**Uniwersytet Jagielloński**

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Instytut Informatyki Stosowanej

Praca magisterska

Sebastian Poręba

# Porównanie silników fizyki 3D

Promotor:

dr hab. Paweł Węgrzyn

Kraków 2013

## OŚWIADCZENIE AUTORA PRACY

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie, i nie korzystałem ze źródeł innych niż wymienione w pracy.

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

PODPIS

**UJ**
**Jagiellonian University in Krakow**

Faculty of Phycics, Astronomy and Applied Computer Science
Department of Applied Computer Science

Master of Science Thesis

Sebastian Poręba

# Comparison of 3D physics engines

Supervisor:
Paweł Węgrzyn Ph.D

Krakow 2013

Serdecznie podziękowania dla ...

# Spis treści

# 1. Introduction

The main objective of presented project is the implementation of parts of 3D engine in a browser environment. Parts of engine are analysed side-by-side with parallel engine compiled from C++. The objective of analysis is comparison of performance and description of possible issues related to the limited browser resources and dynamic features of JavaScript.

Browser-based engine is implemented in JavaScript and analyzed in V8 engine. V8 is maintained by Google and is used in Google Chrome browser. Executable examples are compiled using gcc compiler and are runned on the same platform. For additional comparison EmScripten project is used to automatically generate JavaScript to measure if automated conversion may be as effective as writing code by hand.

Project is based on conference sessions and announcements authored by V8 programmers regarding performance of JavaScript applications. Analysis of available materials is a topic of Chapter 2, where internals of modern engines for dynamic languages are briefly explained. Results of existing works are reproduced and measured to build a base for extension.

Chapter 3 covers particles system found often in graphic engines. It shows techniques related to memory allocation and garbage collection that help to improve performance.

Chapter 4 is focused on sphere collision detection and reaction.

Chapter 5 shows complex object collision.

Chapter 6 recreates previous examples using EmScripten and compares automatically generated code with JavaScript implementation.

# 2. Overview of JavaScript and V8 engine architecture

At present JavaScript is only language widely supported by all browsers. With all it's advantages and quirks is the only choice available for programmers. [1] It was developed in 1995 as NetScape browser's solution for more dynamic web pages.

Historically, JavaScript was considered untyped language, meaning that values had no types attached to variables, either by programmer or compiler. All variables were of single, unified type and procedures called unboxing and boxing, performed before and after each operation on variable, ensured that it was properly used on machine code level. Complete code source was sent from server to the browser and was parsed and executed on fly. Without types attached to variables, all functions were polymorphic and unstable, since parameters may have carried any type of variable. To solve this problem source code of function was parsed each time it was called, each time generating machine code based on current parameters and variables in scope. This approach, called interpretation, is still present in JavaScript engines and used whenever variables don't match set criteria of stability described later in this chapter.

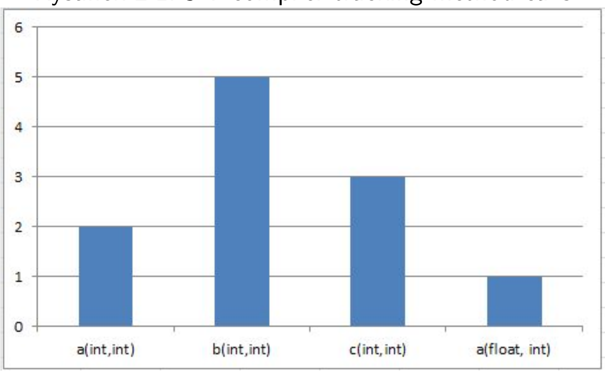
This paper uses as an engine of choice V8 Crankshaft. Choice was made because it's the only engine available at the moment which provides direct command line access, enabling precise performance measurements of code parsing and execution, without browser context and overhead. Executable file of V8 (named d8) is compiled with consideration of target platform.

[1] Currently two new languages are worked on - Dart by Google and TypeScript by MicroSoft. However, to enable cross-compilation to JavaScript, paradigm of these languages is similar and work is focused mainly on better IDE support.

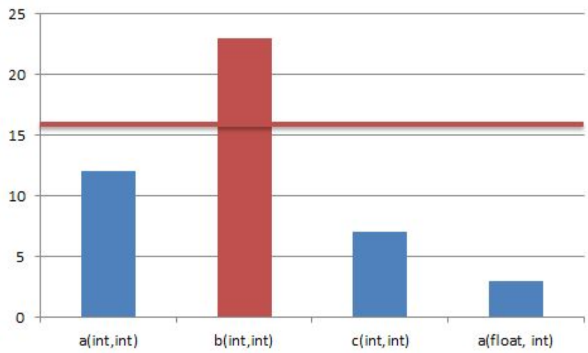
## 2.1. JIT compilation - tracking variable types

As mentioned before, initially JavaScript was treated as untyped language. With release of SpiderMonkey in Firefox 3.5 in 2009 situation has changed. First Just-In-Time compiler for JavaScript, TraceMonkey, was created. Based on works of Prof. Dr. Michael Franz on TraceTrees [2] JIT compiler was collecting all paths that interpreter took with specific types of variables. A path could split to different methods or if statements. Whenever part of code was executed often enough, path was marked as hot and compiler optimised it for given types. If single path was traversed with different set of types compiler could generate another version of optimised code. When path tuned out to be highly polymorphic optimised versions were removed and interpreter was used as a fallback. Initial reports shown speedups between 20x to 40x [3]. However, trace JIT turned out to be very complicated to maintain [4] and eventually was removed from Firefox in 2011.[5]. At the time SpiderMonkey was already equipped with JägerMonkey, JIT engine based on method calls. Instead of collecting complete traces, only method calls are counted. This gives easy track of function parameters and variables in scope.

Rysunek 2.1: JIT compiler tracking method calls




Rysunek 2.2: JIT compiler marking one of methods as hot and recompiling




This proved to be more effective and simpler approach and now used in all JavaScript engines. In V8 Crankshaft step forward was taken and simple methods are compiled even before any statistics on data types are collected. For compiled methods source code is not stored. Instead procedure called deoptimisation is implemented. Whenever engine detects that compiled code doesn't match actual types of variables, code is

[2] http://www.michaelfranz.com/
[3] http://arstechnica.com/information-technology/2008/08/firefox-to-get-massive-javascript-performance-boost/
[4] https://hacks.mozilla.org/2010/03/improving-javascript-performance-with-jagermonkey/
[5] http://blog.mozilla.org/nnethercote/2011/11/23/memshrink-progress-report-week-23/

deoptimised and either optimised again to match new, better set of variables, or kept in interpreter friendly form.

To track these changes two debug options for V8 are available: –trace-opt and –trace-deopt.

```
[marking Point.setX 0x2d6ecb87e568 for recompilation,
reason: small function, ICs with typeinfo: 1/1 (100%)]
[marking Point.setY 0x2d6ecb87e5b0 for recompilation,
reason: small function, ICs with typeinfo: 1/1 (100%)]

[optimizing: Point.setY / 2d6ecb87e5b1 - took 0.037, 0.047, 0.000 ms]
[optimizing: Point.setX / 2d6ecb87e569 - took 0.021, 0.038, 0.000 ms]

[marking Point 0x2d6ecb87e448 for recompilation,
reason: small function, ICs with typeinfo: 0/0 (100%)]
[marking dot 0x2d6ecb87e490 for recompilation,
reason: small function, ICs with typeinfo: 7/7 (100%)]

[optimizing: Point / 2d6ecb87e449 - took 0.004, 0.019, 0.000 ms]
[optimizing: dot / 2d6ecb87e491 - took 0.013, 0.057, 0.000 ms]

**** DEOPT: dot at bailout #2, address 0x0, frame size 0
[deoptimizing: begin 0x2d6ecb87e491 dot @2]
  translating dot => node=3, height=0
[deoptimizing: end 0x2d6ecb87e491 dot => node=3, pc=0x98518d30ac6, state=NO_REGISTERS,
  alignment=no padding, took 0.146 ms]
[removing optimized code for: dot]
```

Listing 2.1: Output from V8 debug run showing optimisation and deoptimisation

## 2.2. Type interference

V8's method of optimising code before it's run relies on type inference. Based on context of variable it's type is guessed. Generated assembler has to support cache miss - whenever inferred type turns out to be incorrect, new type is assigned and another JIT compilation runs. Types of variables are organised in a tree, where Number object may store both Float or Integer, Integer may store SMI (small int), etc.

```
//         Unknown
//           |   \____________
//           |                |
//      Primitive       Non-primitive
//           |   \_______     |
//           |           |    |
//        Number       String |
//         /   \         |    |
//    Double  Integer32  |   /
//        |      |      /   /
//        |     Smi    /   /
//        |      |    / __/
//        Uninitialized.
```

Listing 2.2: Tree of types in JavaScript

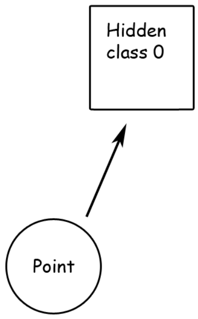
In V8 type inference is tightly connected with JIT compilation and may be tracked with the same flags: –trace-opt and –trace-deopt.

## 2.3. Hidden classes

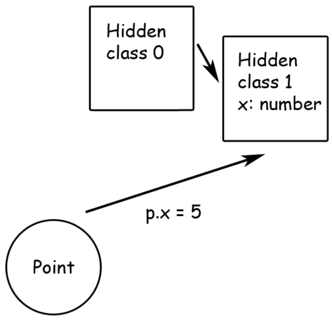
TODO: add paragraph on dictionary mode in objects.

JavaScript is classless language. Object may have defined a prototype which behaves similar to base class in other langauges. However, a property may be added to an Object or it's prototype at any point in runtime. To optimise such dynamic representation engines use a concept of hidden class. Whenever an Object is created it's hidden class is pointed to base, empty Object representation. Then each definition of new property makes a transition on hidden class graph, introducing hidden classes that are not yet defined, as in following example:

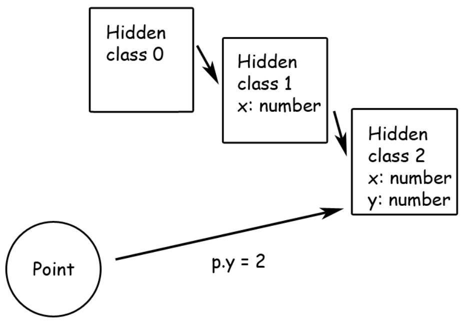
Rysunek 2.3: Initial shape of hidden class for Point




Rysunek 2.4: Shape of hidden class for Point after x property added

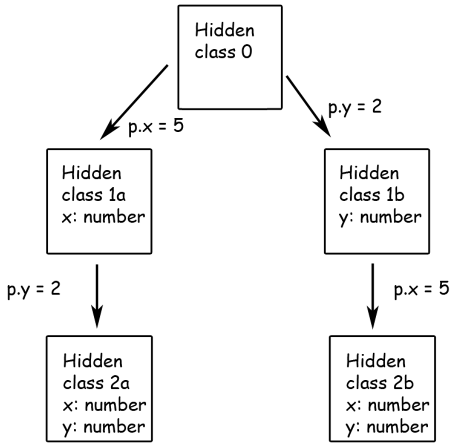



Rysunek 2.5: Shape of hidden class for Point after y property added




Based on hidden class further JIT compiler optimises methods, to generate even simpler assembly code similar to one compiled from C++. Class shape defines address offsets of Object properties. Thus, hidden class graph is actually a tree, where one class can't be reached in more than one way.

