Uniwersytet Jagielloński

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Praca magisterska

Sebastian Poręba

Comparison of 3D physics engines

Promotor:
dr hab. Paweł Węgrzyn

Konsultant:
mgr Bartosz Porębski

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

I would like to thank my advisors Paweł Węgrzyn and Bartek Porębski. Also I need to thank Collin Hover and Ibon Tolosana for their valuable insights and help with tests.

Spis treści

1. Introduction 4
1.1. Influence on the distribution process 5
1.2. Technology 7
2. Overview of JavaScript and V8 engine architecture 8
2.1. JIT compilation – tracking variable types 9
2.2. Type inference 11
2.3. Hidden classes 12
2.4. Garbage collection 14
3. Particle system 15
3.1. System parameters 16
3.2. Implementation with high memory allocation 17
3.3. Implementation with object pool 21
4. Sphere collision 23
4.1. Algorithm description 24
4.2. $O(N^2)$ approach 27
4.3. Octree-partitioned space 29
5. Emscripten 31
5.1. Asm.js overview 34
6. Summary 36
A. Acknowledgements 39
B. Source code 40
B.1. Math utilities 40
B.2. Particle system 43
B.3. Spheres collision detection 57
References 83

1. Introduction

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

At the moment of writing, the majority of games are developed with C++ and usually DirectX. These technologies are mature and have excellent performance. Features of the language give flexibility and high-level solutions to effectively manage application (e.g. operator overloading, multiple inheritance) and enable to fine-tune the internals of application to achieve best execution times. For games with high budget[1] C++ is an obvious choice for having the best possible result.

However, in parallel to the AAA game industry, the casual and independent games sector is growing. In 2013, the market is expected to be worth $8.64 billions in total. A total of 2.4 billion tablets and smartphones with casual games capabilities will be reached before the end of 2013. These games are less focused on creating cutting edge graphics and physics simulations and more on overall experience and social interactions.

JavaScript is a scripting language not designed to perform high-load computations. However, at present it is the only language widely supported by all browsers. With all it's advantages and quirks is the only choice available for programmers.[1]

While suffering from design issues, JavaScript provides a complete environment that makes development very easy for both beginner and advanced programmers. Two very important components of every application are provided out of the box – a rendering system and networking in the browser. They were designed for HTML pages to carry mainly text information, they are still suitable for gaming. Building a simple 2D game is often a matter of few hundred lines of code responsible for transferring user input to the positions of sprites defined in CSS. This is clearly visible during competitions like JS13kGames[2] where all game assets and the code have to be fitted into 13kB package.

Many projects varying from server side solutions[3, 4] to hardware developer boards[5] are taking advantage of this simplicity. From the perspective of game development, it is unlikely at the time of writing that an AAA game may be created to run in the browser. However, growing segments of casual, independent and social games are already targeting the web as a platform.

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

1.1. Influence on the distribution process

Traditionally, games are often still sold in actual boxes and some update system is always incorporated to patch any bugs appearing after initial release. Systems like Steam[6] are making this process easier, but still suffer from the necessity of having to install a game on the hard drive.

Creating an application that works in the browser simplifies distribution significantly. All assets and code are downloaded each time the user enters a website, so no update system is necessary – all users are always playing the latest version. Usually browser games are monetised differently to traditional titles. Playing a basic version is usually free and earnings come from either ads or premium content. This is completely new approach, present also in MMO[2] games. It resulted in psychological research on leveraging compulsive behaviours to maximise profits[7].

Working in a browser gives access to all social networks of users, so usage of Twitter or Facebook based features is very simple – which boosts the promotion of the game. It also enables, though morally questionable, to target people who are not able to install games on company issued computers. Disallowing games in browsers is a far more complicated task for administrator, similar to blocking ads or mature content.

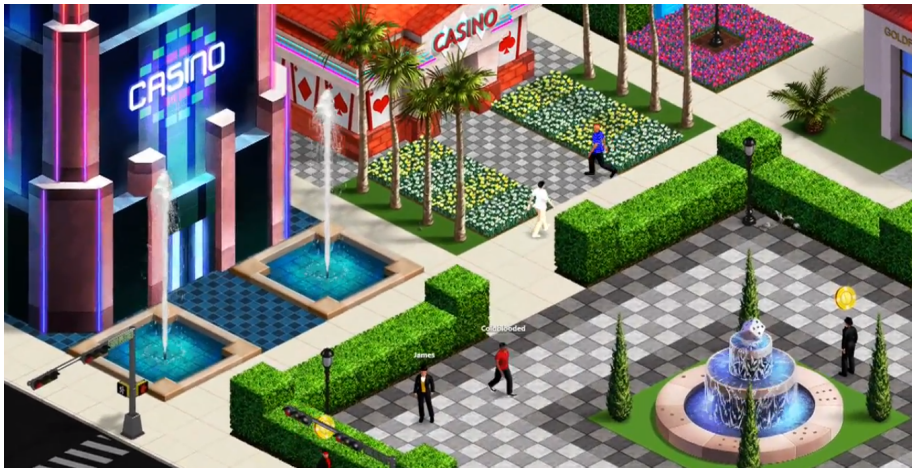
Lastly, the web is better suited to run easily on all platforms. The browser is a layer of abstraction that is transparent for the application, whether it runs on any traditional operating system, gaming console or mobile device. Of course performance and screen size should be taken into consideration, but ability to write one codebase that runs on multiple devices has already encouraged projects that package JavaScript applications as native ones[8], greatly reducing development costs for the growing variety of phones and tablets.

Rysunek 1.1: Game created with ImpactJS

Multiple open-source and commercial game engines are being created lately. Examples worth mentioning are ImpactJS[9], Turbulenz[10] and Isogenic Engine[11].

[2]Massive Multiplayer Online

Rysunek 1.2: Game created with Isogenic Engine



A very important and growing sector is interactive 3D arts with two major targets – music videos and commercials. They are uniquely available only in browsers as a very viral extensions of regular marketing. One of the first occurrences of this technology was the video for Rome music group: ”3 dreams of black”[12]. The project allows to move the camera while an animated 3D story is rendered alongside the music. After the movie is over, the user is allowed to create 3D models that are later incorporated into the experiences of other people watching. This way interactions and social element are enabled in what used to be a one-way transmission of art form.

Rysunek 1.3: Screenshot from ”3 dreams of black”

1.2. Technology

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

The project is based on conference sessions and announcements authored by V8 programmers regarding the performance of JavaScript applications. The analysis of available materials is a topic of Chapter 2, where internals of modern engines for dynamic languages are briefly explained.

Chapter 3 covers particles often used to simulate loosely connected systems like smoke, fire, fog, etc. It shows techniques of memory allocation and garbage collection that help improve performance. Two particle systems are presented – one with high memory allocation that is expected to cause performance issues and the second one, improved by the usage of object pools.

Chapter 4 focuses on sphere collision detection and reaction, with both naive solution and space partitioning using Octree. This benchmark shows an application with high CPU usage and relatively simple math computations. Because of this simplicity, collision detection with spheres is almost always used as a preliminary method of eliminating collision between objects. More complex and expensive algorithms are used to determine the real state of such a pair only if bounding spheres are colliding.

The systems presented in chapters 3 and 4 are not targeted to be full physics engines. They are however representative to general concepts encountered in every game.

Chapter 5 describes Emscripten, a project aimed to convert complete C++ projects to JavaScript. A related library, asm.js, is presented with an overview of architectural choices, which lead to better performance. The generated code is compared to the one created in previous chapters and execution times of all benchmarks are compared and briefly explained.

The last chapter is a summary of all achieved results. The limitations of JavaScript engines are presented alongside future possibilities for the gaming industry.

Benchmarks are by no means a complete physics system and do not represent the current state of the art of physics algorithms. They are aimed to resemble optimal algorithms in used methods and complexity, so that benchmarks reflect how an actual engine uses memory and computing power.

2. Overview of JavaScript and V8 engine architecture

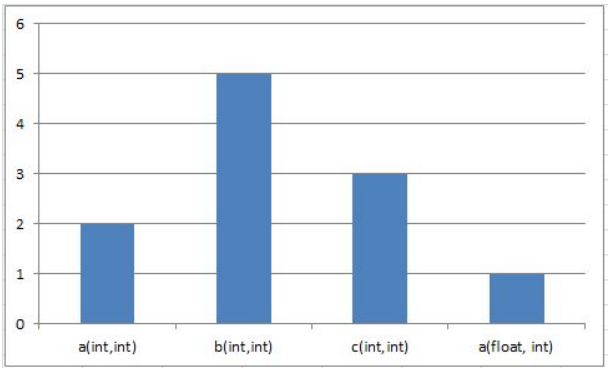
Historically, JavaScript was considered an untyped language, meaning that values had no types attached to variables, either by the programmer or the compiler. All variables were of a single, unified type, and procedures called unboxing and boxing, performed before and after each operation on a variable, ensured that it was properly used on the machine code level. The complete code source was sent from the server to the browser and was parsed and executed on the 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, the source code of the function was parsed every time it was called, each time generating a machine code based on current parameters and variables in scope. This approach, called interpretation, is still present in JavaScript engines, and used whenever variables do not match set criteria of stability described later in this chapter.

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

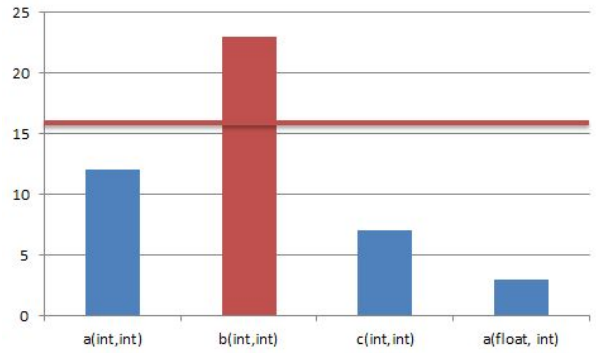
2.1. JIT compilation – tracking variable types

As it was mentioned before, initially JavaScript was treated as an untyped language. With the release of SpiderMonkey in Firefox 3.5 in 2009, the situation changed. The first Just-In-Time compiler for JavaScript, TraceMonkey, was created. Based on the works of Prof. Dr. Michael Franz on TraceTrees [13] JIT compiler collected all paths that the interpreter took with specific types of variables. A path could split into different methods or if statements. Whenever part of the code was executed often enough, the path was marked as hot, and the compiler optimised it for given types. If a single path was traversed with a different set of types, the compiler could generate another version of the optimised code. When the path turned out to be highly polymorphic, optimised versions were removed and the interpreter was used as a fallback. Initial reports show speedups between 20x to 40x [14]. However, trace JIT turned out to be very complicated to maintain [15] and eventually was removed from Firefox in 2011.[16]. At the time SpiderMonkey was already equipped with JagerMonkey, JIT engine based on method calls. Instead of collecting complete traces, only method calls are counted. This provides easy tracking 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 a simpler approach, which is now used in all JavaScript engines. In V8 Crankshaft, a step forward was taken, and simple methods are compiled even before any statistics on data types are collected. For compiled methods the source code is not stored. Instead, a procedure called deoptimisation is implemented. Whenever an engine detects that a compiled code does not match actual types of variables, the code is deoptimised and either optimised again to match the 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 inference

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

```
1  //          Unknown
2  //            |   \-------------
3  //            |                 |
4  //       Primitive         Non-primitive
5  //            |   \--------     |
6  //            |             |   |
7  //         Number         String |
8  //          /   \           |   |
9  //     Double  Integer32    |   /
10 //         |       |       /   /
11 //         |      Smi     /   /
12 //         |       |    / __/
13 //         Uninitialized.
```

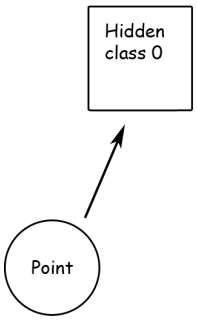
Listing 2.2: Tree of types in JavaScript

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

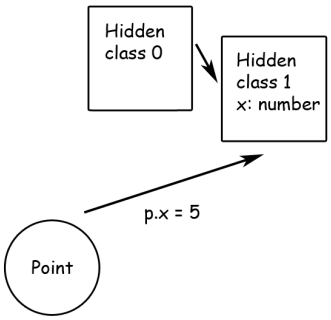
2.3. Hidden classes

JavaScript is a 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 its prototype at any point in runtime. To optimise such a dynamic representation, engines use the concept of hidden class. Whenever an Object is created, its hidden class is pointed to the base, empty Object representation. Then each definition of new property makes a transition on the hidden class graph, introducing hidden classes that are not yet defined, as in the following example:

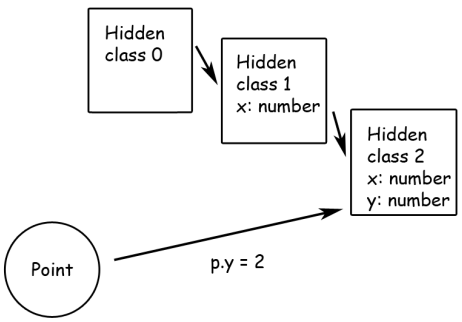
Rysunek 2.3: Initial shape of hidden class for Point



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



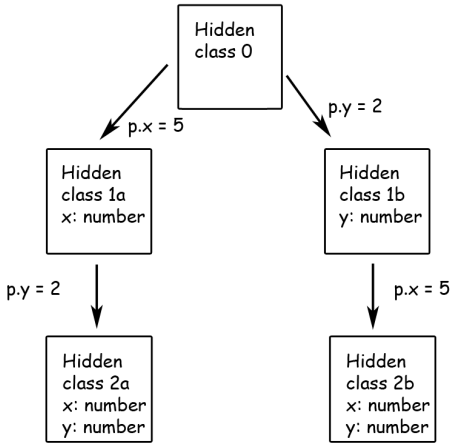
Rysunek 2.5: Shape of hidden class for Point after y property added



Based on hidden class the JIT compiler optimises methods to generate an even simpler assembly code similar to the one compiled from C++. Class shape defines address offsets of Object properties. Thus,

the hidden class graph is actually a tree, where one class cannot be reached in more than one way.

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



At the moment of writing, the 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 the hidden class of this object.

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

```
1 // TODO: update when it lands in V8
2
3 [generalizing xQ] I:s->d (+20 maps) [xQ.Kd+919 at :719]
4 [generalizing xQ] Si:s->d (+3 maps) [xQ.Kd+1057 at :719]
5 [migrating xQ] I:s->d Si:s->d
6 [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 places an object on a memory heap. The 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 takes a long time. Since JavaScript is single-threaded, this operation blocks all other operations. To improve performance, especially in games, the incremental scavange method of garbage collection was introduced. The engine tracks the age of objects, allowing to quickly detect objects allocated temporarily (e.g. for a single frame rendered in a game). When the object is inaccessible, it is queued for deallocation, in chunks that do not cause long UI freezes.[17, 18]

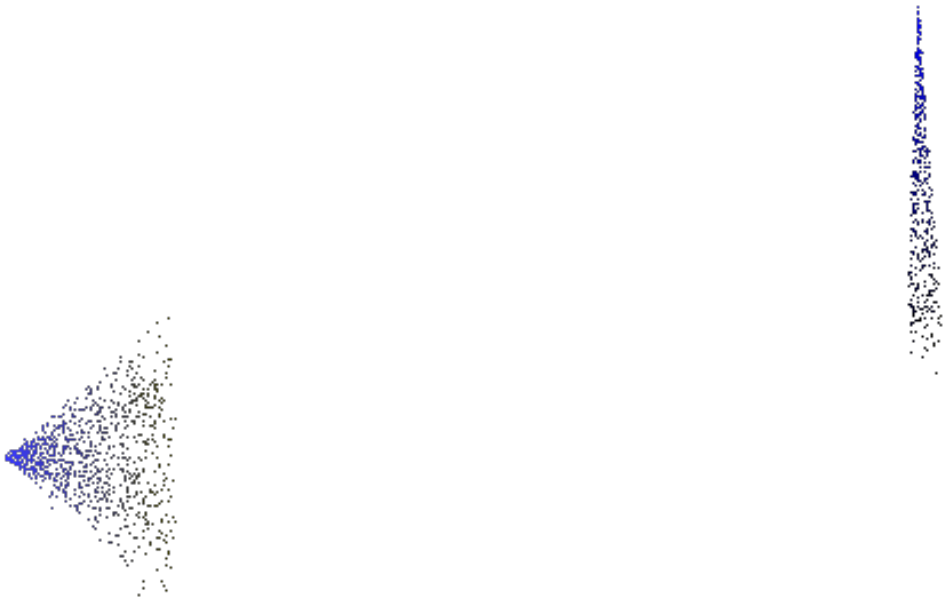
```
[1592]       34 ms: Scavenge 1.6 (18.8) -> 0.9 (18.8) MB, 0.0 ms [Runtime::PerformGC].
[1592]       37 ms: Scavenge 1.6 (18.8) -> 1.2 (19.8) MB, 1.0 ms [Runtime::PerformGC].
[1592]       40 ms: Scavenge 1.9 (19.8) -> 1.7 (19.8) MB, 1.0 ms [Runtime::PerformGC].
[1592]       43 ms: Scavenge 2.4 (19.8) -> 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

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. The system consists of a defined number of emitters, producing lightweight particle objects with certain parameters. Each emitter has a defined production ratio, and each particle a certain lifespan, resulting in the upper limit of total particles on the screen. Some systems also use 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 can be used for different effects with proper configuration.

