JavaScript and Web Integration of AgX Multiphysics Engine

Tomas Brännström

March 25, 2011

Master's Thesis in Computing Science, 30 credits

Supervisor at CS-UmU: Ola Ringdahl
Supervisor at Algoryx: Anders Backman
Examiner: Frank Drewes

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

JavaScript is the *de facto* language for client-side scripting on the web and has in recent years received a great boost in performance thanks to the 'browser wars' where the different browser vendors compete to get higher market shares. AgX is a *physics engine* developed by Algoryx in Umeå, intended for use in simulations. This thesis explores possible methods of integrating JavaScript with AgX and in extension the possibility of integrating AgX into the web browser as a plugin, effectively enabling AgX for use in different web applications. Efforts were made to combine the embedded JavaScript API with the AgX browser plugin. This was found to be unfeasible for different reasons, although the AgX plugin as a separate piece of software was shown to work well.

Contents

1 Introduction **1**

1.1 Algoryx 1

1.2 AgX Multiphysics Engine 2

1.3 Outline 2

2 Problem Description **3**

2.1 Problem Statement 4

2.2 Goals 4

2.3 Purposes 4

2.4 Methods 5

2.5 Related Work 5

3 A Look at Current and Future Web Technologies **7**

3.1 Introduction 7

3.2 JavaScript 8

3.3 The DOM 11

3.4 Browser Plugins 13

3.5 Timeline of Web Technologies 13

3.6 HTML5 17

3.7 Browser Graphics 19

3.8 Conclusion 22

4 JavaScript Embedding 25

4.1 Scripting Languages 25

4.2 V8 26

4.3 Related Technologies 26

4.4 The Good, the Bad and the Ugly 27

4.5 Pipeline for Autogenerating Bindings 28

4.6 Conclusions 40

5 Physics in the Browser 41

5.1 Motivation 41

5.2 V8 vs NPAPI 42

5.3 Firebreath 42

5.4 Autogeneration? 42

5.5 Prototype Plugin 43

5.6 Performance 47

5.7 Installation/Deployment 49

5.8 Conclusion 50

6 Results **53**

7 Conclusions **55**

7.1 Limitations 56

7.2 Future work 56

8 Acknowledgments **59**

References **61**

A Example of Embedding C++ Into V8 **65**

B User's Guide **71**

B.1 Code Generator 71

B.2 Plugin 73

C List of WebGL Libraries **79**

D Source Code **81**

List of Figures

3.1 **The DOM of a sample document.** 12

3.2 **Rough overview of breakthroughs of web technologies.** 14

3.3 **The canvas coordinate system.** 20

3.4 **An example WebGL application.** 22

4.1 **The 'wiz8' class-registration chain.** 30

4.2 **Overview of the codegen system.** 33

4.3 **UML diagram of the V8 interface.** 34

5.1 **UML diagram of plugin. Details are omitted.** 44

5.2 **Example of a scenegraph. Traversing the graph will apply the transformations to all subsequent nodes.** . . . 46

5.3 **Screenshot of a sine curve plotted with the *flot* JavaScript plotting library.** 47

5.4 **Performance of updating SceneJ scenegraph** 48

5.5 **FPS of the system depending on no. bodies** 49

B.1 **Screenshot of a web page with the plugin, running in Chromium.** 74

Chapter 1

Introduction

Embedded scripting is a useful tool for many kinds of applications. It can be useful for plugin/extension development (in web browsers, for instance), rapid prototyping, or to allow for programmable behavior of entities in for example games, just to name a few examples. *JavaScript* is currently one of the most deployed scripting languages in existence, thanks to its extensive use as a 'web language'. A very large number of web pages uses JavaScript for different tasks.

In this thesis the prospect of using JavaScript as an embedded scripting language for the AgX multiphysics engine (see Section 1.2) developed by Algoryx (see Section 1.1) was explored. They were interested in replacing their current Lua solution for embedded scripting with JavaScript. A big reason for this was the possibility of using AgX as a browser plugin, making their engine accessible from the Internet - with JavaScript as the scripting language for the engine, it might be possible to use the same bindings for both 'offline' and 'online' scripting: this is explored in this thesis as well.

1.1 Algoryx

Algoryx is a company based in Umeå and they were founded in 2007 as a spin-off company from Umeå University[8]. Algoryx as a company develops physical simulations for business and education. They are the developers of the AgX physics engine and Algodoo, a 2D physics sandbox.

1.2 AgX Multiphysics Engine

AgX is a *physics engine*, i.e. a software library used for creating physical simulations, written in C++. Physics engines are often developed for speed rather than accuracy, since they are often used in games and other interactive simulations where the appearance might be more important than 100% accurate results. AgX, however, is intended for professional and industrial applications and aims to have a high accuracy of their simulation while still having great performance [7].

Physics engines are, as mentioned, software libraries/toolkits used to simulate (usually Newtonian) physics on a computer. A common usage scenario for a physics engine is in movies where they can be used to simulate buldings being destroyed, water, cloth, and so on. A typical entity in a physics engine is a *rigid body*, i.e. a body that cannot deform. They act as a approximations to real-life physical entities, since in reality there are no perfectly rigid (i.e. an object that can not deform) bodies [45]. To see an example of rigid bodies in use in AgX, see the video available at [7]. In this video, the rocks are represented by rigid bodies. In addition to rigid bodies, AgX can simulate wires and fluids.

1.3 Outline

A rough outline of all the chapters in this report:

- Chapter 2: A general description of the problem this thesis attempts to solve.
- Chapter 3: Study of web technologies.
- Chapter 4: Evaluation of JavaScript as an embedded scripting language.
- Chapter 5: Evaluation of AgX as a browser plugin.
- Chapter 6: Results
- Chapter 7: The conclusions reached during the thesis work.
- Chapter 8: Acknowledgments

Chapter 2

Problem Description

The problem was mainly to determine if JavaScript would be a suitable language for embedded scripting. This also includes actually producing the script bindings in an efficient and stable manner. The approach investigated during the thesis work was the possibility to create the bindings automatically in some way. It would be good if there were existing tools for this or parts of tools for other languages that could be used in some way, so related technologies had to be studied for ideas.

Furthermore, the possibility of implementing a browser plugin for AgX were also to be investigated. From early readings on the subject of plugin development it was quickly realized that working directly with the plugin API would not be doable in a satisfactory manner, in the timeframe given. This was due to the API being scarcely documented, very low level, and not multiplatform. Because of this a framework for this purpose would preferably have to be found and utilized.

The embedded scripting and the plugin should in the best case be unified, but if this was found to be difficult, then the two approaches would still be evaluated.

Using JavaScript and HTML, a number of graphical interface components becomes natively available. Their use in interacting with a possible browser plugin were also to be investigated: mainly buttons, sliders and plots.

2.1 Problem Statement

Evaluate JavaScript for use with embedded scripting in the AgX physics engine; find a good method of exposing the AgX API to JavaScript; evaluate the possibility of using AgX as a browser plugin, preferably scriptable using the same API from the embedding part of the project.

2.2 Goals

The thesis project was basically divided into three parts:

- Evaluate the use of JavaScript as an embedded scripting language targeting the V8 engine.
- Evaluate the use of JavaScript for creating the user interface (UI widgets, plots, graphics).
- Evaluate the possibility of creating a browser plugin for the AgX multiphysics engine.

The aim of the project was to reach an adequate result from the evaluation along with prototype implementations of each part to demonstrate the results.

2.3 Purposes

The main purpose of the project was to evaluate the possibility of taking the AgX physics engine and make it available as a component of the Web; to allow developers to create applications simulating physics in the browser. As a first step in this direction, the possibility of AgX using JavaScript as an embedded scripting language was to be evaluated. At the time of writing, AgX uses Lua for scripting. However, even if it would turn out that JavaScript would be more suitable than Lua for scripting (something that is a rather subjective opinion depending on how someone feels about either language, so this is not something that will be delved into further), if replacing Lua with JavaScript would require investing many hours in developing a pipeline to create script bindings, then there would be little purpose in attempting a switch from Lua. However, if a uniform way to create bindings for embedded scripting and browser plugin was discovered, then the possibility to replace Lua might seem more attractive.

Depending on the results other decisions might be made. If the plugin would turn out to be a good idea then a separate browser plugin could still be maintained, whether it would be scripted in Lua or JavaScript. This lies outside the scope of this thesis, but possible ways for future work is presented in Chapter Chapter 7.2.

2.4 Methods

The methods employed in the thesis were mostly studying and implementation: the different API's, frameworks and languages used to implement the prototypes had to be learned. Furthermore, a lot of thought were put into finding useful frameworks for the different prototypes.

2.5 Related Work

This thesis mainly touches two areas: embedded scripting and larger application API's in the web browser, with focus on physics engines.

2.5.1 Embedded Scripting

There exists a number of different solutions for embedded scripting. Since most languages have rather low-level API's with which to implement the bindings there are often a need for ways to autogenerate code or to abstract away most of the regular API. In Chapter 4.3 a few different tools are described that help with creating script bindings.

2.5.2 Browser Physics

There does not seem to be a lot of work in this area, so this thesis will explore some new ground. However, there are a few implementations. Box2D is a 2D physics engine that was originally written in C++, but there is also an ActionScript port which also lead to a port in JavaScript [53]. Additionally, the Bullet physics engine appears to be available through a Blender game engine plugin [1] (Blender uses Bullet as its physics engine). No further investigation were made into that, so it is unknown if it allows for full scripting of scenes or if it is just a plugin that loads some scene and displays it with no programmability.

Chapter 3

A Look at Current and Future Web Technologies

In this chapter an in-depth study on the Web and different technologies and their evolution is presented. It will focus on different facets e.g. HTML, JavaScript and how graphics on the Web has evolved.

This study is presented before the implementation/evaluation parts of the thesis since it will discuss many things that are relevant to know for the reader.

3.1 Introduction

The Internet has become a very important part of society in just a short time, and as such also a rather fundamental part of computer science. The Web is compromised of a myriad of technologies, both on the client and server side. On the client side we have things such as HTML, CSS and JavaScript to provide the presentation and interface for the user. There are also technologies like Flash: plugins which aim to provide more advanced graphics and such for the user. This is something that the browser (traditionally) have had trouble displaying in an efficient manner. On the server side we find more regular applications that communicate with databases and such to provide dynamic content.

In the thesis the focus lies mostly on the client side, and that is where the focus of the study should be as well: on the evolution of different web technologies and how they are used today, and also on the evolution of graphical content on the web.

3.2 JavaScript

Since this thesis puts a large focus on JavaScript, a short introduction to the language is given in this chapter. It can naturally be skipped if the reader has sufficient knowledge of the language already. The chapter also assumes at least some knowledge of programming/programming languages in general.

The name notwithstanding, JavaScript and Java should not be confused with each other since they are not compatible with each other [37]. JavaScript is a dialect of ECMAScript [38], and has since its introduction by Netscape become the *de facto* scripting language for client-side scripting on the Web.

3.2.1 History

The language was developed by Brendan Eich at Netscape and was at first called *LiveScript* but was renamed in part of a marketing agreement with Sun Microsystem; the name JavaScript was actually trademarked to Sun (and now in extent to Oracle after their acquirement of Sun). When the language was introduced Netscape was the market leader of browsers. This lead to the language growing very fast in popularity and usage; Microsoft responded by developing their own implementation called *JScript* which had a number of incompatibilities with the original language. A generic version of the language was standardized as ECMAScript by the European standards body (with the first edition of the standard surfacing in 1997 [15]), though incompatibilities between browsers and JavaScript implementations have been a problem for years now [42].

While initially being used for simple things, for example changing images as the user hovers over it with the cursor, as more advanced needs have developed JavaScript is used more and more for increasingly complex tasks. The more JavaScript is used, the faster the underlying browser implementations must be in order to provide a better browsing experience for the user. This is currently (at the time of writing) a large focus for the browser developers where they try to outmatch their competition in terms of JavaScript compliance and speed.

3.2.2 Client-Side Scripting

Though JavaScript is a useful language on its own, its primary use is as a host language in web browsers, providing control over the DOM (*Document Object Model*, see Section 3.3) as well as a few other features such as pop-up boxes and so on. The ECMA standard provide the basic language features that the

implementation must follow while the browser provides the host environment [15]. The JavaScript code is 'embedded' into web pages through the HTML `<script>` tags, e.g:

```
<script type="text/javascript">
  window.addEventListener('load',
      function() { alert('hello, world'); });
</script>
```

The above code would display a small box with the text 'hello, world' as the page is loaded. Further examples on how JavaScript can be used to access elements of a web page can be seen in Section 3.3.

3.2.3 Syntax and Types

While JavaScript and Java are two distinctly different languages, the syntax is very similar so a programmer coming from Java or C/C++ will feel at home. Following is a short example of some simple JavaScript:

```
function sumArray(arr) {
  var sum = 0;
  // C-style for-loops
  for (var i = 0; i < arr.length; i++)  {
    sum += arr[i];
  }
  return sum;
}
var arr = [1, 2, 3, 4, 5];
var sum = sumArray(arr);
```

JavaScript is dynamically typed [15]. When declaring a variable, instead of specifying the type, the keyword `var` is used. JavaScript uses *duck typing* (i.e. how an object can be used depends on its properties rather than its type) so if the 'wrong' object is passed to a function, no errors will be detected until a non-existing property is accessed. Dynamic typing is common in scripting languages so this is nothing unique to JavaScript [39].

3.2.4 Object Orientation

JavaScript is an object-oriented language, but unlike languages such as Java and C++ it does not have a concept of classes. Instead, everything in JavaScript (except primitives) are considered objects. Objects are created at run time and

properties can be added dynamically, contrasting with class-based object orientation where classed define a blueprint of sorts from which objects can be created. When an object has been created however, the object itself can act as a the blueprint, the *prototype* for other objects, see 3.2.5.

Additionally, Objects in JavaScript are basically, for all intents and purposes, associative arrays [15]. In the following piece of code, a few ways to create objects are exemplified along with ways to access member variables:

```
var x = {}; // Create an empty object.
var y = { foo : "test" }; // Create an object with attrib
                          // 'foo'
var z = new Array(); // Create an instance of 'Array'
x["bar"] = 3; // Adds a new property 'bar' to x.
y.baz = 5; // Adds a new property 'baz' to y.
// Note that the two above are equivalent ways to add
// properties to objects.
```

The `{ property : value }` syntax of defining objects in JavaScript has become a widely used scheme called JSON (for **J**ava**S**cript **O**bject **N**otation), used for e.g. serialization [4].

