C++ GUI Syntax for Document Object Models

By Anthony Matarazzo

A document object model providing rapid GUI development for the natively compiled C+ + 17 and above language. The produced executables can be used on personal computer microprocessors and embedded devices. This document describes the interface API language and the communication interface to the C++ language and standard library.

History of the Desktop GUI

In this paper and subsequent source code I present a method and model for achieving a high performance completely binary interface for a document object model. The result of this research is a combination of over twenty years of experience using different GUI frameworks, c and c++.

From the invention of the personal computer in the early eighties, the main forefront winner from the series of computers was the IBM personal computer. The first type of operating systems were command line based ones. Unix, CPM, PC/DOS and MS DOS provided the capability for many small businesses in the industry to perform data processing. However many found their interface difficult to use.

Through the progression of science, forms of graphic desktop operating systems provided users with the ability to use them easily. During this era a new device was invented called the mouse also known as a HID (human interface device). The first mouse operated using the rs232 serial port as its communication mechanism.

In the development of the operating system desktops that use graphical front ends, many of them employ "under the hood" the use of a message queue. In this, a routine consumes each of the messages while the application program intercepts conditionally a particular message. Most HID devices issue what is known an interrupt which executes a routine pointed to by the IVT (interrupt vector table). For example the press of a key, the click of the mouse button or a painting region request are transposed into a unique message identifier to which is dispatched to the message handling routine for the application window message consumer. These concepts are used as the basis for many operating system desktops such as X11, Microsoft Windows, and the Apple Macintosh.

Yet as the progressive complexity of the user interface expanded, the programming concepts and requirements gave developers daunting codebases to manage. Microsoft and Apple Macintosh took the lead with their component technology. ActiveX also known as OCX controls provide a discoverable interface and react to internally event messages and attribute modifications. In this, they each have their very own message handler. These components provide well tested user interface technology users are accustomed to. CUA or common user acceptance is the concept name this is known as. Most importantly, this gives the developer more time to focus on business logic.

In the late nineties, the web browser made an appearance as a means of document viewing with a few nice interface design events. The format of HTML gave writers and engineers an easy method of describing text content and forms to be submitted to a web server. The transmission of this data typically uses HTTP over the tcp/ip protocol. Browser programs due to their network nature had to be run within a sandbox so to speak to reduce the transfer of the computer virus. Their access to local computer resources is limited. The market progression also has brought more limitations such as allowable JavaScript AJAX communication.

Much of the development in the industry shifted to browser applications. Therefore most focuses failed in delivering truly inventive native desktop technology since most developers became web technologist. Meaning the application architecture for native desktop processing remains a verbose and steep curve. This leaves a tactical hole within the desktop market.

The architecture and flexibility of the W3C's web browser document object model design, the capabilities in layout, ease of implementation and presentation quality far exceed the capabilities according to coding complexity of a typical C++ Windows or Mac native desktop program.

After studying all of these technologies that transpired for over twenty five years, I was overwhelmed with the fact that modern GUI desktop designs are still using the antiquated message mode instead of a document approach. With most intellectual property and UI designs being advanced within the browser, some developers have sought to incorporate the web browser as a part of the display for applications. I was aghast at the size and performance of these applications. As well, for laptops these applications can reduce battery life significantly. Meaning less portable work can be performed. Experience estimates the time to about a third compared to a native application. I decided to rework all of these technologies into the C++ language syntax for value.

The C and C++ language has been around for numerous years. It is the main development language which any modern tool uses as its base to create even fourth generation languages. Javascript engines, the browser's internal language, is a c++ component. Typically with fourth generation languages, many of the documents such as HTML and subsequent CSS visual classification formats have to be parsed and interpreted. These technologies led to the invention of the JIT, which is the just in time compiler.

The JIT is a type of compiler that resides within the web browser or android phone that translates the fourth generation language’s logic and data layers to machine code.

LLVM is one of the most advanced compiler technologies in existence yet most JIT engines employ their very own mechanism. Oracle JRE is an example, Android uses a technology called ART, while Microsoft uses MSIL for dot net.

C++ is a high performance language that is compiled to machine code for a specific machine architecture. With the newer forms of the c++ syntax being dedicated to be easier to use, memory management, index searching, and array traversal are issues handled both in rapid description and at a better production ratio as far as execution time. In this, processor specific functionality such as Intel L1, L2 caches, branch prediction, function calling mechanisms, loop unrolling, inlining of small routines, address offset encoding sizes and other performance enhancing algorithms are managed more efficiently by a native compiler such as the LLVM BC format. These important low level attributes may be overlooked due to the JavaByte code or MSIL abstraction level and necessity of JIT performance. Some advanced implementations such as google chrome do perform well at optimizations but often lack native memory usage and data structures because of JavaScript and browser architecture. C++ provides a closer to hardware approach in its dialect and STD architecture.

Additionally, the C++ standard library, floating point expression evaluation, expression reduction and predicted linear memory management of structures and data types are faster due to the pure native execution and other optimising compiler algorithms. Recently with the additional registers in 64bit processors, less spill or memory swapping occurs. JIT technology may also overlook some of these machine code execution paradigms due to language features found in web technology.

In embedded technology the operating program is usually not sent over a network and hence compiled binaries are the execution stack. The c++ GUI syntax Template Dom provides this rapid development technology for a completely binary system. Perfect for ROM and native desktop development.

Examples of Various Platform GUI Frameworks

The following examples show the difference in coding style between native desktop applications. It is known that the more code that is written is reflective of the development and maintenance cost. The sample code below was copied from various free source repositories. The Hello World program is a give me program typically to show off syntax.

Windows Message Queue Program (Microsoft)

```
#include <windows.h>
#include <tchar.h>

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine,
                int iCmdShow) {

  static TCHAR szAppName[] = _T("HelloWin");
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wndclass;

  wndclass.cbSize = sizeof(wndclass);
  wndclass.style = CS_HREDRAW | CS_VREDRAW;
  wndclass.lpfnWndProc = WndProc;
  wndclass.cbClsExtra = 0;
  wndclass.cbWndExtra = 0;
  wndclass.hInstance = hInstance;
  wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
  wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
  wndclass.hbrBackground = (HBRUSH)GetStockObject(LTGRAY_BRUSH);
  wndclass.lpszMenuName = NULL;
  wndclass.lpszClassName = szAppName;
  wndclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
  RegisterClassEx(&wndclass);
  hwnd = CreateWindow(szAppName,           // window class name
                    _T("Hello World"),   // window caption
                    WS_OVERLAPPEDWINDOW, // window style
                    CW_USEDEFAULT,       // initial x position
                    CW_USEDEFAULT,       // initial y position
                    CW_USEDEFAULT,       // initial x size
                    CW_USEDEFAULT,       // initial y size
                    NULL,                // parent window handle
                    NULL,                // window menu handle
                    hInstance,           // program instance handle
                    NULL);               // creation parameters

  ShowWindow(hwnd, iCmdShow);
  UpdateWindow(hwnd);

  while (GetMessage(&msg, NULL, 0, 0)) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }
  return msg.wParam;
}

LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM lParam) {
  HDC hdc;
  PAINTSTRUCT ps;
  RECT rect;

  switch (iMsg) {
  case WM_PAINT:
    hdc = BeginPaint(hwnd, &ps);

    GetClientRect(hwnd, &rect);
    SetBkMode(hdc, TRANSPARENT);

    DrawText(hdc, _T("Hello World!"), -1, &rect,
             DT_SINGLELINE | DT_CENTER | DT_VCENTER);
    EndPaint(hwnd, &ps);
    return 0;

  case WM_DESTROY:
    PostQuitMessage(0);
    return 0;
  }

  return DefWindowProc(hwnd, iMsg, wParam, lParam);
}
```

AFX/MFC based Program (Microsoft)

```
// MyApp.h
// application class
class CMyApp : public CWinApp {
public:
  virtual BOOL InitInstance();
};

// frame window class
class CMyFrame : public CFrameWnd {
public:
  CMyFrame();

protected:
  // "afx_msg" indicates that the next two functions are part
  //  of the MFC library message dispatch system
  afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
  afx_msg void OnPaint();
  DECLARE_MESSAGE_MAP()
};
```

And here is the MyApp.cpp implementation file for the MYAPP application:

```
#include <afxwin.h> // MFC library header file declares base classes
#include "myapp.h"

CMyApp theApp; // the one and only CMyApp object

BOOL CMyApp::InitInstance() {
  m_pMainWnd = new CMyFrame();
  m_pMainWnd->ShowWindow(m_nCmdShow);

  m_pMainWnd->UpdateWindow();
  return TRUE;
}

BEGIN_MESSAGE_MAP(CMyFrame, CFrameWnd)
ON_WM_LBUTTONDOWN()
ON_WM_PAINT()
END_MESSAGE_MAP()

CMyFrame::CMyFrame() { Create(NULL, "MYAPP Application"); }

void CMyFrame::OnLButtonDown(UINT nFlags, CPoint point) {
  TRACE("Entering CMyFrame::OnLButtonDown - %lx, %d, %d\n", (long)nFlags,
        point.x, point.y);
}

void CMyFrame::OnPaint() {
  CPaintDC dc(this);
  dc.TextOut(0, 0, "Hello, world!");
}
```

X11 Based Program (Linux)

```
#include <X11/Xlib.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
  Display *d;
  Window w;
  XEvent e;
  const char *msg = "Hello, World!";
  int s;

  d = XOpenDisplay(NULL);
  if (d == NULL) {
    fprintf(stderr, "Cannot open display\n");
    exit(1);
  }

  s = DefaultScreen(d);
  w = XCreateSimpleWindow(d, RootWindow(d, s), 10, 10, 100, 100, 1,
                          BlackPixel(d, s), WhitePixel(d, s));
  XSelectInput(d, w, ExposureMask | KeyPressMask);
  XMapWindow(d, w);

  while (1) {
    XNextEvent(d, &e);
    if (e.type == Expose) {
      XFillRectangle(d, w, DefaultGC(d, s), 20, 20, 10, 10);
      XDrawString(d, w, DefaultGC(d, s), 10, 50, msg, strlen(msg));
    }
    if (e.type == KeyPress)
      break;
  }

  XCloseDisplay(d);
  return 0;
}
```

XCB Based Program (Linux)

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <xcb/xcb.h>

int main() {
  xcb_connection_t *c;
  xcb_screen_t *screen;
  xcb_drawable_t win;
  xcb_gcontext_t foreground;
  xcb_gcontext_t background;
  xcb_generic_event_t *e;
  uint32_t mask = 0;
  uint32_t values[2];

  char string[] = "Hello, XCB!";
  uint8_t string_len = strlen(string);

  xcb_rectangle_t rectangles[] = {
     {40, 40, 20, 20},
  };

  c = xcb_connect(NULL, NULL);

  /* get the first screen */
  screen = xcb_setup_roots_iterator(xcb_get_setup(c)).data;

  /* root window */
  win = screen->root;

  /* create black (foreground) graphic context */
  foreground = xcb_generate_id(c);
  mask = XCB_GC_FOREGROUND | XCB_GC_GRAPHICS_EXPOSURES;
  values[0] = screen->black_pixel;
  values[1] = 0;
  xcb_create_gc(c, foreground, win, mask, values);

  /* create white (background) graphic context */
  background = xcb_generate_id(c);
  mask = XCB_GC_BACKGROUND | XCB_GC_GRAPHICS_EXPOSURES;
  values[0] = screen->white_pixel;
  values[1] = 0;
  xcb_create_gc(c, background, win, mask, values);

  /* create the window */
  win = xcb_generate_id(c);
  mask = XCB_CW_BACK_PIXEL | XCB_CW_EVENT_MASK;
  values[0] = screen->white_pixel;
  values[1] = XCB_EVENT_MASK_EXPOSURE | XCB_EVENT_MASK_KEY_PRESS;
  xcb_create_window(c,                             /* connection    */
                    XCB_COPY_FROM_PARENT,          /* depth         */
                    win,                           /* window Id     */
                    screen->root,                  /* parent window */
                    0, 0,                          /* x, y          */
                    150, 150,                      /* width, height */
                    10,                            /* border_width  */
                    XCB_WINDOW_CLASS_INPUT_OUTPUT, /* class         */
                    screen->root_visual,           /* visual        */
                    mask, values);                 /* masks         */

  /* map the window on the screen */
  xcb_map_window(c, win);
  xcb_flush(c);

  while ((e = xcb_wait_for_event(c))) {
    switch (e->response_type & ~0x80) {
    case XCB_EXPOSE:
      xcb_poly_rectangle(c, win, foreground, 1, rectangles);
      xcb_image_text_8(c, string_len, win, background, 20, 20, string);
      xcb_flush(c);
      break;
    case XCB_KEY_PRESS:
      goto endloop;
    }
    free(e);
  }
endloop:

  return 0;
}
```

GTK+ (Microsoft, Linux and Mac OS X)

```
#include <gtk/gtk.h>

void hello(void) { g_print("Hello World\n"); }

void destroy(void) { gtk_main_quit(); }

int main(int argc, char *argv[]) {
  GtkWidget *window;
  GtkWidget *button;

  gtk_init(&argc, &argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
  gtk_signal_connect(GTK_OBJECT(window), "destroy", GTK_SIGNAL_FUNC(destroy),
                     NULL);
  gtk_container_border_width(GTK_CONTAINER(window), 10);

  button = gtk_button_new_with_label("Hello World");

  gtk_signal_connect(GTK_OBJECT(button), "clicked", GTK_SIGNAL_FUNC(hello),
                     NULL);
  gtk_signal_connect_object(GTK_OBJECT(button), "clicked",
                            GTK_SIGNAL_FUNC(gtk_widget_destroy),
                            GTK_OBJECT(window));
  gtk_container_add(GTK_CONTAINER(window), button);
  gtk_widget_show(button);

  gtk_widget_show(window);

  gtk_main();

  return 0;
}
```

Wxwidgets (Windows, macOS, Linux)

```
// wxWidgets "Hello world" Program
// For compilers that support precompilation, includes "wx/wx.h".
#include <wx/wxprec.h>
#ifndef WX_PRECOMP
#include <wx/wx.h>
#endif
class MyApp : public wxApp {
public:
  virtual bool OnInit();
};
class MyFrame : public wxFrame {
public:
  MyFrame(const wxString &title, const wxPoint &pos, const wxSize &size);

private:
  void OnHello(wxCommandEvent &event);
  void OnExit(wxCommandEvent &event);
  void OnAbout(wxCommandEvent &event);
  wxDECLARE_EVENT_TABLE();
};
enum { ID_Hello = 1 };
wxBEGIN_EVENT_TABLE(MyFrame, wxFrame) EVT_MENU(ID_Hello, MyFrame::OnHello)
    EVT_MENU(wxID_EXIT, MyFrame::OnExit) EVT_MENU(wxID_ABOUT, MyFrame::OnAbout)
        wxEND_EVENT_TABLE() wxIMPLEMENT_APP(MyApp);
bool MyApp::OnInit() {
  MyFrame *frame =
      new MyFrame("Hello World", wxPoint(50, 50), wxSize(450, 340));
  frame->Show(true);
  return true;
}
MyFrame::MyFrame(const wxString &title, const wxPoint &pos, const wxSize &size)
    : wxFrame(NULL, wxID_ANY, title, pos, size) {
  wxMenu *menuFile = new wxMenu;
  menuFile->Append(ID_Hello, "&Hello...\tCtrl-H",
                   "Help string shown in status bar for this menu item");
  menuFile->AppendSeparator();
  menuFile->Append(wxID_EXIT);
  wxMenu *menuHelp = new wxMenu;
  menuHelp->Append(wxID_ABOUT);
  wxMenuBar *menuBar = new wxMenuBar;
  menuBar->Append(menuFile, "&File");
  menuBar->Append(menuHelp, "&Help");
  SetMenuBar(menuBar);
  CreateStatusBar();
  SetStatusText("Welcome to wxWidgets!");
}
void MyFrame::OnExit(wxCommandEvent &event) { Close(true); }
void MyFrame::OnAbout(wxCommandEvent &event) {
  wxMessageBox("This is a wxWidgets' Hello world sample", "About Hello World",
               wxOK | wxICON_INFORMATION);
}
void MyFrame::OnHello(wxCommandEvent &event) {
  wxLogMessage("Hello world from wxWidgets!");
}
```

Chromium Embedded Framework

(Microsoft, macOS, Linux implementation source varies between)
The entire browser is distributed with the application. The application supports the robust W3C document object model but there are several prices to pay for communication to the renderer and the chrome browser. That is, a steeper learning curve and specific data types intrinsic are required. This is primarily because of the architecture requirements of CEF and JavaScript JIT.

```
#include <string>
#include <windows.h>

#include "include/cef_app.h"
#include "include/cef_base.h"
#include "include/cef_browser.h"
#include "include/cef_client.h"
#include "include/cef_command_line.h"
#include "include/cef_frame.h"
#include "include/cef_runnable.h"
#include "include/cef_web_plugin.h"
#include "include/cef_web_urlrequest.h"

#include "ClientHandler.h"

ClientHandler *g_handler = 0;

std::string GetApplicationDir() {
  HMODULE hModule = GetModuleHandleW(NULL);
  WCHAR wpath[MAX_PATH];

  GetModuleFileNameW(hModule, wpath, MAX_PATH);
  std::wstring wide(wpath);

  std::string path = CefString(wide);
  path = path.substr(0, path.find_last_of("\\/"));
  return path;
}

LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam,
                            LPARAM lParam) {
  switch (uMsg) {
  case WM_DESTROY:
    PostQuitMessage(0);
    return 0;

  case WM_SIZE:
    if (g_handler) {
      // Resize the browser window and address bar to match the new frame
      // window size
      RECT rect;
      GetClientRect(hwnd, &rect);

      HDWP hdwp = BeginDeferWindowPos(1);
      hdwp = DeferWindowPos(hdwp, g_handler->GetBrowserHwnd(), NULL, rect.left,
                            rect.top, rect.right - rect.left,
                            rect.bottom - rect.top, SWP_NOZORDER);
      EndDeferWindowPos(hdwp);
    }
    break;

  case WM_ERASEBKGND:
    if (g_handler) {
      // Dont erase the background if the browser window has been loaded
      // (this avoids flashing)
      return 0;
    }
    break;

  case WM_CLOSE:
    if (g_handler) {
      CefRefPtr<CefBrowser> browser = g_handler->GetBrowser();
      if (browser.get()) {
        // Let the browser window know we are about to destroy it.
        browser->ParentWindowWillClose();
      }
    }
    break;

  case WM_PAINT:
    PAINTSTRUCT ps;
    HDC hdc = BeginPaint(hwnd, &ps);
    EndPaint(hwnd, &ps);
    return 0;
  }
  return DefWindowProc(hwnd, uMsg, wParam, lParam);
}

HWND RegisterWindow(HINSTANCE hInstance, int nCmdShow) {
  const wchar_t CLASS_NAME[] = L"CEFSimpleSample";

  WNDCLASS wc = {};

  wc.lpfnWndProc = WindowProc;
  wc.hInstance = hInstance;
  wc.lpszClassName = CLASS_NAME;

  RegisterClass(&wc);

  HWND hwnd =
      CreateWindowEx(0,                    // Optional window styles.
                     CLASS_NAME,           // Window class
                     L"CEF Simple Sample", // Window text
                     WS_OVERLAPPEDWINDOW,  // Window style

                     // Size and position
                     CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,

                     NULL,      // Parent window
                     NULL,      // Menu
                     hInstance, // Instance handle
                     NULL       // Additional application data
                     );

  if (hwnd == NULL)
    return 0;

  ShowWindow(hwnd, nCmdShow);

  return hwnd;
}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE, LPSTR, int nCmdShow) {
  // Register the window class.
  HWND hwnd = RegisterWindow(hInstance, nCmdShow);
  if (hwnd == 0)
    return 0;

  RECT rect;
  GetClientRect(hwnd, &rect);

  CefSettings appSettings;
  CefBrowserSettings browserSettings;
  CefRefPtr<CefApp> cefApplication;
  CefRefPtr<CefClient> client(new ClientHandler());
  g_handler = (ClientHandler *)client.get();

  CefWindowInfo info;
  info.SetAsChild(hwnd, rect);

  CefInitialize(appSettings, cefApplication);

  std::string path = GetApplicationDir();
  path = "file://" + path + "/html/index.html";

  CefBrowser::CreateBrowser(info, client, path, browserSettings);

  CefRunMessageLoop();

  CefShutdown();

  return 0;
}
```

Qt Framework

Windows, macOS, Linux

By far the most easiest and comprehensive solution for cross platform development that has support. However, it is not document API based as the template DOM is. This decreases coding flexibility, readability, and ease of maintenance of an application’s source code. Typically, the standard library data types and common names such as “pushButton” and “Dialog” are modified slightly with the QT namespace. That is adding the Q or other prefixes in front of a nice readable name.