Rysunek 2.6: Two point representations based on order of declared properties




At the moment of writing type of property is not tracked in hidden classes. The only exception are primitive values (see Listing 2.2). In other words, storing an object in property results in the same hidden class regardless of hidden class of this object.

Transitions between hidden classes can be tracked in V8 using flags –trace-generalization tracking when variables are casted to more generic type (e.g. SMI to Integer, or Integer to Number) and –trace-migration (tracking when hidden classes are migrated).

```
// TODO: update when it lands in V8

[generalizing xQ] I:s->d (+20 maps) [xQ.Kd+919 at :719]
[generalizing xQ] Si:s->d (+3 maps) [xQ.Kd+1057 at :719]
[migrating xQ] I:s->d Si:s->d
[migrating xQ] I:s->d Si:s->d
```

Listing 2.3: Log of migration trace in V8

## 2.4. Garbage collection

Memory in JavaScript is managed automatically. Each allocation puts an object on memory heap. First generation of garbage collection traversed the whole tree and freed memory for all inaccessible objects. This type of deallocation is called mark-sweep and is causing taking a long time. Since JavaScript is single-threaded, this operation is blocking all other operations. To improve performance, especially in games, incremental scavange method of garbage collection was introduced. Engine tracks age of objects, allowing to quickly detect objects allocated temporarily (e.g. for a single frame rendered in game). When object is inaccessible, it's queued for deallocation, in chunks that don't cause long UI freezes. [6] [7]

TODO: check and extent

```
[1592]       34 ms: Scavenge 1.6 (18.8) -> 0.9 (18.8) MB, 0.0 ms [Runtime::PerformGC].
[1592]       37 ms: Scavenge 1.6 (18.8) -> 1.2 (19.8) MB, 1.0 ms [Runtime::PerformGC].
[1592]       40 ms: Scavenge 1.9 (19.8) -> 1.7 (19.8) MB, 1.0 ms [Runtime::PerformGC].
[1592]       43 ms: Scavenge 2.4 (19.8) -> 2.2 (19.8) MB, 2.0 ms [Runtime::PerformGC].
[1592]       49 ms: Scavenge 3.7 (19.8) -> 3.3 (20.8) MB, 3.0 ms [Runtime::PerformGC].
[1592]       56 ms: Scavenge 4.8 (20.8) -> 4.3 (21.8) MB, 3.0 ms [Runtime::PerformGC].
[1592]       74 ms: Scavenge 7.2 (21.8) -> 6.5 (23.8) MB, 6.0 ms [Runtime::PerformGC].
[1592]       98 ms: Scavenge 9.4 (23.8) -> 8.6 (24.8) MB, 5.0 ms [Runtime::PerformGC].
[1592]      194 ms: Scavenge 14.4 (24.8) -> 11.8 (25.8) MB, 23.0 ms [Runtime::PerformGC].
[1592]      340 ms: Scavenge 15.9 (25.8) -> 14.1 (30.8) MB, 15.0 ms
    (+ 10.0 ms in 41 steps since last GC) [Runtime::PerformGC].
[1592]      689 ms: Mark-sweep 18.7 (30.8) -> 14.0 (32.8) MB, 7.0 ms
    (+ 20.0 ms in 113 steps since start of marking, biggest step 1.0 ms)
      [StackGuard GC request] [GC in old space requested].
[1592]     1240 ms: Scavenge 21.8 (32.8) -> 14.0 (32.8) MB, 0.0 ms [Runtime::PerformGC].
[1592]     1799 ms: Scavenge 21.8 (32.8) -> 14.0 (32.8) MB, 0.0 ms [Runtime::PerformGC].
[1592]     2350 ms: Scavenge 21.8 (32.8) -> 14.0 (32.8) MB, 1.0 ms [Runtime::PerformGC].
[1592]     2902 ms: Scavenge 21.8 (32.8) -> 14.0 (32.8) MB, 1.0 ms [Runtime::PerformGC].
```

Listing 2.4: Log of garbage collection in V8

[6] http://en.wikipedia.org/wiki/Cheney's_algorithm

[7] http://en.wikipedia.org/wiki/Garbage_collection_(computer_science)

# 3. Particle system

Particle system is one of most commonly used techniques to simulate smoke, fire, rain and other groups of discrete objects, usually independent from each other. System consist of defined number of emitters, producing lightweight particle objects with certain parameters. Each emitter has defined production ratio and each particle a certain lifespan, resulting in upper limit of total particles on screen. Some systems use also attraction points which enable better control over particles, using equations often similar to those of electrostatic forces. Such simulation is independent from rendering. The same particle system may be used for different effects with proper configuration.

Rysunek 3.1: Example rendering of tested particle system

## 3.1. System parameters

Tested system works on two-dimensional Cartesian plane. For purpose of performance analysis movements are calculated based only on frames rather than actual flow of time. This means that systems with different framerate will result in different visualisations, but requesting given amount of frames rendered will result in the same lifespan and total number of particles in both systems.

Emitter supports following parameters:

– position - initial position of created particles

– angle - angle counting clockwise from vector [0, 1]

– spread - parameter controlling random differences in initial angle of particles

– velocity - initial velocity of particles, in pixels per frame

– velocity spread - parameter controlling random differences in initial velocity of particles

– lifespan - initial lifespan of particles

– productionRate - amount of particles initialized in each frame

A Particle has similar properties:

– position

– velocity

– lifespan

– age - counted in frames, when higher then lifespan particle is removed from system

Source code of both implementations is attached in Appendix B.

## 3.2. Implementation with high memory allocation

Initial tested implementation has one very important property of particle emitter. Whenever new particles are created, new array of pointers is allocated and returned from emitter. System appends new particles to existing array. In each frame particle system creates new, empty array of particles and adds there only particles that are still alive. Array from previous frame and all dead particles are removed from system and deallocated. This is clearly suboptimal solution that allocates and deallocates plenty of memory in each frame. Purpose of this exercise is to show how both languages handle bad code and how big impact it has comparing to the optimal solution.

TODO: this part may need to be updated before publication. TODO: Is this accounting properly for v8 startup time? Maybe profiling ticks would be better.

```
$ time browser/static/d8 browser/static/particles1.js

real    0m20.619s
user    0m0.000s
sys     0m0.015s
```

Listing 3.1: Time measurement of unoptimized particle system in JavaScript

```
$ time runtime/static/particles1

real    0m2.606s
user    0m1.950s
sys     0m0.498s
```

Listing 3.2: Time measurement of unoptimized particle system in C++

Time measurement shows that JavaScript version is almost 8 times slower than native one. To analyse situation –prof and –log-timer-events flags may be used. Output file v8.log is parsed using available online tool.[1]

```
Statistical profiling result from null, (28293 ticks, 2631 unaccounted, 0 excluded).

 [Shared libraries]:
   ticks  total  nonlib   name
   9577   37.3%    0.0%  D:\Dropbox\praca_magisterska\physics\browser\static\d8.exe
   1078    4.2%    0.0%  C:\Windows\SysWOW64\ntdll.dll
      3    0.0%    0.0%  C:\Windows\syswow64\kernel32.dll
      2    0.0%    0.0%  C:\Windows\syswow64\KERNELBASE.dll

 [JavaScript]:
   ticks  total  nonlib   name
   4621   18.0%   30.8%  LazyCompile: ~verifyIfAlive :463
   2228    8.7%   14.9%  LazyCompile: ~stepParticle :460
   1800    7.0%   12.0%  LazyCompile: ~smash.ParticleSystem.step :449
   1018    4.0%    6.8%  Stub: CompareICStub
```

[1] http://v8.googlecode.com/svn/branches/bleeding_edge/tools/profviz/profviz.html

```
    949    3.7%    6.3%  Stub: LoadFieldStub {1}
    831    3.2%    5.5%  LazyCompile: ~<anonymous> :466
    799    3.1%    5.3%  Builtin: A builtin from the snapshot
    770    3.0%    5.1%  Stub: CompareICStub {1}
    739    2.9%    4.9%  Stub: CallFunctionStub
    614    2.4%    4.1%  LazyCompile: IN native runtime.js:348
    549    2.1%    3.7%  Stub: LoadFieldStub
    430    1.7%    2.9%  Stub: CEntryStub
    259    1.0%    1.7%  LazyCompile: *forEach native array.js:1188
    246    1.0%    1.6%  LazyCompile: *verifyIfAlive :463
    169    0.7%    1.1%  LazyCompile: *stepParticle :460
    145    0.6%    1.0%  LazyCompile: *<anonymous> :466
    129    0.5%    0.9%  Stub: LoadFieldStub {3}
    127    0.5%    0.8%  LazyCompile: *smash.ParticleEmitter.getNewParticles :429
    121    0.5%    0.8%  Stub: CompareICStub {2}
     86    0.3%    0.6%  Stub: TranscendentalCacheStub {1}
     79    0.3%    0.5%  Stub: ParticleSystem {1}
     79    0.3%    0.5%  Stub: LoadFieldStub {4}
     78    0.3%    0.5%  Stub: LoadFieldStub {2}
     65    0.3%    0.4%  Stub: TranscendentalCacheStub
     52    0.2%    0.3%  Stub: RecordWriteStub
     42    0.2%    0.3%  Stub: CallFunctionStub_Args1
     34    0.1%    0.2%  Stub: LoadFieldStub {5}
     30    0.1%    0.2%  Stub: InterruptStub
     14    0.1%    0.1%  LazyCompile: ~appendNewParticles :455
      8    0.0%    0.1%  Stub: KeyedLoadElementStub
      5    0.0%    0.0%  LazyCompile: ~forEach native array.js:1188
      5    0.0%    0.0%  LazyCompile: *appendNewParticles :455
      3    0.0%    0.0%  LazyCompile: *smash.Particle :389
      3    0.0%    0.0%  Builtin: A builtin from the snapshot {1}
      2    0.0%    0.0%  Stub: CEntryStub {1}
      2    0.0%    0.0%  LazyCompile: ~smash.ParticleEmitter.getNewParticles :429
      1    0.0%    0.0%  Stub: ToBooleanStub
      1    0.0%    0.0%  Stub: FastNewClosureStub
      1    0.0%    0.0%  Stub: CompareICStub {3}
      1    0.0%    0.0%  Stub: CallConstructStub
      1    0.0%    0.0%  Stub: BinaryOpStub_ADD_OverwriteRight_Smi+Number
      1    0.0%    0.0%  LazyCompile: APPLY_PREPARE native runtime.js:432
      1    0.0%    0.0%  LazyCompile: *random native math.js:188
      1    0.0%    0.0%  KeyedLoadIC:  {55}
      1    0.0%    0.0%  Function: ~stepParticle :460

[C++]:
   ticks  total  nonlib   name

[GC]:
   ticks  total  nonlib   name
   2067    8.1%
```

