Typed Arrays: Binary Data in the Browser

By Ilmari Heikkinen

Published: July 20th, 2012
Comments: 0

Introduction

Typed Arrays are a relatively recent addition to browsers, born out of the need to have an efficient way to handle binary data in WebGL. A Typed Array is a slab of memory with a typed view into it, much like how arrays work in C. Because a Typed Array is backed by raw memory, the JavaScript engine can pass the memory directly to native libraries without having to painstakingly convert the data to a native representation. As a result, typed arrays perform a lot better than JavaScript arrays for passing data to WebGL and other APIs dealing with binary data.

Typed array views act like single-type arrays to a segment of an ArrayBuffer. There are views for all the usual numeric types, with self-descriptive names like Float32Array, Float64Array, Int32Array and Uint8Array. There's also a special view which has replaced the pixel array type in Canvas's ImageData: Uint8ClampedArray.

DataView is the second type of view and it is meant for handling heterogeneous data. Instead of having an array-like API, the DataView object provides you a get/set API to read and write arbitrary data types at arbitrary byte offsets. DataView works great for reading and writing file headers and other such struct-like data.

Basics of using Typed Arrays

Typed array views

To use Typed Arrays, you need to create an ArrayBuffer and a view to it. The easiest way is to create a typed array view of the desired size and type.

```
// Typed array views work pretty much like normal arrays.
var f64a = new Float64Array(8);
f64a[0] = 10;
f64a[1] = 20;
f64a[2] = f64a[0] + f64a[1];
```

There are several different types of typed array views. They all share the same API, so once you know how to use one, you pretty much know how to use them all. I'm going to create one of each currently existing typed array views in the next example.

```
// Floating point arrays.
var f64 = new Float64Array(8);
var f32 = new Float32Array(16);

// Signed integer arrays.
var i32 = new Int32Array(16);
var i16 = new Int16Array(32);
var i8 = new Int8Array(64);

// Unsigned integer arrays.
var u32 = new Uint32Array(16);
var u16 = new Uint16Array(32);
var u8 = new Uint8Array(64);
var pixels = new Uint8ClampedArray(64);
```

The last one is a bit special, it clamps input values between 0 and 255. This is especially handy for Canvas image processing algorithms since now you don’t have to manually clamp your image processing math to avoid overflowing the 8-bit range.

For example, here’s how you’d apply a gamma factor to an image stored in a Uint8Array. Not very pretty:

```
u8[i] = Math.min(255, Math.max(0, u8[i] * gamma));
```

With Uint8ClampedArray you can skip the manual clamping:

```
pixels[i] *= gamma;
```

The other way to create typed array views is to create an ArrayBuffer first and then create views that point to it. The APIs that get you external data usually deal in ArrayBuffers, so this is the way you get a typed array view to those.

```
var ab = new ArrayBuffer(256); // 256-byte ArrayBuffer.
var faFull = new Uint8Array(ab);
var faFirstHalf = new Uint8Array(ab, 0, 128);
var faThirdQuarter = new Uint8Array(ab, 128, 64);
var faRest = new Uint8Array(ab, 192);
```

You can also have several views to the same ArrayBuffer.

```
var fa = new Float32Array(64);
var ba = new Uint8Array(fa.buffer, 0,
Float32Array.BYTES_PER_ELEMENT); // First float of fa.
```

To copy a typed array to another typed array, the fastest way is to use the typed array set method. For a memcpy-like use, create Uint8Arrays to the buffers of the views and use set to copy the data over.

```
function memcpy(dst, dstOffset, src, srcOffset, length) {
  var dstU8 = new Uint8Array(dst, dstOffset, length);
  var srcU8 = new Uint8Array(src, srcOffset, length);
  dstU8.set(srcU8);
};
```

DataView

To use ArrayBuffers that contain data with heterogenous types, the easiest way is to use a DataView to the buffer. Suppose we have a file format that has a header with an 8-bit unsigned int followed by two 16-bit ints, followed by a payload array of 32-bit floats. Reading this back with typed array views is doable but a bit of a pain. With a DataView we can read the header and use a typed array view for the float array.

```
var dv = new DataView(buffer);
var vector_length = dv.getUint8(0);
var width = dv.getUint16(1); // 0+uint8 = 1 bytes offset
var height = dv.getUint16(3); // 0+uint8+uint16 = 3 bytes offset
var vectors = new Float32Array(width*height*vector_length);
for (var i=0, off=5; i<vectors.length; i++, off+=4) {
  vectors[i] = dv.getFloat32(off);
}
```

In the above example, all the values I read are big-endian. If the values in the buffer are little-endian, you can pass the optional littleEndian parameter to the getter:

```
...
var width = dv.getUint16(1, true);
var height = dv.getUint16(3, true);
...
vectors[i] = dv.getFloat32(off, true);
...
```

Note that typed array views are always in the native byte order. This is to make them fast. You should use a DataView to read and write data where endianness is going to be an

issue.

