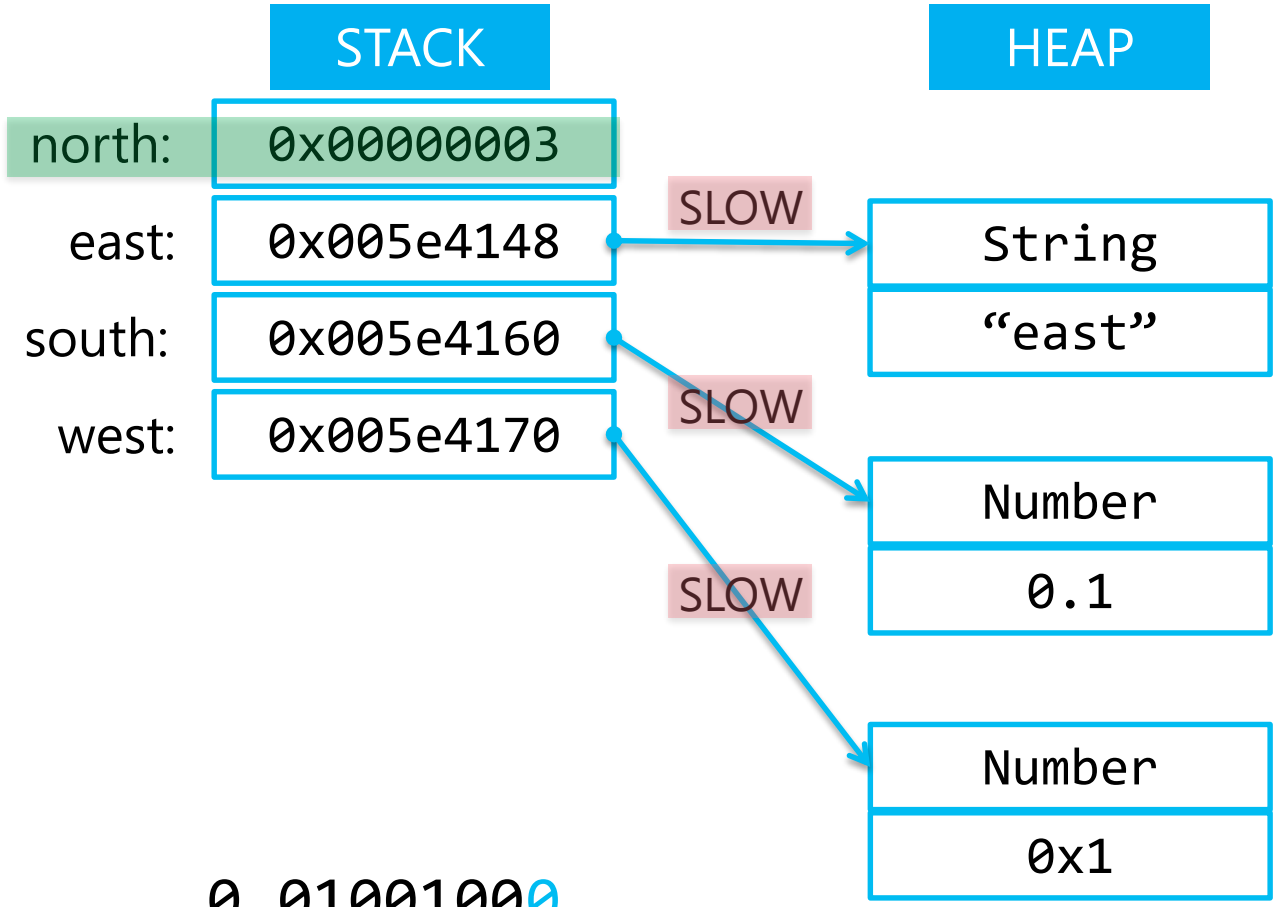
Numbers in JavaScript

- All numbers are IEEE 64-bit floating point numbers
  - Great for flexibility
  - Performance and optimization challenge