Rysunek 3.1: Example rendering of tested particle system



3.1. System parameters

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

Emitter supports the following parameters:

- position – initial position of created particles
- angle – angle counting clockwise from the vector [0, 1]
- spread – parameter controlling random differences in the initial angle of particles
- velocity – initial velocity of particles, in pixels per frame
- velocity spread – parameter controlling random differences in the 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 for given particle it is higher than its lifespan, particle is removed from the system

Source code of both implementations is attached in the Appendix B.

3.2. Implementation with high memory allocation

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

```
$ 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 the JavaScript version is almost 8 times slower than the native one. To analyse the situation –prof and –log-timer-events flags may be used. The output file v8.log is parsed using an available online tool.[19]

```
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
    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 the profiler log, most of the time is spent in unoptimised versions of verifyIfAlive and stepParticle methods.

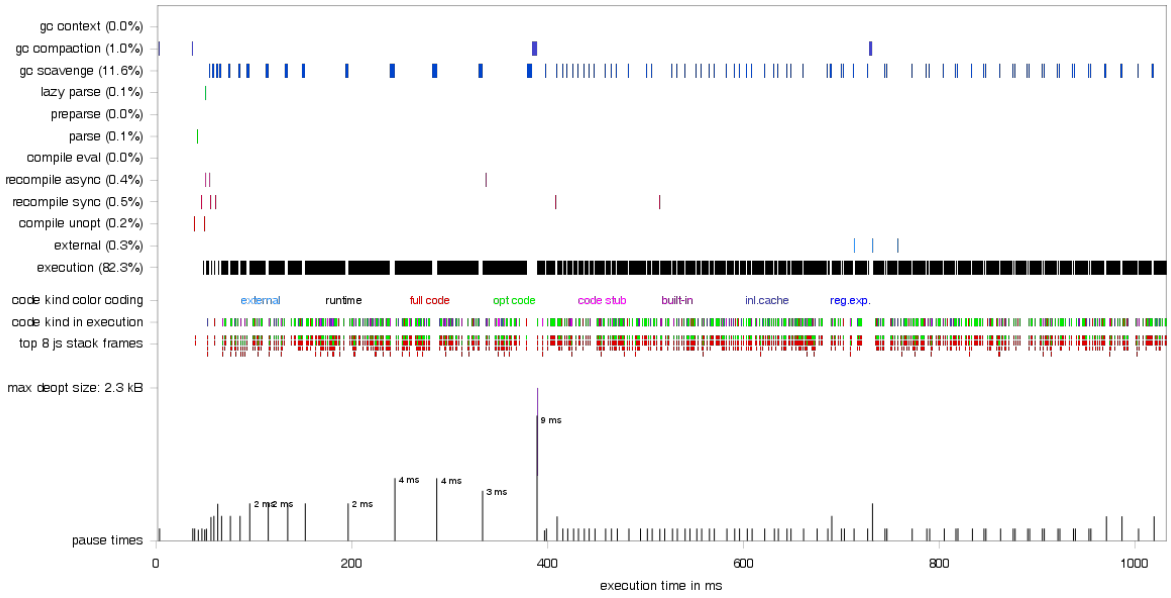
```
smash.ParticleSystem.prototype.step = function() {
  this.emitters.forEach(function appendNewParticles(a) {
    this.particles.push.apply(this.particles, a.getNewParticles())
  }, this);
  var newParticles = [];
  function stepParticle(a) {
```

```
7       a.step();
8     }
9     function verifyIfAlive(a) {
10      if (0 <= a.positionX && a.positionX < smash.ParticleSystem.CANVAS_WIDTH &&
11        0 <= a.positionY && a.positionY < smash.ParticleSystem.CANVAS_HEIGHT &&
12        a.age < a.lifespan) {
13      newParticles.push(a);
14    }
15    }
16    this.particles.forEach(function (a) {
17      stepParticle(a);
18      verifyIfAlive(a);
19    }, this);
20    this.particles = newParticles;
21  };
```

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 is clear that the presented code not only allocates and deallocates too much memory, but also fails to run in optimised mode. It is visible on the 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.

```
1  $ browser/static/d8 --trace-gc browser/static/particles1.js
2  [9696]       10 ms: Scavenge 1.6 (18.8) -> 0.9 (18.8) MB, 0.0 ms [Runtime::PerformGC].
3  [9696]       14 ms: Scavenge 1.6 (18.8) -> 1.3 (19.8) MB, 2.0 ms [Runtime::PerformGC].
4  (...)
5  [9696]      233 ms: Scavenge 15.0 (25.8) -> 9.7 (25.8) MB, 4.0 ms [Runtime::PerformGC].
6  [9696]      277 ms: Scavenge 15.6 (26.8) -> 10.0 (27.8) MB, 4.0 ms
7      (+ 13.0 ms in 22 steps since last GC) [Runtime::PerformGC].
8  [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

3.3. Implementation with object pool

To improve performance, a different approach to particles allocation is used. Each particle has a flag "isDead"telling if it may be safely reused for a new particle. The particle pool is kept along with a list of pointers to dead particles. This way, when the system reaches its maximum congestion (around 15000 particles in the given example) no new allocations occur. The creation of new particles is moved from the particle emitter to the particle system, to avoid allocation of the new array. The 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++

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

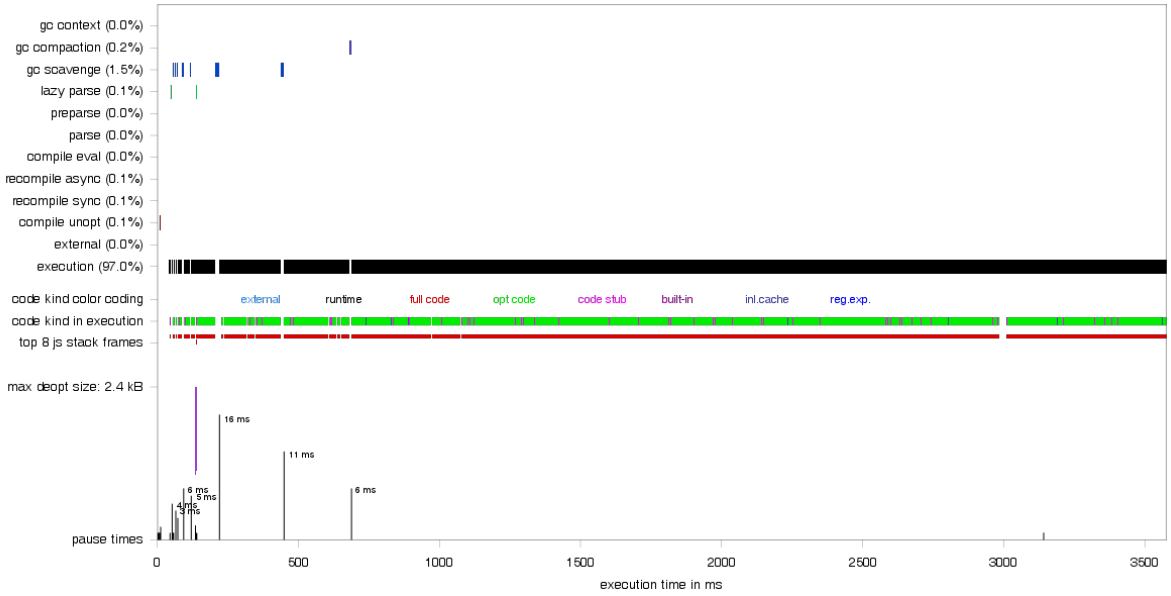
```
Statistical profiling result from null, (3780 ticks, 2 unaccounted, 0 excluded).

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

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



The situation has also improved in the garbage collection log.