```
#include <qapplication.h>
#include <qpushbutton.h>
int main(int argc, char **argv) {
  QApplication a(argc, argv);

  QPushButton hello("Hello world!", 0);
  hello.resize(100, 30);

  a.setMainWidget(&hello);
  hello.show();
  return a.exec();
}
```

C++ 17 Template DOM Program

This is the new format in research for C++ 17 and above. The product is designed to be cross platform. Currently the macOS, iOS, and Android NDK library needs research. The template dom library works precisely with the standard library without adding new data types. It supports rapid development and incorporates a small set of memorable names within its model. it is a templated oriented coded implementation for higher performance than any of the preceding syntaxes. It offers the document object model integrated with C++ language as a natural syntax that appears like HTML because of the use of <>.

The product is designed to be a cross platform product line. The following development platforms and technologies are capable of use for development and distribution.

- Microsoft Windows
- Linux
- macOS
- iOS
- Android NDK (native development kit)

Supported C++ compilers

- Microsoft Visual Studio
- GCC
- clang
- Objective C

Rendering Engines

There are numerous possibilities for the model's occupancy within the developer's toolbox. The system provides these capabilities through well documented preprocessor definitions. That is, compiling for each of these types of environments is established as simply turning a definition on or off. Some of these project types require third party libraries for linking. Optionally linking can be static or runtime dynamic.

- Embedded internal
- Embedded external
- Embedded advanced image processing external
- Cross platform Desktop application model
- Cross platform application model with advanced image processing
- Qt client cross platform Dom
- GTK++ client cross platform Dom

- WxWidgets client cross platform Dom
- Chrome Embedded Framework
- Gecko Layout Engine
- Windows OS specific
- XCB Linux Application
- X11 Linux Application
- X11 Linux Desktop
- Android OS specific

```
#include "viewManager.hpp"

using namespace std;
using namespace ViewManager;

#if defined(__linux__)
int main(int argc, char **argv) {
// handle command line here...

#elif defined(_WIN64)
int WINAPI WinMain(HINSTANCE /* hInstance */, HINSTANCE /* hPrevInstance */,
                LPSTR lpCmdLine, int /* nCmdShow */) {
// command line
#endif -

  auto &vm = createElement<Viewer>(
      objectTop{10_pct}, objectLeft{10_pct}, objectHeight{80_pct},
      objectWidth{80_pct}, textFace{"arial"}, textSize{16_pt}, textWeight{400},
      textIndent{2_em}, lineHeight::normal, textAlignment::left,
      position::relative, paddingTop{5_pt}, paddingLeft{5_pt},
      paddingBottom{5_pt}, paddingRight{5_pt}, marginTop{5_pt},
      marginLeft{5_pt}, marginBottom{5_pt}, marginRight{5_pt});

  vm << "Hello World\n";
  vm.render();
}
```

Application Interface Design

The internal design of the system provides the extensibility to support many types of implementations. Primarily the difference between implementations is the application that the code base will be used for. For embedded applications, without an external dll dependency, all code for controls and GUI box model layout calculations will exist within the underlying binary executable. This does provide for at times the smallest distribution and is exactly what is needed for a turn key embedded system that is ROM based. In an embedded system of this type, there are not any development tools or distributed applications that a system user runs. The device is a whole unit and is programmed with all of the layout facilities that is needed for any network transfers to be rendered. Or the internal programs rely on the main control program for rendering their selected layout portion. In short, it is a closed system that is hardwired yet still can be versatile.

Another type of a more expandible implementation provides the capability of an external object code base that is reusable and contains the rendering code that is dynamic. The possibility of reading multiple image formats and such should be contained within this or other independent object code (DLLS). Additional functions such as image processing, compression utilities, animation bindings, algebraic interpolation, and image aesthetic additives like shadowing may be optional features to include.

Other types of designs are not as holistic yet provide features available in the document object model with less development perhaps. In their existence, these models are not as efficient but can be considered more tested. For example, development of the model to pass information to the CEF model. Since the CEF model is document based, the system will drastically lessen the learning curve to use the CEF in applications. As well, the capability of the model is matured in that many additional rendering attributes can be quickly attained. One drawback is that the memory requirements are much greater. There will be no battery power savings on mobile devices. The data templated architecture does not benefit as data has to be shipped to and from the model to the CEF because it maintains its own dom.

The base Windows architecture and control library can be instantiated using the model. The windows configuration does not directly support a dom but has specific font layout technology that can be used. The activex controls can be used as a wired type of translation. Meaning that list boxes, editing controls, and date picking facilities will duplicate information storage as a layer. This configuration can be very useful as the syntax of the template Dom is easy. Yet the argument of a cross platform implementation is not won. The sustainability of the template Dom as a cross platform product is lost due to the focus of the windows platform and common controls api interface. However, the applications will be high performance, lightweight, and very easy to develop.

Using the QT framework can provide the best of the branch in that syntax and cross platform is achieved. Yet as QT is not implemented as a document based system, the underlying architecture is simply a new cover on top of QT. Data will be duplicate within STD vectors and in the control.

Any modern GUI programmer would argue this is always the case. Data copy and storage format translation is a common practice found when using component technology.

The availability of progress is always enticing for entrepreneurs. Reinventing the wheel is holistic for the greatest benefits to be achieved with the template dom. In this light, many design flaws found in older technology can be overcome by using smarter methodologies in engineering and applauding the inclusion of more sophisticated rendering algorithms for some types of visualizations.

The design of a common user interface library to be incorporated within the library is necessary. The library must directly use the c++ STD namespace to consolidate principles of distribution. For example, editing controls that are international. Controls that work directly in place with base data types like vector, string, int, float, double, long double are beneficial to performance and c++ language interoperability. The provision for ease of data format description and entry verification is a plain attribute. These are important aspects for which the browser and application designers have evolved significantly through progressive implementation. Lastly, Ray tracing of components for new age appeal can summon market demand as a likeable trait.

The coding of the most beneficial branch, a recoded control library, has many great advantages. To invest in the research of usability can be a large cost. However inclusion of the best of the best practices is the most intelligent approach in GUI usability requirements. This requires intellectual agreements to be demonstrated for reimplemented technologies perhaps.

The respected design within the c++ codebase is encapsulated in the viewManager namespace declaration. It persists with the following tree:

- Visualizer (namespace)
    - platform (c++ class with platform specific codebase)
- Element (the main C++ class that all document elements derive from)

Consolidated Parameters that Self Document

The API calling mechanism is tailored to simplify and shrink the language necessary to a rapid yet descriptive syntax. This reduces errors but at the same time increases readability for developers. Below are several parameters and how they are coded within the c++ source code using a consolidated form.

Numeric Formats

The system includes user defined literals, a distinct c++ 11 and greater feature, for ease of labeling numeric expressions. This provides an easy to read and self documenting syntax. Terms expressed as numerics can be written with suffixes that allow the developer to deduce the measurement mode. User defined literals are implemented in the c++ language as a function with a specific prototype. These small routines return the necessary objects as parameters. Usually to describe the numericFormat object, it is necessary to place the type within the second parameter and include a formal class parameter instantiation.

Numerical values with a format specifier can have the following format specification.
using numericFormat = enum option { px, pt, em, percent, autoCalculate };

Example of long form.

```
textSize{20, numericFormat:pt}
```

- pt
- em
- px
- percent
- pct
- normal
- numeric

However by using the user defined literals, values can be described in less space with even better readability. User defined literals are invoked by placing a suffix and be preceded by an underscore '_'. For example:

```
auto &vm = createElement<Viewer>(
    objectTop{10_pct}, objectLeft{10_pct}, objectHeight{80_pct},
    objectWidth{80_pct}, textFace{"arial"}, textSize{16_pt}, textWeight{400},
    textIndent{2_em});
```

Enumerated Options

For ease of use, parameters accepted as an enumerated value can be described by only using the option value within the function parameters. The API internally processes the values to be objects. In this capacity, the following attributes apply:

- display
- position
- textAlignment
- borderStyle
- listStyleType
- dialogStyle

Examples of Short Enumerated attributes

```
auto &div = createElement(indexBy{“divList”}, lineHeight::normal, textAlignment::center,
                position::relative, borderStyle::dotted);
```

Color Parameters

Color needs an expressive form as well. The object provides a binary interface for each of the color formats and provides RGB as a default. The color object provides multiple formats, yet selection of the input format is a description within the first parameter. For example, color can be described in one of the following formats ways.

```
NAME(const double &_v1, const double &_v2, const double &_v3)
    : colorNF(colorFormat::rgb, {_v1, _v2, _v3, 0}) {}
NAME(const std::string_view &_colorName) : colorNF(_colorName) {}
NAME(const colorFormat &_opt, const std::array<double, 4> &_val)
    : colorNF(_opt, _val) {}
NAME(const colorFormat _opt, const double &_v1, const double &_v2,
     const double &_v3)
    : colorNF(_opt, {_v1, _v2, _v3, 0}) {}
NAME(const colorFormat &_opt, const double &_v1, const double &_v2,
     const double &_v3, const double &_v4)
    : colorNF(_opt, {_v1, _v2, _v3, _v4}) {}
NAME(NAME &&) = default;

using colorFormat = enum colorFormat { rgb, hsl, name };
```

Within the base model, the following color objects are supported.

- background
- textColor
- borderColor

Examples Of Various Color Expressions

```
background{64,64,64}   // an RGB color description
background{“blue”}  // a string based color name
background{colorFormat::hsl, [64,64,64,1]}  // format and an array of values that also
provide input of alpha value.
background{colorFormat::hsl, 128, 128, 128}
```

Data as Parameter

At times, it is convenient to combine all of the parameters, including the data, into one function call. This shortens the necessary lines of code. As well, it may provide easier to read syntax. However, only certain types of data are supported. The actual position of these parameters is within what is expected to be an attribute. The system scans an internal table to match any of the attributes to the specific type. The following types are automatically supported:

- char
- double
- float
- int
- stringview
- string

- vector<char>
- vector<double>
- vector<float>
- vector<int>
- vector<stringview>
- vector<string>

- vector<vector<stringview>>
- vector<vector<string>>
- vector<tuple<int,stringview>>
- vector<tuple<int,string>>

```
getElement("mainArea")
        .appendChild<PARAGRAPH>(
        indexBy{"bodyText"}, textColor{"blue"},
        "The information here is added to the document. Text is "
        "wrapped while other items remain. It is hoped that image and "
        "image "
        "processing will be fun. I think so. After I had learned "
        "that "
        "the ImageMagick library had been tuned, works with SSE "
        "in 64bit floating point format, I thought to myself, "
        "should be fast. ");
```

This example retrieves a document element indexed as mainArea and appends a new child paragraph to it with a blue color. Notice that the paragraph data is inlined within the function call. Otherwise, many more statements would be necessary. Internally the attribute routine deduces that the data type is not an official attribute and is of a string type.

```
info.appendChild<ul>(
            indexBy{"idbikes"},
            vector<string_view>{"Huffy", "Schwinn", "Giant", "Road Master"});
```

The example above provides an appending of a child that is an unordered list. Within the function call, the index is set as well as the contents of the list.

```
e.appendChild<UL>(indexBy{"bookletNotes"},
            vector<string_view>{"Endurance training", "Biking", "Meals",
                        "Schedule"})
            .append<UL>(indexBy{"guestCompanies"},
            vector<pair<int, string_view>>{{0, "Gyms"},
                                    {1, "Silvers Gym"},
                                    {1, "Core Fitness"},
                                    {1, "Tommy's Toe nail"},
                                    {0, "Library Lifters"},
                                    {1, "Book Bench Press"},
                                    {1, "Keyboard Curls"},
                                    {1, "Muscle XT Power lifting"}})
            .append<UL>(indexBy{"bookletReferences"},
            vector<string_view>{"The 29inch Road", "Flatters and Hacking Chain",
                       "Wheelers and Handle Bars",
                       "Rim's n Chains", "Smelly Mice Vice"});
```

The preceding example shows how complexity can be clumped up.

The focus of the various forms of parameter compaction shows how scripting like syntax can be achieved within the modern c++ language. This is one focus of the design. In this, the c++ language features are shown to be supportive of the newest forms of document object building likened to a scripting style language, but with all of the compiled binary speed needed for cheap rom based computers or kiosks. In summary, compacting and bringing readability to a syntax gives a very appealing and simple form for expressing graphical user interfaces in C++. In this, distribution of the client brings a distinct new GUI capability to the many disciplines of programmers.

Parsed String DOM Building

At times, the explicit nature of a complete binary interface is more verbose for a large number of elements. The developer may feel that building a string of markup combined with the element's textNode in between the markup symbols is suitable. This functionality is similar to HTML browser functionality in that each time a string is encountered, it is parsed and the items are added to the referenced element. At times, this function may be useful for prototyping interfaces quickly. The following functions support a text based parsed input.

- createElement
- appendChild
- append
- insertBefore
- insertAfter
- createStyle

Examples Of Their Use In C++ Code

```
void test7c(Viewer &vm) {

  vm.appendChild("<ul id=idColors><li>Green</li><li>Blue</li><li>White</li><li>Orange</li></ul>");
 auto &oColors=getElement<UL>("idColors");
  vm.render();
}
```

The example above creates a child linked to the main viewer object passed to the function. The unordered list is created with children that name colors.

Templated Codebase

- Multiple verbose interfaces for reduced binary code and explicit generation
- Rvalue static reduction for string literals
- Comdat reducing compiler options are used to further reduce the binary image. STD containers and algorithms operate more efficiently.

Overuse of a templated codebase may cause larger executables to be produced. This can be shown in the production of the example research code which compiles to about 65k. This is the interface code only. Perhaps a subtle argument that can be found in more research, the modification of the interface can be made in a few entry points to alleviate excessive binary code production. The document API methods of creation (createElement, appendChild, and append) may be modified to restrict using the template parameter in favor of a static based numerical enumerical in the final design. This is an evaluation study for the technical portion of review.

Platform API Inline

A platform specific implementation is achieved using preprocessor #define conditional compilation in conjunction with the abstract platform base class. All operating system specifics are within this platform c++ class. This design provides explicit instantiation for low level implementation that achieve a high degree of performance. Most rendering capabilities use off screen video buffers. Perhaps further development exercises can use open gl directly for compositing of surfaces. However, an in lined software algorithm is the default for 24bit color mixing.

Font rendering may use freetype API for ttf vector and bitmap typefaces. Another option is using platform API such as Microsoft's implementation. Additionally imageMagick++ provides a robust solution but uses freetype internally. The system provides optional selection of these code base traits production oriented distribution.

The communication layer is light between the operating kernel and template dom. The message queue is represented directly with objects. However with a refined model even a tighter implementation can be achieved. Meaning that the gap between the interrupt vector table and event objects is lessened. The most effective kernel for use in this area is the Linux kernel. X11, xserver, and also xcb are the applicable development technologies. Also, udev and other base Linux tools can be used which is what X11 uses internally.

If the need arises, the platform system object can also be used for higher level implementations such as the QT framework, CEF, or Microsoft specific implementation. Achieving a durable architecture is desirable for longevity of the API. That is, it is an application framework for software designers to implement business logic. Yet the rendering portion should remain a black box. This provides the most effective to market idealism while providing the future of hardware supported dom. The system should be designed and implemented for backwards compatibility.

Data Interface

- data<>
- dataTransform<>
- dataHint

The data<> templated function provides a standardized way for data injection and document building to centralize communication for multiple types of raw information or structures that require visualization. At most times, the default operation for the implementation details will be provided as a devoted comprehension of a document element's purpose. e.g. UL elements are composed of LI elements. The system also provides default formatting for string vectors and other common data formats. The data interface is comparable to XML as a means of input but uses c++ initializer lists. It is more flexible because it can accept developer designed structures and classes.

The process of how visualizations and tree building occur is left to the developer. This can happen as a defaulted formatter or by using a specific method. When super structures are contained within the STD:vector, an element tree building lambda can be provided. This lambda function is invoked only when necessary by the visualization system. Classes can be used as a capable instantiation of document tree building as well. Classes do offer a more encapsulated approach that can be better planned within source code organization. There is also a dataTransform<> templated class that provides more advanced capabilities. This is discussed in more detail later.

Data placed into the element is owned by the object’s internal storage but is accessible in place anytime. Each element has a data member that is empty when it is created. This is a desired design for these types of objects as visual presenters as information exists in one place and not duplicated in different parts of the program. This data within the vector is available to add, update or delete from. Doing so directly affects the visualization directly. These aspects of data management occur from the use of the standard c++ STD vector implementation. A reference to the std::vector<T> & can be attained by gathering the return of the data function which allows management in c++ source.

The design allows definition of data structures rather than reliance on parsed structure definition as HTML must have. The W3C technology h1, h2, h3, div, and other elements are contained within the C++ program as objects. If a UL list has one million entities within it, HTML verses STD::vector, a major difference in performance between the two architectures can be readily shown. In HTML form this would be a large data set, yet within the vector C++ design, a few megabytes is a manageable block of memory according to modern microprocessors.

The data<> vector member is also affected by the stream operator. In C++ this is noted as << within the source code. Internally, the input mechanism stores the content within the standard vector<string> in sequential order as each stream operation is processed. Each call to the stream operator is a separate entry into the data vector. Typically, it is wise to use one type of operation for the logical filling of the data<> buffer to offer less confusion within the code base. Usually the stream operator is used for paragraph and textual information that uses the default lambda for display. Most likely the stream operator will be the default way to input information of a textual nature that is read from a buffer or network.

The rendering technology interface uses a hinted signaller to allow the developer to inform the visualization system of any modifications. This exposes the versatility of the design to be very elegant in that fewer errors are made by the developer compared to other types of c++ GUI frameworks. Typical errors of the past have been the flickering of elements on the screen as they are being redrawn. Or another common issue would be that an update does not occur until the user performs another operation which causes a redraw. There are many applications that have these types of problems.

The dataHint function provides informative visual updating functionality in one compact efficient API call. Simply as the developer updates, extends, deletes, or insert items into the vector, the dataHint api should be called with information about where in index terms the vector has changed. For performance, the call can be made once per several updates or on a change by change basis. The system maintains an intelligent historical use to help quickly deduce where changes might have occurred. This method provides fewer errors during development as the API definition and usage is very straightforward.

In summary, there are three functions to use for interface document building that provide the versatility and design of XML/XSL transformations. The data<>, dataTransform<>, and dataHint API provide these capabilities. The data<> function holds the information in a user defined vector internally within the element object. For flexibility, string is the default and need not to be named. The data<> object always expects a vector of data which provides easy to manage syntax. The dataTransform<> provides user implemented reflection of this data on the screen. The dataHint function provides the system with information about elements that have been modified or updated that will need GUI repaint. The dataHint provides a method of quicker realization to repaint yet the visualization system will eventually recognize the modification.

These items may seem difficult to use, however once a C++ developer takes a look at the source for implementing these functions, they can be understood quite easily. The test below shows some of the continuation syntax, stream operations, and smart UL list building. Notice that the data function is never submitted formally within the API. The parameters of the appendChild and append functions filter these types of array items and place them within the data vector internally. Typically this only works with specific types of arrays. Arrays of string, and numerically tagged data.

