Connecting C++ and JavaScript on the Web with Embind

IMVU Inc.

@chadaustin

Hi, my name is Chad Austin, technical director at IMVU, and today we're going to talk about a library we've developed for connecting C++ and JavaScript with Emscripten.

Agenda

- Why IMVU selected Emscripten
  - and thus wrote Embind
- Overview of Embind's features
- C++11 tricks in the Embind implementation
  - code size
  - syntax
- Please hold questions until the end

What is IMVU?

IMVU is an online social platform where you can sign up, dress up an avatar, and meet people from all around the world. We offer other activities such as games as well.

What is IMVU?

- 130 million registered accounts
- 16 million user-generated virtual items

The content in our world is created by our customers, and to our knowledge, we have the largest catalog of 3D virtual goods on the Internet.

Why Emscripten?

We currently offer a downloadable application for Windows and Mac. Windows and Mac are great platforms, but in recent years, other platforms have grown to prominence. We'd like our content available everywhere: mobile platforms, desktop, server-side renderers, and even the web browser!

For almost all platforms, it's obvious that C++ is a great choice for the core engine. However, our big question was, what about the web browser?

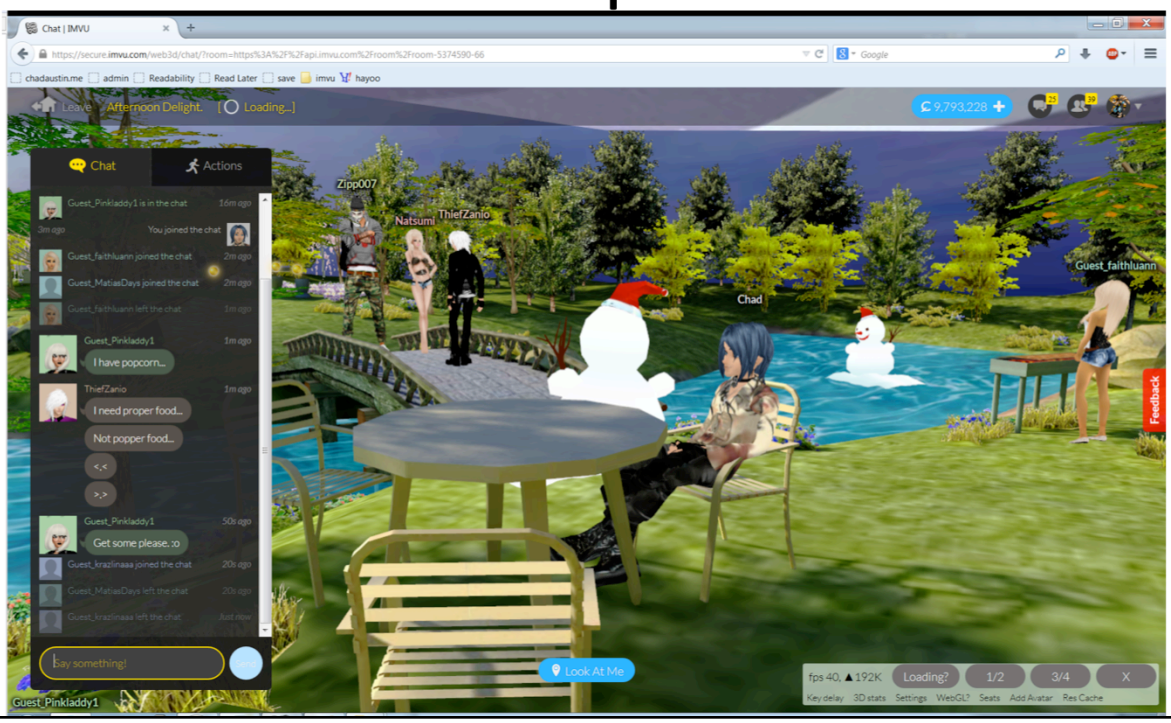
In 2011, I benchmarked an upcoming tool called Emscripten and was quite impressed.