The DataView also has methods for writing values to buffers. These setters are named in the same fashion as the getters, "set" followed by the data type.

```
dv.setInt32(0, 25, false); // set big-endian int32 at byte offset
0 to 25
dv.setInt32(4, 25); // set big-endian int32 at byte offset 4 to 25
dv.setFloat32(8, 2.5, true); // set little-endian float32 at byte
offset 8 to 2.5
```

A discussion of endianness

Endianness, or byte order, is the order in which multi-byte numbers are stored in the computer's memory. The term *big-endian* describes a CPU architecture which stores the most significant byte first; *little-endian*, the least significant byte first. Which endianness is used in a given CPU architecture is completely arbitrary; there are good reasons to choose either one. In fact, some CPUs can be configured to support both big-endian and little-endian data.

Why do you need to be concerned about endianness? The reason is simple. When reading or writing data from the disk or the network, the endianness of the data must be specified. This ensures that the data is interpreted properly, regardless of the endianness of the CPU that is working with it. In our increasingly networked world, it is essential to properly support all kinds of devices, big- or little-endian, that might need to work with binary data coming from servers or other peers on the network.

The DataView interface is specifically designed to read and write data to and from files and the network. DataView operates upon data with a *specified endianness*. The endianness, big or little, must be specified with every access of every value, ensuring that you get consistent and correct results when reading or writing binary data, no matter what the endianness of the CPU on which the browser is running.

Typically, when your application reads binary data from a server, you'll need to scan through it once in order to convert it into the data structures your application uses internally. DataView should be used during this phase. It's not a good idea to use the multi-byte typed array views (Int16Array, Uint16Array, etc.) directly with data fetched via XMLHttpRequest, FileReader, or any other input/output API, because the typed array views use the CPU's native endianness. More on this later.

Let's look at a couple of simple examples. The Windows BMP file format used to be the standard format for storing images in the early days of Windows. The documentation linked above clearly indicates that all of the integer values in the file are stored in little-endian format. Here is a snippet of code which parses the beginning of the BMP header using the DataStream.js library which accompanies this article:

```
function parseBMP(arrayBuffer) {
  var stream = new DataStream(arrayBuffer, 0,
    DataStream.LITTLE_ENDIAN);
  var header = stream.readUint8Array(2);
  var fileSize = stream.readUint32();
  // Skip the next two 16-bit integers
  stream.readUint16();
  stream.readUint16();
  var pixelOffset = stream.readUint32();
  // Now parse the DIB header
  var dibHeaderSize = stream.readUint32();
  var imageWidth = stream.readInt32();
  var imageHeight = stream.readInt32();
  // ...
}
```

Here's another example, this one from the High Dynamic Range rendering demo in the WebGL samples project. This demo downloads raw, little-endian floating-point data representing high dynamic range textures, and needs to upload it to WebGL. Here is the snippet of code which properly interprets the floating-point values on all CPU architectures. Assume the variable "arrayBuffer" is an ArrayBuffer which has just been downloaded from the server via XMLHttpRequest:

```
var arrayBuffer = ...;
var data = new DataView(arrayBuffer);
var tempArray = new Float32Array(
  data.byteLength / Float32Array.BYTES_PER_ELEMENT);
var len = tempArray.length;
// Incoming data is raw floating point values
// with little-endian byte ordering.
for (var jj = 0; jj < len; ++jj) {
  tempArray[jj] =
    data.getFloat32(jj * Float32Array.BYTES_PER_ELEMENT, true);
}
gl.texImage2D(...other arguments...,
  gl.RGB, gl.FLOAT, tempArray);
```

The rule of thumb is: upon receiving binary data from the web server, make one pass over it with a DataView. Read the individual numeric values and store them in some other data structure, either a JavaScript object (for small amounts of structured data) or a typed array view (for large blocks of data). This will ensure that your code works correctly on all kinds of CPUs. Also use DataView to write data to a file or the network, and make sure to appropriately specify the *littleEndian* argument to the various *set* methods in order to produce the file format you are creating or using.

Remember, all data that goes over the network implicitly has a format and an endianness (at least, for any multi-byte values). Make sure to clearly define and document the format of

all data your application sends over the network.