```
void test6(Viewer &vm) {
  testStart(__func__);

  ElementList chapter;
  int m = randomInt(5);

  for (int i = 0; i < m; i++) {

    auto e = createElement<PARAGRAPH>(
        indexBy{"rndTEST5BookletParagraph_" + to_string(i)});

    stringstream ss;

    ss << "Hello "
       << "anthony"
       << "can you do the []";

    e << ss;

    e.appendChild<UL>(indexBy{"bookletNotes_" + to_string(i)},
                  vector<string_view>{"Endurance training", "Biking",
                                      "Meals", "Schedule"})
        .append<UL>(
            indexBy{"guestCompanies_" + to_string(i)},
            vector<pair<int, string_view>>{{0, "Gyms"},
                                          {1, "Gold's Gym"},
                                          {1, "Core Fitness"},
                                          {1, "Tommy Doright's"},
                                          {0, "Tools"},
                                          {1, "Huffy"},
                                          {1, "Scwitchers"},
                                          {1, "Clock Down Industrials"}})
        .append<UL>(indexBy{"bookletReferences_" + to_string(i)},
                 vector<string_view>{"The 26inch Road", "Flatters Chain",
                                     "Wheelers and Handlebars",
                                     "Rim's n Chains"});

    chapter.push_back(e);
  }

  vm.appendChild<DIV>(indexBy{"booklet5"}).appendChild(chapter);

  vm.render();
}
```

The test below shows several variations of the invocation of the data function. By default, the data template parameter is a string, as seen within the first one. Notice in the second one, only the base data type is provided. This is because all data is assumed to be stored within a vector. This simplifies the syntax as a storage mechanism.

```
//! [test7d]
void test7d(Viewer &vm) {
  testStart(__func__);

  auto &divTest = vm.appendChild<DIV>(indexBy{"testAnother"});

  // the append and appendChild with texts.
  divTest.appendChild("<ul></ul>").data() = {"2222", "3333", "444", "6.66",
                                             "7"};

  divTest.appendChild<ul>().data<double>() = {1, 4, 5, 3, 4, 4, 4, 33, 4, 5};

  divTest.appendChild<ul>().data<pair<int, string_view>>() = {
      {0, "AC/DC"},
      {1, "Hell's Bells"},
      {1, "You Shook Me All Night Long"},
      {1, "Have a Drink on Me"},
      {1, "Squeeler"},
      {0, "Books"},
      {1, "Logging for Tree Dwellers"},
      {1, "Tent Building with Trash"},
      {1, "Monopoly Crashing for Dummies"}};

  divTest.appendChild("<Combo></Combo>").data() = {
      "Orange", "Blue", "Purple", "Green", "Crimson"};

  divTest.append("<ul><li>Hello added to the end</li></ul>");
  divTest.appendChild<ul>().data() = {"San Ho Hui", " White Lotus", "Taiping",
        "Boxer Rebellions", "Heaven and Earth Society", "Buddah Bangers", "Tao Teasers",
        "Head Operators with Silk"};
}
```

The syntax below shows the TABLE element and how the columns and subsequent data rows can be clumped together in one function call. The types tableColumns and tableData are defined internally as a communication type to the TABLE element.

```
void test7e(Viewer &view) {
  auto &tblCost = createElement<TABLE>(
      objectLeft{10_pct}, objectTop{10_pct}, objectWidth{80_pct},
      objectHeight{80_pct}, tableColumns{{"Name", "Employment Start", "Salary",
                                         "Sales", "Cost Ratio"}},
      tableData{{"Anthony", "1/1/15", "10.75", "34.16", "4.5 +"},
               {"Candy", "4/16/12", "15.75", "4464.76", "35017.2 +"},
               {"Alvin", "1/1/65", "3.75", "125.16", "2634.9 +"}});

  view.render();
}
```

implementing complex visualization transforms

The dataTranform<> template class provides a method of establishing a relationship between a lambda function, a map of lambda functions, or a parsed formatted string to the STD:vector data. These methods of DOM building are the most advanced techniques to use in applications because it allows a more complex layout and summary of information. However, much of the complexity is obfuscated in less code by using a formatting string. The transformation process can be likened to XML - XSL processing except that only the c++ STD is utilized. The design provides a transform of data automatically to elements when data must be visualized on screen.

The c++ 'using' statement can be used to create an alias of a type - the artificial name used by a coder for their internal coding communication. This name is used as a template parameter and thus is used as a storage allocator within the vector container. When using the string transform method, the data must be stored as a tuple however. The STD::tuple templated class allows the members to be accessed by index expressions. These indexes are offsets into the allocated memory needed for formatting the data.

When using the templated data<> function, the template parameter given provides readable syntax for code review because of the named alias. This is one type of communication interface generation for specific binary data formats. The system includes several named alias for communication to specific elements. tableData and tableColumns are for use with the table object for example. C++ objects can be created for more complex structures and object oriented large scale business architectures as shown in the examples below.

The templated class dataTransform provides document building functionality with ease of implementation. The design separates the data from the document view building. There are four forms of the templated function.

```
template <typename R, typename T>
void dataTransform(const std::function<R &(T &)> &_fn) {}

template <typename R, typename T>
void dataTransform(const std::string_view &txtFn) {}

template <typename R, typename T>
void dataTransform(const std::string_view &txtFn,
                const std::function<bool(T &)> &_fn) {}

template <std::size_t I, typename T>
void dataTransform(
    const std::unordered_map<const typename std::tuple_element<I, T>::type &,
                        std::function<Element &(T &)>> &_transformList) {}
```

One form of the dataTransform<>() templated class accepts two template parameters: child element type and storage type. All items within the system are stored within a vector. The actual transform lambda is a std::function. The function can be expressed as a lambda or std::function compatible object. The function is expected as the formal parameter to the dataTransform<>() function. This function is called once for each item within the view. The sample function below shows how this is achieved.

```
void test7f(Viewer &view) {
  // build a fancy <ul> list with <li> children consisting
  // of varying details...

  static array<string_view, 3> sIconNames = {"base.raw", "contentIcon.raw",
                                             "normal.raw"};

  // dataAdaptor
  using tagInfo = tuple<int, string_view, float>;

#define idRecords "id_records"

  // format detailed views easily ...
  view.appendChild<ul>(indexBy{idRecords}).data<tagInfo>() = {
      {0, "AC/DC", 9.9},
      {1, "Hell's Bells", 9.8},
      {1, "You Shook Me All Night Long", 7.2},
      {1, "Have a Drink on Me", 4.3},
      {1, "Squeeler", 9.1},
      {0, "Books", 10},
      {1, "The Button Bubble - Digital Economics By Anthony Matarazzo", 5.5}};

  auto &urecords = getElement<ul>(idRecords);
  auto &recordPlex = urecords.data<tagInfo>();

  recordPlex.push_back({0, "Variodic Blackhearts", .2});
  recordPlex.push_back({1, "Sympathetic Voting Machines", .001});
  recordPlex.push_back(
      {1, "Vectorized Signaling Auto Typed constexpr Candy", .101});
  recordPlex.push_back({1, "Ya Got To Write It All screamin interfaces", .22});
  recordPlex.push_back(
      {1, "TemPlated Desktops with INLINED ANIMATION MOTION _data", .667});
  recordPlex.push_back({1,
                        "a neutral index of finger motion path skeletal data "
                        "stored in reusable gesture rom",
                        .665});
  recordPlex.push_back({0, "The Stairsouppe Choordettes", .4});
  recordPlex.push_back(
      {1, "My Binary Blister Coniquebine <ani=greenGlow>megatron remix</ani>",
       .19});

  // allowing for very fast expansion for meta creations. Most likely,
  // the tags may mean something more preverse than numeric. So the balance is
  // in definition

  // formatting lambda
  // passed  T & - type, class or tuple
  auto &fnTransform = [](tagInfo &tag) -> li & {
    textColor vip =
        std::get<2>(tag) < 5.0 ? textColor{"blue"} : textColor{"purple"};

    auto &o = createElement<li>(textColor{"blue"});

    // set icon of the li element
    auto &n = std::get<0>(tag);
    o.appendChild<image>(sIconNames[n]);

    // set text
    o.appendChild<span>(std::get<1>(tag));
    return o;
  };

  urecords.dataTransform<li, tagInfo>(fnTransform);
```

```
// Data insertion using the stdandard libary
//
// get reference to actual memory. std interface.
auto &ulRecords = getElement<ul>(idRecords);

// tagFormatter is needed for hash map to get the structure.
// templated for user defined storage and reflexion.
// state information is saved when getAdaptor is invoked,
// so sensing what changes according to screen rules is
// the problem to solve for the system that will create
// high performance "implementation error free" solution

// exception raised from non created item from absorb.
auto &recordPlex2 = ulRecords.data<tagInfo>();

for (int i = 1; i < 10; i++)
  recordPlex2.push_back({i, "STD::MOVE", randomDouble(0, 10)});

// senses that 10 added to the end of a list of 1000 already/ - no display
// change.
ulRecords.dataHint<tagInfo>(10);

for (auto &n : recordPlex2) {
  if (std::get<2>(n) > 5.5 && std::get<2>(n) < 6.9) {

    auto &sv = std::get<1>(n);
    sv = std::string(sv) + std::string(" Wowza");

    // takes info and induces the change.
    ulRecords.dataHint<tagInfo>(n);
  }
}
```

The test example below shows the format to be used with class initializers. An important aspect is that multiple versions of the class constructor can be maintained and therefore the initializer list can support multiple forms of initialization. Notice that the transform lambda simply invokes a function from the passed object. This leaves room for a good design to be implemented by the developer.

```
void test7g(Viewer &view) {
  // class based linkage for more complex formatters and
  // builders. This class and formatting logic would be in a separate file,
  // here for testing.
  using uniRecord = class uniRecord {
    enum uniRecordSchema {
      contact,
      filelist,
      pictureThumb,
      ratingVisual,
      videoThumb,
      audioThumb
    };

  public:
    size_t index;
    uniRecordSchema schema;
    uniRecord(const string_view &broadName) {}
    uniRecord(const string_view &fname, const string_view &lname,
              const string_view &phone, const string_view &email) {
      schema = contact;
    };
    uniRecord(const string_view &_fileName, const size_t &fileSizeKiloByte,
              const size_t &indexPreviewKey) {
      schema = filelist;
    };
    uniRecord(const string_view &fname, const string_view &format,
              const int &depth, const tuple<double, double, double> gps,
              const size_t summaryIndexKey) {
      schema = pictureThumb;
    };
    uniRecord(const float &stars) { schema = ratingVisual; };

    /*... for each record type */

    auto &build(void) {
      auto o = createElement<li>();

      // build interface based on the schema selected from enumeration
      // as a simple example. so the data input would be a parameter to
      // some kind of specialized formatter, user preferences,
      // who knows - linked to shared memory, or some disk data.
      switch (schema) {
      case contact:
        o.appendChild("contact info");
        break;
      case filelist:
        break;
      case pictureThumb:
        break;
      case ratingVisual:
        break;
      case videoThumb:
        break;
      case audioThumb:
        break;
      }

      // link the element object to this one.
      o.setAttribute(*this);
      return o;
    }
  };
  /* format detailed views easily ...
  now records within the vector template type,
  combined with
  the std::function creates the view through dynamic visualization.

  Because the
  record format is usertyped always, only defaulted for std::string vectors
  adapted by constructor. ie:

  fname, string_view lname, string_view phone,
  string_view email
  */

  view.appendChild<ul>(indexBy{"uniRec"}).data<uniRecord>() = {
      {"anthony", "matarazzo", "(666) 123-4567", "simple@sss.com"},
      {"kevin", "styemark", "222 333 4444", "kevin.styemark@sss.com"},
      {"brenda", "rollerbank", "222 333 4444", "brenda.rollerbank@sss.com"},
      {"cindy", "trustinalo", "222 333 4444", "cindy.trustinalo@sss.com"},
      {"samantha", "skapity", "(123) 457-8906", "samantha.skapity@sss.com"},
      {"Tony", "Mowkar", "(777) 765-4321", "zitapps@sss.com"},
      {"mark", "maglich", "(800) 869-1234", "mark@sss.com"},
      {"cyclemount.jpg", 33234, 0x994834},
      {"greybirds.jpg", 8877394, 0x7564774},
      {"grapefruit.jpg", 47764, 0x95785777},
      {"cantelope.jpg", 997787, 0x645654},
      {"bannana.jpg", 12217767, 0x77667844},
      {"apple.jpg", 2344, 0x676778534},
    {"orange.jpg", 5545, 0x99887},
    {"pineapple.jpg", 1346, 0x454567}};

  auto &ulItems = getElement<ul>("uniRec");

  // transform lambda
  ulItems.dataTransform<li, uniRecord>([](auto &o) -> auto & {
    return o.build();
  });

  vector<uniRecord> vegetables = {{"Broccoli"},        {"Broccoli Rabe "},
                                  {"Brussel Sprouts"}, {"Cabbage, Green"},
                                  {"Cabbage, Red"},    {"Carrot"},
                                  {"Cassava"},         {"Cauliflower"}};

  auto &vecItems = ulItems.data<uniRecord>();
  vecItems.insert(vecItems.end(), vegetables.begin(), vegetables.end());

  // takes info and induces the change.
  ulItems.dataHint<uniRecord>(vegetables.size());

  /*
  then building complex user interfaces is more objected oriented, separated
  from the data, and the data interface is the standard library.
  */
}
```

These series of tests all provide the easiest most concise building factory methods. The formatting string is parsed once upon creation. The c++ binary string literal interface is used for versatility. Within the string format, appearing and communicating with a based initializer list.

The formatting string, viewManager object notation, is a parsed input function providing the ability to generate a very complex dataTransform function for the data. That is, at times lambda's of this sort may be difficult to implement. Yet the syntax provides a templated mechanism for building tree and event patterns which format the information within the data<> easily.

The parameter is given as a string. The significance of the first value enclosed within the parenthesis notes what "name" the main input type is referred to as. This must be a container that supports std::get<#>. Ends with a colon. Next is the markup document mapped to the value input column (a tuple in this case). There are some inclusion of format descriptors in form borrowed from python. However this form uses the standard form %s and expands it. A nice time performance about this function is that the information is parsed only once. The internal mechanism creates a deep cloneable object that can also reflect vectorized expansion during build. Notice the ... expansion for tuple column 4. Formatting and gathering of values from the tuple are iterative based upon index. The function returned contains the logic to build the tree from vectors. The vectors also contain the formatter object calls themselves. This internal design makes visualization much quicker as creation of the object, tree structure, attributes, etc will be purely binary. So, small price - parse at transform creation. Yet still using and not interrupting data flow, nor decreasing the volume of data the transform may handle.

Adding an evaluator to the internal storage can provide selection of different templates based on the item. Therefore, multiple forms are given as input. The second form of the text template, provides the ability to have the system evaluate which "text template" to use based upon the boolean return of the given lambda.

The example below shows an example of using a dataTransform described as a string.

```
void test7h(Viewer &view) {

  /*   */
  /* a record type such as this, containing dynamic children
can be described in the form :*/
  using tagInfo = std::tuple<int, std::string_view, float, std::string,
                            std::vector<std::tuple<std::string_view, float>>>;

  auto o = view.appendChild<ul>(indexBy{"music"});

  o.dataTransform<li, tagInfo>(
                R"(  (t) :
                <li>{t[0],%0xd}
                <ul>
                <li>{t[1],:20}</li>
                <li>{t[2]}</li>
                <li>{t[3]}</li>
                <li><table>
                            <{t[4]...v, odd-even} tr>
                            <td>{v[0], titlecap}</td>
                            <td>{v[1], Currency}</td>
                            </tr>
                </table></li>
                </ul>
                </li>
)");

  vector<tagInfo> rr = {0,
                            "Anthony",
                            134.5,
                            "Horror films, science fiction",
                            {{"Planet Crack Core", 7.0},
                            {"Nachzehrer Fires PG", 43.4},
                            {"Alien Spacecraft R+", 97.1},
                            {"Jupiter Time Shift R", 64.3}}};

  view.appendChild<ul>().data<tagInfo>() = {0,
                                            "Anthony",
                                            134.5,
                                            "Horror films, science fiction",
                                            {{"Planet Crack Core", 7.0},
                                            {"Nachzehrer Fires PGMA", 43.4},
                                            {"Alien Spacecraft R+", 97.1},
                                            {"Jupiter Time Shift R", 64.3}}};