3.2.5 Prototypes

Each object is associated with something called a *prototype*. The prototype contains attributes and methods shared by instances of objects. For example, consider the following code:

```
var Foo = function() { ... } // define a 'class' Foo.
var foo = new Foo(); // create an instance of Foo
var bar = new Foo(); // -||-
Foo.prototype.baz = 3; // Add a property 'baz' to Foo.
print(foo.baz + bar.baz); // Will print 6.
```

Each object in JavaScript has an internal object called `__proto__`. When methods or attributes are accessed on objects, if they can't be found in the object directly, the `__proto__` object is recursively searched. Constructor objects, that is when we have for example `var Foo = function() {}`, are assigned the object called `prototype`. The members of the 'class' can be accessed from this object. When objects are instantiated using `new`, their `__proto__` object will be referencing `prototype`, containing the attributes and methods. Because of this, properties added to the prototype object will be added to all instances.

Inheritance

Inheritance works a bit different in JavaScript than in more conventional object-oriented languages, due to the fact that there are no classes. To inherit the properties of an object, the sub object should set its `prototype` property to a new instance of the parent object [36]:

```
function Base() { this.foo = "base"; }
function Sub() { this.bar = "sub"; }
Sub.prototype = new Base;
var sub = new Sub;
print(sub.foo); // prints "base"
```

3.2.6 Functional

In JavaScript, *everything* are objects, even functions. Functions are treated as any other object. It is also possible to declare anonymous functions, so JavaScript has good support for functional programming. For example, the arrays have a method `map` common in functional programming languages and languages supporting the functional paradigm:

```
var arr = [1, 2, 3];
var arrMapped = arr.map(function(x) {
                return x + 1;
              });
print(arrMapped); // will output [2, 3, 4]
```

3.3 The DOM

The DOM, Document Object Model, is an API for valid HTML and well-formed XML, defining the structure of documents [23]. The DOM allows the programmer to view, create and manipulate documents through a programming language. Commonly this is done through JavaScript on the client side; for example, to fetch a node in the DOM, and add a new element to it:

```
var node = document.getElementById('identifier');
var elem = document.createElement('elem');
node.appendChild(elem);
```

The logical structure of the DOM is that of a tree, or more precisely a forest, i.e. the DOM can contain several trees. A document can have zero or one

doctype node, one document *element* node and any number of comment nodes. The doctype specifies what kind of document that the parser is dealing with. An example would be

```
<!DOCTYPE html PUBLIC
    "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
```

which specifies an HTML 4.01 document. The document element node can be considered the root node of the document and would in an HTML document be the `<html>` tag. The following simple document

```
<html>
  <head>
    <title>Title</title>
  </head>
  <body>
    <h1>A heading</h1>
    <p>A paragraph</p>
  </body>
</html>
```

would yield a DOM as seen in Figure 3.1.

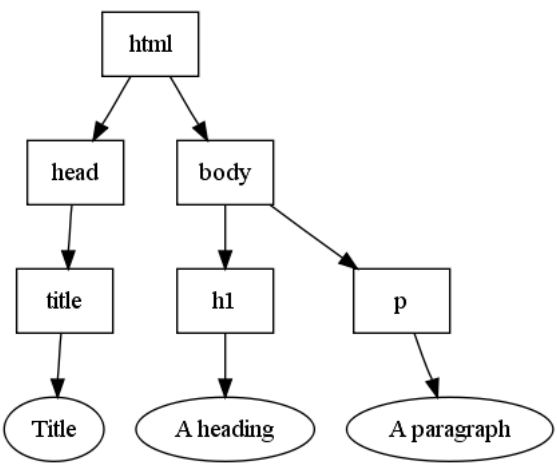

Figure 3.1: **The DOM of a sample document.**

To access unique elements in the DOM, they must be given an identifier which is done in HTML with the 'id' attribute:

```
<div id="theDiv"> ... </div>
```

The element can then be found in the document using the `getElementById` method.

3.4 Browser Plugins

All major browser has support for plugins and extensions. The reason for using plugins is to offer some kind of functionality that the browser lacks, or to offer enhanced functionality, for example: graphics, video, audio, ad-blocking functionality and so on.

The difference between a 'plugin' and an 'extension' might seem vague in this case since they basically mean the same thing, but here plugin will refer to an add-in software library written in a language such as C or C++ and compiled as a dynamic library from which the browser can run code, whereas an extension will refer to something written using some kind of API exposed by the browser.

To write plugins, some kind of API is required. NPAPI, short for *Netscape Plugin API*, was as the name suggests developed by Netscape for their browser. It has since become the standard for plugin development in web browsers, with all major browser except Internet Explorer having an implementation. Microsoft abandoned NPAPI in favor of their own technology ActiveX with Internet Explorer 5.5 Service Pack 2, stating that NPAPI would be insecure [14]. ActiveX is a general framework for plugin software on Windows platforms [34].

NPAPI exposes the DOM of the web page and provides ways to register objects and methods to JavaScript. Since it has access to the DOM documents can be manipulated from the plugin if so desired. Browsers supporting NPAPI loads the plugins from dynamic libraries, for example DLL's on a Windows platform. The plugins are not sandboxed and thus *do* pose a security risk to the user [24], albeit not more than a regular downloaded application would.

In 2009 Google presented a project called 'Pepper' which aims to be a modification of NPAPI to make plugins more portable and secure. They also developed a new SDK which they call 'Native Client' which relies on extensions to the Pepper project [33].

3.5 Timeline of Web Technologies

Since the introduction of the World Wide Web, a number of different technologies have been introduced. In Figure 3.2 a rough timeline of each technology

can be seen.

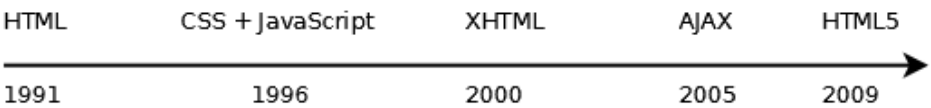

Figure 3.2: **Rough overview of breakthroughs of web technologies.**

In this section the more important breakthroughs are detailed to give a history of how the Web has developed.

HTML

HTML is an *SGML*, or *Standard Generalized Markup Language*[1]. It was originally developed by Tim Berners-Lee during his time at CERN and made popular through the Mosaic browser. There have been a number of different revisions of HTML since its creation: the current standard is HTML4 which extends the HTML with support for style sheets, scripting, frames, and embedding of external plugins just to name a few features [50].

CSS

CSS stands for Cascading Style Sheet and is a language for specifying the style and formatting of markup languages, e.g. HTML, XML and so forth. Its primary purpose is to separate the content of documents with their presentation: the markup language should be used to specify the contents while the CSS is then used to describe the appearance.

The separation of content and presentation allows one style sheet to be used for several different pages, as well as making it easier to change the design of a page since only the style sheet should be required to be modified. Following is a short example of HTML with CSS. The style sheet:

```
/* Example.css */
p.ex { text-decoration: underline; }
```

The HTML file:

```
<!-- Include the style sheet -->
<link rel="stylesheet"
  href="Example.css"
  type="text/css" />

<p class="ex">The text in this paragraph is underlined</p>
```

[1] SGML is a system for defining markup languages, so HTML is defined through SGML.

If the above style sheet is included, any HTML 'p' tag which is assigned the class 'ex' will yield underlined text.

XHTML

XHTML is a more 'strict' version of HTML. XHTML is XML, and as such it has a certain interface it must follow and which it can be validated against, e.g. DTD's or XML schemas. Since XHTML is XML, it can be parsed by any valid XML parser, whereas HTML which is rather 'flexible,' requires a more lenient parser and as such is a lot more complex to parse than XHTML [48].

A large problem with XHTML is that due to its strictness, the preferred behavior of web pages was to present an error message to the user in case the code failed to conform to the XHTML schema [27]. Coupled with the fact that XHTML was not backwards compatible, and thus old web pages would need to be updated, XHTML was not really a success: most web pages will be served with the MIME tag defining it as regular HTML (`text/html`) despite XHTML (`application/xhtml+app`) being used [31].

AJAX

AJAX, or Asynchronous JavaScript and XML, is not exactly *a* technology, but rather a collection of technologies working together [16]. It is based on the `XmlHttpRequest` JavaScript object, that allows requests to be made to the server from the client, using JavaScript, dynamically without requiring a reload of the page. The term AJAX was coined in 2005 by Jesse James Garrett in [16], though its use predates the actual term.

Despite the name, requests can be either asynchronous *or* synchronous, and XML does not necessarily have to be involved. What happens when an AJAX-request is made is the following: the browser sends an HTTP request to the host serving the current web page. The server will handle it as any HTTP request and send a response back, which will be stored in the XmlHttpRequest object. Since the request can be either synchronous or asynchronous, the programmer may either handle the request directly or specify a callback that will be executed when the request is finished.

The requests can be in different *states*. The XmlHttpRequest object has a property `readyState` that is set to different values during the request lifetime. The different values are:

0. **Uninitialized** - The initial value as the object is created.

1. **Open** - The `open` method which creates the request has been successfully called.

2. **Sent** - The request has been successfully sent, although no data has been received yet.

3. **Receiving** - Immediately before the response has been received.

4. **Loaded** - The data transfer has been completed. We can check if `readyState == 4` to determine if there are data to be read in our request object.

Below are two examples on how to create and send a request, either synchronously or asynchronously:

```
function getSynchRequest(req) {
  // Here, 'AJAXRequest' is a function
  // returning an XmlHttpRequest object;
  // the way this is done might be
  // browser dependent.
  var request = AJAXRequest();

  if (!request) {
    return;
  }

  // Using the last parameter as 'false'
  // makes this a synchronous request.
  request.open("GET", req, false);
  request.send(null);

  // When the request has finished we return
  // the contents of the response.
  // readyState == 4 indicates that the request
  // has finished and has loaded the response,
  // as indicated in the list earlier.
  if (request.readyState == 4) {
    return request.responseXML;
  }
}
```

Asynchronous request:

```
function getAsynchRequest(req, callback) {
  var request = AJAXRequest();

  if (!request) {
    return;
  }

  // The 'onreadystatechange' property
  // wants a function that will be called
```

```
    // whenever the request reaches a new
    // state. Here we set it to an anonymous
    // function that will call the 'callback'
    // function with the request as a parameter.
    request.onreadystatechange =
      function() {
        callback(request);
      }

    request.open("GET", req, true);

    request.send(null);
  }

  getAsynchRequest("example.html",
    function(request) {
      if (request.readyState == 4) {
        // Do something with the contents.
      }
    });
```

3.6 HTML5

HTML5 introduces a number of new features to HTML, as well as the browser's JavaScript API's. The current draft is not expected to be finished until a few years from now, but there are still many features that are available in more or less functional condition in most of the major browsers.

In June 2004, the World Wide Web Consortium (W3C), the main international standards organization for the World Wide Web, held the Workshop on Web Applications and Compound Documents. At this workshop there were a number of parties interested in furthering the development of HTML. The W3C decided not to participate in such a development and instead continue the development on XHTML [27]. Thus the *WHAT Working Group* [20] was founded. This group started development on HTML5 outside of the W3C [40], though W3C has since then adopted it as well.

When talking about 'HTML5' people are generally not talking about the markup language alone, even though there are many new tags introduced into the language; HTML5 could basically be said to be the conflation of new features in HTML, JavaScript and CSS. [47] contains slides for a presentation detailing much of HTML5's new feature set; the presentation is itself HTML5 however, and as such requires a compliant browser.

A large number of new features were introduced to HTML5 to meet the needs of the Internet applications of today and a few of these will be discussed in this section.

New and Deprecated Tags

HTML5 introduces a large number of new markup tags; they are too many to list here. The more 'famous' of these new tags include the canvas, video and audio-tags which should provide browsers with native support for low-level graphics, and video and audio playback.

HTML5 also provides a number of *semantic* tags, tags that give no immediate benefits but that can be used to layout the web page in a more meaningful way [31]. An example of such a tag includes the `<footer>` tag that might be used as a footer of a web page; traditionally web developers have used the `<div>` tag with an appropriate identifier for most sections of the web page, for example `<div id="footer">` for a footer. These semantic tags can also simplify for search engines and other kinds of web crawlers to analyze web pages.

HTML5 also deprecates a number of tags from older versions of the markup language. These mostly include tags for inline styling such as `<font>` whose functionality are moved to CSS instead [31].

WebSockets

Since AJAX was introduced, it has become widely adopted as the standard for dynamic content on the Web, i.e. to allow for in-browser chat applications and such that work without needing page reloads. The big downside with this is that AJAX relies on the HTTP protocol, as well as the fact that AJAX can not receive updates from the server directly. To make a chat application work, the browser application must continuously poll the server for updates. Furthermore, each message will have the additional overhead of the HTTP headers, resulting in quite a lot of extraneous bandwidth usage [22].

With *WebSockets*, the aim is to circumvent this by allowing the browser to set up a two-way connection with a WebSocket server using TCP. The overhead of HTTP headers is gone since WebSockets uses pure TCP, and both the server and client can send and receive data at any time.

At the time of writing, the support is scarce; while the latest versions of browser such as Firefox, Chrome, Opera and Safari all have implementations, the current existing server applications are experimental at best and the WebSocket API and protocol are still being standardized by the W3C and IETF. Additionally, some browsers does not have WebSockets enabled by default due to security concerns [21]; WebSockets being as new as it is will most likely have problems with security for some time to come before it has reached a mature

enough state.

Local Storage

HTML5 introduces a number of different mechanisms for persistent storage, which can be used by web pages to store information that must persist between reloads, and even between different visits to the page or using the page in offline mode, and so on.

The different proposed mechanisms include *Webstorage*, *Web SQL storage* and *IndexedDB*. Webstorage uses a simple key-value storage [51], similar to cookies. However, cookies are sent in HTTP headers and they expire after some time, while Webstorage are loaded by the browser directly and will persist until explicitly deleted. Web SQL uses an SQL backend that is accessible through JavaScript; however, it appears that Web SQL will not be widely adopted [41] and instead a third technology named IndexedDB was proposed, where only JavaScript is used to access the database rather than SQL.

Web Workers

Web workers are a way for browsers to run JavaScript in separate threads in the background [40]. They can be used for different background tasks such as making lengthy computations or making network connections. The workers are relatively heavyweight and should thus not be used in large numbers or for short amounts of time, but for relatively lengthy tasks [46].