Listing 3.3: Profiler output for unoptimized particles

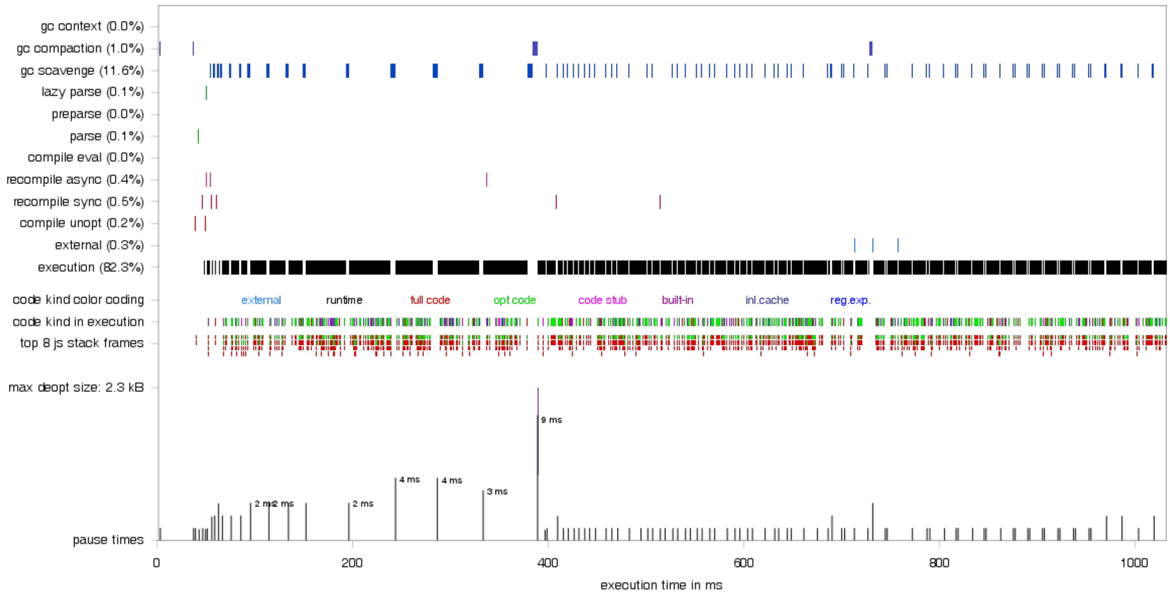
Methods prefixed with  are unoptimized, the ones prefixed with * are JIT compiled. As seen in profiler log, most of the time is spent in unoptimised versions of verifyIfAlive and stepParticle methods of particle system step.

```
smash.ParticleSystem.prototype.step = function() {
  this.emitters.forEach(function appendNewParticles(a) {
    this.particles.push.apply(this.particles, a.getNewParticles())
  }, this);
  var newParticles = [];
  function stepParticle(a) {
    a.step();
  }
  function verifyIfAlive(a) {
    if (0 <= a.positionX && a.positionX < smash.ParticleSystem.CANVAS_WIDTH &&
      0 <= a.positionY && a.positionY < smash.ParticleSystem.CANVAS_HEIGHT &&
      a.age < a.lifespan) {
    newParticles.push(a);
  }
  }
  this.particles.forEach(function (a) {
    stepParticle(a);
    verifyIfAlive(a);
  }, this);
  this.particles = newParticles;
};
```

Listing 3.4: Annotated part of source

The same methods are also used in optimised versions, where they take significantly less ticks to run. It's clear that presented code not only allocates and deallocates too much memory, but also fails to run in optimised mode. It's visible on chart obtained from the same tool - stripe labelled ćode kind in executionśhows multiple kinds of code running and is interrupted often with garbage collection cycles.

Rysunek 3.2: Chart of time used in unoptimised verion of JavaScript




Garbage collection cycles blocking execution are also visible with –trace-gc flag.

```
$ browser/static/d8 --trace-gc browser/static/particles1.js
[9696]        10 ms: Scavenge 1.6 (18.8) -> 0.9 (18.8) MB, 0.0 ms [Runtime::PerformGC].
```

```
[9696]       14 ms: Scavenge 1.6 (18.8) -> 1.3 (19.8) MB, 2.0 ms [Runtime::PerformGC].
(...)
[9696]      233 ms: Scavenge 15.0 (25.8) -> 9.7 (25.8) MB, 4.0 ms [Runtime::PerformGC].
[9696]      277 ms: Scavenge 15.6 (26.8) -> 10.0 (27.8) MB, 4.0 ms
    (+ 13.0 ms in 22 steps since last GC) [Runtime::PerformGC].
[9696] Limited new space size due to high promotion rate: 1 MB
[9696]      284 ms: Mark-sweep 10.6 (27.8) -> 10.4 (28.8) MB, 6.0 ms
    (+ 14.0 ms in 23 steps since start of marking, biggest step 1.0 ms)
  [StackGuard GC request] [GC in old space requested].
[9696]      294 ms: Scavenge 11.5 (28.8) -> 11.0 (29.8) MB, 1.0 ms [Runtime::PerformGC].
[9696]      295 ms: Scavenge 11.9 (29.8) -> 11.6 (30.8) MB, 0.0 ms [Runtime::PerformGC].
(...)
[9696]      555 ms: Scavenge 43.4 (68.8) -> 43.4 (69.8) MB, 1.0 ms [Runtime::PerformGC].
[9696] Increasing marking speed to 3 due to high promotion rate
[9696]      564 ms: Scavenge 43.9 (69.8) -> 43.6 (69.8) MB, 1.0 ms
    (+ 4.0 ms in 3 steps since last GC) [Runtime::PerformGC].
[9696]      576 ms: Scavenge 44.2 (69.8) -> 44.2 (70.8) MB, 1.0 ms
    (+ 8.0 ms in 2 steps since last GC) [Runtime::PerformGC].
[9696]      581 ms: Mark-sweep 44.9 (70.8) -> 13.4 (61.8) MB, 3.0 ms
    (+ 14.0 ms in 6 steps since start of marking, biggest step 2.0 ms)
  [StackGuard GC request] [GC in old space requested].
[9696]      591 ms: Scavenge 14.3 (61.8) -> 14.1 (61.8) MB, 0.0 ms [Runtime::PerformGC].
(...)
```

Listing 3.5: Garbage collection in unoptimised particle system

To improve performance different approach to particles allocation is used. Each particle has a flag "isDead"telling if it may be safely reused for new particle. Particle pool is kept along with a list of pointers to dead particles. This way when system reaches it's maximum congestion (around 15000 particles in given example) no new allocations occur. Creation of new particles is moved from particle emitter to particle system, to avoid allocation of new array. Emitter works now as a structure describing behaviour but not implementing one.

```
$ time browser/static/d8 browser/static/particles2.js

real    0m3.275s
user    0m0.000s
sys     0m0.015s
```

Listing 3.6: Time measurement of optimized particle system in JavaScript

```
$ time runtime/static/particles2

real    0m1.483s
user    0m1.387s
sys     0m0.000s
```

Listing 3.7: Time measurement of optimized particle system in C++

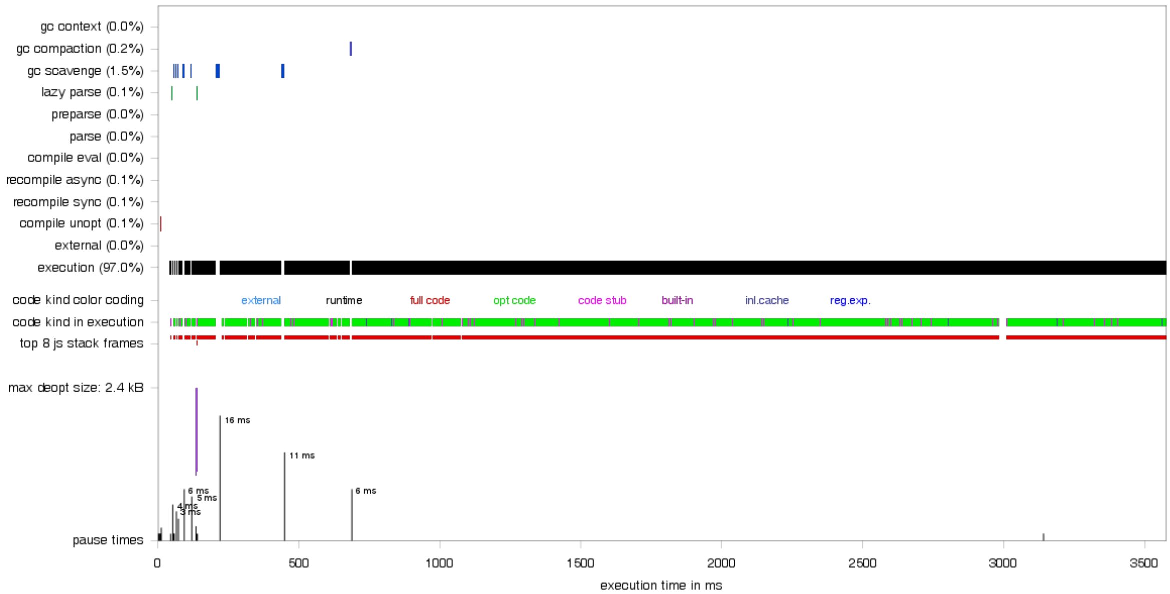
Optimised version shows overall improvement of 85% for JavaScript and 45% for C++ making JavaScript version only 2.2 times slower than native one. It's clearly visible that JavaScript is more sensitive to unwise memory management.

```
Statistical profiling result from null, (3780 ticks, 2 unaccounted, 0 excluded).

 [Shared libraries]:
   ticks  total  nonlib   name
     98    2.6%    0.0%  D:\Dropbox\praca_magisterska\physics\browser\static\d8.exe
      6    0.2%    0.0%  C:\Windows\SysWOW64\ntdll.dll

 [JavaScript]:
   ticks  total  nonlib   name
   3476   92.0%   94.6%  LazyCompile: *f.step browser/static/particles2.js:28
    101    2.7%    2.7%  Stub: TranscendentalCacheStub {1}
     89    2.4%    2.4%  Stub: TranscendentalCacheStub
      5    0.1%    0.1%  Script: ~browser/static/particles2.js
      1    0.0%    0.0%  Stub: TranscendentalCacheStub {2}
      1    0.0%    0.0%  Stub: BinaryOpStub_MUL_OverwriteLeft_Number+Number
      1    0.0%    0.0%  LazyCompile: *sin native math.js:199
      1    0.0%    0.0%  Builtin: A builtin from the snapshot

 [C++]:
   ticks  total  nonlib   name

 [GC]:
   ticks  total  nonlib   name
     59    1.6%
```

Listing 3.8: Profiler output for optimized particles

Profiling shows that step method of particle system now runs always in optimised mode and almost no time is spent on other methods. The same is visible on profiling chart, where ćode kind in executionśtripe shows only optimised code.