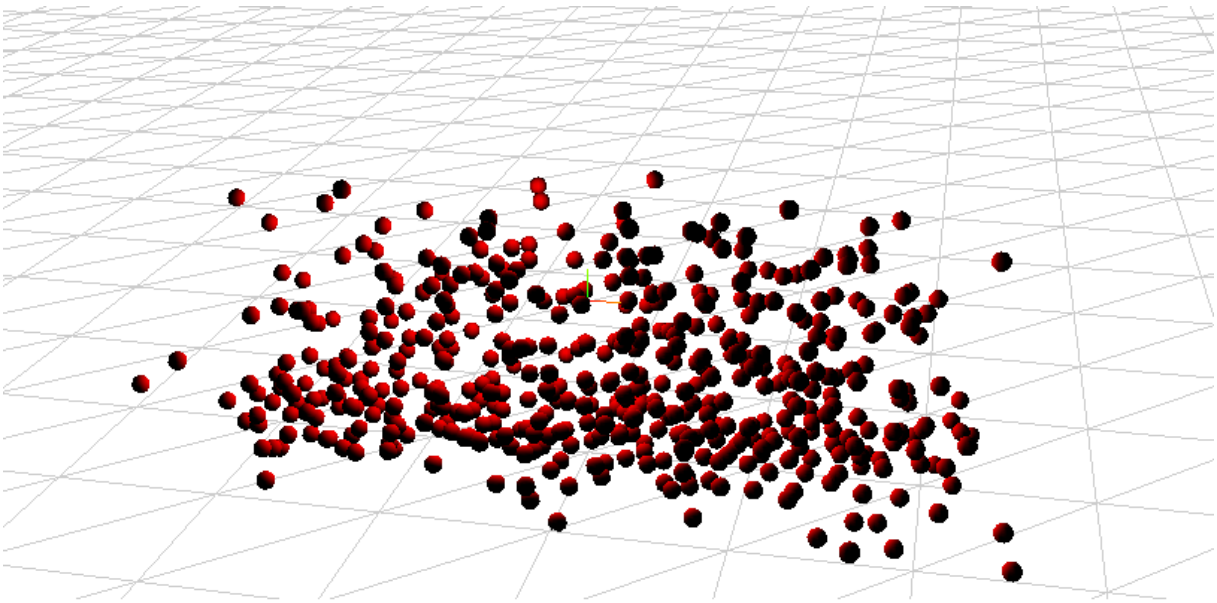
```
$ browser/static/d8 --trace-gc browser/static/particles2.js
[8348]        10 ms: Scavenge 1.6 (18.8) -> 0.9 (18.8) MB, 1.0 ms [Runtime::PerformGC].
[8348]        13 ms: Scavenge 1.6 (18.8) -> 1.2 (19.8) MB, 1.0 ms [Runtime::PerformGC].
[8348]        16 ms: Scavenge 1.9 (19.8) -> 1.7 (19.8) MB, 1.0 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 the simpler algorithm has a far greater number of collision checks per frame, it allocates almost no memory per frame. A more complex method will minimise the number of checks, but additional structure and steps added may influence overall execution time in an unexpected way.

Rysunek 4.1: Example rendering of tested sphere collision system



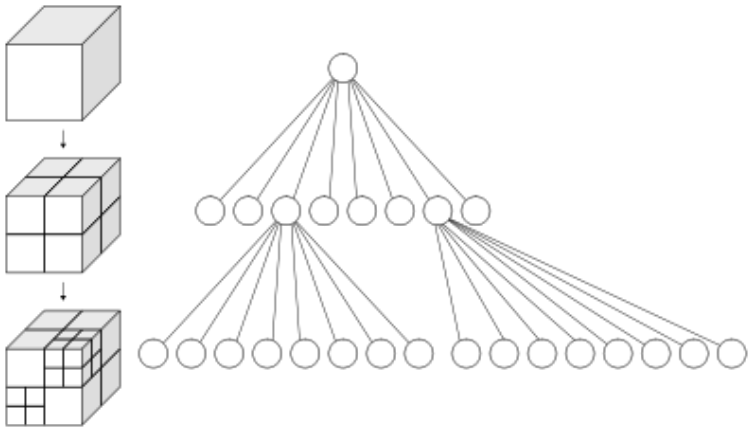
4.1. Algorithm description

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

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



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

When collision is detected, collision response is calculated. From the rule of conservation of momentum:

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

Meaning that change of both momentums is of equal value.

$$m_1 * \vec{v_1'} = m_1 * \vec{v_1} - \Delta P$$

$$m_2 * \vec{v_2'} = m_2 * \vec{v_2} + \Delta P$$

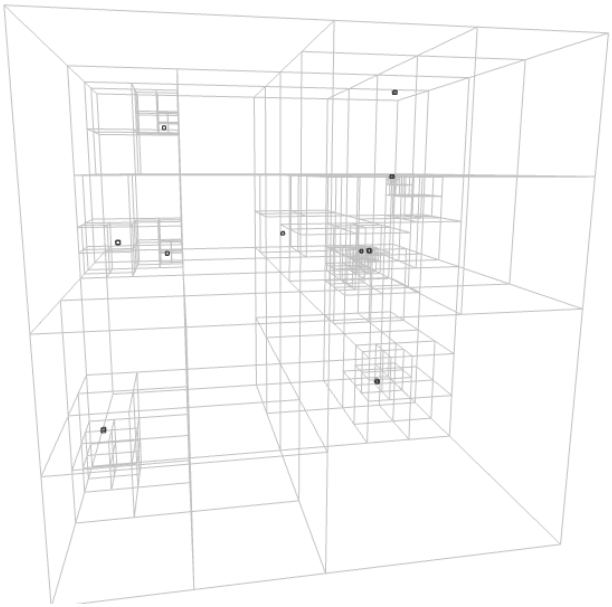
$$\vec{v_1'} = \vec{v_1} - \frac{\Delta P}{m_1}$$

$$\vec{v_2'} = \vec{v_2} + \frac{\Delta P}{m_2}$$

To simplify response, rotation and deformation of spheres are ignored. This does not affect performance analysis, since operations in tests are performed all in the same way.

Let

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



$$P = |\Delta P|$$

$$N = pos_1 \hat{-} pos_2$$

Since transference of momentum occurs only along single points of contact:

$$\Delta P = P * \hat{\vec{N}}$$

$$\vec{v_1'} = \vec{v_1} - \frac{P}{m_1} * \vec{N}$$

$$\vec{v_2'} = \vec{v_2} + \frac{P}{m_2} * \vec{N}$$

Let us split each velocity into two scalars, the perpendicular and parallel value of velocity vector, and introduce $\vec{Q}$, similar to $\vec{N}$, a perpendicular normalised vector lining along the exchanged momentum.

$$\vec{v_1} = a_1 * \vec{N} + b_1 * \vec{Q}$$

$$\vec{v_2} = a_2 * \vec{N} + b_2 * \vec{Q}$$

$$\vec{v_1'} = a_1' * \vec{N} + b_1' * \vec{Q}$$

$$\vec{v_2'} = a_2' * \vec{N} + b_2' * \vec{Q}$$

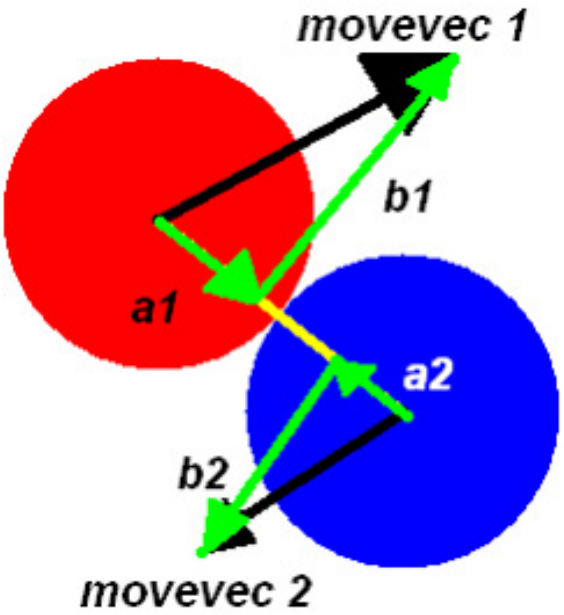
Deriving from previous equations:

$$a_1' = a_1 - \frac{P}{m_1}$$

$$b_1' = b_1$$

$$a_2' = a_2 + \frac{P}{m_2}$$

Rysunek 4.4: Illustration for collision response



$$b'_2 = b_2$$

Now let us use the rule of energy conservation to solve P:

$$\frac{m_1}{2} * ||\vec{v_1}||^2 + \frac{m_2}{2} * ||\vec{v_2}||^2 = \frac{m_1}{2} * ||\vec{v'_1}||^2 + \frac{m_2}{2} * ||\vec{v'_2}||^2$$

$$\frac{m_1}{2} * ({a_1}^2 + {b_1}^2) + \frac{m_2}{2} * ({a_2}^2 + {b_2}^2) = \frac{m_1}{2} * ({a'_1}^2 + {b'_1}^2) + \frac{m_2}{2} * ({a'_2}^2 + {b'_2}^2)2$$

$$P = \frac{2*m_1*m_2*(a_1-a_2)}{m_1+m_2}$$

and finally, using the result from the conservation of momentum:

$$\vec{v'_1} = \vec{v_1} - \frac{2*(a_1-a_2)}{m_1+m_2} * m_2 * \vec{N}$$

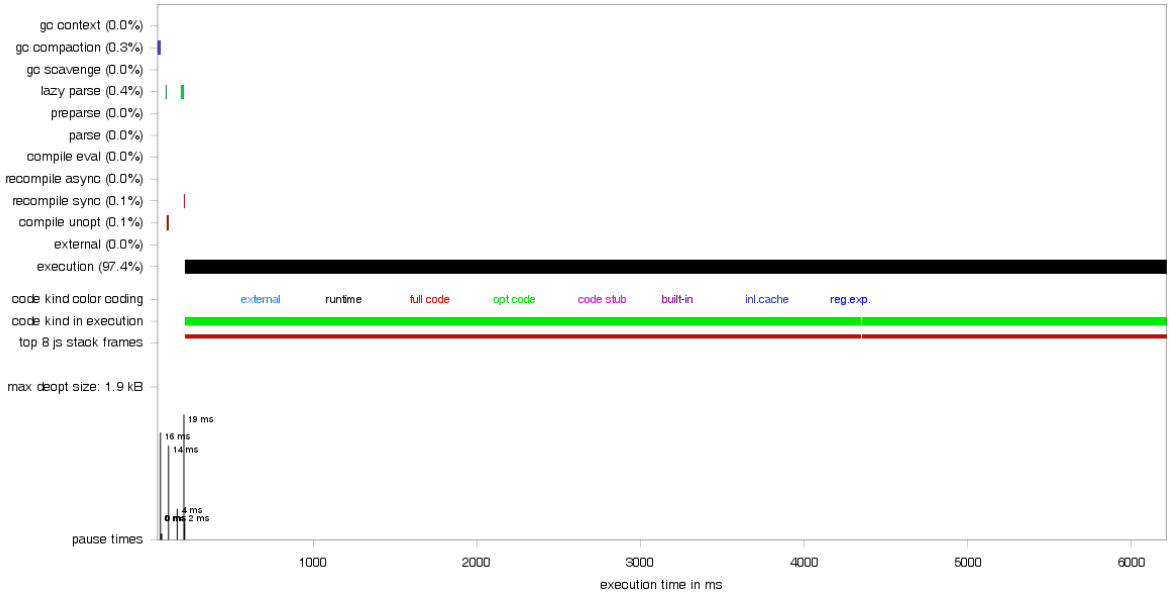
$$\vec{v'_2} = \vec{v_2} + \frac{2*(a_1-a_2)}{m_1+m_2} * m_1 * \vec{N}$$

From this result, using only the dot product of velocity vectors and normalised vector $pos_1 - pos_2$ the correct response to collision is calculated. In tested scenarios, the mass of all spheres is equal since it does not affect the complexity of calculations and produces less random results.

4.2. $O(N^2)$ approach

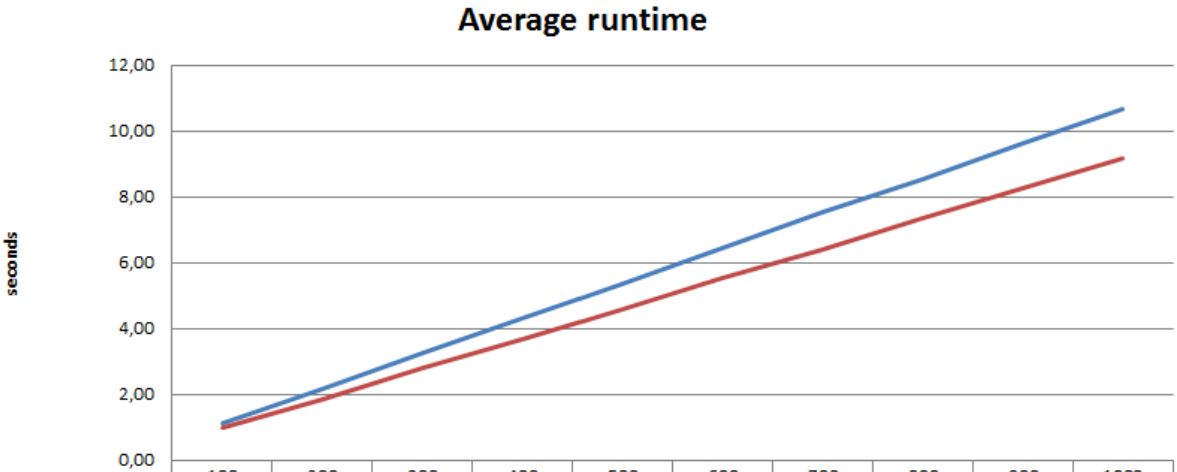
The naive approach for collision detection proves to by easy to implement in JavaScript. Since almost no memory is allocated in each frame, no garbage collection issues appear. All methods are well defined and work mostly on floats. This results in a highly optimised binary code produced by the compiler, as shown on 4.5.

Rysunek 4.5: Chart of time used in optimised version of JavaScript



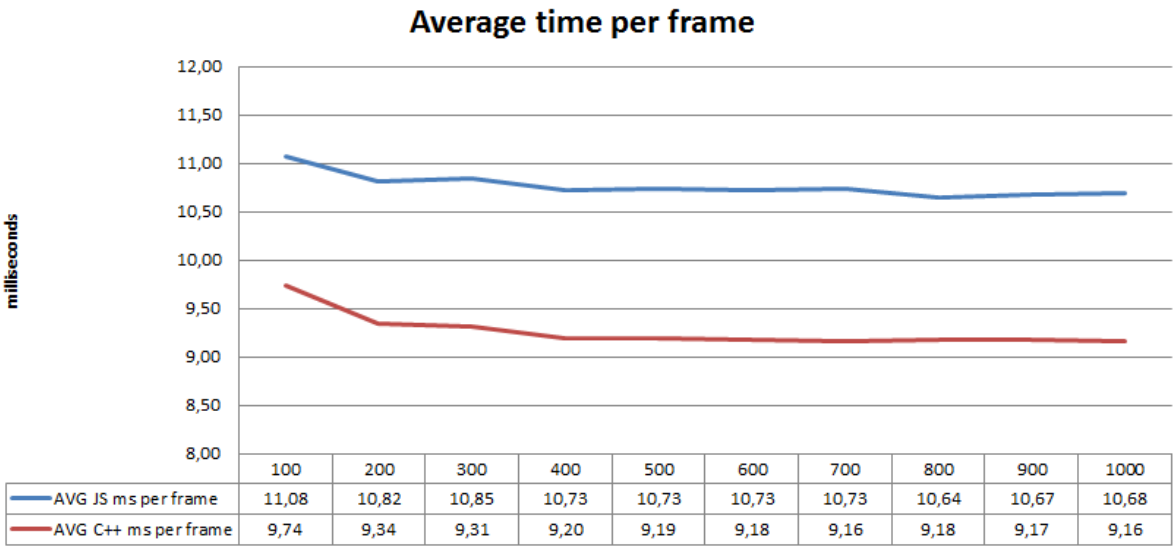
Multiple tests with N=1000 and different number of frames rendered show that for simple mathematical tasks, the performance of JavaScript is very close to that of C++. On average, the JavaScript version of benchmark runs 15% longer than a C++ one.

Rysunek 4.6: 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

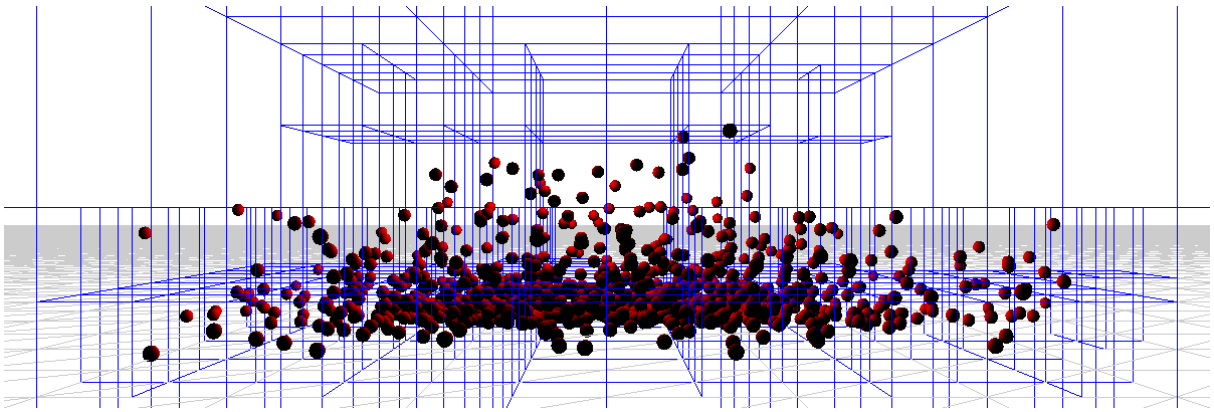
Rysunek 4.7: Comparison of execution time per frame. N = 1000, varying number of frames.



	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

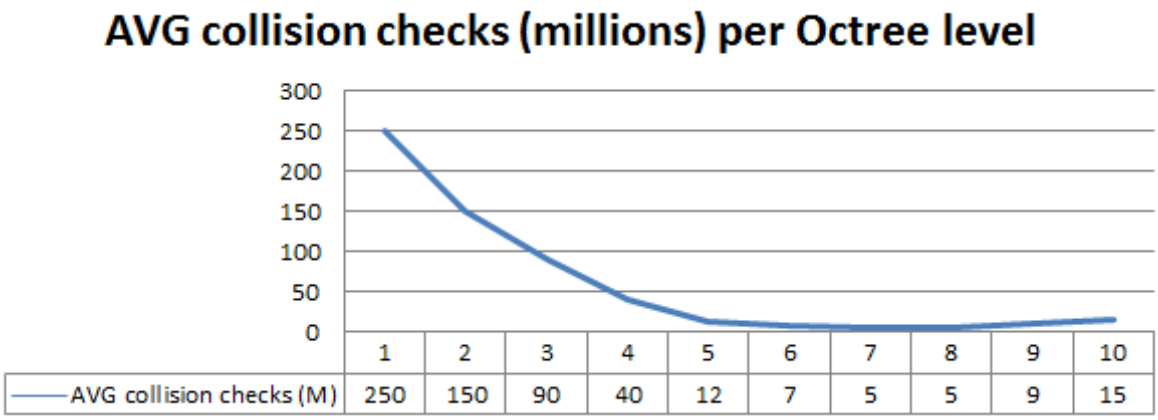
4.3. Octree-partitioned space

Rysunek 4.8: Octree partitioned sphere collision system



Tests with Octree partitioning were executed with N=1000 spheres and T=1000 frames. The varying value is the maximum depth of Octree, ranging from 1 to 10. Changing the maximum depth reduces the number of collision checks between spheres, as shown on 4.9.

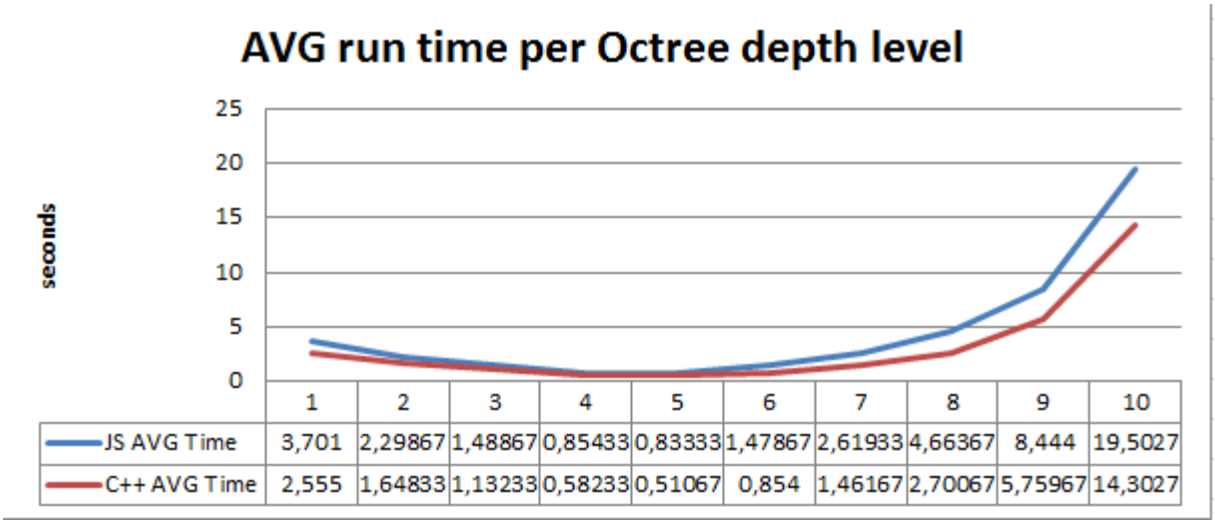
Rysunek 4.9: Number of collisions in Octree



For low values, the overall complexity of checks does not change significantly, since most spheres are in one or a few bounding cubes, and no checks are skipped. Additional operations related to Octree actually make this solution slower than $O(n^2)$ approach. For depth values in the optimal zone, the number of collisions is reduced by a factor of at least 10, while keeping Octree overhead reasonable. Interesting thing happens when the maximum level of Octree is very high and the edge of the smallest Octree cube approaches the size of the spheres. The number of transitions between partitioning cubes, related memory allocation and cleanups actually make this approach much slower, as shown on 4.10. Moreover, some spheres are references in more than one cube, raising again the number of collision checks.

It's clearly visible that the number of collision checks and run time is correlated only up to a certain point. For deep Octrees, the number of checks does not improve further, but the overall run time gets longer. The performance of JavaScript in relation to C++ varies between 30% to 80% overhead. In comparison with $O(n^2)$ approach, optimal Octree in JavaScript runs over 92% faster and C++ over 94% faster.

Rysunek 4.10: Run times in Octree system



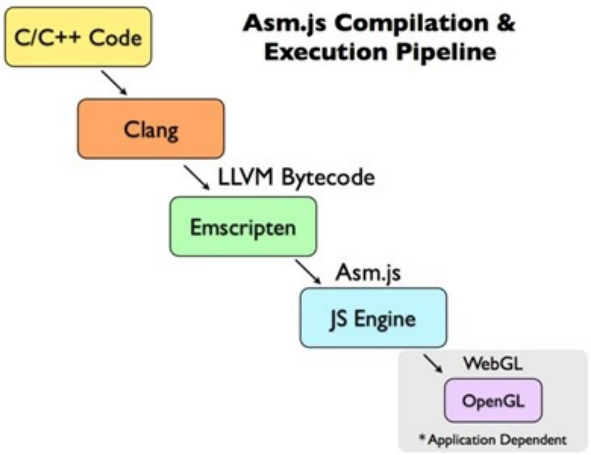
	1	2	3	4	5	6	7	8	9	10
JS AVG Time	3,701	2,29867	1,48867	0,85433	0,83333	1,47867	2,61933	4,66367	8,444	19,5027
C++ AVG Time	2,555	1,64833	1,13233	0,58233	0,51067	0,854	1,46167	2,70067	5,75967	14,3027

5. Emscripten

JavaScript is often called an assembly language of the Web.[20] One could argue that since only one language is supported by browsers it could be made a compilation target similar to an assembler for CPU. This statement is flawed, since eventually JavaScript is translated to assembly making it only an intermediate step. Probably, a resemblance to ByteCode in JVM, which is a compilation target of multiple languages like Java, Scala and Clojure is more in place. Nevertheless, recent years have shown multiple projects aimed at converting code to JavaScript. Some introduce new syntax like CoffeeScript, Dart or TypeScript, while still serving the same purpose – providing human readable code that is interpreted in browser on the fly. Others, which are the focus of this chapter, aim to convert existing projects to run in a browser.

Several new projects are connected to make this happen. The first steps in conversion between languages were made with the LLVM project[21] which currently is a collection of tools and compilers converting code to and from intermediate representation (LLVM IR). For C++ Clang[22] is a conversion tool.

Rysunek 5.1: Pipeline of Emscripten conversion. Source: [20]



The code in LLVM is suitable for further conversion to language like JavaScript. This part is handled by the Emscripten[23] project. Initially, the compilation target for Emscripten was plain JavaScript. With recent developments the asm.js[24] library was created. It provides syntax built on top of JavaScript, which is strongly typed and easily translatable to assembly language. Asm.js details are explained in section 5.1.