3.7 Browser Graphics

In the early days of the Web, the graphics that could be presented in the browser were simple text and UI components along with images. Thanks to plugins such as Java Applets and Flash, more advanced graphics could be displayed. Today, with the increased performance of JavaScript, browsers are beginning to be able to natively handle high-performing graphics through the use of technologies such as the HTML5 canvas element or WebGL.

3.7.1 Canvas Element

The canvas element was introduced into HTML5 and is a context intended for low-level rendering of graphics [49]. The canvas was originally created by Apple for use in the WebKit HTML rendering engine to create widgets for their Dashboard software [31].

The canvas itself is only an HTML tag that can be added to a page, with all manipulation of it done through JavaScript. If the browser lacks support for the canvas element or if scripting is disabled for some reason, the browser will display fallback content instead of the canvas.

A canvas will initially be blank. To render something to it we must retrieve a rendering context through JavaScript. To get a context, the canvas object has a method `getContext`. To get the standard 2D context, the method should be called with the string `"2d"` as argument:

```
var canvas = document.getElementById("theCanvas");
var context = canvas.getContext("2d");
```

In Figure 3.3 the coordinate system of the canvas element can be seen. The origin is placed in the top left corner and moving down the grid will increase the y-value. 1 unit in the grid corresponds to 1 pixel in the canvas. Note that these are mainly true for the default 2d rendering context from above; if, for example, a WebGL context is used instead then other rules will apply.

It is also possible to apply transformation matrices to the canvas, to resize or rotate the grid or translate the origin.

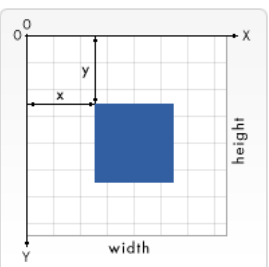

Figure 3.3: **The canvas coordinate system.**

The canvas supports a single primitive shape: rectangles. However, the canvas has many methods for drawing paths which can be used to compose complex shapes. While the canvas has a few high-level method of drawing it also gives access to the entire underlying pixel buffer [35].

3.7.2 WebGL

WebGL is an implementation of OpenGL for the browser. It is based on the OpenGL ES2.0 specification. It aims to provide hardware accelerated graphics

in the browser, but the availability will depend on if the browser has implemented it and if the graphics hardware supports it. WebGL is maintained by the Khronos Group[19].

At the time of writing, the beta/development versions of Firefox, WebKit and Chromium all has support for WebGL, though it must be explicitly enabled [18]. Microsoft, despite aiming for a good HTML5 and JavaScript support in Internet Explorer 9, seems, at the time of writing, to be reluctant to add support for WebGL in their browser [43].

Being based on OpenGL ES, WebGL lacks the 'immediate mode' that regular OpenGL has; for example calls to `glVertex` to upload vertex data to the graphics card are not possible. Instead, vertex, color, texture and normal data must be stored in buffers that are then uploaded to the graphic card in batches. Furthermore, there are no built in lighting or matrix capabilities, everything needs to be done through shaders [2]. This approach is a bit different from many web API's that aim to be easy to use; it allows WebGL to be lightweight and flexible to apply to the many use cases that 3D graphics has [29]. Because of this, WebGL is extremely 'bare bones' and can be difficult to use in itself unless the programmer has a lot of previous experience with OpenGL. Therefore, it can be advantageous if functionality is wrapped in some kind of scenegraph or similar. For information on scene graphs, see Chapter 5. Currently there exists quite a lot of libraries that do this. See Appendix C for a (not comprehensive) list of different libraries.

To create a WebGL rendering context, a regular HTML5 canvas element must be available. If the browser has support for WebGL, then the canvas method `getContext` should accept the string `"webgl"` (case sensitive) as argument; this method then returns an object containing methods and properties related to OpenGL [29]. Past this, working with WebGL will then be very similar to working with OpenGL ES, except in JavaScript instead of the more conventional languages with OpenGL bindings. Details of the inner workings of WebGL can be found at the WebGL specification [29].

In Appendix D the source code for a simple WebGL demo application can be seen, the classic spinning cube. For a screenshot of the demo, see Figure 3.4.

[2] *Shaders* is a technology allowing the graphics card to be programmed, basically. The graphics card uses a pipeline to process the data that is sent to it. Shaders are used to program the different pipeline stages

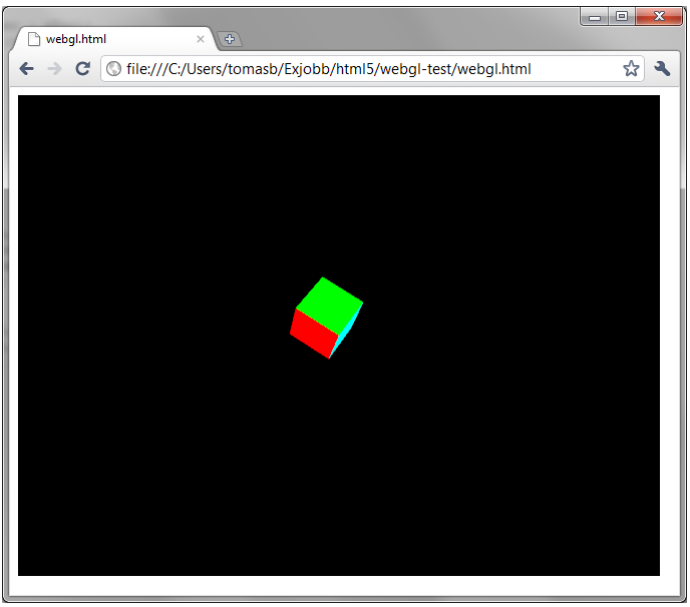

Figure 3.4: **An example WebGL application.**

3.7.3 SVG

An alternate way to display graphics is through the `<svg>` tag. SVG stands for Scalable Vector Graphics and is an XML based image format for displaying vector graphics. The tag allows for inline SVG in a HTML5 document and an upside of using this method is that the graphics will be part of the DOM, something that the canvas element is unable to provide [31]. However, neither of the two methods are in any way superior to each other since they serve two different purposes.

3.8 Conclusion

With HTML5, third-party plugins such as Flash gets a serious contender, since the need for plugins to achieve, for example, high-performing graphics in the browser will be smaller. In many ways, the new features of HTML5 should supersede a lot of current technologies. For example, the HTML5 `video` tag will make the use of flash to display video on a web page unnecessary; faster JavaScript combined with `canvas` tags will also in turn make the use of Flash for say, games, unnecessary.

It will take some time for HTML5 to fully establish itself, especially considering that the draft is *not yet finished*, and that many of today's browsers, while having some rudimentary support, lacks support for a lot of the new features that HTML5 aims to provide. In an interview done in 2008 with Ian Hickson, HTML5 editor, he predicted that the finalized HTML5 draft would not be available until 2022 [26]. In another interview, with W3C interaction domain leader Philippe Le Hegare, he states that he does not think that developers should use HTML5 yet because it is in such an immature state, and that many parts of the draft is still subject to change [30]. While this is true, actually using the technology is a good way to see what works and what does not and it gives developers experience with the technology.

A problem that has been apparent since the world wide web was conceived is the fact that all the different browsers behave ever so slightly different, have a few different features, and they all have different standards compliance. Two tests were created to test the standards compliance of browsers: the ACID tests. For a few years now all browsers have aimed to reach good scores in these tests since standards compliance has come to be a big selling point, but there are still problems with browsers displaying web pages different and developers must often use workarounds to make their pages work across all browsers. Since all major browser vendors are implementing HTML5 from the presented standard, these problems should hopefully be mitigated.

To conclude, these coming years should be interesting for the development of browser based applications; with the HTML5 draft becoming more finalized, HTML5 more widely adopted and the performance of JavaScript increasing, the browser as a serious contender for desktop applications suddenly becomes a reality.

Chapter 4

JavaScript Embedding

The reason for embedding a scripting language often differ depending on the application, though the main uses are probably either for extensions/plugins or to allow for some programmability without requiring access to the real source code/API. In this case we are interested in the latter: using a scripting language it is possible to rapidly create scenes for testing and prototyping.

AgX already has bindings to the Lua scripting language, but one of the motivations of using JavaScript was the prospect of having bindings that would work seamlessly for offline embedded scripting with a scripting language that has similar syntax to the host language, *as well* as being able to use AgX as a plugin for web-browsers, scriptable in the same fashion as regular web-pages are with the same API used for offline scripting.

4.1 Scripting Languages

What are scripting languages and what are they good for? While conventional languages such as C++ or Java are used to write stand-alone applications, scripting languages are often designed for a specific problem domain. They also tend to work in a way as to allow for rapid development: they use dynamic typing, they are interpreted/just-in-time compiled, and often have a rich set of builtin functions and data structures such as associative arrays and lists [42][39].

The line between a scripting language and a 'regular' systems language has blurred somewhat since a scripting language can be, and often are, used for stand-alone applications, though they are still mostly used as 'glue' or extension

languages [42].

4.2 V8

There exists many different JavaScript engines. A few of them include SpiderMonkey from Mozilla, Nitro from Apple, and V8 from Google. For this thesis Algoryx was interested in using V8 since it has been shown to be very fast: V8 actually compiles the JavaScript to machine code rather than bytecode before execution.

Describing V8 in detail is beyond the scope of this thesis but information on V8 can be found at [25] and a short example on exposing a C++ class to the engine is available at Appendix A for reference.

4.3 Related Technologies

There exists a plethora of different solutions for embedding scripting languages. To give some background into this area a few of these solutions are described in this section.

The most popular languages used for embedded scripting seems to be either JavaScript, Lua or Python, though there are a number of other languages that also see some use (Perl, Scheme, etc.).

The API's provided by scripting languages are usually rather low-level, requiring the programmer to write 'glue' functions and a lot of boilerplate code to expose parts of their API. Because of this it is often time consuming to implement scripting support for an API, especially if it was not designed with this in mind from the start. Thankfully, there exists solutions to make this easier, as described below.

Tolua++

Tolua++ takes C++ header files as input and will 'partially' parse them and produce Lua bindings as output [32]. Since C++ is difficult to parse the tolua++ parser only attempts to extract the information required for the bindings from the header files. To help the system with this the programmer should use *cleaned-up* versions of the header files. AgX uses tolua++ for its Lua bindings.

Py++

Autogenerates python bindings by directly parsing C++ code. It uses a program called *GCCXML* to generate an XML representation of the C++ code and then parses the resulting XML using Python. The system developed during the thesis project and described in Section 4.5 is partly influenced by Py++, using the same 'toolchain.' Py++ generates code for Boost.Python [9] which is a library that heavily uses C++ template metaprogramming techniques to achieve an IDL[1]-like interface for bindings C++ classes and functions to Python [10].

SWIG

SWIG is a tool/library that aims to provide an easy method to create bindings to several different languages. It uses an IDL that should contain all the declarations that the user is interested in exposing to a scripting language. The IDL used by SWIG is intended to let the programmer create the bindings while making virtually no changes to the original source; it should be as simple as copying the required source to the interface files, though some manual changes will be needed depending on the complexity of the code [12].

4.4 The Good, the Bad and the Ugly

The circumstances for success depends on several things. Preferably we want a language that does not differ too much from the source language, and one that allow us to export the API in a one-to-one fashion, as not to cause confusion for the user or require new documentation. While it is natural to assume that a scripting language will differ quite a bit from a language like C++ there should still be some fundamental similarities. Thankfully, the syntax of JavaScript is very similar to that of C++.

Syntax is not enough, though. C++ is a very complex language and has many features that does not exist in JavaScript. While this fact remains for a lot of other languages that has been proven to work well for embedded scripting it is still a factor that must be taken into consideration; how should the JavaScript API map to the C++ API when there are some fundamental semantic differences between the languages.

Finally, exposing API's for embedded scripting is often an messy business, as demonstrated in Appendix A. While it is certainly possible to export a large API by hand, it would be a very dull, time-consuming business. For many of the established languages there exists tools to make this easier, even allowing autogeneration of bindings, see Section 4.3. Since V8 is a relatively new engine, there does not yet appear to be any way to autogenerate bindings. A tool for

[1] Interface Definition Language

this purpose was developed during the thesis work, see Section 4.5.

4.5 Pipeline for Autogenerating Bindings

From the early experience working with V8 it was clear that it should be possible to automatically generate the bindings (see Appendix A). This section describes the efforts to write a tool for this purpose, called the *pipeline*. The aim of this tool was a system that could read C++ header files as input and, maybe with a little additional help from a config file, automatically produce working bindings allowing access to the code in the header from JavaScript.

For information on actually using the tool, see Appendix B.1.

4.5.1 Motivation

At the time of writing, AgX uses Lua for embedded scripting. The Lua API is a fairly one-to-one mapping where it mirrors parts of the AgX API; these bindings are autogenerated through tolua++ (see Section 4.3 for a description of tolua++). Considering that the already existing bindings are autogenerated it would be unacceptable for a new language to require writing by hand, not to mention that it would be far too much work unless the desired target API would differ substantially from the C++ one, which is not the case here.

4.5.2 Alternatives

The aim of the pipeline tool was to get nearly full autogeneration with little outside help from the programmer; though this is not necessarily the 'best' approach. Scripting languages engines generally provide rather low-level APIs to create bindings and using these to export an API, especially a large and complex one like AgX, is hardly an alternative. Here we shall look at a few approaches that *would* be decent alternatives.

Cleaned-up Header Files

One approach is to use *cleaned-up* header files, rather than the header files as-is, as input for the code generator. The idea is to strip all information from the header files except that which is intereseting for the code generator. This is mostly done for ease of parsing the headers. C++ is a complex language that

is difficult to parse and these cleaned headers makes it simpler to implement a 'partial' parser. This also makes it possible to include additional information in the headers that might be good to have for the script bindings. Tolua++ uses this approach.