Use 31-bit integers for faster arithmetic

```
var north = 1;

var east = "east";
var south = 0.1;
var west = 0x1;


function Player(north, south, east, west)
{
    ...
}

var p = new Player(north,south,east,west);
```




```
0x005e4148:          0…01001000
0x03 represents 1:   0…00000011
```

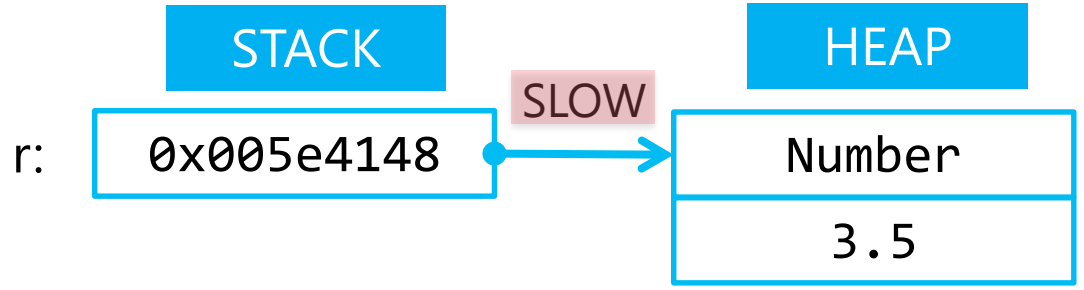
Avoid creating floats if they are not needed

Fastest way to indicate integer math is |0

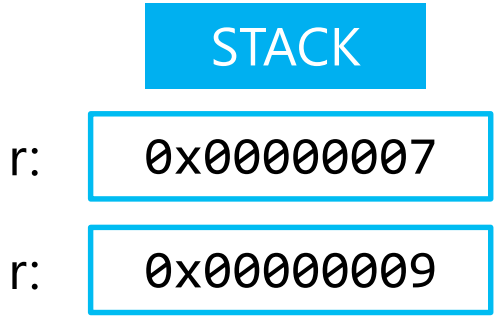
```
var r = 0;

function doMath(){
  var a = 5;
  var b = 2;
  r = ((a + b) / 2);                    // r = 3.5
}
...
var intR = Math.floor(r);
```

```
var r = 0;

function doMath(){
  var a = 5;
  var b = 2;
  r = ((a + b) / 2) |0 ;                // r = 3
  r = Math.round((a + b) / 2);          // r = 4
}
```

SLOW



FAST



Take advantage of type-specialization for arithmetic

Create separate functions for ints and floats; use consistent argument types

```
function Distance(p1, p2) {
  var dx = p1.x - p2.x;
  var dy = p1.y - p2.y;
  var d2 = dx * dx + dy * dy;
  return Math.sqrt(d2);
}



var point1 = {x:10, y:10};
var point2 = {x:20, y:20};
var point3 = {x:1.5, y:1.5};
var point4 = {x:0x0AB, y:0xBC};



Distance(point1, point3);
Distance(point2, point4);
```

SLOW

```
function DistanceFloat(p1, p2) {
  var dx = p1.x - p2.x;
  var dy = p1.y - p2.y;
  var d2 = dx * dx + dy * dy;
  return Math.sqrt(d2);
}
function DistanceInt(p1,p2) {
  var dx = p1.x - p2.x;
  var dy = p1.y - p2.y;
  var d2 = dx * dx + dy * dy;
  return (Math.sqrt(d2) | 0);
}
var point1 = {x:10, y:10};
var point2 = {x:20, y:20};
var point3 = {x:1.5, y:1.5};
var point4 = {x:0x0AB, y:0xBC};

DistanceInt(point1, point2);
DistanceFloat(point3, point4);
```