Rysunek 3.3: Chart of time used in optimised verion of JavaScript




Situation is also improved in garbage collection log.

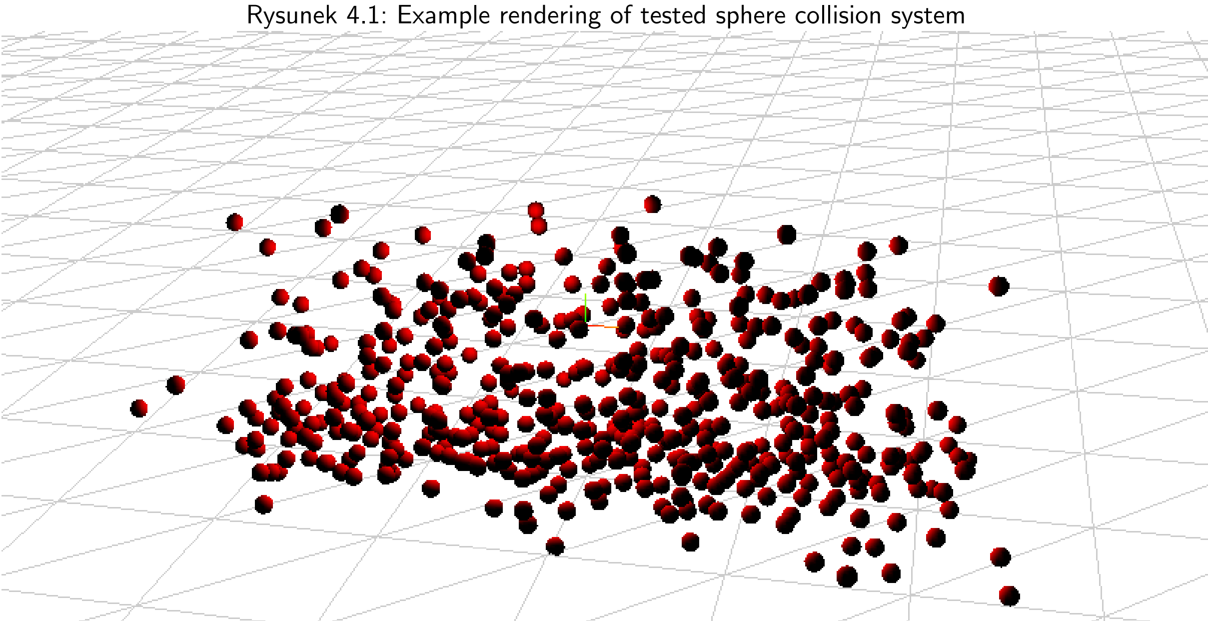
```
$ browser/static/d8 --trace-gc browser/static/particles2.js
[8348]       10 ms: Scavenge 1.6 (18.8) -> 0.9 (18.8) MB, 1.0 ms [Runtime::PerformGC].
[8348]       13 ms: Scavenge 1.6 (18.8) -> 1.2 (19.8) MB, 1.0 ms [Runtime::PerformGC].
[8348]       16 ms: Scavenge 1.9 (19.8) -> 1.7 (19.8) MB, 1.0 ms [Runtime::PerformGC].
[8348]       19 ms: Scavenge 2.4 (19.8) -> 2.2 (19.8) MB, 2.0 ms [Runtime::PerformGC].
[8348]       25 ms: Scavenge 3.7 (19.8) -> 3.3 (20.8) MB, 3.0 ms [Runtime::PerformGC].
[8348]       32 ms: Scavenge 4.8 (20.8) -> 4.3 (21.8) MB, 2.0 ms [Runtime::PerformGC].
[8348]       50 ms: Scavenge 7.2 (21.8) -> 6.5 (23.8) MB, 5.0 ms [Runtime::PerformGC].
[8348]       75 ms: Scavenge 9.4 (23.8) -> 8.6 (24.8) MB, 5.0 ms [Runtime::PerformGC].
[8348]      173 ms: Scavenge 14.4 (24.8) -> 11.8 (25.8) MB, 24.0 ms [Runtime::PerformGC].
[8348]      319 ms: Scavenge 15.9 (25.8) -> 14.1 (30.8) MB, 15.0 ms
    (+ 9.0 ms in 44 steps since last GC) [Runtime::PerformGC].
[8348]      669 ms: Mark-sweep 18.7 (30.8) -> 14.0 (32.8) MB, 7.0 ms
    (+ 17.0 ms in 116 steps since start of marking, biggest step 1.0 ms)
  [StackGuard GC request] [GC in old space requested].
[8348]     1229 ms: Scavenge 21.8 (32.8) -> 14.0 (32.8) MB, 0.0 ms [Runtime::PerformGC].
[8348]     1793 ms: Scavenge 21.8 (32.8) -> 14.0 (32.8) MB, 0.0 ms [Runtime::PerformGC].
[8348]     2353 ms: Scavenge 21.8 (32.8) -> 14.0 (32.8) MB, 0.0 ms [Runtime::PerformGC].
[8348]     2914 ms: Scavenge 21.8 (32.8) -> 14.0 (32.8) MB, 1.0 ms [Runtime::PerformGC].
```

Listing 3.9: Garbage collection in optimised particle system

# 4. Sphere collision

Spheres are the simplest of bounding shapes used in collision detection. This chapter presents tests for two versions of algorithms - naive $O(N^2)$ approach and with partitioned space. While simpler algorithm has far greater number of collision checks per frame, it allocates almost no memory per frame. More complex method will minimise number of checks, but additional structure and steps added may influence overall execution time in unexpected way.

Rysunek 4.1: Example rendering of tested sphere collision system



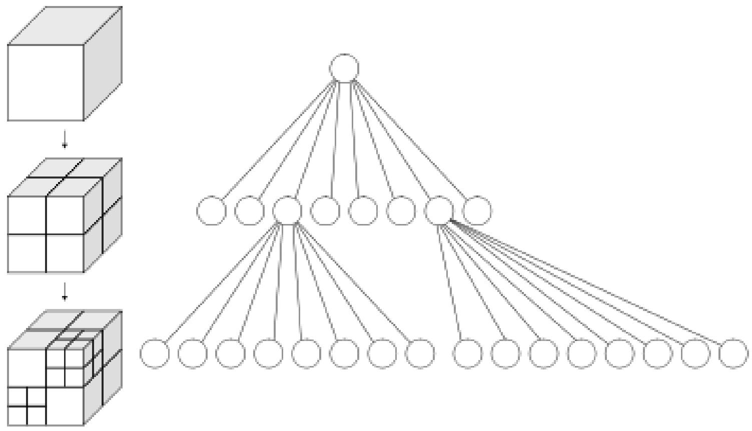
## 4.1. Algorithm description

Collision detection for spheres is a trivial task. If distance between two spheres is smaller than sum of their radiuses, spheres collide.

$$\sqrt{(S_1.x - S_2.x)^2 + (S_1.y - S_2.y)^2 + (S_1.z - S_2.z)^2} < S_1.radius + S_2.radius$$

While the equation is simple, with large number N of colliding objects complexity of this detection is $O(N^2)$. Methods of space partitioning are used to reduce number of checks. One used in this benchmark is Octree. Base for algorithm is a tree-like structure of bounding boxes. Whenever a box contains more than one colliding object, it's divided into eight smaller boxes, by partitioning each edge by 2. When maximum tree depth is reached, multiple objects are stored in one box. One object may be referenced from multiple boxes, when it's size and position make them intersect. Each movement requires a check if object has already moved to one of neighbour boxes.

Rysunek 4.2: Octree structure. Source: http://en.wikipedia.org/wiki/File:Octree2.svg/



Having objects grouped in boxes reduces complexity of collision check. Since an object may collide only with objects in the same box, number of checks is much smaller. Overall complexity of Octree checks is $O(NlogN)$. TODO: reference for complexity.

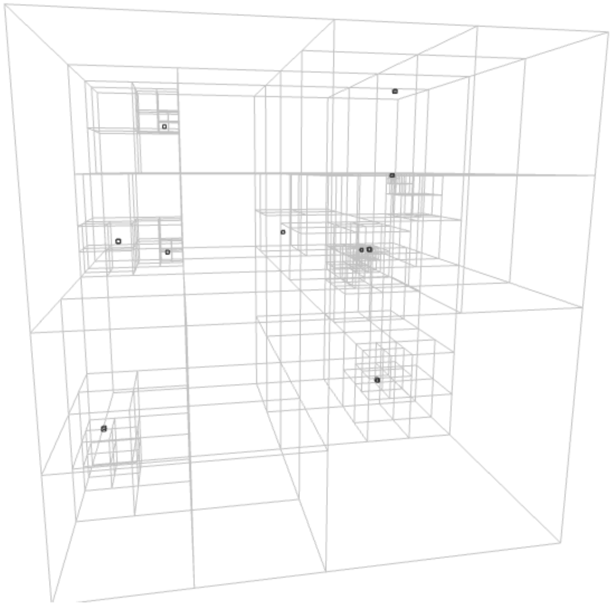
When collision is detected, collision response is calculated. It's result is directly derived from rule of conservation of momentum and conservation of kinetic energy.

$$m_1 * \vec{v_1} + m_2 * \vec{v_2} = m_1 * \vec{v_1'} + m_2 * \vec{v_2'}$$

$$\frac{m_1}{2} * \vec{v_1}^2 + \frac{m_2}{2} * \vec{v_2}^2 = \frac{m_1}{2} * \vec{v_1'}^2 + \frac{m_2}{2} * \vec{v_2'}^2$$

TODO: add math here, finish this chapter.

