WebGL based 3D Game Engine

Master thesis
Morten Nobel-Jørgensen
mnob@itu.dk

Supervisors
Mark Jason Nelson / Daniel Povlsen

ITU – Games
Marts 2012

Abstract

With the introduction of WebGL Web browsers became capable of using hardware accelerated 3D graphics, which opened up the possibility of creating a wide range of applications including 3D games.

In this thesis I will investigate how a 3D game engine can be implemented in a browser using WebGL and JavaScript.

First I will take a look at the characteristics of JavaScript and JavaScript engines, by investigating some techniques and software patterns for writing high performance JavaScript, and then examine some of the challenges there are when comes to creating 3D games.

To get hands-on experience I have implemented KickJS – a WebGL based 3D game engine. KickJS is a shader-based engine with a range of built-in shaders. Using the KickJS engine I have also created a shader editor, a scene editor and several small 3D demos.

Finally I have benchmarked the raw performance of JavaScript and WebGL. I have approached this problem from two sides: First comparing JavaScript performance with C++ performance and then seeing how the rendering performance of KickJS compares with the Unity game engine.

In general I found that the JavaScript CPU performance is around 2-4 times slower than C++ based applications. Since much of the calculations are pushed to the GPU, this may not be an issue for small and medium sized games.

The availability of WebGL makes the WebGL browsers a very attractive platform. I expect many small and medium sized games to appear on the platform in the near future.

Table of content

1 Introduction

1.1 Problem definition

The purpose of this thesis is to investigate how a 3D game engine can be implemented in a browser using WebGL, JavaScript and other browser APIs such as HTML5. The main focus is to compare traditional game engines (C++ based) with a browser based game engine in respect to performance, maintainability and ease of use.

The knowledge I share in this thesis is gathered from literature, books, blogs, articles and from building KickJS, a small WebGL based 3D engine that I created during the thesis to get hands-on experience with WebGL game engine development.

1.2 A brief history of game engines

The distinction between a game engine and a game can in some cases be unclear or non-existent. In the past games didn't even have a game engine, but were coded without reuse in mind – or the reusable components were isolated in libraries or modules. But as games grew larger and more complex, it made sense to invest development time in building game engines instead of starting from scratch every time.

Over time game engines have become much more general purpose. The following shows the game engine reusability gamut.

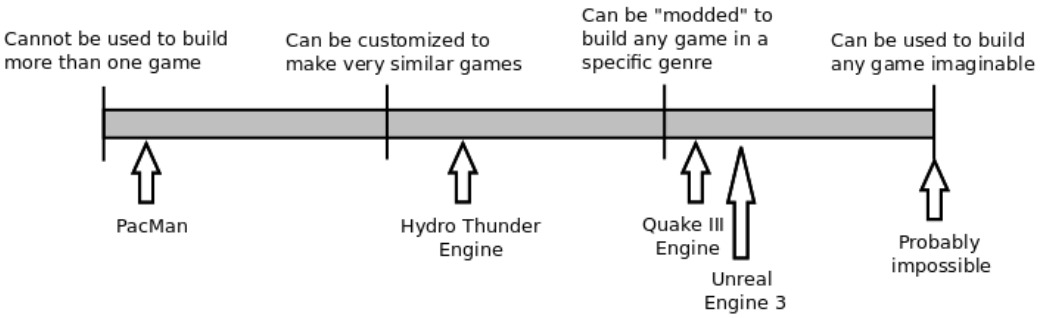


Figure 1 The game engine reusability gamut taken from [Gregory09] page 12

Even though game engines are becoming much more reusable in general, they are often targeting specific game genres, such as first person shooters, real-time strategy, platform games, etc. One reason for this trend is that many AAA game engines are developed alongside a AAA game used for showcasing the engine (e.g. Crysis is created with the CryEngine and Unreal is created with the Unreal Engine).

The Unity game engine is an example of a more general-purpose 3D game engine that is not particular genre related.

1.3 More than just a runtime-framework

Modern game engines is much more that just an application framework giving the foundation of your game. Many game engines include content creation tools, such as scene editors, material editors, particle editors, scripts editors and light baking.

Game engines may also include tools like profilers and debuggers, helping the developer achieving high performance and minimize bugs.

Another feature of game engines is the asset pipeline. The asset pipeline is responsible for seamlessly importing assets (such as scripts, 3d models and 2d textures) into the game engine from external tools. This process includes many stages where the assets are optimized for the game engine and for specific platforms by using different formats and resolutions.

An element of a game engine can either be a runtime component, a development component or in some cases both. In the figure below I have tried to list some of the common elements of a game engine.

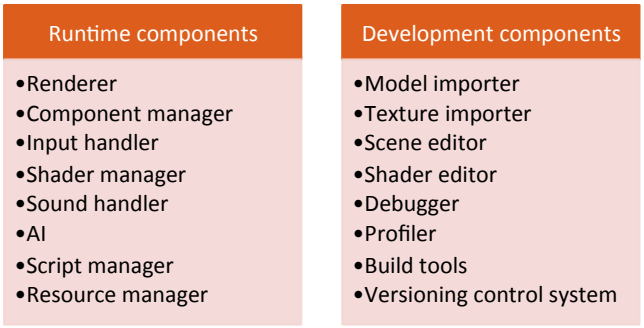


Figure 2 Typical runtime and development components of a game engine

During game development both types of components are being used, but only runtime components are used in the final game.

1.4 The evolution of web applications

When the World Wide Web started to take over the world in the late 90-ties, web applications was created using a web server. A typical use-case would be a user filling out a form on a webpage, and when the user clicked the submit button, the form would be send to a webserver that would create a respond send back also as HTML. The web browser basically worked as a very thin client only responsible for rendering the html and images, and accepting input from the user.

There were a few problems with this simple request-response approach:

- All web pages were static (except for animated gif images)
- Validation of input forms must be done on the server

In 1996 the browser Netscape Navigator 2.0 was introduced. This browser had support for two new features[1]:

- **JavaScript**[2] which allowed more interactivity in the browser such as validation of form data and simple UI changes (mouse-over effects).
- **Support for browser plugins**. The first widely spread plugin was support for Java Applets, which allowed running bytecode directly in the browser. Later came plug-in technologies like ActiveX, Flash, and Silverlight.

The next significant change in browser history was the introduction of Cascading Style Sheet (CSS). This was introduced with Internet Explorer 3.0 in 1996. The purpose of CSS is to separate the styling and formatting from the content of a webpage.

In 2005, Jesse J. Garrett described the Ajax technique[3]. Ajax shorthand for Asynchronous JavaScript + XML and is a technique for communicating with a webserver after the browser has loaded a webpage. Ajax is the foundation of modern web applications since makes the web application much more interactive than would otherwise be possible. A typical use of Ajax is Google's search suggestions that pop up when you start typing a query in the Google search.

During the short history of web browsers the capabilities have increased at an amazing speed. Modern web browsers now have access to the local file system, include a build in database, have support for video and have full screen support. All this without any plug-ins required. At the same time web browsers have become magnitudes faster, both in terms of JavaScript performance and rendering.

One way to look at a modern web browser is as an application platform, which offers a lot of different services, all exposed to client side programs through the JavaScript API.

1.5 The future of browser plug-ins

Since web browsers have become both fast and have a wide range of capabilities, there is no longer the same need for plugin as previously. Today many of the things you previously had to use a plugin to run in a browser are now supported directly by the browser. In other words, plugins now replicate many of the same features that the browsers support (canvas rendering, sound, file access, databases, cache, etc.). Today the browser is no longer the thin client it used to be, but is usually larger in both memory footprint and file size than most plugins.

---

[1] Metzger11
[2] Note that the JavaScript programming language is unrelated to the Java programming language, except for few similarities in syntax. JavaScript was later standardized into the ECMA script language.
[3] Garrett05

Besides many of the web browsers run on mobile phones, tables and other devices that do not have support for browser plug-ins due to limited resources on the devices.

The trend in web application development is to not use plugins, but to rely on solutions created in pure HTML, CSS and JavaScript. Even major players on the marked are turning their back to their own plug-ins in favour of HTML:

- Adobe has discontinued Flash for mobile devices to focus on HTML5 instead[4].
- Microsoft was announced that Windows 8 supports applications build using web technology (HTML5 and JavaScript)[5].

Even though browser plug-ins definitely will be less used in the future, I'm convinced that they will exist in many years from now. The reasons for this is:

- Legacy applications. A lot of time and money has been invested into applications that use plugins. It may not be profitable to port these applications to pure web applications.
- Even through browsers are getting much more standardized, there is still quite a few differences that can make web development painful.
- Even though performance of JavaScript and browsers has improved a lot, plug-ins usually gives much better performance.

One example of a modern, high-performance plug-in is Google's Native Client (NaCl), which allows the users to run native binary code written in C++ in the browsers sandboxed environment.

1.6 Adding a splash of 3D

One of the hot topics of web development today is support for 3D accelerated graphics. WebGL 1.0 was released in March 2011[6]. WebGL is an API based on the OpenGL ES 2.0 graphics API - a shader-based graphics API targeted to run on handheld devices. Since OpenGL ES 2.0 is based OpenGL 2.0 it will run on any desktop computer capable of running OpenGL 2.0.

WebGL has the potential to be quite successful since most of 3D data and calculations can be uploaded to the GPU. This means that all the heavy work are done by the GPU whereas the JavaScript part of a WebGL application is responsible for invoking draw calls and changing state (such as shaders, vertex buffers and textures).

[4] Winokur11
[5] Larson-Green11
[6] Webgl11

A simple rendering of a full 3D model can be done as simple as this[7]:

```
gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
gl.vertexAttribPointer(0, 3, gl.FLOAT, false, 0, 0);
gl.uniformMatrix4fv(uMVPMatrixLocation, false, mvpMatrix);
gl.drawArrays(gl.TRIANGLES, 0, 3);
```

The great about this approach is the complexity of the models rendered does not affect the workload of the JavaScript; the exact same sequence of WebGL calls is made in both cases. The GPU on the other hand will be affected by the size of the mesh. This is great since the CPU is likely to be the bottleneck in a WebGL powered program.

WebGL has been available since Firefox 4.0 and Chrome 9.0 both from 2011. Microsoft is the only major browser vender who is not planning to include WebGL. To use WebGL in Internet Explorer you need to install a plugin[8].

1.7 HTML games

Ever since JavaScript was introduced as a scripting language for web-browsers, it has been used for creating simple games. Since then both browsers and computer hardware has gotten faster, which has made it possible to create increasingly complex games and interactive applications. Games created in JavaScript are slowly taking over games previously created using plugin technologies such as Flash and Java.

I think it is important to think of games in terms of complexity. The figure below shows the complexity of games genres.

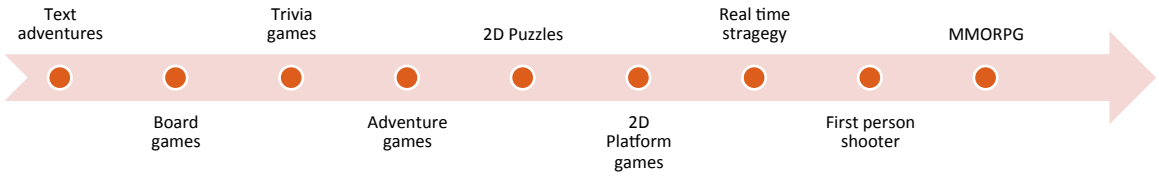


Figure 3 Game genres sorted by complexity

JavaScript based games have over the last years been used for creating more complex games and with the introduction of WebGL it will be capable of handling even more complex games. I expect that a lot of the 2D and 3D casual games will end up being made in pure JavaScript.

[7] Note that prior to the rendering a setup-phase needs to upload the mesh to the GPU and compile and bind a shader.
[8] Chrome Frame (the full Chrome browser as a plugin) or IEWebGL a more traditional plugin

On the other hand, I don't expect JavaScript will ever be the language for creating cutting edge games, like C++ is today. JavaScript is simply a too high-level language, which adds a runtime overhead and may not be able to take full advantage of the hardware. This will be discussed in details in chapter 2.2.

1.8 Summary

WebGL makes is possible to create hardware-accelerated 2D or 3D games running inside a browser without needing any plugin. This makes it possible for creating more complex games for web browsers since it makes the rendering hardware accelerated.

I believe that the games written in JavaScript will slowly take over games that were previously written using browser plug-ins – especially for less resource demanding games.

2 JavaScript based game engine

Based on working with game engines I find that a good game engine should have the following goals:

1. **Performance**. The game engine must run as smoothly as possible. Writing efficient code is traditionally done in a low level language such as C or C++ (with inline assembler code in critical sections). This approach gives the engine programmer absolute control for optimizing critical sections of the code where the compiler is not creating the optimal solution.
2. **Ease of development**. Game programmers and level designers are the primary users of a game engine. Usually they work using a more high-level programming language such as Lua, Unreal Script or Python or perhaps they use a visual programming language such as Unreal's Kismet.
3. **Abstraction**. By taking away some of the complexity of the platform game developers can work with concepts close to their problem domain and not worry about the low level details.

These three goals are the same regardless of the platform and the technology of the game.

An important thing to realize is that these goals often contradict one another. High performance often contradicts abstraction, since abstraction often adds a level of indirectness to the code.

In this chapter I will investigate how suitable JavaScript is as a language for writing a game engine. I will first take a look at the best practices for writing high performance JavaScript and then evaluate how suitable JavaScript is for 3D game engine programming.

2.1 Writing efficient and maintainable JavaScript

To get the maximum performance out of JavaScript code it is important to understand the characteristics of the core of the language and the runtime characteristics of JavaScript engines. Even though modern JavaScript code looks like code written in an object-oriented language, JavaScript is not object oriented but prototype-based. However it can mimic some of the object oriented ideas. In this section I will go through some of the language features and how efficient code can be written for each of these features.

As [Gove11] writes in the chapter "How Not to Optimize", "In general, it is best to avoid optimizations that make the code less easy to read. The best approach is to make minimal changes to the source code or to selected improved compiler flags." This rule is still valid in JavaScript, but the JavaScript may not optimize as aggressively as a C++ compiler, which means that for performance critical parts of your code, you may sacrifice simplicity over performance.

2.1.1 Avoid implicit typecast

Since JavaScript has dynamic types, the equal operator == tries to typecast before comparing the values. This means that you should always use === when comparing two objects, since this operator performs a strict comparison without attempting any typecast. In the rare cases where you allow objects to be type casted you should use ==.

```
'3' == 3; // evaluates to true
'3' === 3; // evaluates to false
```

Code 1 Implicit type casts (==) explained

By avoiding implicit typecast you gain a performance improvement of around 15 % (measured using a Unit Test)[9].

2.1.2 Use of Typed Arrays

Typed Arrays is a new feature in JavaScript that allows you to do two things:

1. Allocate a byte buffer in memory (using `ArrayBuffer`)
2. Give a view on that byte buffer as a array of a predefined type (a subclass of `ArrayBufferView`).

Typed arrays should be used in the following two use cases:

1. **Binary data**: Typed arrays allow you to efficiently work with binary data
2. **Optiming memory usage**: The programmer now has full control of the memory layout (allowing performance tuning memory access patterns)

The performance gain from using typed arrays is between 20% and 40% over JavaScript arrays[10]. The actual gain depends on the data-type.

2.1.3 Avoid chaining

JavaScript is indeed a dynamic language. It is not possible to declare a constant variable. This has some interesting side effects.

In a language as Java an expression like: `Math.PI*2` will by the compiler be translated to into `6.28318531` because `Math.PI` is declared final (and therefore it is considered safe to copy it's constant value).

However if a JavaScript has to evaluate the same expression it would have to lookup PI on the Math object every invocation and then multiply the result with two[11].

---

[9] See 7 Appendix A: performance tests
[10] See Appendix 7.1.1 for more details.
[11] The JIT compiler is good at optimizing this, but there still is a small overhead compared to using constants.

One solution to this problem is to declare a variable outside the function like this:

```
var PI = Math.PI;
function foo(){
  print(PI*2);
}
```

The function `foo` no longer has to lookup `PI` in an object, but can access the cached variable through its closure. The longer chain you cache in a variable, the bigger saving (this is very important when using the namespace pattern – it may not give any significant performance gain in the example above).

A second solution to the problem would be to simply replace PI*2 with the constant 6.28318531, however this is bad programming practice and result in unmaintainable code, since many programmers would probably have a hard time figuring out what the number actual means.

A third solution is to use a pre-processor to replace any value of symbols with actual content.

2.1.4 The namespace pattern

On common problem with writing code is that you often want to wrap your code in a namespace to avoid conflicts with other libraries and to keep things separated. JavaScript does not support namespaces, but it is easy to write a function that simulates namespaces. This function defines an object unless it is already defined. The function supports nested namespaces when a dot-separated string is used.

The pattern is closely related to the lazy instantiation pattern. More details of the namespace pattern can be found in [Crockford08] and [Stefanov10].

2.1.5 The constructor invocation pattern

One of the classic mantras in object-oriented design is high cohesion and low coupling (these are two of the GRASP patterns described in [Larman97]). At first sight JavaScript doesn't seem support neither classes nor encapsulation.

However you can easily use JavaScript in a way, where functions acts as if they were constructors.

```
function Car(){
    // private variables
    var position = 0;

    // private methods
    var lockDoors = function(){
        // …
    };

    // public methods
    this.drive = function(){
```

```
            lockDoors();
            position++;
        };

        // public variables
        this.name = "My car";
    }
```

Code 2 Constructor pattern

The use of capital letter in Car is very important. It tells the programmer that the function acts as a class constructor and that he must use the `new` keyword to create a new instance of the class. The `position` and `lockDoors` variables are private and are not accessible on the object. But since the two variables are used in the `drive` method, they are a part of this methods closure, so this method does have access to them. Note that new methods added to the object after construction does not have access to the private variable either.

The constructor invocation pattern is described in [Crockford08]. The private properties and methods are described in [Stefanov10]. This approach is one of the more straightforward ways to implement a class in JavaScript, but there are several other alternatives not discussed here. One of the problems of this approach is that it does not support true inheritance that works with the `instanceof` operator.

2.1.6 Using Object Oriented API design

The main reason for using class-like structures in a prototypical and functional language like JavaScript is for the design and documentation of the application.

Object oriented design is by far the most common method for software design. Object oriented design scales well to large problems and there exist many tools and techniques for dealing with design challenges. One such tool is Unified Modelling Language (UML) which objective is to "provide system architects, software engineers, and software developers with tools for analysis, design, and implementation of software-based systems as well as for modelling business and similar processes"[12].

Since object oriented design is currently the most widely used methodology for designing software, programmers are very familiar with the concepts and the mind-set. For this reason it also make sense to use an object oriented programming style and documentation style. This will make the design decisions much more understandable and make it easier for programmers to transform software design into code.

[12] [UML11] Page 1

2.1.7 Using properties

One of the new features in JavaScript 1.8.5 is the introduction of properties. This allows the developer to specify getter and setter methods, called when a property is accessed on an object. The feature also allows you to have a property that is read-only or write-only, and you can specify whether the property should be included when the object is enumerated[13].

