wv: A General Web-based 3D Viewer

Bob Haimes (haimes@mit.edu)
Aerospace Computational Design Laboratory
Department of Aeronautics & Astronautics
Massachusetts Institute of Technology

March 2013

Objective

The objective of this work is to generate a **visual** tool targeted for the 3D needs found within the MDAO process. A **WebBrowser-based** approach is considered, in that it provides the broadest possible platform for deployment.

Outline

- **System Architecture**
  - Browser / WebGL / WebSockets
  - Server or Data Generator(s)
- **Data Model**
  - VBO based
  - Primitives
  - Graphic Objects
- **Functionality at the Viewer**
  - IO Handling
  - Rendering / GUI Loop
- **Binary Data Packets**
- **GUI Call-backs**
- **Procedural-based Server-side API**

System Architecture

Goal: Effective 3D component that can support the viewing of:

- Geometry
- Meshing
- Scientific Visualization Tools (including transient data)
- Multi-dimensional Design Space Examination
- Other 3D needs

Contains no GUI but the *hooks* (in the form of *JavaScript* call-backs) to graft a customized UI specifically designed for the task at hand.

System Architecture -- Viewer

Efficient Browser Implementation

- Must support WebGL (& *JavaScript*)
- Use of WebSockets (part of HTML5)
  - Asynchronicity
  - Segregation of data-streams (via *protocols*)
  - Data handling consistent with WebGL
- Extensive use of Vertex Buffer Objects (**VBO**s)
- IO from the *server*
  - Packed messages -- few network packets that are unpacked into typed *JavaScript* Arrays at the Browser
  - Binary -- known common types, allows avoiding the costs of serialization / deserialization (WebSocket *binary*)
  - Techniques to provide data to the GUI (WebSocket *text*)
- IO to the *server*
  - Nothing from the Viewer by default
  - Only data from the customized GUI (WebSocket *text*)

System Architecture -- Server

Data Generation and Handling

- Web server (or acts as one -- *libwebsockets*)
- Visualization state must be maintained (note: Viewer is stateless except for viewMatrix & current plotting attributes)
- **VBO** components generated and sent
- IO to the Viewer
  - Aggregated **VBO**s with metadata
  - What is sent is based on changes from the GUI or from transient data
  - Data for the GUI portion of the Viewer
- IO from the Viewer
  - Only data from the customized GUI

Data Model -- VBO Components

- *Vertices*
  - Coordinates (3 by *float* -- *Float32Array*)
  - length
- *Indices* (optional)
  - The index into the Vertex Array (*unsigned short* -- *Uint16Array*)
  - length (can be different from Vertex length)
- *Colors* (optional)
  - RGBs associated with the Vertices(3 by *unsigned byte* -- *Uint8Array*)
  - Must be same length as *Vertices*
- *Normals* (optional, used for *Triangles* or Decorated *Lines*)
  - The normal pointing vector for lighting (3 by *float* -- *Float32Array*) or Decorated *Triangles* and normals (no stripes)
  - Must be same length as *Vertices* (*Triangles*)

Note: the *unsigned short* of *Indices* limits the size of the VBO used, so larger data needs to be *striped.*

Data Model -- VBO Types

- *Points*
  - *Vertices* [*Colors* & *Indices*]
- *Lines* (2 vertices per -- disjoint segments)
  - *Vertices* [*Colors* & *Indices*]
  - Optional *Normals* for Decorations (i.e. 3D Arrows)
- *Triangles* (3 vertices per -- also disjoint)
  - *Vertices* [*Normals*, *Colors* & *Indices*]

Notes:

1. Constant element coloring of *Lines*/*Triangles* requires non-indexed **VBO**s and the duplication of color information (per vertex)
2. Facetted lighting requires similar treatment with *Normals*
3. Any non-planar set of *Triangles* requires *Normals* **VBO** component

Data Model -- Graphic Primitives

- *Locations* (GPType 0 -- 0D)
  - Collections of one or more *Points*
  - Foreground Color
  - Size (in pixels)
  - Coloring & Transparency Flags