```
1 do {
```

```
2     if (a[k + 22 | 0] << 24 >> 24 == 30) {
3       h = b[k + 14 >> 1] | 0;
4       if ((h - 1 & 65535) > 1) {
5         break
6       }
7       l = c[j >> 2] | 0;
8       p = (c[1384465] | 0) + 3 | 0;
9       if (p >>> 0 < 26) {
10        s = (2293760 >>> (p >>> 0) & 1 | 0) != 0 ? 0 : -1e3
11      } else {
12        s = -1e3
13      } if (!(Vq(d, l, k | 0, h << 16 >> 16, s) | 0)) {
14        break
15      }
16      g[(c[f >> 2] | 0) + (l * 112 & -1) + 56 >> 2] = +(b[k + 12 >> 1] << 16 >> 16 | 0);
17      h = (c[f >> 2] | 0) + (l * 112 & -1) + 60 | 0;
18      l = k + 28 | 0;
19      c[h >> 2] = c[l >> 2] | 0;
20      c[h + 4 >> 2] = c[l + 4 >> 2] | 0;
21      c[h + 8 >> 2] = c[l + 8 >> 2] | 0;
22      c[h + 12 >> 2] = c[l + 12 >> 2] | 0;
23      c[h + 16 >> 2] = c[l + 16 >> 2] | 0;
24      c[h + 20 >> 2] = c[l + 20 >> 2] | 0;
25      c[h + 24 >> 2] = c[l + 24 >> 2] | 0
26    }
27  } while (0);
```

Listing 5.1: Example of code using asm.js

The project, built in cooperation with Mozilla Foundation, has its own engine for Firefox – OdinMonkey, designed to run faster for this limited and well-defined syntax.

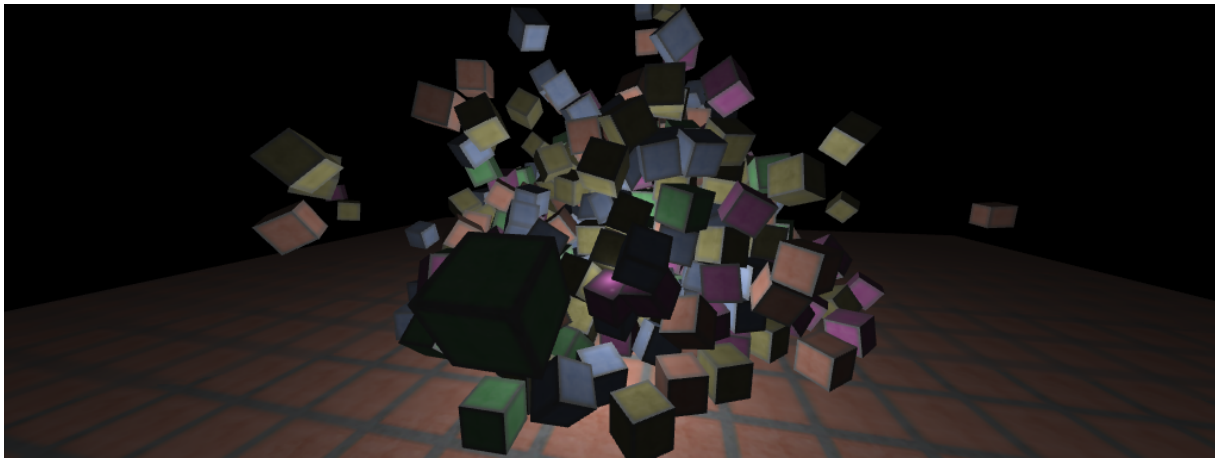
Altogether these projects resulted in multiple libraries and games converted from the native version to JavaScript.

Proof-of-concept demo, made in cooperation between Mozilla and Unreal, is Epic Citadel HTML5 – Unreal Engine 3 technology demo[25] instance running in the browser.[26] The companies claim it took only four days to complete the conversion.

Another example of a successfully converted project is ammo.js[27] – originating from the Bullet physics engine.

Rysunek 5.2: Epic Citadel screenshot

Rysunek 5.3: Ammo.js demo colliding 500 boxes at 30fps, available at http://kripken.github.io/ammo.js/examples/new/ammo.html



5.1. Asm.js overview

Asm.js introduces some improvements targeted to fix performance problems of JavaScript. Ahead of time compilation enforces very specific rules on the coding style. Documentation[24] lists:

5.1.1. Unboxed representations of integers and floating-point numbers

In asm.js the only types allowed are integers and doubles. All numbers have annotations indicating static type (see: 2.2). This way, the compiler does not have to detect possible transitions between variable types, and the overall code runs faster.

5.1.2. Absence of runtime type checks

Since asm.js works only on well-defined numbers, all function calls are monomorphic and stable. In OdinMonkey ahead-of-time compilation is able to compile them to the most optimised version without tracking method calls. In engines using JIT, methods are compiled early and newer deoptimised.

5.1.3. Absence of garbage collection

As shown in previous chapters, garbage collection calls are often a performance bottleneck. Asm.js solves this problem by eliminating garbage collection completely. Memory is stored in short-life variables, deallocated after the method exits and in global heaps, which are never resized or deallocated.

5.1.4. Efficient heap loads and stores

Heaps are global arrays of statically typed arrays, in JavaScript implemented as objects listed in 5.1. The size of the heap does not change during runtime – it has to be calculated during the compilation and incorrect prediction or memory leaks may result in buffer overflow. Heaps are always passed as an argument to asm.js modules. Each module may reference part of the heap and use it as long as necessary.

Tablica 5.1: Statically typed arrays in JavaScript

View Type	Element Size (Bytes)	Element Type
Uint8Array	1	intish
Int8Array	1	intish
Uint16Array	2	intish
Int16Array	2	intish
Uint32Array	4	intish
Int32Array	4	intish
Float32Array	4	doublish
Float64Array	8	doublish

5.1.5. Summary

These solutions remove some of the performance bottlenecks described in previous chapters. The code suitable to run with asm.js is almost unreadable by the programmer, and resembles an assembly code. Asm.js is not designed to be a language used by a programmer, it is mainly a target for compilation using converters like Emscripten.

Most optimisation used by asm.js is consistent with the code that is expected by V8 engine – variables and methods are monomorphic, garbage collection is close to zero. It is worth noting that a code written by hand is almost always shorter than one generated by Emscripten. A partial solution to this is the built-in Google Closure Compiler, used on output of Emscripten and different levels of optimisation. In tests mentioned in this paper, all solutions take 2 to 4kB compiled, while Emscripten produces over 450kB of code for each. This affects real life performance by lengthening at least transfer and parse time for code. The average parse time of 1kB of JS is believed to be up to 1ms[28]. The global average bandwidth was 3.1 Mbps in Q1 2013[29] thus transfer of each kilobyte is around 2.5ms. In total, the load time of the tested code is approximately 1.5 second on an average bandwidth and platform. In the case of larger applications and games, the load time may be significantly greater and should be taken into consideration.

6. Summary

All four tests were compiled and run times were measured on different platforms. Results obtained from Windows platform are significantly better than on Linux or Mac OS X. Reason for this is probably process of compilation of V8 itself, which by design is a 32-bit program. All tested operating systems are 64-bit. Results on Windows will be used as a baseline, since it is a platform most often used for gaming. It is expected that performance on all platforms will become similar once V8 switches to 64-bit.

The unoptimised version of the particle system (see: 3.2) was designed to perform a lot of memory operations. In the tests it is visible that C++ handles this task best. The code generated using Emscripten benefits from static memory heap, which effectively works similar to the object pool introduced later, and runs approximately twice as long as C++. Plain JavaScript suffers greatly from memory allocation issues and an unoptimised code, resulting in 6 times slower execution than C++.

Optimised particles (see: 3.3) with an object pool and low garbage collection show how improvements of the algorithm result in much faster JavaScript execution. It still takes twice as long to run the particle system in V8, but the Emscripten version takes three times longer than C++. The additional overhead of a long and complex code is clearly visible, and since JS code employs the same techniques that Emscripten uses automatically, there is no improvement in the related areas. It is worth mentioning that the unoptimised version of C++ particle system is slightly slower than an optimised JavaScript one, showing that code quality improvement that does not affect algorithmic complexity of an algorithm may be more important than the choice of the environment.

Tablica 6.1: Particle tests on different platforms

Platform	Unoptimised particles			Optimised particles		
	C++	JavaScript	Emscripten	C++	JavaScript	Emscripten
Fedora 19, Intel i7 2670QM, 4GB RAM, g++ 4.8.1	3.21s	19.51s	4.85s	1.63s	4.96s	5.10s
Windows 7, Intel i7 2670QM, 4GB RAM, g++ 4.7.3, Cygwin	3.51s	20.77s	6.46s	1.71s	3.47s	5.57s
Mac OS X 10.8.5, Intel i7 3720QM, 8GB RAM, XCode 5.0 hline	1.83s	16.91s	4.16s	1.22s	2.35s	4.23s

The sphere collision test is putting a high load on CPU.

In $O(n^2)$ version (see: 4.2) exactly 1 000 000 000 checks for object collisions are made. The overall execution time is surprisingly good for JavaScript with an overhead of approximately 15% and for the Emscripten generated version – 25%. This is a result of focusing on pure mathematical operations that are quickly compiled by V8 and processed on unboxed numbers in asm.js.

Space partitioning using Octree greatly reduces the number of collisions (to less than 100 000) and execution time. To give meaningful results simulation time is increased from 1000 frames to 10000. Under these circumstances difference in run time changes to 20% for Javascript and remains similarly around 25% for Emscripten. This is the result of small memory operations related to Octree areas.

Tablica 6.2: Spheres tests on different platforms

Platform	$O(n^2)$ spheres			Octree spheres		
	C++	JavaScript	Emscripten	C++	JavaScript	Emscripten
Fedora 19, Intel i7 2670QM, 4GB RAM, g++ 4.8.1	4.96s	9.02s	12.35s	3.44s	14.14s	11.20s
Windows 7, Intel i7 2670QM, 4GB RAM, g++ 4.7.3, Cygwin	9.52s	10.81s	11.82s	14.10s	16.95s	17.79s
Mac OS X 10.8.5, Intel i7 3720QM, 8GB RAM, XCode 5.0 hline	5.05s	8.62s	13.88s	6.67s	14.34s	17.23s

The above results show how well-written JavaScript or properly converted C++ are able to perform in a browser with similar performance to native programs, rarely exceeding 100% overhead and sometimes getting as close as 15% to C++ execution time.

Conducted experiments show a gap between JavaScript and C++ performance. Significant language design differences result in code that is often easier to write, but also easier to abuse. Benchmarks show differences between 15% and 100% overhead for correctly designed JavaScript code and over 500% for incorrect patterns. Considering Moore's law, stating that computers double speed every 18 months, it safe to say that JavaScript is very close to being suitable for any type of development.

Tests show a clear pattern regarding dynamic variable types in JavaScript. Whenever boxing and unboxing happens, JIT compilation is not able to properly optimise code and bring it up to the performance of C++. This affects both simple variables and properties and is especially visible for numbers. Transitions between integer and floats are expensive, while easy to overlook.

Types also significantly affect the cost of method calls. Keeping methods monomorphic in core parts of the physics engine is very important. An additional cost of polymorphism of parameters is not only boxing and unboxing of parameters, but also time spent on optimising and deoptimising the compiled method, which makes initial warmup of the engine longer. Exporting well defined methods to be called from polymorphic ones is an easy workaround for this performance bottleneck.

Lastly, memory management proven to be one of the most important problems in JavaScript. Automated garbage collection connected with the popular pattern of creating and returning arrays is

an important problem. Memory allocation of objects is also a bottleneck, but not very dissimilar to the one in C++. As shown in the second version of the particle system, the usage of object pools and changing architecture to avoid array creation are techniques that can be employed to fight it. It is worth mentioning that while garbage collection always introduces some overhead, it is reasonable to avoid it at all costs. The sphere collision system with octree partitioning is also introducing and destroying objects, but overhead is significantly smaller than gained speedup. Advice for memory operations is to avoid objects living only for a single frame i.e. temporary variables and helpers. Long-living objects are in general unavoidable and should be used whenever suitable.

General advice for programming in JavaScript is to use techniques similar to those found in asm.js – keep types static, method calls monomorphic and work carefully with memory.

Conducted tests show that while a gap between JavaScript and the native application exists and is not insignificant, there is a lot of potential in such an approach. It is expected, that with the growing community and interest from the game industry, new games will be released on browser within a few years. Performance issues may prevent works on AAA titles, but companies focused more on the social aspect of games and new trends in monetisation may create games targeted for different users. With capabilities of browsers equal to having an 18-month older machine, less graphically demanding titles like The Sims or World or Warcraft may certainly be ported to run in JavaScript.

A.Acknowledgements

B.Source code

B.1. Math utilities

```
1  /**
2   * @fileoverview Math utils.
3   * @author sebastian.poreba@gmail.com (Sebastian Poreba)
4   */
5
6  goog.provide('smash.math');
7
8
9  /**
10  * @param {number} x
11  * @return {number}
12  */
13 smash.math.square = function(x) {
14   return x * x;
15 };
16
17
18 /**
19  * @param {number} x1
20  * @param {number} y1
21  * @param {number} z1
22  * @param {number} x2
23  * @param {number} y2
24  * @param {number} z2
25  * @return {number}
26  */
27 smash.math.vectorDistance = function(x1, y1, z1, x2, y2, z2) {
28   return Math.sqrt(smash.math.square(x1 - x2) +
29       smash.math.square(y1 - y2) +
30       smash.math.square(z1 - z2));
31 };
32
33
34 /**
35  * @param {number} x
36  * @param {number} y
37  * @param {number} z
38  * @return {number}
39  */
```

```
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;
};


/**
 * @param {number} x1
 * @param {number} y1
 * @param {number} z1
 * @param {number} x2
 * @param {number} y2
 * @param {number} z2
 * @return {number}
 */
smash.math.dot = function(x1, y1, z1, x2, y2, z2) {
  return x1 * x2 + y1 * y2 + z1 * z2;
};
```

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}
```

```
21  */
22 float smash::math::vectorDistance(float x1, float y1, float z1, float x2,
23     float y2, float z2) {
24   return sqrt(smash::math::square(x1 - x2) +
25       smash::math::square(y1 - y2) +
26       smash::math::square(z1 - z2));
27 };
28
29
30 /**
31  * @param {number} x
32  * @param {number} y
33  * @param {number} z
34  * @return {number}
35  */
36 float smash::math::vectorLength(float x, float y, float z) {
37   return sqrt(smash::math::square(x) +
38       smash::math::square(y) +
39       smash::math::square(z));
40 };
41
42 /**
43  * @param {number} x
44  * @param {number} y
45  * @param {number} z
46  * @return {number}
47  */
48 float smash::math::dot(float x1, float y1, float z1, float x2, float y2,
49     float z2) {
50   return x1 * x2 + y1 * y2 + z1 * z2;
51 };
52
53
54 /**
55  * @param {!smash.Sphere} sphere1
56  * @param {!smash.Sphere} sphere2
57  * @return {boolean}
58  */
59 bool smash::math::checkCollidingSpheres(smash::Sphere* sphere1,
60     smash::Sphere* sphere2) {
61   return smash::math::vectorDistance(
62       sphere1->positionX, sphere1->positionY, sphere1->positionZ,
63       sphere2->positionX, sphere2->positionY, sphere2->positionZ) <
64       sphere1->radius + sphere2->radius;
65 };
```

Listing B.2: Math utilities in C++

B.2. Particle system

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

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;
};


```

```
53 /**
54  *
55  */
56 smash.Particle.prototype.step = function() {
57   this.positionX += this.velocityX;
58   this.positionY += this.velocityY;
59   this.age++;
60 };
61
62
63 /**
64  * Recover defaults.
65  */
66 smash.Particle.prototype.reset = function() {
67   this.positionX = 0;
68   this.positionY = 0;
69   this.velocityX = 0;
70   this.velocityY = 0;
71   this.age = 0;
72   this.lifespan = 0;
73   this.isDead = false;
74 };
```

Listing B.3: Particle object in JavaScript

```
1  /**
2   * @fileoverview Particle object.
3   * @author sebastian.poreba@gmail.com (Sebastian Poreba)
4   */
5  #include "particle.h"
6
7  smash::Particle::Particle() {
8    positionX = 0;
9    positionY = 0;
10   velocityX = 0;
11   velocityY = 0;
12   age = 0;
13   lifespan = 0;
14   isDead = false;
15 }
16
17 /**
18  * @param deltaTime
19  */
20 void smash::Particle::step() {
21   this->positionX += this->velocityX;
22   this->positionY += this->velocityY;
23   this->age++;
24 };
25
26
27 /**
28  * Recover defaults.
29  */
30 void smash::Particle::reset() {
```

```
31    this->positionX = 0;
32    this->positionY = 0;
33    this->velocityX = 0;
34    this->velocityY = 0;
35    this->age = 0;
36    this->lifespan = 0;
37    this->isDead = false;
38  };
```

Listing B.4: Particle object in C++