Properties can be used for doing validity checks on variables in the setter function. Prior to the introduction of properties you would have to expose a getter and setter functions yourself for this to work. A short example of properties usage:

```
function Car(){
    var speed = 0;
    Object.defineProperty(this,"speed",{
        // define getter function
        get:function(){
            return speed;
        },
        // defines setter function that checks type
        set:function(newValue){
            if (typeof newValue === 'number'){
                speed = newValue;
            }
        }
    });
}
```

Code 3 Example of type check in property getter function

2.1.8 Using strict JavaScript

Traditionally JavaScript allows the developer to use variables without declaring them first. Doing this will declare the variable in the global scope. This means that you are actually cluttering up the global namespace while having global object references that you might not intend.

```
function foo(){
  x = 10;
}
foo();
this.x; // now evaluates to 10
```

Another related problem is if there is a typo in the variable name, the JavaScript engine will allocates a new variables in the global namespace. This leads to runtime bugs.

To solve these issues, using a variable without declaring it first is no longer legal in the newest JavaScript (ECMAScript 5). To enable this rule, the JavaScript engine needs to be run in strict mode. Strict mode is valid in the current context it is enabled in, meaning that

[13] Iterating over the property names of an object

you can enable strict mode in either global scope or in a function. To enable strict mode the “use strict” has to be defined. Example

```
function foo(){
"use strict";
// [...] function body now uses strict mode
}
// outside uses backwards-compatible mode
```

Another benefit of using strict mode is that a modern JavaScript editor (such as WebStorm) can identify problems like these while the code is being written.

2.2 Evaluation of JavaScript as a language for 3D engines

In this chapter I will reflect on some of the pros and cons of using JavaScript for a 3D game engine.

The chapter mainly focus on the problems that I have encountered during the implementation of the KickJS engine and methods I have used to overcome the problems.

2.2.1 No compile time

One problem with many larger game engines is the amount of time spent on compiling the game. Even small games often take longer time compiling than brewing a cup of coffee. This breaks your concentration and forces you to suddenly do something else.

Image 1 http://xkcd.com/303/

One of the brilliant things about using JavaScript is there is no noticeable compile time. You simple write your code and switch to your browser to see if it works. In cases it don’t work, you often tweak the implementation by altering the code from the browsers JavaScript console – or use the browsers debugger to identify the problems. Or use the browser’s built-in profiler to instantly find the performance bottlenecks.

2.2.2 Serialization

JavaScript has built-in support for a few way of serializing and deserializing:

Encoding	Pros	Cons
String	Good for simple data and textural data.	For more advanced data structures JSON or XML should be preferred.

Encoding	Pros	Cons
XML	Can be verified against XML schema Good for communication with other system (such as webservers), since XML is widely supported. The data is human readable, since the tag name provides meta-data.	XML is not a compact data format. Binary data needs to be encoded into a string using Base64 encoding (or similar string encoding) XML is a bit cumbersome to use in JavaScript.
JSON	Very natural to use in JavaScript More compact than XML. Human readable (but without meta-data) Build in support using JSON class.	Binary data needs to be encoded into a string using Base64 encoding (or similar string encoding)
Binary Data (ArrayBuffer)	Compact data layout Good for low-level data access	No built-in string support

Table 1 Serialization options in JavaScript

Serialization in JavaScript is easy and usually quite elegant. JSON is the only method that can serialize objects automatically. When deserializing data, the programmer must in all cases do some coding for the correct object to be created.

2.2.3 Lack of operator overload

3D game programming uses a lot of math with vectors, matrices and quaternions. Many languages supports operator overloading, which allows you to define behaviour on how objects should behave when operators (such as `*`, `/`, `+`, and `++`) are used with them. Operator overloading is just syntactic sugar and will in the end result in a normal function call. However the benefit of using operator overloading is to increase the readability of the code.

As an example the reflect vector can be calculated this way in C++:

```
vec3f reflect(vec3f& L, vec3f N){
   return L - 2.0f * L.dot(N) * N;
}
```

Code 4 Reflection vector in C++ (with a vec3f class using operator overload)

JavaScript does not support operator overloading. This means that the same code would look like this.

```
function reflect(L, N){
  // performs L - 2.0f * L.dot(N) * N
  return L.subtract(N.scale(- 2 * L.dot(N)));
}
```

Code 5 Reflection vector in JavaScript (L and N is a vector objects with the methods subtract, scale and dot)

Note that I had to reorder the expression a bit to make it work. When working with large mathematical expressions the code gets hard to read and debug. For this reason you would often write code in pseudo code in a comment next to the expression as documentation.

2.2.4 Memory allocation

In C++ it is up to the programmer if he want to allocate an object on the stack or on the heap. Stack allocation is very fast since the runtime just need to increase the stack-pointer with the size of the data you are allocating. Another benefit is the stack is always "hot" – memory around the stack pointer is likely to be in CPU cache. In heap allocation the memory manager has to find a suitable free chunk of memory and mark it as used, this makes heap allocation slower than stack allocation. The choice of heap vs. stack storage also depends on the lifetime of the data; data on the stack has the lifetime of the current function, whereas data on the heap can live as long as the program is running.

C++ even go one step further and allows the programmer to use inline assembler for cases where manual written assembler code can beat the machine code generated by the compiler. Assembler code gives the programmer full control over which assembler instructions are called and what CPU registers are used, but it makes the code less portable and more complex to read and maintain.

JavaScript is a much more high-level language. All objects in JavaScript are allocated on the heap, but note that not everything in JavaScript is an object[14]. The idea behind this is that the memory allocation should not be the programmer's responsibility, but instead something that the JavaScript engine should find the optimal solution for.

The problem is that the runtime does not always handle this situation well. In my opinion, the real problem is that there is no way of creating temporary objects in a cheap way and that the JavaScript Engines are not good at optimizing cases where temporary objects are being used.

[14] See more details here [Rauschmayer11]

Let's revisit the reflect method once again – this time refactored to show how the method performs in terms of memory allocation.

```
function reflect(L, N){
  var lDotN = L.dot(N); // Allocate primitive (stack)
  var lDotN2 = - 2 * lDotN; // Allocate primitive (stack)
  var scaledNormal = N.scale(lDotN2); // allocate temporary (heap)
  return L.subtract(scaledNormal); // allocate result (heap)
}
```

Code 6 Memory allocation in JavaScript reflect method

As can be seen the `scaledNormal`-variable is really only used internally and therefor it is inefficient that the variable is allocated on the heap. It may be ok that the results is allocated on the heap as a new object, but in many cases what you really want to do, is updating an existing object.

High performance JavaScript math libraries, such as glMatrix used in KickJS, are aware of this issue. They are implemented with a strict rule of not allocating new objects in the methods.

Optimizing the reflect function to not allocate new objects would look like this:

```
function reflect(L, N, res){
    if (!res) res = vec3.create();
    var lDotN = vec3.dot(L,N);
    var lDotN2 = - 2 * lDotN;
    var scaleNorm =
      vec3.scale(N,lDotN2, res);
    return vec3.subtract(L,
      scaleNorm,res);
}
```

Code 7 JavaScript reflect method without object allocation

```
vec3f reflect(vec3f& L, vec3f N){
   return L - 2.0f * L.dot(N) * N;
}
```

Code 8 The original C++ version of the reflect function

In the JavaScript code above no objects are allocated unless the `res` input variable is not specified. The performance gain by using this approach is several hundred percent[15]. The performance improvement is gained from the reduced time spent on memory allocation, but also from the time spent by the garbage collector, which will be discussed in the chapter below.

The cost of this performance gain is the reduced code readability and increased code complexity. The original C++ code was one line, whereas the optimized JavaScript is 5 lines of code.

[15] More details in the Appendix 7.1.2

2.2.5 Garbage collection

In a language like C/C++ it is the programmers responsibility to deallocate memory allocated on the heap. In many cases it can be hard to figure out when this deallocation should occur, and the programmer has to make sure that this deallocation is done correct so that memory is not leaking.

JavaScript is a garbage-collected (GC) language, which means that you no longer have worry about memory leaks. However to achieve high performance the best strategy is to reduce object allocation, this will reduce both the allocation time and the time spend on garbage collection.

JavaScript engines uses different garbage collection algorithms as described in [Mandelin11]:

- Mark and sweep
  - A naïve two-step algorithm that first marks all reachable objects and then recycle object that are not reachable.
  - The algorithms results in long GC pauses (100 milliseconds or more)
- Generational
  - Objects are divided into groups based on their age. Young objects are checked frequently, where as old object are checked rarely.
  - Generally this makes GC less resource demanding.
  - In a nursery collection the young objects are checked. Nursery collections tend to run fast.
  - In a tenured collection the old objects are checked. Tenured collection tends to run slow (but still better than mark and sweep).
- Incremental
  - Instead of running a full garbage collection, the incremental algorithm divides the garbage collection into small time-slices.
  - Removes the long GC pauses and is ideal for interactive applications.

The current generation of JavaScript engines use generational garbage collectors. The next generation of garbage collectors for JavaScript engines will be incremental[16].

2.2.6 Resource management

JavaScript is a language that uses garbage collections, which takes care of deallocating objects that is no longer referenced. This makes life easy for programmers: the only have

[16] This will be introduced in Chrome 17 (http://blog.chromium.org/2011/11/game-changer-for-interactive.html). For Firefox incremental is scheduled to Firefox 11 (https://bugzilla.mozilla.org/show_bug.cgi?id=641025)

to clear all references to objects no longer in use and the system will do the rest. This is usually a simple task, but the usage of closures can complicate the task.

Another significant task of a resource management system is to avoid loading the same resource more than once. The problems of loading a resource more than once are the time it takes to load the resource, the memory the resource occupies and finally by reusing existing resources often gives much performance due to the cache utilization of modern computer architecture. When JavaScript is used for scripting a web-browser, the web-browser makes sure that images and other resources are not loaded unneeded. This means that frontend JavaScript developers do not need to think much about resource management and resource sharing when creating web applications.

Even though WebGL is a JavaScript API, it is not an object oriented API. WebGL is a based on OpenGL ES 2.0 API and inherits much of its C-based structure. This means that WebGL resources are identified using integer identifiers and not object references as JavaScript developers are familiar with.

Developers using WebGL has to manage resources much more explicit than other JavaScript APIs, this includes both allocation and deallocation. Let's see how other languages deals with resource management.

Resource management in C++

C++ is an object-oriented language, where each class has both a constructor and a destructor. The usual solution to resource management in C++ is the Resource Acquisition Is Initialization programming idiom (RAII). In RAII resources are allocated in the constructor and deallocated in the destructor. This approach is very simple to implement and maintain.

Resource management in Java

Java is a language with garbage collection, just as JavaScript. However Java is an object-oriented language, where classes have constructors and also an optional `finalize` method. The `finalize` method is guaranteed to be called when the object is marked for garbage collection.

A typical example is the `FileInputStream` object, where the `finalize` method calls the `close` method if the programmer has not explicit called the method.

Dealing with explicit resource management in JavaScript

Unfortunately JavaScript cannot implement RAII, since objects in JavaScript does not have destructors. Using a reference counting schema like C++ smart pointers is also not possible in JavaScript, since there is no copy constructor in JavaScript either. Reference

counting can be implemented if the counter is increased or decreased explicit with a function call, but this approach is tedious to use.

The `finalize` method approach works great in Java, but since JavaScript does not call any `finalize` method when an object is marked as garbage, this approach cannot be used.

JavaScript simple does not handle explicit resource management as used in WebGL very elegant. Care must be taken to loading resources only once and releasing them at the right time.

2.2.7 Late binding

JavaScript is a dynamic programming language with late binding. This means that the attributes and methods of an object can change at runtime, which means that when the program invokes a method or is accessing an attribute of an object, the runtime needs to lookup the method/attribute in a map that the runtime has associated with the object[17].

This is a very fundamental language feature of JavaScript and allows you to do a lot of clever things with the language, such as lazy instantiation of properties, the namespace pattern and a lot of the other JavaScript patterns.

The late binding does have a price: There is no ways of specifying true constant variables.

Constant values are important to work with to give meaning to numbers and objects instances. One good example of why constants would be nice to have is when working WebGL. The WebGL context object exposes all WebGL enums (such as `gl.ZERO`). There are two problems with this approach:

- There is some overhead involved in reading a property; The JavaScript engine has to lookup property value in the objects property-map every time the property is read. If the language supported constant objects the value would be set at compile time, which allows the compiler to do some more optimizations, such as constant folding where expressions are evaluated at compile time.
- The properties are modifiable. So it is possible to assign the value one to the enum `gl.ZERO`, which is unlikely to make sense. There is workarounds for this – see chapter below.

A related topic is pre-processor macros (used in languages like C/C++). A pre-processor is an advanced text-replacement that runs before the actual code is compiled.

[17] This has similarities with the virtual method table used in C++ runtimes.

I have implemented a pre-compiler for the KickJS engine. The KickJS pre-compiler replaces WebGL enums to constants and also replaces debug and an assert flag with either a true value or a false value. The debug and assert flag are used to add extra checks to the debug builds.

The performance gained by introducing this pre-compiler is between 2% and 6% and the code size is reduced with 10%. The performance gain is however highly dependent of the actual usage and the JavaScript engine.

Objects are always modifiable

Prior to JavaScript 1.8.5[18] there was another issue: Objects are always modifiable. Even though this is also listed as one of the benefits of JavaScript, in some cases this can feature can be problematic. A good example of when this can be problematic is when creating a JavaScript library, where you usually are not interested in other JavaScript programs modifies the included library.

This issue has been resolved in JavaScript 1.8.5, where the `Object.freeze` method was introduced. Frozen objects cannot change its signature and the value of the existing properties can neither be changed. A related method is the `Object.seal` that only put restriction on changing the signature.

Another related technique to define constants is to use read-only properties using one of the `Object.defineProperty` / `Object.defineProperties` methods.

2.2.8 Just-In-Time compilers

Modern JavaScript engine all use Just-In-Time (JIT) compilers to increase their performance. A JIT compiler compiles JavaScript bytecode to machine code executed directly on the CPU. Performance critical parts of JavaScript are compiled using a more efficiently type-specializing JIT compiler. Rare cases are still executed by the JavaScript interpreter.

[18] ECMAScript 5 compatible

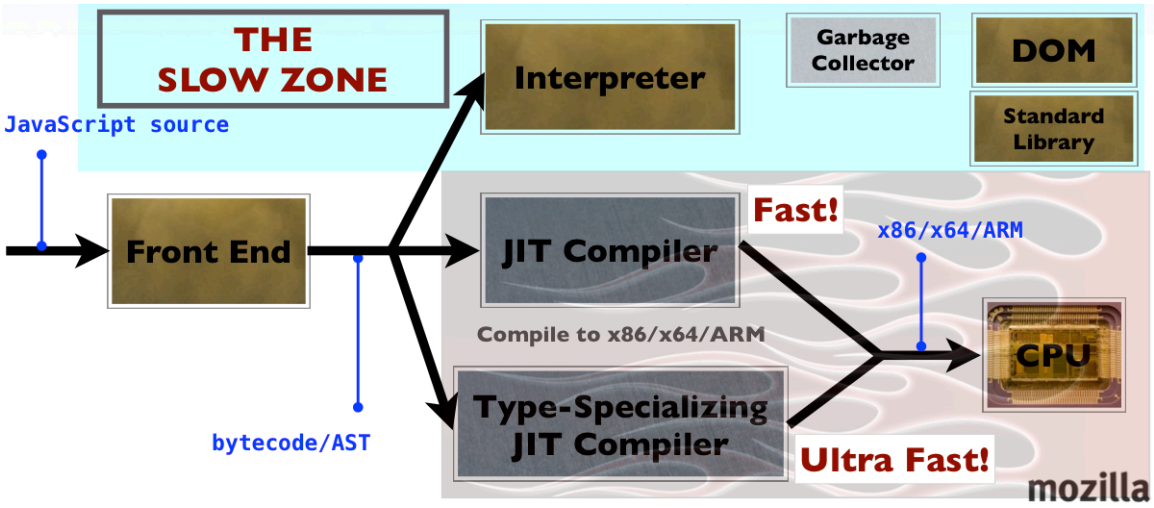


Image 2 JIT compiler illustrated (taken from [Mandelin11])

The JIT compiler does a lot of performance tricks such as inline caching, which reduces the overhead of accessing properties on objects and accessing closure variables.

One problem the JIT compiler has to deal with is that JavaScript doesn't have types. This means that type-checks still need to be executed whenever a variable is accessed and that values need to be boxed/unboxed. To solve this problem a type-specializing JIT compiler is used. The type-specializing JIT compiler monitors the types of objects and recompiles if the types of an object is changed. Type-specializing JIT compilers takes longer time to compile code, and is only used on hotspots in the code. The JavaScript engine finds these hotspots by sampling the running code.

The JIT compiler is the main reason for the performance improvements we have seen in JavaScript engines during the last 5 to 10 years. However the JIT compiler also introduces some complexity, since you need to know the behaviour of different JavaScript JIT compilers to write JavaScript code that performs well on all platforms.

2.2.9 Number precision and performance

In JavaScript there is only one numeric type with double floating point precision (64-bit)[19]. This simplifies calculation in JavaScript a great deal and since the type with the most precision is chosen floating point accuracy problems is less likely to occur.

One of the new features in JavaScript is typed-arrays. In some way this adds numeric types to JavaScript. But one important thing to realize is that when a number is read from

[19] At least if you look at the ECMAScript specification. JavaScript engines may have a SMI (SMall Integer) type as a performance optimization. But this doesn't really affect the discussion in this chapter, since it is really the single precision floating point that is needed.

a typed array, it is casted to the default numeric type. This means that all calculation in JavaScript is still done using double precision[20].

Game engines usually prefer 32-bit floating-point precision for all their floating-point calculations, since this gives sufficient precision for games. The main reason for using 32-bit precision is in terms of memory bandwidth. One fundamental problem with modern hardware is the CPU is significant faster that the memory. Since 64-bit data-types of takes up twice as much bandwidth as 32-bit data-types, it is much faster to use 32-bit precision when working with large data sets.

Note that it may be beneficial to use typed arrays in JavaScript in cases where the performance is memory bound. (Even though each access to the typed array will add a small overhead, since the number will be casted to a 64-bit floating point). This behaviour can be seen in appendix '7.1.1 Benchmark: Typed arrays', which clearly shows performance improvements in using smaller data-types.

SIMD instructions

3D games have in many ways had a major influence on hardware design such as CPUs and GPUs. One of these enhancements was the introduction of single-instruction-multiple-data (SIMD) instructions that boost vector and matrix calculations on the CPU.

On the desktop marked the most common SIMD instruction set is currently SSE2 and SSE3, used in both AMD and Intel CPUs. The instruction set is mainly designed for 32-bit floating-point values, which unfortunately means that the JavaScript engines cannot use this instruction-set for much.

2.2.10 Multi-threaded programming

JavaScript has traditionally been single threaded. A JavaScript engine context, such as a web-browser, uses an event based programming model, where you hook up JavaScript to different event listeners or timers. This means that all JavaScript is running in the same event thread in callback functions scheduled by the event system. If some JavaScript code runs for a long time, the context it runs in becomes unresponsive.