- *Disjoint Lines* (GPType 1 -- 1D)
  - Optional collected Indexed *Points* into the *Lines* Vertex Array
  - Collections of one or more *Lines*
  - Line Color
  - Foreground Color for Decorations
  - Back-facing Color for Decorations
  - Line Width (in pixels)
  - Point Color
  - Point size (in pixels)
  - Coloring & Transparency Flags

Data Model -- Graphic Primitives

- *Disjoint Triangles* (GPType 2 -- 2D)
  - Optional collected Indexed *Points* into the *Triangles* Vertex Array
  - Optional collected Indexed *Lines* into the *Triangles* Vertex Array
  - Collections of one or more *Triangles*
  - Foreground Color
  - Back-facing Color
  - Planar Normal (if planar)
  - Line Color
  - Line Width (in pixels)
  - Point Color
  - Point size (in pixels)
  - Coloring, Transparency, Orientation & *Point*/*Line* Flags

Note: Simple two-sided (ambient & diffuse) lighting is applied by default (modification to **wv_render.js** is required for other lighting models)

Data Model -- Graphic Objects

- ***Graphic Object***
  - ID -- Unique character string assigned by the *server*
  - GPType
  - Number of *Striped* Primitives in the Collection
  - GPType specific metadata
  - Graphic Primitive data

- **VBO** Internal Reference

– ID	*string*
– Stripe #	*24bits*
– One of *Point*, *Line*, *Triangle* Data (3)	*byte*
– One of *Vertices*, *Indices*, *Colors*, *Normals* (4)	*byte*

Functionality at the Viewer

IO Handling

- Initialize (connect to *server*)
- Handshake to ensure compatibility

- Arrays generated by "unpacking" received **VBO**s (with metadata) via *binary protocol*
- Handle any GUI related data (*text protocol*) via the call-back *wvServerMessage*
- Continue until End-of-Frame marker
- Inform *Rendering Loop* that there is new data and accept no new data until released

Asynchronously performed by WebSocket event handling

Functionality at the Viewer

Rendering / GUI Loop

- Initialize (generate canvas on WebGL context)
- Execute GUI setup call-back *wvInitUI*

1. Setup scene
   - Blank canvas and depth buffer
   - Adjust viewMatrix (*wvUpdateView* call-back)
2. Render any *Graphics Objects*
3. Add custom renderings by call-back *wvUpdateCanvas*
4. Execute GUI call-back *wvUpdateUI*
5. Do we have an End-of-Frame marker?
   - If **no** -- has anything changed in the GUI?
     - No -- Wait then goto 4
     - Yes -- goto 1
6. Handshake with IO Handling, update the *Graphics Objects* & release the IO hold
7. goto 1

Functionality at the Viewer

Rendering Model

- WebGL requires fragment & vertex shaders
- Lighting & texture mapping done in the shaders
- The supplied shaders support:
  - Two-sided lighting
  - Ambient & Diffuse lighting model
  - Back-face coloring
  - Constant and/or linearly interpolated color-space mapping
  - Simple transparency
  - Picking
  - Bumping of lines forward (in screen Z)
- Any other requirements will involve modifying the shaders (which can be found in **wv-render.js**)

Binary Data Packets

- Individual data collections should be aggregated to reduce network latencies -- large packets
- All data is tightly packed and **VBO** "ready"
- Data collections begin with an Opcode (1 byte):
  - 0 -- end of packet (but not End-of-Frame)
  - 1 -- new Graphic Object
  - 2 -- delete Graphic Object
  - 3 -- new Data for Graphic Object
  - 4 -- update Data in Graphic Object
  - 7 -- End-of-Frame Marker (must be last in total packet)
  - 8 -- Initialize Packet
- All data is aligned on 4-byte boundaries
  - Colors are *unsigned byte*
  - Indices are *unsigned short*
  - The ID is a *string*