Browser APIs that use Typed Arrays

I'm going to give you a brief overview of the different browser APIs that are currently using Typed Arrays. The current crop includes WebGL, Canvas, Web Audio API, XMLHttpRequests, WebSockets, Web Workers, Media Source API and File APIs. From the list of APIs you can see that Typed Arrays are well suited for performance-sensitive multimedia work as well as passing data around in an efficient fashion.

WebGL

The first use of Typed Arrays was in WebGL, where it is used to pass around buffer data and image data. To set the contents of a WebGL buffer object, you use the gl.bufferData() call with a Typed Array.

```
var floatArray = new Float32Array([1,2,3,4,5,6,7,8]);
gl.bufferData(gl.ARRAY_BUFFER, floatArray);
```

Typed Arrays are also used to pass around texture data. Here's a basic example of passing in texture content using a Typed Array.

```
var pixels = new Uint8Array(16*16*4); // 16x16 RGBA image
gl.texImage2D(
  gl.TEXTURE_2D, // target
  0, // mip level
  gl.RGBA, // internal format
  16, 16, // width and height
  0, // border
  gl.RGBA, //format
  gl.UNSIGNED_BYTE, // type
  pixels // texture data
);
```

You also need Typed Arrays to read pixels from the WebGL context.

```
var pixels = new Uint8Array(320*240*4); // 320x240 RGBA image
gl.readPixels(0, 0, 320, 240, gl.RGBA, gl.UNSIGNED_BYTE, pixels);
```

Canvas 2D

Recently the Canvas ImageData object was made to work with the Typed Arrays spec. Now you can get a Typed Arrays representation of the pixels on a canvas element. This is

helpful since now you can also create and edit canvas pixel arrays without having to fiddle around with the canvas element.

```
var imageData = ctx.getImageData(0,0, 200, 100);
var typedArray = imageData.data // data is a Uint8ClampedArray
```

XMLHttpRequest2

XMLHttpRequest got a Typed Array boost and now you can receive a Typed Array response instead of having to parse a JavaScript string into a Typed Array. This is really neat for passing fetched data directly to multimedia APIs and for parsing binary files fetched from the network.

All you have to do is set the responseType of the XMLHttpRequest object to 'arraybuffer'.

```
xhr.responseType = 'arraybuffer';
```

Recall that you must be aware of endianness issues when downloading data from the network! See the section on endianness above.

File APIs

The FileReader can read file contents as an ArrayBuffer. You can then attach typed array views and DataViews to the buffer to manipulate its contents.

```
reader.readAsArrayBuffer(file);
```

You should keep endianness in mind here as well. Check out the endianness section for details.

Transferable objects

Transferable objects in postMessage make passing binary data to other windows and Web Workers a great deal faster. When you send an object to a Worker as a Transferable, the object becomes inaccessible in the sending thread and the receiving Worker gets ownership of the object. This allows for a highly optimized implementation where the sent data is not copied, just the ownership of the Typed Array is transferred to the receiver.

To use Transferable objects with Web Workers, you need to use the webkitPostMessage method on the worker. The webkitPostMessage method works just like postMessage, but it takes two arguments instead of just one. The added second argument is an array of objects you wish to transfer to the worker.

```
worker.webkitPostMessage(oneGBTypedArray, [oneGBTypedArray]);
```

To get the objects back from the worker, the worker can pass them back to the main thread in the same fashion.

```
webkitPostMessage({results: grand, youCanHaveThisBack:
oneGBTypedArray}, [oneGBTypedArray]);
```

Zero copies, woo!

Media Source API

Recently, the media elements also got some Typed Array goodness in the form of the Media Source API. You can directly pass a Typed Array containing video data to a video element using webkitSourceAppend. This makes the video element append the video data after the existing video. SourceAppend is awesome for doing interstitials, playlists, streaming and other uses where you might want to play several videos using a single video element.

```
video.webkitSourceAppend(uint8Array);
```

Binary WebSockets

You can also use Typed Arrays with WebSockets to avoid having to stringify all your data. Great for writing efficient protocols and minimizing network traffic.

```
socket.binaryType = 'arraybuffer';
```

Whew! That wraps up the API review. Let's move onto looking at third-party libraries for handling Typed Arrays.

Third-party libraries

jDataView

jDataView implements a DataView shim for all browsers. DataView used to be a WebKit-only feature, but now it's supported by most other browsers. The Mozilla developer team is in the process of landing a patch to enable DataView on Firefox as well.