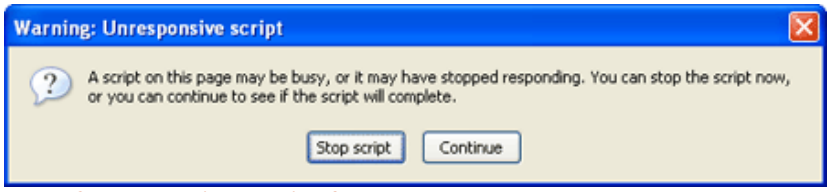


Image 3 Unresponsive warning from a web-browser

20 See 'Double precision test' in Appendix 7.2.3

To keep the browser responsive, some built-in functions in JavaScript can execute asynchronous in its own thread. The callback events from an asynchronous operation are delegated back to the event thread, so all JavaScript executions happens in the same thread. A typical example of this, is script-based image loading where you specify an `onload` event handler that will be triggered in the event thread once the image has been fully loaded.

Until recently it was not possible to have user-defined code to run in a separate thread, but that changed with the introduction of the Web Workers API. Web Workers are a simple way to create a true multithreaded program in JavaScript, which allows you to run heavy computations without blocking the JavaScript context.

Another reason why Web Workers are important is the rise of multicore CPUs. Today both PCs and handheld devices are shipped with multicore CPUs with each CPU-core possible having multiple hardware threads. In order to use the hardware to its full potential, JavaScript simply needed a way to do threading.

Multithreading usually adds a lot of complexity to a programming language. Multithreaded programs can suffer from data races, where multiple threads work on the same data. To solve this problem programming languages often introduce a synchronization mechanism, such as mutex-locks and critical regions, but this yields another set of problems: deadlocks and livelocks, where two or more threads get stuck waiting for each other.

JavaScript Web Workers take a different approach: message parsing. In Web Workers you simply cannot share data, which means that there is no data races. Web Workers communicate using message passing, where the message content is copied by value not by reference[21]. Web Workers does not have access to resources in the JavaScript context, such as the DOM elements.

One potential problem with Web Workers is that data needs to be copied whenever passed between threads. This makes Web Workers best suitable for problems with low communication between threads[22].

In relation to game engines Web Workers also is a bit problematic, since there is no way to join two threads. This makes Web Workers only useful for jobs running "between" two frames.

---

[21] Web Workers are very similar to the Message Parsing Interface (MPI) programming model

[22] This is solved by the Transferable Objects introduced in Chrome 17, which transfers ownership (and access) instead of copy data.

2.2.11 Hardware abstraction layer

Both browsers and WebGL acts as a hardware abstraction layer, which generalizes the access to the hardware so that the hardware can be accessed in a uniform way while the layer fixes known bugs on problematic platforms.

One good example of this is how WebGL is implemented on the Windows platform in Firefox and Chrome. For WebGL to run an OpenGL 2.0 driver needs to be available. On windows platforms such driver may not exist or may be in poor shape. So instead of requiring a driver update, browsers use the open source project ANGLE (Almost Native Graphics Layer Engine[23]) which builds the OpenGL ES 2.0 API on top of DirectX 9 API, which is generally very well supported on windows hardware.

The hardware abstraction layer let the developers focus on their problem domain instead of spending their time on dealing with different hardware architecture, drivers and low-level bugs. This also means that the same WebGL code should be able to run on any platform with OpenGL 2.0, OpenGL ES 2.0 or DirectX 9 supports, which includes most new smart phones, tables and computers.

The wide range of devices running WebGL does have different capabilities. WebGL deals with this in two ways:

- **Hardware queries:** Programs running WebGL can query the API to determine different capabilities. Usually these capabilities specify the maximum numbers of different resources based on the actual hardware. This includes things as the number of texture sampling units for both the vertex, the fragment shaders and in total.
- **Extensions:** Allows vendor specific extensions to be used. WebGL extensions require that both the browser and the hardware support that extension. One such extension is the floating-point textures.

The problem with using a hardware abstraction layer is that hardware often has capabilities that are not supported in the hardware abstraction layer. One example of this is that modern GPUs support geometry shaders and tesselation shaders, but no WebGL implementation supports these shaders yet.

2.3 Summary

JavaScript is an easy-to-use high-level programming language that programmers can use without needing to deal the underlying details such as memory management, data types, and other low level complexity. JavaScript usually result in clean code that is very easy to read and understand.

[23] http://code.google.com/p/angleproject/

Writing high performance JavaScript is a bit of a challenge. The programmer needs to know not only the language specification but also how different JavaScript engines work internally. When working with 3D graphics, the JavaScript code often gets less readable due to the lack of operator overload in the language and due to code that avoids memory allocation.

If you characterize JavaScript-based game engines in terms of the four goals listed in the beginning of the chapter you get:

1. Performance. Even through the speed of JavaScript has increased over the last 10 years; there are still areas where JavaScript cannot achieve the same performance as native C++ code.
2. Ease of development. JavaScript is a really easy and simple language and browsers today includes profilers, debuggers and command-line interpreters. JavaScript game engines can build on top of this and provide game engine specific tools such as shader editors and scene editors.
3. Abstraction. JavaScript engines can easily encapsulate the complicated WebGL API and expose a simplified API for the game developers with meaningful abstractions.

3 KickJS – a WebGL game engine

In this chapter I will discuss the implementation of the KickJS game engine. This includes the design decisions I made during the implementation. The chapter will highlight the most important features of the engine, but will not give a complete description of all the details. For more in depth information see the API documentation and the source code.

3.1 High-level overview

KickJS is a WebGL based game engine built for modern web browsers such as the most recent versions of Chrome and Firefox[24]. The engine provides the infrastructure for WebGL based games and takes away the low-level complexity of WebGL. The engine targets JavaScript programmers who want an easy to use and well-documented engine. The engine is shader based, but ships with built-in shaders.

The source code is released as Open Source under the BSD New License. The engine, the documentation and the examples can be found here:

http://www.kickjs.org/

3.2 Programming style and documentation

KickJS uses both an object oriented programming style as well as an object oriented API documentation. The API documentation is created using YUI Doc, which is a mark-up based documentation. YUI Doc is very similar to JavaDoc, but with one significant exception: all documentation comes from the documentation mark-up tags, where in JavaDoc package-names, class-names and method-names are extracted from the source code. YUI Doc generates documentation in HTML files. An example of YUI Doc documentation tag can be seen here:

```
/**
 * A driveable car
 * @class Car
 * @constructor
 */
function Car(){
    /**
     * Moves the car forward
     * @method drive
     */
    this.drive = function(){ /* … */ };
}
```

Code 9 YUI Doc tag used to document JavaScript

[24] At the time writing this is Chrome 17 and FireFox 10

3.3 Scenegraph, rendering and the game loop

In this section I will describe the main ideas behind the core parts of the KickJS engine; the scenegraph, the rendering and the game loop.

3.3.1 The entity based approach

In the past game programmers often used deep class hierarchies to represent game entities. The idea here is that game entities exist in a game world and have spatial information. Typical entities found in a game could be: player, car, weapon, grenade, etc.

```
Entity
 +- Static
 |  +- Ladder
 |  +- Portal
 |
 +- Moveable
 |   +- Vehicle
 |   |   +- Car
 |   |   +- Boat
 |   |
 |   +- Human
 |       +- Player
 |       +- Enimy
 +- Gun
```

Figure 4 Example of deep class hierarchy

The legacy way of modelling game entities is to use class inheritance where general types are pushed upwards in the class hierarchy (such as Moveable in Figure 4 Example of deep class hierarchy). As new classes are added, shared functionality is again pushed up to parent classes by introducing new methods. This eventually results in very heavy super classes also known as “the blob” (an anti-pattern).

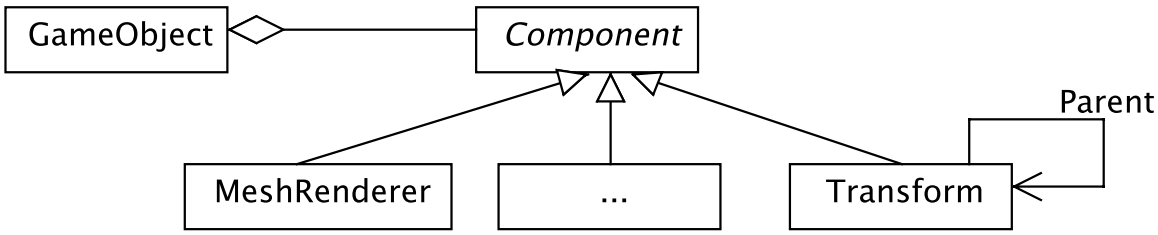
Another related problem is that hard to combine objects (such as having a vehicle that is both a car and a boat).

One of the main reasons that this approach has problems is the high coupling that exists between classes. This makes it hard to introduce new classes and to change existing classes. A deeper discussion of entity based vs. component-based design is found in [West07].

3.3.2 The component based approach

The main idea in the component based approach is to have game entities acting as containers of component objects. This is done by aggregation, meaning that the game entity owns its components (and components cannot be shared). The obvious benefit of this approach is the reduced coupling between classes. There still exists some dependencies between components, but the responsibility of each component is much clearer.

KickJS implements a component-based architecture. As can be seen from the UML figure below, each game object contains a number of components. Each of these components defines a behaviour or property of the `GameObject`. All `GameObjects` has a `Transform` component attached that defines the spatial information (position, rotation and scale). The `Transform` component also has a reference to a parent `Transform` object, which allows the scenegraph to be hierarchical.



UML 1 Scenegraph design

The pros of using a component-based scenegraph are:

- Extensible structure. It is very easy to add a new behaviour to a program using the game engine, simply by creating a new component.
- Clean code with clear responsibility of each class
- The architecture scales well

The cons of a component-based scenegraph are:

- Adds a little overhead compared to a hardcoded structure due to the increased indirectness.
- In some cases, there is too much overhead in representing each entity in its own object. One such case is particle system, which should be implemented by using data oriented design [Llopis09]. Here all particles should be managed by a single component and there is not longer a single object that represents a single particle – instead each particle is represented as a part of an array.

3.3.3 Component and scripting

UML 1 shows that all components inherit from a `Component` class. The `Component` class actually doesn't exist, but is only introduced for documentation purpose. Any object can act as a component and its behaviour is defined by the methods it implements:

- `update()` is called every frame.
- `render()` is called whenever the object are to be rendered
- `activated() / deactivated()` are called when a `GameObject` is loaded. The activated method works in many cases like a constructor function, where references to other game objects can be looked up (this cannot be done when the actual constructor function is called, since the scene may not be loaded completely).

Component objects can also specify other important properties:

- `uid`: Unique identifier (will be added automatically if not exists)
- `scriptPriority`: Specifies the order of script executing. This way some scripts are guaranteed to run before other scripts. This is useful for scripts like camera controllers, which often needs to run after all objects have updated their positions.
- `renderOrder`: determine in what order that the rendering must take place in.

3.3.4 Updating and rendering

On each frame all game objects are updated and rendered.

A naïve algorithm would iterate over the scenegraph from the root of the scenegraph and then for each game-object call the method all of its components. This method works but is slow since all nodes of the scenegraph are visited even though only a subset of the scenegraph is actually needed.

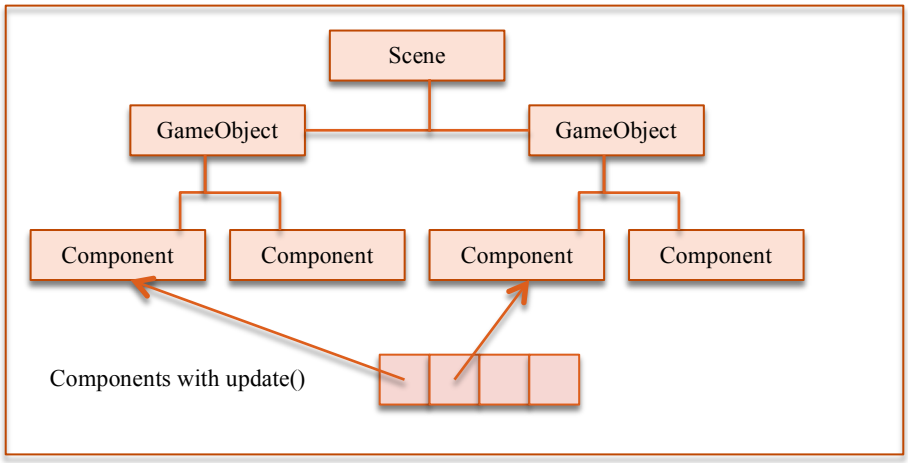


Figure 5 Updateable component list

Instead KickJS maintains lists of components of objects based on the methods of the components. There are multiple advantages of this approach:

- No need to iterate over GameObjects

- The listed are sorted when they are created (using insertion sort). This means that it is easy and cheap to implement features like `scriptPriority` and `renderOrder`.
- The lists only contains the relevant objects

The scene object maintains a list of updateable script components and the camera objects maintains lists of renderable components.

The actual rendering of renderable components is a little more complex:

- In the active scene, the render method of each active camera is called (sorted by camera index)
- When rendering each camera, the cameras list internal list of renderable components are iterated, and the render method on each of the component are called. The list is actually implemented as three lists: one for normal object, one for translucent object and one for overlay objects. The translucent object list is sorted in a back-to-front order based on distance to the camera in each frame. The other two lists are sorted by `renderIndex` on creation.
- When rendering a `MeshRender` component, first the material and its shader are bound and then the mesh object is bound.
- The draw command is finally invoked

To speed up the rendering, objects are only bound if needed. This means if you render two objects with the same material and mesh right after each other, the material and mesh objects are only bound once. For materials this will give a performance gain of around 16%[25].

3.3.5 KickJS Event queue

`Component.update()` work great for code that needs to run every frame. But sometimes you want code that executes only in one frame or within a short time range. Or maybe you need to schedule an event to some point of time in the future.

A JavaScript developer would take a look in his toolbox and find the find the `setTimeout` and `setInterval` functions and use these to trigger code execution. However this approach is not guaranteed to work well with KickJS engine, since the code execution will be triggered outside update-phase of the game loop. Besides it will be impossible to sync `setTimeout/setInterval` to the frame-rate KickJS is running.

[25] See test case in Appendix 7.2.1

KickJS implement its own event queue, where events can be added with a start and an end time (relative to current time). To run code in the next frame, use 0 in start time parameter. It is also possible to cancel an already scheduled event at any given time.

The KickJS event queue is not a replacement for the web browser's event queue. KickJS uses the browser's event queue to get frame-updated events, mouse events and keyboard events. KickJS components should always use the `update` method or the KickJS event queue to ensure that your events are invoked at the right time.

3.3.6 Game loop

To sum up what's happening in a single frame. First the queued events are fired, then the components are updated and finally the scene is rendered.

The frame-rate is kept consistent using the `RequestAnimationFrame API`, which also has the benefit of throttle or lower the frame-rate for background tabs.

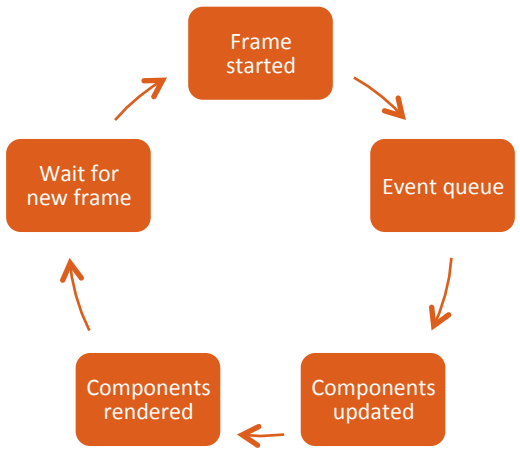


Figure 6 The game loop

3.3.7 Camera class

The camera object gives a view into the game world. KickJS supports multiple active cameras in a scene. Each camera maintains a `layerMask`, which selects only the renderable components where the components gameObject matches the bitmask (each GameObject object has a `layer` property).

The main properties of a camera are the camera projection. Currently two types of projection are supported: perspective and orthogonal. The view-frustum of the perspective camera is defined using the field-of-view Y, the near and the far plane. The view volume of the orthogonal camera is defined by its left, right, top, bottom, near and far planes. Based on these inputs the projection matrix is computed. The matrix is cached and only updated whenever the projection properties of the camera are changed.

A camera also defines a viewport (defined relative to screen size). When setting the viewport, the camera will render to a rectangular part of the screen.

Before a camera is rendered its viewport is cleared, based on the values of the `clearColor`, `clearFlagColor` and `clearFlagDepth`.

The default behaviour of a camera is to render to the default render context (the screen), however it is possible to render to a texture by attaching RenderTexture object as a

render-target. The RenderTexture class is implemented using WebGL's frame buffer objects (FBOs).

Having multiple cameras in a scene combined with layer-masks and render-targets opens up for a long list of render effects and postprocessing effects such as:

- Deferred rendering
- Screenspace ambient occlusion
- Reflections
- Shadows (using shadow maps)
- Picture in picture
- Ping pong textures

WebGL only supports binding one FBO as a color buffer, so multiple color FBOs or depth FBOs are not supported. This does makes some effects such as deferred rendering a bit trickier to implement. One workaround is to use a floating-point texture for the Frame Buffer Object and then pack the values into the 4 floating point channels of the texture.

3.3.8 Light

The engine supports 3 different types of light:

- **Directional light:** Usually used for emulating sunlight where the light rays are parallel.
- **Point light:** Emulates a point light source, such as a light bulb. Point lights supports attenuation by providing a constant, linear and quadric factor[26].
- **Ambient light:** Emulates a constant indirect light, which is light reflected from the environment.

A scene can have up to one directional light and up to one ambient light, but the maximum number of point light depends on the scene settings.

During rendering the lights are transformed into eye-space and send to the shader, if the shader defines the light uniforms. All light sources have a colour property and intensity property. The point light also specifies attenuation.

The light model used is very close to light model used in the fixed-function pipeline of OpenGL including some of the limitations, such as a fixed number of lights.

[26] The attenuation is calculated by $1/(att_{const}+att_{linear}*dist+att_{quad}*dist^2)$. Where `dist` is distance between point light and light source.

3.3.9 WebGL

KickJS uses WebGL for all rendering. WebGL is hosted in a canvas tag in a HTML document. It is possible to mix this canvas tag with other html components (such as text, 2D images and other HTML-components). The typical use-case would be creating 3D games with KickJS, where all menus, heads-on-display (HUD) and other 2D graphics implemented using traditional HTML-components positioned on top of the canvas.

Even though KickJS uses WebGL internally, its main purpose is to abstract the low-level rendering API away by exposing a high-level API for the game programmer. This has the benefit, that programmers can use KickJS without being familiar with the low-level WebGL API, and instead work with abstractions such as `Scenes`, `GameObjects`, `Components`, `Camera`, and `Materials`.

For advanced usage of KickJS, a reference to the WebGL context is still available. Usage of the WebGL context should be done with care, since it can change the WebGL state and disturb the rendering.

3.4 Resource management

The first attempt to create a resource management system in KickJS was to implement a simple reference counting system, where the programmer was responsible for managing reference counts using function calls. While this works great in theory, in practise this approach is too cumbersome to use, and I believe that programmers would tend to forget to release resources.

Instead KickJS uses a much simpler approach: To load all resource in a project on start-up and keep the resources in memory during the lifetime of the project. The benefits of doing this are:

- It is easy to predict the resource usage during the game, since this is constant (except for dynamically allocated resources)
- Resource loading is simplified and it is easy to implement a loading bar that shows the loading progress
- Works well for small for small and medium sized games where the resources can fit in memory.