```
1   /**
2    * @fileoverview Particle emitter.
3    * @author sebastian.poreba@gmail.com (Sebastian Poreba)
4    */
5
6   goog.provide('smash.ParticleEmitter');
7
8   goog.require('smash.Particle');
9
10
11
12  /**
13   * @constructor
14   */
15  smash.ParticleEmitter = function() {
16    /**
17     * @type {number}
18     */
19    this.positionX = 0.1;
20
21    /**
22     * @type {number}
23     */
24    this.positionY = 0.1;
25
26    /**
27     * @type {number}
28     */
29    this.angle = 0.1;
30
31    /**
32     * @type {number}
33     */
34    this.velocity = 10.1;
35
36    /**
37     * @type {number}
38     */
39    this.velocitySpread = 0.2;
40
41    /**
42     * @type {number}
43     */
44    this.spread = Math.PI * 10 / 180;
```

```
45
46    /**
47     * In ticks.
48     * @type {number}
49     */
50    this.lifespan = 50;
51
52    /**
53     * @type {number}
54     */
55    this.productionRate = 10;
56  };
57
58
59  /**
60   * @param {number} angle
61   */
62  smash.ParticleEmitter.prototype.setAngle = function(angle) {
63    this.angle = Math.PI * angle / 180;
64  };
65
66
67  /**
68   * @param {number} velocity
69   */
70  smash.ParticleEmitter.prototype.setVelocity = function(velocity) {
71    this.velocity = velocity;
72  };
73
74
75  /**
76   * @param {number} velocitySpread
77   */
78  smash.ParticleEmitter.prototype.setVelocitySpread = function(velocitySpread) {
79    this.velocitySpread = velocitySpread;
80  };
81
82
83  /**
84   * @param {number} spread
85   */
86  smash.ParticleEmitter.prototype.setSpread = function(spread) {
87    this.spread = Math.PI * spread / 180;
88  };
89
90
91  /**
92   * @param {number} lifespan
93   */
94  smash.ParticleEmitter.prototype.setLifespan = function(lifespan) {
95    this.lifespan = lifespan;
96  };
97
98
99  /**
100  * @param {number} rate
101  */
```

```
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;
};


```

```
27 /**
28  * @param velocity
29  */
30 void smash::ParticleEmitter::setVelocity(float velocity) {
31   this->velocity = velocity;
32 };
33
34
35 /**
36  * @param velocitySpread
37  */
38 void smash::ParticleEmitter::setVelocitySpread(float velocitySpread) {
39   this->velocitySpread = velocitySpread;
40 };
41
42
43 /**
44  * @param spread
45  */
46 void smash::ParticleEmitter::setSpread(float spread) {
47   this->spread = M_PI * spread / 180;
48 };
49
50
51 /**
52  * @param lifespan
53  */
54 void smash::ParticleEmitter::setLifespan(float lifespan) {
55   this->lifespan = lifespan;
56 };
57
58
59 /**
60  * @param rate
61  */
62 void smash::ParticleEmitter::setProductionRate(int rate) {
63   this->productionRate = rate;
64 };
65
66
67 /**
68  * @return {!Array.<!smash::Particle>}
69  */
70 std::vector<smash::Particle*>* smash::ParticleEmitter::getNewParticles() {
71   std::vector<smash::Particle*> *newParticles =
72       new std::vector<smash::Particle*>;
73   for (int i = 0; i < this->productionRate; i++) {
74     smash::Particle* p = new smash::Particle();
75     p->lifespan = this->lifespan;
76     p->positionX = this->positionX;
77     p->positionY = this->positionY;
78     p->velocityX = sin(this->angle +
79         (((float) rand() / (RAND_MAX)) - 0.5) * this->spread) *
80         this->velocity *
81         (1 + (((float) rand() / (RAND_MAX)) - 0.5) * this->velocitySpread);
82     p->velocityY = cos(this->angle +
83         (((float) rand() / (RAND_MAX)) - 0.5) * this->spread) *
```

```
84          this->velocity *
85          (1 + (((float) rand() / (RAND_MAX)) - 0.5) * this->velocitySpread);
86      newParticles->push_back(p);
87    }
88    return newParticles;
89  };
```

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

```
1   /**
2    * @fileoverview Particle system.
3    * @author sebastian.poreba@gmail.com (Sebastian Poreba)
4    */
5
6   goog.provide('smash.ParticleSystem');
7
8
9   goog.require('smash.flags');
10
11
12
13  /**
14   * @constructor
15   */
16  smash.ParticleSystem = function() {
17    /**
18     * @type {!Array.<!smash.Particle>}
19     */
20    this.particles = [];
21
22    /**
23     * @type {!Array.<!smash.ParticleEmitter>}
24     */
25    this.emitters = [];
26
27    if (smash.flags.DRAWING_ENABLED) {
28      /**
29       * @type {!Element}
30       */
31      this.canvas = window.document.createElement('canvas');
32      this.canvas.width = smash.ParticleSystem.CANVAS_WIDTH;
33      this.canvas.height = smash.ParticleSystem.CANVAS_HEIGHT;
34      window.document.body.appendChild(this.canvas);
35
36      /**
37       * @type {!CanvasRenderingContext2D}
38       */
39      this.context = this.canvas.getContext('2d');
40
41      /**
42       * @type {!ImageData}
43       */
44      this.imageData = this.context.getImageData(0, 0,
45          smash.ParticleSystem.CANVAS_WIDTH, smash.ParticleSystem.CANVAS_HEIGHT);
46
```

```
47      /**
48       * @type {!CanvasPixelArray}
49       */
50      this.pixels = this.imageData.data;
51    }
52  };
53
54
55  /**
56   * @const {number}
57   */
58  smash.ParticleSystem.CANVAS_WIDTH = 1200;
59
60
61  /**
62   * @const {number}
63   */
64  smash.ParticleSystem.CANVAS_HEIGHT = 400;
65
66
67  /**
68   *
69   */
70  smash.ParticleSystem.prototype.step = function() {
71    if (smash.flags.DRAWING_ENABLED) {
72      for (var i = 0; i < smash.ParticleSystem.CANVAS_WIDTH *
73          smash.ParticleSystem.CANVAS_HEIGHT * 4; i += 4) {
74        this.pixels[i] = 0;
75        this.pixels[i + 1] = 0;
76        this.pixels[i + 2] = 0;
77        this.pixels[i + 3] = 0;
78      }
79    }
80
81    this.emitters.forEach(function(emitter) {
82      this.particles.push.apply(this.particles,
83          emitter.getNewParticles());
84    }, this);
85
86    var newParticles = [];
87    this.particles.forEach(function(p) {
88      p.step();
89      if (p.positionX >= 0 &&
90          p.positionX < smash.ParticleSystem.CANVAS_WIDTH &&
91          p.positionY >= 0 &&
92          p.positionY < smash.ParticleSystem.CANVAS_HEIGHT &&
93          p.age < p.lifespan) {
94        newParticles.push(p);
95      }
96
97      if (smash.flags.DRAWING_ENABLED) {
98        var baseIndex =
99            (Math.round(p.positionY) *
100               smash.ParticleSystem.CANVAS_WIDTH +
101               Math.round(p.positionX)) * 4;
102       this.pixels[baseIndex] = 255;
103       this.pixels[baseIndex + 1] = 0;
```

```
104       this.pixels[baseIndex + 2] = 0;
105       this.pixels[baseIndex + 3] = 255;
106     }
107   }, this);
108   if (smash.flags.DRAWING_ENABLED) {
109     this.context.putImageData(this.imageData, 0, 0);
110   }
111
112   this.particles = newParticles;
113 };
114
115
116 /**
117  * @param {!smash.ParticleEmitter} emitter
118  */
119 smash.ParticleSystem.prototype.addEmitter = function(emitter) {
120   this.emitters.push(emitter);
121 };
```

Listing B.7: Initial particle system object in JavaScript

```
1  /**
2   * @fileoverview Particle system.
3   * @author sebastian.poreba@gmail.com (Sebastian Poreba)
4   */
5  #include "particleSystem.h"
6
7  smash::ParticleSystem::ParticleSystem() {
8    this->particles = new std::vector<smash::Particle*>;
9    this->emitters = new std::vector<smash::ParticleEmitter*>;
10 };
11
12 smash::ParticleSystem::~ParticleSystem() {
13   this->particles->erase(this->particles->begin(), this->particles->end());
14   delete this->particles;
15   this->emitters->erase(this->emitters->begin(), this->emitters->end());
16   delete this->emitters;
17 };
18
19 void smash::ParticleSystem::step() {
20   for (std::vector<smash::ParticleEmitter*>::iterator it =
21       this->emitters->begin(); it != this->emitters->end(); it++) {
22     std::vector<smash::Particle*>* particleFromEmitters =
23       (*it)->getNewParticles();
24     this->particles->insert(this->particles->end(),
25       particleFromEmitters->begin(), particleFromEmitters->end());
26     delete particleFromEmitters;
27   }
28
29   std::vector<smash::Particle*> newParticles;
30
31   for (std::vector<smash::Particle*>::iterator it =
32       this->particles->begin(); it != this->particles->end(); it++) {
33     smash::Particle* p = *it;
34     p->step();
```

```
35      if (p->positionX >= 0 &&
36          p->positionX < smash::ParticleSystem::CANVAS_WIDTH &&
37          p->positionY >= 0 &&
38          p->positionY < smash::ParticleSystem::CANVAS_HEIGHT &&
39          p->age < p->lifespan) {
40        newParticles.push_back(p);
41      } else {
42        delete p;
43      }
44    };
45    this->particles->swap(newParticles);
46    newParticles.clear();
47  };
48
49  /**
50   * @param emitter
51   */
52  void smash::ParticleSystem::addEmitter(smash::ParticleEmitter* emitter) {
53    this->emitters->push_back(emitter);
54  };
```

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

```
1   /**
2    * @fileoverview Particle system.
3    * @author sebastian.poreba@gmail.com (Sebastian Poreba)
4    */
5
6   goog.provide('smash.ParticleSystem2');
7
8   goog.require('smash.Particle');
9   goog.require('smash.flags');
10
11
12
13  /**
14   * @constructor
15   */
16  smash.ParticleSystem2 = function() {
17    /**
18     * @type {!Array.<smash.Particle>}
19     */
20    this.particles = [];
21
22    /**
23     * @type {!Array.<number>}
24     */
25    this.deadParticles = [];
26
27    /**
28     * @type {!Array.<smash.ParticleEmitter>}
29     */
30    this.emitters = [];
31
32    if (smash.flags.DRAWING_ENABLED) {
```

```
33      /**
34       * @type {!Element}
35       */
36      this.canvas = window.document.createElement('canvas');
37      this.canvas.width = smash.ParticleSystem2.CANVAS_WIDTH;
38      this.canvas.height = smash.ParticleSystem2.CANVAS_HEIGHT;
39      window.document.body.appendChild(this.canvas);
40
41      /**
42       * @type {!CanvasRenderingContext2D}
43       */
44      this.context = this.canvas.getContext('2d');
45
46      /**
47       * @type {!ImageData}
48       */
49      this.imageData = this.context.getImageData(0, 0,
50          smash.ParticleSystem2.CANVAS_WIDTH,
51          smash.ParticleSystem2.CANVAS_HEIGHT);
52
53      /**
54       * @type {!CanvasPixelArray}
55       */
56      this.pixels = this.imageData.data;
57    }
58  };
59
60
61  /**
62   * @const {number}
63   */
64  smash.ParticleSystem2.CANVAS_WIDTH = 1200;
65
66
67  /**
68   * @const {number}
69   */
70  smash.ParticleSystem2.CANVAS_HEIGHT = 400;
71
72
73  /**
74   *
75   */
76  smash.ParticleSystem2.prototype.step = function() {
77    if (smash.flags.DRAWING_ENABLED) {
78      for (var i = 0; i < smash.ParticleSystem2.CANVAS_WIDTH *
79          smash.ParticleSystem2.CANVAS_HEIGHT * 4; i += 4) {
80        this.pixels[i] = 0;
81        this.pixels[i + 1] = 0;
82        this.pixels[i + 2] = 0;
83        this.pixels[i + 3] = 0;
84      }
85    }
86
87    for (var ei = 0; ei < this.emitters.length; ei++) {
88      var emitter = this.emitters[ei];
89      for (var i = 0; i < emitter.productionRate; i++) {
```

```
90        var pIndex = this.deadParticles.pop();
91        if (pIndex !== undefined) {
92          var p = this.particles[pIndex];
93          p.reset();
94        } else {
95          var p = new smash.Particle();
96          this.particles.push(p);
97        }
98
99        p.lifespan = emitter.lifespan;
100       p.positionX = emitter.positionX;
101       p.positionY = emitter.positionY;
102       p.velocityX = Math.sin(emitter.angle +
103           (Math.random() - 0.5) * emitter.spread) *
104           emitter.velocity * emitter.velocitySpread;
105       p.velocityY = Math.cos(emitter.angle +
106           (Math.random() - 0.5) * emitter.spread) *
107           emitter.velocity *
108           (1 + (Math.random() - 0.5) * emitter.velocitySpread);
109     }
110   }
111
112   for (var i = 0; i < this.particles.length; i++) {
113     var p = this.particles[i];
114     p.step();
115     if (p.positionX < 0 ||
116         p.positionX >= smash.ParticleSystem2.CANVAS_WIDTH ||
117         p.positionY < 0 ||
118         p.positionY >= smash.ParticleSystem2.CANVAS_HEIGHT ||
119         p.age > p.lifespan) {
120       this.deadParticles.push(i);
121       p.isDead = true;
122     }
123
124     if (smash.flags.DRAWING_ENABLED && !p.isDead) {
125       var baseIndex =
126           (Math.round(p.positionY) *
127               smash.ParticleSystem2.CANVAS_WIDTH +
128               Math.round(p.positionX)) * 4;
129       this.pixels[baseIndex] = Math.round(p.velocityX * 80);
130       this.pixels[baseIndex + 1] = Math.round(p.velocityX * 80);
131       this.pixels[baseIndex + 2] = 255 - Math.round(p.age / p.lifespan * 255);
132       this.pixels[baseIndex + 3] = 255;
133     }
134   }
135
136   if (smash.flags.DRAWING_ENABLED) {
137     this.context.putImageData(this.imageData, 0, 0);
138   }
139 };
140
141
142 /**
143  * @param {!smash.ParticleEmitter} emitter
144  */
145 smash.ParticleSystem2.prototype.addEmitter = function(emitter) {
146   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->emitters->end(); it++) {
    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);
    }
```

```
52    }
53
54    for (std::vector<smash::Particle*>::iterator it = this->particles->begin();
55        it != this->particles->end(); it++) {
56      smash::Particle* p = *it;
57      if (p->isDead) {
58        continue;
59      }
60      p->step();
61      if (p->positionX < 0 ||
62          p->positionX >= smash::ParticleSystem2::CANVAS_WIDTH ||
63          p->positionY < 0 ||
64          p->positionY >= smash::ParticleSystem2::CANVAS_HEIGHT ||
65          p->age > p->lifespan) {
66        this->deadParticles->push(p);
67        p->isDead = true;
68      }
69    };
70  };
71
72  /**
73   * @param emitter
74   */
75  void smash::ParticleSystem2::addEmitter(smash::ParticleEmitter* emitter) {
76    this->emitters->push_back(emitter);
77  };
```

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

B.3. Spheres collision detection

```
1  /**
2   * @fileoverview Sphere file.
3   * @author sebastian.poreba@gmail.com (Sebastian Poreba)
4   */
5
6
7  goog.provide('smash.Sphere');
8
9
10
11 /**
12  * @struct
13  * @constructor
14  */
15 smash.Sphere = function() {
16   /**
17    * @type {number}
18    */
19   this.positionX = 0.1;
20
21   /**
22    * @type {number}
23    */
24   this.positionY = 0.1;
25
26   /**
27    * @type {number}
28    */
29   this.positionZ = 0.1;
30
31   /**
32    * @type {number}
33    */
34   this.velocityX = 0.1;
35
36   /**
37    * @type {number}
38    */
39   this.velocityY = 0.1;
40
41   /**
42    * @type {number}
43    */
44   this.velocityZ = 0.1;
45
46   /**
47    * @type {number}
48    */
49   this.radius = 5.5;
50
51   /**
52    * @type {number}
```

```
53     */
54   this.mass = 3.5;
55 };
56
57
58 /**
59  * @param {number} stepTime
60  */
61 smash.Sphere.prototype.step = function(stepTime) {
62   this.positionX += this.velocityX * stepTime;
63   this.positionY += this.velocityY * stepTime;
64   this.positionZ += this.velocityZ * stepTime;
65 };
```

Listing B.11: Sphere object in JavaScript