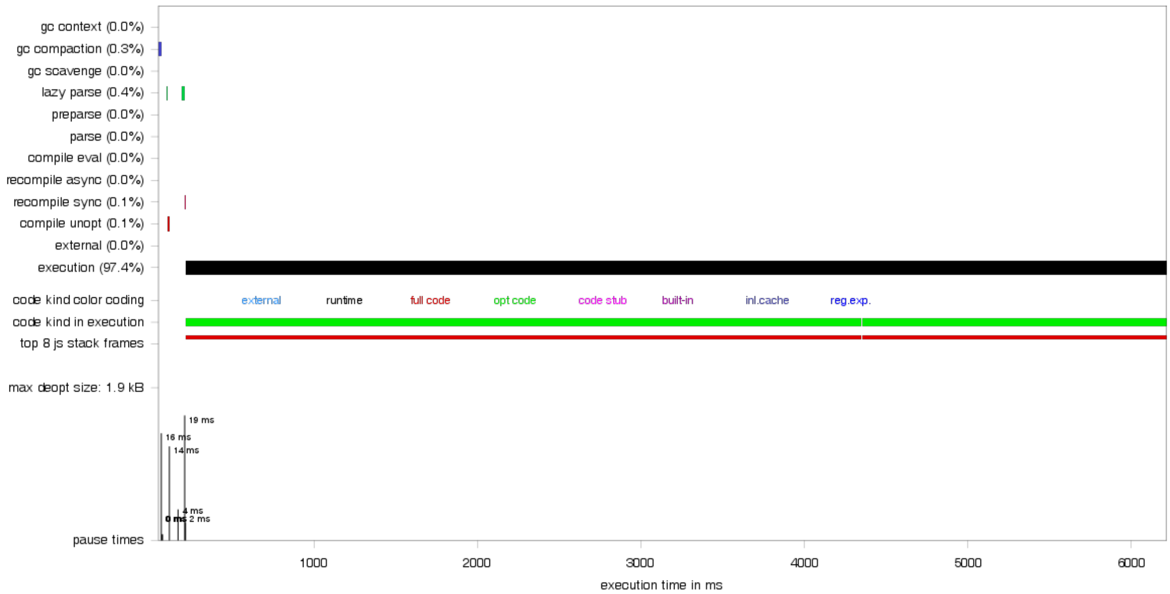
Rysunek 4.3: Example of WebGL Octree debug rendering. Available online at http://pawlowski.it/octtree/



## 4.2. $O(N^2)$ approach

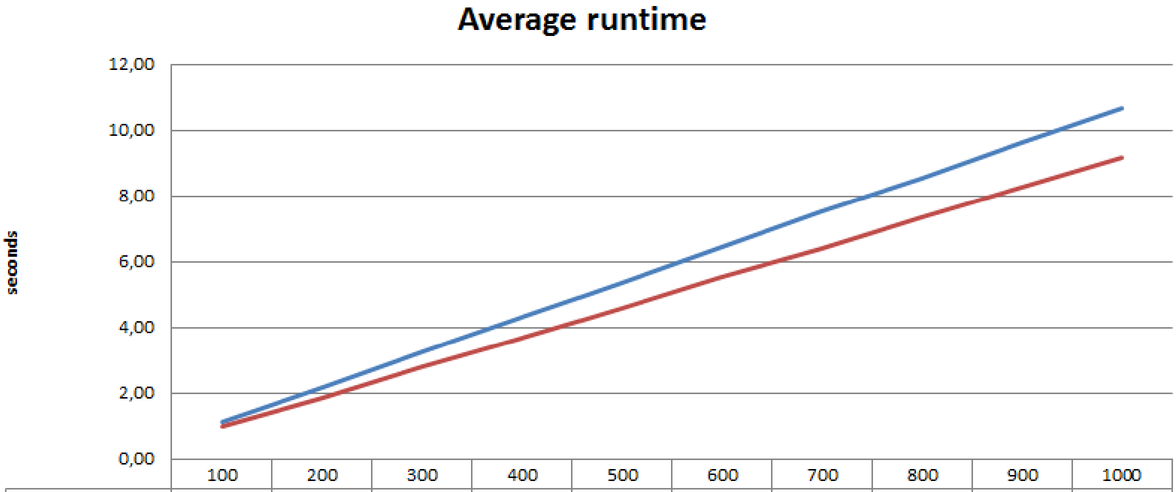
Naive approach for collision detection proves to by easy to implement in JavaScript. Since almost no memory is allocated in each frame, no garbage collection issues appear. All methods are well defined and work mostly on floats. This results in highly optimised binary code produced by compiler, as show on 4.4.

Rysunek 4.4: Chart of time used in optimised version of JavaScript



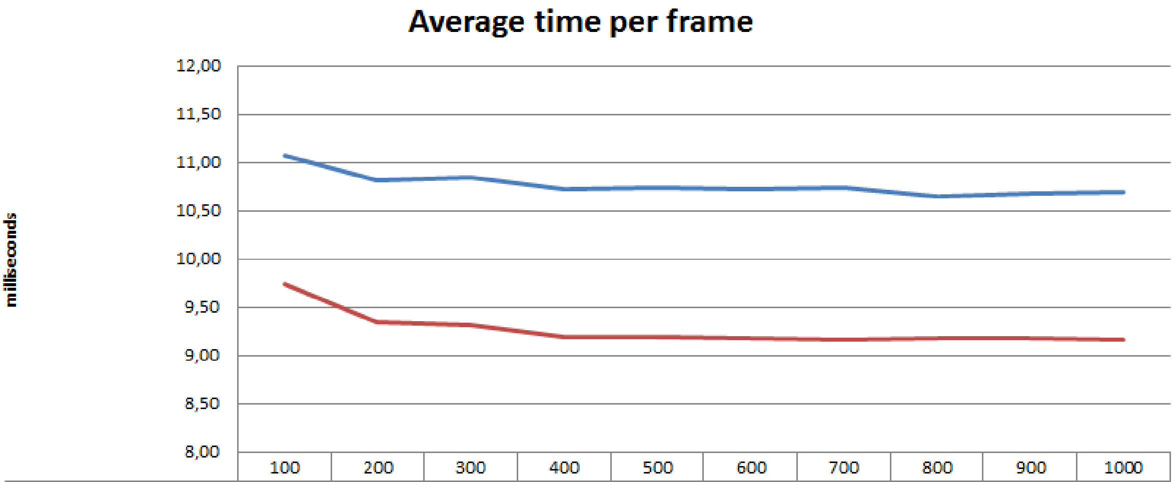

Multiple tests with N=1000 and different number of frames rendered show, that for simple mathematical task performance of JavaScript is very close to those of C++. On average, JavaScript version of benchmark runs 15% longer than C++ one.

Rysunek 4.5: Comparison of total execution time. N = 1000, varying number of frames.




| | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900 | 1000 |
|---|---|---|---|---|---|---|---|---|---|---|
| AVG JS time | 1,11 | 2,16 | 3,25 | 4,29 | 5,37 | 6,44 | 7,51 | 8,51 | 9,61 | 10,68 |
| AVG C++ time | 0,97 | 1,87 | 2,79 | 3,68 | 4,59 | 5,51 | 6,41 | 7,34 | 8,26 | 9,16 |

Rysunek 4.6: Comparison of execution time per frame. N = 1000, varying number of frames.




| | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900 | 1000 |
|---|---|---|---|---|---|---|---|---|---|---|
| AVG JS ms per frame | 11,08 | 10,82 | 10,85 | 10,73 | 10,73 | 10,73 | 10,73 | 10,64 | 10,67 | 10,68 |
| AVG C++ ms per frame | 9,74 | 9,34 | 9,31 | 9,20 | 9,19 | 9,18 | 9,16 | 9,18 | 9,17 | 9,16 |

## 4.3. Octree-partitioned space

# 5. Boxes collision

## 5.1. Algorithm description

## 5.2. Three-dimensional SAT

## 5.3. $O(N^2)$ approach

## 5.4. Octree-partitioned space

# 6. Summary

## 6.1. Encountered environment limitations

## 6.2. Browser advantages

## 6.3. Recommended techniques

## 6.4. Final thoughts

# A. Acknowledgements

# B. Source code

## B.1. Math utilities

```
/**
 * @fileoverview Math utils.
 * @author sebastian.poreba@gmail.com (Sebastian PorÄ™ba)
 */

goog.provide('smash.math');


/**
 * @param {number} x
 * @return {number}
 */
smash.math.square = function(x) {
  return x * x;
};


/**
 * @param {number} x1
 * @param {number} y1
 * @param {number} z1
 * @param {number} x2
 * @param {number} y2
 * @param {number} z2
 * @return {number}
 */
smash.math.vectorDistance = function(x1, y1, z1, x2, y2, z2) {
  return Math.sqrt(smash.math.square(x1 - x2) +
      smash.math.square(y1 - y2) +
      smash.math.square(z1 - z2));
};


/**
 * @param {number} x
 * @param {number} y
 * @param {number} z
 * @return {number}
 */
smash.math.vectorLength = function(x, y, z) {
  return Math.sqrt(smash.math.square(x) +
      smash.math.square(y) +
      smash.math.square(z));
};


/**
 * @param {!smash.Sphere} sphere1
 * @param {!smash.Sphere} sphere2
 * @return {boolean}
 */
smash.math.checkCollidingSpheres = function(sphere1, sphere2) {
```

```
  return smash.math.vectorDistance(
      sphere1.positionX, sphere1.positionY, sphere1.positionZ,
      sphere2.positionX, sphere2.positionY, sphere2.positionZ) <
      sphere1.radius + sphere2.radius;
};
```

Listing B.1: Math utilities in JavaScript

```

#include "math.h"

/**
 * @param {number} x
 * @return {number}
 */
float smash::math::square(float x) {
  return x * x;
};


/**
 * @param {number} x1
 * @param {number} y1
 * @param {number} z1
 * @param {number} x2
 * @param {number} y2
 * @param {number} z2
 * @return {number}
 */
float smash::math::vectorDistance(float x1, float y1, float z1, float x2, float y2, float z2) {
  return sqrt(smash::math::square(x1 - x2) +
      smash::math::square(y1 - y2) +
      smash::math::square(z1 - z2));
};


/**
 * @param {number} x
 * @param {number} y
 * @param {number} z
 * @return {number}
 */
float smash::math::vectorLength(float x, float y, float z) {
  return sqrt(smash::math::square(x) +
      smash::math::square(y) +
      smash::math::square(z));
};


/**
 * @param {!smash.Sphere} sphere1
 * @param {!smash.Sphere} sphere2
 * @return {boolean}
 */
bool smash::math::checkCollidingSpheres(smash::Sphere* sphere1, smash::Sphere* sphere2) {
```

```cpp
  return smash::math::vectorDistance(
      sphere1->positionX, sphere1->positionY, sphere1->positionZ,
      sphere2->positionX, sphere2->positionY, sphere2->positionZ) <
      sphere1->radius + sphere2->radius;
};
```

Listing B.2: Math utilities in C++

## B.2. Particle system

```
/**
 * @fileoverview Particle object.
 * @author sebastian.poreba@gmail.com (Sebastian PorÄ™ba)
 */

goog.provide('smash.Particle');

/**
 * @struct
 * @constructor
 */
smash.Particle = function() {
  /**
   * @type {number}
   */
  this.positionX = 0.1;

  /**
   * @type {number}
   */
  this.positionY = 0.1;

  /**
   * @type {number}
   */
  this.velocityX = 0.1;

  /**
   * @type {number}
   */
  this.velocityY = 0.1;

  /**
   * @type {number}
   */
  this.age = 0;

  /**
   * In seconds.
   * @type {number}
   */
  this.lifespan = 0;

  /**
   * @type {boolean}
   */
  this.isDead = false;
};

/**
 *
 */
```

```
smash.Particle.prototype.step = function() {
  this.positionX += this.velocityX;
  this.positionY += this.velocityY;
  this.age ++;
};


/**
 * Recover defaults.
 */
smash.Particle.prototype.reset = function() {
  this.positionX = 0;
  this.positionY = 0;
  this.velocityX = 0;
  this.velocityY = 0;
  this.age = 0;
  this.lifespan = 0;
  this.isDead = false;
};
```