Eric Bidelman on the Chrome Developer Relations team wrote a small MP3 ID3 tag reader example that uses jDataView. Here's a usage example from the blog post:

```
var dv = new jDataView(arraybuffer);

// "TAG" starts at byte -128 from EOF.
// See http://en.wikipedia.org/wiki/ID3
if (dv.getString(3, dv.byteLength - 128) == 'TAG') {
  var title = dv.getString(30, dv.tell());
  var artist = dv.getString(30, dv.tell());
  var album = dv.getString(30, dv.tell());
  var year = dv.getString(4, dv.tell());
} else {
  // no ID3v1 data found.
}
```

stringencoding

Working with strings in Typed Arrays is a bit of a pain at the moment, but there's the stringencoding library that helps there. Stringencoding implements the proposed Typed Array string encoding spec, so it's also a good way to get a feel for what's coming.

Here's a basic usage example of stringencoding:

```
var uint8array = new TextEncoder(encoding).encode(string);
var string = new TextDecoder(encoding).decode(uint8array);
```

BitView.js

I have written a small bit manipulation library for Typed Arrays called BitView.js. As the name says, it works much like the DataView, except it works with bits. With BitView you can get and set the value of a bit at a given bit offset in an ArrayBuffer. BitView also has methods for storing and loading 6-bit and 12-bit ints at arbitrary bit offsets.

12-bit ints are nice for working with screen coordinates, as displays tend to have fewer than 4096 pixels along the longer dimension. By using 12-bit ints instead of 32-bit ints, you get a 62% size reduction. For a more extreme example, I was working with Shapefiles that use 64-bit floats for the coordinates, but I didn't need the precision because the model was only going to be shown at screen size. Switching over to 12-bit base coordinates with 6-bit deltas to encode changes from the previous coordinate brought the filesize down to a tenth. You can see the demo of that at here. Here's an example of using BitView.js:

```
var bv = new BitView(arrayBuffer);
bv.setBit(4, 1); // Set fourth bit of arrayBuffer to 1.
bv.getBit(17); // Get 17th bit of arrayBuffer.
```

```
bv.getBit(50*8 + 3); // Get third bit of 50th byte in arrayBuffer.

bv.setInt6(3, 18); // Write 18 as a 6-bit int to bit position 3 in
arrayBuffer.
bv.getInt12(9); // Read a 12-bit int from bit position 9 in
arrayBuffer.
```

DataStream.js

One of the most exciting things about typed arrays is how they make it easier to deal with binary files in JavaScript. Instead of parsing a string character by character and manually converting the characters into binary numbers and such, you can now get an ArrayBuffer with XMLHttpRequest and directly process it using a DataView. This makes it easy to e.g. load in an MP3 file and read the metadata tags for use in your audio player. Or load in a shapefile and turn it into a WebGL model. Or read the EXIF tags off a JPEG and show them in your slideshow app.

The problem with ArrayBuffer XHRs is that reading struct-like data from the buffer is a bit of a pain. DataView is good for reading a few numbers at a time in an endian-safe fashion, typed array views are good for reading arrays of element-size-aligned native endian numbers. What we felt missing is a way to read in arrays and structs of data in a convenient endian-safe fashion. Enter DataStream.js.

DataStream.js is a Typed Arrays library that read and writes scalars, strings, arrays and structs of data from ArrayBuffers in a file-like fashion.

Example of reading in an array of floats from an ArrayBuffer:

```
// without DataStream.js
var dv = new DataView(buffer);
var f32 = new Float32Array(buffer.byteLength / 4);
var littleEndian = true;
for (var i = 0; i<f32.length; i++) {
  f32[i] = dv.getFloat32(i*4, littleEndian);
}

// with DataStream.js
var ds = new DataStream(buffer);
ds.endianness = DataStream.LITTLE_ENDIAN;
var f32 = ds.readFloat32Array(ds.byteLength / 4);
```

Where DataStream.js gets really useful is in reading more complex data. Suppose you have a method that reads in JPEG markers:

```
// without DataStream.js
var dv = new DataView(buffer);
var objs = [];
```

```
for (var i=0; i<buffer.byteLength;) {
  var obj = {};
  obj.tag = dv.getUint16(i);
  i += 2;
  obj.length = dv.getUint16(i);
  i += 2;
  obj.data = new Uint8Array(obj.length - 2);
  for (var j=0; j<obj.data.length; j++,i++) {
    obj.data[j] = dv.getUint8(i);
  }
  objs.push(obj);
}

// with DataStream.js
var ds = new DataStream(buffer);
ds.endianness = ds.BIG_ENDIAN;
var objs = [];
while (!ds.isEof()) {
  var obj = {};
  obj.tag = ds.readUint16();
  obj.length = ds.readUint16();
  obj.data = ds.readUint8Array(obj.length - 2);
  objs.push(obj);
}
```