Template-based Library

The template feature of C++ allows the programmer to create very powerful abstractions. Since the API's for scripting languages are usually rather low-level and requires a lot of manual work, with this approach a higher-level, template-based API would be written to hide all of the arduous details. While using a template-based approach is not strictly necessary it can lead to very clean-looking code. Examples of this approach include luabind [44] and Boost.Python. Since it still requires all bindings to be written by hand, albeit in a much easier and quicker way, this was not really considered, though for smaller API's it could be a good method. Following is an example of how Boost.Python code can look like [10]:

```
#include <boost/python.hpp>

struct World
{
  void set(std::string msg) { this->msg = msg; }
  std::string greet() { return msg; }
  std::string msg;
};

BOOST_PYTHON_MODULE(hello)
{
  class_<World>("World")
    .def("greet", &World::greet)
    .def("set", &World::set)
  ;
}
```

The code exposed a class `World` to Python in the `hello` module/namespace. A system similar to this but for V8 was developed as an learning experiment. This system was called 'wiz8'. The current implementation is faulty and not usable other than for demonstration purposes. The class `World` above could be exposed to V8 in the same fashion with the following code:

```
wiz8::registerClass<World>("hello", "World")
  .constructor()    // Default constructor
  .def<void, std::string, &World::set>("set")
  .def<std::string, &World::greet)("greet")
  .seal();
```

The syntax is a bit different from Boost.Python. In this system, the `def` methods requires template arguments stating the return type of the method, the types of the method arguments, and finally a reference to the method itself. The only argument passed to the `def` method itself is the name that the exported method should have in JavaScript. All of this works through a concept that was called 'chains'. The `registerClass` function creates a new `Chain` object of the appropriate class type. The methods of the `Chain` class returns new 'nodes' in the chain that 'inherits' information from previous nodes. Finally, the `seal` method will register the created class with a namespace that can be supplied as an argument to the `registerClass` method, to finalize and close the chain. See Figure 4.1 for an overview of how the system operates.

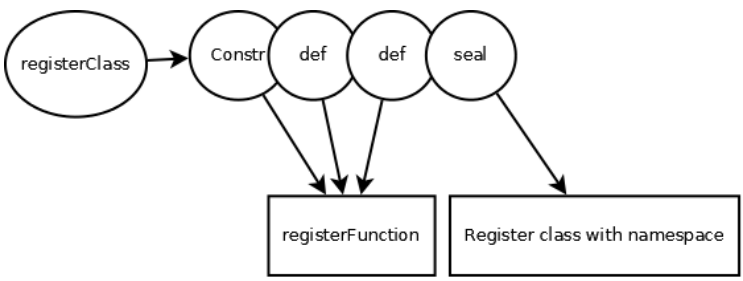

Figure 4.1: **The 'wiz8' class-registration chain.**

At the heart of the template based system there is a lot of code to discern types of variables and functions, and to be able to export functions correctly we need to know exactly how many arguments the function accepts. Due to limitations of C++ templates there is a lot of duplicate code required. Consider a function that takes a (generic) pointer to member function. The syntax for pointers to member functions is the following in C++ [13]:

```
R (T::*)(A1, ... An)
```

where `R` is the return value, `T` is the class name, and `A1` to `An` is the types of its arguments (if any). Then, a function accepting pointer to a generic method with a single argument might look something like this:

```
template <typename R, typename T, typename A1>
void foo(T* obj, R (T::*funcName)(A1)) {
  R ret = obj->funcName(new A1);
}
```

Now, in order to be able to have a `foo` function that can accept functions of any number of arguments we require a function for every possible number of arguments due to the aforementioned limitation. Wiz8 also has a few layers of functions that need to know the number of arguments in order to work correctly, which quickly causes the size of the source code to be quite large. The next standard of C++, C++0x, has a feature called *variadic templates* that should be something to help these kinds of systems, since they allow templates to have an unspecified number of arguments [6].

Interface Files

A third approach is to use 'interface files' that state what should or what should not be exported along with any extra information required. This was used by the code generator at the early stage of development but has since been abandoned in favor of just automatically generate the code directly, though there are still config files used to help the codegen with different things. An example of a system using this approach is SWIG.

4.5.3 Implementation

The pipeline uses four main components: GCCXML [3], PyGCCXML [52], the code generator written in the Python programming language, and finally the V8 interface written in C++.

GCCXML

GCCXML is a program that uses the codebase from the GNU C Compiler to generate an XML representation of C/C++ code. Below is an example of XML generated with GCCXML:

```
<?xml version="1.0"?>
<GCC_XML cvs_revision="1.135">
  <Namespace id="_1" name="::" members="[...] _132 [...]"
    mangled="_Z2::" demangled="::"/>
  <Namespace id="_2" name="std" context="_1" members=""
    mangled="_Z3std" demangled="std"/>
  <!-- [...] -->
  <Variable id="_132" name="foo" type="_142"
    init="0" context="_1" location="f1:1"
    file="f1" line="1"/>
  <!-- [...] -->
  <FundamentalType id="_142"
    name="int" size="32" align="32"/>
  <!-- [...] -->
```

```
  <File id="f1" name="foo.c"/>
</GCC_XML>
```

from a C file containing only the code

```
  int foo = 0;
```

The `[...]` in the XML snippet above are just some manually stripped code created by GCCXML representing builtins and different types. The information interesting for the small example code snippet remains. Each tag is given an identifier; for example, the global namespace '`::`' is given the identifier `_1`, and each entity that belongs in the global namespace has an attribute `context` with the value `_1`. In this way, tags can reference other tags through their identifiers. A parser would create a table mapping identifiers to table. Thankfully, since PyGCCXML existed no new parser was necessary.

PyGCCXML

PyGCCXML is a Python package that basically wraps GCCXML and parses the resulting XML to provide an in-memory representation of the code. Finally, the code generator is written in Python to take advantage of PyGCCXML, but also because Python allows for a rapid prototyping process. The V8 interface is a C++ program that starts the V8 virtual machine and loads the bindings produced by the code generator.

Pipeline

As the name suggests, everything is organized into a pipeline of sorts:

1. XML Generation
2. Parse XML
3. Code Analyzation
4. Code Generation
5. 'Merging'

The first two stages are done through PyGCCXML. The third and fourth stages are done with the Python program. The final stage is a bit special and exist mostly due to the way the codegen works at the time of writing. When run it will produce a set of files for the desired bindings, containing the language bindings and initialization code that is run on startup. If several bindings are wanted, it will create these files for all the bindings, resulting in a lot of files.

The 'merging' step creates a single file that will include all these files and run their startup code, essentially merging them together.

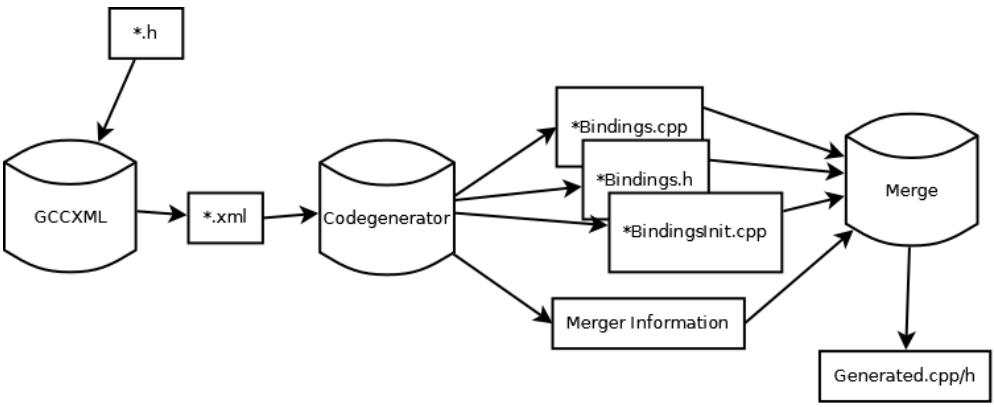

Figure 4.2: **Overview of the codegen system.**

In Figure 4.2 an overview of the system can be seen. A header file is fed to the system which invokes GCCXML, producing an XML file as output. PyGCCXML will parse this file and produce objects representing the code. The codegen then produces three files containing the bindings, the binding declarations, and a file containing a function that is run on startup. It will also write to a file that the merger uses to determine which files it should merge together. Finally the merger can be run which uses the information file to produce a final C++ source and header file containing a function that will run all of the initialization functions, exposing the bindings to V8.

V8 Interface

This is the C++ part of the system; it contains classes, functions, and macros to provide a more high-level interface to V8.

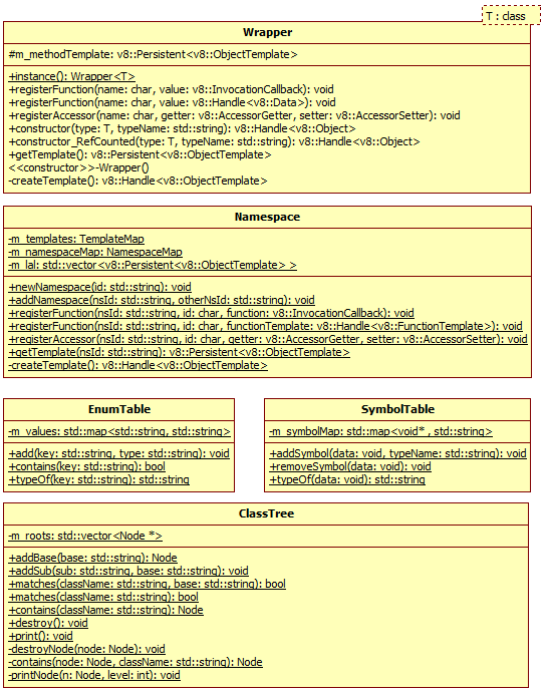

Figure 4.3: **UML diagram of the V8 interface.**

Figure 4.3 shows an overview of the important classes in the V8 interface. What is not shown is different macros and a few utility functions to help with readability and to make it easier to work with the V8 API. The interface is written in such a way that it should be usable even without autogenerated code.

The most important class is the `Wrapper`. It is a templated class that enables JavaScript objects to be instantiated of the template type. It is a singleton class because of the way the V8 API is constructed. Example usage of the class:

```
// This sets up so that the class will be
  // available from JavaScript. No methods
// etc. available yet though.
Wrapper<Foo> wrapper = Wrapper<Foo>::instance();
// This registers a method to be available
// on the object from JavaScript. 'testGlue'
// should be a function that calls the real
// class method.
wrapper.registerFunction("test", testGlue);
// The constructor method returns a V8 object
```

```
// wrapping an instance of the templated class;
// this should be used to actually get an instance
// of the object to V8.
instance = wrapper.constructor(new Foo(), "Foo");
```

The `Namespace` class is basically used to create objects similar to the (for example) JavaScript `math` object, i.e. objects that act like namespaces. It is a static class, also due to some implementation details of V8 that requires us to have only one of some objects.

```
// Creates a new object called 'io'
Namespace::newNamespace("io");
// Register a funciton called 'print', allowing
// us to call io.print(...);
Namespace::registerFunction("io", "print", print);
// Also allow nested namespaces:
Namespace::newNamespace("system");
Namespace::addNamespace("system", "io");
// Allows us to call: system.io.print(...);
```

The other classes are used for different mappings that the code generator requires to deduce types of objects. The `EnumTable` class maps enum values to their type, that is if we have:

```
enum Test { VALUE0, VALUE1 };
```

Then the `EnumTable` should map the strings `"VALUE0"` and `"VALUE1"` to `"Test"`. These mappings are done by the code generator and are used to determine if a string passed into a function is an enum value.

The `SymbolTable` class maps an object to a type, used to determine types of objects and to allow a `typeOf` function in JavaScript. This table is populated in the `constructor` method of the `Wrapper` class.

The `ClassTree` class contains a tree with inheritance information, populated by the code generator at startup. It is used to determine if a certain type inherits from another type to get correct behavior if the arguments of a JavaScript method has a class that inherits from another class. To explain more clearly: when a glue function created by the code generator is called, it will do type checking on all the arguments sent into the function to see if they match. On classes, this is done through the `ClassTree`, which can answer any inheritance-related query if the codegenerator has registered the information. If we have something like the following exposed to JavaScript:

```
class A {};
class B : public A {};
```

```
void foo(A a);
```

then `foo` should accept variables of both type `A` and `B`, and the `ClassTree` allows us to make this work.

4.5.4 Problems and Solutions

There were quite a lot of C++ features identified that might be problematic if we want an (nearly) equivalent JavaScript API:

- Pointers, references or regular values
- Arrays
- Garbage collection (not a C++ feature but still a problem)
- Non-visible destructors
- Overloaded methods/methods with default arguments
- Polymorphism
- Templates
- Operator overloading

Following will be each of these presented in turn, stating what the problem was and how or if it was solved.

Pointers, References or Regular Values

Problem: Pointers and references are similar, but not equivalent. On the JavaScript side, objects are passed by reference, and all native C++ objects are wrapped in pointers on the JavaScript side. The problem lies mostly in distinguishing pointers/references from regular objects and how they should be treated, but also in how the subtle differences between pointers and references should be handled.

Solution: C++ functions taking references or pointers as argument is not a difficult problem; during the code analyzation stage of the pipeline we can detect whether the argument is a pointer or a reference. All C++ objects are wrapped in pointers on the JavaScript side, so if the argument to the

C++ function should be either a copy or a reference, the pointer can simply be dereferenced (i.e. get the value the pointer points at) as we pass it to the function. If the argument should be a pointer we can pass the object as normal.

Returning values from functions works in similar fashion. When returning pointers or references we should be able to alter the value from JavaScript later, and thus we simply return the value from the C++ function as normal after it has been wrapped in a pointer. If it is a value, however, a copy should be returned and thus a new object must be constructed from the return value.

Arrays

Problem: While JavaScript has native support for arrays, this is still a rather difficult problem due to how the way GCCXML works. In GCCXML arrays are detected as pointers, since that is basically what arrays are in C/C++. This fact makes it difficult to determine the code that should be generated; V8 has objects representing JavaScript arrays that would be used but first it must be determined if the type should be treated as a pointer or as an array.

Solution: This is not solved in the current version of the pipeline, mostly due to not being a high-priority issue. A possible solution is to check if the method has received an array as argument, then check if the function accepts a pointer and finally check if the contents of the array matches the type of the pointer.

Garbage Collection

Problem: JavaScript is a language with built-in garbage collection but the native C++ objects will not be removed correctly by this. The wrappers holding the objects will be removed but the memory for the pointers will not be freed and thus our program will leak.

Solution: The V8 engine stores JavaScript objects in *handles*. Handles can be *local* or *persistent*. Local handles are removed as they fall out of scope while persistent handles will stick around. Persistent handles can be made *weak*, causing a callback to fire when they fall out of scope and no other handles refer to the stored object. Using this callback objects can be deleted when nothing is referencing them, allowing for garbage collection.