The problems with this simplistic approach is:

- Only works when all resources can fit in memory.
- No support for streaming levels while you play.

[Gregory08] suggest a slightly more advanced approach, where resources are divided into three categories based on the resource lifetime requirement:

1. Load-and-stay-resident (LSR). Resources are loaded at start-up and kept in memory

2. Resources associated with a scene, such as scene-specific scripts or models.
3. Resources with a shorter timespan than a full level, such as resources used in cut-scenes.

Adding #2 or #3 to KickJS would need a reference counting scheme, to guarantee that resources are loaded and released correctly.

3.4.1 High-level resource loading: Loading project resources

The `Project` object is responsible for loading all resources upon start-up and during the runtime of a game it will keep track of any additional resources created or loaded runtime.

References to resources in a project can be received using the method `load(uid)`. The `Project` object is responsible for caching a reference to all resources to prevent any response is loaded twice.

Finally the `Project` object supports serialization.

Built-in resources are included in all projects (and cannot be removed). They are loaded on demand, but are all very small and fast to load. The build-in resources include shaders, textures and mesh-objects.

3.4.2 Low-level resource loading: URI based resource loading

Resources in a web-browser are usually loaded using a URL. Inspired by this approach KickJS uses an URI based resource loading scheme for loading external resources. The engine supports multiple resource providers and custom resource providers can be added as well.

The resource loader starts up using the following two resource providers:

- **URLResourceProvider**: This resource provider is responsible for downloading resources using http requests. It also works as the fallback resource provider that will be used if no resource provider matches the URI scheme name of the resource. This allows usage of relative URLs.
- **BuiltInResourceProvider**: This resource provider is responsible for loading built-in resources such as basic textures (white, black, grey), shaders, and meshes (cube, UV-sphere, plane)

The resource providers used are found based on the URI scheme name of the resource URI. An example could be “kickjs://mesh/plane/” where the “kickjs:” is the URI scheme name.

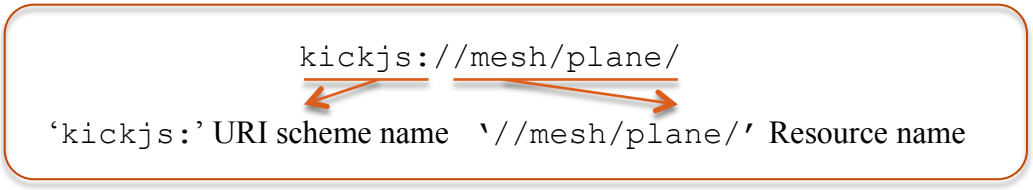


Figure 7 KickJS resource URI

The resource loader will always load requested resources. No caching will be done in the resource provider.

Usually the `Project` object should be used to load resources, but if new instances are needed the resource loader can be used.

3.5 Mesh

The `Mesh` class is responsible for rendering a 3D mesh object. The Mesh class is an abstraction of the WebGL vertex attribute buffers and vertex index buffers.

The `Mesh` class uses interleaved vertex attribute buffers (see Figure 8). This means that the data is organized in the way that the GPU needs it which will result in a faster rendering. Another nice side effect is when the interleaved vertex attribute buffer is used; it is much faster to bind this single buffer than binding a separate buffer for each vertex attribute. The disadvantage of using interleaved vertex attribute buffers is if the CPU needs to update parts of the vertex attribute data (such as the vertex positions only). When using interleaved vertex attribute buffers this is simple not possible – instead a full update of the buffers needs to be done instead. This is usually not a problem, since most interactive updates (such as animations etc.) would usually be performed on the GPU based on uniform input variables.

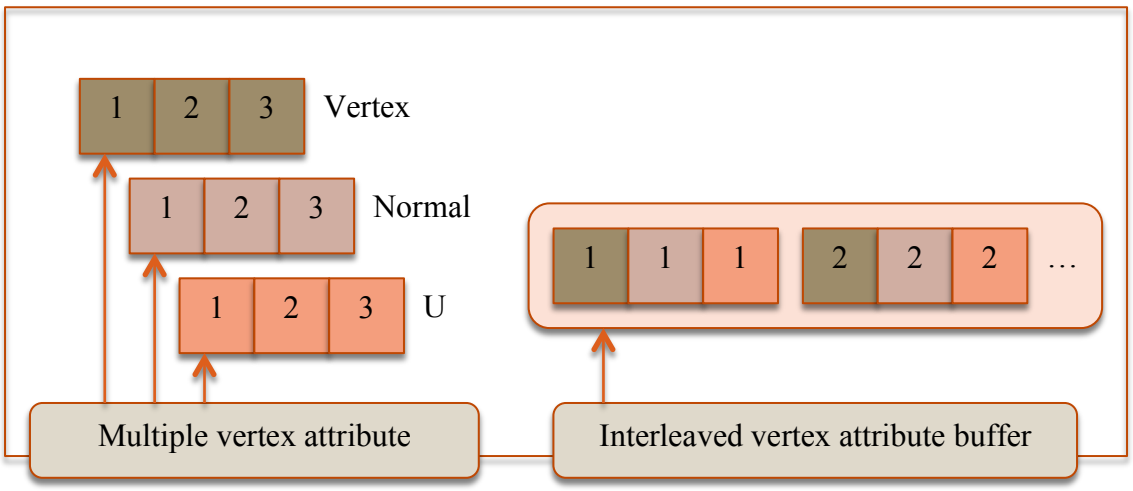


Figure 8 Interleaved vertex attribute buffers

To support 3D geometry with multiple materials, a mesh can have multiple vertex index buffers (one for each material). This feature is referred to as sub-mesh.

The Mesh class is a small class with only a few important methods:

- `bind(shader)`: Binds the mesh to a shader object (match the vertex attributes with the shader vertex attribute locations)
- `render(submeshIndex)`: Render the sub-mesh of the mesh.

The mesh data is encapsulated in a `MeshData` object.

3.5.1 MeshRenderer

The `MeshRenderer` class is a `Component` that references both a `Mesh` object and `Material` objects. The main reason why a `Mesh` is not a `Component` is to support `Mesh` objects to be shared across several components (note that components cannot be shared – they only have one owning `GameObject`).

`MeshRenderer` uses multiple materials to support different materials for each sub-mesh.

The `MeshRenderer` component implements the render method that binds the materials and mesh objects and renders the mesh object.

3.5.2 MeshData

The `MeshData` objects contains the geometry and other data that the `Mesh` object uses. WebGL does not give semantic meaning to vertex attributes – they are only values sent to the vertex shader. KickJS on the other hand do add semantic value to certain vertex attribute names. This simplifies binding of meshes to shaders (but do require that shaders uses the predefined names).

The following lists the vertex attributes supported by the `MeshData` object.

Vertex attribute name (in MeshData and in shader)	Data type	Description
vertex	Float Vec3	The vertex position. This will be converted to homogeneous coordinates (x,y,z,1.0)
normal	Float Vec3	The normal to the vertex
tangent	Float Vec4	The tangent to the vertex

Vertex attribute name (in MeshData and in shader)	Data type	Description
uv1	Float Vec2	The primary texture coordinate
uv2	Float Vec2	The secondary texture coordinate
int1 .. int4	Integer Vec1-4	Generic integer fields of size (1,2,3,4)
color	Float Vec4	Vertex colour

Table 2 Vertex attribute names

Using this fixed structure makes it much easier to create and reuse shaders across applications, since the mapping of vertex attributes is straightforward.

It is important to note that only the vertex attributes used by the shaders needs to be specified. Another important thing is that when developing custom shaders, you can use the vertex attributes to whatever you want (such as using the colour attribute to store temperature instead).

Sometimes mesh data may be provided without normal and tangent information. For situations where the normal or tangent information is needed but not provided the class has two methods for re-computing normal and tangent information based on the geometry.

3.5.3 3D model importers

KickJS can import two widely used 3D formats:

- **Wavefront obj**: This is one of the most simple and widespread formats for 3D models. It is a text based data format. KickJS supports loading polygon models from an OBJ file including texture coordinates and normal. OBJ may also contain lines and NURBS-surfaces; these are not imported by KickJS.
- **Collada (dae-files)**: Collada is an XML-based format for interchanging 3D data between applications. Collada is an open format originally created by Sony and is now maintained by the Khronos group. The Collada format is huge and complex, so KickJS only supports the most commonly used features for 3D models, which includes triangle meshes, texture coordinates, normal, tangent, and scene hierarchy.

External models can be imported from a Wavefront obj-file or a Collada dae-file at any time in a game. However it is encouraged to convert the model to binary `MeshData` when building the game. The binary `MeshData` has the same memory layout as used on the GPU, which make them very fast to load.

3.6 Serialization

In KickJS serialization is used in multiple parts of the engine. The most important parts are serialization of project assets, scene object and mesh-data.

The serialization is used heavily be the KickJS Editor when loading and saving projects. The KickJS Editor will be described in chapter 4.

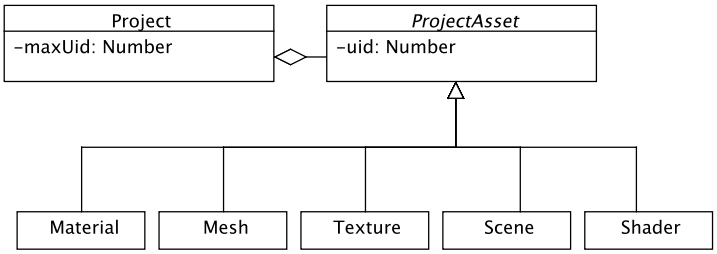
3.6.1 Unique identifier

Objects in JavaScript are uniquely identified by their reference which is essentially is an encapsulated memory pointer. This works well when working with runtime objects, but when serializing a JavaScript object you will need to be able to identify objects uniquely to be able to restore object relations correctly when deserializing.

The KickJS engine does this by assigning a unique id (`uid`) to all assets, game objects and components. The unique id generator is a number starting from 1 and increased with one every time a new id is needed.

When serializing a project the maximum unique-id is also saved to the serialization and restored upon deserialization.

3.6.2 Serialization of project assets



UML 2 Usage of uids in project

The `Project` class maintains assets used in a project. This class contains a list of asset descriptions, which includes the type of asset (class name), a `uid` and a configuration of the asset. All assets should have a constructor that takes two parameters: a reference to the KickJS engine and a JSON configuration, which is used when deserializing the objects. An asset object should also have a `toJSON` method that returns a configuration object in JSON format.

Serialization and deserialization of project assets is straight forward, since each asset is self-contained and does not reference other project assets (with the exception of the `Scene` asset discussed below).

3.6.3 Serialization of scene objects

The `Scene` object contains all `GameObjects` on the scene including all `Components` for each `GameObjects`. The `GameObjects` of a scene is just a list, but the components used in a scene can reference other `Assets`, `GameObjects`, or `Components`. This gives a graph structure, where a little more care needs to be taken when serializing and deserializing.

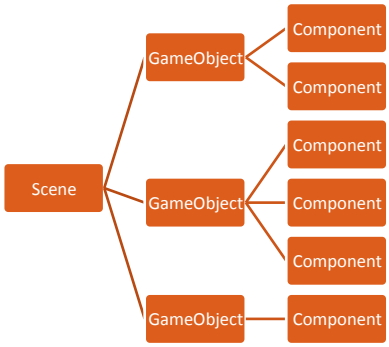


Figure 9 Scene structure

Serialization of `Scene` objects is also done to JSON formatted objects. When serializing the components, all references to other `Assets`, `GameObjects` or `Components`, the reference is replaced with a reference object that contain the type and uid.

Deserializing the scene happens in two phases:

1. `GameObjects` and `Components` are deserialized. But the configurations of the `Components` are not applied to the `Component` objects, since they may contain references to objects not initialized yet.
2. When all `GameObjects` and `Components` are constructed the configuration of all `Components` are applied. This includes replacing any reference object with reference to the actual object.

Serialization of user-defined objects is also supported. The object then needs to expose all its properties and to have a `toJSON` method.

3.6.4 Serialization of MeshData

Serialization of `Project` and `Scene` both used JSON. This made sense since the data was very hierarchical and contained a lot of structural information.

`MeshData` on the other hand consist mainly of arrays with floating point numbers and arrays with integer numbers. For this reason KickJS uses `ArrayBuffers` and `ArrayBufferViews` to store `MeshData`[27].

To serialize the `MeshData` into binary data a chunk-based scheme is used. The chunk functionality is encapsulated in the class `ChunkData`.

---

[27] `ArrayBuffers` and `ArrayBufferViews` (such as `Uint8Array` and `Float32Array`), also known as Typed Arrays, are a new way to work with binary data and data types in JavaScript.

The chunk layout is as follows:

Chunk Data (container of chunks)

Magic Number Uint16	Version number Uint16	Number of chunks Uint32	Chunk data N bytes

Chunk

Chunk id Uint16	Chunk type Uint16	Chunk data length Uint32	Chunk data N bytes	Chunk padding 0-7 bytes

Table 3 Data layout of chunk data and chunk

The data types supported in a chunk are all `String`, `Number`, and `ArrayBufferViews` (`Uint8Array`, `Float32Array`, etc.). `String` and `Number` chunks are implemented with simple wrapper functions that convert strings into UTF8 encoded arrays and numbers into a `Float64Array`.

Padding is added to arrays so that data is always aligned with 8 bytes. This is required for `ArrayBufferViews` to work in all cases.

A benefit of storing mesh data is faster loading, since data is stored the same way on disk and in memory. Another benefit is the size of the data. By using the binary KickJS format over JSON, the file-size is reduced with approximately 50%[28].

3.7 Materials and Shaders

In KickJS the `Shader` object is responsible for loading and compiling shaders.

KickJS add a few extensions of GLSL shaders:

- `#pragma include <filename>`. Allow you to include KickJS shader functionality (such as lights)
- Auto mapping of predefined keywords: A lot of keywords are used in several shaders, such as normal-matrix, model-view-matrix, etc. The engine will set these keywords automatically during rendering. The full list is described in Table 2 page 40.
- Defines the GLSL constant `LIGHTS` (Integer) based on the current configuration of the engine (cannot be modified runtime). The constant defines the maximum number of point lights.

[28] See chapter 7.2.4 in Appendix

Besides encapsulating the vertex shader and the fragment shader, the `Shader` class is also responsible for changing the WebGL states that relates to rendering, such as blending, Z-test and face culling.

After a shader has been compiled and linked, the shader program is automatically inspected and the available uniform locations and vertex attribute locations are saved in the shader. These values are used when the shaders are bound.

The `Shader` class also has a `renderOrder` attribute, which makes it possible to define in what orders the shaders should be rendered.

Finally the `Shader` can also contain a default configuration, which makes it easy to bind materials in cases where the material configuration doesn't match the shader.

Shaders can be created using the KickJS Shader editor, which will be discussed in chapter 4.6.2.

3.7.1 Materials

Materials are instances of shaders, providing values for the uniform variables of the shaders. The values are stored in the uniform object and the name of the value should match the uniform name in the shader.

Materials supports tree types of uniforms:

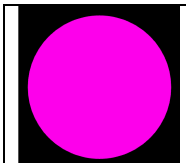
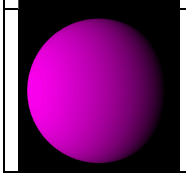
- **Floating point values:** Vec2, vec3, vec3, mat2, mat3, mat4, float
- **Integer values:** Vec2i, vec3i, vec4i, mat2i, mat3i, mat4i, integer, boolean (integer with 0 or 1 value)
- **Textures:** 2D textures and cube map textures

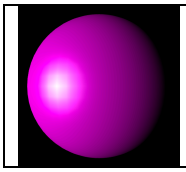
Shaders are usually always used together with a material.

3.7.2 Built-in shaders

Shader development is very low-level and requires a deep understanding of render pipeline, light equations and 3D math.

To ease development KickJS includes the following predefined shaders:

	**Unlit**: The colour is purely based on the specified colour and texture.
	**Diffuse**: The colour is based on the specified colour and the texture and lighting based on Lambert's light equation. This should be used for materials with diffuse reflections.

	**Specular**: The colour is based on the specified colour and the texture and lighting based on the Phong light equation. The specular highlight can be adjusted in colour and in size. This should be used for reflective materials, such as cars, eyes and wet surfaces.

These shaders all exist in both an opaque and a transparent version.

The engine also uses a number of internal shaders. The internal shaders are used for picking and for fallback shader (error shader) when another shader is unable to compile.

3.8 Input management

To create interactive applications such as games, you need input from the user. KickJS supports two kinds of input: keyboard input and mouse input.

Both input managers are only initialized if it is used. This means that if a game only uses mouse input the KickJS engine will not listen for key events.

3.8.1 Keyboard input

KickJS implements a state based keyboard input, where the browser DOM only provides event based keyboard input. The reason why a state based approach is used is that keyboard input in games is often interested in knowing whether a key is being pressed.

KickJS implements keyboard input the following way. The `KeyInput` object keeps internally three lists:

- `keyDown`: A list of keys that has been pressed this frame
- `keyUp`: A list of keys that has been released this frame
- `key`: A list of keys that is being pressed

The object also contains methods for getting the state for a particular key. The internal lists are updated each frame:

- KickJS listens for key events on the document.
  - On key down: The key-code is added to the `keyDown` list and the key list
  - On key up: The key-code is added to the `keyUp` list and removed from the key list
- During the update phase of the components, the scripts can query the object for key states.
- After the update phase the `keyDown` and `keyUp` list are cleared.

This approach also guarantees that script code dealing with key input runs during the update phase, where using the event-based approach the scripts would be run outside the update phase while making the programming more predictable.

3.8.2 Mouse input

The mouse input is state based and implemented the same way as `KeyInput` object.

The mouse input object gives access to information about movement (delta position), position, button status and scroll wheel status (horizontally and vertically). The mouse position is relative to the top left corner of the WebGL canvas element.

One of the upcoming features in new web browsers is the ability to lock mouse (with the Mouse Lock API). This feature can lock the mouse position, while still sending mouse events to JavaScript. This makes it easy to implement games like first person shooters, without risking that the user moves and click outside the game. The Mouse Lock API is currently only available in the nightly builds of Chrome and Firefox, and for this reason it has not yet been implemented in KickJS.

3.8.3 Picking

One commonly used feature when working with 3D scenes is the ability to pick objects in the scene with the mouse.

In KickJS I have implemented a simple picking functionality that works the following way:

- A script invokes the pick method on the `Camera` object with the position and a callback function. This method call is added to a picking queue.
- On camera rendering if the picking queue is not empty, the scene will be rendered to a picking texture (using the `RenderTexture` class) with a replacement shader that writes `GameObject` unique-id packed into the RGBA values of the texture.
- For each pick the pixels from the picking texture are sampled and the RGBA value are converted to a uid.
- Each object found in picking is added to the KickJS event queue and will call the callback function when executed.

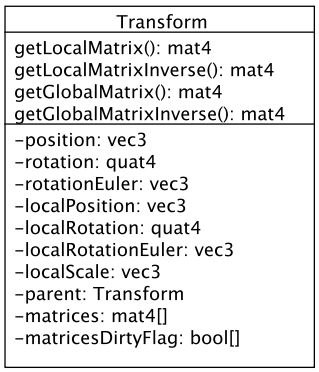
The normal use case for picking is to select objects with mouse, however the technique can also be used for visibility tests.