```
1  /**
2   * @fileoverview Sphere file.
3   * @author sebastian.poreba@gmail.com (Sebastian Poreba)
4   */
5
6  #include "sphere.h"
7
8  smash::Sphere::Sphere() {
9    positionX = 0;
10   positionY = 0;
11   positionZ = 0;
12   velocityX = 0;
13   velocityY = 0;
14   velocityZ = 0;
15   radius = 5.5;
16   mass = 3.5;
17 };
18
19 /**
20  * @param stepTime
21  */
22 void smash::Sphere::step(float stepTime) {
23   this->positionX += this->velocityX * stepTime;
24   this->positionY += this->velocityY * stepTime;
25   this->positionZ += this->velocityZ * stepTime;
26 };
```

Listing B.12: Sphere object in C++

```
1  /**
2   * @fileoverview Sphere collision detection system.
3   * @author sebastian.poreba@gmail.com (Sebastian Poreba)
4   */
5
6  goog.provide('smash.SphereSystem');
7
8  goog.require('smash.Sphere');
9  goog.require('smash.flags');
```

```
10 goog.require('smash.math');
11
12
13
14 /**
15  * @constructor
16  */
17 smash.SphereSystem = function() {
18   var generalVelocity = 1;
19   /**
20    * @type {!Array.<!smash.Sphere>}
21    */
22   this.spheres = new Array(smash.SphereSystem.SPHERES_COUNT);
23   for (var i = 0; i < smash.SphereSystem.SPHERES_COUNT; i++) {
24     var sphere = new smash.Sphere();
25     sphere.positionX = (Math.random() - 0.5) * 400;
26     sphere.positionY = (Math.random() - 0.5) * 200;
27     sphere.positionZ = (Math.random() - 0.5) * 100;
28     sphere.velocityX = (Math.random() - 0.5) * generalVelocity;
29     sphere.velocityY = (Math.random() - 0.5) * generalVelocity;
30     sphere.velocityZ = (Math.random() - 0.5) * generalVelocity;
31     this.spheres[i] = sphere;
32   }
33
34   if (smash.flags.DRAWING_ENABLED) {
35     /**
36      * @type {!THREE.PerspectiveCamera}
37      */
38     this.camera = new THREE.PerspectiveCamera(20,
39         smash.SphereSystem.CANVAS_WIDTH /
40             smash.SphereSystem.CANVAS_HEIGHT,
41         1, 10000);
42     this.camera.position.z = 1000;
43
44     var controls = new THREE.OrbitControls(this.camera);
45     controls.addEventListener('change', goog.bind(function() {
46       this.renderer.render(this.scene, this.camera);
47     }, this));
48
49     /**
50      * @type {!THREE.Scene}
51      */
52     this.scene = new THREE.Scene();
53
54     var spotLight = new THREE.PointLight(0xffffff);
55     spotLight.position.set(-40, 60, -10);
56     this.scene.add(spotLight);
57
58     var axes = new THREE.AxisHelper(20);
59     this.scene.add(axes);
60
61     var planeGeometry = new THREE.PlaneGeometry(
62         10000, 10000, 100, 100);
63     var planeMaterial = new THREE.MeshBasicMaterial({
64       color: 0xcccccc,
65       wireframe: true
66     });
```

```
67
68      var plane = new THREE.Mesh(planeGeometry, planeMaterial);
69      plane.rotation.x = -0.5 * Math.PI;
70      plane.position.x = 0;
71      plane.position.y = smash.SphereSystem.FLOOR_LEVEL;
72      plane.position.z = 0;
73
74      this.scene.add(plane);
75
76
77
78      var material = new THREE.MeshLambertMaterial({
79        color: 0xff0000
80      });
81      /**
82       * @type {!Array.<!THREE.SphereGeometry>}
83       */
84      this.threeSpheres = new Array(smash.SphereSystem.SPHERES_COUNT);
85      for (var i = 0; i < smash.SphereSystem.SPHERES_COUNT; i++) {
86        var sphere = new THREE.SphereGeometry(this.spheres[i].radius, 10, 10);
87        var mesh = new THREE.Mesh(sphere, material);
88        mesh.position.x = this.spheres[i].positionX;
89        mesh.position.y = this.spheres[i].positionY;
90        mesh.position.z = this.spheres[i].positionZ;
91        this.threeSpheres[i] = mesh;
92
93        this.scene.add(mesh);
94      }
95
96
97
98      /**
99       * @type {!THREE.WebGLRenderer}
100      */
101     this.renderer = new THREE.WebGLRenderer();
102     this.renderer.setSize(smash.SphereSystem.CANVAS_WIDTH,
103         smash.SphereSystem.CANVAS_HEIGHT);
104     document.body.appendChild(this.renderer.domElement);
105     this.renderer.render(this.scene, this.camera);
106   }
107
108   /**
109    * @type {number}
110    */
111   this.collisions = 0;
112 };
113
114
115 /**
116  * @const {number}
117  */
118 smash.SphereSystem.SPHERES_COUNT = 1000;
119
120
121 /**
122  * @const {boolean}
123  */
```

```
124 smash.SphereSystem.GRAVITY_ENABLED = true;
125
126
127 /**
128  * @const {number}
129  */
130 smash.SphereSystem.GRAVITY_FORCE = 0.1;
131
132
133 /**
134  * @const {number}
135  */
136 smash.SphereSystem.FLOOR_LEVEL = -100;
137
138
139 /**
140  * (1 - energy lost on floor hit)
141  * @const {number}
142  */
143 smash.SphereSystem.FLOOR_FRICTON = 0.8;
144
145
146 /**
147  * @const {number}
148  */
149 smash.SphereSystem.CANVAS_WIDTH = 1200;
150
151
152 /**
153  * @const {number}
154  */
155 smash.SphereSystem.CANVAS_HEIGHT = 400;
156
157
158 /**
159  * @param {!smash.Sphere} sphere1
160  * @param {!smash.Sphere} sphere2
161  */
162 smash.SphereSystem.collide = function(sphere1, sphere2) {
163   var distanceX = sphere1.positionX - sphere2.positionX;
164   var distanceY = sphere1.positionY - sphere2.positionY;
165   var distanceZ = sphere1.positionZ - sphere2.positionZ;
166   var distanceLength = smash.math.vectorLength(
167       distanceX, distanceY, distanceZ);
168
169   // normalize
170   distanceX /= distanceLength;
171   distanceY /= distanceLength;
172   distanceZ /= distanceLength;
173
174   var a1 = smash.math.dot(sphere1.velocityX, sphere1.velocityY,
175       sphere1.velocityZ, distanceX, distanceY, distanceZ);
176   var a2 = smash.math.dot(sphere2.velocityX, sphere2.velocityY,
177       sphere2.velocityZ, distanceX, distanceY, distanceZ);
178   var optimizedP = (2.0 * (a1 - a2)) / (sphere1.mass + sphere2.mass);
179
180   sphere1.velocityX -= optimizedP * sphere2.mass * distanceX;
```

```
181   sphere1.velocityY -= optimizedP * sphere2.mass * distanceY;
182   sphere1.velocityZ -= optimizedP * sphere2.mass * distanceZ;
183   sphere2.velocityX += optimizedP * sphere1.mass * distanceX;
184   sphere2.velocityY += optimizedP * sphere1.mass * distanceY;
185   sphere2.velocityZ += optimizedP * sphere1.mass * distanceZ;
186 };
187
188
189 /**
190  * @param {!smash.Sphere} sphere
191  */
192 smash.SphereSystem.prototype.applyGravity = function(sphere) {
193   if (smash.SphereSystem.GRAVITY_ENABLED) {
194     sphere.velocityY -= smash.SphereSystem.GRAVITY_FORCE;
195   }
196 };
197
198
199 /**
200  * @param {!smash.Sphere} sphere
201  */
202 smash.SphereSystem.prototype.applyFloor = function(sphere) {
203   if (sphere.positionY - sphere.radius <
204       smash.SphereSystem.FLOOR_LEVEL) {
205     sphere.velocityY *= -smash.SphereSystem.FLOOR_FRICTON;
206   }
207 };
208
209
210 /**
211  *
212  */
213 smash.SphereSystem.prototype.step = function() {
214   for (var i = 0; i < smash.SphereSystem.SPHERES_COUNT; i++) {
215     this.applyGravity(this.spheres[i]);
216     this.applyFloor(this.spheres[i]);
217
218
219     this.spheres[i].step(1);
220     for (var j = 0; j < smash.SphereSystem.SPHERES_COUNT; j++) {
221       if (i != j &&
222           smash.math.checkCollidingSpheres(this.spheres[i], this.spheres[j])) {
223         this.collisions++;
224         smash.SphereSystem.collide(this.spheres[i], this.spheres[j]);
225       }
226     }
227
228     if (smash.flags.DRAWING_ENABLED) {
229       this.threeSpheres[i].position.x = this.spheres[i].positionX;
230       this.threeSpheres[i].position.y = this.spheres[i].positionY;
231       this.threeSpheres[i].position.z = this.spheres[i].positionZ;
232     }
233   }
234
235   if (smash.flags.DRAWING_ENABLED) {
236     this.renderer.render(this.scene, this.camera);
237   }
```

```
238 };
```

Listing B.13: Sphere system object in JavaScript

```
1  /**
2   * @fileoverview Sphere collision detection system.
3   * @author sebastian.poreba@gmail.com (Sebastian Poreba)
4   */
5
6  #include "sphereSystem.h"
7
8  smash::SphereSystem::SphereSystem() {
9    this->spheres = new std::vector<smash::Sphere*>;
10   collisions = 0;
11
12   float generalVelocity = 1;
13     for (int i = 0; i < smash::SphereSystem::SPHERES_COUNT; i++) {
14     smash::Sphere* sphere = new smash::Sphere();
15     sphere->positionX = (((float) rand() / (RAND_MAX)) - 0.5) * 400;
16     sphere->positionY = (((float) rand() / (RAND_MAX)) - 0.5) * 200;
17     sphere->positionZ = (((float) rand() / (RAND_MAX)) - 0.5) * 100;
18     sphere->velocityX = (((float) rand() / (RAND_MAX)) - 0.5) * generalVelocity;
19     sphere->velocityY = (((float) rand() / (RAND_MAX)) - 0.5) * generalVelocity;
20     sphere->velocityZ = (((float) rand() / (RAND_MAX)) - 0.5) * generalVelocity;
21     this->spheres->push_back(sphere);
22   }
23 };
24
25
26 smash::SphereSystem::~SphereSystem() {
27   this->spheres->erase(this->spheres->begin(), this->spheres->end());
28   delete this->spheres;
29 };
30
31
32 /**
33  * @param {smash::Sphere*} sphere1
34  * @param {smash::Sphere*} sphere2
35  */
36 void smash::SphereSystem::collide(smash::Sphere* sphere1,
37       smash::Sphere* sphere2) {
38   float distanceX = sphere1->positionX - sphere2->positionX;
39   float distanceY = sphere1->positionY - sphere2->positionY;
40   float distanceZ = sphere1->positionZ - sphere2->positionZ;
41   float distanceLength = smash::math::vectorLength(
42       distanceX, distanceY, distanceZ);
43
44   // normalize
45   distanceX /= distanceLength;
46   distanceY /= distanceLength;
47   distanceZ /= distanceLength;
48
49   float a1 = smash::math::dot(sphere1->velocityX, sphere1->velocityY,
50       sphere1->velocityZ, distanceX, distanceY, distanceZ);
51   float a2 = smash::math::dot(sphere2->velocityX, sphere2->velocityY,
```

```
52        sphere2->velocityZ, distanceX, distanceY, distanceZ);
53    float optimizedP = (2.0 * (a1 - a2)) / (sphere1->mass + sphere2->mass);
54
55    sphere1->velocityX -= optimizedP * sphere2->mass * distanceX;
56    sphere1->velocityY -= optimizedP * sphere2->mass * distanceY;
57    sphere1->velocityZ -= optimizedP * sphere2->mass * distanceZ;
58    sphere2->velocityX += optimizedP * sphere1->mass * distanceX;
59    sphere2->velocityY += optimizedP * sphere1->mass * distanceY;
60    sphere2->velocityZ += optimizedP * sphere1->mass * distanceZ;
61  };
62
63  void smash::SphereSystem::applyGravity(smash::Sphere* sphere) {
64    if (smash::SphereSystem::GRAVITY_ENABLED) {
65      sphere->velocityY -= smash::SphereSystem::GRAVITY_FORCE;
66    }
67  };
68
69
70  void smash::SphereSystem::applyFloor(smash::Sphere* sphere) {
71    if (sphere->positionY - sphere->radius <
72        smash::SphereSystem::FLOOR_LEVEL) {
73      sphere->velocityY *= -smash::SphereSystem::FLOOR_FRICTON;
74    }
75  };
76
77
78  void smash::SphereSystem::step() {
79    for (std::vector<smash::Sphere*>::iterator it = this->spheres->begin();
80        it != this->spheres->end(); it++) {
81      smash::Sphere* s = *it;
82      applyGravity(s);
83      applyFloor(s);
84
85
86      s->step(1);
87      for (std::vector<smash::Sphere*>::iterator it2 = this->spheres->begin();
88        it2 != this->spheres->end(); it2++) {
89        smash::Sphere* s2 = *it2;
90        if (s != s2 &&
91            smash::math::checkCollidingSpheres(s, s2)) {
92          this->collisions++;
93          smash::SphereSystem::collide(s, s2);
94        }
95      }
96    }
97  };
```

Listing B.14: Sphere system in C++

```
1  /**
2   * @fileoverview Octree space partitioning.
3   * @author sebastian.poreba@gmail.com (Sebastian Poreba)
4   */
5
6  goog.provide('smash.Octree');
```

```
7
8  goog.require('goog.array');
9
10
11
12 /**
13  * @param {number} left
14  * @param {number} right
15  * @param {number} top
16  * @param {number} bottom
17  * @param {number} near
18  * @param {number} far
19  * @param {number} maxDepth
20  * @constructor
21  */
22 smash.Octree = function(left, right, top, bottom, near, far, maxDepth) {
23
24   /**
25    * @type {number}
26    */
27   this.maxDepth = maxDepth;
28
29   /**
30    * @type {Array.<!smash.Octree>}
31    */
32   this.childNodes = [];
33
34   /**
35    * @type {number}
36    */
37   this.left = left;
38
39   /**
40    * @type {number}
41    */
42   this.right = right;
43
44   /**
45    * @type {number}
46    */
47   this.top = top;
48
49   /**
50    * @type {number}
51    */
52   this.bottom = bottom;
53
54   /**
55    * @type {number}
56    */
57   this.near = near;
58
59   /**
60    * @type {number}
61    */
62   this.far = far;
63
```

```
64    /**
65     * @type {Array.<!smash.Sphere>}
66     */
67    this.objects = [];
68  };
69
70
71  /**
72   *
73   */
74  smash.Octree.prototype.split = function() {
75    var middleX = (this.left + this.right) / 2;
76    var middleY = (this.top + this.bottom) / 2;
77    var middleZ = (this.near + this.far) / 2;
78    var maxDepth = this.maxDepth - 1;
79
80    this.childNodes[0] = new smash.Octree(this.left, middleX,
81        this.top, middleY, this.near, middleZ, maxDepth);
82    this.childNodes[1] = new smash.Octree(middleX, this.right,
83        this.top, middleY, this.near, middleZ, maxDepth);
84
85    this.childNodes[2] = new smash.Octree(this.left, middleX,
86        middleY, this.bottom, this.near, middleZ, maxDepth);
87    this.childNodes[3] = new smash.Octree(middleX, this.right,
88        middleY, this.bottom, this.near, middleZ, maxDepth);
89
90    this.childNodes[4] = new smash.Octree(this.left, middleX,
91        this.top, middleY, middleZ, this.far, maxDepth);
92    this.childNodes[5] = new smash.Octree(middleX, this.right,
93        this.top, middleY, middleZ, this.far, maxDepth);
94
95    this.childNodes[6] = new smash.Octree(this.left, middleX,
96        middleY, this.bottom, middleZ, this.far, maxDepth);
97    this.childNodes[7] = new smash.Octree(middleX, this.right,
98        middleY, this.bottom, middleZ, this.far, maxDepth);
99  };
100
101
102 /**
103  *
104  * @return {boolean}
105  */
106 smash.Octree.prototype.hasAnyObjects = function() {
107   return this.objects.length > 0 ||
108       this.childNodes.some(function(node) {
109         return node.hasAnyObjects();
110       });
111 };
112
113
114 /**
115  *
116  * @param {number} left
117  * @param {number} right
118  * @param {number} top
119  * @param {number} bottom
120  * @param {number} near
```