FAST

Best practices for fast arithmetic

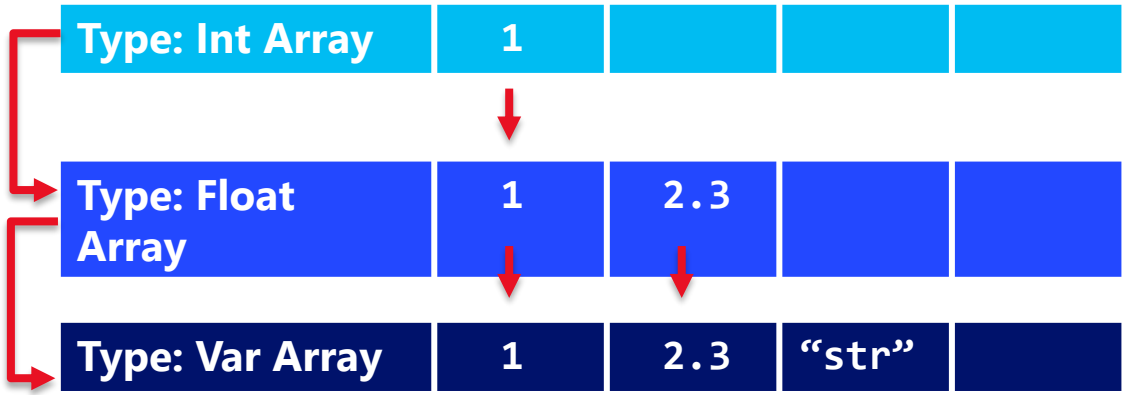
- Use 31-bit integer math when possible
- Avoid floats if they are not needed
- Design for type specialized arithmetic

Principle #4

Use fast Arrays

Array internals

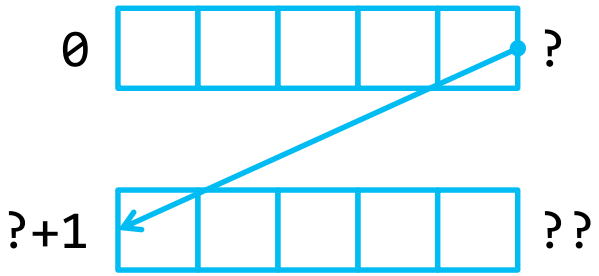
```
01  var a = new Array();
02  a[0] = 1;
03  a[1] = 2.3;
04  a[2] = “str”;
```



Pre-allocate arrays

```
var a = new Array();
for (var i = 0; i < 100; i++) {
  a.push(i + 2);
}
```

SLOW




```
var a = new Array(100);
for (var i = 0; i < 100; i++) {
  a[i] = i + 2;
}
```

FAST

For mixed arrays, provide an early hint

Avoid delayed type conversion and copy

```
var a = new Array(100000);



for (var i = 0; i < a.length; i++) {
  a[i] = i;
}
...
//operations on the array
...
a[99] = “str”;
```

SLOW

```
var a = new Array(100000);

a[0] = “hint”;

for (var i = 0; i < a.length; i++) {
  a[i] = i;
}
...
//operations on the array
...
a[99] = “str”;
```

FAST

Use Typed Arrays when possible

Avoids tagging of integers and allocating heap space for floats

```
var value = 5;


var a = new Array(100);
a[0] = value;              // 5      - tagged
a[1] = value / 2;          // 2.5    - boxed
a[2] = "text";             // "text" – var array
```

SLOW

```
var value = 5;

var a = new Float64Array(100);
a[0] = value;          // 5   - no tagging required
a[1] = value / 2;      // 2.5 - no boxing required
a[2] = "text";         // 0

var a = new Int32Array(100);
a[0] = value;          // 5 - no tagging required
a[1] = value / 2;      // 2 - no tagging required
a[2] = "text";         // 0
```

FAST

Keep values in arrays consistent

Numeric arrays treated like Typed Arrays internally

```
var a = [1,0x2,99.1,5];          //mixed array
var b = [0x10,8,9];              //mixed array

function add(a,i,b,j)
{
  return a[i] + b[j];
}

add(a,0,b,0);
add(a,1,b,1);
```

SLOW

```
var a = [1,5,8,9];               //int array
var b = [0x02,0x10,99.1];        //float
array

function add(a,i,b,j)
{
  return a[i] + b[j];
}

add(a,0,b,0);
add(a,1,b,1);
```