  view.render();
}
```

Designer Notes

Some key concepts to consider when designing the c++ code base are:

- Not all members of the data adaptor will have to be displayed. therefore not all subsequent dom members need reflection allowing operation as a clipped viewport.
- The size() method and index access according to display rendering position can show the scroll bar graph information.
- Internally, graphic buffers are lighter in memory management by using a clipped cache prune method of dom tree reflection of the templated typed vector data.
- The system provides operating in an error free rate according to limitations of vector standard interface conventions - that is no change. The actual data is exposed.
- The display of large information sets as these structures permits is inaccessible for screen resolutions. That is typically less than 100k of text at a time. So, a scope for the focus of code performance can be made in favor of clipped viewports over a std::vector.

Hint hashes about the location of the data is saved within the element structure. As well, within the onscreen data structure, hashed information that describes how the data is displayed on the screen is also summarized in hash. Attributes such as text, color, etc summarizes the attribute data. When the GUI programmer invokes the adaptor signaler with a dataHint, the displayed cache is search for numerical relationships that affect the rendering of the visualized or cached offscreen clipped non visible element. This combined with the fact the the on screen 'built' elements are kept to a minimal can be a design to research for the rendering engine and big data performance. This makes the dirtiness of textual attributes more efficient according to their necessity. That is, are items displayed?

Internally, the usageAdaptorState structure holds information about what is in the possibly large amount of unknown data. It holds also the information that is not clipped from view, a memory range, and an index range. From this standpoint, it should be kept current periodically and or logically. It can be updated by the hinting system since the system knows about the addresses of the data. In proper structure, vectors offer a range based approach to hinting as each element is stored contiguously in memory. Therefore, any numerical address given within a hint can reveal several aspects about an element. For example, inclusion within the set as a vector element simply by using a numerical less than and greater than of the pointer numerical address.

Once set inclusion is established, further investigation about the quality of the hint information and other numerical parameters can be used to find if a particular data item is reflected currently within the document object model proper. That is most of these items may be visited by the renderer. When the element is not within dom visibility, its textual visual surface area allocated for pixel data is freed. Other aspects such as graphic objects, audio, video, medium may be handled externally for better cache management according to principles of modern hardware operation.

Relocation and geometric allocation provided within c++ std::vector will provide also effective use of this as applications operate within well planned c++ memory subsystems. It appears as if the vector memory reserve grows logically according to the necessity of only demand. The worst case scenario happens rather quickly where visible items are stripped of their pointer reference data.

Always keeping in mind of the large data possibility, keeping the dom cache tight around the visible contained area will provide quicker recovery when the worst case happens. And after the vector memory has grown geometrically because of all the data movement, data movement not impeded at all by this design, it is assumed to be put off for larger amounts of time. New geometric changes occuring will be recorded in the _lastWorkLoad to indicate locations or other information needed to make the new remapping just a numerical range insertion for the visible element's data and gui elements. Index information to the element's address within the data adaptor is saved and used to compute their new data pointers.

Other references to consider

- https://docs.python.org/3/library/string.html#string-formatting
- https://docs.python.org/3/library/string.html#formatspec
- https://en.wikipedia.org/wiki/Polymer_(library)

Multicore Threaded

The system is designed as a multithreaded implementation. This is a consistent design pattern for modern microprocessors. That is, multicore processing is typical and necessary to encompass within the codebase for intelligent, efficient and lightweight designs. Without a multithreaded design the use of the base hardware is not comprehended within the system.

Currently the design is sustained for three top level threads. One thread for the user's codebase. Within the user's codebase the template dom interface acts as a communication mechanism to the visualization system. In this, the visualization system is notified of the data, parameters, options and box layout information. This is conceptually a shared memory resource and information is never duplicated. Depending on the implementation of the underlying renderer selected, the type of APIs used for memory communication will differ. As a shared memory resource, there are different capabilities present on the various supported operating systems.

The third thread is where the visualization system performs the actual work or rendering. This thread handles drawing, calculating the clipping region, calculating the layout, loading graphic resources from files, rendering and wrapping textual areas, and scanning for data that has changed. When changes are found that affect the screen, they are scheduled for the next screen update. The process of scanning is a low priority thread. It provides a safety net when dataHints are not provided. At times, depending on application utilization, this may be a welcomed fault tolerance.

The visual rendering system is designed as a cycle process that can update at 15, 30 or 60 frames per second. The cycle is designed to be a low resource consumer as the default operating mode. Yet when active rendering is scheduled, more processing power such as multicore processing is allocated on demand. One way this is achieved is by idle thread pooling. Meaning resources may be intelligently allocated yet paused. This thread pool is architecturally related to the third top level thread. This may also be conceptually noted by technologists as a “fiber thread”.

The C++ Document Object Model

To establish the Document Object model within a C++ program the API and objects below are used. Most importantly, the c++ native and STD data types are used. This makes development a natural process for any c++ developer.

Base Document Elements

The objects listed below can be compared to the HTML5 model as in the name. However they are distinctly different in that they are binary objects. These objects are used within the template parameter of the function building API. that is createElement<>, or appendChild<>, or append<>. It is a preference that the name and the design match as closely as possible the implementation of the W3C to decrease the learning curve.

Viewer

The Viewer object is the main window in which all elements are visualized. It can be considered the root of a document object model. An application requires this one in order for a view to occur. It contains the processing message queue for the operating system.

BR

Provides a carriage return within the text. Within the parsed textual input this tag can be represented as either capital or lower enclosed within greater and less than.

H1

Provides a document title of level 1. The label is primarily used to style titles and subtitles within a document. Within the parsed textual input this tag can be represented as either capital or lower enclosed within greater and less than.

H2

Provides a document title of level 2. This tag is usually referred to a sub title or the second level. Many scripts search the collection of these tags to create a table of contents. Within the parsed textual input this tag can be represented as either capital or lower enclosed within greater and less than.

H3

Provides a document title of level 3. This tag is the third level of a title. Within the parsed textual input this tag can be represented as either capital or lower enclosed within greater and less than.

PARAGRAPH

A paragraph of information. The paragraph has a default textIndent to space the first line. Within the parsed textual input this tag can be represented as either capital or lower enclosed within greater and less than. As well, <P> can also be used within the parsed textual information.

DIV

A divisor block of information. The default is that the layout flow is blocked to return.

SPAN

A span of information. The default is that the layout flow is continued.

UL

Establishes an unordered or bullet list. The siblings are expected to be LI elements.

OL

Establishes an ordered or numerically labeled list. The siblings are expected to be LI elements.

LI

This is the list item. Usually there are multiples of this element within the UL or OL containers. Textual content should be encapsulated as a textNode.

TABLE

A grid of information. Cells exist as a two dimensional vector of type tableData. The headings of the table are of type tableColumns.

IMAGE

Establishes a display of a 24bit RGBA image. Provides pixel access as array of colors. Images must be transformed to this type for view. JPeg or PNG has to be transferred for example.

MENU

A menu structure. The menu is the same functionality as a drop down menu.

textNode

Each element contains a textNode for textual information. The class object holds a STD: string.

See Also

- https://www.w3schools.com/tags/default.asp

Global Document API

These functions are accessible within the global viewManager namespace. They can be called without a referencing object. Typically these functions are used for instantiation of primary elements. Or to query the entire document object model.

createElement

The function allocates a new document element of the type passed within the template argument. The function accepts a variable number of attributes which override the base element's defaults. A reference to the newly created element is returned. However, the object is not part of any DOM until it is appended as a sibling or a child.

createStyle

The template function accepts a variable number of attributes. The call must contain a string that refers to an identification. This string identification token should be used in any element's public style vector for it to affect qualities of rendering.

query

The function accepts three types of query parameters as it is overloaded. The string based identification is set by the element's indexBy attribute. The string form of the parameter is also a powerful query language that is likened to CSS selectors yet provides more jQuery like searching. Searches can be scoped to a specific area of the document. Queries can also rely on order and relationships to other elements. It might also be advisable to create more functionality that provides index management for these types of searches.

The query function exists as a global function and as a public function of the element base class. The global function provides global document searching. The element’s inclusion of the member provides searching for the element and children.

The function also accepts a lambda function as a parameter. The lambda function must return a Boolean value noting a match. The return value is of type ElementList.

The star operator is used to note all items within the tree. Additionally the format of the string can provide operations to search specific document elements and their tree. Or sibling elements, or types of elements, or elements of a certain class. This type of functionality is present within the CSS such as selection of odd-even or n-th child are also supported.

```
// All elements within the document
for (Element &n : query("*")) {
  n.setAttribute(textColor{30,30,40});
}

// All elements within the document that are H1
for (Element &n : query("<H1>.*")) {
  n.setAttribute(textColor{30,30,40});
}

// All elements within the document that match the regular expression
for (Element &n : query("indexBy=\\employ*\\")) {
  n.setAttribute(textColor{30,30,40});
}

// All elements within the document that are paragraph elements and
// have content that match the regular expression
for (Element &n : query("<p>.data(\\employ*\\")) {
  n.setAttribute(textColor{30,30,40});
}

// All elements within the document that are paragraph and span elements and
// have content that match the regular expression
for (Element &n : query("<p>,<span>.data(\\employ*\\")) {
  n.setAttribute(textColor{30,30,40});
}
```

See Also

- https://www.w3schools.com/css/css_attribute_selectors.asp
- https://www.w3schools.com/cssref/css_selectors.asp

getElement

The function is a templated one. The template argument names the type to return. The formal parameter should be a string that notes an element's indexBy attribute.

hasElement

The function returns a Boolean noting the existence of an element with the identification.

See Also

- https://en.wikipedia.org/wiki/Document_Object_Model
- https://en.wikipedia.org/wiki/XML_tree

Element Document API

Each document element has the following public functions. Most of these functions return a reference to the element created by the operation. This provides for the chaining methodology within the invocation syntax.

data

A template function that provides the main formatting and building of document contents. The STD: vector is used as the container internally. The template parameter must name a structure, tuple type or class type. This format can be used within the initializer lists of a push operation, or as a complete array initialization. For simple, one line text operations, an array is still expected. So the function always has the same input format. This is the function that should be used rather than building the complete underlying document tree.

dataHint

The function is used to inform the visual rendering engine of data changes. That is with the STD:vector push invocation, a call to this function can update the layout quicker. The parameters are multi-datatype tolerant in that information passed may have completely different meanings. Such as records are added, records are deleted or records are modified.

dataTransform<>

The overloaded template class creates an object that internally that is used as a data transform of any data that is pushed into the element's vector. The first over load type provides the capability of using a function object. The second type provides a mapped value and function object relationship whereby the named tuple column is used as a lookup entry from any data. Another one is a Boolean comparison function which evaluates the data for the given transform function. The last overload provides a powerful generation that uses a string of a document tree snippet to build elements from the vector data. The string is parsed once and provides formatting capabilities of the data. Internally a generalized routine uses the parsed output in a binary fashion to build the document hierarchy with the specified formatters.

This example shows the dataTransform in action to build a colorized UL list. As well, the list is appended to and all of the data is then modified again. Notice the order in which the data is pushed and the dataTransform is added later. The system must have both of these items present in order for the rendering to operate. However, the system is tolerant in that if they do not, a type of view will occur, it may not be specific as the lambda presents. As well, this is a use case that must be tested for in the coded implementation. Non breakage.

```
void test7f(Viewer &view) {
  // build a fancy <ul> list with <li> children consisting
  // of varying details...

  static array<string_view, 3> sIconNames = {"base.raw", "contentIcon.raw",
                                             "normal.raw"};

  // dataAdaptor
  using tagInfo = tuple<int, string_view, float>;

  #define idRecords "id_records"

  // format detailed views easily ...
  view.appendChild<ul>(indexBy{idRecords}).data<tagInfo>() = {
      {0, "AC/DC", 9.9},
      {1, "Hell's Bells", 9.8},
      {1, "You Shook Me All Night Long", 7.2},
      {1, "Have a Drink on Me", 4.3},
      {1, "Squeeler", 9.1},
      {0, "Books", 10},
      {1, "The Button Bubble - Digital Economics By Anthony Matarazzo", 5.5}};

  auto &urecords = getElement<ul>(idRecords);
  auto &recordPlex = urecords.data<tagInfo>();

  recordPlex.push_back({0, "Variodic Blackhearts", .2});
  recordPlex.push_back({1, "Sympathetic Voting Machines", .001});
  recordPlex.push_back(
      {1, "Vectorized Signaling Auto Typed constexpr Candy", .101});
  recordPlex.push_back({1, "Ya Got To Write It All screamin interfaces", .22});
  recordPlex.push_back(
      {1, "TemPlated Desktops with INLINED ANIMATION MOTION _data", .667});
  recordPlex.push_back({1,
                        "a neutral index of finger motion path skeletal data "
                        "stored in reusable gesture rom",
                        .665});
  recordPlex.push_back({0, "The Stairsouppe Choordettes", .4});
  recordPlex.push_back(
      {1, "My Binary Blister Coniquebine <ani=greenGlow>megatron remix</ani>",
       .19});

  // allowing for very fast expansion for meta creations. Most likely,
  // the tags may mean something more preverse than numeric. So the balance is
  // in definition

  // formatting lambda
  // passed  T & - type, class or tuple
  auto &fnTransform = [](tagInfo &tag) -> li & {
    textColor vip =
        std::get<2>(tag) < 5.0 ? textColor{"blue"} : textColor{"purple"};

    auto &o = createElement<li>(textColor{"blue"});

    // set icon of the li element
    auto &n = std::get<0>(tag);
    o.appendChild<image>(sIconNames[n]);

    // set text
    o.appendChild<span>(std::get<1>(tag));
    return o;
  };
```

```
urecords.dataTransform<li, tagInfo>(fnTransform);

// Data insertion using the standard library
//
// get reference to actual memory. std interface.
auto &ulRecords = getElement<ul>(idRecords);

// get a reference to the data vector
auto &recordPlex2 = ulRecords.data<tagInfo>();

for (int i = 1; i < 10; i++)
  recordPlex2.push_back({i, "STD::MOVE", randomDouble(0, 10)});

// senses that 10 added to the end of a list of 1000 already/ - no display
// change.
ulRecords.dataHint<tagInfo>(10);

for (auto &n : recordPlex2) {
  if (std::get<2>(n) > 5.5 && std::get<2>(n) < 6.9) {

    auto &sv = std::get<1>(n);
    sv = std::string(sv) + std::string(" Wowza");

    // takes info and induces the change.
    ulRecords.dataHint<tagInfo>(n);
  }
}
```

query

The function works with the indexBy identifiers within the element’s attributes. Names can be matched using a regular expression. The function returns a vector of STD:reference wrappers. The method provides also a search language that is likened to jQuery. See the global document query function for more details.

parent

The function returns a STD:reference_wrapper element's owner. Note that the reference can be returned as an STD:nullopt.

firstChild

The function returns a reference wrapper to the first child owned by the referring element within the call. The function can be used to walk the document hierarchy within loops for example as an initial value. Note that the function can return a STD:nullopt when no children exists.

lastChild

A function that returns the last child element as a STD:nullopt within the document hierarchy of the referenced element. This can be used in a comparison for example that tests an unknown element with the property that it is the last one.

nextChild

The function returns the very next child of the referring element. A type use of this function is within a loop as an increment to advance.

previousChild

The function provides a decrement operator according to the document hierarchy. It returns a STD:reference_wrapper which can have a STD:nullopt value when walking the model in that direction is exhausted.

nextSibling

The function returns a STD:reference_wrapper which can contain a STD: nullopt value. The operation is useful in walking the document hierarchy in the sibling chain. The value represents the advance position. For example, all LI elements within a UL are siblings.

previousSibling

The function returns a STD:reference_wrapper which can contain a STD: nullopt value. The operation is useful in walking the document hierarchy in the sibling chain. The value represents the preceding position. For example, all LI elements within a UL are siblings.

childCount

The function returns an integer representative of the number of child elements within the document hierarchy. The count can be useful when numeric iteration is used in a loop such as a for loop.

styles

STD:vector of styles.to use. Only the textual identifier should be used to link the style. Use ceateStyle to group attributes.

appendChild

when document structure must be composed as a hierarchy, this function will add a child element to the referenced object. Typically this function is used within the lambda expression of the data templated interface.

append

The function is a dual document operation in that it creates elements and adds the element as a sibling of the referring element within the call.

setAttribute

A templated function for attribute handling. The function accepts a variable number of arguments. The arguments supplied should be one of the list of attributes. As well, data of the specified default supported types can be supplied as parameters.
Some examples of the setAttribute function in action.

```
getElement("mainArea").setAttribute(textColor{70, 70, 70});
getElement("mainArea").setAttribute(objectLeft{100_px});
```

getAttribute

The templated function retrieves a reference to data structure of the attribute. The function is useful in that it is templated as well passes a reference to the internal memory. This provides a very quick accept to representative updates. In conjunction, if only the internal memory information is modified, the dataHint pointing to the actual element that the data is part of, this will modify the view. Most attributes are stored within classes. These classes have typically two settings or more, refer to the documentation. For syntactic convenience, the names value and option are used to reference the primary and secondary settings. Within the c++ syntax, the auto keyword can be used to automatically create the necessary data type as the functions support an auto return deduction.

The examples below show the most common forms of the syntax:

```
auto [idRefText] = mainArea.getAttribute<indexBy>();

idRefText = "idView";

// must use set to invoke indexing of elementById
mainArea.setAttribute(indexBy{idRefText});

auto [d, opt] = mainArea.getAttribute<objectLeft>();

d = 900;

auto d2 = mainArea.getAttribute<objectLeft>();
```

```
d2.option = numericFormat::percent;
d2.value = 50;
```

clear

The function cleans all content from the object and children. But leaves the document structure intact.

move

The function sets the objectLeft, and objectTop in one callable function.

resize

The function provides one operation to adjust the objectWidth and objectHeight.

addListener

The function accepts two parameters the event type of the event enumeration. The second parameter is a std function, a lambda may be appropriate.

auto Element::addListener(eventType evtType, eventHandler evtHandler)

removeListener

The function removes the named event processing function.
auto Element::removeListener(eventType evtType, eventHandler evtHandler)

insertBefore

auto insertBefore(Element &newChild, Element &existingElement) -> Element &;

The function inserts an element within the document hierarchy. This operation is before the element. The first parameter is the new element. While the second parameter is an existing element.

insertAfter

auto insertAfter(Element &newChild, Element &existingElement) -> Element &;

The function inserts an element within the document hierarchy. This operation is after the element. The first parameter is the new element. While the second parameter is an existing element.

remove

The method removes the referred to element within the call. After the operation, the memory will no longer be available. All reference to the element will be invalid as well.

removeChild

The function removes named child element. After the operation is complete, the memory associated with the element is no longer valid.

removeChildren

The function deletes all child nodes associated with the element.

replaceChild

deletes the specified child and replaces it with the new one specified.

printf

an implementation of the c standard library function. Internally it uses the standard library and passes output to the element's child textNode.

ingestMarkup

a routine to parse HTML style text to document elements. Only the C++ template elements are supported. The function is useful for adhoc building.

print

A python style formatting output function.

