plan.js: A Motion Planning Library for Javascript

Clinton Freeman

***Abstract*—In the past decade, there has been a large movement in industry toward web browser based applications and away from native desktop programs. My semester project was an attempt to build a JavaScript library, plan.js, for motion planning in web applications. This document recounts attempts at converting two native motion planning libraries into JavaScript by using Mozilla's Emscripten. The primary contribution of this project is an evaluation of the work required to convert each library, preliminary headway on converting both projects, and a functional demo of the PQP collision detection library converted to JavaScript.**

I. Introduction

In the past decade, there has been a large movement in industry toward web browser based applications and away from native desktop programs. This is due in large part because of the inherent cross-platform nature of in-browser applications. As a part of this movement, numerous open source libraries have been developed that provide functionality that has been available in other languages such as C/C++ for quite some time. While several libraries have covered important areas such as 3D graphics (three.js) and collision detection (ammo.js), there is not a widely used motion planning library to the best of my knowledge.

Plan.js is a JavaScript library that aims to address this gap in available software. It should contain many commonly used motion planning algorithms such as RRT, RRT*, PRM, and exact geometric methods for 2D motion planning. It should be open source with the source code available to others on GitHub, and users should be able to view visualizations of the algorithms live in the browser.

II. Related Work

A. Native Planning Libraries

The Open Motion Planning Library (OMPL) is a standard resource for motion planning algorithms for desktop C++ applications [1]. It contains implementations of a wide variety of algorithms, including RRT*, PRM, and others. Many years of work have gone into the implementations available in this library.

The motion strategy library (MSL) is a C++ library built by Steve Lavalle's group at the University of Iowa and subsequently at the University of Illinois [2]. It also contains some popular algorithms, but has not been updated in many years.

B. Native to JavaScript Conversion Tools

Emscripten is a compiler developed by Mozilla that compiles native C/C++ code into a subset of Javascript known as *asm.js* [3]. asm.js javascript files are able to be compiled by the browser ahead of time into machine code in order to gain a significant boost in performance. Mozilla claims that using Emscripten and asm.js can result in performance that is only twice as slow as native applications. Major game engines including Unity3D and UnrealEngine are providing asm.js ports of their software that can run at interactive speeds.

Portable Native Client (PNaCL) is Google's alternative to Emscripten [4]. It follows a similar scheme of converting C++ source code to a LLVM-esque bytecode, but stops short of converting this into JavaScript, instead choosing to add a bytecode interpreter into the browser. This approach gives PNaCL a significant advantage over Emscripten in terms of speed, since the compiled code is able to run at native speeds.

C. Examples of Converting Large Native Projects

As mentioned in the introduction, ammo.js is a javascript library for performing collision detection. This library is created by compiling the popular Bullet collision detection library with Emscripten. This is one route that one can take when attempting to port a library over to JavaScript.

Google has converted many popular C++ projects to run under PNaCL in order to demonstrate its capability to handle a broad range of inputs. In particular, they provide a fully working version of the Boost libraries including boost_thread. This is important to note since the dependency on Boost and threads is the primary roadblock to a full conversion of OMPL with Emscripten.

III. Problem Definition

Native code is composed of machine instructions specific to a particular computer architecture (e.g. x86). This is in contrast to *interpreted* code, which is composed of instructions that are translated into actual machine instructions by a virtual machine at run time. Native code has historically run significantly faster than interpreted code, although over time the gap has become much smaller. While native code (generated by languages such as C++) has an edge when it comes to runtime performance, interpreted code (generated by languages such as Java) is generally able to run across a larger number of platforms without modification.

A *web worker* is a JavaScript utility that is essentially a thread that has no shared state and may only communicate via message passing. This is a problem for converting native programs to JavaScript because there is no isomorphism between normal threads and web workers. Thus, any program that uses multithreading of any sort must be modified to account for this problem.

Emscripten is a compiler from low level virtual machine (LLVM) bytecode to JavaScript. Clang is a compiler that converts C++ source code into LLVM bytecode. gcc is a

compiler that converts C++ source code directly into machine code. emcc is Emscripten's replacement for gcc that uses Clang behind the scenes to compile to bytecode.

Most large projects use Makefiles to automate the build process. A make file specifies how output files are generated from input files by specifying a set of rules. In order to build a project, a user will typically change directory to the root of the project and invoke the command make ¡target¿. Emscripten allows users to instead run emmake make ¡target¿ to compile javascript outputs instead of normal exectuables, archives, and shared objects.

IV. Methods

My goal with this project was to build a motion planning library that is actually useful to web programmers. There are (broadly) two ways to reach this goal.