Non-Visible Destructors

Problem: A problem relating to garbage collection is when class destructors are non-visible. This is done when objects should not be allowed to be deleted in common ways (i.e. calling `delete foo;`). This can be due to several reasons, but a popular one is when the objects are *reference counted* (or *refcounted*). This is the case in AgX. Refcounted objects keep track of how many times it has been referenced, and instead of deleting the object its refcount is decremented. When the refcount has reached zero, the object can delete itself.

Solution: This is a tricky problem, and it is very hard to get a general solution that is acceptable. Currently there is only support for the AgX refcounted objects. If, during code analyzation, a class is detected with a non-visible destructor, the instances of the class will be assigned a different callback when their handles fall out of scope, decrementing their reference count instead of deleting them. When the objects are created or referenced in some way, their counter will be increased as well. However, as stated, this currently only works for the AgX reference counted objects, which is a rather large problem. Since this was not the main focus of the thesis work no further attempts to solve this was made, but for future work it would be a high priority issue.

Overloaded Methods and Default Arguments

Problem: JavaScript does not support either overloaded methods or the concept of default arguments (i.e. `void foo(int x, int y = 0)` can be called with either `foo(0)` or `foo(0, 1)`). The two concepts are closely related as a method with n default arguments can be considered the same as $n + 1$ methods with the same name.

Solution: This is not a trivial problem and there are probably many different solutions to it. It could also have been accepted as a limitation of JavaScript and thus been ignored. However, a solution was attempted.

Each 'glue' method in C++ are given one 'base' method and a number of 'sub' methods. The base method will be called from JavaScript. The sub methods are mapped to (in theory) a pair $< n, a >$ where n is the number of expected arguments and a a list of argument *types*. This way, the correct sub method can be called by analyzing the types of the arguments passed to the base method. To determine the types of the arguments, each object created through JavaScript will be inserted into a table together with their corresponding type, which can then be queried for run-time type information (RTTI). C++ actually has native support for RTTI through the `typeid` operator but since the objects passed to C++ from JavaScript will be of type `void *` this will not help here, unfortunately.

Note that this solution introduces a new problem: if a method accepts a base class `Base`, and it receives a subclass `Sub` as input, it will fail to call the correct sub method since the subclass will have the wrong expected type. A solution to this is presented below.

Polymorphism

Problem: If a C++ method takes a class `Base` as argument, it should also accept a class `Sub`. The way support for overloaded methods were implemented in the pipeline requires a way to determine if a type inherits from another type, so the correct method can be called with the correct arguments.

Solution: During startup an 'inheritance tree' is created for all types. The trees contain the inheritance information for all types, and as such can be used to determine if a certain type is related to another type. These trees are queried when methods are called to determine if the type of an object is a subtype of the required type. This will cause a small performance hit when calling methods, unfortunately.

Templates

Problem: C++ templates require an instance to be usable, and GCCXML can not currently resolve template instances on its own, so to speak. Because of this, for the system to be able to export a, say, `std::vector<Foo*>`, it must find an instance of this somewhere in the code it is parsing.

Solution: There is currently no good solution implemented, but one way to deal with templated classes is to create 'proxy classes' that contains an instance of the template, allowing the code generator to export the proxy class instead of the real class. Currently to export a templated class GCCXML must parse a file where it is used, but not where it is defined, which is rather unintuitive.

Operator Overloading

Problem: JavaScript does not support operator overloading at all which causes big problems when exporting for example vector classes that has operators for dot and cross products, etc.

Solution: All the overloaded operators will be renamed to their 'textual' version, so '+' will be renamed to `addOp`, so in JavaScript we could write: `foo.addOp(4);` instead of `foo + 4;`. Since some operators are overloaded in

such a way that causes their default names to not make sense in the context, the aliasing feature of the config file format can be used to rename an operator, e.g. '*' could be named 'dot' for a vector class.

4.6 Conclusions

JavaScript

JavaScript as a language is similar enough in syntax to C++ that using it as a scripting language is rather attractive. It will be an easy process to go from one language to the other and the two API's (the C++ API and the scripting API) will look rather similar. Additionally, the prospect of having a scriptable API that would work seamless for offline and online scripting is very interesting (though there are difficulties with this, as described in Chapter 5).

Autogeneration

Despite a number of difficulties, it is possible to create a system that autogenerate code in an adequate fashion, as shown here. The problem with autogeneration lies mostly with parsing and getting a good representation of the code: finding a good way to do this would solve a big part of the problem.

'Pure' autogeneration as done here might not be the best approach if the tools for parsing and analyzing the code are faulty. Writing a simple parser that can find class declarations, method declarations and so on should not be too difficult and then a system similar to tolua++ could be used. Since the actual code generation part is decoupled from the parsing/analyzing part in the pipeline, this can be changed to use something different than GCCXML if desired.

GCCXML

Using GCCXML was, in hindsight, most likely a mistake even if it allowed for a working prototype implementation. It has too many bugs and other faults that makes it impractical to use for future development. Because it is basically a compiler, it will follow `#include` directives and so forth, something that is both good and bad. The problem is that it might delve into some system header which it can not understand and thus fail 'compilation.' It also fails to understand some code that regular compilers will handle, like DLL export directives. In this case something that *only* works on the specified header file, as in tolua++'s case, would be preferred. A piece of software that simply parses the C++ file given, not really caring to understand what can be found there and give a good representation of it would be preferred to GCCXML's all-or-nothing approach.

Chapter 5

Physics in the Browser

The increased performance of JavaScript and graphics in the browser could lead to many interesting developments in the browser as a visualization and simulation platform. In the physical simulations area there are ports of libraries such as box2d (see Chapter 2.5) which is a 2D physics engine. 3D physics require a lot more performance in general, however. While it should be possible to create a simple 3D physics engine in JavaScript, a full featured engine like AgX would in no way be feasible.

In this chapter the prospect of using AgX as a browser *plugin* is explored. This way, the power of the engine could be incorporated in the browser with the full performance of a regular C++ application, while having it scriptable in JavaScript, exposing the engine for use as a browser application.

5.1 Motivation

Because HTML and JavaScript in the browser provides a set of user interface widgets and a number of different libraries available for plotting and so on, using the browser as a platform for testing and demonstration purposes of AgX is attractive. Generally using the browser as a user interface for simulations were also a big motivation for this.

5.2 V8 vs NPAPI

Even if Google Chrome uses V8 as their JavaScript engine, NPAPI is still what must be used for plugin writing since V8 is just a library that projects can link with to take advantage of embedded JavaScripting. This does naturally complicate things if we want to have both JavaScript bindings for offline scripting as well as the plugin API. While it certainly is possible to maintain both, it does not seem very cost-effective. The path taken in this thesis is to make the plugin a separate entity but ways to combine the two are discussed in Section 5.8.

5.3 Firebreath

Since NPAPI, the Netscape Plugin API, is rather low-level and scarcely documented, some kind of framework for plugin development was required for rapid development. For this purpose, a framework called *Firebreath* [11] was used. It is a C++ framework that encapsulates not only NPAPI, but also ActiveX, allowing for multiplatform, multibrowser plugins to be easily developed. In fact, simply creating a new project with Firebreath will yield a fully functional plugin, ready to be run in a web browser.

The Native Client SDK that was touched upon briefly in Chapter 3.4 might also have been interesting to look into, though it is only available for Google brand browsers at the time of writing.

5.4 Autogeneration?

A question to ask is how much of an API is desired to expose to a browser plugin. Autogeneration of code to expose a large part of the API has been proven to work in Chapter 4. But the larger part of the API that gets exposed, the larger the plugin will become (in file size), and the complexity will increase (but complexity is often necessary in large and powerful API's) so the question is how much functionality that should actually be available to a browser plugin.

The assumption made here is that when creating a scene in the browser it should be rather simple to demonstrate a bit of the functionality and allow a user to play around a bit with the engine. For this, we allow the plugin to wrap the engine on a higher level, providing a number of methods to create bodies and so on that will be automatically added to a simulation running in the background. Complex functionality will mostly be hidden. Thus the bindings were written by hand in order to get more control over the resulting API.

5.5 Prototype Plugin

The prototype plugin was developed to provide a demonstration of AgX as a browser plugin. It is a separate application from the system developed in Chapter 4. It was decided that instead of an approach that required a large part of AgX exposed to JavaScript, either through autogeneration or handwriting, a more high-level API would be more suitable for the plugin.

The motivation for this was partly to get a prototype plugin developed quicker and partly because, since there were no uniform way of using the V8 bindings and the NPAPI bindings, a different approach should be taken for the plugin and the embedded scripting system.

The plugin exposes a few functions to the browser, allowing for creation of objects and object materials. Creating objects automatically adds them to the simulation; the plugin does not allow JavaScript direct access to objects since this could lead to problems with garbage collection and memory leaks.

In this report, when talking about the 'plugin,' it will generally mean both the plugin and the demo web page that implements a small web application running the plugin itself.

User Story

The *user story* is a short description of what the user wants to achieve, and is thus a short and rough description of features that the finished application should provide. For the plugin, it is the following:

As a user, I want to surf to a web page to view/use a physical simulation. The simulation should be interactive in some way, i.e. I should be able to add objects to the simulation, change the properties of objects, and the camera should be movable. I should also be able to view simulation data in some way, through plots or other means.

Structure

The plugin consists of a number of 'proxy' classes that are meant to represent AgX classes that are exposed to the browser JavaScript, as well as a single 'API' class (called `AgXAPI`) that represents the plugin object itself, providing methods to initialize new simulations and so on. Firebreath has a class called `JSAPI` that classes should derive from in order to be available in JavaScript. In the plugin a convenience class called `ClassBind` was created, which is a templated class that derives from `JSAPI`. The purpose of this class is to allow each proxy class to derive from it while also storing an instance of the 'real' object: in the constructor of the proxy classes, the constructor of `ClassBind` is

also called with a pointer to the real object as argument. As an example, the following code shows a rigid body proxy class:

```
class RigidBody : public ClassBind<agx::RigidBody> {
public:
  // When creating a new proxy, pass a pointer to the
  // 'real' object in as argument.
  RigidBody(agx::RigidBody* body) :
    // Call the constructor of ClassBind
    ClassBind<agx::RigidBody>(body)
  {
  }
  // [...]
};
```

In the plugin, the rigid body proxy class is used as a base class for spherical bodies and box shaped bodies. See figure 5.1 for a brief overview of the system.

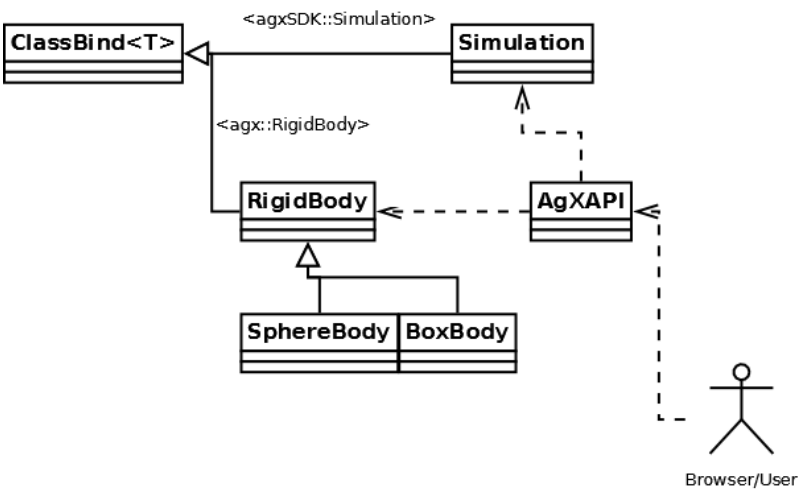

Figure 5.1: **UML diagram of plugin. Details are omitted.**

Rendering

Rendering of plugin contents are generally done by the plugin itself. For example, Flash does not rely on any outside code for rendering. This is done by claiming a rendering context in the browser and then use whatever rendering mechanism provided by the operating system to render to this context.

However, one of the goals of this project was to test and evaluate WebGL for

hardware accelerated graphics in the browser, so that was the approach used for the plugin in this case. The pros and cons of both methods are given below.

- Through Plugin

It is possible to set up a context in the browser to which we can render from the plugin. Firebreath does not seem to have any cross-platform way of doing this without requiring a lot of extra work if we want a plugin that works on several platforms. However, the result will be more efficient than the alternative described below since JavaScript code will simply be slower than C++ code. Since the graphics should be hardware accelerated either way the difference might not be too big in the end though.

- Through WebGL

If the plugin does not rely on graphics it can be used in a more general fashion. Simulations can be run for the purpose of producing statistical data only, for example. It is also possible to use whatever rendering capabilities the browser possesses to display the simulation. In the past this has been rather limited and very slow, but since the introduction of HTML5 the rendering capabilities of the browser has been greatly enhanced. WebGL provides hardware accelerated graphics if the computer has graphics hardware that supports it.

For the prototype plugin a WebGL scenegraph called *SceneJS* [28] is used which produces acceptable results, though the performance is not the best in scenes with many objects. See Section 5.6. A scenegraph is a data structure used to organize different facets of rendering, i.e. geometries, state changes, transformations, and so forth. For an example of a typical scenegraph, see Figure 5.2.

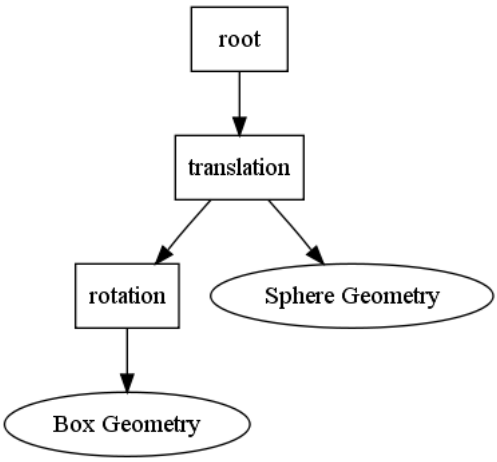

Figure 5.2: **Example of a scenegraph. Traversing the graph will apply the transformations to all subsequent nodes.**

A problem with the WebGL approach is that it is not yet widely adopted in all browsers; while most browsers apart from Internet Explorer 9 will have WebGL support, currently mostly the development versions of browsers actually do have it, and if they do it is not guaranteed that it is enabled by default. Also, considering that older browsers do not support WebGL, this already alienates a large portion of potential users.