```
getElement(“divTest”).print("Hello, {}!", "world");  // Python-like format string syntax
```

There is a library that can be used as a reference if not inclusion for the functionality.

https://github.com/fmtlib/fmt

<< operator

The stream operator is supported on all elements. Parsing of the stream contents can be turned on and off using the literal DOMTAG and NODOMTAG which can speed up processing when necessary. The stream functions as a modern implementation provides.

Attributes

The list below is the base attribute model. In the c++ language, each of these are class objects. The element base class stores each attribute within a STD:map using the hash code of the type_id. This is efficient in that the list is a sparse storage area and retrieval uses the STD:find algorithm. The getAttribute and setAttribute templated functions should be used for attribute manipulation. An excellent feature of this mechanism is that the actual structure can be modified unencumbered by the mechanisms once getAttribute is invoked.

Another distinct difference is the use of "text" instead of "font" as a prefix in the character rendering collective attribute's name. This is highly regarded as a necessary change due to compilation requirements. That is, the full names are within the compiler namespace already. The windows.h header file consumes the "font" prefix.

As a specialized attribute, the indexBy property when set through API is automatically indexed for the query and other functions to use. The named string of characters is used as case sensitive during search unless changed through options within the regular expression.

The list is more consolidated compared to the W3C model and also provides great functionality. There are a few other changes made to the names that correct intention within the name, focusIndex for example is more readable.

indexBy

A string attribute that is used to index the element in a searchable STD:map. There are several functions that use this as a key. getElement, query, and hasElement are the major functions that use it.

display

An enumerated value that can have one of the following values:

- in_line
- block
- none

The setting controls the layout flow of the element within the rendered view. The in_line setting provides a continued horizontal flow while the block setting provides a returned flow to the next available position vertically with the horizontal reset. Finally, a setting of none hides the element from view yet it still exists within the DOM. The none is a typical way to turn visibility on and off for any element.

position

An enumerated value that can contain the following two settings:

- absolute
- relative

The default for all element's position attribute is relative. When the relative setting is active, the objectLeft and objectTop coordinates are used in the layout calculation based on their respective parent's position. When the setting is absolute, the coordinates are based on the 0,0 base of the document root.

objectTop

A numeric attribute with format representing the position of the object's upper location.

objectLeft

A numeric attribute with format representing the position of the object's left location.

objectHeight

A numeric attribute with format representing the height of the object.

objectWidth

A numeric attribute with format representing the width of the object.

scrollTop

A numeric attribute with format representing the top portion of the scroll viewport within the view.

scrollLeft

A numeric attribute with format representing the left portion of the scroll viewport within the view.

background

A color attribute specification that gives the element a color behind the content.

opacity

The value is a numerical value that controls the mixing strength of the element. The range is between one (1) and zero(0). Zero being not visible while one is set at full strength.

textFace

A string attribute that reflects the ttf filename. The current directory is first and then the operating system font directory is used. The system appends the .ttf extension if not provided.

textSize

A numeric attribute with format specifier the specifies the textFace rendering size. Typically the size should be expressed in pt or em units. An 11pt is the typical typographic size for paragraph size text.

textWeight

A numeric attribute that controls the impact of the textFace. 400 is default while 800 is bold.

textColor

A color specification that describes the rendering quality of the text.

textAlignment

An enumerated value that controls the alignment of all children including text. The following values are valid:

- left
- center
- right
- justified

textIndent

A numeric value with format that controls the first line within a text block that may be wrapped within its container. This is most appropriate for paragraphs.

tabSize

A numeric attribute with format specifier that controls the spacing when tab characters are encountered. This distance is evenly used once for each tab character within the textNode string.

lineHeight

a numeric value with an enumeration option. valid enum options are: normal, numeric marginTop

This sets the space outside of the document element at the top. The margin properties control the space outside of the border, including the content inside.

marginLeft

This sets the space outside of the document element at the left. The margin properties control the space outside of the border, including the content inside.

marginBottom

This sets the space outside of the document element at the bottom. The margin properties control the space outside of the border, including the content inside.

marginRight

This sets the space outside of the document element at the right. The margin properties control the space outside of the border, including the content inside.

paddingTop

The attribute sets the space at the top of the element before the content. The padding of an element specifies the space around the content and the border.

paddingLeft

The attribute sets the space at the left of the element before the content. The padding of an element specifies the space around the content and the border.

paddingBottom

The attribute sets the space at the bottom of the element before the content. The padding of an element specifies the space around the content and the border.

paddingRight

The attribute sets the space at the right of the element before the content. The padding of an element specifies the space around the content and the border.

borderStyle

a numeric value with an enumeration specification. valid enum options are:

- none

- dotted
- dashed
- solid
- doubled
- groove
- ridge
- inset
- outset

borderWidth

The borderWidth is a numeric value with a format specifier. The attribute controls the width of the border. A border width of zero is not visible.

borderColor

The borderColor attribute is a color setting. It controls the color of the line around the content.

borderRadius

When a curved border is desired, setting this to a degree will control the curvature of the line drawn. 90 is square for example while 45 is perfect circle.

focusIndex

The focusIndex property sets the tab order of the control. That is, when the user tabs from field to field, this number sequentially informs the order. A setting of zero (0) means that it is not within the tab order. One (1) is the first field within the tab index. Automatically keyboard focus is placed at this position.

zIndex

The zIndex controls the order of the layout and overlapping order. A value of 0 is the default plane.

listStyleType

An enumerated value. valid options are:

- none
- disc
- circle
- square
- decimal
- alpha

- greek
- latin
- roman

When UL or OL lists appear, this setting controls the style of the icon.

See Also

- https://en.wikipedia.org/wiki/Attribute_(computing)
- https://en.wikipedia.org/wiki/HTML_attribute

Events

The events within the system are designed to be compatible with the same meanings as the W3C model. These events can be canceled as well as changing the bubbling properties of the event's flow within the process of event evaluation.

The system will be expanded to handle touch and gesture events. Future processing will likely include gestures recognized by the on board camera as a gesture interrupt occurring as a result of secondary asynchronous hardware. But that is in the future and not needed for the beta.

paint

This event is activated for any event listeners before rendering of the content is started.

focus

This event is activated for any event listeners after an element has gained the keyboard attention.

blur

This event is activated for any event listeners after a field has lost the keyboard attention.

resize

This event is activated for any event listeners after an element's width or height has changed.

keydown

This event is activated for any event listeners when a button on the keyboard is pressed. It occurs before any other keyboard event.

keyup

This event is activated for any event listeners when a button on the keyboard is released. It occurs after other keyboard events, last in the order.

keypress

This event is activated for any event listeners when a button on the keyboard is pressed and released. It occurs in between up and down within the event model.

mouseenter

The mouseenter event is activated with the pointer is moved over an element.

mousemove

The mousemove event occurs often for each tracked mouse movement within an element that has captured the event. Typically this event may be used for certain types of user interface actions that require highlighting for a non tracked region.

mousedown

The mousedown event occurs when any mouse button is pressed within an element that has the event captured. The button is part of the event package sent to the applicable lambda.

mouseup

The mouseup event occurs when any mouse button is depressed within an element that has the event captured. The button is part of the event package sent to the applicable lambda.

mouseleave

The mouseleave event occurs when the element that had the mouse within its area is no longer a container for the mouse pointer.

mouseCursor

The mouse pointers are supported by the mouseCursor property within the base Element class. The following cursors are available as per the W3C environment. They are designed as a c++ class scope enumeration under the cursors namespace.

See also
W3C Cursors

```
enum class cursors : uint8_t {
alias
allScroll
auto
cell
contextMenu
copy
crosshair
default
grab
grabbing
help
move
noDrop
none
notAllowed
pointer
progress
text
url
wait
zoomIn
zoomOut

// resizing cursors are named a with an _ for better readability
col_resize
row_resize

e_esize
ew_esize
n_resize
ne_resize
nesw_resize
ns_resize
nw_resize
nwse_resize
s_resize
se_resize
sw_resize
w_resize

};
```

click

The typical event to be captured for a user interface event. The click event is defined as the first mouse button up and down within an element’s region.

dblclick

The typical event to be captured for a double click user interface event. The double click event is defined as the first mouse button up and down within an element’s region twice. The operating system has the specific amount of type set by the user’s preferences.

contextmenu

The contextmenu button is typically a right mouse button click. The event usually shows a menu at the specific area. The event passed to the lambda function allows for easy instantiation of the menu and structure as it contains the necessary coordinates translated for use by the MENU document element.

wheel

The wheel event occurs when the wheel is clicked or rotated. The event contains the delta from the previous amount. A positive and negative value indicates the direction. Typically this is used for scrolling a document's content vertically. The click event, optionally available on most pointing devices.

See also

- https://www.w3schools.com/jsref/dom_obj_event.asp
- https://en.wikipedia.org/wiki/Human_interface_device
- https://en.wikipedia.org/wiki/Interrupt_handler
- https://en.wikipedia.org/wiki/Event_(computing)

UX Common Data Entry Control Library

viewManagerUX.hpp

This part of the document contains the namespace design for the c++ GUI DOM UX base control library. The UX namespace is a series of classes that implement common data entry controls within the viewManagerUX.hpp header file. The common user interface control library contains a set of data entry components that operate intrinsically on STD types or base c++ data types as input and output. The purpose is to demonstrate a robust internal design such that the greatest benefits can be achieved in coding productivity when providing GUI applications.

UX Library Graphics

One important focus of the library is the instantiation of graphic modeling using three dimensional geometry. It is preferable to incorporate ray tracing as an offline process. In this, pre rendered content is loaded from a graphic cache.

Blender is the selected raytracer that will be used. The application provides the capability for modeling within its interactive interface. As well, by invoking Blender using python scripts, attributes specified by the developer can be evaluated for interface aesthetics. The output of this process is stored within a two dimensional image database for low cost block level image transfer also known as a BLIT.

Blender uses Python as its scripting interface. Each control will have its very own model file. For simplicity, these model scripts read an external file to propagate the selection of model attributes such as texture, bump mapping, and lighting color. As most user interface components are comprised of various sub elements, there will be a common set of attributes each share within it model space such as base color. As well, there may be several controllable attributes that are specific to a particular control. The communication file format should be human readable for ease of this purpose. This file can be compared to activex ocx property bags and COM interface discovery. Further refinement suggests that documenting these can aid both the art maker and developer.

The focus of using a pre rendered graphic database should be that of a performance orientation. However in doing so, there are numerous design trade-off decisions to be accounted for. For example, resizable adherence becomes one concern. And solving the issue of screen resolution, or dpi size usability, must be achieved within the design.

One way this can be accomplished is by using uniquely named databases for each of the supported LCD measurements.

Creating surface areas within the model to promote readable text is another major focus of modeling and design that has to be maintained within the image composition. Text labels will be composited in real time by the c++ template dom so this aspect of GUI visualization is not a Blender modeling responsibility.

Controls that are comprised of parts must provide addressable sub elements as images for reuse or to be used as a clipped resource within the resulting compiled database structure. Typically these sub elements are smaller items such as handles on the scroll bar control. The spinboxes are another example. The corners and beveled shadowed regions of alphanumeric editing controls should also be designed as distinct elements. This provides the capability to piece these items together for multiple rectangular sizes as needed by a document element's configuration. By storing large continuous areas as a complete rendering, the visual quality is not lost during the paste up and compositing phase. For example, the background of the paragraph text editing component might be configured to have a distinct yet subtle paper bump map attached. Yet when visualized on the screen layout according to the DOM configuration, only a portion is needed for the calculated region.

One purpose of the pre rendered rendered image database structure is to store related event animations of a GUI control. The push button and other handles can have these alternate series of images associated for example. As a database resource, it is understood that frames per second as a primary file storage criteria is applicable. This means that multiple files will exist for various screen resolutions and required frames per second. Not to worry, screen resolution is hard coded for the embedded device.

Gathering data of a graphic nature such as these shows storage size must be accountable within the design. The best known method of data reduction for these formats is the lossy mpeg format. A special implementation of this codec that is supported as a database format such is required. An existing mpeg codec can be used as the database holding the mpeg videos can be indexed by animation event name.

Blender as it performs its rendering work, sequentially numbers separate files for each frame. Therefore after all rendering tasks are completed, a program must build the graphic database file to be used in real time by the embeddable c++ template GUI DOM. The program should be written in the Python programming language as it exists within the model file.

Blender as a selected GUI element modeling tool has other benefits that can provide artist and development interoperability. The Blender user interface can be specialized for this purpose as part of its program design. It can be configured to show the necessary options that allow artists to design, model, and compile template DOM user interface controls. This is accessible through automation using python.

Within the specialized user interface, the visual artist must have several abilities provided for within the interface. The artist must be able to name animation events that are specific to the control. They must be able to establish keyframe points for common events such as hover, focus, blur, click or key press. Adding additional aesthetic behavior such as an interesting morph for a background surface of an application dialog. The communication of textual compositing regions is also a property that must be fulfilled at the artist design level within this user interface specialization.

Sub element component such as scroll handles and continuous regions to use for paste up are created in a per control element basis. So the control elements will be named by the developer. These names should appear within the blender user interface specialization.

The product produced by the artist will be a model file that has a property reader attached within its internal programming. In essence the combination of these various data files acts as a parameterized python rendering program that Blender can invoke in a batch oriented fashion. Blender has a command line option that can silence the user interface from showing known as the "headless" configuration. This allows the distribution of artist tailored control designs that have published and discoverable parameterized aesthetic attributes. The distributed python Blender model compiles the necessary template DOM graphic database for use by the developer in an easy to use package form. This process creates a production pipeline between creative artistic design idioms and a development application framework. In short, the process promotes a teamwork oriented approach in GUI visual communication to sustain an elevated quality better than modern methods provide. Noted that Blender as a resource is freely available however the ethical maneuver is to support the development of this tool as a royalty based payment for using it as a user interface rendering library when products are developed.

See Also

- This video shows the basic actions to use 2d polygons and extrude them to a three dimensional object. This process can be useful in creating many user interface objects. Blender Curve To Custom Shape
- These videos show how scripting and interactive editing can be combined to create scripts from complex editing actions.

    - Blender Tutorial Introduction to Python Scripting Writing a Script That Makes a Simple 3D Model
    - Blender Python Scripting, Bmesh Example Scripts Explained & Assigning Different Colors To Faces
    - Python scripting #1 in blender 2.8
    - Create an Add Mesh Addon in Blender (Python tutorial)
- Advanced Blender Python Programming
- Python Programming Language

UX Three Dimensional Rendering Properties

The following properties are available on UX controls. Check the specific control for the available properties for the control. Each of these properties are described in detail here but individually may not be applicable to all controls. As within the DOM operation, these properties are inheritance based., The developer can provide them once for the entire look and feel of the application controls.

zPlane

The zPlane property controls the position from the background plane of the window. It is a decimal value. This can cause a raised or sunken appearance depending on positive or negative values given.

displaceStrength

The displaceStrength controls the parameters which affect surface rendering characteristics. Internally this is a parameter that is used by the script which develops the surfaces. The blender Displace Modifier has several parameters yet including this one as a main input provides simplistic controls. Thus leaving the interpretation and actual aesthetics up to the artists definition.

displaceFrequency

The displaceFrequency controls the parameters which affect surface rendering characteristics. Internally this is a parameter that is used by the script which develops the surfaces. The blender Displace Modifier has several parameters yet including this one as a secondary input to provide simplistic controls.

shininess

The shininess property controls the surface's reflective properties. In modeling terms this is part of the material's definition.

normalColor

The normalColor controls the color that the control appears without focus.

focusColor

The focusColor controls the color that is faded into when the control gains keyboard focus. The exact implementation of the color’s usage is defined by the UX artist within the control package.

hoverColor

The hoverColor controls the color that is faded into when the mouse is moved over the area.

disabledColor

The disabledColor controls the color that is faded into when the control becomes disabled.

animationLength

The animationLength controls the length of each of the effects. It is a decimal number that is expressed in seconds. 1.0 is one second while .5 is half a second.

dataAreaRadius

The dataAreaRadius controls the squareness of the labeling area. While the data property controls the textual information that is displayed on the button itself.

sceneDefinition

A filename pointing to a scene definition object that is essentially a blender Python script. The major functionality is purposeful in that business graphics are generated using a component oriented approach. A cache is maintained and these are used first.

Base sceneDefinition Objects

fancyText

The fancyText scene definition provides a rendered raytraced static text that may be textured, bump mapped, and beveled. The control extrudes the font with beveling to achieve a three dimensional model that provides beautified static text. The is great for special looking titles. The control provides many great looking presents with included textures and presets. Typically this is used for title headings within document pages for improved aesthetic qualities. The control provides background graphic settings upon a layered composite image. The text can follow a selected curve or a curve specified within a binary bitmap mask.

graph
For graphing, a very animated approach to plotting and part selection can refine presentations. There are numerous resources to draw from for the design implementation. Mainly the JavaScript approach seems to be the best design component for parameter reduction. In this, perhaps the knowledge of plotting, types of useful graphs and direct stories can be learned. The https://plot.ly library is a great javascript component.

present
A presentation suite that provides easy input of multiple navigation elements within a templated approach. The ability to tie other sceneDefinition objects is necessary. Within this category, perhaps an out liner is necessary. Page transition and interesting animations. The ability to integrate models that rotate and translate according to media communication necessity. The comparable modern technology is Microsoft Powerpoint and Libre Impress. Several themed approaches is suited.

UX Supported Base Controls

The library for phase one will have the following entry components available. The controls as mentioned are derived from the W3C standard. The implementation of the control library, as it supports an expanding number of third party controls, is provided in a series of C++ classes. To allocate one of the UX controls, the UX::controlName is placed within the template parameter. Each of these controls derive from the Element base class.

The UX name space provides the capability of accessing specialized input format controls that have prefabricated user interface response programming. These objects are typically the text editor or check box for example.

See also W3C Input Types

text
password
multiline
number
masked

These controls provide entry of various formats of data within a defined rectangular area. When in focus, the keyboard cursor shows. These controls are comprised of the margin area (a), border (b), pad area (c), and entry area (d). When information is entered, the character is displayed within area (d) while the cursor advances to the next available position. Information is defaulted to be aligned on the left however this may change due to format requirements or even language. When data has filled area (d), the information scrolls to open new space on the right or left depending on requirements. The margin area of the control provides the ability to define the geometric design flow between the background and the border. The border area of the control provides the box or enclosing shape. The padding area is the space between the text entry portion and the border. These properties are inherited from the main Element class.

Graphic Parts

1. border upper left corner
2. border upper right corner
3. border bottom left corner
4. border bottom right corner

5. border upper line connector
6. border lower line connector
7. continuous entry area

Theses editing controls provide a customizable bevel that surrounds the editing box. The border width supplies the ability to define a more or less obtrusive border. Other attributes such as flow and blending into the background are informed by the margin and padding properties. Typically the frame is geometrically lite from top to bottom implying shininess on the top part of the beveled border while darker within the crevices. As well, the zPlane either raised or sunken affects the amount of shadow rendered.

Properties
In addition to the derived element attributes, the following also control the rendering. It is presumed that the list will grow, however providing a very simple interface for parameterization allows some global control over rendering properties yet provides the artist complete freedom in pronouncing the qualities of interactiveness or fanciful play.

Applicable UX 3D Properties
- zPlane
- bevelRadius
- displaceStrength
- shininess.
- normalColor
- focusColor
- disabledColor
- animationLength

Events
change - the change event occurs when information is modified.

The standard Element events are handled through polymorphism.

The artist can supply animations that automatically play when the event occurs within the system. See Element for the list of supported events.

pushButton

The control provides the ability to process a click from the mouse button to perform an action. The button appears as that of one on a remote control or a piece of stereo equipment. When the mouse is located over the button area and pressed the area depresses to show that it has been touched. When the mouse button is released while over the button area, the action set for the control is fired. However, if the button is pressed down and the mouse is moved away from the button area, the event is not fired.

The new looking buttons provide the ability to control the depth and height of the button, the shape of the border as well shape of the textual labeling area. Several colors can control rendering aspects of the animations such as glowing on mouse over. This glowing light is controlled by the focusColor property for example.

Graphic Parts

1. button upper left corner
2. button upper right corner
3. button bottom left corner
4. button bottom right corner
5. button upper connector
6. button lower connector
7. data label area

In addition to the derived element attributes, the following also control the rendering.

Properties

- dataAreaRadius
- zPlane
- bevelRadius
- displaceStrength
- shininess
- normalColor
- focusColor
- hoverColor
- disabledColor
- animationLength

The artist can supply animations that automatically play when the event occurs within the system. See Element for the list of supported events.

radioButton

The control provides the selection of a one radio button from a displayed group. When the item is selected a colored dot appears to mark its selection within the interface. It is left to the artist to define how this dot appears. This user interface metaphor is similar to the scantron testing form in which five or less options are selectable for a particular question.

Graphic Parts

1. radio area non selected
2. radio area selected
3. radio area data label

Properties

- dataAreaRadius
- zPlane
- displaceStrength
- shininess
- normalColor
- focusColor
- hoverColor
- disabledColor
- animationLength

hotImage

The hotImage control provides a three dimensional rendering of static text and layered graphic images that relies on the sceneDefinition data convention. It is useful for titles on documents or title pages. Complex data graphs is another useful component. The text can be beveled on the background zPlane and front face. The text is extruded for depth to give it a three dimensional appearance. The text geometry may be textured and bump mapped. The background can be textured as well. Text may be set to follow a curve.

Properties

- sceneDefinition

group

The group container provides a nice frame around the objects. When used with the radio button, the group container provides easy keyboard control using the up, down, left and right arrow keys to move the radio selection between the items within the group.

Two groups can be connected with using the resizers. The resizers accept two groups as attributes. This allows user interaction with the interface.

Otherwise, when the item is used as a group it provides only the border around the items acting as a picture frame. There are no data entry options for the item as it does not specifically provide editing features. However, because it inherits from the Element as its base class, all keyboard and user interface events can be employed and intercepted. This may be an unusual circumstance, as the programmer may wish to use a specific keyboard operating facility for the frame.

Graphic Parts

- border upper left corner
- border upper right corner
- border bottom left corner
- border bottom right corner
- border upper line connector
- border lower line connector
- continuous frame area

Properties

- zPlane
- displaceStrength
- shininess
- normalColor
- focusColor
- hoverColor
- disabledColor
- animationLength

checkBox

The check box control provides an on / off switch that is not grouped with any other items. The user interface metaphor is similar to that of a paper form in which the user pencils in a check next to the item if it applies to the question. The control can be turned off by clicking on the label or upon the box. The space bar on the keyboard can also be used to turn the check box on and off.

Graphic Parts

1. check box off

2. check box on
3. data label area

Properties

- dataAreaRadius
- zPlane
- displaceStrength
- shininess
- normalColor
- focusColor
- hoverColor
- disabledColor
- animationLength

date

The date control provides entry of a valid date. The date can be entered numerically. Also a drop down calendar can be used in selection.The std::time_t type is used for input and output.

dateTime

The date time control provides the selection of both components of a std::time_t type. The UI operation is exactly the W3C implementation.

week

The control provides selection of a week numerical value within a calendar year.

time

The control allows selection and numerical of a time. AM and PM may be modified using spinbox. Twenty four hour format is also available. The std::time_t type is used for input and output.

file

The control provides a filter single or multiple selection of a file object. When considering the view style, there are many design considerations. The dialog provides both input and output file name entry. As well, locking the area to a directory structure is important for types of secure applications.

In modern desktops, particular icon and previews exist. However, for cross platform availability, the system provides its own cataloging feature. If enabled, a database exists with the information. The nature of this feature may be scoped to certain file types based on the complexity of indexing.

verticalScrollbar

The control provides the vertical range graph where a handle is used to select a scroll position.
There are two directional buttons on either side, a range based sliding button and a graph where the range slider moves upon.

horizontalScrollbar

The control provides the horizontal range graph where a handle is used to select a scroll position. There are two directional buttons on the top and bottom, a range based sliding button and a graph where the range slider moves upon.

resizerVertical

The resizerVertical provides the ability to arrange two group boxes.

resizerHorizontal

The resizerHorizontal provides the ability to arrange two group boxes.

listSelector

The listSelector is the combination of a drop down list and a listbox. The default for the listSize , or number of items displayed is one. When one, it is a drop down list. However, changing the listSize property creates a different view of the items into a scrollable list box. This is the same basic operation as the <select> control within W3C HTML.

menu

The menu UX provides a drop down menu that is generated via LI and nested UL elements. However, for simplicity, the data<> templated function is used and arsed for easy menu creation.

gridEdit

The grid editor provides a flexible yet consolidated way to provide grid based editing of data using other controls. It supports sorting and items such as moving rows. The base feature set is derived from the JS Grid control

tabbedPanel

The control provides a tab notebook interface.

sliderRange
knobRange

The range selection provides the ability to choose numeric information using the mouse.

Events
change - the change event occurs when information is modified

accordion

The accordion control provides a collapsable space to show information is less space. Within the data input mechanism, the stream is provided as a two dimensional std::string array. The first column of the array provides the input for the name that appears on the tab. The second column provides a space for tagged content. The control comes from the jQuery library (jQuery Accordion).

progress

The progress bar provides the graphical representation of a processing graph.

dialog

The dialog control provides a draggable inner window. A message can be displayed within the dialog using the data<> vector string. An option from the dialogStyle enumeration can be also supplied provide common message box buttons that automatically format a “Yes” or “No” and “OK” or “Cancel” operation.

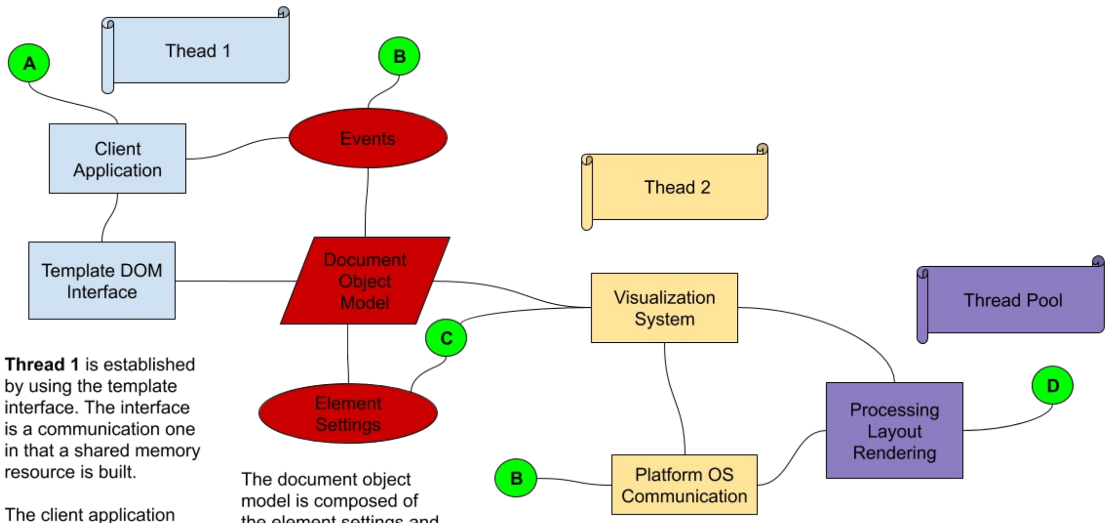
```
enum dialogStyle {
default,
yesNo,
okConfirm
}
```

Example UX DOM Interface

The example below shows a complete user interface using the UX DOM Interface. Notice the inclusion of the viewManagerUX.hpp header file. This header file controls the platform linkages to the UX control library held locally on the machine. The UX namespace contains the entire object classes for the common editing controls. The names are very simple as previously mentioned. The one downside is the UX:: must be given within the template parameter as the names of the controls are very simplified and will interrupt the client.