FAST

Keep arrays dense

Deleting elements can force type change and de-optimization

```
var a = new Array(1000);          //type int
...
for (var i = 0; i < boardSize; i++) {
    matrix[i]  = [1,1,0,0];
}

//operating on the array
...
delete matrix[23];
...
//operating on the array
```

SLOW

```
var a = new Array(1000);          //type int
...
for (var i = 0; i < boardSize; i++) {
    matrix[i]  = [1,1,0,0];
}

//operating on the array
...
matrix[23] = 0;
...
//operating on the array
```

FAST

Enumerate arrays efficiently

Explicit caching of length avoids repetitive property accesses

```
var a = new Array(100);
var total = 0;

for (var item in a) {
  total += item;
};

a.forEach(function(item){
  total += item;
});

for (var i = 0; i < a.length; i++) {
  total += a[i];
}
```

SLOW

```
var a = new Array(100);
var total = 0;


cachedLength = a.length;


for (var i = 0; i < cachedLength; i++) {
  total += a[i];
}
```

FAST

Best practices for using arrays efficiently

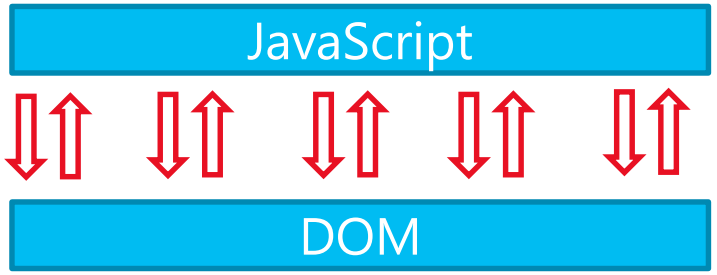
- Pre-allocate arrays
- Keep array type consistent
- Use typed arrays when possible
- Keep arrays dense
- Enumerate arrays efficiently

Principle #5

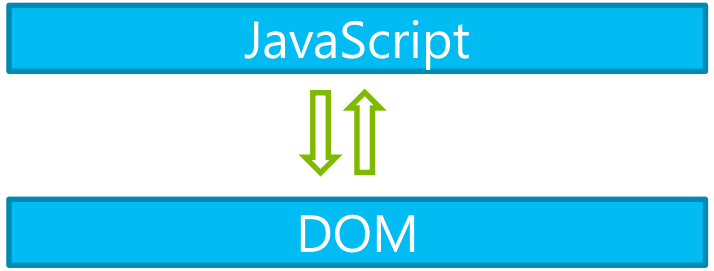
Do less (cross-subsystem) work

Avoid chattiness with the DOM

```
...
//for each rotation
document.body.game.getElementById(elID).classList.remove(oldClass)
document.body.game.getElementById(elID).classList.add(newClass)
...
```




```
var element = document.getElementById(elID).classList;

//for each rotation
element.remove(oldClass)
element.add(newClass)
...
```



Avoid automatic conversions of DOM values

Values from DOM are strings by default

```
this.boardSize = document.getElementById("benchmarkBox").value;

for (var i = 0; i < this.boardSize; i++) {        //this.boardSize is
“25”
  for (var j = 0; j < this.boardSize; j++) {      //this.boardSize is
“25”
      ...
  }
}
```

SLOW

```
this.boardSize = parseInt(document.getElementById("benchmarkBox").value);

for (var i = 0; i < this.boardSize; i++) {         //this.boardSize is 25
  for (var j = 0; j < this.boardSize; j++) {       //this.boardSize is 25
      ...
  }
}
```

FAST
(25% marshalling cost reduction in init function)

Paint as much as your users can see

Align timers to display frames

```
setInterval(animate, 0);
setTimeout(animate,  0);
```

MORE WORK

```
requestAnimationFrame(animate);

setInterval(animate, 1000 / 60);
setTimeout(animate,  1000 / 60);
```

LESS WORK

demo

Fix It!