Emscripten

- Compiles C++ into JavaScript
- asm.js has ~50% of native performance
- No download or plugins!
- Portable across Windows, Mac, Linux, Mobile, Tablets
- C++: high-performance language on ALL platforms, including web

Emscripten works very well in practice, so the implication is that C++ is the portable, high-performance language EVERYWHERE.

Emscripten!



Here is our Emscripten application running in Firefox. UI is HTML and CSS. Chat over WebSockets, graphics in WebGL.

asm.js

- Statically-compilable,
- Machine-language-translatable,
- Typed,
- Garbage-collection-free,
- Subset of JavaScript

asm.js is the subset of JavaScript that can be statically compiled into machine code.

More information at http://asmjs.org/

asm.js

- Integer arithmetic mapped to JS operators
- Heap represented as one **ArrayBuffer**
  - 8 **TypedArrayViews** alias into that buffer:
    - {signed, unsigned} {8, 16, 32} bit integers
    - 32- and 64-bit floats
  - See Alon's presentation and engineering.imvu.com for more details

The C heap is stored in an ArrayBuffer in JavaScript. One contiguous blob of memory.

This memory is indexed by eight different typed array views that alias each other. This is how C memory semantics are implemented.

asm.js example

```
// C++
void increment(unsigned* p) {
   ++(*p);
}

// JavaScript
function _increment(p) {
   p = p | 0; // p is an unsigned integer
   HEAPU32[p>>2] = (HEAPU32[p>>2] + 1) | 0;
}
```

In this example, we can see how the load, increment, and store are translated into JavaScript.

The p = p | 0 line indicates that p is an unsigned 32-bit integer.
The second line is a load, addition, and store. It can get compiled into a single instruction in a good JavaScript JIT.

Emscripten

- Compiling C++ into JS is just half of the platform
- Implementations of many POSIX functions
- Some hand-rolled APIs to access browser capabilities from C++
  - setTimeout()
  - eval()

Compilation is just half of the story. Without being able to access the platform's capabilities, your program can't do anything interesting.

Emscripten provides a whole bunch of APIs. It implements a bunch of POSIX and has some hand-rolled APIs for specific bits of browser functionality.

Browser Integration

- JavaScript

```
setTimeout(function() {
    ...
}, 1000);
```

- Emscripten

```
emscripten_async_call([](void* arg) {
    ...
}, arg, 1000);
```

Here's an example of how I can translate a bit of JavaScript into the equivalent Emscripten C++. The body of the lambda will run a second in the future.

Functions like this are very useful when available. But we can't hope that Emscripten will implement handy dandy wrappers for everything. Browsers have hundreds and thousands of APIs and are always adding more.

Embind sets out to make it possible to access these JavaScript APIs directly from C++.

Web Applications Want C++

- High-performance C++ components
- Existing C++ libraries

In addition, Embind makes it possible to use C++ libraries from web applications. You can take existing C++ and expose it to JavaScript.

From a high level

EMBIND

Embind

- C++ ⇔ JavaScript binding API
- Bidirectional!
- Inspired by Boost.Python
- Included with Emscripten
- Heavy use of C++11 features
  - variadic templates
  - constexpr
  - <type_traits>

Boost.Python

- Almost every project I've worked on in the last decade has used Boost.Python
- Some things I've never liked about Boost.Python
  - Significant C++ <-> Python call overhead
  - Huge generated code size
  - Huge compile times
  - Too much is implicit (e.g. automatic copy constructors)

Embind Design Spirit

- Bindings written in C++
  - no custom build step
- Using JavaScript terminology
- Minimal runtime overhead
  - generates high-performance glue code at runtime
- Short, concise implementation

It's important, when building bindings, to realize you're building a JavaScript API. You need to think about how it's going to be used from JavaScript. Thus, Embind tries to use JavaScript terminology when appropriate.

Embind tries to have minimal overhead – unlike Boost.Python.

