



Clipcode's Guide to the Typed Array Spec

Specification

<http://www.khronos.org/registry/typedarray/specs/latest/>

Compatibility

<http://caniuse.com/#feat=webgl> (Implementors of WebGL will also implement Typed Array Spec)

Overview

The Typed Array Specification is used to help Javascript work efficiently with binary data.

Until recently, Javascript worked exclusively with data as text or as opaque blobs (e.g. A JPEG, which was simply rendered). It was very difficult to efficiently edit and interact with binary data from within script in the browser.

Starting with the WebGL 3D specification, there is a growing need to work with binary, and hence the Typed Array Specification. It was created by the Khronos Group, a standards body focused on graphics, including OpenGL and WebGL. Use of binary goes far beyond 3D, and so is actively being investigated with Web Workers, Indexed DB API, File API and Web Messaging.

Concepts

The Typed Array Spec contains the following constructs (API definition are from khronos.org and copyrighted by them):

ArrayBuffer – The underlying buffer

```
[ Constructor(unsigned long length) ]  
interface ArrayBuffer {  
    readonly attribute unsigned long byteLength;  
    readonly attribute boolean readOnly;  
  
    ArrayBuffer slice(long begin, optional long end);  
};
```

ArrayBufferView is a view of a buffer.

```
interface ArrayBufferView {  
    readonly attribute ArrayBuffer buffer;  
    readonly attribute unsigned long byteOffset;  
    readonly attribute unsigned long byteLength;  
};
```

Note that ArrayBufferView is not derived from ArrayBuffer, rather it makes an instance of one available as an attribute.

TypedArray – arrays of specific types

There are typed array view types for Int8, Uint8, Int16, Uint16, Int32, Uint32, Float32 and Float64.

These are type-specific derivations of ArrayBufferView.

```
[
  Constructor(unsigned long length),
  Constructor(TypedArray array),
  Constructor(type[] array),
  Constructor(ArrayBuffer buffer,
    optional unsigned long byteOffset, optional unsigned long length)
]
interface TypedArray : ArrayBufferView {
  const unsigned long BYTES_PER_ELEMENT = element size in bytes;

  readonly attribute unsigned long length;

  omittable getter type get(unsigned long index);
  omittable setter void set(unsigned long index, type value);
  void set(TypedArray array, optional unsigned long offset);
  void set(type[] array, optional unsigned long offset);
  TypedArray subarray(long begin, optional long end);
};
```

The various TypedArray constructs derive from ArrayBufferView and provides setters and getters and a way to make subarrays.

DataView View Type

A view to support endianism and alignment.

```
[
  Constructor(ArrayBuffer buffer,
    optional unsigned long byteOffset,
    optional unsigned long byteLength)
]
interface DataView : ArrayBufferView {
  // Gets the value of the given type at the specified byte offset
  // from the start of the view. There is no alignment constraint;
  // multi-byte values may be fetched from any offset.
  //
  // For multi-byte values, the optional littleEndian argument
  // indicates whether a big-endian or little-endian value should be
  // read. If false or undefined, a big-endian value is read.
  //
  // These methods raise an exception if they would read
  // beyond the end of the view.
  byte getInt8(unsigned long byteOffset);
  unsigned byte getUint8(unsigned long byteOffset);
  short getInt16(unsigned long byteOffset,
    optional boolean littleEndian);
  unsigned short getUint16(unsigned long byteOffset,
    optional boolean littleEndian);
  long getInt32(unsigned long byteOffset,
    optional boolean littleEndian);
  unsigned long getUint32(unsigned long byteOffset,
    optional boolean littleEndian);
  float getFloat32(unsigned long byteOffset,
    optional boolean littleEndian);
  double getFloat64(unsigned long byteOffset,
    optional boolean littleEndian);
};
```

```
// Stores a value of the given type at the specified byte offset
// from the start of the view. There is no alignment constraint;
// multi-byte values may be stored at any offset.
//
// For multi-byte values, the optional littleEndian argument
// indicates whether the value should be stored in big-endian or
// little-endian byte order. If false or undefined, the value is
// stored in big-endian byte order.
//
// These methods raise an exception if the underlying ArrayBuffer
// is read-only, or if they would write beyond the end of the view.
void setInt8(unsigned long byteOffset,
             byte value);
void setUint8(unsigned long byteOffset,
              unsigned byte value);
void setInt16(unsigned long byteOffset,
              short value,
              optional boolean littleEndian);
void setUint16(unsigned long byteOffset,
               unsigned short value,
               optional boolean littleEndian);
void setInt32(unsigned long byteOffset,
              long value,
              optional boolean littleEndian);
void setUint32(unsigned long byteOffset,
               unsigned long value,
               optional boolean littleEndian);
void setFloat32(unsigned long byteOffset,
                float value,
                optional boolean littleEndian);
void setFloat64(unsigned long byteOffset,
                 double value,
                 optional boolean littleEndian);
};
```