Listing B.3: Particle object in JavaScript

```
/**
 * @fileoverview Particle object.
 * @author sebastian.poreba@gmail.com (Sebastian Poreba)
 */
#include "particle.h"

smash::Particle::Particle() {
  positionX = 0;
  positionY = 0;
  velocityX = 0;
  velocityY = 0;
  age = 0;
  lifespan = 0;
  isDead = false;
}

/**
 * @param deltaTime
 */
void smash::Particle::step() {
  this->positionX += this->velocityX;
  this->positionY += this->velocityY;
  this->age++;
};


/**
 * Recover defaults.
 */
void smash::Particle::reset() {
  this->positionX = 0;
  this->positionY = 0;
  this->velocityX = 0;
```

```
  this->velocityY = 0;
  this->age = 0;
  this->lifespan = 0;
  this->isDead = false;
};
```

Listing B.4: Particle object in C++

```
/**
 * @fileoverview Particle emitter.
 * @author sebastian.poreba@gmail.com (Sebastian PorÄ™ba)
 */

goog.provide('smash.ParticleEmitter');

goog.require('smash.Particle');

/**
 * @constructor
 */
smash.ParticleEmitter = function() {
  /**
   * @type {number}
   */
  this.positionX = 0.1;

  /**
   * @type {number}
   */
  this.positionY = 0.1;

  /**
   * @type {number}
   */
  this.angle = 0.1;

  /**
   * @type {number}
   */
  this.velocity = 10.1;

  /**
   * @type {number}
   */
  this.velocitySpread = 0.2;

  /**
   * @type {number}
   */
  this.spread = Math.PI * 10 / 180;

  /**
   * In ticks.
   * @type {number}
   */
```

```
  this.lifespan = 50;

  /**
   * @type {number}
   */
  this.productionRate = 10;
};


/**
 * @param {number} angle
 */
smash.ParticleEmitter.prototype.setAngle = function(angle) {
  this.angle = Math.PI * angle / 180;
};


/**
 * @param {number} velocity
 */
smash.ParticleEmitter.prototype.setVelocity = function(velocity) {
  this.velocity = velocity;
};


/**
 * @param {number} velocitySpread
 */
smash.ParticleEmitter.prototype.setVelocitySpread = function(velocitySpread) {
  this.velocitySpread = velocitySpread;
};


/**
 * @param {number} spread
 */
smash.ParticleEmitter.prototype.setSpread = function(spread) {
  this.spread = Math.PI * spread / 180;
};


/**
 * @param {number} lifespan
 */
smash.ParticleEmitter.prototype.setLifespan = function(lifespan) {
  this.lifespan = lifespan;
};


/**
 * @param {number} rate
 */
smash.ParticleEmitter.prototype.setProductionRate = function(rate) {
  this.productionRate = rate;
};
```

```
/**
 * @return {!Array.<!smash.Particle>}
 */
smash.ParticleEmitter.prototype.getNewParticles = function() {
  var newParticles = [];
  for (var i = 0; i < this.productionRate; i++) {
    var p = new smash.Particle();
    p.lifespan = this.lifespan;
    p.positionX = this.positionX;
    p.positionY = this.positionY;
    p.velocityX = Math.sin(this.angle +
        (Math.random() - 0.5) * this.spread) *
        this.velocity * this.velocitySpread;
    p.velocityY = Math.cos(this.angle +
        (Math.random() - 0.5) * this.spread) *
        this.velocity *
        (1 + (Math.random() - 0.5) * this.velocitySpread);
    newParticles.push(p);
  }
  return newParticles;
};
```

Listing B.5: Particle emitter object in JavaScript

```
/**
 * @fileoverview Particle emitter.
 * @author sebastian.poreba@gmail.com (Sebastian Poreba)
 */
#include "particleEmitter.h"

smash::ParticleEmitter::ParticleEmitter() {
  positionX = 0;
  positionY = 0;
  angle = 0;
  velocity = 10;
  velocitySpread = 0.2;
  spread = M_PI * 10 / 180;
  lifespan = 50;
  productionRate = 10;
};


/**
 * @param angle
 */
void smash::ParticleEmitter::setAngle(float angle) {
  this->angle = M_PI * angle / 180;
};


/**
 * @param velocity
 */
void smash::ParticleEmitter::setVelocity(float velocity) {
  this->velocity = velocity;
```

```
};


/**
 * @param velocitySpread
 */
void smash::ParticleEmitter::setVelocitySpread(float velocitySpread) {
  this->velocitySpread = velocitySpread;
};


/**
 * @param spread
 */
void smash::ParticleEmitter::setSpread(float spread) {
  this->spread = M_PI * spread / 180;
};


/**
 * @param lifespan
 */
void smash::ParticleEmitter::setLifespan(float lifespan) {
  this->lifespan = lifespan;
};


/**
 * @param rate
 */
void smash::ParticleEmitter::setProductionRate(int rate) {
  this->productionRate = rate;
};


/**
 * @return {!Array.<!smash::Particle>}
 */
std::vector<smash::Particle*>* smash::ParticleEmitter::getNewParticles() {
  std::vector<smash::Particle*> *newParticles = new std::vector<smash::Particle*>;
  for (int i = 0; i < this->productionRate; i++) {
    smash::Particle* p = new smash::Particle();
    p->lifespan = this->lifespan;
    p->positionX = this->positionX;
    p->positionY = this->positionY;
    p->velocityX = sin(this->angle +
        (((float) rand() / (RAND_MAX)) - 0.5) * this->spread) *
        this->velocity *
        (1 + (((float) rand() / (RAND_MAX)) - 0.5) * this->velocitySpread);
    p->velocityY = cos(this->angle +
        (((float) rand() / (RAND_MAX)) - 0.5) * this->spread) *
        this->velocity *
        (1 + (((float) rand() / (RAND_MAX)) - 0.5) * this->velocitySpread);
    newParticles->push_back(p);
  }
  return newParticles;
};
```

Listing B.6: Particle emitter object in C++

```
/**
 * @fileoverview Particle system.
 * @author sebastian.poreba@gmail.com (Sebastian PorÄ™ba)
 */

goog.provide('smash.ParticleSystem');

goog.require('smash.Particle');
goog.require('smash.ParticleEmitter');

/**
 * @constructor
 */
smash.ParticleSystem = function() {
  /**
   * @type {!Array.<smash.Particle>}
   */
  this.particles = [];

  /**
   * @type {!Array.<smash.ParticleEmitter>}
   */
  this.emitters = [];

  if (smash.ParticleSystem.DRAWING_ENABLED) {
    /**
     * @type {!Element}
     */
    this.canvas = window.document.createElement("canvas");
    this.canvas.width = smash.ParticleSystem.CANVAS_WIDTH;
    this.canvas.height = smash.ParticleSystem.CANVAS_HEIGHT;
    window.document.body.appendChild(this.canvas);

    /**
     * @type {!CanvasRenderingContext2D}
     */
    this.context = this.canvas.getContext('2d');

    /**
     * @type {!ImageData}
     */
    this.imageData = this.context.getImageData(0, 0,
        smash.ParticleSystem.CANVAS_WIDTH, smash.ParticleSystem.CANVAS_HEIGHT);

    /**
     * @type {!CanvasPixelArray}
     */
    this.pixels = this.imageData.data;
  }
};

/**
```

```
 * @const {boolean}
 */
smash.ParticleSystem.DRAWING_ENABLED = false;

/**
 * @const {number}
 */
smash.ParticleSystem.CANVAS_WIDTH = 1200;

/**
 * @const {number}
 */
smash.ParticleSystem.CANVAS_HEIGHT = 400;


smash.ParticleSystem.prototype.step = function() {
  if (smash.ParticleSystem.DRAWING_ENABLED) {
    for (var i = 0; i < smash.ParticleSystem.CANVAS_WIDTH *
        smash.ParticleSystem.CANVAS_HEIGHT * 4; i+=4) {
      this.pixels[i] = 0;
      this.pixels[i + 1] = 0;
      this.pixels[i + 2] = 0;
      this.pixels[i + 3] = 0;
    }
  }

  this.emitters.forEach(function(emitter) {
    this.particles.push.apply(this.particles,
        emitter.getNewParticles());
  }, this);

  var newParticles = [];
  this.particles.forEach(function(p) {
    p.step();
    if (p.positionX >= 0 &&
        p.positionX < smash.ParticleSystem.CANVAS_WIDTH &&
        p.positionY >= 0 &&
        p.positionY < smash.ParticleSystem.CANVAS_HEIGHT &&
        p.age < p.lifespan) {
      newParticles.push(p);
    }

    if (smash.ParticleSystem.DRAWING_ENABLED) {
      var baseIndex =
          (Math.round(p.positionY) *
              smash.ParticleSystem.CANVAS_WIDTH +
              Math.round(p.positionX)) * 4;
      this.pixels[baseIndex] = 255;
      this.pixels[baseIndex + 1] = 0;
      this.pixels[baseIndex + 2] = 0;
      this.pixels[baseIndex + 3] = 255;
    }
  }, this);
  if (smash.ParticleSystem.DRAWING_ENABLED) {
    this.context.putImageData(this.imageData, 0, 0);
  }

```

```
  this.particles = newParticles;
  window.console.log(this.particles.length);
};

/**
 * @param {!smash.ParticleEmitter} emitter
 */
smash.ParticleSystem.prototype.addEmitter = function(emitter) {
  this.emitters.push(emitter);
};
```

Listing B.7: Initial particle system object in JavaScript

```
/**
 * @fileoverview Particle system.
 * @author sebastian.poreba@gmail.com (Sebastian Poreba)
 */
#include "particleSystem.h"

smash::ParticleSystem::ParticleSystem() {
  this->particles = new std::vector<smash::Particle*>;
  this->emitters = new std::vector<smash::ParticleEmitter*>;
};

smash::ParticleSystem::~ParticleSystem() {
  this->particles->erase(this->particles->begin(), this->particles->end());
  delete this->particles;
  this->emitters->erase(this->emitters->begin(), this->emitters->end());
  delete this->emitters;
};

void smash::ParticleSystem::step() {
  for (std::vector<smash::ParticleEmitter*>::iterator it = this->emitters->begin(); it != this->emitt
    std::vector<smash::Particle*>* particleFromEmitters = (*it)->getNewParticles();
    this->particles->insert(this->particles->end(), particleFromEmitters->begin(), particleFromEmitte
  }

  std::vector<smash::Particle*>* newParticles = new std::vector<smash::Particle*>;
  for (std::vector<smash::Particle*>::iterator it = this->particles->begin(); it != this->particles->
    smash::Particle* p = *it;
    p->step();
    if (p->positionX >= 0 &&
        p->positionX < smash::ParticleSystem::CANVAS_WIDTH &&
        p->positionY >= 0 &&
        p->positionY < smash::ParticleSystem::CANVAS_HEIGHT &&
        p->age < p->lifespan) {
      newParticles->push_back(p);
    }
  };
  this->particles = newParticles;
};

/**
 * @param emitter
 */
```

```
void smash::ParticleSystem::addEmitter(smash::ParticleEmitter* emitter) {
  this->emitters->push_back(emitter);
};
```