Based on the collision point the KickJS engine could be extended to return the world-space position, the UV coordinates or other information available in the shader. For this to work, you would have to add another render pass that save the data to a texture and then sample that texture[29].

[29] WebGL only supports binding one colour texture to Frame Buffer Objects. In OpenGL you would bind a texture to the depth buffer of the Frame Buffer Object, and only need one render pass.

3.9 Math library and the Transform object

3D math such as matrices, vectors and quaternions are essential for any 3D game engine. The math library used in KickJS is based on glMatrix developed by Brandon Jones[30]. The math library is optimized for speed and carefully avoids allocating objects.



UML 3 Transform class

One of the main usages of the math library is in the `Transform` component. `Transform` objects represents the spatial position of an object in the scene.

The transform is always represented using local-position, local-rotation and local-scale. Internally the rotation is stored as a quaternion. For convenience a rotation property using Euler's angle is also exposed and will translate between the two representations. A matrix representing the transform can be obtained using the method `getLocalMatrix()`. To speed things up the transform matrix is cached when computed and will only be recomputed if the position, rotation or scale is changed. This is being tracked using a dirty flag. The same method is used for computing the inverse of the local transform.

The transform object may have a parent object, which makes its transformation relative to its parent. For convenience the object exposes two properties for reading and writing the global position and rotation. Note that there is not global scale, since this may be non-linear due to parent rotation. The object also exposes two methods for reading the global matrix and its inverse. Global matrices are cached the same way as local matrices, except that dirty flag is set when its local matrix is dirty or any parent is dirty.

If no parent exist the global methods and properties is the same as their local counterparts.

3.10 Future improvements

KickJS is still a young game engine and a lot of more advanced topics can be added to the engine.

---

[30] Available under a BSD-like license here: https://github.com/toji/gl-matrix

3.10.1 Sound

Sound is one of the cornerstones in any game. Unfortunately sound support in a browser is still in the works. There exists a number of sound APIs (such as the audio-tag and the Web Audio API), but not all major browsers do support them. A related problem is the mixed support for audio file format. Currently there doesn't exist one file-format supported by all major browsers. Finally the audio support the browsers have today is mainly focused around playing music or sound clips. To add audio support to a game, you would need audio to play instantly, multiple times, simultaneously and possible with some effect that can be tweaked by the game engine. Many HTML5 games today use the Flash plugin as a workaround to get good audio support[31].

3.10.2 Physics engine

Today the majority of the 3D games use physics engines as a part of the game engines. A physics engines is responsible for two important things:

- Collision detection: The ability to know when two colliders intersect (such as a mesh or a simplified physics placeholder)
- Rigid body dynamics: How rigid body move and intersect over time under influence of forces[32].

3.10.3 Animation systems

Animation systems in 3D game engines usually includes one or more of the following sub systems:

- **Rigid hierarchical animations** used for animating mechanical things such as robots, cars and doors.
- **Per-vertex animations** where each vertex position is animated by an artist and the engine interpolates between these positions
- **Skinned animations** where the animation is driven by a skeleton constructed by rigid bones. The artist will animate the skeleton and the engine animates the position of the skin (the mesh).

3.10.4 Deferred rendering

KickJS uses forward rendering, meaning that when rendering an object the result are written to the frame buffer as colours. This is the fastest and most simple approach for rendering 3D graphics in the general case.

In deferred rendering, you would render the information about the object to different channels of an off-screen texture. This information includes colour (also known as

---

[31] Mentioned in [Webber11]
[32] [Gregory09] page 595

albedo), normal in screen-space, emission, specular values, depth, etc. These off-screen textures are known as the G-buffer. The final rendered images is then composed using one or more render passes that uses the G-buffer. The benefits of using this technique is that you now have information about the full rendered scene, which makes it easy to add post-processing effects like ambient occlusion, edge detection, field-of-view and bloom.

One problem with the WebGL compared to OpenGL, is that WebGL does not support multiple render targets. This means that in order to render the G-buffer; you would have to render the scene multiple times, one for each texture of the G-Buffer. Alternative it is possible to pack the G-buffer into a floating-point texture.

Currently KickJS does support multiple renderings (using multiple cameras) as well as render targets, which writes to an off-screen texture. This means that deferred render already is possible in KickJS if the programmer writes the deferred rendering shaders. But to make deferred rendering easy to use the engine would need to be restructured considerably and should bundle the needed shaders the same way as the Phong shaders are bundled today.

3.10.5 Static geometry batching

Scenes in a typical game often consist of a large number of static geometric objects (such as walls, buildings, etc.) and a smaller number of movable geometric objects (cars, animals and humans).

A large number of geometric objects can result in poor performance due to the increased number of draw-calls. To reduce the number of draw-calls static geometry can be batched together in one draw-call if the following conditions are met:

- Must have same shader
- Must use same material

The geometry batching optimization should try to batch only objects close to each other to make view volume culling behave well.

The actual batching will transform the vertex position and vertex normals into world space and then concatenated together. If different textures are used, the batching optimizer can also combine these textures into texture atlases, by combining the textures and updating the texture coordinates accordingly[33].

Static batching is usually performed in a pre-processor step when building the game.

[33] [Pharr05] has a discussion about different batching strategies in ”Chapter 3: Inside Geometry Instancing”

KickJS would benefit a lot from adding static batching. The `MeshData` class already contains methods for combining geometry, but ideally the static batching should work out-of-the-box.

3.11 Summary

In this chapter I have described the implementation details of the KickJS engine that I have implemented during my thesis.

The KickJS is a component based game engine, which makes it easy to extend and customize to any game. The engine uses mesh optimized for the memory layout of the GPU, and the engine provides both a low level and a high level access to shaders, to ease development while still supporting advanced custom shaders.

4 KickJS Editor – A scene editor for KickJS

In this chapter I will discuss my implementation of a scene editor for KickJS. The editor is available online at the following website:

http://editor.kickjs.org/

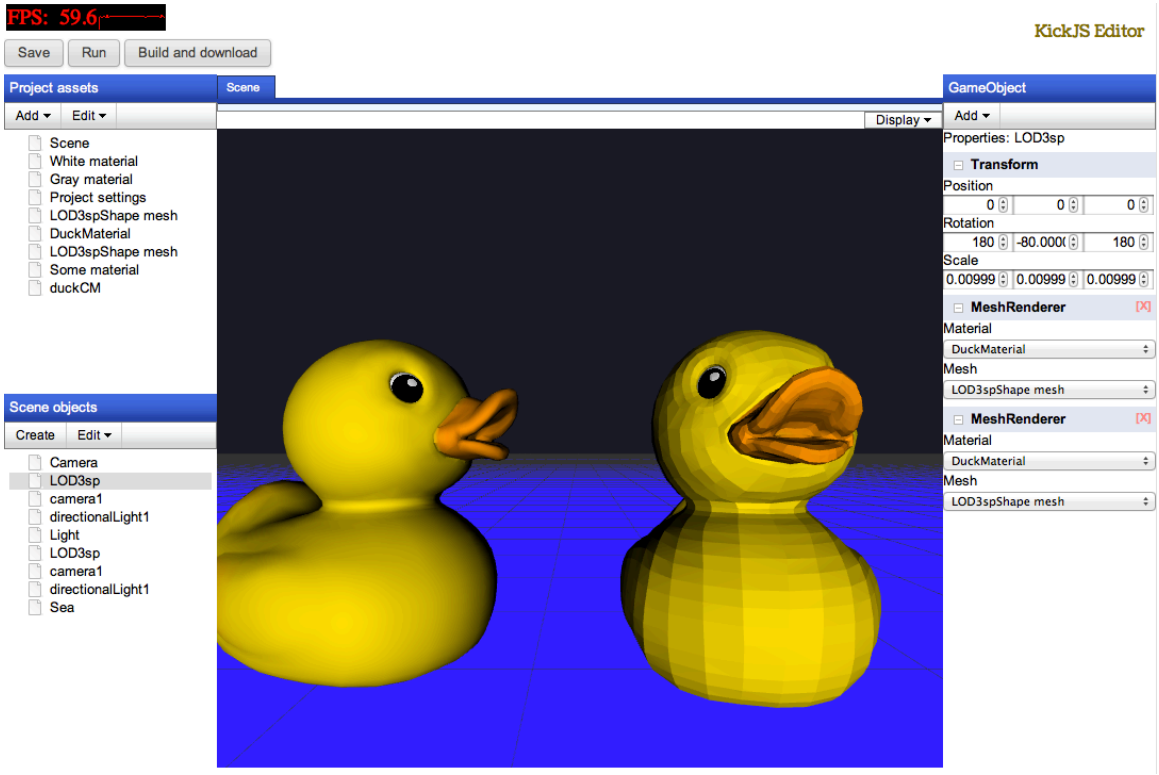


Image 4 Screenshot of the scene editor

The motivation for building a scene editor is to make scene composition an intuitive and visual task, instead of needing to hardcode the scene in JavaScript. This way an artist or a level designer can build the scene without involvement of programmers.

The KickJS editor also allows you create small WebGL applications without any coding involved. This can be used as a web for artists for publishing their 3D models to webpages.

The editor is creating scenes that run in the KickJS game engine, but the editor itself also uses the KickJS for all its rendering, scene navigation and object picking. The GUI elements of the editor is created using YUI 3.5[34].

[34] http://yuilibrary.com/ YUI an open source JavaScript and CSS framework for building web applications (Open source under BSD license)

The KickJS editor is in its current state more a proof-of-concept than a full-fledged game development environment.

4.1 Usability

The scene editor has the following UI elements:

- **Main buttons**: Save, Run and Build Project. These buttons exposes the main functionality of the application.
- **Content window**: This tabbed pane shows the main content of the application. The most important content is the scene view that let you look into the scene.
- **Project assets**: List the current assets used in the project (excluding any built-in asset). On top of the project list is a menu bar, with access to modifying assets or importing new assets.
- **Scene objects**: Lists the game objects in the current scene. Over the list is a menu that let you create, rename or delete any game object. The content of the scene objects will change dependent of which scene is active.
- **Property editor**: This shows the components of the selected game objects or the properties of the selected asset.

4.2 Scene view

The scene view let you navigate through the scene and let you construct the scene.

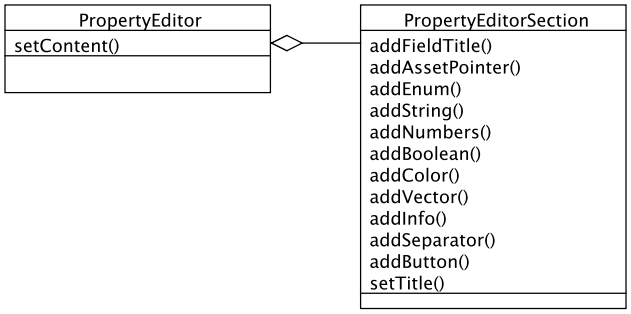
The way the scene view is implemented is loading the existing scene and decorating it. The following objects are added to the scene:

- **A virtual camera** including a camera controller, which allows you to navigate the scene using the mouse.
- **A grid mesh** object that visually shows the ground level.
- **A component selection listener** that listens for mouse clicks and uses picking to see if any object is selected.

These objects are filtered away when the scene is build and exported.

4.3 Property editor

The property editor is responsible for displaying the properties of an object. For project assets the current properties are displayed, and for `GameObjects` the components and the properties of the components are displayed.



UML 4 Property editor GUI

The property editor GUI component can contain one or more sections. Each section is either a `Component` object or a `ProjectAsset`. The sections are built using the `addXXX`-methods of the `PropertyEditorSection` object.

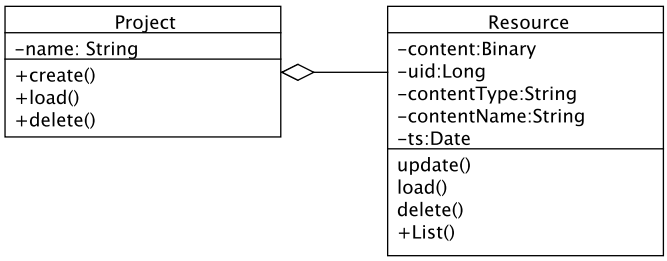
To find out how each component or asset should be built, one of the following methods are used:

- If the object has a `createEditorGUI` method, this method is invoked and is responsible for layout of the component. Such methods for most KickJS objects are defined in the editor.
- Otherwise the component is created by inspecting the objects properties. Since JavaScript is a dynamically typed language it is not always possible to find the correct GUI component.

Changes to the object will be reflected to the actual object either instantly or when focus is lost.

4.4 Persistence

To support persistent data, the following model is used for storing data:



UML 5 Design of the persistency model for KickJS Editor

This model is related to how objects are serialized in the KickJS engine. The project is serialized to the resource with id 0, and all other assets (such as textures, mesh, shaders and scripts) are serialized to a resource based on the objects unique id. The project works as meta-data describing all assets in the project.

The actual storage of the model is either server based or using the browser's local storage.

4.4.1 Server based persistence

The server-based persistence has been implemented using the Google App Engines Datastore. Data are updated using asynchronous HTTP POST-request from JavaScript, and downloaded using normal GET-requests.

The Google App Engines Datastore uses a distributed architecture very different from a relational database. It stores entities each with a set of properties, very similar to JavaScript objects. The database schema is modelled based on the storage model described above.

Due to the cost of using the Google App Engines Datastore, server-based persistence is not enabled per default. Instead local storage is the default storage option.

4.4.2 Local storage persistence

Local storage is a new browser feature that allows web applications to store data locally in the web-browser. Currently there is three APIs for persistent storage:

- Web SQL: An SQLite database instance that runs in the browser. Not supported by Firefox.
- IndexedDB: A simple database API for storing large amount of data. Supported by both Firefox and Chrome.
- Web Storage: Key-value storage for storing simple data. Supported by all new browsers.

KickJS editor uses Web Storage for storing projects names and IndexedDB for storing resources.

Binary data are packed into number arrays, since IndexedDB does not support binary data (or binary Blobs) in current browser versions.

Local storage can store up to 5 MB, but can be expanded if the application requests it and the user accepts the request.

4.5 Build and download

A game written in KickJS consists of the following types of files:

- **HTML:** A web page hosting the game.
- **JavaScript:** The KickJS library, some source code for starting the game as well as game scripts
- **Textures:** Image files in formats that browsers support
- **Shader source code:** For custom shaders not bundled with the engine

- **3D models:** Models in the binary KickJS format

KickJS applications must be executed from a webserver. For convenience a tiny Java based web-server is included in the build[35].

In the build process the project description is modified to fetch assets from relative URLs. Then a zip archive is created with all the resources and finally the browser is asked to download the archive.

All these steps happen inside the browser – not on the server as one might expect. The zip-archive is generated using the JavaScript library JSZip[36] and the browser download is using the Flash plugin Downloadify[37] (downloading client generated data is not well supported by Firefox).

4.6 Future improvements

In this chapter I will discuss features that could be implemented in future versions of the KickJS Editor.

4.6.1 JavaScript editor

The next thing that will be implemented in the KickJS Editor will be a code editor. This will allow users to create full WebGL games inside the browser.

The code editor would be able to edit new script components saved as assets in the project view. These script components can then be added to `GameObjects` in the scene.

When building the game, the scripts will be bundled with the rest of the code.

The actual code editor will use the Ace (a web based code editor).

4.6.2 Shader editor

One of the example applications implemented in KickJS is a GLSL Shader Editor.

[35] https://github.com/mortennobel/SimpleWebServer
[36] http://jszip.stuartk.co.uk/
[37] https://github.com/dcneiner/downloadify.

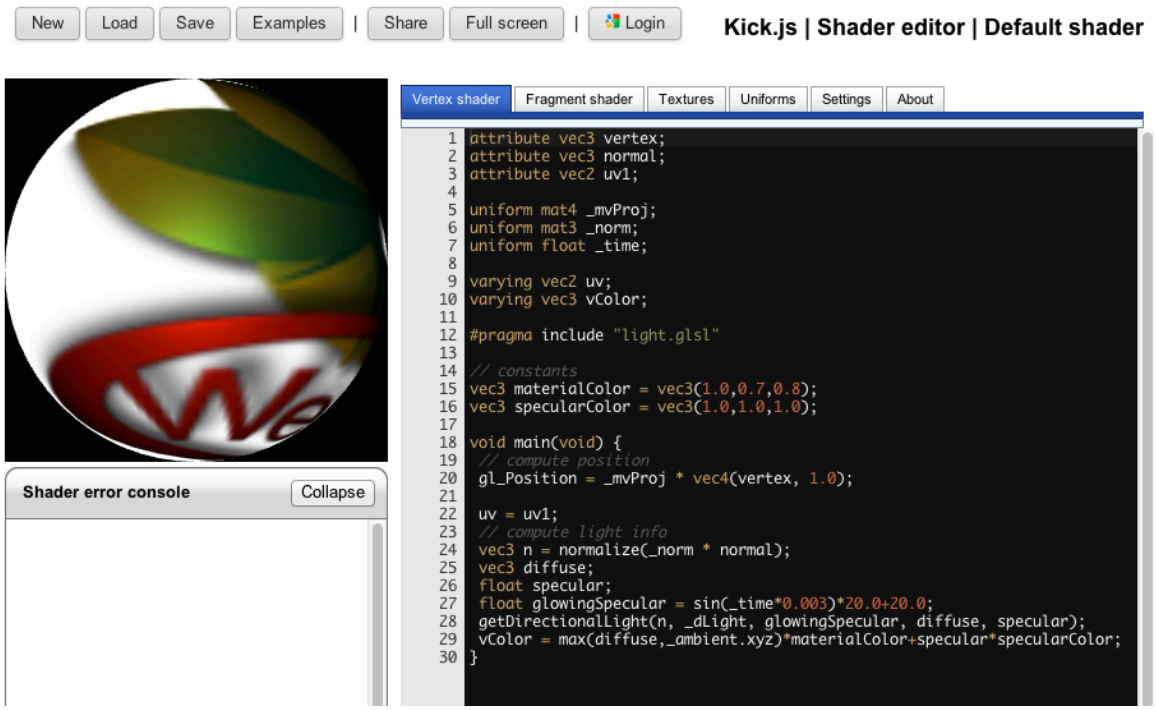


Image 5 KickJS shader editor

The KickJS Shader Editor is an interactive GLSL editor that let you see the result while you write the shader. The editor supports the built-in uniforms from KickJS and a simple property editor for changing uniforms. Under settings it is possible to change scene properties, such as geometry, perspective, and light.

The KickJS Shader Editor is not yet a part the of KickJS editor. The shader editor was developed to help development of internal shaders in KickJS and is targeted advanced users, whereas the KickJS editor currently targets beginners and intermediate users. For users who want's to create custom shaders for KickJS, the shader editor will still be a great help.

The editor is build using the Ace – a web based code editor[38].

4.6.3 Node based editor

One commonly used approach in the game industry is to use node editors to replace programming. This has two benefits:

- Game designers with no programming skills can do simple scripting
- Node editors can in some cases give a better overview over a solution and how parts of the solution work together.