The first is to hand-write JavaScript implementations of the desired algorithms. While this method is an attractive way to cement my understanding of each algorithm I successfully implement, it is very work intensive and will take a great deal of time and effort to produce something that may be considered useful to other programmers.

The second is to convert an existing native library using an automated tool such as Emscripten or PNaCL. This method is attractive from a productivity point of view: if successful, I can reuse a great deal of work in order to benefit other programmers. A downside is that the problem becomes not implementing motion planning algorithms (i.e. learning the subject matter of the course), but instead is an engineering challenge of converting code from one language to another.

I chose the second method because I wanted the project to have a real product that could be used by others at the end. I chose to use Emscripten over PNaCL to convert an existing library because the results would be immediate and the general consensus on the web is that PNaCL will not enjoy wide adoption beyond Google Chrome. I initially chose to convert OMPL instead of MSL because OMPL is significantly more active and contains newer algorithms.

V. Results

A. Converting OMPL

OMPL depends on the Boost C++ libraries for much of its functionality. In particular, the planner classes all inherit from a base class that inherits from boost_thread. Removing this dependence is rather problematic since it occurs at such an abstract level of OMPL and affects virtually all motion planning algorithms. It is possible that a programmer more intimately familiar with the inner workings of OMPL would be able to remove boost_thread without too much trouble, but it was not feasible for me to do so within the time frame of this class.

B. Converting MSL

While converting OMPL is an attractive goal since it is more widely supported than MSL, MSL is an order of magnitude easier to convert using Emscripten. This is due to three reasons: it does not use threads at all, it comes with a Makefile tailored to using gcc in a Linux environment, and its only dependence is on PQP.

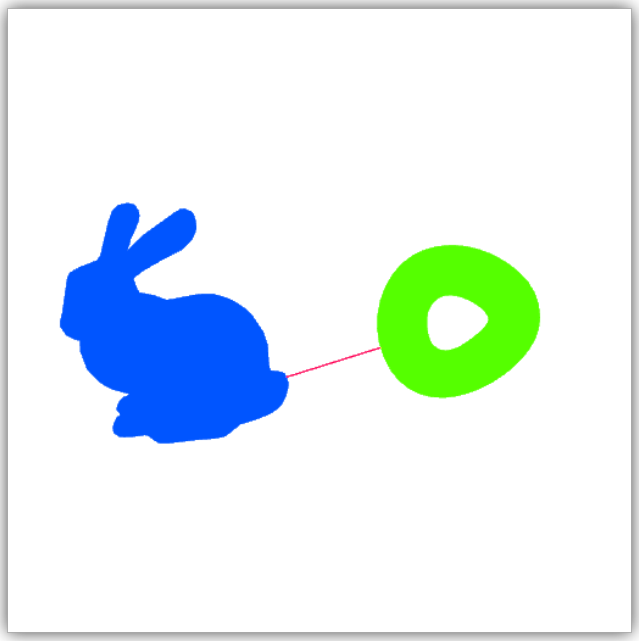

Fig. 1. PQP's "spinning" demo converted to WebGL with Emscripten. The red line denotes the shortest distance between the blue bunny and green torus.

The proximity query package (PQP) is a C++ collision detection library written by the GAMMA group at UNC-Chapel Hill [5], [6]. The implementation and source code is extremely simple in comparison to monolithic packages such as Bullet. It operates on pairs of triangle meshes and places no restriction on mesh topology. It comes with several demo applications that show various objects and the computed results from PQP (intersection set, shortest distance, etc).

Converting PQP with Emscripten was very straightforward. The primary challenge was removing outdated OpenGL calls that are not supported by Emscripten/WebGL.

References

[1] I. A. Şucan, M. Moll, and L. E. Kavraki, "The Open Motion Planning Library," *IEEE Robotics & Automation Magazine*, vol. 19, no. 4, pp. 72–82, December 2012, http://ompl.kavrakilab.org.

[2] "Motion strategy library," http://msl.cs.uiuc.edu/msl/, accessed: 2014-05-01.

[3] A. Zakai, "Emscripten: an llvm-to-javascript compiler," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM, 2011, pp. 301–312.

[4] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE, 2009, pp. 79–93.

[5] S. Gottschalk, M. C. Lin, and D. Manocha, "Obbtree: A hierarchical structure for rapid interference detection," in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. ACM, 1996, pp. 171–180.

[6] E. Larsen, S. Gottschalk, M. C. Lin, and D. Manocha, "Fast proximity queries with swept sphere volumes," Tech. Rep.