```
121  * @param {number} far
122  * @return {Array.<number>}
123  */
124 smash.Octree.prototype.getAllOffsets = function(left, right, top, bottom,
125     near, far) {
126   var middleX = (this.left + this.right) / 2;
127   var middleY = (this.top + this.bottom) / 2;
128   var middleZ = (this.near + this.far) / 2;
129
130   var offset = 0;
131   var bothX = false, bothY = false, bothZ = false;
132
133   if (left < middleX && right < middleX) {
134     offset += 0;
135   } else if (left > middleX && right > middleX) {
136     offset += 1;
137   } else {
138     bothX = true;
139   }
140
141   if (top < middleY && bottom < middleY) {
142     offset += 0;
143   } else if (top > middleY && bottom > middleY) {
144     offset += 2;
145   } else {
146     bothY = true;
147   }
148
149   if (near < middleZ && far < middleZ) {
150     offset += 0;
151   } else if (near > middleZ && far > middleZ) {
152     offset += 4;
153   } else {
154     bothZ = true;
155   }
156
157   var allOffsets = [offset];
158   if (bothZ) {
159     for (var i = 0, l = allOffsets.length; i < l; i++) {
160       allOffsets.push(allOffsets[i] + 4);
161     }
162   }
163   if (bothY) {
164     for (var i = 0, l = allOffsets.length; i < l; i++) {
165       allOffsets.push(allOffsets[i] + 2);
166     }
167   }
168   if (bothX) {
169     for (var i = 0, l = allOffsets.length; i < l; i++) {
170       allOffsets.push(allOffsets[i] + 1);
171     }
172   }
173   return allOffsets;
174 };
175
176
177 /**
```

```
178  * @param {!smash.Sphere} sphere
179  * @return {boolean}
180  */
181 smash.Octree.prototype.sphereLeft = function(sphere) {
182   return (sphere.positionX + sphere.radius < this.left ||
183       sphere.positionX - sphere.radius > this.right ||
184       sphere.positionY + sphere.radius < this.top ||
185       sphere.positionY - sphere.radius > this.bottom ||
186       sphere.positionZ + sphere.radius < this.near ||
187       sphere.positionZ - sphere.radius > this.far);
188 };
189
190
191 /**
192  * @param {!smash.Sphere} sphere
193  */
194 smash.Octree.prototype.removeSphere = function(sphere) {
195   goog.array.remove(this.objects, sphere);
196 };
197
198
199 /**
200  * @param {!smash.Sphere} sphere
201  */
202 smash.Octree.prototype.addSphere = function(sphere) {
203   if (this.objects.indexOf(sphere) != -1) {
204     return; // this happens when sphere is re-added from two
205     // different nodes after removal.
206   }
207
208   var left = sphere.positionX - sphere.radius;
209   var right = sphere.positionX + sphere.radius;
210   var top = sphere.positionY - sphere.radius;
211   var bottom = sphere.positionY + sphere.radius;
212   var near = sphere.positionZ - sphere.radius;
213   var far = sphere.positionZ + sphere.radius;
214
215   if (this.maxDepth == 0 || this.objects.length == 0) {
216     this.objects.push(sphere);
217   } else {
218     if (this.childNodes.length == 0) {
219       this.split();
220     }
221     var offsets = this.getAllOffsets(left, right, top, bottom, near, far);
222     offsets.forEach(function(offset) {
223       this.childNodes[offset].addSphere(sphere);
224     }, this);
225   }
226 };
227
228
229 /**
230  *
231  */
232 smash.Octree.prototype.log = function() {
233   window.console.log('On level ', this.maxDepth,
234       ' octree node with', this.objects.length, 'objects');
```

```
235   this.childNodes.forEach(function(node) {
236     node.log();
237   });
238 };
```

Listing B.15: Octree in JavaScript

```
1  /**
2   * @fileoverview Octree space partitioning.
3   * @author sebastian.poreba@gmail.com (Sebastian Poreba)
4   */
5
6  #include "octree.h"
7
8
9  smash::Octree::Octree(float left, float right,
10     float top, float bottom,
11     float near, float far,
12     int maxDepth) {
13   this->maxDepth = maxDepth;
14   this->childNodes = new std::vector<smash::Octree*>;
15   this->left = left;
16   this->right = right;
17   this->top = top;
18   this->bottom = bottom;
19   this->near = near;
20   this->far = far;
21   this->objects = new std::vector<smash::Sphere*>;
22 };
23
24 smash::Octree::~Octree() {
25   delete this->objects;
26   for (std::vector<smash::Octree*>::iterator it = this->childNodes->begin();
27       it != this->childNodes->end(); it++) {
28     delete *it;
29   }
30   delete this->childNodes;
31 }
32
33 /**
34  *
35  */
36 void smash::Octree::split() {
37   float middleX = (this->left + this->right) / 2;
38   float middleY = (this->top + this->bottom) / 2;
39   float middleZ = (this->near + this->far) / 2;
40   int maxDepth = this->maxDepth - 1;
41
42   this->childNodes->push_back(new smash::Octree(this->left, middleX,
43       this->top, middleY, this->near, middleZ, maxDepth));
44   this->childNodes->push_back(new smash::Octree(middleX, this->right,
45       this->top, middleY, this->near, middleZ, maxDepth));
46
47   this->childNodes->push_back(new smash::Octree(this->left, middleX,
48       middleY, this->bottom, this->near, middleZ, maxDepth));
```

```
49    this->childNodes->push_back(new smash::Octree(middleX, this->right,
50        middleY, this->bottom, this->near, middleZ, maxDepth));
51
52    this->childNodes->push_back(new smash::Octree(this->left, middleX,
53        this->top, middleY, middleZ, this->far, maxDepth));
54    this->childNodes->push_back(new smash::Octree(middleX, this->right,
55        this->top, middleY, middleZ, this->far, maxDepth));
56
57    this->childNodes->push_back(new smash::Octree(this->left, middleX,
58        middleY, this->bottom, middleZ, this->far, maxDepth));
59    this->childNodes->push_back(new smash::Octree(middleX, this->right,
60        middleY, this->bottom, middleZ, this->far, maxDepth));
61  };
62
63  bool smash::Octree::hasAnyObjects() {
64    if (this->objects->size() > 0) {
65      return true;
66    }
67
68    for (std::vector<smash::Octree*>::iterator it = this->childNodes->begin();
69        it != this->childNodes->end(); it++) {
70      if ((*it)->hasAnyObjects()) {
71        return true;
72      }
73    }
74    return false;
75  };
76
77
78  int smash::Octree::getTotalObjectCount() {
79    int count = this->objects->size();
80
81    for (std::vector<smash::Octree*>::iterator it = this->childNodes->begin();
82        it != this->childNodes->end(); it++) {
83      count += (*it)->getTotalObjectCount();
84    }
85    return count;
86  };
87
88
89  int smash::Octree::getTotalTreeSize() {
90    int count = this->childNodes->size();
91    for (std::vector<smash::Octree*>::iterator it = this->childNodes->begin();
92        it != this->childNodes->end(); it++) {
93      count += (*it)->getTotalTreeSize();
94    }
95    return count;
96  };
97
98
99
100 std::vector<int>* smash::Octree::getAllOffsets(float left, float right,
101       float top, float bottom, float near, float far) {
102   float middleX = (this->left + this->right) / 2;
103   float middleY = (this->top + this->bottom) / 2;
104   float middleZ = (this->near + this->far) / 2;
105
```

```
106   float offset = 0;
107   bool bothX = false, bothY = false, bothZ = false;
108
109   if (left < middleX && right < middleX) {
110     offset += 0;
111   } else if (left > middleX && right > middleX) {
112     offset += 1;
113   } else {
114     bothX = true;
115   }
116
117   if (top < middleY && bottom < middleY) {
118     offset += 0;
119   } else if (top > middleY && bottom > middleY) {
120     offset += 2;
121   } else {
122     bothY = true;
123   }
124
125   if (near < middleZ && far < middleZ) {
126     offset += 0;
127   } else if (near > middleZ && far > middleZ) {
128     offset += 4;
129   } else {
130     bothZ = true;
131   }
132
133   std::vector<int>* allOffsets = new std::vector<int>;
134   allOffsets->push_back(offset);
135
136   if (bothZ) {
137     for (std::vector<int>::iterator it = allOffsets->begin(),
138         itEnd = allOffsets->end(); it != itEnd; it++) {
139       int n = *it;
140       allOffsets->push_back(n + 4);
141     }
142   }
143   if (bothY) {
144     for (std::vector<int>::iterator it = allOffsets->begin(),
145         itEnd = allOffsets->end(); it != itEnd; it++) {
146       int n = *it;
147       allOffsets->push_back(n + 2);
148     }
149   }
150   if (bothX) {
151     for (std::vector<int>::iterator it = allOffsets->begin(),
152         itEnd = allOffsets->end(); it != itEnd; it++) {
153       int n = *it;
154       allOffsets->push_back(n + 1);
155     }
156   }
157   return allOffsets;
158 };
159
160 bool smash::Octree::sphereLeft(smash::Sphere *sphere) {
161   return (sphere->positionX + sphere->radius < this->left ||
162       sphere->positionX - sphere->radius > this->right ||
```

```
163       sphere->positionY + sphere->radius < this->top ||
164       sphere->positionY - sphere->radius > this->bottom ||
165       sphere->positionZ + sphere->radius < this->near ||
166       sphere->positionZ - sphere->radius > this->far);
167 };
168
169
170 void smash::Octree::removeSphere(smash::Sphere *sphere) {
171   this->objects->erase(std::remove(this->objects->begin(),
172       this->objects->end(), sphere));
173 };
174
175
176 void smash::Octree::addSphere(smash::Sphere *sphere) {
177   if (std::find(this->objects->begin(), this->objects->end(), sphere) !=
178         this->objects->end()) {
179     return; // this happens when sphere is re-added from two
180     // different nodes after removal.
181   }
182
183   float left = sphere->positionX - sphere->radius;
184   float right = sphere->positionX + sphere->radius;
185   float top = sphere->positionY - sphere->radius;
186   float bottom = sphere->positionY + sphere->radius;
187   float near = sphere->positionZ - sphere->radius;
188   float far = sphere->positionZ + sphere->radius;
189
190   if (this->maxDepth == 0 || this->objects->size() == 0) {
191     this->objects->push_back(sphere);
192   } else {
193     if (this->childNodes->size() == 0) {
194       this->split();
195     }
196     std::vector<int> *offsets = this->getAllOffsets(left, right, top,
197       bottom, near, far);
198     for (std::vector<int>::iterator it = offsets->begin();
199       it != offsets->end(); it++) {
200       int offset = *it;
201       if (offset > 8) {
202         // weird offset, investigate
203         continue;
204       }
205       this->childNodes->at(offset)->addSphere(sphere);
206     }
207     offsets->clear();
208     delete offsets;
209   }
210 };
```

Listing B.16: Octree in C++

```
1 /**
2  * @fileoverview Sphere collision detection system.
3  * @author sebastian.poreba@gmail.com (Sebastian Poreba)
4  */
```

```
5
6  goog.provide('smash.SphereSystem2');
7
8  goog.require('smash.Octree');
9  goog.require('smash.Sphere');
10 goog.require('smash.flags');
11 goog.require('smash.math');
12
13
14
15 /**
16  * @constructor
17  */
18 smash.SphereSystem2 = function() {
19   var generalVelocity = 1;
20   /**
21    * @type {!Array.<!smash.Sphere>}
22    */
23   this.spheres = new Array(smash.SphereSystem2.SPHERES_COUNT);
24   for (var i = 0; i < smash.SphereSystem2.SPHERES_COUNT; i++) {
25     var sphere = new smash.Sphere();
26     sphere.positionX = (Math.random() - 0.5) * 400;
27     sphere.positionY = (Math.random() - 0.5) * 200;
28     sphere.positionZ = (Math.random() - 0.5) * 100;
29     sphere.velocityX = (Math.random() - 0.5) * generalVelocity;
30     sphere.velocityY = (Math.random() - 0.5) * generalVelocity;
31     sphere.velocityZ = (Math.random() - 0.5) * generalVelocity;
32     this.spheres[i] = sphere;
33   }
34
35   if (smash.flags.DRAWING_ENABLED) {
36     /**
37      * @type {!THREE.PerspectiveCamera}
38      */
39     this.camera = new THREE.PerspectiveCamera(20,
40         smash.SphereSystem2.CANVAS_WIDTH /
41             smash.SphereSystem2.CANVAS_HEIGHT,
42         1, 10000);
43     this.camera.position.z = 1000;
44
45     var controls = new THREE.OrbitControls(this.camera);
46     controls.addEventListener('change', goog.bind(function() {
47       this.renderer.render(this.scene, this.camera);
48     }, this));
49
50     /**
51      * @type {!THREE.Scene}
52      */
53     this.scene = new THREE.Scene();
54
55     var spotLight = new THREE.PointLight(0xffffff);
56     spotLight.position.set(-40, 60, -10);
57     this.scene.add(spotLight);
58
59     var axes = new THREE.AxisHelper(20);
60     this.scene.add(axes);
61
```

```
62      var planeGeometry = new THREE.PlaneGeometry(
63          10000, 10000, 100, 100);
64      var planeMaterial = new THREE.MeshBasicMaterial({
65        color: 0xcccccc,
66        wireframe: true
67      });
68
69      var plane = new THREE.Mesh(planeGeometry, planeMaterial);
70      plane.rotation.x = -0.5 * Math.PI;
71      plane.position.x = 0;
72      plane.position.y = smash.SphereSystem2.FLOOR_LEVEL;
73      plane.position.z = 0;
74
75      this.scene.add(plane);
76
77
78
79      var material = new THREE.MeshLambertMaterial({
80        color: 0xff0000
81      });
82      /**
83       * @type {!Array.<!THREE.SphereGeometry>}
84       */
85      this.threeSpheres = new Array(smash.SphereSystem2.SPHERES_COUNT);
86      for (var i = 0; i < smash.SphereSystem2.SPHERES_COUNT; i++) {
87        var sphere = new THREE.SphereGeometry(this.spheres[i].radius, 10, 10);
88        var mesh = new THREE.Mesh(sphere, material);
89        mesh.position.x = this.spheres[i].positionX;
90        mesh.position.y = this.spheres[i].positionY;
91        mesh.position.z = this.spheres[i].positionZ;
92        this.threeSpheres[i] = mesh;
93
94        this.scene.add(mesh);
95      }
96
97
98
99      /**
100      * @type {!THREE.WebGLRenderer}
101      */
102     this.renderer = new THREE.WebGLRenderer();
103     this.renderer.setSize(smash.SphereSystem2.CANVAS_WIDTH,
104         smash.SphereSystem2.CANVAS_HEIGHT);
105     document.body.appendChild(this.renderer.domElement);
106     this.renderer.render(this.scene, this.camera);
107   }
108
109   /**
110    * @type {number}
111    */
112   this.collisions = 0;
113
114   /**
115    * @type {number}
116    */
117   this.collisionChecks = 0;
118
```

```
119   /**
120    * @type {smash.Octree}
121    * @private
122    */
123   this.octreeRoot_ = new smash.Octree(-1000, 1000, -1000, 1000,
124       -1000, 1000, smash.SphereSystem2.OCTREE_DEPTH);
125
126   for (var i = 0; i < smash.SphereSystem2.SPHERES_COUNT; i++) {
127     this.octreeRoot_.addSphere(this.spheres[i]);
128   }
129
130   if (smash.SphereSystem2.DRAWING_OCTREE_ENABLED) {
131     /**
132      * @type {Array}
133      * @private
134      */
135     this.octreeCubes_ = [];
136   }
137 };
138
139
140 /**
141  * @const {number}
142  */
143 smash.SphereSystem2.SPHERES_COUNT = 1000;
144
145
146 /**
147  * @const {boolean}
148  */
149 smash.SphereSystem2.DRAWING_OCTREE_ENABLED =
150     smash.flags.DRAWING_ENABLED && true;
151
152
153 /**
154  * @const {number}
155  */
156 smash.SphereSystem2.OCTREE_DEPTH = 5;
157
158
159 /**
160  * @const {boolean}
161  */
162 smash.SphereSystem2.GRAVITY_ENABLED = true;
163
164
165 /**
166  * @const {number}
167  */
168 smash.SphereSystem2.GRAVITY_FORCE = 0.1;
169
170
171 /**
172  * @const {number}
173  */
174 smash.SphereSystem2.FLOOR_LEVEL = -100;
175
```