[38] http://ace.ajax.org/ - Open Source under MPL/LGPL/GPL

Two examples of node editors that I find working particular well is from the Unreal Development Kit:

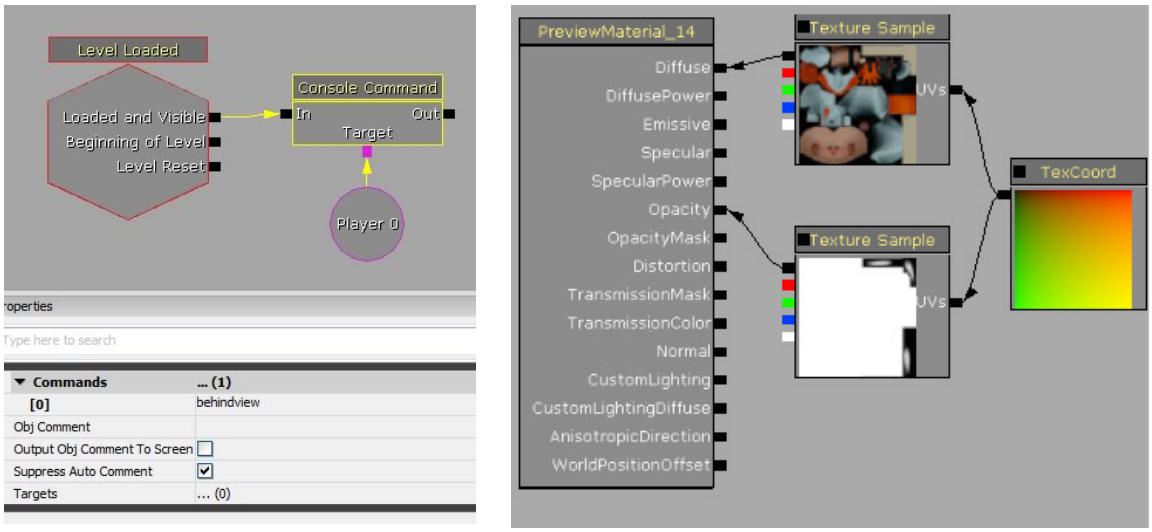


Image 6 UDK's Kismet Script Editor (left) and material editor (right)

4.6.4 Collaborative game development

It would be obvious to extend KickJS Editor to support multiple people working on the same game simultaneously, but even through both web-browsers and its JavaScript APIs are created with distributed system in mind, it is still not trivial to create such system. You need to take care of distributed computing problems such as concurrency, unreliable network connections and scalability.

For this reason collaborative game development is not implemented in the KickJS Editor.

4.6.5 Building other than WebGL games

When a game engine provides a well-defined API it is easy to port games to other platforms. It should not be hard to use a JavaScript engine like NodeJS to port the engine runtime to other platforms that will not have WebGL support in near future (such as Windows Phone). Once the engine runtime has been ported any game created in the engine should be able to run.

4.7 Summary

The KickJS Editor is a tool for creating scenes to use in KickJS applications and games. But at the same time the editor is an example of how KickJS can be used for creating advanced 3D applications.

The editor uses the new HTML5 APIs for implementing client-side persistence and uses the YUI 3 for building the advanced GUI elements of the editor.

5 Benchmark

In this chapter I will give an estimate of JavaScript and WebGL performance in general. The estimate will be based on a benchmark on JavaScript vs. C++ and a benchmark on KickJS vs. Unity.

All benchmarks performed in this chapter are micro benchmarks. They should give a hint on the general performance, but micro benchmarks may not uncover problems of large applications and games.

One of the trickier areas to benchmark is memory-access patterns. In C++ the programmer can optimize their data for efficient processing using data oriented design – see [Llopis09] for a further discussion. This is much more difficult in JavaScript since the programmer has no control over how the heap is organized[39]. I will not benchmark memory access patterns directly, but they are likely to have influence on test results.

Unless other is specified the benchmarks is run on a 2.2 GHz Intel Core i7 with 8 GB 1333 MHz DDR3 and a AMD Radeon HD 6750M 1024 MB. All tests are preformed using Mac OS X 10.7.3.

5.1 JavaScript vs. C++

This chapter tries to answer what performance can be achieved running JavaScript compared to C++ code.

5.1.1 The Computer Language Benchmarks Game

The Computer Language Benchmarks Game[40] is a website which tries to compare several languages to find out the runtime characteristics on different programming languages[41].

http://shootout.alioth.debian.org/

The websites compares a number of different micro benchmarks implemented in different languages. The benchmarks problems are typical computer science problems ranging from binary trees to Mandelbrot computations.

It is important to realize that the programs may not be evenly optimized in the different languages. The benchmarks are performed on an Intel Q6600 single core CPU running Ubuntu.

---

[39] To a certain degree this is still possible in JavaScript by using large ArrayBuffers to store objects.
[40] Also known as ”The Great Computer Language Shootout”
[41] Note that you cannot benchmark programming languages or programming language implementations, only programs written in different programming languages. This adds a certain uncertainty to the test results.

The computations in C++ uses either double precision floats or integers, which maps well to the JavaScript Number and the JavaScript Engine SMI type (Small Integer – 31 bit unsigned integer).

Test result:

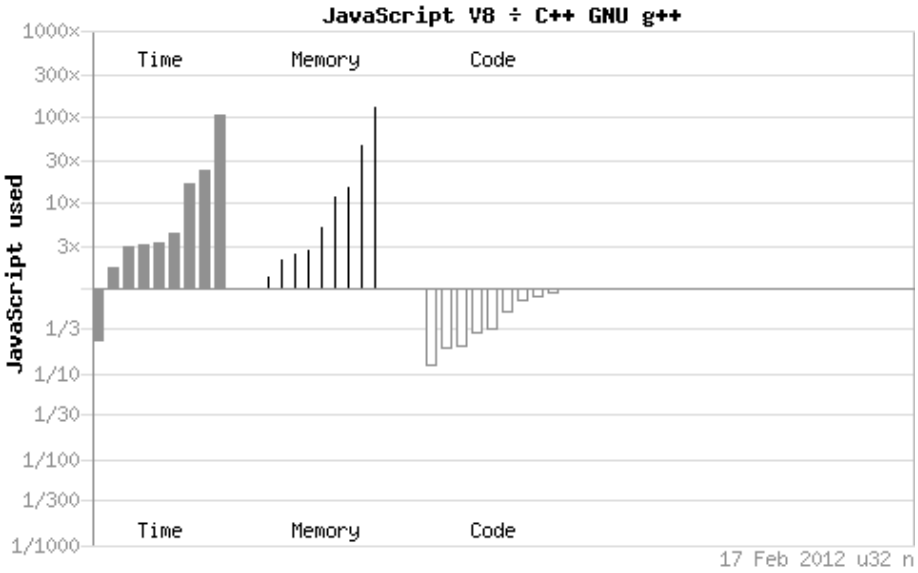


Image 7 JavaScript (V8) compared to GCC C++ (See 7.2.1 for more details)

Evaluation of test result

In average the JavaScript spend 3 times as long on solved the same problem. The memory is likewise around three times higher.

The one case where JavaScript beats C++ performance is a test case with regular expression matching and replacement, which is a native feature of JavaScript and hence very optimized.

At the other end of the spectrum, where JavaScript performs up to100 times slower, I believe that the main reason is that the JavaScript source code it not optimized as much as the C++ code.

It is interesting to note the correspondence between memory and time. This could indicate that one reason that JavaScript has lower performance is due to inefficient memory usage.

5.1.2 Math library benchmark

When working with 3D graphics one of the most important features is a high performance math library.

In this benchmark I will compare the performance between two widely used math libraries:

- **glMatrix**: High performance JavaScript matrix and vector operations for WebGL created by Brandon Jones. It is the same library that is used in KickJS.
- **GLM**: OpenGL Mathematics (GLM) is a C++ mathematics library for graphics software.

I have written a benchmark based on a WebGL Matrix Benchmark and implemented the same tests in C++. The benchmark suite is available here:

https://github.com/mortennobel/glMatrix-vs-GLM-Benchmark

The benchmark uses both single and double precision floats.

Test results:

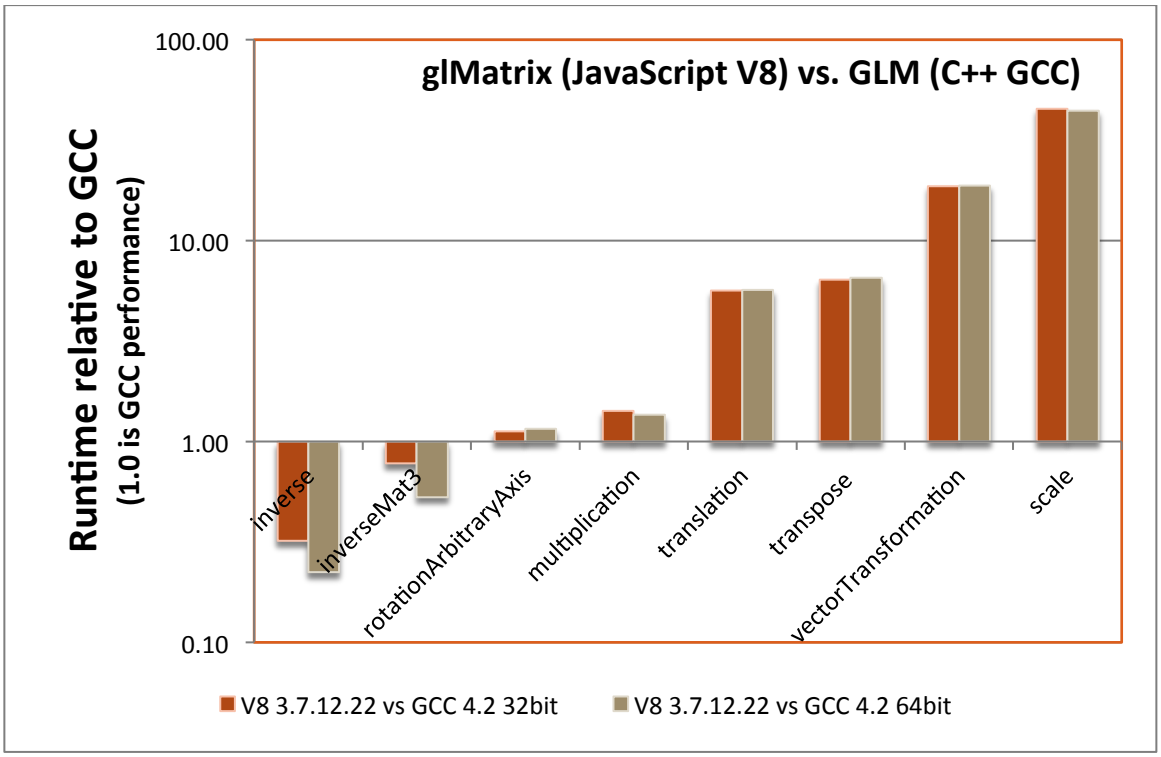


Image 8 glMatrix (JavaScript V8) vs. GLM (GCC C++)

	GCC 4.2 32 bit float	GCC 4.2 64 bit float	V8 3.7.12.22
inverse	197.7	284.6	63.5
inverseMat3	97.5	143.6	75.9
rotationArbitraryAxis	137.1	133.4	154.2
multiplication	83.7	87.9	119.1
translation	8.7	8.6	48.9
transpose	10.0	9.7	63.7
vectorTransformation	12.0	11.9	224.3

scale	3.2	3.3	145.9

Table 4 Runtime in milliseconds of 2.000.000 operations

Evaluation of test result

In general the test result shows that glMatrix math library runs approximately 10 times slower than the GLM library.

One noteworthy observation is that glMatrix actually is faster for computing the inverse. One reason for this may be that inverse matrix methods in glMatrix supports calculating the result in-place, whereas the GLM functions returns an inverse matrix.

For vector transformations and scale, the GML performs a lot faster (18x to 45x times). I believe the reason for this is that GLM uses SIMD optimizations, and for this reason can gain a performance boost.

5.2 KickJS vs. Unity

In this chapter I will compare the KickJS game engine with the Unity game engine. The Unity game engine is a commercial game engine with a lot of advance features. The comparison between the two engines will be only of the core features and should give an answer to what capabilities WebGL powered games have today and what we can expect from the future from this technology. The goal of the comparison is not to find be best of the two engines, this would be an uninteresting comparison where Unity would come out as a clear winner.

The comparison is done by first listing features of the two engines side by side to get an overview of the two technologies. Then a number of benchmarks will be evaluated to highlight the performance characteristics of the two engines.

The tests are run using KickJS 0.3.1 running in Chrome 17.0.963.56 and Unity 3.4.2 running inside its editor.

5.2.1 Side by side comparison

Listed below is a very rough overview of some of the most basic features of the two engines.

	KickJS (WebGL)	Unity
Platform	Runs in browsers with WebGL support. This includes PCs, and few mobile phones and tables.	Runs in PC browsers, on Mac and Windows computers, on iOS/ Android devices, and on game consoles (Wii, X-Box 360 and PlayStation 3).
Graphics API	Runs on top of WebGL, which underneath may be powered by OpenGL, OpenGL ES or DirectX (using ANGLE).	Unity uses an abstraction layer that supports OpenGL, OpenGL ES and DirectX graphics pipelines.
Shader language	GLSL extended with a few engine specific features (such as file include and predefined uniform mappings)	Unity uses their own shader language ShaderLab, which supports embedded Cg. ShaderLab are internally being translated to Cg and on OpenGL platforms translated to GLSL.
Engine language	JavaScript	C++
Scripting language	JavaScript	C#, Unity Script (a JavaScript like language) and Boo.
Scripting runtime	JavaScript engine (Such as V8 for Chrome,)	The script engine is the Mono runtime – an open source implementation of the .Net platform (including the .Net 2.0 libraries).
Rendering pipeline	Forward rendering	Forward rendering or deferred rendering
Exposed low-level graphics API	Yes using a reference to the WebGL Context.	Yes – using the GL object, which emulates the OpenGL API.
Third party libraries	Any JavaScript library	DLL-libraries are supported.

5.2.2 Benchmark: Unique draw calls

Purpose of test

One important feature of any game engine or render engine is how fast the engine can draw unique mesh instances. For each mesh instance the engine has to bind it to match the current shader and then issue the draw call.

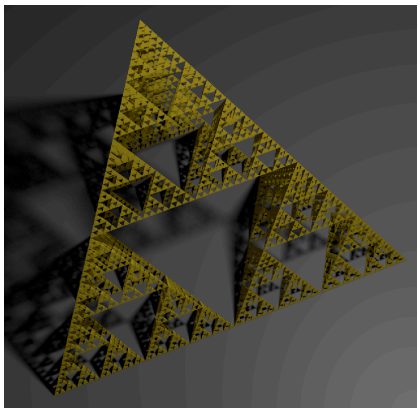


Figure 10 Sierpinsky Pyramid with recursion depth of 6

Description of test case

This test case uses the Sierpinsky Pyramid as a mesh instance (see Figure 10). The geometry is copied to each mesh instance and therefor exists in the GPU memory multiple times. This is done to simulate a more common use case where a scene has different geometry. If the geometry had not been duplicated, both engines are smart enough to only bind the geometry once.

The Sierpinsky Pyramid is created by recursive subdivision with a depth of 1. Usually Unity would batch such draw calls, but this has been turned off to be able to compare performance without batching. In this test we are interested in testing draw-calls, not how well batching works (besides batching of geometry is currently not implemented in KickJS).

The mesh instances are all put inside the view volume, to prevent the view volume frustum to optimize the rendering (resulting in fewer draw calls).

Finally the viewport size is chosen to be 1x1, since we are not interested in the performance of the fragment shader.

Test result

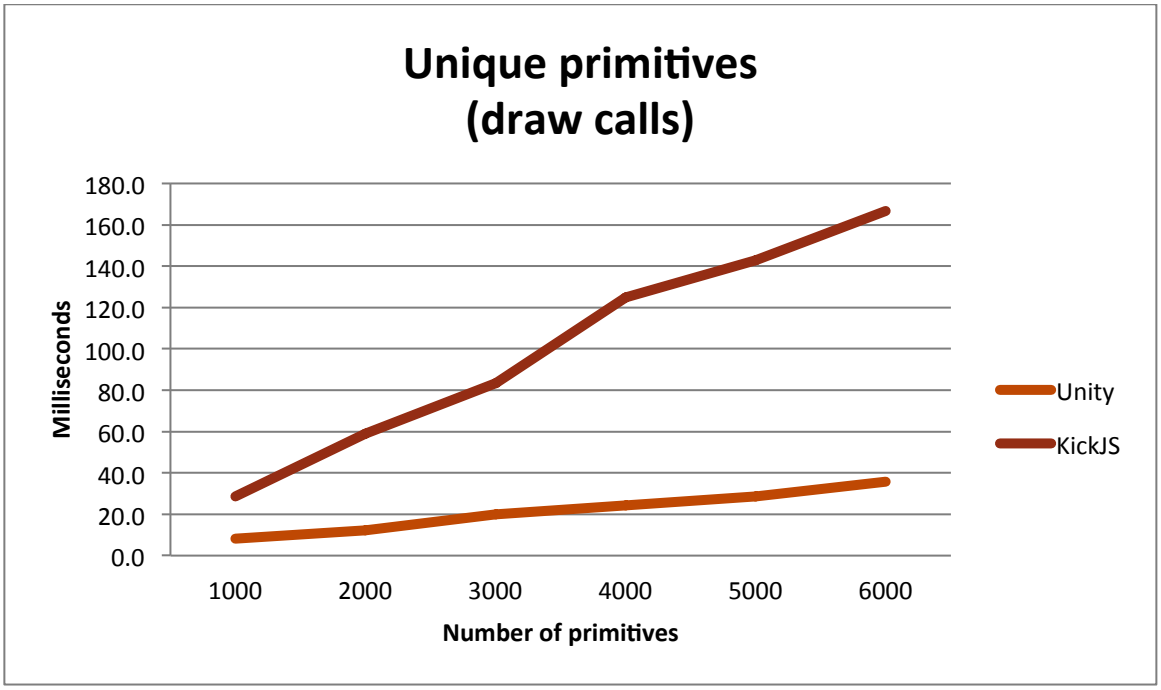


Figure 11 Performance of unique draw calls

Evaluation of the test result

In general the performance of the Unity engine is much higher (approximately a factor of two to four). I believe the reason for this is the performance penalty of the WebGL / JavaScript engine in contrast to the highly optimized C++ code in Unity.

5.2.3 Benchmark: View volume culling

Purpose of test

In this test I will see the effect of using view volume culling (also known as view frustum culling). In view volume culling geometry are culled way before the rendering, which reduces the cost setting up the rendering on the CPU as well as the rendering cost on the GPU.