Binary Data Packets

- Each collection starts with:
  - Opcode (MSB)
  - Stripe # or Number of Stripes (3 bytes -- LSB)
  - Complete for Opcode 0 & 7
- Next 32 bits (all but Opcode 0, 7 & 8):
  - GPType (1 byte -- MSB)
  - vflag -- bits can be summed (1 btye):
    - Vertices 1
    - Indices 2
    - Colors 4
    - Normals 8
    - Point Indices 16
    - Line Indices 32
  - ID character Length (integer factor of 4) (2 bytes -- LSB)
- ID Character string (number of bytes above)

Opcode 2 (delete) requires no more data

Binary Data Packets

- Opcode 1 (new Graphic Object)
  - Plotting Attributes (bit flag -- *int*):
    1 - Render On
    2 - Transparent
    4 - Color Interpolation
    8 - Show orientation
    16 - Plot Points
    32 - Plot Lines
  - Point size (*float*)
  - Point color (3**float*)

    [Done for Point Objects]

  - Line width (*float*)
  - Line color (3**float*)
  - Foreground constant color (3**float*)
  - Background color (3**float*)

    [Done for Line Objects]

  - Constant Normal (3**float*)

Binary Data Packets

- Opcode 3 (new data) & 4 (update data)
  - Number of data elements for the Graphic Primitive stripe (*int*):
    - Total number of primitive *words* is found by multiplying by 3 for Vertices (xyz), Colors (rgb) & Normals
    - Applying "*sizeof()*" to the above provides the total byte length (plus any required padding)
  - The **VBO** data (type based on bit in *vflag*)
  - *Repeated for each bit in vflag in LSB order (Opcode 3), i.e. vertices always first*

Notes:

- Opcode 4 can only have a single bit in *vflag* set
- Data types shorter than 32 bits must be padded at the end so that the next read can be 4-byte aligned

Binary Data Packets

- Opcode 8 (Initialize) -- 56 bytes long
  - Opcode field (4*bytes*)
  - Field of View (*float*)
  - zNear (*float*)
  - zFar (*float*)
  - Eye location (3*float*)
  - Focus position (3*float*)
  - Up direction (3*float*)
  - End-of-Frame (4*bytes*)
- Examples of IO routines:
  - Reading in **wv-socket.js**
  - Writing in **wsServer/wv.c**

GUI Call-backs (need to be supplied)

- function **wvInitUI()**
  - Invoked once to initialize the UI variables and state
- function **wvUpdateUI()**
  - Called in the rendering loop so that the state of the UI can be adjusted
  - Note: if the state is modified directly in an event handler the rendering for that frame may be corrupted
- function **wvUpdateView()**
  - Allows for the adjustment of the viewMatrix before the scene is rendered again
- function **wvUpdateCanvas(*gl*)**
  - Allows for the customization of what is rendered by additional WebGL calls
  - ***gl*** is the WebGL context

GUI Call-backs & A Useful Function

- function **wvServerMessage(*text*)**
  - Called when an ASCII ***text*** message has been received from the server (*UI text protocol*)
  - Note: this is invoked from a WebSocket event handler

- **g.socketUt.send(*text*)**
  - g (**wv** globals), socketUt (*UI text* interface)
  - Send the text string ***text*** to the server using WebSockets
  - This should obviously be the routine called to communicate GUI information to the server
  - Can be used from within any call-back

Usage examples can be found in **SimpleUI.js**

wv Status

- Viewer
  - Tested against:
    - Google Chrome
    - Mozilla FireFox (& SeaMonkey)
    - Apple Safari (at Rev 6.0 or higher)
  - Greater than 18 MegaTriangles per second for large **VBO**s on a MacBook Pro i7 2.8MHz 15" Mid-2010 (Chrome about 20% slower than SeaMonkey)

- Server-like Implementation
  - Python options:
    - pywebsockets
    - ws4py
    - gevent-websocket
  - Use of **libwebsockets** open source project (http://git.warmcat.com/cgi-bin/cgit/libwebsockets)
    - C API to specify data and allow for GUI IO
    - Used to generate the *Procedural-based Server-side API*

Procedural-based Server-side API

- **createContext**

```
wvContext *context =
     wv_createContext(int bias,   float  fov, float  zNear,
                      float zFar, float *eye, float *center,
                      float *up )
call iv_createContext(I*4 bias,   R*4    fov,  R*4   zNear,
                      R*4   zFar, R*4    eye,  R*4   center,
                      R*4    up,  I*8    context)
```