```
#include "viewManager.hpp"
#include “viewManagerUX.hpp”

using namespace std;
using namespace ViewManager;

#if defined(__linux__)
int main(int argc, char **argv) {
// handle command line here...

#elif defined(_WIN64)
int WINAPI WinMain(HINSTANCE /* hInstance */, HINSTANCE /* hPrevInstance */,
                LPSTR lpCmdLine, int /* nCmdShow */) {
// command line
#endif -

  auto &vm = createElement<Viewer>(
      objectTop{10_pct}, objectLeft{10_pct}, objectHeight{80_pct},
      objectWidth{80_pct}, textFace{"arial"}, textSize{16_pt}, textWeight{400},
      textIndent{2_em}, lineHeight::normal, textAlignment::left,
      position::relative, paddingTop{5_pt}, paddingLeft{5_pt},
      paddingBottom{5_pt}, paddingRight{5_pt}, marginTop{5_pt},
      marginLeft{5_pt}, marginBottom{5_pt}, marginRight{5_pt}, zPlane{.2},
      displaceStrength{.1}, shininess{.5}, normalColor{“white”},
      focusColor{“blue”}, hoverColor{“lightblue”}, disabledColor{“grey”},
      animationLength{1}, dataAreaRadius{30});

  vm.appendChild<UX::text>(indexBy{“txtFirstName”},”Anthony”);

  vm.appendChild<UX::text>(indexBy{“txtLastName”}, ”Matarazzo”);

  vm.appendChild<UX::pushButton>(indexBy{“txtLastName”},”Ok”);

  vm.render();
}
```

DOM Renderer Architecture



Thread 1 is established by using the template interface. The interface is a communication one in that a shared memory resource is built.

The client application can have other threads yet **Thread 1** is applicable as a design within this view.

The document object model is composed of the element settings and the Events. This is a shared memory resource

Thread 2 is where the conceptual management of the visualization and platform are achieved. The platform object contains the windows os message queue resides.

Thread Pool is where the visualization occurs. This continuously runs as a loop. There may be several worker threads that perform tasks. The loop may be partly idle if waiting for work or looking for needed changes. Otherwise processing of font rendering and drawing occurs.

Connector A notes the application start which is the standard c++ main function. When the client application builds the Viewer object, the system is initialized with the platform object and proper threading pool. The threading model uses the standard library which supports cross platform operation. Thread 1 is the communication thread where information is built into a document object model using smart pointers. When textNodes are inserted, the thread pool will analyze the text to find wrapping positions within the thread pool. Another thread within the pool may be locating layout positions and the applicable elements. To sequentially prioritize rendering work to items necessary according to viewport clipping is the desired functionality.

Blender Automation and Specialization

The blender specialization provides artists with the capability to package UX rendering controls that operate on input data, provide input of defined attributes, and allow c++ developer to process events named. The packages are referred to a sceneDefinition objects.

When the rendering of the element occurs, the element's data<> is streamed in the proper version format. The rendering occurs by a headless invocation of the script. As always, proper authoring of a component that performs a rendering function should provide well named attributes. For example, a textual rendering component may include properties for a textured map. The data in this type of call would contain the title. Or a graphing package may include options that inform the visualization of the title and axis labels. Reliance is placed within the art package for publishing interfaces. Within the blender user interface specialization, the artist simply names parameters using camel case without spaces for the property name and specifies a data input type.

The following data types for properties are supported within the C++ attribute storage code. These are C++ macros stored within the viewManager.hpp file.

- _STRING_ATTRIBUTE(propertyName);
- _ENUMERATED_ATTRIBUTE(propertyName, opt1,opt2,opt3,...);
- _NUMERIC_WITH_FORMAT_ATTRIBUTE(propertyName);
- _COLOR_ATTRIBUTE(propertyName);
- _NUMERIC_ATTRIBUTE(propertyName);
- _VECTOR_ATTRIBUTE(propertyName,dataType);

The sceneDefinition UX object offers a component based approach to complex modeling tasks that would take experts days to visualize and create. As a distinct server or externalized batch oriented process that utilizes a versioned stream, components may extend the blender artist's ability to create data interfaces that transform input data into a rendering. This can provide provocative animations for events within the C++ DOM layer.

Several methods exist for transferring the data to the Python blender script. A c++ header and cpp file can be generated by the blender interface specialization which provides the UX control Element's specific attributes, events and methods. This c++ class along with its attribute declarations, form the interface. The package is within its own c++ namespace. This generated file provides encapsulated behavior that makes generating components less error prone. The communication can be directed to the rendering platform using a variety of methods. Options are to a client side process renderer using the Python calling convention directly, through a tagged data stream such as Google protobuf provides or a format such as XML for remote network rendering. An interface version must note the client's stream data formats. The google protobuf api provides features that may be used at this layer.

A great benefit of cpp class objects being used is that the interface is compiled type specific. It provides the UX artist with the tools to provide a welcoming interface to their rendering object library. However this does complicate their responsibilities making necessity of more developer knowledge useful.

This process leaves the story boarding concept to some types of complex presentation animations up to artistic implementation using the specialized blender interface. Image compositing and supportive client communication polygon tracking positions are stored within the subsequent graphic UX databases to enable hot region as compiled native c++ language. Multiple events of a type can be received from the UX control to uniquely identify a specific region. Within the process of business graphics modeling, the enhanced blender interface gives UX artists and developers the ability to create blender packages that output branded login screens, graphs with a story, fanciful static title texts and branded visualizations that composite with the other document elements for exquisite user interfaces.

Graphic Database

The codebase for reading and writing the format exists within two binary c++ compositions. Both of the objects provide python style bindings however C++ is the main entry point. There are three types of objects stored within the database mpeg animations, hotspot regions, and event notifications. The database api supports storage of existing mpeg files that are named as objects within a directory structure. The hotspot regions are a series of 2d polygon coordinates that are transformed from the three dimensional blender objects to the coordinates on the animation for hit testing, convex and concave.

Tested Well

As any tool requires, the system will require many types of tests such that programs of distinct qualities are generated for loop based repetitive iteration testing. It has been my experience that multiple threads can cause types of issues especially when user interface events occur at break or bug times. It would be preferable to have all of the various combinations of device and configurations to be auto tested with a user interface event simulation. Have a solid code base for global security.

Future Technology

The future of the platform is completely open once the base model is developed. The plan is to evolve the base into a faster production tool. Such as tools that provide drag and drop functionality for device printing. Or newer languages that provide LLVM intrinsic support while using BC as the transmission form.

Embedded and system on a chip ready

The c++ Template document object model, as it is binary, provides the pathway to completely offload the box model layout calculation to a newer type of video processor. This video processor will encompass true type font rendering to the graphics memory. Most video memories are addressed at the A000 block. It is hoped that perhaps the IBM legacy portions of communication be removed such as text mode. BIOS bloat removed as well.

Simply, the design of the c±+ template library will be strictly a communication interface to video memory. As such only the necessary clipped portions reside. Perhaps with proper design main memory areas will be sent as linear address pointers while providing read only contexts to shared memory for rendering.

Legacy Free codebase

- Mixing colors for 24bit only
- Only latest version of font technology.

Gaining Intelligence Learned from W3C

It is obvious why frameworks exist for the browser - ease of interface description. jQuery, Angular, Dojo, and MooTools are commonly used kits. While each have their place in the developer's toolbox, they exist to simplify application creation. In this, the c++ template DOM includes some of the lessons learned. For example, notice the query function.

It is perceived that these types of functional implementations be set apart from the main viewmanager.hpp file and exist within their own header library. So, as a base resource, viewManager supplies the same type of linear growth. Different types of applications may require forms of super user interface building unseen within the base model. This capability is supported by the architecture in all ways.

One type of base feature implementation that is a good idea is the idea of a streaming data adapter. For large datasets such as database recordsets of a significant payload size, a buffering algorithm that manages consumed RAM would be an important feature.

HTML5 element base architecture does include the feature. Planning to incorporate Oracle Berkeley DB and its standard vector library is important for types of persistent application storage.

Relational database recordset handling should also be offered. MySQL, Oracle, postgreSQL and InnoDB are popular. Other formats such as office documents, spreadsheet, XML, and CSV are support as a data layer. This feature is designed as a layer on top of the base yet resides transparently below super frameworks.

Secure Database Terminals For Government Facilities

With the creative ways in which the code base can be used as a low level construct, a network terminal that provides a secure code base and data entry facilities is applicable for many government instutitions. In this tool, the operating system present has only the software used for data entry and the Linux kernel. The printed operating system is designed by an application that is similar to the user friendly LibreBase or Microsoft Access. Yet the tools provide distinct capabilities of interdepartmental updating and operate on secure networks intrinsically. That is, all of these devices are resistant to the modern hacker and viruses. This provides the world of security where there are enormous problems currently.

Applicable Lightweight Embedded Devices

Paid service computing platforms are common within the cell phone market place. Yet with the low computing resources available through the C++ Template GUI DOM, the cost effective manufacturing capabilities provide new market paths in many other types of consumer products. For example, imagine a tailored device specific for the media giant Netflix. With a subscription only purchase the cost of the device can be accountable to pay for itself. This entices the manufacturer to embellish their product designs to encapsulate an aesthetic look for the content provider. The Netflix Viewer device might be manufactured for a type of shelf life that lasts approximately six months. It might have Bluetooth connectivity supporting headphones that can reach HD quality. As well, perhaps the sound transducers accumulate a higher decibel than a laptop would.

Providing the consumer with paid service providers that offer the long term but transferable operating system software will entice competition within the stagnant desktop market. Such embedded laptops will provide more effective use of the client device as its networking intelligence will be more advanced than web based consoles. Many of the applications that run formally upon the client can use local native execution time. Most importantly is the newer design for network editing to minimize client and provider communication.

The operating system providers applies more robust streaming and summarization technology for quality of user experiences in that logic and applicable data payload that is framed for the user's desire. As well, the user's predicted application usage.

Job studio kiosks that provide the employer artificially intelligent interviews. The purpose is to attain jobs on the spot and immediate next day employment services. By characterizing the terminals of this type as on the spot, potential employees will be at ease when entering the workforce.

Embedded cash registers with an onboard GPS can provide the small business owner with marketing published materials they manage. The GPS provides the capability for entry and management of third party advertising such as yelp or Yahoo addresses. The business can be entered into an electronic mall much easier. Customers that are specific shoppers can have tailored materials that they desire for the business. This provides the corporate persistence with a locally owned and managed persona capability. Small business will also be able to compete with quality advertising that meets market demands. Employee schedules are available on workers phone applet. The business system should adopt the platform services offered for the type of store.

Music creation embedded devices with quality server side rendering is an often overlooked market due to the real time player. However, the modern loop maker often spends hours and days creating the special type of sounds. With the resources of the rendering left to server processing, the device can be very cost effective.

Offline Vast Storage ROM / EPROM embedded devices

Offline devices potentially will not need rechargeable batteries yet last several months without the need for maintenance. The electronic paper display may be best suited. With EPROM devices the secure access to the information is provided with usability locks.

- Education books manufactured and printed by public schools. Designing interactive published content for computer aided instruction gives the student the capability to be involved at their own pace. The portability of the sincerely strengthen compared to laptops. Cost effective for government funded education. Offline can be argued as a strength of focus. The device is used one way. This gives the emotional bond necessary for the tool itself in believing in the purpose as iconic.
- StoryTeller Writing Tool. A full size keyboard and half height flipping cover screen. Many writers enjoy just doing that. Writing and creating. This does not require the internet connectivity or the wasting of battery resources for fancy animations. Simply, they wish to bring a complex story or film to light. The readability of the display is prefered to be paper. That is, in the sunlight, at the beach, a laptop is useless.
- Media viewers containing a set of non-transferable movies.
- Media posters
- Interactive tourists sites
- ROM music library from genre or prints from an artist group. Imagine the time when you are running along the beach and the cell phone giggling is just too much. With a device such as this, your very own music library can be burned into

a nice rom device that is compact, has a longer battery life, and is Bluetooth compatible for those great headphones.

- Returnable product catalogs from high end sellers. Boat and yacht sales of progress slowly but have large finance.

Development Roadmap and Supportive Technologies

The first phase of development is to secure the code base for two platforms: Microsoft Windows and Linux. The order to develop products and plan for support of numerous DOM renderers is based on proof of this product's cross platform operation. The first phase may be considered a complete implementation and not a beta. This phase must be completed with high precision planning with a complete unit stress test and quality analysis cycle. Any modifications to design must occur at this stage to save production time for later phases.

Additional DOM renderers will be incorporated to support the base model format as the next phase of development. The desire of releasing these additional DOM renderers is that no source code changes will occur in developed applications. As well, only proceeding to phase two after development API and architecture review. Regression tests and the developed test bed are to be incorporated in all phases of release cycles.

The first phase of product delivery will encompass the most beneficial model for embedded device applications. Focus on the development phases must be maintained to keep productive coding requirements. The holistic nature of the first phase release of the codebase is perhaps the largest undertaking. It is demonstrated by the completeness of a communication interface, a common user interface library supporting data entry components, and a document object model renderer. The communication interface is as described in previous discussion.

There are numerous items that require attention within this area. As a developer myself, I find that installing platform requirements for building applications using new technology is often cumbersome and elusive without reading. The desire for this technology with third party requirements is that it is easy and effortless to use. The installation should consider that it has many supported implementations types. To accomplish this goal, requires a very well tested codebase and integration within the tool chains. As such, here is a list of items that are necessary for global productivity.

- Install package building is a very important feature for releasing applications to the global market. The package integrity must be maintained to be free of harmful and malicious programs. As well, the program must at times maintain the licensing and lock code that is handled by software payment. As a caveat, perhaps daily or per use rental can be an established cycle for the consumer.

- A program that can parse a textual markup document and write a c++ program for compilation.
- Bootable image creation with Linux kernel. ROM, EPROM, img, iso, vmware, and virtual box appliance format support.
- Windows embedded support
- For a successful development technology, superb documentation must exist. The more professional the documentation is an axis point to the sustainability of the market place. This is one great reason that Microsoft has often excelled in some of their technology patterns. So, the documentation must be established as a manual not developer made notes. An organization that invests in technical writing and the development can pursue the market with ease.
- IDE integration. There are a few popular IDEs on the market. A times, the inclusion and debugging facilities of a DOM is often considered a browser only resources. In this way, we can learn from the Firefox and Chromium debugging facilities of layout and design. Often complex layouts have many nested siblings within their visualization. Other points of integration such as visual markup to static code base can simplify creation and offer designers a tool to create the necessary business forms. For example, visual basic made the world round with this type of editing facilities.
- Installing third party libraries as a wizard implementation is necessary. Consider the build of the imageMagick++ library a difficulty. However, it does have its very own wizard that is supposed to accomplish this. One issue I found when using this is a registry key needed to be set for my particular development setup.
- Integrated batch building for server nightly building is a feature most large corporate facilities implement along with automated testing. One would ask how does this library affect such a dedicated process. The changes would be minimal, but require attention. Items such as link and runtime dependencies are important.
- Quality analyst support for automated testing of business logic is a great capability that any software firm would like to employ. Because this technology is a GUI visualization tool, and uses a low level C++ language, there are many items to consider for quality analysis automated testing. The testing tool must have access to the document object model as a secure facility. That is, most likely a build and scripting support for the testing. Python and Basic are often applicable to use for testing. As a product, automated testing platforms can be a robust item to engineer. However, for a technology of this sort, it is best to include the facility.
- Developer GUI lint checking provides resource tracking.
- Providing various framework implementations for types of suggest application design models is a great method for showing how easy to use interfaces can be

built using the technology. Typically these paradigms piece together in multiple forms within a complete application. One aspect, along with example programs, is that the IDE integration support super application framework structures. This will make implementation by the developer easier.

Conclusion

The model presented here is a simple but similar document object model to the rapid design methods employed by the W3C. The syntax provides a completely native compilable GUI layout for the modern c++ language. As such, the many benefits are imprintable ROM storage. The lighter model provides less stress on computing resources to save in portable power. The newer application framework is more structured than preceding ones to simplify code bases world wide. The opportunity to refine the available rendering is present. This provides the capability for newer types of desktop interfaces to be develop using less software engineers within the team. As a result, creativity may flourish in user experiences. Finally, the capability to reduce modern desktop operating systems and their storage size provides sincere performance gains to computer users. In short, it is a new market technology across the board waiting for investment.

C++ Research Source

main.cpp

```
#include "viewManager.hpp"

using namespace std;
using namespace ViewManager;

void test0(Viewer &vm);
void test1(Viewer &vm);
void test1a(Viewer &vm);
void test2(Viewer &vm);
void test3(Viewer &vm);
void test4(Viewer &vm);
void test5(Viewer &vm);
void test6(Viewer &vm);
void test7(Viewer &vm);
void test7a(Viewer &vm);
void test7b(Viewer &vm);
void test7c(Viewer &vm);
void test7d(Viewer &vm);
void test7e(Viewer &vm);
void test7f(Viewer &vm);
void test7g(Viewer &vm);
void test7h(Viewer &vm);
void test7i(Viewer &view);
void test8a(Viewer &vm);
void test8(Viewer &vm);

void test10(Viewer &vm);

void testStart(string sFunc) {}

/**************************************************************************************************


**************************************************************************************************/
#if defined(__linux__)
int main(int argc, char **argv) {
  // handle command line here...