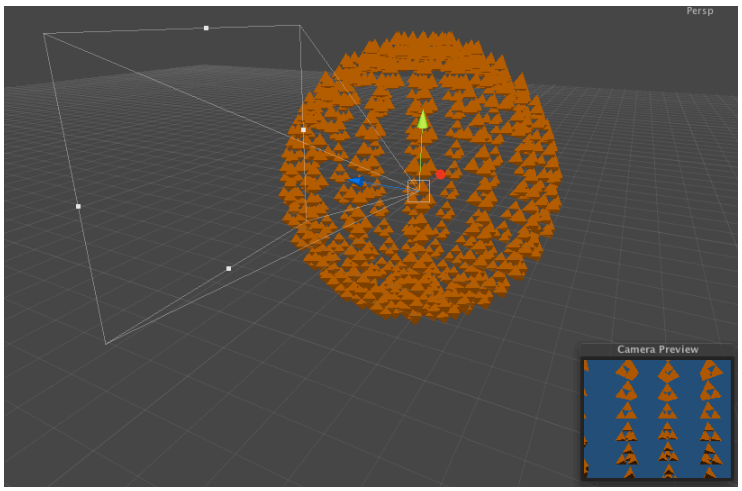
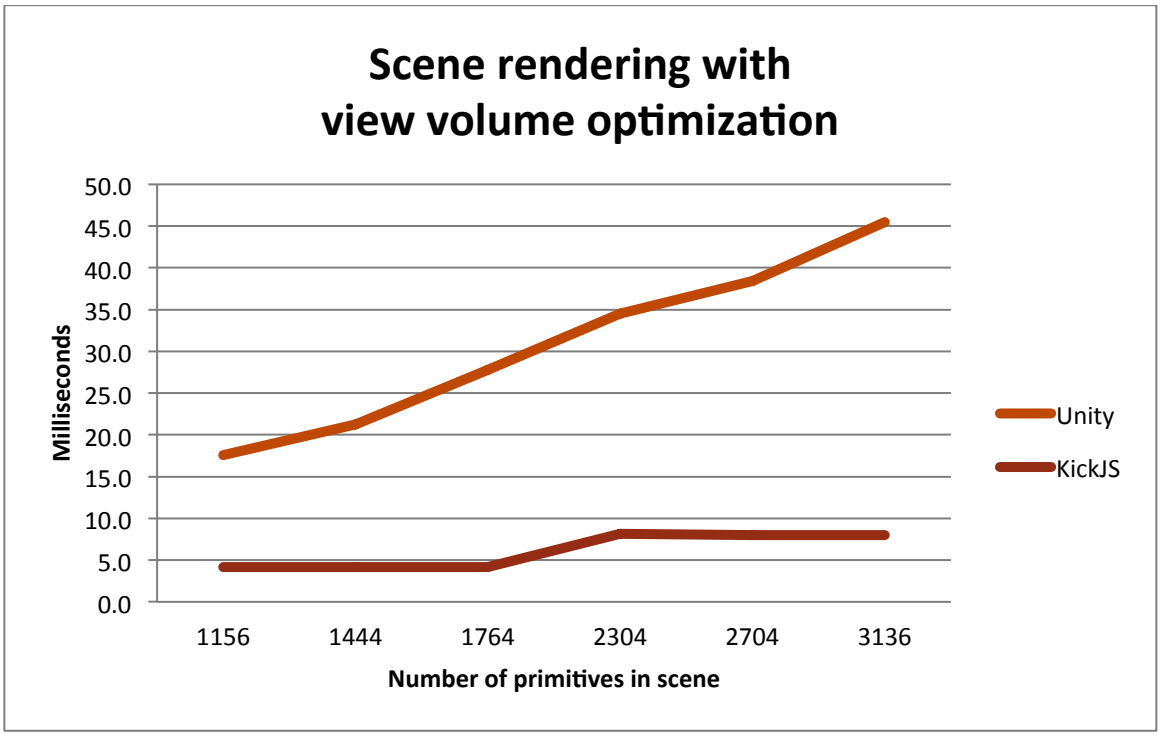


Image 9 View volume culling setup (screenshot from Unity)

Description of test case: The Sierpinsky Pyramid from the test case above is reused. But in this setup the pyramids are uniformly distributed over a sphere around the view frustum as shown in Image 9. The scene is rendered to a 500x500 buffer (fill-rate is not going to be a bottle neck in this case).

Test result:



Evaluation of the test result:

Both Unity and KickJS reduces the cost of rendering large scenes significant when using view volume culling. The performance gain mainly comes from the cost of doing binding the mesh and invoking the draw call.

Unity clearly uses a more optimized for taking advantage of view volume culling. Unity here performs 4-6 times faster.

5.2.4 Benchmark: Material changes

Purpose of test: One of the most important things in the rendering is change of materials. In complex scenes with many visible game objects, changing material can easy become a bottleneck.

Description of test case: A number a game objects each with a mesh-renderer and a sphere attached is put inside the view volume, which prevents any view volume culling. Each mesh will be assigned a unique material with a random colour. All materials share the same shader. By using different materials, Unity cannot make any batching of geometry.

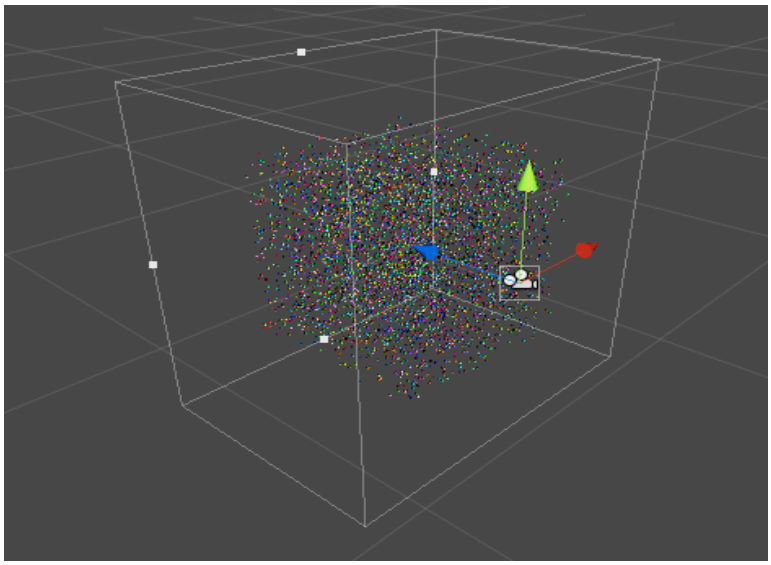
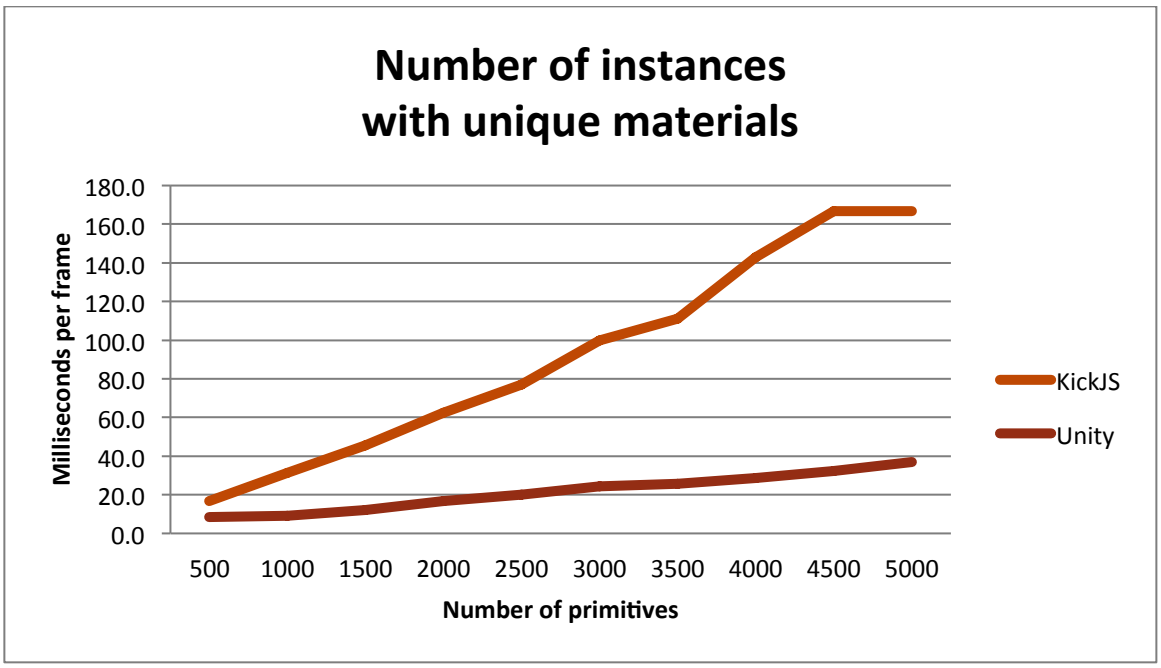


Image 10 Setup of material test

Test result:



Evaluation of the test result: Currently Unity outperforms KickJS with a factor 2-4. I believe that the KickJS engine could be optimized a bit to improve cost of binding materials.

5.3 Summary

JavaScript is significant slower that optimized code written in C++. With the current generation of V8 the JavaScript runs in general between 3 to 10 times slower than C++ code. One reason JavaScript has worse performance may be caused by JavaScript using more memory than C++.

Unity performs better than KickJS in all tests. One obvious reason is that Unity is much more optimized than KickJS.

However I do find the performance of WebGL is reasonable and think it is a good choice for 3D games that doesn't need to push the hardware to the limits.

6 Conclusion

Web Applications are becoming more and more advanced and the browser is now considered a platform on which large and complex applications can be built. One of the main reasons why this is possible is the evolution of JavaScript engines, which have improved significantly over the last years. JavaScript is no longer used just for making small visual effects on web pages, but is also used for implementing advanced client-side application code.

With the introduction of WebGL, browsers entered a new era - hardware accelerated 3D graphics. This means that 3D games and applications now can run in a WebGL capable browser. The WebGL architecture also has the additional benefit that most of the heavy computations are performed on the GPU and not in the JavaScript code.

Even though JavaScript has improved significantly, it is still not running as fast as programs written in C++. The performance benchmarks show that JavaScript runs around 3 times more slowly in general and for 3D math operations it runs around 10 times slower. Note that these numbers are based on micro benchmarks and may not reflect performance of actual applications and games.

I have created the WebGL based game engine KickJS to illustrate the creation on an engine in high performance JavaScript. The engine encapsulates much of the complexity of 3D game development while at the same time ensuring a well-designed architecture for games. The engine is shader based and ships with a number of built-in shaders. The engine supports importing 3D models in Collada or Wavefront OBJ format. For building advanced 3D scenes I have created a scene editor, which gives a visual way of working with 3D scenes.

When benchmarking KickJS against the commercial game engine Unity (C++/C# based), the performance of Unity was in approximately 2-4 times higher than KickJS. However this comparison is far from equal, since Unity offers a lot of capabilities that KickJS does not have.

Even though JavaScript has some limitations, WebGL will likely bring a lot of small and medium sized 2D and 3D accelerated games to the Internet. This includes versions of classic 3D game titles, which is much less resource demanding than current productions.

7 Appendix A: performance tests and other tests

7.1 JavaScript technique benchmark

All benchmarks below can be found in the KickJS project:

```
/test/unittest/JSPerformanceTest.html
```

The benchmarks are written as YUI Unit tests which also tracks time.

7.1.1 Benchmark: Typed arrays

Milliseconds for calculating the sum of an array of 10 million numbers.

	Time:
Chrome 16.0.912.77	
ArraySum	7482.00
Float32Sum	6145.00
Float64Sum	5836.00
Uint8Sum	5798.00
Firefox 9.0.1	
ArraySum	2983.00
Float32Sum	2355.00
Float64Sum	2504.00
Uint8Sum	2123.00

7.1.2 Benchmark: Object allocation

Milliseconds for running 10 million reflect invocations.

See chapter 2.2.4 page 18

	Time:	Performance gain:
Chrome 16.0.912.77		
testReflectNäive (baseline)	65248.00	1.00
testReflectOptimized	6984.00	9.34
Firefox 9.0.1		
testReflectNäive (baseline)	13689.00	1.00
testReflectOptimized	3419.00	4.00

7.1.3 Benchmark: Object constants vs. pre-compiler

Benchmarks the fps when using a pre-compiler and without a pre-compiler

Rækkenavne	FPS:	Performance gain:
Chrome 16.0.912.77		
Precompiler on	34.54	1.02

Precompiler off	34.03	1.00
Firefox 9.0.1		
Precompiler on	34.45	1.07
Precompiler off	32.27	1.00

7.2 Other tests

7.2.1 JavaScript V8 vs. GCC C++

V8 version 3.9.5

gcc version 4.6.1

Ubuntu/Linaro 4.6.1-9ubuntu3

From

http://shootout.alioth.debian.org/u32/benchmark.php?test=all&lang=v8&lang2=gpp

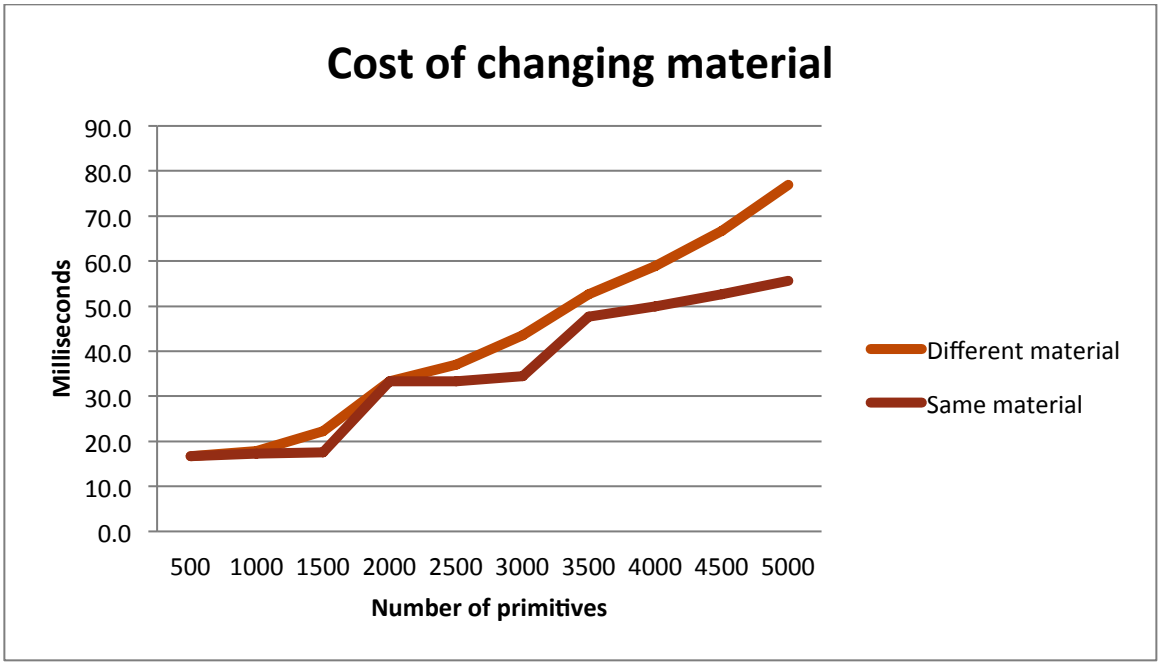
Program Source Code	CPU secs	Elapsed secs	Memory KB	Code B	≈ CPU Load
regex-dna					
JavaScript V8	4.19	**4.20**	202,952	373	0% 0% 0% 100%
C++ GNU g++	17.01	17.01	94,660	1759	0% 0% 0% 100%
fannkuch-redux					
JavaScript V8	80.52	80.55	4,756	472	0% 0% 0% 100%
C++ GNU g++	47.85	47.86	952	1440	0% 0% 0% 100%
binary-trees					
JavaScript V8	38.79	38.83	393,256	467	0% 0% 0% 100%
C++ GNU g++	12.93	12.95	148,448	892	0% 1% 1% 100%
fasta					
JavaScript V8	19.89	19.90	32,736	923	0% 0% 1% 100%
C++ GNU g++	6.31	6.31	256	1266	1% 0% 0% 100%
spectral-norm					
JavaScript V8	33.13	33.15	6,884	311	0% 0% 0% 100%

C++ GNU g++	10.00	10.01	608	1044	0% 0% 1% 100%
n-body					
JavaScript V8	85.86	85.92	13,024	1287	0% 0% 0% 100%
C++ GNU g++	20.11	20.12	284	1659	0% 0% 0% 100%
reverse-complement					
JavaScript V8	17.38	17.40	321,684	456	0% 0% 0% 100%
C++ GNU g++	1.07	1.09	245,348	2275	3% 2% 2% 100%
k-nucleotide					
JavaScript V8	274.79	274.93	332,832	423	0% 0% 0% 100%
C++ GNU g++	11.86	11.88	133,472	3415	0% 0% 0% 100%
pidigits					
JavaScript V8	277.87	278.07	24,088	609	0% 1% 0% 100%
C++ GNU g++	2.74	2.74	1,680	682	0% 1% 0% 100%

7.2.2 Performance gain by reducing change of materials

The goal of this test is to estimate the performance cost of changing materials.

The test case renders a number of spheres in a scene with either the same material or with different materials – both using the same browser. From this the approximated cost of changing a material can be seen.



In this test case the performance improvement of keeping material is around 16 %, but it fluctuates a bit.

7.2.3 Double precision test

To test that the browser is actually is performing all calculations with double precision even on data stored in 32-bit precision the following test is run:

```
var x = [12345678,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1];
var y = new Float32Array(x);
// if this calculation is done with only single precision, you will
// end up with only y[0], since all other values are rounded to zero
var sumY = y[0]+y[1]+y[2]+y[3]+y[4]+y[5]+y[6]+y[7]+y[8]+y[9]+y[10];
Assert.areNotEqual(sumY, y[0], "Browser does not perform calculations
with double precision");
```

In JavaScript the `sumY` is evaluated to 12345679.0.

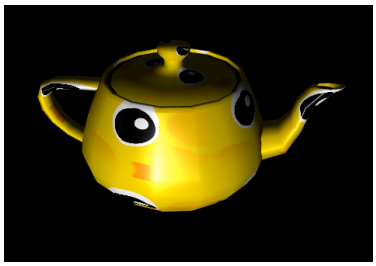
If you perform the same calculation in C++ using floats the result is evaluated to 12345678.0

```
float x[11] = {12345678,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1};
float sumY = x[0]+x[1]+x[2]+x[3]+x[4]+x[5]+x[6]+x[7]+x[8]+x[9]+x[10];
printf("Float sum is : %f\n",sumY);
```

From this “proof” we can conclude that all calculations are done with 64 bit precisions even though the values can be stored with only 32-bit precision. This also means that the JavaScript engine adds a small overhead on calculations using Float32Arrays for converting between 32-bit and 64-bit precision.

7.2.4 Teapot storage: JSON vs. Binary

A model of the Utah teapot with normals, UVs and texture coordinates are stored. The model has 792 vertices.



	JSON	KickJS binary	Reduction
No compression	116 KB	57 KB	51%
Zip compression	22 KB	14 KB	37%

8 Appendix B: Example applications

In this appendix I will describe some of the examples that I have implemented using the KickJS engine.

8.1 Snake

Snake is a very simple game where two players compete in collecting most dots while avoiding crashing into walls or snakes.