Or use the DataStream.readStruct method to read in structs of data. The readStruct method takes in a struct definition array that contains the types of the struct members. It's got callback functions for handling complex types and handles arrays of data and nested structs as well:

```
// with DataStream.readStruct
ds.readStruct([
  'objs', ['[]', [ // objs: array of tag,length,data structs
    'tag', 'uint16',
    'length', 'uint16',
    'data', ['[]', 'uint8', function(s,ds){ return s.length - 2;
}], // get length with a function
  '*'] // read in as many struct as there are
]);
```

As you can see, the struct definition is a flat array of [name, type]-pairs. Nested structs are done by having an array for the type. Arrays are defined by using a three-element array where the second element is the array element type and the third element is the array length (either as a number, as a reference to previously read field or as a callback function). The first element of the array definition is unused.

The possible values for the type are as follows:

```
Number types

Unsuffixed number types use DataStream endianness.
To explicitly specify endianness, suffix the type with
'le' for little-endian or 'be' for big-endian,
e.g. 'int32be' for big-endian int32.

  'uint8' -- 8-bit unsigned int
  'uint16' -- 16-bit unsigned int
  'uint32' -- 32-bit unsigned int
  'int8' -- 8-bit int
  'int16' -- 16-bit int
  'int32' -- 32-bit int
  'float32' -- 32-bit float
  'float64' -- 64-bit float

String types

  'cstring' -- ASCII string terminated by a zero byte.
  'string:N' -- ASCII string of length N.
  'string,CHARSET:N' -- String of byteLength N encoded with
given CHARSET.
  'u16string:N' -- UCS-2 string of length N in DataStream
endianness.
  'u16stringle:N' -- UCS-2 string of length N in little-
endian.
  'u16stringbe:N' -- UCS-2 string of length N in big-endian.

Complex types

  [name, type, name_2, type_2, ..., name_N, type_N] -- Struct

  function(dataStream, struct) {} -- Callback function to read
and return data.

  {get: function(dataStream, struct) {}, set:
function(dataStream, struct) {}}
  -- Getter/setter functions to reading and writing data.
Handy for using the
     same struct definition for both reading and writing.

  ['', type, length] -- Array of given type and length. The
length can be either
```

```
                        a number, a string that references a
previously-read
                        field, or a callback function(struct,
dataStream, type){}.
                        If length is set to '*', elements are
read from the
                        DataStream until a read fails.
```

You can see a live example of reading in JPEG metadata here. The demo's using DataStream.js for reading the tag-level structure of the JPEG file (along with some EXIF parsing), and jpg.js for decoding and displaying the JPEG image in JavaScript.

History of Typed Arrays

Typed Arrays got their start in the early implementation stage of WebGL, when we found that passing JavaScript arrays to the graphics driver was causing performance problems. With JavaScript arrays, the WebGL binding had to allocate a native array and fill it by walking over the JavaScript array and cast every JavaScript object in the array to the required native type.

To fix the data conversion bottleneck, Mozilla's Vladimir Vukicevic wrote CanvasFloatArray: a C-style float array with a JavaScript interface. Now you could edit the CanvasFloatArray in JavaScript and pass it directly to WebGL without having to do any extra work in the binding. In further iterations, CanvasFloatArray was renamed to WebGLFloatArray, which was further renamed to Float32Array and split into a backing ArrayBuffer and the typed Float32Array-view to access the buffer. Types were also added for other integer and floating-point sizes and signed/unsigned variants.

Design considerations

From the beginning, the design of Typed Arrays was driven by the need to efficiently pass binary data to native libraries. For this reason, the typed array views operate upon aligned data in the host CPU's *native endianness*. These decisions make it possible for JavaScript to reach maximum performance during operations such as sending vertex data to the graphics card.

DataView is specifically designed for file and network I/O, where the data always has a *specified endianness*, and might not be aligned for maximum performance.

The design split between in-memory data assembly (using the typed array views) and I/O (using DataView) was a conscious one. Modern JavaScript engines optimize the typed array views heavily, and achieve high performance on numerical operations with them. The current levels of performance of the typed array views were made possible by this design decision.

References

- Typed Array Specification
- jDataView
- DataStream.js
- BitView.js
- stringencoding
- On webkitSourceAppend