Initializes a WebViewer Context.

bias	the offset used for indexing (usually either 0 or 1)
fov	the field of view for the perspective (angles)
zNear	the Z value for the clipping plane closest to the observer
zFar	the Z value for the clipping plane farthest from the observer
eye	the position of the observer (X,Y,Z)
center	the focus for the viewing matrix
up	a normalized vector referring to positive Y
context	the returned WebViewer context

Procedural-based Server-side API

- **startServer**

```
status = wv_startServer(int port, char *dev, char *path,
                        char *key, int opts, wvContext *context)
status = iv_startServer(I*4 port, C**   dev, C**   path,
                        C** key,  I*4  opts, I*8        context)
```

Starts a server thread on the WebViewer Context. The calling thread of execution continues. Use `statusServer` to determine the state of the connections.

port	the socket port to use for communication
dev	the network interface device name (can be NULL)
path	the path to locate certificate (if secure transmissions are used)
key	the file path for the private key (if secure transmissions are used)
opts	0, or 1 (Defeat the client mask)
context	the WebViewer context (from `createContext`)
status	the server instance/return status (negative is an error)

Procedural-based Server-side API

- **statusServer**

```
status = wv_statusServer(int server)
status = iv_statusServer(I*4 server)
```

Checks the state of the server connections.

server — the server instance (from `startServer`)
status — the state (negative is an error):
0 - all clients have disconnected
1 - active

- **cleanupServers**

```
     wv_cleanupServers()
call iv_cleanupServers()
```

Cleans up all memory associated with this API. Should be used as the last function in this suite.

Procedural-based Server-side API

- **setData**

```
status = wv_setData(int dtype, int len, void *data,
                    int VBOcomp, wvData *item)
status = iv_setData(I*4 dtype, I*4 len, ANY   data,
                    I*4 VBOcomp, I*8     item)
```

Sets the data associated with an item to be used with **`addGPrim`** and **`modGPrim`**. Striping is internally performed where necessary.

dtype	the type of the data array (see **`wsss.h`** or **`wsserver.inc`**)
len	the number of elements in the data array (*Vertices*, *Normals*, and *Colors* require 3 words per element)
data	the data array of type dtype
VBOcomp	the type of the VBO component (see **`wsss.h`** or **`wsserver.inc`**)
item	the output placement for the item
status	the return status (negative is an error)

Procedural-based Server-side API

- **adjustVerts**

```
     wv_adjustVerts(wvData *item, float *focus)
call iv_adjustVerts(I*8     item, R*4    focus)
```

Allows for the adjustment of the vertex coordinates so they fit into screen coordinates (not clipped away).

item	the *Vertices* component (from `setData`)
focus	a vector of length 4 that is used to adjust the coordinates the first is subtracted from the X coordinate the second is subtracted from the Y coordinate the third is subtracted from the Z coordinate the forth is used to normalize (divide) all coordinates

Procedural-based Server-side API

- **addGPrim**

```
status = wv_addGPrim(wvContext *context, char *name, int gtype,
                     int attrs, int nItems, wvData *items)
status = iv_addGPrim(I*8        context, C**   name, I*4 gtype,
                     I*4 attrs, I*4 nItems, I*8     items)
```

Creates and adds this Graphics Primitive to the scene graph associated with this context.

context	the WebViewer context (from `createContext`)
name	unique (in the scene graph) name of the primitive
gtype	the graphics primitive type: *Point*, *Line*, *Triangle* (see `wsss.h` or `wsserver.inc`)
attrs	the initial plotting attributes (see `wsss.h` or `wsserver.inc`)
nItems	the number of components used to define the primitive
items	the components (from `setData`)
status	the index created for the primitive (where negative is an error)

Procedural-based Server-side API

- **modGPrim**

```
status = wv_modGPrim(wvContext *context, int index,
                     int nItems, wvData *items)
status = iv_modGPrim(I*8        context, I*4 index,
                     I*4 nItems, I*8     items)
```