BINDING C++ TO JAVASCRIPT

Example

```
EMSCRIPTEN_BINDINGS(foo_library) {
    function("foo", &foo);
    class_<C>("C")
        .constructor<int, std::string>()
        .function("method", &C::method)
        ;
}
```

Features

- classes
  - member functions
  - ES5 properties
  - raw pointer ownership
  - smart pointer ownership
- enums (both **enum** and **enum class**)
- named arbitrary constant values
- JavaScript extending C++ classes
- overloading by argument count (not type)

ES5 Properties

```
struct Character {
   int health = 100;
   void setHealth(int p) { health = p; }
   int getHealth() const { return health; }
};

class_<Character>("Character")
   .constructor<>()
   .property("health",
      &Character::getHealth,
      &Character::setHealth)
   ;
```

When the 'health' property is accessed from JavaScript, it actually calls the underlying C ++ getter and setter.

Enums

```
enum Color { RED, GREEN, BLUE };

enum_<Color>("Color")
    .value("RED", RED)
    .value("GREEN", GREEN)
    .value("BLUE", BLUE)
    ;
```

Enums are nice and simple, but it's important to give everything a name so JavaScript can find it.

Constants

```
constant(
   “DIAMETER_OF_EARTH”,
   DIAMETER_OF_EARTH);
```

Memory Management

- JavaScript has NO weak pointers or GC callbacks
- Manual memory management of C++ objects from JavaScript
  - simple refcounting support provided

Memory Management

```
struct Point { int x, y; };
Point makePoint(int x, int y);

class_<Point>("Point")
    .property("x", &Point::x)
    .property("y", &Point::y)
    ;
function("makePoint", &makePoint);
```

Memory Management

```
> var p = makePoint(10, 20);
> console.log(p.x);
10
> console.log(p);
[Object]
> p.delete(); // ☹
```

What is p? Well, it's actually an Embind Instance Handle. That is, it's a special JavaScript object, provided by Embind, that holds a raw pointer into the Emscripten heap, where a Point struct lives.

If you don't delete p, the Point in the Emscripten heap leaks!

Memory Management (con't)

- "value types"
  - by-value conversion between C++ types and JavaScript Objects
    - {x: 10, y: 20}
  - conversion between C++ types and JavaScript Arrays
    - [10, 20]

To that end, Embind provides value types, which denote that a type will be passed by value across the language boundary.

You can either bind to JavaScript Objects or JavaScript Arrays

Value Objects Example

```
// C++
value_object<Point>("Point")
    .field("x", &Point::x)
    .field("y", &Point::y)
    ;

// JS
var p = makePoint(10, 20);
console.log(p);
// {x: 10, y: 20}
// no need to delete
```

I wish value_object .field was called .property, given we want Embind to use JavaScript terminology. ☺ Maybe we'll rename that.

USING JAVASCRIPT FROM C++

Calling JS from C++

- **emscripten::val**
- allows manipulation of JS values from C++

```
// JavaScript
var now = Date.now();

// C++
double now = val::global("Date").call<double>("now");
```

Using Web Audio from C++

```
#include <emscripten/val.h>
using namespace emscripten;

int main() {
  val context = val::global("AudioContext").new_(); // new AudioContext()
  val oscillator = context.call<val>("createOscillator");

  oscillator.set("type", val("triangle")); // oscillator.type = “triangle”
  oscillator["frequency"].set("value", val(262)) // oscillator.frequency.value = 262

  oscillator.call<void>("connect", context["destination"]);
  oscillator.call<void>("start", 0);
}
```

IMPLEMENTATION

Type IDs & Wire Types

- Every C++ type has a **Type ID**
- **Type IDs** have a name
- Every C++ type has a corresponding **Wire Type**
  - C++ can produce a **Wire Type** for any value
  - JS can produce a **Wire Type**

Wire Types

C++ Type	Wire Type	JavaScript Type
int	int	Number
char	char	Number
double	double	Number
std::string	struct { size_t, char[] }*	String
std::wstring	struct { size_t, wchar_t[] }*	String
emscripten::val	_EM_VAL*	arbitrary value
class T	T*	Embind Handle

Function Binding

```
float add2(float x, float y) { return x + y; }

EMSCRIPTEN_BINDINGS(one_function) {
   function("add2", &add2);
}

// Notify embind of name, signature, and fp
```

function(name, &fp) generates the signature information at compile time from the function pointer.