Plotting

An interesting prospect was to get real-time plotting of data from the simulation, e.g. forces, energy in the system, and so on. Plotting libraries written in JavaScript for the browser would be easy to integrate into the page with the plugin. There exists a few of these for JavaScript and a library called *Flot* [2] was chosen in this case; no real comparison between different plotting libraries were made because it was not considered very important. Flot was easy to use and setup and creates good looking plots, as seen in Figure 5.3.

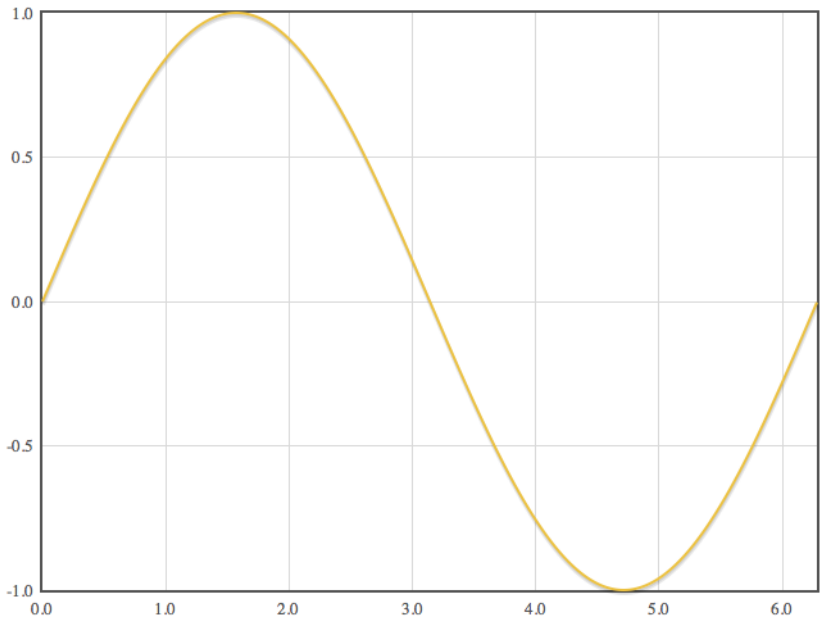

Figure 5.3: **Screenshot of a sine curve plotted with the *flot* JavaScript plotting library.**

5.6 Performance

Performance was not the number one concern in the prototype implementation produced here but it is still a very important factor. If the performance would turn out to be too low, the possibility for further development is slim.

SceneJS

To move an object, its transformation node in the graph must be updated and it seems this is a rather heavy operation on the scenegraph. SceneJS does not appear to be designed with dynamic content in mind. The update/rendering loop is basically:

```
simulation.stepForward(); // Step the AgX simulation
updateScenegraph(); // Update transformations of objects
                    // in scene graph.
renderSceneGraph(); // Render the contents of the scene
                    // graph.
```

where the call to `updateScenegraph()` appears to be the bottleneck. Measuring the time for different calls shows that `updateScenegraph()` takes a substantially longer time compared to the rest of the rest of the calls, which can be seen in Figure 5.4.

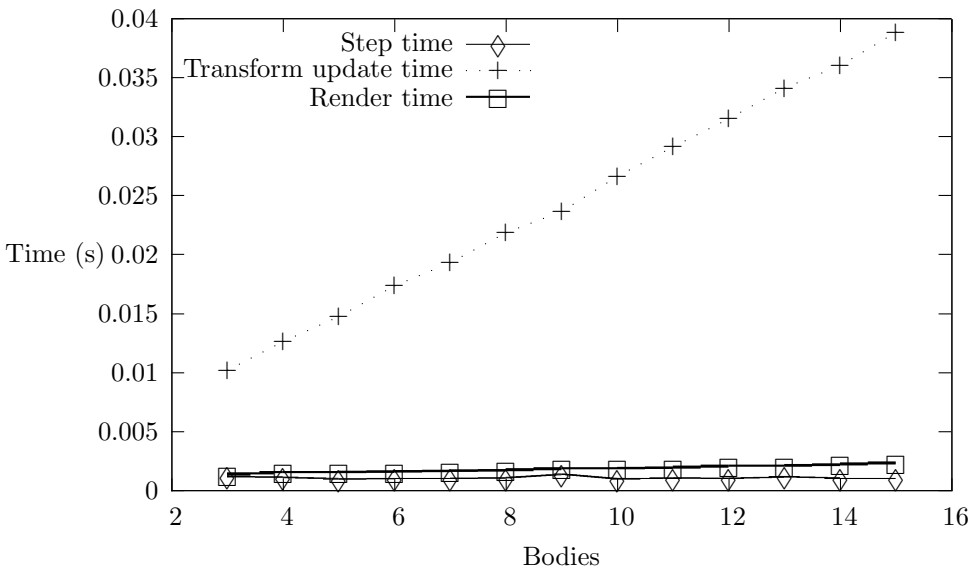

Figure 5.4: **Performance of updating SceneJ scenegraph**

How the overall FPS of the rendering is effected by this can be seen in Figure 5.5.

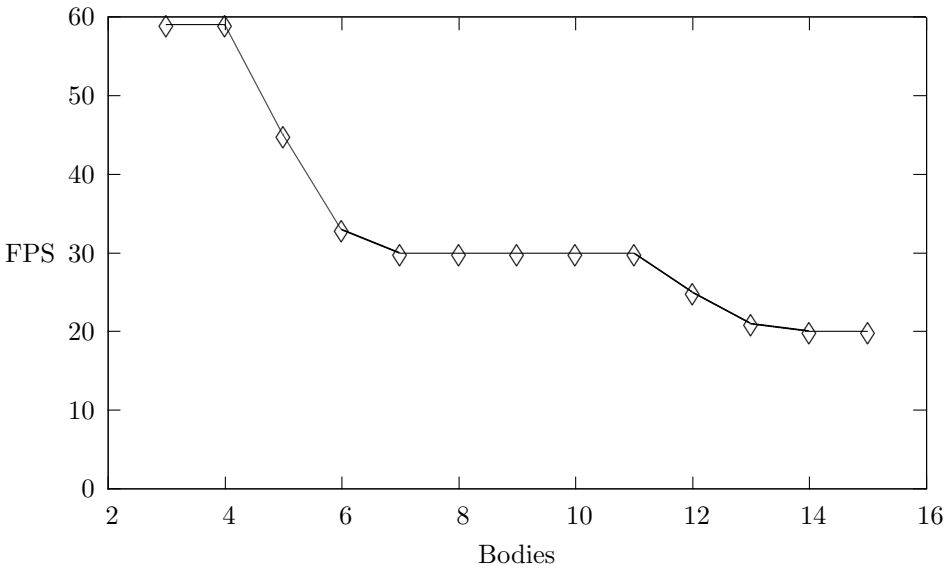

Figure 5.5: **FPS of the system depending on no. bodies**

WebGL

As seen in Figure 5.4 the rendering itself is relatively fast. The reason for the bad performance of SceneJS lies with the scenegraph implementation; the OpenGL rendering through WebGL is very fast. Considering that the rendering side of the plugin is entirely JavaScript, SceneJS can be replaced by something that might be more adequate for dynamic content.

5.7 Installation/Deployment

If the plugin were to be developed further and become a generally usable piece of software, it requires a painless installation procedure for the end-user. This is not something that was really looked into during the thesis work since the resulting plugin would only be a rough prototype.

How the plugin should be installed largely depends on the platform, but it does lead to some problems. AgX uses a system of plugins itself to deal with collision detection etc., and for it to work correctly these plugins must be loaded together with the core physics engine. By default AgX searches in the current

working directory for a folder called 'plugins.' The current working directory will in this case be the directory the browser was launched from, which will differ largely depending on system and browser.

Taking Windows as an example, the installation would have to register the plugin in the registry which would make it available to the browser installed on the computer. Then the installer could, for example, ask the user where the extra plugins should be installed and then write that location into the registry so that the browser plugin can find these later. There should be a number of ways to solve this but they will all depend on the platform where the plugin is installed.

5.8 Conclusion

At the start of the project one of the aims was to get a script integration that would seamlessly work both online and offline, however, during the thesis work the conclusion was reached that this would be a rather difficult goal to achieve due to the fact that one API is used for offline scripting (V8 in this case) and another one for online scripting (NPAPI/Firebreath). There is a possible workaround that is rather clunky, used by a (now deprecated) plugin for 3D graphics in browsers called o3d [17]: Embed V8 with the autogenerated bindings produced with the pipeline software in the plugin and then have functions where strings of JavaScript code can be passed to the plugin and be evaluated, so we would have something like (in the web page JavaScript):

```
  var code =
   "var vector = new agx.Vec3(1, 2, 3);\n" +
   "var body = new agx.RigidBody();\n" +
   "body.setTranslation(vector);\n" +
   "simulation.addBody(body);";
plugin.evaluate(code);
```

Since NPAPI has access to the DOM of the web browser, it should also be possible to set up special script tags on the Web page that the plugin can find upon startup, read the contents of and execute:

```
<script type="agx/javascript">
  var vector = new agx.Vec3(1,2,3);
  var body = new agx.RigidBody();
  body.setTranslation(vector);
  simulation.addBody(body);
</script>
```

The problem with this is that the embedded JavaScript does *not* have access to the DOM as the browser JavaScript does (unless we explicitly add methods to access the DOM through NPAPI to the embedded V8 engine). So that would require the page to have both these special tags and the regular script tags, since to, for example, render the page through JavaScript the plugin must expose some methods to get the contents of the simulation or similar. Furthermore, after some minor tests to try to implement this, it quickly became clear that using this approach would make it difficult to update and render the contents of the simulation through the browser. The reason for this is that in the current plugin, the simulation is updated using a JavaScript function called `setInterval` which calls a function (in this case an update function) continuously every number of milliseconds. To get the same behavior, this function would have to be reimplemented on the plugin side to be called from V8. Additionally, the plugin side JavaScript would basically be entirely decoupled from the browser side JavaScript, something that is not really desired since this makes it more difficult to get statistics about the simulation for plotting and so on.

In conclusion, this chapter demonstrates a way to use AgX as a browser plugin by wrapping functionality in favor of a higher-level, easier to use API. The plugin can be used as a test-bed for simulations or for demonstration purposes. The current plugin is very much a proof of concept but if further work is done there is nothing that would prevent it from being used as a component for various physical simulations in the browser. Of course, it is up to Algoryx to decide which course they want to take and if they want to put further work in this area.

Chapter 6

Results

The fundamental result of the thesis work are the two prototype implementations of the V8 code generator and the AgX browser plugin. These two pieces of software demonstrates the possibility of using JavaScript together with AgX, as well as the possibility of using the physics engine in a web browser with rendering done in WebGL. Since they are only prototypes they are in no way definite solutions, but demonstrate plausible directions that either project could go.

Both of these two implementations are not without problems. A rather large fault of the code generator is due to GCCXML. Since GCCXML works as a regular C/C++ compiler in many ways, it will fail 'compilation' in the same fashion as a regular compiler when it encounters something it does not understand. Due to faults/limitations of the system, this can happen rather often. AgX has a class called `ExampleApplication` that creates an example simulation with graphics; attempting to run the code generator on this to export this class to get visualization with the JavaScript bindings will fail because of different OpenGL data types that GCCXML will fail to understand for some reason.

As often stated in the report, a desired result is to have the V8 bindings and plugin work together in a uniform fashion. Unfortunately, no good way of achieving this was found which can be seen as a bit of a failure. However, the high-level API exposed by the plugin worked really well, so the plugin itself could be considered a success, and it might be in this direction that further work could be headed.

Chapter 7

Conclusions

The goal was to evaluate the possibility of a JavaScript API for AgX that would work both for offline scripting (i.e. as an embedded scripting language) and for online scripting (i.e. as a plugin for web browsers, also scriptable in JavaScript using the same API as for offline scripting). While it is certainly possible to have *both*, the result would be clunky and a bit strange, since bindings would be required for first the embedded JavaScript engine and second for NPAPI.

Furthermore, if the same API was desired for both the embedded scripting API and the plugin API, there could be two ways to approach that. Either, a way to autogenerate the code for both V8 and NPAPI could be done and there would be *two* large codebases that would need to be maintained. In the other way, the plugin could embed the JavaScript engine and accept strings of JavaScript code passed into it from the browser, either through some 'evaluate' method that the plugin exposes, or through special script tags the plugin can find when it loads. Of these, the second is a more elegant solution, though it will give rise to a whole set of new problems: communicating information between the browser and the plugin becomes more difficult, and rendering and updating the simulation through JavaScript also becomes problematic.

Regarding embedding V8: It is possible without too much work to embed simple C++. With larger API's, like AgX, things become more difficult. JavaScript is a good language for embedded scripting; it is comparable to, for example, Lua, since both are prototype-based, dynamically typed languages. Due to the syntax of JavaScript it should be easy for a programmer going from programming in C++ to script with JavaScript. The *main* problem with this is deciding on a way to create the bindings. Autogenerating the code is basically a must if the API is big enough, since writing glue functions for every class, function, method, enum and so forth is just too much work. As shown in

Chapter 4 it is possible to autogenerate the code. The approach taken there is to parse the original header files directly using GCCXML but it was discovered that this was difficult to get right due to how GCCXML works, so for future work another approach might be considered.

Regarding browser plugin: This part of the project produced more positive results. Because it was decided that a higher-level API should be exposed by the plugin rather than a larger part of AgX the code was written by hand, since the plugin is basically treated as a program that links to AgX and exposes a few methods to manipulate an underlying simulation. The firebreath framework made it possible to create a working plugin in a considerable faster and easier way than if NPAPI had been used directly.

7.1 Limitations

The current implementation of the pipeline can only export classes. It does not have support for global variables, enums or free functions. It did support enums at one point but the implementation was rather bad so it was removed from the prototype. Furthermore, due to problems with GCCXML the program will fail on some inputs.

The plugin mostly works as intended but a few features that would have been good to have did not make it into the prototype, most notably 'picking,' i.e. being able to click individual bodies in the simulation and interact with them. SceneJS has built in picking so if that is used it should not be too difficult to implement, but other features took priority.

7.2 Future work

To further the development of the prototypes, the direction in which the project should head must be decided.

A possible and interesting direction would be to attempt to combine the V8 bindings and the plugin. This was deemed impractical in the evaluation, but one possible way to do this is still described in the report, recall Chapter 5.8. A few tests on this were made but the results were unsatisfactory; getting information from the simulation to the browser for e.g. rendering or statistics is difficult with this approach. Rendering can of course be made by the plugin itself to some rendering context in the browser, similar to how Flash and such plugins work.