Modifies an existing Graphics Primitive in scene graph associated with this context.

context	the WebViewer context (from `createContext`)
index	the index created for the primitive (from `addGPrim`)
nItems	the number of components to modify in the primitive
items	the components (from `setData`)
status	the return status (where negative is an error)

Procedural-based Server-side API

- **addArrowHeads**

```
status = wv_addArrowHeads(wvContext *context, int index,
                          float size, int nHeads, int *heads)
status = iv_addArrowHeads(I*8        context, I*4 index,
                          R*4   size, I*4 nHeads, I*4  heads)
```

Add Arrow Head decorations to an existing *Line* Graphics Primitive in scene graph associated with this context.

context	the WebViewer context (from `createContext`)
index	the index created for the primitive (from `addGPrim`)
size	the size of the arrow head
nHead	the number of head definitions
heads	the head definitions (index into the line segments -- if negative the head position (and direction) is associated with the first point in the segment, otherwise it is the second position. This is always bias 1.
status	the return status (where negative is an error)

Procedural-based Server-side API

- **removeGPrim**

```
     wv_removeGPrim(wvContext *context, int index)
call iv_removeGPrim(I*8        context, I*4 index)
```

Removes an existing Graphics Primitive in scene graph associated with this context.

context	the WebViewer context (from `createContext`)
index	the index created for the primitive (from `addGPrim`)

- **removeAll**

```
     wv_removeGPrim(wvContext *context)
call iv_removeGPrim(I*8        context)
```

Removes all Graphics Primitive from the scene graph associated with this context.

context	the WebViewer context (from `createContext`)

Procedural-based Server-side API

- **indexGPrim**

```
status = wv_indexGPrim(wvContext *context, char *name)
status = iv_indexGPrim(I*8        context, C**   name)
```

Finds the index given the name for an existing Graphics Primitive in scene graph associated with this context.

context	the WebViewer context (from `createContext`)
name	the name of the GPrim in the scene graph
status	the index (where a negative value is an error)

- **printGPrim**

```
     wv_printGPrim(wvContext *context, int index)
call iv_printGPrim(I*8        context, I*4 index)
```

Prints the Graphics Primitive to standard output.

context	the WebViewer context (from `createContext`)
index	the index created for the primitive (from `addGPrim`)

Procedural-based Server-side API

Call-back Required to *catch* Client Messages

- **browserMessage**

```
browserMessage(struct libwebsocket *wsi, char *text, int len)
subroutine browserMessage(I*8 wsi, C** text)
```

This required routine gets called for each message sent from a client.

wsi	the WebSocket Interface Structure
text	the ASCII text received from the Browser
len	the length of the text

Procedural-based Server-side API

Text based communication to the Client(s)

- **broadcastText**

```
     wv_broadcastText(char *text)
call iv_broadcastText(C**   text)
```

Sends the text to all active clients (Browsers).

text	the text to send

- **sendText**

```
     wv_sendText(struct libwebsocket *wsi, char *text)
call iv_sendText(I*8                  wsi, C**   text)
```

Sends the text to the specific client designated by wsi.

wsi	the WebSocket Interface Structure (from `browserMessage`)
text	the text to send

Procedural-based Server-side API

FORTRAN Only Utility Functions

- **setPsize**

```
call iv_setPisze(I*8 context, I*4 index, R*4 size)
```

Sets the Point Size in an existing Graphics Primitive in scene graph associated with this context.

context	the WebViewer context (from `createContext`)
index	the index created for the primitive (from `addGPrim`)
size	the point size in pixels

- **setLwidth**

```
call iv_setLwidth(I*8 context, I*4 index, R*4 width)
```

Sets the Line Width in an existing Graphics Primitive.

context	the WebViewer context (from `createContext`)
index	the index created for the primitive (from `addGPrim`)
width	the line width in pixels

Procedural-based Server-side API

- **usleep**

```
call iv_usleep(I*4 micsec)
```

Suspends the calling thread for the specified number of microseconds

micsec the number of microseconds