Function Binding (con't)

```
void _embind_register_function(
    const char* name,
    unsigned argCount,
    const TYPEID argTypes[],
    const char* signature,
    GenericFunction invoker,
    GenericFunction function);
```

_embind_register_function is the internal implementation of function().

Function Binding Under The Covers

```
function("add2", &add2);

// becomes

TYPEID argTypes[3] = {getTypeID<float>(), getTypeID<float>(), getTypeID<float>()};
_embind_register_function(
   "add2",
   3,
   argTypes,
   "fff",
   &Invoker<float, float, float>,
   &add2);
```

Function Binding (con't)

```
_embind_register_function: function(name, argCount, rawArgTypesAddr, signature, rawInvoker,
fn) {
    var argTypes = heap32VectorToArray(argCount, rawArgTypesAddr);
    name = readLatin1String(name);
    rawInvoker = requireFunction(signature, rawInvoker);

    exposePublicSymbol(name, function() {
        throwUnboundTypeError('Cannot call ' + name + ' due to unbound types', argTypes);
    }, argCount - 1);

    whenDependentTypesAreResolved([], argTypes, function(argTypes) {
        var invokerArgsArray = [argTypes[0], null].concat(argTypes.slice(1));
        replacePublicSymbol(name, craftInvokerFunction(name, invokerArgsArray, null, rawInvoker,
fn), argCount - 1);
        return [];
    });
},
```

_embind_register_function is called from C++ but implemented in JavaScript.

C++ TECHNIQUES AND TRICKS

C++ Techniques and Tricks

- Code Size
  - Using **static constexpr** to create static arrays
  - RTTI Light
- Syntax
  - select_overload
  - optional_override

Why is code size so important?

- Native Application
  - mmap .exe on disk
  - begin executing functions
  - page in instructions on demand
- JavaScript Application
  - download JavaScript
  - parse
  - codegen on user's machine
  - execute JavaScript, maybe JIT on the fly

Native executables pay almost no penalty for huge executables, since they're just memory-mapped and paged in as executed.

However, JavaScript applications require the customer to use the network connection to download and their CPU to parse and optimize the JavaScript. So even dead code hurts in JavaScript. Careful!

STATIC ARRAYS

Function Binding (con't)

- name
  - "add2"
- signature
  - 3 (1 return value, 2 arguments)
  - argTypes = {FLOAT, FLOAT, FLOAT}
  - asm.js signature string: "fff"
  - invoker = arg reconstruction from wiretype
- function pointer

Signatures

- Signatures are known at compile-time
- Signatures are constant
- Often reused
  - e.g. float operator+, float operator*, and powf

- constexpr!

asm.js Signature Strings

- asm.js function table signature strings
- <void, float, int, char*> → “vfii”

- Wanted: compile-time string literal generation

asm.js function pointers are typed. They are indices into typed function pointers. So a function pointer isn’t enough information to look up the corresponding JavaScript function: you need to know which table to look at too.

SignatureCode

```
template<typename T> struct SignatureCode {
   static constexpr char get() { return 'i'; }
};

template<> struct SignatureCode<void> {
   static constexpr char get() { return 'v'; }
};

template<> struct SignatureCode<float> {
   static constexpr char get() { return 'f'; }
};

template<> struct SignatureCode<double> {
   static constexpr char get() { return 'd'; }
};
```

First, we need to be able to map a type to the corresponding asm.js signature code.

getSignature

```
template<typename Return, typename... Args>
const char* getSignature(Return (*)(Args...)) {
   static constexpr char str[] = {
      SignatureCode<Return>::get(),
      SignatureCode<Args>::get()...,
      0 };
   return str;
}
```

Take a function pointer as argument, but don't use its value, just its deduced type.