#elif defined(_WIN64)
int WINAPI WinMain(HINSTANCE /* hInstance */, HINSTANCE /* hPrevInstance */,
                   LPSTR lpCmdLine, int /* nCmdShow */) {
  // command line
#endif
  // create the main window area. This may this is called a Viewer object.
  // The main browsing window. It is an element as well.
  auto &vm = createElement<Viewer>(
      objectTop{10_pct}, objectLeft{10_pct}, objectHeight{80_pct},
      objectWidth{80_pct}, textFace{"arial"}, textSize{16_pt}, textWeight{400},
      textIndent{2_em}, lineHeight::normal, textAlignment::left,
      position::relative, paddingTop{5_pt}, paddingLeft{5_pt},
      paddingBottom{5_pt}, paddingRight{5_pt}, marginTop{5_pt},
      marginLeft{5_pt}, marginBottom{5_pt}, marginRight{5_pt});

  vm << "Hello World\n";
  vm.render();

  vm << "<h1>Hello World</h1>";
  vm << "<blue>Got to be a pretty day.</blue>";
  vm.render();

  test0(vm);
  test1(vm);
  test1a(vm);
  test2(vm);
  test3(vm);
  test4(vm);
  test5(vm);
  test6(vm);
  test7(vm);
  test7a(vm);
  test7b(vm);
  test7c(vm);
  test8(vm);
  test8a(vm);
  test10(vm);
}

string_view randomString(int nChars);
double randomDouble(double a, double b);
int randomInt(int a);
void randomAttributeSettings(Element &e);

/************************************************************************

************************************************************************/
void test0(Viewer &vm) {
  vm << "Hello World\n";
  vm.render();

  vm << "<h1>Hello World</h1>";
  vm << "<blue>Got to be a pretty day.</blue>";
  vm.render();
}

/************************************************************************

************************************************************************/
//! [test0]
void test0b(Viewer &vm) {
  auto e = createElement<DIV>(indexBy{"divTTT"}, "test");
  vm.appendChild(e);
  e.data() = {"This is replace as a data node"};
  vm.appendChild("<div id=testAdd/>");
  vm.render();
}
//! [test0]

/************************************************************************

************************************************************************/
//! [test1]
void test1(Viewer &vm) {
  testStart(__func__);

  auto &mainArea = createElement<DIV>(
      indexBy{"mainArea"}, objectTop{10_pct}, objectLeft{10_pct},
      objectWidth{90_pct}, objectHeight{90_pct}, textColor{50, 50, 50},
      background{100, 200, 200}, textFace{"FiraMono-Regular"}, textSize{20_pt},
      textWeight{400});

  auto &appTitle = createElement<H1>(indexBy{"title"}, objectTop{10_px},
                                     textAlignment::center, "Type XCB");

  auto &appSubTitle =
      createElement<H2>(indexBy{"subtitle"}, objectTop{10_px},
                        textAlignment::center, "A starter testing Application");
```

```
appTitle.setAttribute(objectTop{20_percent});
appTitle.data() = {"New Text in there."};

appTitle.data() = {"data(). New Text in there."};
appTitle.data() = {"But after we add it to the dom."};

mainArea.appendChild(appTitle);
mainArea.appendChild(appSubTitle);

vm.appendChild(mainArea);
for (Element &n : query("*")) {
}

getElement("title").clear().data() = {
   "A new copy is never created when appending happens., just std::move"};

vm.render();
for (auto &n : query("*")) {
}

getElement("title").data() = {
   "It uses the reference to the created object createElement."};

vm.render();

getElement("title").appendChild<H2>(indexBy{"subtitle2"}, objectTop{20_px},
                                    textAlignment{textAlignment::center},
                                    "by Anthony Matarazzo");

getElement("mainArea")
   .appendChild<PARAGRAPH>(
      indexBy{"bodyText"}, textColor{"blue"},
      "The information here is added to the document. Text is "
      "wrapped"
      "while other items remain. It is hoped that image and "
      "image "
      "processing will be fun. I think so. After I had learned "
      "that "
      "the ImageMagick library had been tuned, works with SSE "
      "in 64bit floating point format, I thought to myself, "
      "should be fast. ");

vm.render();
getElement("mainArea").setAttribute(textColor{70, 70, 70});
getElement("mainArea").setAttribute(objectLeft{100_px});

getElement("title")
   .setAttribute(objectLeft{100_px}, "The Visualizer")
   .data() = {"Mini App Title"};

getElement("bodyText").setAttribute(indexBy{"bodyInformation"}).data() = {
   "The new text information is not quite as long"};

vm.render();

auto statusText =
   createElement<H3>(indexBy{"statusText"}, "Status Line Updated:");

mainArea.insertBefore(statusText, getElement("bodyInformation"));

vm.render();
mainArea.insertAfter(
   createElement<H1>(indexBy{"titleText2"}, "A app built with gui tags."),
   getElement("bodyInformation"));

mainArea.removeChild(getElement("bodyInformation"));
vm.render();

getElement("mainArea")
```

```
    .appendChild<PARAGRAPH>(indexBy{"bodyText"}, textColor{"orange"},
                            "Re added and updated. Scatered coverered "
                            "diced and smashed.");
vm.render();

mainArea.replaceChild(
    createElement<paragraph>(indexBy{"bodyTextNew"}, textColor{"green"},
                             "And now, for a limited time. the all new ... "),
    getElement("bodyText"));
vm.render();

// attributes and references
/* to quickly change attribute values wihin the dom
                                                                 you can
   get a reference to the actual value stored within the tree. this is
   accomplished using the getAttribute function. The first parameter is
   the attribute name you wish to get. The second is the format of the
   characteristic. At most time, the first one is the one that is
   commonly used. However, most attributes contain a format specifier or
   other characteristics associated with the attribute. To provide
   efficient usage, the second parameter is the type of the
   characteristic. The type is is defaulted to double. That is, the
   common characteristic type is a double storage type. IE, objectLeft,
   marginTop
         ...



*/

if (mainArea.parent())
  mainArea.data() = {"Hey attached to another container."};

// walk children
auto n = mainArea.firstChild();
while (n != std::nullopt) {
  n = mainArea.nextChild();
}

for (auto n = mainArea.firstChild(); n != std::nullopt;
     n = mainArea.nextChild()) {
}

auto [idRefText] = mainArea.getAttribute<indexBy>();

idRefText = "idView";

// must use set to invoke indexing of elementById
mainArea.setAttribute(indexBy{idRefText});

auto [d, opt] = mainArea.getAttribute<objectLeft>();

d = 900;

auto d2 = mainArea.getAttribute<objectLeft>();

d2.option = numericFormat::percent;
d2.value = 50;

vm.render();

// styles and CSS
auto boldTexts = createStyle(
    indexBy{"boldText"}, textColor{"green"}, background{100, 200, 200},
    textFace{"FiraMono-Regular"}, textSize{20_pt}, textWeight{800});

mainArea.styles.push_back(boldTexts);
appSubTitle.styles.push_back(boldTexts);
vm.render();
```

```
}
//! [test1]

//! [test1a]
/**
 * \brief The test shows that entry of lowercase names is valid as the names
 * are aliased by the 'using' feature. 'div' is reserved, so 'dblock' is used
 * instead. h1,h2,h3,h4
 *
 * \note A very important item to embrace is method chaining. Method chaining
 * provides a consolidated description of elements and their hierarchy of
 * children. Yet when setting a reference, to correctly receive the proper value
 * at most times you CANNOT chain the createElement method. This is so, because
 * the LAST return within the chain is the value that the reference is set to.
 *
 */
void test1a(Viewer &vm) {
  testStart(__func__);

  ElementList chapter;
  int m = randomInt(100);

  // notice here that the createElement is not chained because a reference
  // is being attained to the element. Because of the way chaining works,
  // this must be a single function so that the correct reference is
  // computed.
  for (int i = 0; i < m; i++) {

    auto info = createElement<paragraph>(
        indexBy{"rndTEST3BookletParagraph_" + to_string(i)},
        vector<string_view>{randomString(200), randomString(200),
                            randomString(200)});

    info.appendChild<ul>(
        indexBy{"idbikes"},
        vector<string_view>{"Huffy", "Schwinn", "Giant", "Road Master"});

    chapter.push_back(info);
  }

  auto booklet = createElement<dblock>(indexBy{"booklet3"});
  vm.appendChild(booklet);
  booklet.appendChild(chapter);
  vm.render();
}
//! [test1a]

/************************************************************************

************************************************************************/
//! [test2]
void test2(Viewer &vm) {
  testStart(__func__);

  for (int i = 0; i < randomInt(100); i++) {
    auto &information = createElement<DIV>(indexBy{"rndDIV_" + to_string(i)});

    for (int j = 0; j < randomInt(100); j++) {
      information.appendChild<PARAGRAPH>(
          indexBy{"rndParagraph_" + to_string(j)}, randomString(200));
    }

    vm.appendChild(information);
  }

  // randomize attributes
  for (int i = 0; i < 1000; i++) {
    for (auto n : query("*"))
      randomAttributeSettings(n);
  }

  vm.render();
}
//! [test2]

/***********************************************************************

***********************************************************************/
//! [test3]
void test3(Viewer &vm) {
  testStart(__func__);

  ElementList chapter;
  int m = randomInt(100);

  for (int i = 0; i < m; i++) {
    chapter.push_back(createElement<PARAGRAPH>(
        indexBy{"rndTEST3BookletParagraph_" + to_string(i)},
        randomString(200)));
  }

  auto &booklet = createElement<DIV>(indexBy{"booklet3"});
  vm.appendChild(booklet);
  booklet.appendChild(chapter);
  vm.render();
}
//! [test3]

//! [test4]
void test4(Viewer &vm) {
  testStart(__func__);

  int m = randomInt(100);

  Element &eBooklet = vm.appendChild<DIV>(indexBy{"booklet4"});

  for (int i = 0; i < m; i++)
    eBooklet.appendChild<PARAGRAPH>(
        indexBy{"rndTEST4BookletParagraph_" + to_string(i)}, randomString(200));

  vm.render();
}
//! [test4]

//! [test5]
void test5(Viewer &vm) {
  testStart(__func__);

  ElementList chapter;
  int m = randomInt(100);

  for (int i = 0; i < m; i++) {

    auto &e = createElement<PARAGRAPH>(
        indexBy{"rndTEST5BookletParagraph_" + to_string(i)}, randomString(200));

    e.appendChild<PARAGRAPH>(indexBy{"rndTEST5BookletNotes_" + to_string(i)},
                             randomString(200))

        .appendChild<PARAGRAPH>(indexBy{"rndTEST5GuestSpeaker_" + to_string(i)},
                                randomString(200))

        .append<PARAGRAPH>(indexBy{"rndTEST5BookletReferences_" + to_string(i)},
                           randomString(200));

    chapter.push_back(e);
  }
  vm.appendChild<DIV>(indexBy{"booklet5"}).appendChild(chapter);

  vm.render();
}
//! [test5]

/***************************************************************************

***************************************************************************/
//! [test6]
void test6(Viewer &vm) {
  testStart(__func__);

  ElementList chapter;
  int m = randomInt(5);

  for (int i = 0; i < m; i++) {

    auto e = createElement<PARAGRAPH>(
        indexBy{"rndTEST5BookletParagraph_" + to_string(i)});

    stringstream ss;

    ss << "Hello "
       << "anthony"
       << "can you do the []";

    e << ss;

    e.appendChild<UL>(indexBy{"bookletNotes_" + to_string(i)},
                      vector<string_view>{"Endurance training", "Biking",
                                          "Meals", "Schedule"})
        .append<UL>(
            indexBy{"guestCompanies_" + to_string(i)},
            vector<pair<int, string_view>>{{0, "Gyms"},
                                           {1, "Gold's Gym"},
                                           {1, "Core Fitness"},
                                           {1, "Tommy Doright's"},
                                           {0, "Tools"},
                                           {1, "Huffy"},
                                           {1, "Scwitchers"},
                                           {1, "Clock Down Industrials"}})
        .append<UL>(indexBy{"bookletReferences_" + to_string(i)},
                    vector<string_view>{"The 26inch Road", "Flatters Chain",
                                        "Wheelers and Handlebars",
                                        "Rim's n Chains"});

    chapter.push_back(e);
  }

  vm.appendChild<DIV>(indexBy{"booklet5"}).appendChild(chapter);

  vm.render();
}
//! [test6]

/***************************************************************************

***************************************************************************/
//! [test7]
void test7(Viewer &vm) {
  testStart(__func__);

  ElementList vParagraphs;
  vParagraphs.push_back(createElement<PARAGRAPH>(indexBy{"testpara1"}));
  vParagraphs.push_back(createElement<PARAGRAPH>(indexBy{"testpara2"}));
  vParagraphs.push_back(createElement<PARAGRAPH>(indexBy{"testpara3"}));
  vParagraphs.push_back(createElement<PARAGRAPH>(indexBy{"testpara5"}));
  vm.appendChild<DIV>(indexBy{"test"});
  getElement("test").appendChild(vParagraphs);

  vm.appendChild<DIV>(indexBy{"mainArea"}, objectTop{10_pct},
                      objectLeft{10_pct}, objectWidth{90_pct},
                      objectHeight{90_pct}, textColor{50, 50, 50},
                      background{100, 200, 200}, textFace{"FiraMono-Regular"},
                      textSize{20_pt}, textWeight{400})

      .appendChild<H1>(indexBy{"title"}, textAlignment::center, "Type XCB")
      .append<H2>(indexBy{"subtitle"}, objectTop{40_pct}, textAlignment::center,
                  "A starter testing Application")

      .appendChild<SPAN>(indexBy{"idCompany"}, "research company name[].");
  // one part of the syntax that is error prone is snding the last item.
  // you have to have sure that the entire parameter is encapsolated inside
  // brackets... make several points about this in the documentation
  // since it is necessary....
  vm.appendChild<DIV>(indexBy{"footer"}, "Footer of Page")
      .appendChild<SPAN>(indexBy{"timeOfDay"})
      .append<SPAN>(indexBy{"currentDate"});

  vm.append("<ul><li>Hello added to the end</li></ul>");

  vm.render();
}
//! [test7]

//! [test7a]
void test7a(Viewer &vm) {
  testStart(__func__);

  ElementList vParagraphs;
  vParagraphs.push_back(createElement<PARAGRAPH>(indexBy{"testpara1"}));
  vParagraphs.push_back(createElement<PARAGRAPH>(indexBy{"testpara2"}));
  vParagraphs.push_back(createElement<PARAGRAPH>(indexBy{"testpara3"}));
  vParagraphs.push_back(createElement<PARAGRAPH>(indexBy{"testpara4"}));
  vParagraphs.push_back(createElement<PARAGRAPH>(indexBy{"testpara5"}));

  auto &divTest = vm.appendChild<DIV>(indexBy{"test"});

  divTest.appendChild(vParagraphs);

  divTest.append("<ul><li>Hello added to the end</li></ul>");
  divTest.appendChild("<p>Just adhoc dom building.</p>");

  // the append and appendChild with texts.
  divTest.appendChild("<ul></ul>")
      .appendChild("<li>test1</li>")
      .append("<li>test2</li>")
      .append("<li>test3</li>")
      .append("<li>test4</li>")
      .append("<li indexBy=ttt/>");

  vm.render();
}
//! [test7a]

//! [test7b]
void test7b(Viewer &vm) {
  testStart(__func__);

  ElementList vParagraphs;
  vParagraphs.push_back(createElement<PARAGRAPH>(indexBy{"testpara1"}));
  vParagraphs.push_back(createElement<PARAGRAPH>(indexBy{"testpara2"}));
  vParagraphs.push_back(createElement<PARAGRAPH>(indexBy{"testpara3"}));
  vParagraphs.push_back(createElement<PARAGRAPH>(indexBy{"testpara4"}));
  vParagraphs.push_back(createElement<PARAGRAPH>(indexBy{"testpara5"}));
  auto &divTest = vm.appendChild<DIV>(indexBy{"test"});

  divTest.appendChild(vParagraphs);
  divTest.append("<p>Appended after divtest as a sibling.</p>");
  divTest.appendChild("<p>Just adhoc dom building.</p>");

  // the append and appendChild with texts.
  // notice that you can appendChild to an element expressed by text.
  // The complete object and terminal must be given in one line.
  divTest.appendChild("<ul/>")
      .appendChild("<li>test1</li>")
      .append("<li>test2</li>")
      .append("<li>test3</li>")
      .append("<li>test4</li>");

  vm.render();
}
//! [test7b]

//! [test7c]
void test7c(Viewer &vm) {
  testStart(__func__);

  auto &divTest = vm.appendChild<DIV>(indexBy{"testAnother"});

  // the append and appendChild with texts.
  divTest.appendChild("<ul></ul>")
      .appendChild("<li>test1</li>")
      .append("<li>test2</li>")
      .append("<li>test3</li>")
      .append("<li>test4</li>");

  divTest.append("<ul><li>Hello added to the end</li></ul>");
  divTest.appendChild("<p>Just adhoc dom building.</p>");

  vm.render();
}

//! [test7d]
void test7d(Viewer &vm) {
  testStart(__func__);

  auto &divTest = vm.appendChild<DIV>(indexBy{"testAnother"});

  // the append and appendChild with texts.
  divTest.appendChild("<ul></ul>").data() = {"2222", "3333", "444", "6.66",
                                              "7"};

  divTest.appendChild<ul>().data<double>() = {1, 4, 5, 3, 4, 4, 4, 33, 4, 5};

  divTest.appendChild<ul>().data<pair<int, string_view>>() = {
      {0, "AC/DC"},
      {1, "Hell's Bells"},
      {1, "You Shook Me All Night Long"},
      {1, "Have a Drink on Me"},
      {1, "Squeeler"},
      {0, "Books"},
      {1, "Yada yada yada"}};

  divTest.appendChild("<Combo></Combo>").data() = {
      "option a", "option b", "option c", "option d", "option e"};

  divTest.append("<ul><li>Hello added to the end</li></ul>");
  divTest.appendChild<paragraph>().data() = {"fdff", "fdfdfdf", "Yttyty",
                                              "ghhhht"};
}

void test7e(Viewer &view) {
  auto &tblCost = createElement<TABLE>(
      objectLeft{10_pct}, objectTop{10_pct}, objectWidth{80_pct},
      objectHeight{80_pct},
      tableColumns{
          {"Name", "Employment Start", "Salary", "Sales", "Cost Ratio"}},
      tableData{{"Anthony", "1/1/15", "10.75", "34.16", "4.5 +"},
               {"Candy", "4/16/12", "12.75", "4464.76", "35017.2 +"},
               {"Alvin", "1/1/65", "4.75", "125.16", "2634.9 +"}});

  /// tblCost.edit(1,4);

  view.render();
}

// complex form - most common in applications because of the layout and
// summary information used to visualize.