Because of some faults of GCCXML (problems compiling some files etc.), if more work should be put into the code generator for embedded scripting then it is likely that another approach should be taken when generating the bindings. Bugs in GCCXML are hard to circumvent: if GCCXML reports a 'compilation error' due to something in the code that GCCXML does not understand, then this can be fixed by adding preprocessor guards around the offending block of code and setting an appropriate macro. However, if the 'error' lies in some header outside of developer control (system headers, external libraries and so on) that are still required for compilation to succeed, then GCCXML will be basically useless. The approach used in tolua++ is a lot 'safer' but would require a (partial) C++ parser; this does not have to be all too complex though since different restrictions can be accepted depending on needs.

Continuing the work on the plugin might be more interesting than continuing on the embedded scripting part if no good way of unifying them is found. The prototype plugin uses a high-level API but that is not a definite solution: the API can of course be tailored after the aims of the plugin. It is also possible to autogenerate code, though the GCCXML approach should not be used. Interesting things to do would include picking, allowing users to upload scenes to the web page and having them load, and of course more of AgX exposed.

Chapter 8

Acknowledgments

I'd first like to thank my supervisors at Algoryx and Umeå University, Anders and Ola, for their help and support. Anders helped a lot with AgX and with pushing the project in the right direction and Ola gave great feedback on the report. I'd also like to thank the fine folks at Algoryx for showing an interest in my project which always bolsters confidence. Finally, I'd like to thank my family for general support during the span of my education.

References

[1] Burster, blender game engine browser plugin. http://geta3d.com/index.php?option=com_content\&view=category\&layout=blog&id=3&Itemid=19 (Visited on December 2010).

[2] Flot. http://code.google.com/p/flot/ (Visited in December 2010).

[3] Gccxml. http://www.gccxml.org/HTML/Index.html (Visited in September 2010).

[4] Javascript object notation. http://www.json.org/.

[5] Yaml. http://www.yaml.org/ (Visited on 21 January 2011).

[6] Working draft, standard for programming language c++, October 2008. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2798.pdf (Visited on 26 January 2011).

[7] Algoryx Simulation AB. Agx multiphysics toolkit. http://www.algoryx.se/agx (Visited on 24 January 2011).

[8] Algoryx Simulation AB. Algoryx - multiphysics and 3d simulation. http://www.algoryx.se/ (Visited on 24 January 2011).

[9] D. Abrahams. *Boost.Python*. Boost C++ Libraries. http://www.boost.org/doc/libs/1_45_0/libs/python/doc/index.html (Visited in September 2010).

[10] D. Abrahams and R. W. Grosse-Kunstleve. Building hybrid systems with boost.python. Technical report, Boost Consulting, 2003.

[11] Richard Bateman. Firebreath. http://www.firebreath.org (Visited on 26 January 2011).

[12] D. M. Beazley. Swig: An easy to use tool for integrating scripting languages with c and c++. In *Proceedings of the 4th annual Tcl/Tk Workshop*, 1996.

[13] M. Cline. Pointers to member functions, 2010. http://www.parashift.com/c++-faq-lite/pointers-to-members.html (Visited on 26 January 2011).

[14] Microsoft Corporation. Description of internet explorer support for netscape-style plug-ins. http://support.microsoft.com/kb/306790 (Visited on 18 January 2011).

[15] ECMA International. *ECMAScript Language Specification, 5th Edition*, 2009.

[16] J. J. Garrett. Ajax: A new approach to web applications, 2005.

[17] Google Inc. *Using the V8 Engine*. http://code.google.com/apis/o3d/docs/v8engine.html (Visited on 11 January 2011).

[18] Khronos Group. Getting a webgl implementation. http://www.khronos.org/webgl/wiki/Getting_a_WebGL_Implementation" (Visited on 14 December 2010).

[19] Khronos Group. Webgl - opengl es 2.0 for the web. http://www.khronos.org/webgl/ (Visited on 25 January 2011).

[20] WHAT Working Group. Web hypertext application technology working group. whatwg.org (Visited in January 2011).

[21] C. Heilmann. Websocket disabled in firefox 4. http://hacks.mozilla.org/2010/12/websockets-disabled-in-firefox-4/ (Visited on 20 January 2011), 2010.

[22] I. Hickson. The websocket protcol. Technical report, Google, Inc., 2010. http://www.whatwg.org/specs/web-socket-protocol/ (Visited on 21 January 2011).

[23] P. L. Hégaret, L. Wood, and J. Robie. *What is the Document Object Model?* World Wide Web Consortium. http://www.w3.org/TR/DOM-Level-3-Core/introduction.html (Visited on 24 January 2011).

[24] Google Inc. Npapi plugins. http://code.google.com/chrome/extensions/npapi.html (Visited on 2 February 2011).

[25] Google Inc. V8 javascript engine. http://code.google.com/apis/v8/intro.html (Visited in September 2010).

[26] J. James. Html 5 editor ian hickson discusses features, pain points, adoption rates, and more, 2008. http://blogs.techrepublic.com.com/programming-and-development/?p=718.

[27] B. Johnson. The web is reborn. *Technology Review*, December 2010. Published by MIT.

[28] L. Kay. Scenejs, 2009. http://scenejs.org/ (Visited in December 2010).

[29] Khronos Group. *WebGL Specification*, 2011. https://cvs.khronos.org/svn/repos/registry/trunk/public/webgl/doc/spec/WebGL-spec.html (Visited on January 25 2011).

[30] P. Krill. W3c: Hold off on deploying html5 in websites, 2010. http://www.infoworld.com/d/developer-world/w3c-hold-html5-in-websites-041 (Visited on 12 January 2011).

[31] P. Lubbers, B. Albers, and F. Salim. *Pro HTML 5 Programming: Powerful APIs for Richer Internet Application Development.* Apress, 2010.

[32] A. Manzur. tolua++ - binding c/c++ code to lua. http://www.codenix.com/~tolua/ (Visited on 26 January 2011).

[33] C. Metz. Google heats up native code for chrome os. *The Register*, May 2010. http://www.theregister.co.uk/2010/05/13/google_native_client_sdk/ (Visited on January 27 2011).

[34] Microsoft Corporation. *Introduction to ActiveX Controls.* http://msdn.microsoft.com/en-us/library/aa751972%28VS.85%29.aspx (Visited on 12 January 2011).

[35] Mozilla Developer Network. *Canvas Tutorial.* https://developer.mozilla.org/en/Canvas_tutorial (Visited on 17 January 2011).

[36] Mozilla Developer Network. *Details of the Object Model.* https://developer.mozilla.org/en/JavaScript/Guide/Details_of_the_Object_Model (Visited on 24 January 2011).

[37] Mozilla Developer Network. *JavaScript and Java.* https://developer.mozilla.org/en/JavaScript/Guide/JavaScript_Overview#JavaScript_and_Java (Visited on 14 January 2011).

[38] Mozilla Developer Network. *JavaScript and the ECMAScript Specification.* https://developer.mozilla.org/en/JavaScript/Guide/JavaScript_Overview#JavaScript_and_the_ECMAScript_Specification (Visited on 24 January 2011).

[39] J. K. Ousterhout. Scripting: Higher-level programming for the 21st century. Technical report, Sun Microsystems Laboratories, 1998.

[40] M. Pilgrim. *HTML5 Up and Running.* O'Reilly Media, 2010.

[41] A. Ranganathan. Beyond html5: Database apis and the road to indexdb. http://hacks.mozilla.org/2010/06/beyond-html5-database-apis-and-the-road-to-indexeddb/ (Visited on 20 January 2011).

[42] M. L. Scott. *Programming Language Pragmatics, 2nd Edition.* Morgan Kaufmann Publishers, 2006.

[43] S. Shankland. With ie9, web video issue remains deadlocked. `http://news.cnet.com/8301-30685_3-20000610-264.html` (Visited on 17 Jan 2011).

[44] Rasterbar Software. Luabind. `http://www.rasterbar.com/products/luabind.html` (Visited in September 2010).

[45] D. E. Stewart. Rigid-body dynamics with friction and impact. Technical report, Society of Industrial and Applied Mathematics, 2000.

[46] WHAT Working Group. *Web Workers*, 2011. `http://www.whatwg.org/specs/web-workers/current-work/` (Visited on 25 Januari 2011) (Note that this document is expected to change *daily*, but the general idea should remain the same).

[47] M. Wichary. Html5 presentation. `http://slides.html5rocks.com` (Visited on 18 January 2011).

[48] World Wide Web Consoritum. *XHTML^{TM}1.0 The Extensible HyperText Markup Language, 2nd Edition*, 2002. `http://www.w3.org/TR/xhtml1/` (Visited on 24 January 2011).

[49] World Wide Web Consortium. *The Canvas Element.* `http://www.w3.org/TR/html5/the-canvas-element.html` (Visited on 12 January 2011).

[50] World Wide Web Consortium. *Introduction to HTML 4.* `http://www.w3.org/TR/html4/intro/intro.html` (Visited on 17 January 2011).

[51] World Wide Web Consortium. *Web Storage*, 2011. `http://dev.w3.org/html5/webstorage/` (Visited on 27 January 2011).

[52] R. Yakovenko. Pygccxml, 2009. `http://www.language-binding.net/pygccxml/pygccxml.html` (Visited in September 2010).

[53] A. Yasushi. Box2djs, 2008. `http://box2d-js.sourceforge.net/` (Visited on December 2010).

Appendix A

Example of Embedding C++ Into V8

Assume that we have a C++ class called Rectangle that we want to expose to JavaScript using V8.

```
class Rectangle {
public:
  Rectangle(int w, int h) : m_width(w), m_height(h) {}
  int width() const { return m_width; }
  int height() const { return m_height; }
  int area() const { return m_width * m_height; }
private:
  int m_width, m_height;
};
```

While this is a very simple class, it will work decently as an example on how involving it is to make this code available to JavaScript. First, some general information on the API.

JavaScript objects are made available to C++ using *handles*. Whenever a JavaScript object is to be instantiated from C++, they need to be encapsulated in a `Handle` object:

```
Handle<String> v8String = String::New("Hello");
```

The API also have classes for all the basic primitives, i.e. strings, numbers and so on. Calling the static method `New` on a number of these classes will create a new instance encapsulated in a handle. There are two types of handles:

local and persistent. Local handles are removed as they fall out of scope, while persistent handles stick around until explicitly removed. A quirk with the persistent handles is that they can be made 'weak' to trigger a callback when only weak, persistent handles are referring to an object, allowing the programmer to do cleanup. Scopes must be defined by hand. V8 has a class called `HandleScope` that allows us to define a new JavaScript scope; this should usually be done at the start of functions that are to be exported to JavaScript and such. Whenever a new `HandleScope` is defined, all newly created handles will be put in that scope. When the scope exits the garbage collector is free to remove the objects residing in that scope.

V8 also has the concept of *contexts*. A context runs an unrelated piece of JavaScript code. The reasoning behind them is due to the way browser work with tabs and frames that each require their own unique JavaScript execution context. In this example we will only require a single context, stored in a persistent handle. We can also pass an optional global object to the context in which we might have defined functions and objects. The following piece of code creates a context, executes a piece of code in it, then destroys the context:

```
// Create a new context, with a set of global objects stored
// in globalVals.
Persistent<Context> context;
context = Context::New(NULL, globalVals);
// Create a scope for this context.
Context::Scope contextScope(context);

// Loads a piece of code, compiles and runs it.
Handle<String> source = String::New("'Hello'");
Handle<Script> script = Script::Compile(source);
Handle<Value> result = script->Run();

// Extract an ascii string from the result.
String::AsciiValue ascii(result);
// Prints the result, i.e. Hello.
std::cout << *ascii << std::endl;

// Destroy the context.
context.Dispose();
```

To expose functions and objects from C++ to JavaScript, V8 uses *templates*. Not to be confused with regular C++ templates, the V8 templates are blueprints for JavaScript functions and objects. There are two kinds of templates: **object** and **function** templates. Object templates registers functions and properties of JavaScript objects while function templates binds callable functions. The following piece of code binds a C++ function called `print` to a JavaScript object:

```
Handle<ObjectTemplate> globals = ObjectTemplate::New();
globals->Set(String::New("print"),
  FunctionTemplate::New(print));
```

The `print` function could have the following appearance:

```
// Required function signature
Handle<Value> print(const Arguments& args) {
  HandleScope scope;  // Create a new scope.
  for (size_t i = 0; i < args.Length(); i++) {
    // Extract a string from an argument.
    String::AsciiValue str(args[i]->ToString());
    std::cout << *str << std::endl;
  }
  // For void-like functions, return an Undefined value.
  return Undefined();
}
```

As mentioned earlier in Chapter 3.2 functions are also objects. Our rectangle class would be defined in JavaScript as:

```
function Rectangle(width, height) {
  this.width = width;
  this.height = height;
  this.area = function() {
    return width * height;
  }
}
```

and then instantiated in the same fashion as in C++:

```
var rect = new Rectangle(20, 20);
print(rect.area()); // Will print 400
```

Now, we want to be able to run the latter code-snippet without needing the former. Thus, we need a way to represent what will happen when we call `new Rectangle`. Now, that part is easy. We simply create a new function template and assign a function that creates a new rectangle object. What happens inside this function is a bit more complicated, however. Each JavaScript object requires a corresponding object template. To assign properties to objects we in turn assign templates to the aforementioned template. To actually get C++ data to JavaScript, V8 provides *externals* that wraps the data in a `void` pointer. Object templates also have an *internal field count* that should be set upon template creation. Objects instantiated from the template each have a number of 'internal fields' that refer to some object in C++. The number of internal fields

can generally be set to 1, to map one C++ object to each JavaScript object[1]. When an instance has been created from an object template and some data has been wrapped into an external, we set the corresponding internal field of the instance to the external. To summarize:

1. Create an object template for the class.
2. Set the template's internal field count.
3. Create an instance of the template.
4. Wrap our C++ object into an external.
5. Map the external to the appropriate internal field.