Then, create a static constexpr character array where the first entry is the SignatureCode of the first type, followed by the SignatureCodes of the argument types, followed by the terminating zero.

This produces a compile-time constant string!

If you call getSignature, the call is completely inlined and optimized down to a single constant pointer into the application's static data segment.

RTTI LIGHT

RTTI Light

```
void _embind_register_function(
    const char* name,
    unsigned argCount,
    const TYPEID argTypes[],
    const char* signature,
    GenericFunction invoker,
    GenericFunction function);
```

- TYPEID is an integer or void* that identifies the type
- Used as index into type registry

The return types and argument types are passed in the argTypes array. The return type is the first element.

Remember that TYPEID is simply a unique identifier for a type, and we can compute a TYPEID from a type at compile time.

Original TYPEID Implementation

- Originally used **typeid**()
- **typedef const** std::type_info* TYPEID;

- Problem: code size!

Problems with typeid

- typeid pulls in a lot of extra junk
  - e.g. long string constants for mangled names
- Embind already associates human names with every type, typeid name is only necessary for errors
  - "Error: tried to call function X but argument 2 has unbound type Y"
  - Errors only used for debugging
  - #define EMSCRIPTEN_HAS_UNBOUND_TYPE_NAMES 0

RTTI Light Requirements

- All embind needs, per type, is:
  - unique word-sized identifier per type
  - unique string name
- Lookup should constexpr (we'll see why later)
- Important: still need full RTTI for runtime identification of polymorphic pointers!
- LightTypeID must inhabit the same namespace as typeid to avoid namespace collisions

TYPEID lookup

```
typedef const void* TYPEID;

template<typename T>
static constexpr TYPEID getLightTypeID() {
   return std::is_polymorphic<T>::value
      ? &typeid(C)
      : LightTypeID<C>::get();
}
```

LightTypeID

```
template<typename T>
struct LightTypeID {
   static char c;
   static constexpr TYPEID get() {
      return &c;
   }
};

// Warning: how does linkage work here?
template<typename T>
char LightTypeID<T>::c;
```

The implementation of LightTypeID allocates a byte in the static data segment… and uses that byte's address as the TYPEID!
We know it doesn't conflict with typeid, since two objects can't share an address.

I don't fully understand the linkage rules that make the template definition legal in a header, but someone in the audience said what I'm doing is okay. ☺

RTTI Light

- Allocates a single byte in the static data segment per type, uses its address
- Same namespace as typeid
- Huge code size savings!
- 175 KB off of our minified JavaScript build

Signature TYPEID[]

```
template<typename… Args>
static const TYPEID* getTypeIDs() {
   static constexpr TYPEID types[] = {
      TypeID<Args>::get()…
   };
   return types;
}
```

Once we have the ability to look up TYPEIDs from argument lists, we can build a static array of them at compile time…

And then use its constant address as the signature description.

Back to Function Registration

```
_embind_register_function(
    50001482, // address of “add2”
    3, // argCount=3
    50001830, // address of TYPEID[3]
    50001497, // address of “fff”
    106, // function pointer of invoker
    80); // function pointer of add2
```

After the above optimizations, function registration is entirely table-driven, resulting in small generated code.