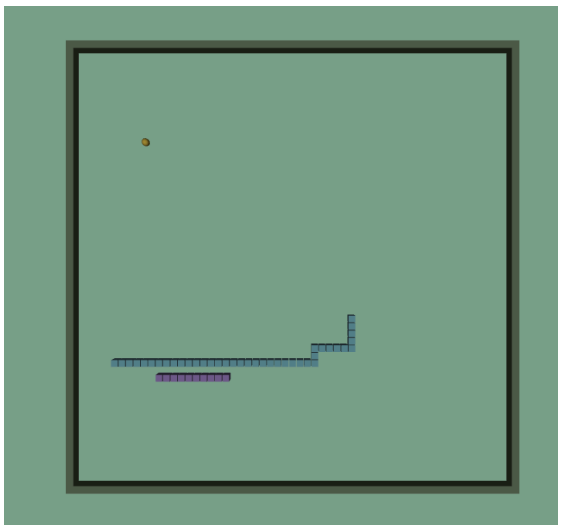
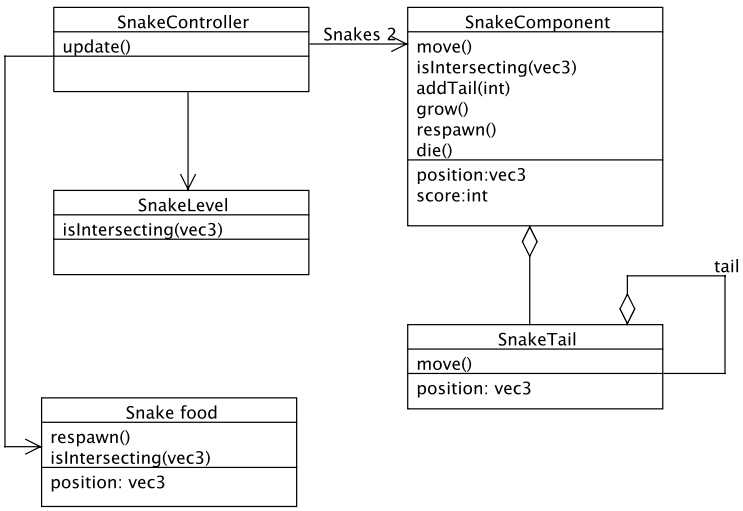


Image 11 KickJS Snake game

The game is modelled with a simple component based structure, with one game game-controller responsible for running the gameplay:



8.2 Model viewer

The model viewer was developed as a test-tool for importing Collada and Wavefront OBJ files into KickJS. The functionality has been extended so that the tool is now able to also export JSON files and binary KickJS models.

The tool also supports loading PNG and JPEG images, which will be used as textures in all materials of the model.

The tool automatically rotates the camera around the origin, but the user can control the camera rotation using left mouse drag and zoom using the mouse scroll wheel.

The tool contains tree built-in models: A duck, a cube and a teapot. The default texture is the yellow duck texture.

Image 12 Model viewer

8.3 Cloth simulation

The cloth simulation is a port of the tutorial "Mosegaards Cloth Simulation Coding Tutorial"[42] – a C++/OpenGL introduction to cloth simulation.

The code is translated to JavaScript as closely as possible. This unfortunately has the downside that the simulation allocates memory in every frame, which is one of the reason why the code runs much slower that the C++ equivalent.

[42] http://cg.alexandra.dk/2009/06/02/mosegaards-cloth-simulation-coding-tutorial/

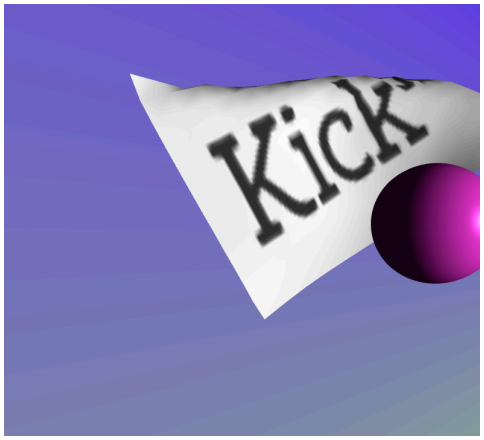


Image 13 Cloth simulation in KickJS

The memory allocation issue could be fixed, but the code would then be harder to read and understand.

To get good performance in WebGL I believe that the computation should be moved from the CPU (JavaScript) to the GPU (GLSL).

8.4 Video ASCII art

Shaders can be used for many other things than shading 3D geometry. In this example I have created a post processing effect that adds a post-effect to a video and show it on the screen. The post effects transforms the videos to ASCII art by first pixelate the content and then choose the best ASCII character for each pixel.

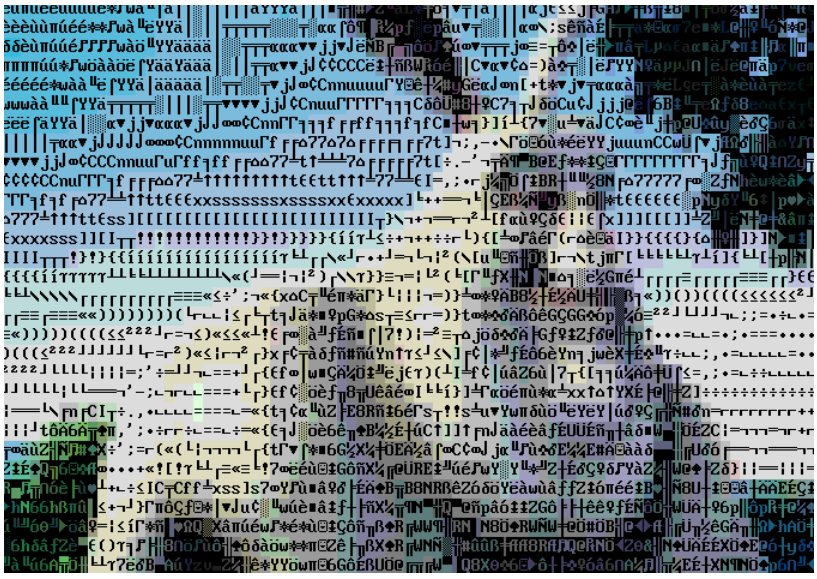


Image 14 Video ASCII shader

The program grabs the video frame and updates an internal WebGL texture with it every frame. Then the texture is rendered using a screen-filling quad with the ASCII shader.

9 Appendix C: Documentation

The full documentation of each class is available online:

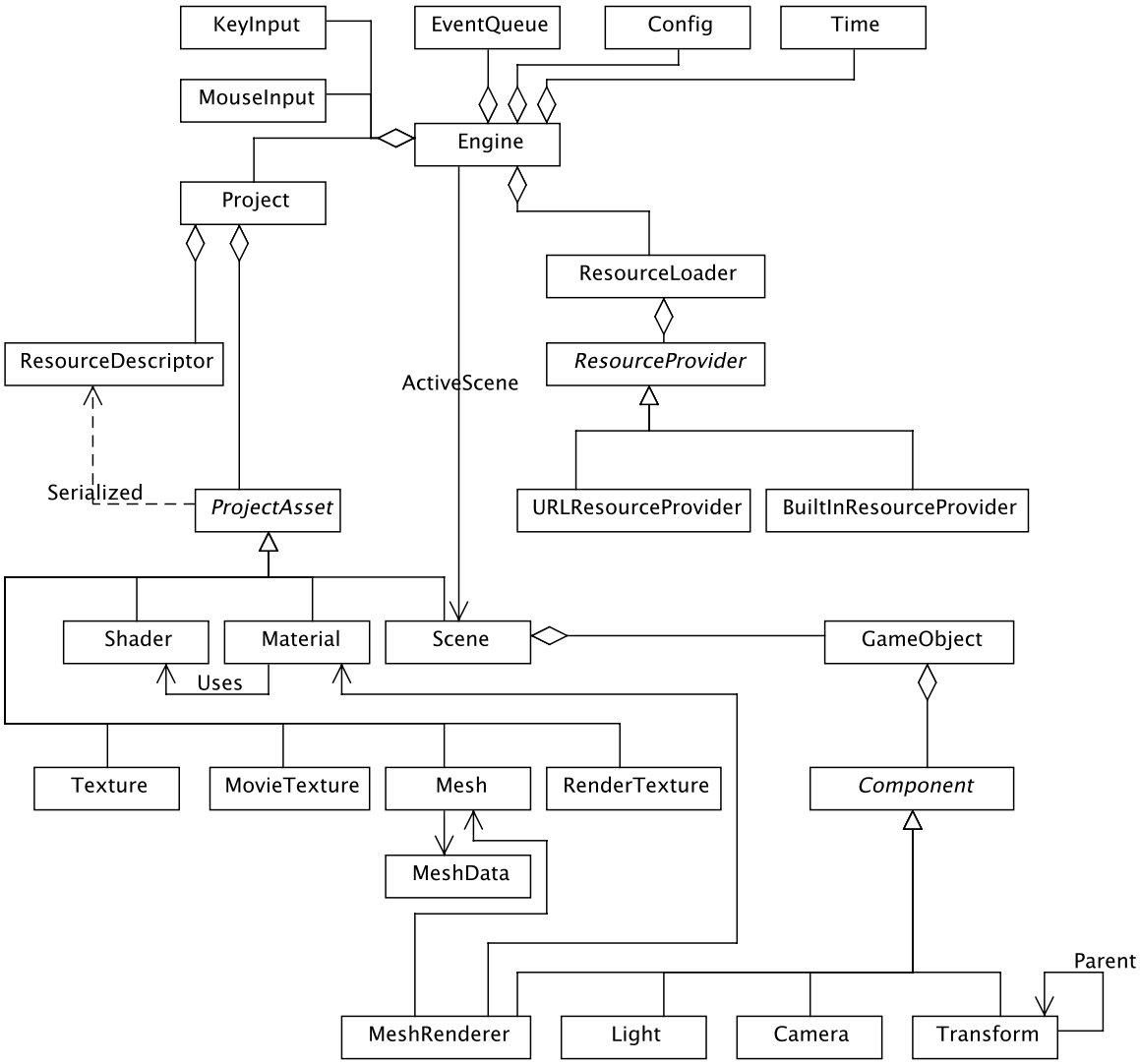
http://www.kickjs.org/api/

Class name	Description
KICK.core.ChunkData	Chunk data format object
KICK.core.Config	The global configuration of the engine. Cannot be changed during runtime.
KICK.core.Constants	This class contains references to WebGL constants.
KICK.core.Engine	Game engine object
KICK.core.EventQueue	Event queue let you schedule future events in the game engine.
KICK.core.KeyInput	This class encapsulates keyboard input and makes it easy to test for key input.
KICK.core.MouseInput	Provides an easy-to-use mouse input interface.
KICK.core.Project	A project is a container of all resources and assets used in a game.
KICK.core.ProjectAsset	A project asset is an object that can be serialized into a project and restored at a later state.
KICK.core.ResourceDescriptor	A project is a container of all resources and assets used in a game.
KICK.core.ResourceLoader	Responsible for loading of resources.
KICK.core.ResourceProvider	Responsible for creating or loading a resource using a given URI
KICK.core.Time	A global timer object
KICK.core.Util	Utility class for miscellaneous functions.
KICK.importer.ColladaImporter	Imports a Collada meshes into a scene
KICK.importer.ObjImporter	Imports a Wavefront OBJ mesh into a scene.
KICK.material.GLSLConstants	Contains GLSL source code constants
KICK.material.Material	Material configuration
KICK.material.Shader	GLSL Shader object which encapsulates a GLSL shader programs and WebGL states.
KICK.math.aabb	Axis-Aligned Bounding Box.
KICK.math.mat3	3x3 Matrix
KICK.math.mat4	4x4 Matrix
KICK.math.quat4	Quaternions
KICK.math.vec2	2 dimensional vector
KICK.math.vec3	3 dimensional vector
KICK.math.vec4	4 dimensional vector

Class name	Description
KICK.mesh.Mesh	A Mesh object allows you to bind and render a MeshData object
KICK.mesh.MeshData	Mesh data class. Allows for modifying mesh object easily. This is a pure data class
KICK.mesh.MeshFactory	Class responsible for creating Mesh objects
KICK.renderer.ForwardRenderer	Forward renderer
KICK.renderer.NullRenderer	Does not render any components
KICK.renderer.Renderer	Defines interface for render classes.
KICK.scene.Camera	Creates a game camera
KICK.scene.Component	This class only specifies the interface of a component.
KICK.scene. ComponentChangedListener	Specifies the interface for a component listener.
KICK.scene.GameObject	Game objects. (Always attached to a given scene).
KICK.scene.Light	A light object. (Directional, ambient or point light)
KICK.scene.MeshRenderer	Renders a mesh
KICK.scene.Scene	A scene objects contains a list of game objects
KICK.scene.SceneLights	Data structure used pass light information
KICK.scene.Transform	Position, rotation and scale of a game object.
KICK.texture.MovieTexture	A movie texture associated with a video element will update the content every frame.
KICK.texture.RenderTexture	Render texture (used for camera's render target)
KICK.texture.Texture	Encapsulate a texture object and its configuration

10 Appendix D: UML Class diagram of KickJS

The following contains the main classes of KickJS:



11 Appendix E: Glossary

AAA game	Big budget commercial game
Application framework	A framework is a reusable, ``semi-complete'' application that can be specialized to produce custom applications[43]. Using a framework has the benefits of[44]: • **Inversion of control:** The framework is responsible for flow of control. A program using the framework responds to events triggered by the framework. • **Reusability:** A framework implements common behaviour • **Extensibility:** Allowing customization through inheritance • **Modularity:** Encapsulating its implementation and exposes an API for the program
Chrome Frame	A plugin for Internet Explorer that embeds the full Google Chrome browser inside on web pages that contains a special meta tag. This allows usage of some of the more advanced features of Chrome, such as Native Client and WebGL applications. The plugin can be installed on any windows machine even for users without administrator access.
Closure	[Crockford08]: "[Closure is] inner functions get access to the parameters and variables of the functions they are defined within". Note that inner functions can have longer lifetime than outer functions.
ECMAScript	The official name for standardized JavaScript
Game engine	A framework used for creating games. A game engine gives a separation between engine logic, such as rendering systems, sound system, resource management and game logic such as rules and game specific code. A game engine usually consists of several sub-modules each with a single responsibility, such as rendering, event management, input management, sounds, etc.
GLSL	OpenGL Shader Language, used for programming the GPU to perform shading operations. In WebGL there exists two types of shaders: vertex shaders (transforms vertices into clip-space) and fragment shaders (defines the pixel colour).
Google App Engine	A software stack that allows developers to create web applications that runs on Google hosted webservers. The goals of Google App Engine are: Easy to build, easy to maintain, and easy to scale
HTML5	An umbrella specification that covers many parts such as HTML based markup, Video capabilities, Audio capabilities, Canvas 2D API and Web Storage API. The specification is not approved yet, but many browsers implements large parts of the standard.

[43] Johnson88
[44] Fayad97

IEWebGL	A traditional plugin that adds WebGL functionality
JSON	JSON (JavaScript Object Notation) is a lightweight data-interchange format. The format uses two structures: key/value objects and ordered lists. Values can be strings, numbers, objects, arrays, true, false or null.
Library	Provides implementation of common function for modular programming
NodeJS	Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications.
Sandbox	Secure environment programs can run in without restricted access to file system, memory and other resources.
Typed Arrays	A new way JavaScript can work with binary data. It allows creation of a binary array buffer and seeing this array buffer through array buffer views of a certain type (such as 32-bit float or unsigned integer 8 bit)
Web application	An application created in HTML, JavaScript and CSS, hosted on a webserver and accessed over the Internet using a web browser.
XML	Extensible Markup Language, data definition language that is written using text based markup.

12 Bibliography

Books:

Crockford08 Crockford, Douglas, 2008, "JavaScript: The Good Parts". Yahoo Press

Gove11 Gove, Darryl, 2011, "Multicore Application Programming – for Windows, Linux, and Oracle Solaris", Addison Wesley

Gregory09 Gregory, Jason, 2009, "Game Engine Architecture". A K Peters

Larman97 Larman, Craig, 1997, "Applying UML and patterns: an introduction to object-oriented analysis and design". Prentice Hall PTR

Pharr05 Pharr, Matt, 2005, "GPU Gems 2: Programming Techniques for High-performance Graphics and General-purpose Computation", Addison Wesley

Stefanov10 Stefanov, Stoyan, 2010, "JavaScript Patterns". O'Reilly Media

Articles:

Fayad97 Fayad, M., Schmidt, D. C., 1997, "Object-Oriented Application Frameworks" Communications of the ACM, Special Issue on Object-Oriented Application Frameworks, Vol. 40, No. 10, October 1997. http://www1.cse.wustl.edu/~schmidt/CACM-frameworks.html

Johnson88 Ralph Johnson and Brian Foote. ``Designing Reusable Classes." Journal of Object-Oriented Programming. SIGS, 1, 5 (June/July. 1988), 22-35.

Llopis09 Llopis, Noel, "Data-Oriented Design" Game Developer, Vol 16 No 8, September 2009

Blogs and web articles:

Egorov11 Egorov , Vyacheslav , "Understanding V8", June 11, 2011, Nodecamp.eu http://s3.mrale.ph/nodecamp.eu/

Garrett05 Garrett, J.J. "Ajax: A New Approach to Web Applications", February 18, 2005 http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications

Larson-Green11 Larson-Green, J., corporate vice president, Windows Experience "Previewing 'Windows 8'", Microsoft News Center, June 1, 2011 http://www.microsoft.com/presspass/features/2011/jun11/06-01corporatenews.aspx

Mandelin11 Mandelin, Dave, "Know Your Engines - How to Make Your JavaScript Fast", June 15, 2011, O'Reilly Velocity http://people.mozilla.com/~dmandelin/KnowYourEngines_Velocity2011.pdf

Meisinger10 Meisinger, G., "Why I switched from component-based game engine architecture to functional reactive programming", August 16, 2010 http://lambdor.net/?p=171

Metzger11 Metzger,H., "Netscape history", October 30, 2011 http://www.holgermetzger.de/Netscape_History.html

Rauschmayer11 Rauschmayer, Dr. A., "JavaScript values: not everything is an object", Match 14, 2011
http://www.2ality.com/2011/03/javascript-values-not-everything-is.html

Stroustrup11 Stroustrup, B., "Bjarne Stroustrup's C++ Style and Technique FAQ", December 17, 2011
http://www2.research.att.com/~bs/bs_faq2.html#finally

Webber11 Webber, Joel, "Angry Birds on HTML5", October 10 2011, GOTO Conference,
http://www.infoq.com/presentations/Angry-Birds-on-HTML5

West07 West, M., "Evolve Your Hiearchy – Refactoring Game Entities with Components", 1st of May, 2007
http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/

Winokur11 Winokur, D., vice president & general manager, interactive development at Adobe
"Flash to Focus on PC Browsing and Mobile Apps; Adobe to More Aggressively Contribute to HTML5", Adobe blog, November 9, 2011
http://blogs.adobe.com/conversations/2011/11/flash-focus.html

Specifications:

Ecma11 "ECMAScript Language Specification - ECMA-26 - 5.1 Edition", June 2011
http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

UML11 "OMG Unified Modeling LanguageTM (OMG UML), Superstructure - Version 2.4.1", June 8, 2011
http://www.omg.org/spec/UML/2.4/Superstructure

Webgl11 Khronos Press release: "Khronos Releases Final WebGL 1.0 Specification", March 3, 2011
http://www.khronos.org/news/press/khronos-releases-final-webgl-1.0-specification