Finally, example code. First, create a function template for the constructor:

```
globals->Set(String::New("Rectangle"),
  FunctionTemplate::New(RectangleConstructor));
```

The constructor wrapper function:

```
Handle<Value> RectangleConstructor(const Arguments& args)
{
  HandleScope scope;

  // There can only be _one_ instance available
  // of each template at a time.
  if (rectangleTemplate.IsEmpty()) {
    // I will show the createTemplate function
    // further down. Note that the template is
    // a persistant so it wont be destroyed when
    // the scope exits.
    rectangleTemplate =
      Persistent<ObjectTemplate>
        ::New(createTemplate());
  }

  // Convert the arguments to integers. Error checking
  // is omitted here.
  Handle<Integer> w
    = Integer::New(args[0]->IntegerValue());
  Handle<Integer> h
    = Integer::New(args[1]->IntegerValue());
  // Create a new rectangle. In this example the
  // C++ objects will stick around even when
```

[1] I'm actually not sure what the purpose of using more than one internal field count is. I haven't found any example where more than one is used.

```
  // JavaScript objects are destroyed.
  // Solving this is left as an exercise to the reader.
  Rectangle* rect
    = new Rectangle(w->Value(), h->Value());

  // Create a new instance of the JavaScript object from the
  // template.
  Handle<Object> result
    = rectangleTemplate->NewInstance();
  // Wrap the rectangle pointer in an external.
  Handle<External> external
    = External::New(rect);
  // Map internal field 0 to the external.
  result->SetInternalField(0, external);

  // This will let the resulting handle escape
  // as the scope exits. If this wasn't used,
  // the handle would be caught by the garbage
  // collector and errors would ensure.
  return scope.Close(result);
}
```

The function to create the template. Here the use of *accessors* is also demonstrated. Accessors allow us to access properties of objects.

```
Handle<ObjectTemplate> createTemplate() {
  HandleScope scope;
  Handle<ObjectTemplate> resultTemplate
    = ObjectTemplate::New();
  resultTemplate->SetInternalFieldCount(1);

  resultTemplate->SetAccessor(String::New("width"),
    RectWidth);
  resultTemplate->SetAccessor(String::New("height"),
    RectHeight);
  resultTemplate->Set(String::New("area"),
    FunctionTemplate::New(RectArea));

  return scope.Close(resultTemplate);
}
```

Accessor functions have the following signatures:

```
Handle<Value> Getter(Local<String> property,
  const AccessorInfo& info);
void Setter(Local<String> property,
  const AccessorInfo& info);
```

Example of the width accessor:

```
Handle<Value> RectWidth(Local<String> property,
  const AccessorInfo& info) {
  // Get the calling object.
  Local<Object> self = info.Holder();

  // Get the external wrapper object from the
  // object at internal field 0.
  Local<External> external =
    Local<External>::Cast(self->GetInternalField(0));
  Rectangle* rect =
    static_cast<Rectangle*>(external->Value());

  // Call the area getter from the rectangle class and
  // return the result.
  return Integer::New(rect->width());
}
```

The area function:

```
Handle<Value> RectangleArea(const Arguments& args) {
  HandleScope scope;

  // Get the calling object from the args structure.
  Local<Object> self = args.Holder();
  // Get the external
  Local<External> external =
    Local<External>::Cast(self->GetInternalField(0));

  Rectangle* rect =
    static_cast<Rectangle*>(external->Value());

  // Call the function.
  int result = rect->area();
  v8::Handle<v8::Integer> resultHandle
    = v8::Integer::New(result);

  // Return the resulting handle.
  return scope.Close(resultHandle);
}
```

As can be seen, there are quite a lot of code needed for a very simple class , so writing all the code by hand is a tremendous task for a large API. However, there are a lot of repetitive tasks that can easily be wrapped into classes and functions, and autogeneration of code is possible.

Appendix B

User's Guide

Information on how to use the different software implemented during the thesis work is presented in this chapter.

B.1 Code Generator

The pipeline tool currently works on one file at a time. The user must specify the source header, which class(es) should be exported, if a config file should be used (see Section B.1 for more information on the configuration file format), the namespace(s) where the class(es) can be found, the file name of the generated code, as well as any additional include directories that GCCXML will need to know about.

Installing/Building

The pipeline tool requires working Python, GCCXML and PyGCCXML installations, though the tool itself does not require any installation since it is a collection of Python files.

To use the files produced by the code generator they should simply be included with the project where JavaScript should be embedded. The project will need to link and include V8.

Usage

A sample execution of the pipeline tool can look like (from the command line):

```
$ python pipeline.py \
    --file=agx/RigidBody.h \
    --class=::agx::RigidBody \
    --config=export-3 \
    -o RigidBodyBindings \
    -n agx \
    -Iagx/include \
    -Idependencies/agx_dependencies/include
```

The available options for the program is:

Option	Action
`-h, --help`	Print program help
`-I, --include-path=`	Path to search for additional includes.
`-f, --file=`	The source file name on which to run the pipeline on.
`-c, --class=`	The class name to export. Program accepts any number of these.
`-C, --config=`	The config file to use.
`-o, --output=`	The filename of the file with the generated code.
`-n, --namespace=`	The namespace where to look for classes.

The generator produces three files per binding: a header file containing glue function declarations, a source file containing declarations and an additional source file that has initializing code that sets up all the bindings.

After the program has been run on the desired files, these must then be 'linked' together so that all the initializing functions can be called from the program supposed to use the bindings. Thus as a final step the user must run

```
$ python linker.py
```

This produces a source and header file called 'Generated.cpp' and 'Generated.h,' respectively. The header file includes all the headers of the generated files and declares a function called `loadGenerated` and the source file contains the declaration of this function. The function calls all the initializing code created by the code generator, so a call to `loadGenerated()` must appear somewhere before the V8 interpreter is started.

Config File Format

The configuration file format uses a data serialization format called *YAML*. There is a lot of information available about it on the web but a good start would be the official YAML website [5]. The configuration format has three different directives:

- `gccxml-path`
  Tells the system where GCCXML can be found. If this is not defined a default value is chosen.

- `special-export`
  Allows aliases to be given to classes and their methods.

- `redefines`
  Tells the code generator to treat types as a different type; this is mostly used to handle smart pointers and such.

An example config:

```
gccxml-path: C:/Program Files (x86)/gccxml 0.9/bin

# Rename the agx::Vec3T<double> type
# to Vec3, and rename its two operators
# to something that makes sense in
# JavaScript.
special-export:
  - name: ::agx::Vec3T<double>
    new-name: Vec3
    method-alias:
     - operator^: cross
     - operator*: dot

redefines:
  - treat: ::agx::ref_ptr<agx::RigidBody>
    as: ::agx::RigidBody
  - treat: ::agx::ref_ptr<agx::Constraint>
    as: ::agx::Constraint
```

B.2 Plugin

This section will deal with the usage of the example web application written for the prototype plugin, and also how the plugin can be used in general.

Sample Web Application

In Figure B.1 a screenshot of the web application can be seen. The sample app has a rendering context, a JavaScript console where a few commands can be entered, a plot that displays the total kinetic energy of the simulation as well as a few sliders that can be used to modify material properties.

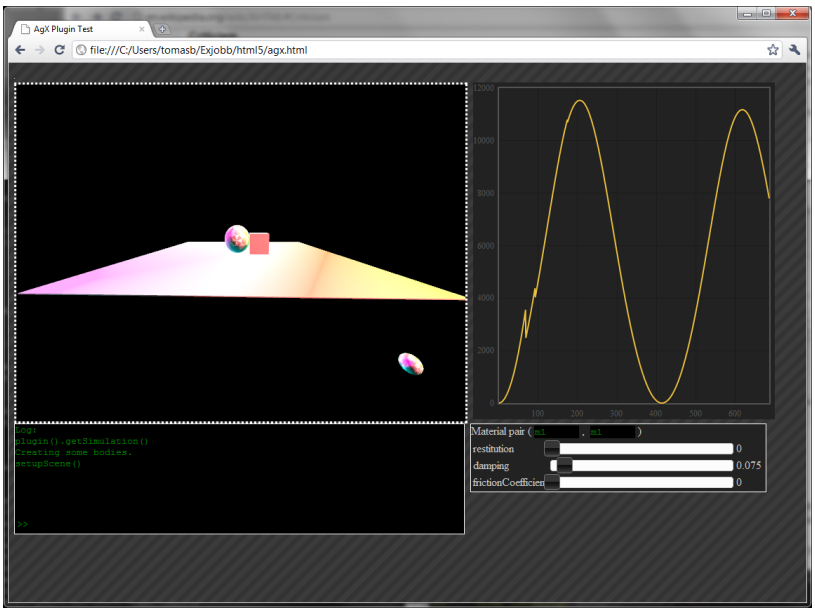

Figure B.1: **Screenshot of a web page with the plugin, running in Chromium.**

The camera can be moved with the mouse; by pressing the left mouse button the camera can be moved around and by holding shift and pressing the left mouse button the camera can be zoomed in or out by moving the mouse up or down. It is also possible to move around using the W, A, S and D keys where W and S moves forward/backward, and A and D strafes left/right.

All interaction with the simulation is currently done through the console where JavaScript code can be entered. So for example to create a sphere with radius 1.0 and mass 10 at the origin, type `createSphere( [0, 0, 0], 1.0, 10 );` and the sphere will appear.

The plot is updated every second, but there is a small performance hit every time this is done. To disable (or enable) the plot type `togglePlot()`.

The sliders can be dragged to change some properties of materials; the 'material pair' boxes must correspond to a contact material. The default values for the boxes are 'm1' which is a predefined contact material.

The Plugin

The plugin itself is agnostic to rendering and only contains the Agx physics engine. If rendering is desired it must be done through JavaScript. The cur-

rent API is rather simple with only a few usable objects and methods but it is enough to create a simple simulation with spheres and boxes colliding with each other.

Notation: `foo()` means method, `bar` means property.

Plugin Object

- `getSimulation()`
  Retrieves the simulation object created by the plugin.
- `getDebugRenderer()`
  Gets a *debug renderer* which is a class that can render geometries in the simulation. Callbacks can be registered to this object through JavaScript. This is one way to render the scene.
- `useDebugRenderer()`
  Tells the plugin to use the debug renderer for rendering.
- `startTimer()`
  Starts a high-precision timer.
- `checkTimer()`
  Get the time elapsed since the timer was started.

Simulation Object

- `stepTo(timestamp)`
  Steps the simulation to `timestamp`.
- `stepForward()`
  Steps the simulation one timestep.
- `addSphereBody(id, position, radius, mass, materialId, isStatic)`
  Creates a spherical rigid body and adds it to the simulation. Also returns the created body so that its properties can be accessed from JavaScript. If `isStatic` is `true`, then the body will be static and won't be effected by external forces.
- `addBoxBody(id, position, halfExtents, mass, materialId, isStatic)`
  Similar to `addSphereBody`, this method instead creates a box.
- `createMaterial(id)`
  Creates a new material with the specified identifier.
- `combineMaterials(id1, id2, properties)`
  Combines two materials to create a contact material, i.e. the properties of a contact when two objects with the different materials collide.

`properties` is a JavaScript object that can have the following properties set: restitution, frictionCoefficient, surfaceViscosity, adhesion, lineStickFriction, lineSlideFriction, enableSurfaceFriction, youngs and damping.

- `setMaterialProperties(contactMaterial, properties)`
  Set the properties of a contactmaterial, which should be a pair `[material1, material2]` that has been previously combined using `combineMaterials`. Properties should be the same as in `combineMaterials`.

- `createDistanceJoint(body1id, attachA, body2id, attachB, params)`
  Creates a distance joint constraint between two bodies, attached at `attachA` and `attachB` in body coordinates. `params` is a JavaScript object that can have two properties: compliance and damping.

Rigid Body Object

There are two kinds of rigid bodies: spheres and boxes. It is not necessary to keep them as separate objects but that was the approach taken here.

- `position`
  Gets the position as a 3 element array.

- `transform`
  Gets the transformation matrix as a 16 element array.

- `type`
  Gets the type, which here is either box or sphere.

- `name`
  Gets the name or id of the body.

- `potentialEnergy`
  Gets the potential energy of the body.

- `rotationalEnergy`
  Gets the rotational energy of the body.

- `translationalEnergy`
  Gets the translational energy of the body.

- `kineticEnergy`
  Gets the kinetic energy of the body, which is $E_r + E_t$ or the sum of the rotational and the translational energy.

JSDebugRenderer

The `JSDebugRenderer` inherits from the AgX `DebugRenderer` class.

- `registerRenderCallback(objectType, callback)`
  Registers `callback` to be called when `objectType` should be rendered. There are many different objects that can be rendered, for example spheres, boxes and planes, to name a few.

- `render()`
  Renders the contents of the DebugRenderer.

Example on how to use the debug renderer from JavaScript:

```
var renderer = plugin.getDebugRenderer();
renderer.registerRenderCallback("sphere",
  function(radius, transform, color) {
    // Assuming we have some library
    // cabable of the following.
    setTransform(transform);
    setColor(color);
    renderSphere(radius);
  });
```

Then, assuming that the simulation have some spherical body added to it, calling `renderer.render();` would then cause a sphere to be rendered. Note that the debugrenderer is not the only way to get graphics; but it was used during early development of the plugin and there was no reason to remove it from the codebase since it worked rather well.

Appendix C

List of WebGL Libraries

There exists many different libraries and frameworks that can help with the development of WebGL applications. A few of these are presented below.

- **SceneJS** (`http://www.scenejs.org`) - A scene graph using JSON notation to describe the graph. SceneJS was used for the prototype plugin in this thesis.
- **Three.js** (`https://github.com/mrdoob/three.js/`) - The aim of `three.js` is a lightweight 3D engine that should be easy to use. It can use either WebGL for hardware acceleration or just the regular 2D canvas for software rendering. The github page also contains a few impressive demos showcasing the capabilities of the library/WebGL.
- **o3d** (`http://code.google.com/p/o3d/`) - Another scenegraph, o3d is developed by Google.
- **SpiderGL** (`http://spidergl.org/`)
- **GLGE** (`http://www.glge.org/`)

Appendix D

Source Code

The Python source code for the code generator is available at `http://www.cs.umu.se/~tomasb/v8codegen`. Also, the source code for the experimental template based library, wiz8, can be browsed at `http://www.cs.umu.se/~tomasb/wiz8`.

The source code for an example WebGL application can be viewed at `http://www.cs.umu.se/~tomasb/webgl-demo/gl-src.js` and the demo itself at `http://www.cs.umu.se/~tomasb/webgl-demo/webgl.html`. The demo will display a cube spinning in the center of a black viewport. The matrix and vector functions comes from a different library. A screenshot of the result of the code can be seen in Figure 3.4 in Chapter 3. Note that the browser must have WebGL support enabled for the demo to show, for information on how this can be done, see [18].