SELECT_OVERLOAD

select_overload

- Want to bind overloaded function e.g. pow()

```
// ambiguous: pow is overloaded
function(“pow”, &pow);
```

- You can C-style cast to select the function signature

```
function(“powf”, (float(*)(float,float))&pow);
function(“powd”, (double(*)(double,double))&pow);
```

C-style casts are gross

- Ugly (*) sigil
- Dangerous when function is refactored to not be overloaded
  - C-style cast will still succeed!
  - Undefined behavior

Better Way

```
function("powf",
   select_overload<float(float,float)>(&pow));
function("powd",
   select_overload<double(double,double)>(&pow));
```

select_overload Implementation

```
template<typename Signature>
Signature* select_overload(Signature* fn) {
   return fn;
}
```

select_overload on Member Functions

```
struct HasProperty {
   int prop();
   void prop(int);
};
```

The Old Way

- C-style casting requires duplicating class name

```
class_<HasProperty>("HasProperty")
    .method("prop",
        (int(HasProperty::*)())&HasProperty::prop)
    .method("prop",
        (void(HasProperty::*)(int))&HasProperty::prop)
    ;
```

Using select_overload

```
class_<HasProperty>("HasProperty")
    .method("prop", select_overload<int()>(
        &HasProperty::prop))
    .method("prop", select_overload<void(int)>(
        &HasProperty::prop))
    ;
```

- Does not repeat class name

select_overload simplifies overload selection syntax by only requiring the interesting bit of knowledge – the desired function signature – while not requiring inferred information such as the class type.

select_overload Implementation

```
template<
    typename Signature,
    typename ClassType>
auto select_overload(
    Signature (ClassType::*fn)
) -> decltype(fn) {
    return fn;
}
```

select_overload is overloaded: once for function pointers, and once for member function pointers.

aka deducing signature of captureless lambda

OPTIONAL_OVERRIDE

optional_override in use

```
struct Base {
   virtual void invoke(const std::string& str) {
      // default implementation
   }
};

class_<Base>("Base")
   .allow_subclass<BaseWrapper>()
   .function("invoke", optional_override([](Base& self, const std::string& str) {
      return self.Base::invoke(str);
   }))
   ;
```

optional_override

- Sometimes you want to bind a captureless lambda
  - Use case is too subtle for discussion here
  - Captureless lambdas can be coerced into C function pointers
- But what's a lambda's signature?

Lambdas are Sugar for Objects with Call Operators

```
[](int a) { return a + 2; }

// desugars to

struct __AnonymousLambda {
   int operator()(int a) { return __body(a); }
   typedef int(*__FP)(int);
   operator __FP() { return &__body; }
private:
   static int __body(int a) { return a + 2; }
};
```

- We want type of function pointer: int(*)(int) in this case

Given a captureless lambda type with unknown signature, we want to convert the lambda into a function pointer, so that we can use the function pointer to deduce the signature for the binding.

optional_override Implementation

```
// this should be in <type_traits>, but alas, it's not
template<typename T> struct remove_class;
template<typename C, typename R, typename... A>
struct remove_class<R(C::*)(A...)> { using type = R(A...); };
template<typename C, typename R, typename... A>
struct remove_class<R(C::*)(A...) const> { using type = R(A...); };
template<typename C, typename R, typename... A>
struct remove_class<R(C::*)(A...) volatile> { using type = R(A...); };
template<typename C, typename R, typename... A>
struct remove_class<R(C::*)(A...) const volatile> { using type = R(A...); };
```

Maybe this should be in the standard.

optional_override Implementation

```
template<typename LambdaType>
using LambdaSignature =
  typename remove_class<
    decltype(&LambdaType::operator())
  >::type;
```

Give a lambda, we can grab a pointer to its call operator, remove the “member-ness” from the call operator’s type, and then we have the appropriate function type for the lambda.

optional_override Implementation

```
template<typename LambdaType>
LambdaSignature<LambdaType>*
optional_override(const LambdaType& fp) {
   return fp;
}
```

optional_override simply calls the lambda's implicit function pointer conversion operator.

WHEW…

Overview

- C++ has bright future on the web
- C++ libraries now available to JavaScript
- C++ can call JavaScript code
- Low-overhead: 200 ns overhead per call
  - more optimizations possible!
- Emphasis on small generated code size
- Without C++11, writing embind would have been really annoying
- Hope you learned a few tricks!

We're Hiring!
Questions?

@chadaustin
chad@imvu.com
http://emscripten.org/