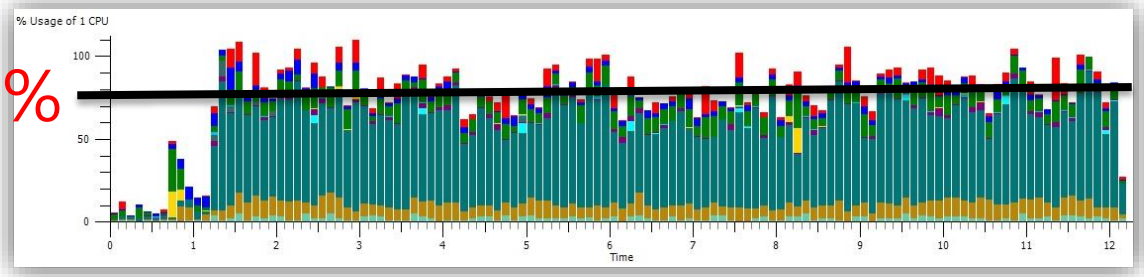
Results

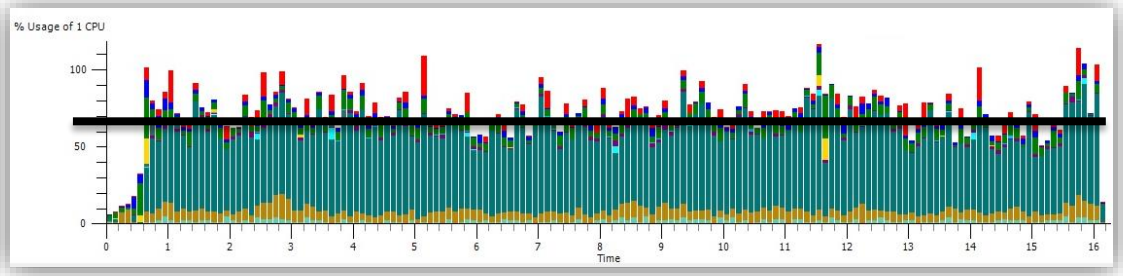
Save CPU cycles

```
setTimeout(animate,  0);
```

```
requestAnimationFrame(animate);
```

75%





65%

Problem solved…

Why performance matters?

- Great user experience
- Longer battery life
- Make your apps richer

Optimized JS code: http://aka.ms/FasterJS

In-review: Writing fast sites & apps with JavaScript

Understand and target modern engines

- *Principle#1: Stay lean – use less memory*
- *Principle#2: Use fast objects and do fast manipulations*
- *Principle#3: Write fast arithmetic*
- *Principle#4: Use fast arrays*
- *Principle#5: Do less (cross-subsystem) work*

Resources

Resources

Overview Concepts

- High Performance Websites
  Steve Souders, September 2007
- Event Faster Websites: Best Practices
  Steve Souders, June 2009
- High Performance Browser Networking
  Ilya Grigorik, September 2013

JavaScript Patterns

- High Performance JavaScript
  Nicholas Zakas, March 2010
- JavaScript the Good Parts
  Douglas Crockford, May 2008
- JavaScript Patterns
  Stoyan Stefanov, September 2010
- JavaScript Cookbook
  Shelley Powers, July 2010

Microsoft Guidance

- Windows Store App: JavaScript Best Practices
- Internet Explorer Architectural Overview

W3C Web Performance

- Web Performance Working Group Homepage
- Navigation Timing Specification

Blog Posts

- Key Advances to JavaScript Performance in Windows 10
- Measuring Browser Performance in Lab Environments
- What Common Benchmarks Measure

Tools

- Debugging/Tuning Browser Performance with the Windows Performance Tools
- How to use F12 tool

HTML5 Resources

- Responsive Web Design and CSS3
  - http://bit.ly/CSS3Intro
- HTML5, CSS3 Free 1 Day Training
  - http://bit.ly/HTML5DevCampDownload
- Using Blend to Design HTML5 Windows 8 Application (Part II): Style, Layout and Grid
  - http://bit.ly/HTML5onBlend2
- Using Blend to Design HTML5 Windows 8 Application (Part III): Style Game Board, Cards, Support Different Device, View States
  - http://bit.ly/HTML5onBlend3
- Feature-specific demos
  - http://ie.microsoft.com/testdrive/
- Real-world demos
  - http://www.beautyoftheweb.com/