```
176
177 /**
178  * (1 - energy lost on floor hit)
179  * @const {number}
180  */
181 smash.SphereSystem2.FLOOR_FRICTON = 0.8;
182
183
184 /**
185  * @const {number}
186  */
187 smash.SphereSystem2.CANVAS_WIDTH = 1200;
188
189
190 /**
191  * @const {number}
192  */
193 smash.SphereSystem2.CANVAS_HEIGHT = 400;
194
195
196 /**
197  * @param {!smash.Sphere} sphere1
198  * @param {!smash.Sphere} sphere2
199  */
200 smash.SphereSystem2.collide = function(sphere1, sphere2) {
201   var distanceX = sphere1.positionX - sphere2.positionX;
202   var distanceY = sphere1.positionY - sphere2.positionY;
203   var distanceZ = sphere1.positionZ - sphere2.positionZ;
204   var distanceLength = smash.math.vectorLength(
205       distanceX, distanceY, distanceZ);
206   // normalize
207   distanceX /= distanceLength;
208   distanceY /= distanceLength;
209   distanceZ /= distanceLength;
210
211   var a1 = smash.math.dot(sphere1.velocityX, sphere1.velocityY,
212       sphere1.velocityZ, distanceX, distanceY, distanceZ);
213   var a2 = smash.math.dot(sphere2.velocityX, sphere2.velocityY,
214       sphere2.velocityZ, distanceX, distanceY, distanceZ);
215   var optimizedP = (2.0 * (a1 - a2)) / (sphere1.mass + sphere2.mass);
216
217   sphere1.velocityX -= optimizedP * sphere2.mass * distanceX;
218   sphere1.velocityY -= optimizedP * sphere2.mass * distanceY;
219   sphere1.velocityZ -= optimizedP * sphere2.mass * distanceZ;
220   sphere2.velocityX += optimizedP * sphere1.mass * distanceX;
221   sphere2.velocityY += optimizedP * sphere1.mass * distanceY;
222   sphere2.velocityZ += optimizedP * sphere1.mass * distanceZ;
223 };
224
225
226 /**
227  * @param {!smash.Sphere} sphere
228  */
229 smash.SphereSystem2.prototype.applyGravity = function(sphere) {
230   if (smash.SphereSystem2.GRAVITY_ENABLED) {
231     sphere.velocityY -= smash.SphereSystem2.GRAVITY_FORCE;
232   }
```

```
233 };
234
235
236 /**
237  * @param {!smash.Sphere} sphere
238  */
239 smash.SphereSystem2.prototype.applyFloor = function(sphere) {
240   if (sphere.positionY - sphere.radius <
241       smash.SphereSystem2.FLOOR_LEVEL) {
242     sphere.velocityY *= -smash.SphereSystem2.FLOOR_FRICTON;
243   }
244 };
245
246
247 /**
248  * @param {!smash.Octree} node
249  */
250 smash.SphereSystem2.prototype.addOctreeMesh = function(node) {
251   if (!node.debugMesh) {
252     var width = node.right - node.left;
253     var height = node.bottom - node.top;
254     var depth = node.far - node.near;
255     var geom = new THREE.CubeGeometry(width, height, depth);
256     geom.applyMatrix(
257         new THREE.Matrix4().makeTranslation(
258             node.left + width / 2,
259             node.top + height / 2,
260             node.near + depth / 2
261         )
262     );
263
264     node.debugMesh = new THREE.BoxHelper(new THREE.Mesh(geom));
265     this.octreeCubes_.push(node.debugMesh);
266
267     this.scene.add(node.debugMesh);
268   }
269
270   for (var i = 0; i < node.childNodes.length; i++) {
271     this.addOctreeMesh(node.childNodes[i]);
272   }
273 };
274
275
276 /**
277  * @param {!smash.Octree} node
278  * @private
279  */
280 smash.SphereSystem2.prototype.collideFromOctree_ = function(node) {
281   for (var i = 0; i < node.objects.length; i++) {
282     for (var j = 0; j < node.objects.length; j++) {
283       this.collisionChecks++;
284       if (i != j &&
285           smash.math.checkCollidingSpheres(
286               node.objects[i], node.objects[j])) {
287         this.collisions++;
288         smash.SphereSystem2.collide(
289             node.objects[i], node.objects[j]);
```

```
290       }
291     }
292   }
293   for (var i = 0; i < node.childNodes.length; i++) {
294     this.collideFromOctree_(node.childNodes[i]);
295   }
296 };
297
298
299 /**
300  * @param {!smash.Octree} node
301  * @private
302  */
303 smash.SphereSystem2.prototype.stepOctree_ = function(node) {
304   var removedSpheres = [];
305   for (var i = 0; i < node.objects.length; i++) {
306     var sphere = node.objects[i];
307     if (node.sphereLeft(sphere)) {
308       node.removeSphere(sphere);
309       removedSpheres.push(sphere);
310     }
311   }
312
313   for (var i = 0; i < node.childNodes.length; i++) {
314     this.stepOctree_(node.childNodes[i]);
315   }
316
317   if (node !== this.octreeRoot_) {
318     for (var i = 0; i < removedSpheres.length; i++) {
319       this.octreeRoot_.addSphere(removedSpheres[i]);
320     }
321   }
322
323   if (!node.hasAnyObjects()) {
324     if (smash.SphereSystem2.DRAWING_OCTREE_ENABLED) {
325       for (var i = 0; i < node.childNodes.length; i++) {
326         this.scene.remove(node.childNodes[i].debugMesh);
327       }
328     }
329     node.childNodes.length = 0;
330   }
331 };
332
333
334 /**
335  *
336  */
337 smash.SphereSystem2.prototype.step = function() {
338   this.stepOctree_(this.octreeRoot_);
339   this.collideFromOctree_(this.octreeRoot_);
340
341   for (var i = 0; i < smash.SphereSystem2.SPHERES_COUNT; i++) {
342     this.applyGravity(this.spheres[i]);
343     this.applyFloor(this.spheres[i]);
344
345     this.spheres[i].step(1);
346
```

```
347
348     if (smash.flags.DRAWING_ENABLED) {
349       this.threeSpheres[i].position.x = this.spheres[i].positionX;
350       this.threeSpheres[i].position.y = this.spheres[i].positionY;
351       this.threeSpheres[i].position.z = this.spheres[i].positionZ;
352     }
353   }
354
355   if (smash.SphereSystem2.DRAWING_OCTREE_ENABLED) {
356     this.addOctreeMesh(this.octreeRoot_);
357   }
358
359   if (smash.flags.DRAWING_ENABLED) {
360     this.renderer.render(this.scene, this.camera);
361   }
362 };
```

Listing B.17: Octree-based sphere system object in JavaScript

```
1  /**
2   * @fileoverview Sphere collision detection system.
3   * @author sebastian.poreba@gmail.com (Sebastian Poreba)
4   */
5
6  #include "sphereSystem2.h"
7
8  smash::SphereSystem2::SphereSystem2() {
9    this->spheres = new std::vector<smash::Sphere*>;
10   collisions = 0;
11   collisionChecks = 0;
12
13   octreeRoot = new smash::Octree(-1000, 1000, -1000, 1000,
14       -1000, 1000, smash::SphereSystem2::OCTREE_DEPTH);
15
16   float generalVelocity = 1;
17   for (int i = 0; i < smash::SphereSystem2::SPHERES_COUNT; i++) {
18     smash::Sphere* sphere = new smash::Sphere();
19     sphere->positionX = (((float) rand() / (RAND_MAX)) - 0.5) * 400;
20     sphere->positionY = (((float) rand() / (RAND_MAX)) - 0.5) * 200;
21     sphere->positionZ = (((float) rand() / (RAND_MAX)) - 0.5) * 100;
22     sphere->velocityX = (((float) rand() / (RAND_MAX)) - 0.5) * generalVelocity;
23     sphere->velocityY = (((float) rand() / (RAND_MAX)) - 0.5) * generalVelocity;
24     sphere->velocityZ = (((float) rand() / (RAND_MAX)) - 0.5) * generalVelocity;
25     this->spheres->push_back(sphere);
26     this->octreeRoot->addSphere(sphere);
27   }
28 };
29
30
31 smash::SphereSystem2::~SphereSystem2() {
32   this->spheres->clear();
33   delete this->spheres;
34   delete this->octreeRoot;
35 };
36
```

```
37
38 /**
39  * @param {smash::Sphere*} sphere1
40  * @param {smash::Sphere*} sphere2
41  */
42 void smash::SphereSystem2::collide(smash::Sphere* sphere1,
43       smash::Sphere* sphere2) {
44   float distanceX = sphere1->positionX - sphere2->positionX;
45   float distanceY = sphere1->positionY - sphere2->positionY;
46   float distanceZ = sphere1->positionZ - sphere2->positionZ;
47   float distanceLength = smash::math::vectorLength(
48       distanceX, distanceY, distanceZ);
49
50   // normalize
51   distanceX /= distanceLength;
52   distanceY /= distanceLength;
53   distanceZ /= distanceLength;
54
55   float a1 = smash::math::dot(sphere1->velocityX, sphere1->velocityY,
56       sphere1->velocityZ, distanceX, distanceY, distanceZ);
57   float a2 = smash::math::dot(sphere2->velocityX, sphere2->velocityY,
58       sphere2->velocityZ, distanceX, distanceY, distanceZ);
59   float optimizedP = (2.0 * (a1 - a2)) / (sphere1->mass + sphere2->mass);
60
61   sphere1->velocityX -= optimizedP * sphere2->mass * distanceX;
62   sphere1->velocityY -= optimizedP * sphere2->mass * distanceY;
63   sphere1->velocityZ -= optimizedP * sphere2->mass * distanceZ;
64   sphere2->velocityX += optimizedP * sphere1->mass * distanceX;
65   sphere2->velocityY += optimizedP * sphere1->mass * distanceY;
66   sphere2->velocityZ += optimizedP * sphere1->mass * distanceZ;
67 };
68
69
70 void smash::SphereSystem2::applyGravity(smash::Sphere* sphere) {
71   if (smash::SphereSystem2::GRAVITY_ENABLED) {
72     sphere->velocityY -= smash::SphereSystem2::GRAVITY_FORCE;
73   }
74 };
75
76
77 void smash::SphereSystem2::applyFloor(smash::Sphere* sphere) {
78   if (sphere->positionY - sphere->radius <
79       smash::SphereSystem2::FLOOR_LEVEL) {
80     sphere->velocityY *= -smash::SphereSystem2::FLOOR_FRICTON;
81   }
82 };
83
84
85 void smash::SphereSystem2::collideFromOctree(smash::Octree *node) {
86   for (std::vector<smash::Sphere*>::iterator it = node->objects->begin();
87       it != node->objects->end(); it++) {
88     for (std::vector<smash::Sphere*>::iterator it2 = node->objects->begin();
89       it2 != node->objects->end(); it2++) {
90       this->collisionChecks++;
91       smash::Sphere* s = *it;
92       smash::Sphere* s2 = *it2;
93       if (s != s2 &&
```

```
94            smash::math::checkCollidingSpheres(s, s2)) {
95          this->collisions++;
96          smash::SphereSystem2::collide(s, s2);
97        }
98      }
99    }
100   for (std::vector<smash::Octree*>::iterator it = node->childNodes->begin();
101       it != node->childNodes->end(); it++) {
102     this->collideFromOctree(*it);
103   }
104 };
105
106 void smash::SphereSystem2::stepOctree(smash::Octree *node) {
107   std::vector<smash::Sphere*> removedSpheres;
108
109   for (std::vector<smash::Sphere*>::iterator it = node->objects->begin();
110       it != node->objects->end(); it++) {
111     smash::Sphere* s = *it;
112     if (node->sphereLeft(s)) {
113       removedSpheres.push_back(s);
114       it = node->objects->erase(it);
115       if (s == *it) {
116         break; // last element removal
117       }
118     }
119   }
120
121   for (std::vector<smash::Octree*>::iterator it = node->childNodes->begin();
122       it != node->childNodes->end(); it++) {
123     this->stepOctree(*it);
124   }
125
126   if (node != this->octreeRoot) {
127     for (std::vector<smash::Sphere*>::iterator it = removedSpheres.begin();
128         it != removedSpheres.end(); it++) {
129       this->octreeRoot->addSphere(*it);
130     }
131   }
132
133   if (!node->hasAnyObjects()) {
134     for (std::vector<smash::Octree*>::iterator it = node->childNodes->begin();
135         it != node->childNodes->end(); it++) {
136       delete *it;
137     }
138     node->childNodes->clear();
139   }
140 };
141
142
143 void smash::SphereSystem2::step() {
144   this->stepOctree(this->octreeRoot);
145   this->collideFromOctree(this->octreeRoot);
146   for (std::vector<smash::Sphere*>::iterator it = this->spheres->begin();
147       it != this->spheres->end(); it++) {
148     smash::Sphere* s = *it;
149     applyGravity(s);
150     applyFloor(s);
```

```
151       s->step(1);
152     }
153 };
```

Listing B.18: Octree-based sphere system in C++

Bibliografia

[1] "GTA V dev costs over $137 million, says analyst." http://www.gamesindustry.biz/articles/2013-02-01-gta-v-dev-costs-over-USD137-million-says-analyst.

[2] "js13k HTML5 game development competition." http://js13kgames.com/.

[3] "Node.js – server side JavaScript." http://nodejs.org/.

[4] Google, "Coder for raspberry pi." http://googlecreativelab.github.io/coder/.

[5] "Espruino – hardware board running JavaScript." http://www.espruino.com/.

[6] "Steam store." http://store.steampowered.com/.

[7] E. McNeill, "Exploitative Game Design: Beyond the F2P Debate." http://www.emcneill.com/exploitative-game-design-beyond-the-f2p-debate/.

[8] "Phonegap." http://phonegap.com/.

[9] "ImpactJS – html5 game engine." http://impactjs.com/.

[10] "Turbulenz – html5 game engine." http://biz.turbulenz.com/turbulenz.

[11] "Isogenic – html5 game engine." http://www.isogenicengine.com/.

[12] "Rome – 3 dreams of black." http://www.ro.me/.

[13] M. S. O. Franz, "Code-Generation On-the-Fly: A Key to Portable Software," 1994.

[14] "Firefox to get massive javascript performance boost." http://arstechnica.com/information-technology/2008/08/firefox-to-get-massive-javascript-performance-boost/.

[15] "Improving JavaScript performance with JagerMonkey." https://hacks.mozilla.org/2010/03/improving-javascript-performance-with-jagermonkey/.

[16] "Trace JIT removed from Firefox." http://blog.mozilla.org/nnethercote/2011/11/23/memshrink-progress-report-week-23/.

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

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

[19] "V8 profile log plotter." http://v8.googlecode.com/svn/branches/bleeding_edge/tools/profviz/profviz.html.

[20] S. Hanselman, "JavaScript is Web Assembly Language and that's OK.." `http://www.hanselman.com/blog/JavaScriptIsWebAssemblyLanguageAndThatsOK.aspx`.

[21] "The LLVM Compiler Infrastructure." `http://llvm.org/`.

[22] "clang: a C language family frontend for LLVM." `http://clang.llvm.org/`.

[23] "Emscripten – LLVM to JavaScript compiler." `https://github.com/kripken/emscripten/wiki`.

[24] "Asm.js specification draft." `http://asmjs.org/spec/latest/`.

[25] "Epic Citadel – Unreal Engine 3 technology demo." `http://www.unrealengine.com/en/showcase/udk/epic_citadel/`.

[26] "Epic Citadel converted to JavaScript." `http://www.unrealengine.com/html5/`.

[27] "Ammo.js - physics engine for JavaScript." `https://github.com/kripken/ammo.js/`.

[28] "Optimize for mobile – best practices." `https://developers.google.com/speed/docs/best-practices/mobile`.

[29] Akamai, "State of the Internet," 2013.

[30] M. Ager, "Google I/O 2009 – V8: High Performance Javascript Engine," 2009.

[31] casualconnect.org, "Social Network Games 2012 Casual Games Sector Report," 2013.

[32] D. Clifford, "Google I/O 2012 – Breaking the Javascript Speed Limit with V8," 2012.

[33] J. Conrod, "A tour of V8: full compiler." `http://www.jayconrod.com/posts/51/a-tour-of-v8-full-compiler`.

[34] D. Crockford, *JavaScript: The Good Parts*. O'Reilly, 2008.

[35] J. Dalziel, "The race for speed series." `http://creativejs.com/2013/06/the-race-for-speed-part-1-the-javascript-engine-family-tree/`.

[36] V. Egorov, "Series of articles on V8." `http://mrale.ph/blog/`.

[37] V. Egorov, "Understanding V8," 2011.

[38] C. Ericson, *Real-Time Collision Detection*. CRC Press, 2004.

[39] F. Loitsch, "Optimizing fo V8 series." `http://floitsch.blogspot.com/`.

[40] I. Millington, *Game physics engine development*. CRC Press, 2010.

[41] J. Resig, "Asm.js: The Javascript Compile Target." `http://ejohn.org/blog/asmjs-javascript-compile-target/`.

[42] L. Thompson, "GDC 2012: From Console to Chrome," 2012.

[43] G. Trefry, *Casual Game Design: Designing Play for the Gamer in ALL of Us*. CRC Press, 2010.

[44] V. Vukicevic, "Javascript Developers Conference Taiwan 2013 – High performance games and more in Javascript with asm.js," 2013.

[45] A. Wingo, "Series of articles on V8." `http://wingolog.org/tags/v8`.

[46] N. C. Zakas, *High Performance JavaScript*. O'Reilly, 2010.

[47] "Gartner Says Worldwide PC, Tablet and Mobile Phone Combined Shipments to Reach 2.4 Billion Units in 2013." `http://www.gartner.com/newsroom/id/2408515`.

[48] "Pool Hall Lessons: Fast, Accurate Collision Detection Between Circles or Spheres." `http://www.gamasutra.com/view/feature/131424/pool_hall_lessons_fast_accurate_.php`.

[49] "V8 documentation." `https://developers.google.com/v8/design`.