Listing B.8: Initial particle system in C++

```
/**
 * @fileoverview Particle system.
 * @author sebastian.poreba@gmail.com (Sebastian PorÄ™ba)
 */

goog.provide('smash.ParticleSystem2');

goog.require('smash.Particle');
goog.require('smash.ParticleEmitter');


/**
 * @constructor
 */
smash.ParticleSystem2 = function() {
  /**
   * @type {!Array.<smash.Particle>}
   */
  this.particles = [];

  /**
   * @type {!Array.<number>}
   */
  this.deadParticles = [];

  /**
   * @type {!Array.<smash.ParticleEmitter>}
   */
  this.emitters = [];

  if (smash.ParticleSystem2.DRAWING_ENABLED) {
    /**
     * @type {!Element}
     */
    this.canvas = window.document.createElement("canvas");
    this.canvas.width = smash.ParticleSystem2.CANVAS_WIDTH;
    this.canvas.height = smash.ParticleSystem2.CANVAS_HEIGHT;
    window.document.body.appendChild(this.canvas);

    /**
     * @type {!CanvasRenderingContext2D}
     */
    this.context = this.canvas.getContext('2d');

    /**
     * @type {!ImageData}
     */
    this.imageData = this.context.getImageData(0, 0,
        smash.ParticleSystem2.CANVAS_WIDTH, smash.ParticleSystem2.CANVAS_HEIGHT);
```

```

    /**
     * @type {!CanvasPixelArray}
     */
    this.pixels = this.imageData.data;
  }
};

/**
 * @const {boolean}
 */
smash.ParticleSystem2.DRAWING_ENABLED = true;

/**
 * @const {number}
 */
smash.ParticleSystem2.CANVAS_WIDTH = 1200;

/**
 * @const {number}
 */
smash.ParticleSystem2.CANVAS_HEIGHT = 400;


smash.ParticleSystem2.prototype.step = function() {
  if (smash.ParticleSystem2.DRAWING_ENABLED) {
    for (var i = 0; i < smash.ParticleSystem2.CANVAS_WIDTH *
        smash.ParticleSystem2.CANVAS_HEIGHT * 4; i+=4) {
      this.pixels[i] = 0;
      this.pixels[i + 1] = 0;
      this.pixels[i + 2] = 0;
      this.pixels[i + 3] = 0;
    }
  }

  for (var ei = 0; ei < this.emitters.length; ei++) {
    var emitter = this.emitters[ei];
    for (var i = 0; i < emitter.productionRate; i++) {
      var pIndex = this.deadParticles.pop();
      if (pIndex !== undefined) {
        var p = this.particles[pIndex];
        p.reset();
      } else {
        var p = new smash.Particle();
        this.particles.push(p);
      }

      p.lifespan = emitter.lifespan;
      p.positionX = emitter.positionX;
      p.positionY = emitter.positionY;
      p.velocityX = Math.sin(emitter.angle +
          (Math.random() - 0.5) * emitter.spread) *
          emitter.velocity * emitter.velocitySpread;
      p.velocityY = Math.cos(emitter.angle +
          (Math.random() - 0.5) * emitter.spread) *
          emitter.velocity *
          (1 + (Math.random() - 0.5) * emitter.velocitySpread);
```

```
    }
  }

  for (var i = 0; i < this.particles.length; i++) {
    var p = this.particles[i];
    p.step();
    if (p.positionX < 0 ||
        p.positionX >= smash.ParticleSystem2.CANVAS_WIDTH ||
        p.positionY < 0 ||
        p.positionY >= smash.ParticleSystem2.CANVAS_HEIGHT ||
        p.age > p.lifespan) {
      this.deadParticles.push(i);
      p.isDead = true;
    }

    if (smash.ParticleSystem2.DRAWING_ENABLED && !p.isDead) {
      var baseIndex =
          (Math.round(p.positionY) *
              smash.ParticleSystem2.CANVAS_WIDTH +
              Math.round(p.positionX)) * 4;
      this.pixels[baseIndex] = Math.round(p.velocityX * 80);
      this.pixels[baseIndex + 1] = Math.round(p.velocityX * 80);
      this.pixels[baseIndex + 2] = 255 - Math.round(p.age / p.lifespan * 255);
      this.pixels[baseIndex + 3] = 255;
    }
  }

  if (smash.ParticleSystem2.DRAWING_ENABLED) {
    this.context.putImageData(this.imageData, 0, 0);
  }
};

/**
 * @param {!smash.ParticleEmitter} emitter
 */
smash.ParticleSystem2.prototype.addEmitter = function(emitter) {
  this.emitters.push(emitter);
};
```

Listing B.9: Optimised particle system object in JavaScript

```
/**
 * @fileoverview Particle system.
 * @author sebastian.poreba@gmail.com (Sebastian Poreba)
 */
#include "particleSystem2.h"

smash::ParticleSystem2::ParticleSystem2() {
  this->particles = new std::vector<smash::Particle*>;
  this->deadParticles = new std::stack<smash::Particle*>;
  this->emitters = new std::vector<smash::ParticleEmitter*>;
};

smash::ParticleSystem2::~ParticleSystem2() {
  this->particles->erase(this->particles->begin(), this->particles->end());
```

```
  delete this->particles;
  while (!this->deadParticles->empty()) {
    delete this->deadParticles->top();
    this->deadParticles->pop();
  }
  delete this->deadParticles;
  this->emitters->erase(this->emitters->begin(), this->emitters->end());
  delete this->emitters;
};

void smash::ParticleSystem2::step() {
  for (std::vector<smash::ParticleEmitter*>::iterator it = this->emitters->begin(); it != this->emitt
    smash::ParticleEmitter* emitter = *it;
    for (int i = 0; i < emitter->productionRate; i++) {
      smash::Particle* p;
      if (!this->deadParticles->empty()) {
        p = this->deadParticles->top();
        this->deadParticles->pop();
        p->reset();
      } else {
        p = new smash::Particle();
        this->particles->push_back(p);
      }

      p->lifespan = emitter->lifespan;
      p->positionX = emitter->positionX;
      p->positionY = emitter->positionY;
      p->velocityX = sin(emitter->angle +
          (((float) rand() / (RAND_MAX)) - 0.5) * emitter->spread) *
          emitter->velocity *
          (1 + (((float) rand() / (RAND_MAX)) - 0.5) * emitter->velocitySpread);
      p->velocityY = cos(emitter->angle +
          (((float) rand() / (RAND_MAX)) - 0.5) * emitter->spread) *
          emitter->velocity *
          (1 + (((float) rand() / (RAND_MAX)) - 0.5) * emitter->velocitySpread);
    }
  }

  for (std::vector<smash::Particle*>::iterator it = this->particles->begin(); it != this->particles->
    smash::Particle* p = *it;
    if (p->isDead) {
      continue;
    }
    p->step();
    if (p->positionX < 0 ||
        p->positionX >= smash::ParticleSystem2::CANVAS_WIDTH ||
        p->positionY < 0 ||
        p->positionY >= smash::ParticleSystem2::CANVAS_HEIGHT ||
        p->age > p->lifespan) {
      this->deadParticles->push(p);
      p->isDead = true;
    }
  };
};

/**
 * @param emitter
```

```
 */
void smash::ParticleSystem2::addEmitter(smash::ParticleEmitter* emitter) {
  this->emitters->push_back(emitter);
};
```

Listing B.10: Optimised particle system in C++

## B.3. Spheres collision detection

```
/**
 * @fileoverview Sphere file.
 * @author sebastian.poreba@gmail.com (Sebastian PorÄ™ba)
 */


goog.provide('smash.Sphere');



/**
 * @struct
 * @constructor
 */
smash.Sphere = function() {
  /**
   * @type {number}
   */
  this.positionX = 0.1;

  /**
   * @type {number}
   */
  this.positionY = 0.1;

  /**
   * @type {number}
   */
  this.positionZ = 0.1;

  /**
   * @type {number}
   */
  this.velocityX = 0.1;

  /**
   * @type {number}
   */
  this.velocityY = 0.1;

  /**
   * @type {number}
   */
  this.velocityZ = 0.1;

  /**
   * @type {number}
   */
  this.radius = 5.5;
};


```

```
/**
 *
 */
smash.Sphere.prototype.step = function(stepTime) {
  this.positionX += this.velocityX * stepTime;
  this.positionY += this.velocityY * stepTime;
  this.positionZ += this.velocityZ * stepTime;
};
```

Listing B.11: Sphere object in JavaScript

```
/**
 * @fileoverview Sphere file.
 * @author sebastian.poreba@gmail.com (Sebastian PorÄ™ba)
 */

#include "sphere.h"

smash::Sphere::Sphere() {
  positionX = 0;
  positionY = 0;
  positionZ = 0;
  velocityX = 0;
  velocityY = 0;
  velocityZ = 0;
  radius = 5.5;
};

/**
 * @param stepTime
 */
void smash::Sphere::step(float stepTime) {
  this->positionX += this->velocityX * stepTime;
  this->positionY += this->velocityY * stepTime;
  this->positionZ += this->velocityZ * stepTime;
};
```

Listing B.12: Sphere object in C++

```
/**
 * @fileoverview Sphere collision detection system.
 * @author sebastian.poreba@gmail.com (Sebastian PorÄ™ba)
 */

goog.provide('smash.SphereSystem');

goog.require('smash.Sphere');
goog.require('smash.math');



/**
 * @constructor
 */
```

```
smash.SphereSystem = function() {
  var generalVelocity = 1;
  /**
   * @type {!Array.<!smash.Sphere>}
   */
  this.spheres = new Array(smash.SphereSystem.SPHERES_COUNT);
  for (var i = 0; i < smash.SphereSystem.SPHERES_COUNT; i++) {
    var sphere = new smash.Sphere();
    sphere.positionX = (Math.random() - 0.5) * 400;
    sphere.positionY = (Math.random() - 0.5) * 200;
    sphere.positionZ = (Math.random() - 0.5) * 100;
    sphere.velocityX = (Math.random() - 0.5) * generalVelocity;
    sphere.velocityY = (Math.random() - 0.5) * generalVelocity;
    sphere.velocityZ = (Math.random() - 0.5) * generalVelocity;
    this.spheres[i] = sphere;
  }

  if (smash.SphereSystem.DRAWING_ENABLED) {
    /**
     * @type {!THREE.PerspectiveCamera}
     */
    this.camera = new THREE.PerspectiveCamera(20,
        smash.SphereSystem.CANVAS_WIDTH /
            smash.SphereSystem.CANVAS_HEIGHT,
        1, 10000);
    this.camera.position.z = 1000;

    var controls = new THREE.OrbitControls(this.camera);
    controls.addEventListener('change', goog.bind(function() {
      this.renderer.render(this.scene, this.camera);
    }, this));

    /**
     * @type {!THREE.Scene}
     */
    this.scene = new THREE.Scene();

    var spotLight = new THREE.PointLight(0xffffff);
    spotLight.position.set(-40, 60, -10);
    this.scene.add(spotLight);

    var axes = new THREE.AxisHelper(20);
    this.scene.add(axes);

    var planeGeometry = new THREE.PlaneGeometry(
        10000, 10000, 100, 100);
    var planeMaterial = new THREE.MeshBasicMaterial({
      color: 0xcccccc,
      wireframe: true
    });

    var plane = new THREE.Mesh(planeGeometry, planeMaterial);
    plane.rotation.x = -0.5 * Math.PI;
    plane.position.x = 0;
    plane.position.y = smash.SphereSystem.FLOOR_LEVEL;
    plane.position.z = 0;

```

```
    this.scene.add(plane);



    var material = new THREE.MeshLambertMaterial({
      color: 0xff0000
    });
    /**
     * @type {!Array.<!THREE.SphereGeometry>}
     */
    this.threeSpheres = new Array(smash.SphereSystem.SPHERES_COUNT);
    for (var i = 0; i < smash.SphereSystem.SPHERES_COUNT; i++) {
      var sphere = new THREE.SphereGeometry(this.spheres[i].radius, 10, 10);
      var mesh = new THREE.Mesh(sphere, material);
      mesh.position.x = this.spheres[i].positionX;
      mesh.position.y = this.spheres[i].positionY;
      mesh.position.z = this.spheres[i].positionZ;
      this.threeSpheres[i] = mesh;

      this.scene.add(mesh);
    }



    /**
     * @type {!THREE.WebGLRenderer}
     */
    this.renderer = new THREE.WebGLRenderer();
    this.renderer.setSize(smash.SphereSystem.CANVAS_WIDTH,
        smash.SphereSystem.CANVAS_HEIGHT);
    document.body.appendChild(this.renderer.domElement);
    this.renderer.render(this.scene, this.camera);
  }

  /**
   * @type {number}
   */
  this.collisions = 0;
};


/**
 * @const {number}
 */
smash.SphereSystem.SPHERES_COUNT = 1000;


/**
 * @const {boolean}
 */
smash.SphereSystem.DRAWING_ENABLED = false;


/**
 * @const {boolean}
 */
smash.SphereSystem.GRAVITY_ENABLED = true;
```

```


/**
 * @const {number}
 */
smash.SphereSystem.GRAVITY_FORCE = 0.1;


/**
 * @const {number}
 */
smash.SphereSystem.FLOOR_LEVEL = -100;


/**
 * (1 - energy lost on floor hit)
 * @const {number}
 */
smash.SphereSystem.FLOOR_FRICTON = 0.8;


/**
 * @const {number}
 */
smash.SphereSystem.CANVAS_WIDTH = 1200;


/**
 * @const {number}
 */
smash.SphereSystem.CANVAS_HEIGHT = 400;


/**
 * @param {!smash.Sphere} sphere1
 * @param {!smash.Sphere} sphere2
 */
smash.SphereSystem.collide = function(sphere1, sphere2) {
  var sumVelocitiesLength = smash.math.vectorLength(
      sphere1.velocityX, sphere1.velocityY, sphere1.velocityZ) +
      smash.math.vectorLength(
          sphere2.velocityX, sphere2.velocityY, sphere2.velocityZ);

  var centerDiffX = sphere1.positionX - sphere2.positionX;
  var centerDiffY = sphere1.positionY - sphere2.positionY;
  var centerDiffZ = sphere1.positionZ - sphere2.positionZ;
  var centerLength = smash.math.vectorLength(
      centerDiffX, centerDiffY, centerDiffZ);
  var velocityAdjust = centerLength * sumVelocitiesLength / 2;
  centerDiffX /= velocityAdjust;
  centerDiffY /= velocityAdjust;
  centerDiffZ /= velocityAdjust;

  sphere1.velocityX = centerDiffX;
  sphere1.velocityY = centerDiffY;
  sphere1.velocityZ = centerDiffZ;
  sphere2.velocityX = centerDiffX * -1;
```

```
  sphere2.velocityY = centerDiffY * -1;
  sphere2.velocityZ = centerDiffZ * -1;
};


/**
 * @param {!smash.Sphere} sphere
 */
smash.SphereSystem.prototype.applyGravity = function(sphere) {
  if (smash.SphereSystem.GRAVITY_ENABLED) {
    sphere.velocityY -= smash.SphereSystem.GRAVITY_FORCE;
  }
};


/**
 * @param {!smash.Sphere} sphere
 */
smash.SphereSystem.prototype.applyFloor = function(sphere) {
  if (sphere.positionY - sphere.radius <
      smash.SphereSystem.FLOOR_LEVEL) {
    sphere.velocityY *= -smash.SphereSystem.FLOOR_FRICTON;
  }
};


/**
 *
 */
smash.SphereSystem.prototype.step = function() {
  for (var i = 0; i < smash.SphereSystem.SPHERES_COUNT; i++) {
    this.applyGravity(this.spheres[i]);
    this.applyFloor(this.spheres[i]);


    this.spheres[i].step(1);
    for (var j = 0; j < smash.SphereSystem.SPHERES_COUNT; j++) {
      if (i != j &&
          smash.math.checkCollidingSpheres(this.spheres[i], this.spheres[j])) {
        this.collisions++;
        smash.SphereSystem.collide(this.spheres[i], this.spheres[j]);
      }
    }

    if (smash.SphereSystem.DRAWING_ENABLED) {
      this.threeSpheres[i].position.x = this.spheres[i].positionX;
      this.threeSpheres[i].position.y = this.spheres[i].positionY;
      this.threeSpheres[i].position.z = this.spheres[i].positionZ;
    }
  }

  if (smash.SphereSystem.DRAWING_ENABLED) {
    this.renderer.render(this.scene, this.camera);
  }
};
```

Listing B.13: Sphere system object in JavaScript

```
/**
 * @fileoverview Sphere collision detection system.
 * @author sebastian.poreba@gmail.com (Sebastian PorÄ™ba)
 */

#include "sphereSystem.h"

smash::SphereSystem::SphereSystem() {
  this->spheres = new std::vector<smash::Sphere*>;
  collisions = 0;

  float generalVelocity = 1;
    for (int i = 0; i < smash::SphereSystem::SPHERES_COUNT; i++) {
    smash::Sphere* sphere = new smash::Sphere();
    sphere->positionX = (((float) rand() / (RAND_MAX)) - 0.5) * 400;
    sphere->positionY = (((float) rand() / (RAND_MAX)) - 0.5) * 200;
    sphere->positionZ = (((float) rand() / (RAND_MAX)) - 0.5) * 100;
    sphere->velocityX = (((float) rand() / (RAND_MAX)) - 0.5) * generalVelocity;
    sphere->velocityY = (((float) rand() / (RAND_MAX)) - 0.5) * generalVelocity;
    sphere->velocityZ = (((float) rand() / (RAND_MAX)) - 0.5) * generalVelocity;
    this->spheres->push_back(sphere);
  }
};


smash::SphereSystem::~SphereSystem() {
  this->spheres->erase(this->spheres->begin(), this->spheres->end());
  delete this->spheres;
};


/**
 * @param {smash::Sphere*} sphere1
 * @param {smash::Sphere*} sphere2
 */
void smash::SphereSystem::collide(smash::Sphere* sphere1, smash::Sphere* sphere2) {
  float sumVelocitiesLength = smash::math::vectorLength(
      sphere1->velocityX, sphere1->velocityY, sphere1->velocityZ) +
      smash::math::vectorLength(
          sphere2->velocityX, sphere2->velocityY, sphere2->velocityZ);

  float centerDiffX = sphere1->positionX - sphere2->positionX;
  float centerDiffY = sphere1->positionY - sphere2->positionY;
  float centerDiffZ = sphere1->positionZ - sphere2->positionZ;
  float centerLength = smash::math::vectorLength(
      centerDiffX, centerDiffY, centerDiffZ);
  float velocityAdjust = centerLength * sumVelocitiesLength / 2;
  centerDiffX /= velocityAdjust;
  centerDiffY /= velocityAdjust;
  centerDiffZ /= velocityAdjust;

  sphere1->velocityX = centerDiffX;
  sphere1->velocityY = centerDiffY;
  sphere1->velocityZ = centerDiffZ;
  sphere2->velocityX = centerDiffX * -1;
  sphere2->velocityY = centerDiffY * -1;
```

```
  sphere2->velocityZ = centerDiffZ * -1;
};

void smash::SphereSystem::applyGravity(smash::Sphere* sphere) {
  if (smash::SphereSystem::GRAVITY_ENABLED) {
    sphere->velocityY -= smash::SphereSystem::GRAVITY_FORCE;
  }
};


void smash::SphereSystem::applyFloor(smash::Sphere* sphere) {
  if (sphere->positionY - sphere->radius <
      smash::SphereSystem::FLOOR_LEVEL) {
    sphere->velocityY *= -smash::SphereSystem::FLOOR_FRICTON;
  }
};


void smash::SphereSystem::step() {
  for (std::vector<smash::Sphere*>::iterator it = this->spheres->begin(); it != this->spheres->end();
    smash::Sphere* s = *it;
    applyGravity(s);
    applyFloor(s);


    s->step(1);
    for (std::vector<smash::Sphere*>::iterator it2 = this->spheres->begin(); it2 != this->spheres->en
      smash::Sphere* s2 = *it2;
      if (s != s2 &&
          smash::math::checkCollidingSpheres(s, s2)) {
        this->collisions++;
        smash::SphereSystem::collide(s, s2);
      }
    }
  }
};
```

Listing B.14: Sphere system in C++

# Bibliografia


[1] Lista selektorów zgodnych z jQuery
http://api.jquery.com/category/selectors/.

[2] Przykładowe zastosowanie rachunku macierzowego w OpenGL
http://www.opengl.org/sdk/docs/man/xhtml/glortho.xml.

[3] Rodzaje projekcji 3D
http://frank.mtsu.edu/ csjudy/planeview3d/tutorial-parallel.html.

[4] Specyfikacja 3DXML
http://www.3ds.com/products/3dvia/3d-xml/overview/.

[5] Specyfikacja Javascript dla <canvas>
http://www.whatwg.org/specs/web-apps/current-work/multipage/the-canvas-element.html.

[6] Specyfikacja X3D http://www.web3d.org/x3d/specifications/.

[7] Strona projektu Collada
http://www.khronos.org/collada/.

[8] Wykład Metody matematyczne w grafice komputerowej - dr inż. Marcin Szpyrka.

[9] COLLADA 1.5.0 specification, 2008.

[10] A. Brandon. Wtyczka jQuery Mousewheel
http://brandonaaron.net/code/mousewheel/demos.

[11] W. Gajda. *JQuery. Poradnik programisty.* Helion, 2010.

[12] K. Group. Grupa Khronos
http://www.khronos.org/webgl/.

[13] K. Group. Specyfikacja WebGL
https://cvs.khronos.org/svn/repos/registry/trunk/public/webgl/doc/spec/webgl-spec.html.

[14] A. LaMothe. *Triki najlepszych programistów gier 3D. Vademecum profesjonalisty*. Helion.

[15] E. Lengyel. *Mathematics for 3D Game Programming and Computer Graphics*. Charles River Media, 2004.

[16] Mozilla. Testowe implementacje Canvas 3D
http://blog.vlad1.com/2007/11/26/canvas-3d-gl-power-web-style/, 2006.

[17] D. R. Tim Berners-Lee. Pierwsza konferencja WWW w Genewie. 1999.