/* The 'using' statement below creates an alias of a type - the artificial
name used by a coder for their internal coding communication. This name
is used a template parameter and thus is used to store in
a vector container internally within the element object.
Therefore the data is stored as an efficient tuple whose members are
accessed by const expression which are offsets into the allocated memory.

When using the data<> function, also templated, as a parameter, the
using storage alias provides readible syntax easily. This is one
type of communication interface generation for specific binary data
formats may be attained. Others such as objects can be created for
more complex structures and object oriented large scale business architectures.

The data<> function is robust. As well as possessing a non-copied live
binary data input mechnism, the formatting and document building are
created by compiled lambda's using std::functions. The mechnism for
creating a dataAdaptor, which is a c++ std::tuple data structure in this
example, is simply accomplished by the alias declaration 'using NAME=tuple<...>;
The mechnism for creating a 'real world' formatter is also compile
time checked. It is stored within a std::function<> which is called
to summon the formatted document element when it is required for visualization.

The templated class dataTransform provides document building functionality
efficiently.
The design separates the data from the "objects". Another captivativing
purity is that each element object created is linked to the
storage container's memory index when it is created. The dataTransform stores
a std::function internally. The object created becomes internally related to
the data all through using type information. That is, it is expected that
a particular type have a unique formatter.

The dataTransform templated class accepts one parameter: storage type.
All items within the system are stored within a vector. This cannot be changed
and makes the interface easier. The actual transform is a std::function.
The function can be expressed as a lambda or std::function compatible object.
The function is called once in iteration for each item within the vector.
The system after calling the document creation function links the element
to it's index if it is on the screen. When elements are not on the display.
their relationship is no longer linked.  The element is linked in two ways: by
unordered mapped key and direct data reference. The unordered map is useful when
data is edited, or rows inserted into the vector data structure using the data()
interface. This makes detailed interface creation simpler in syntax than a
parsing to memory structure does - less room.

The std::function option provides very easy syntax for binding to
class member functions and other types. Therefore, in particular
applications, or as a company architecture, formatting of this particular
sort may be accomplished as a business component. The architecture
supports these facets of modern design benefits.

*/

void test7f(Viewer &view) {
  // build a fancy <ul> list with <li> children consisting
  // of varying details...

  static array<string_view, 3> sIconNames = {"base.raw", "contentIcon.raw",
                                             "normal.raw"};

  // dataAdaptor
  using tagInfo = tuple<int, string_view, float>;

#define idRecords "id_records"

  // format detailed views easily ...
  view.appendChild<ul>(indexBy{idRecords}).data<tagInfo>() = {
      {0, "AC/DC", 9.9},
      {1, "Hell's Bells", 9.8},
      {1, "You Shook Me All Night Long", 7.2},
      {1, "Have a Drink on Me", 4.3},
      {1, "Squeeler", 9.1},
      {0, "Books", 10},
      {1, "The Button Bubble - Digital Economics By Anthony Matarazzo", 5.5}};

  auto &urecords = getElement<ul>(idRecords);
  auto &recordPlex = urecords.data<tagInfo>();

  recordPlex.push_back({0, "Variodic Blackhearts", .2});
  recordPlex.push_back({1, "Sympathetic Voting Machines", .001});
  recordPlex.push_back(
      {1, "Vectorized Signaling Auto Typed constexpr Candy", .101});
  recordPlex.push_back({1, "Ya Got To Write It All screamin interfaces", .22});
  recordPlex.push_back(
      {1, "Templated Desktops with INLINED ANIMATION MOTION _data", .667});
  recordPlex.push_back({1,
                        "a neutral index of finger motion path skeletal data "
                        "stored in reusable gesture rom",
                        .665});
  recordPlex.push_back({0, "The Stairsouppe Choordettes", .4});
  recordPlex.push_back(
      {1, "My Binary Blister Coniquebine <ani=greenGlow>megatron remix</ani>",
       .19});

  // allowing for very fast expansion for meta creations. Most likely,
  // the tags may mean somthing more preverse than numeric. So the balance is
  // in defintion

  // formatting lambda
  // passed  T & - type, class or tuple
  auto fnTransform = [](tagInfo &tag) -> li & {
    textColor vip =
        std::get<2>(tag) < 5.0 ? textColor{"blue"} : textColor{"purple"};

    auto &o = createElement<li>(textColor{"blue"});

    // set icon of the li element
    auto &n = std::get<0>(tag);
    o.appendChild<image>(sIconNames[n]);

    // set text
    o.appendChild<span>(std::get<1>(tag));
    return o;
  };

  urecords.dataTransform<li, tagInfo>(fnTransform);

  // Data insertion using the stdandard libary
  //
  // get reference to actual memory. std interface.
  auto &ulRecords = getElement<ul>(idRecords);

  // tagFormatter is needed for hash map to get the structure.
  // templated for user defined storage and reflexion.
  // state information is saved when getAdaptor is invoked,
  // so senseing what changes according to screen rules is
  // the problem to solve for the system that will create
  // high performance "implementation error free" solution

  // exception raised from non created item from absorb.
  auto &recordPlex2 = ulRecords.data<tagInfo>();

  for (int i = 1; i < 10; i++)
    recordPlex2.push_back({i, "STD::MOVE", randomDouble(0, 10)});

  // senses that 10 added to the end of a list of 1000 already/ - no display
  // change.
  ulRecords.dataHint<tagInfo>(10);

  for (auto &n : recordPlex2) {
    if (std::get<2>(n) > 5.5 && std::get<2>(n) < 6.9) {

      auto &sv = std::get<1>(n);
      sv = std::string(sv) + std::string(" Wowza");

      // takes info and induces the change.
      ulRecords.dataHint<tagInfo>(n);
    }
  }

  /* Using a hinted signaller will expose the versitility of the design
  and standard library usage. As well define typical usage
  intrinsic with data structure usage rather than reliance on
  the neccessity of parsed structure definition as html must have. w3c
  technology h1, h2, h3, div, and ^element. Meaning that these concepts
  of visual layout technology are the intellectual property of the w3c.
  So after it works, perhaps some working will have to be attained.

  a) not all members of the data adaptor will have to be displayed. therefore
  not all subsequent dom members need reflexition as a clipped viewport.
  the .size() method and index access according to display rendering
  position can show the information.

  b) lighter memory mangement according to clipped cache prune methods
  of dom tree reflection of templated typed data.

  c) operating in an error free rate according to limitations of
  vector standard interface conventions - that is no change. The actual
  data is exposed. While scrollbar numberical values offer higher
  memory performance than a browser . The display of large information
  sets as these structures permits is inaccessible for screen resolutions.
  That is typically less than 100k of text at a time.

  Hint hashes about the location of the data are saved within the element
  strcture. As well, within the onscreen datastructure, hashed information that
  describe how the data is displayed on the screen. Such as text,color,etc which
  summarizes the attribute data. When the gui programmer invokes the smart
  adaptor signaler with the hint, the displayed cache is search for numerical
  relationships that affect the rendering of the visualized or cached offscreen
  clipped non visible element. This uery combined with the fact the the on
  screen 'built' elements are kept to a minimal can be a design to research for
  the rendering engine and big data performance. This makes the dirtiness of
  textual attributes more efficient according to their necessity - displayed?.

  */
}

void test7g(Viewer &view) {
  // class based linkage for more complex formatters and
  // builders. This class and formatting logic would be in a separate file,
  // here for testing.
  using uniRecord = class uniRecord {
```

```
  enum uniRecordSchema {
    contact,
    filelist,
    pictureThumb,
    ratingVisual,
    videoThumb,
    audioThumb
  };

public:
  size_t index;
  uniRecordSchema schema;
  uniRecord(const string_view &broadName) {}
  uniRecord(const string_view &fname, const string_view &lname,
          const string_view &phone, const string_view &email) {
    schema = contact;
  };
  uniRecord(const string_view &_fileName, const size_t &fileSizeKiloByte,
          const size_t &indexPreviewKey) {
    schema = filelist;
  };
  uniRecord(const string_view &fname, const string_view &format,
          const int &depth, const tuple<double, double, double> gps,
          const size_t summaryIndexKey) {
    schema = pictureThumb;
  };
  uniRecord(const float &stars) { schema = ratingVisual; };

  /*... for each record type */

  auto &build(void) {
    auto o = createElement<li>();

    // build interface based on the schema selected from enumeration
    // as a simple example. so the data input would be a parameter to
    // some kind of specialized formatter, user preferences,
    // who knows - linked to shared memory, or some disk data.
    switch (schema) {
    case contact:
      o.appendChild("contact info");
      break;
    case filelist:
      break;
    case pictureThumb:
      break;
    case ratingVisual:
      break;
    case videoThumb:
      break;
    case audioThumb:
      break;
    }

    // link the element object to this one.
    o.setAttribute(*this);
    return o;
  }
};
/* format detailed views easily ...
now records within the vector template type,
combined with
the std::function creates the view through dynamic visualization.

Because the
record format is usertyped always, only defaulted for std::string vectors
adapted by constructor. ie:

fname, string_view lname, string_view phone,
string_view email
```

```
  */

  view.appendChild<ul>(indexBy{"uniRec"}).data<uniRecord>() = {
      {"anthony", "matarazzo", "(666) 123-4567", "simple@sss.com"},
      {"kevin", "styemark", "222 333 4444", "kevin.styemark@sss.com"},
      {"brenda", "rollerbank", "222 333 4444", "brenda.rollerbank@sss.com"},
      {"cindy", "trustinalo", "222 333 4444", "cindy.trustinalo@sss.com"},
      {"samantha", "skapity", "(123) 457-8906", "samantha.skapity@sss.com"},
      {"Tony", "Mowkar", "(777) 765-4321", "zitapps@sss.com"},
      {"mark", "maglich", "(800) 869-1234", "mark@sss.com"},
      {"cyclemount.jpg", 33234, 0x994834},
      {"greybirds.jpg", 8877394, 0x7564774},
      {"grapefruit.jpg", 47764, 0x95785777},
      {"cantelope.jpg", 997787, 0x645654},
      {"bannana.jpg", 12217767, 0x77667844},
      {"apple.jpg", 2344, 0x676778534},
      {"orange.jpg", 5545, 0x99887},
      {"pineapple.jpg", 1346, 0x454567}};

  auto &ulItems = getElement<ul>("uniRec");

  // transform lambda
  ulItems.dataTransform<li, uniRecord>(
      [](auto &o) -> auto & { return o.build(); });

  vector<uniRecord> vegetables = {{"Broccoli"},        {"Broccoli Rabe "},
                                  {"Brussel Sprouts"}, {"Cabbage, Green"},
                                  {"Cabbage, Red"},    {"Carrot"},
                                  {"Cassava"},         {"Cauliflower"}};

  auto &vecItems = ulItems.data<uniRecord>();
  vecItems.insert(vecItems.end(), vegetables.begin(), vegetables.end());

  // takes info and induces the change.
  ulItems.dataHint<uniRecord>(vegetables.size());

  /*
  then building complex user interfaces is more objected oriented, separated
  from the data, and the data interface is the standard library.
  */
}

/*
These series of tests all provide the easiest most concise
building factory methods however at the cost of more parsed data.
Specifically using the C++ binary string literal interface.
Within the string format, appearing and communicating with
a based initializer list,

View manager object notation is a parsed input function prvoding
the ability to generate very complex dataTransform function for
the standard input. That is, at times lambda's of this sort may be
difficult to implement. Yet the syntax provides a templated mechnism
for building tree and event patterns which format the information within
the data<>.

*/

void test7h(Viewer &view) {

  /* A dataTransform is required when using the direct data interface.
  By default, for simple types input into the model, a default exists.
  However, in most cases, overriding the default format is necessary.
  To compensate in the complexity of building large document fragments,
  the system offers a dataTransform that may be described using a string.
  The form is very simple, integrated within the same type of html and c++
  style. This is the most compact form to use as well, it can achieve high
  performance due to the creation.
  The parameter is given as a string. The significance of the
  first value enclosed within the paranthesis notes what "name" the main input
  type is reffereed to as. This must be a container that supports std::get<#>.
  Ends with a colon. Next is the markup document mapped to the value input
  column (a tuple in this case). There are some inclusion of format
  descriptors in form borrowed from python. However this form uses the
  standard form %s and expands it. A nice time performance about this function
  is that the information is parsed only once. The internal mechnism creates a
  deep cloneabled object that can also reflect vectorized expansion during
  build. Notice the ... expansion for tuple column 4. Formatting and gathering
  of values from the tuple are iterative based upon index. The
  function returned will contain the logic to build the tree from vectors. The
  vectors will also contain the formatter object calls themselves. This will
  make visualization much quicker as creation of the object, tree structure,
  attributes, etc will be purely binary. So, small price - parse
  at transform creation. Yet still using and not interuppting data flow, nor
  decreasing the volumne of data the transform may handle.

  This creates multiple expansion using the data interface. As well,
  advanced formatting.

        ** adding an evaluator to the internal storage can provide
  selection of different templates based on the item. Thereform, multiple
  forms are given as input. The second form of the text template, provides the
  ability to have the system evaluate which "text template" to use based upon
  the boolean return of the given lambda.

     https://docs.python.org/3/library/string.html#string-formatting
     https://docs.python.org/3/library/string.html#formatspec
         - control alignment using standard attributes,
     https://en.wikipedia.org/wiki/Polymer_(library)

  */
  /* a record type such as this, containing dynamic children
     can be described in the form :*/
  using tagInfo = std::tuple<int, std::string, float, std::string,
                  std::vector<std::tuple<std::string, float>>>;

  auto o = view.appendChild<ul>(indexBy{"music"});

  o.dataTransform<li, tagInfo>(

    R"(  (t) :
          <li>{t[0],%0xd}
                  <ul>
                           <li>{t[1],:20}</li>
                           <li>{t[2]}</li>
                           <li>{t[3]}</li>
                           <li><table>
                                                      <{t[4]...v, odd-even} tr>
                                                               <td>{v[0], titlecap}</td>
                                                               <td>{v[1], Currency}</td>
                                                      </tr>
                                             </table></li>
                  </ul>
          </li>

)");

#if 0
  view.appendChild<ul>().data<tagInfo>() = {{0,
             "Anthony",
             134.5,
             "Horror films, science fiction",
             {{"Planet Crack Core", 7.0},
              {"Nachzehrer Fires PG", 43.4},
              {"Alien Spacecraft R+", 97.1},
               {"Jupiter Time Shift R", 64.3}}}};
#endif
```

```
  view.render();
}

/*
        The string form of the transform may be expanded further to increase
        application production speed. decisions, plugin necessity. This is
   merely a considerable technique that can be differentated from the draft
   approach. However as always, the operation of a system at a binary level as
   complete as possible does reduce complexity once inside of an embedded device
   such as a video card. While one may jump to conclusions about the complexity
   of such devices, their operation to make selective slot choices are the very
        component that creates advances as well as facilitates that speed boost.
   To start with the std::unordered_map to be accepted as a function slot choice
        machine, creates a great communication device for c++ programmers.
   However, in this device, nothing is ever copied, merely described in syntax.

        My next techniques for building interfaces lies within the very
        necessity of those concepts. Abstraction of data and format. An
        interesting type of mid and large application layout formula takes place
        during the maintenance stage. Most forms of these presentation factors
   are coded in easily changed components of differnt formatting possibilities
   plus data system logic support. Labels, consildated and easily matched
   through a long list of options. However this can become the bottle neck at
   times. So, the balance at times is considered, yet is specific enough that
   langauge and its operation capacity are questioned for these types of
   products. In effect one may say that these are the concepts of a complete
   system.

        So my next capacity of interface building relies on the strict data
        input interface, however also uses a condensor for functional attributes
        of gui interfaces. So most complex, most used, most pro would like
        something like this following:


*/
void test7i(Viewer &view) {

  /* dynamic or static. dynamic most likely...
  These processors are very short for effeciveness. Typically, many
  other things can occur for these states. This format allows these states
  to exist, be automatically instaniated, maintained on a separate basis,
  reduced, and defaulted. So defaulted must be handled as a case as well.

  And a very robust secret of this type is how c++ programmers can
  abstract these again into other captured data sets for just
  modifing specific summaries based upon data these routines
  may share with internal and exteral communication.

  */
  using processList = std::tuple<int, std::string, float, std::string>;

  view.appendChild<ul>(indexBy{"corp"}).data<processList>() = {
      {0, "favorites", 4.5, "Umbrella Corporation"},
      {1, "less", 7.5, "Sky blink jinks"},
      {2, "grand", 4.5, "No touch, no voice, giant fingers"},
      {3, "flavor", 5.5, "360 flyers"},
      {0, "color", 2.9, "unknown not found error"},
      {2, "magnetism", 8.3, "perhaps but much after quantium decays"},
      {1, "alertness", 3.7, "can monitor laughter pinches"}};

  auto &o = getElement<ul>("corp");

#if 0
  o.dataTransform<0, processList>(
      {{0,
        [](processList &r) {
          return createElement<li>(
              textColor{"red"}, get<1>(r), get<2>(r), get<3>(r));
        }},
       {1,
        [](processList &r) {
          return createElement<li>(
              textColor{"green"}, get<1>(r), get<2>(r), get<3>(r));
        }},
       {2,
        [](processList &r) {
          return createElement<li>(
              textColor{"blue"}, get<1>(r), get<2>(r), get<3>(r));
        }},
       {3, [](processList &r) {
          return createElement<li>(
              textColor{"plum"}, get<1>(r), get<2>(r), get<3>(r));
        }}});
#endif
}

/*********************************************************************
My next thoughtre of a very nice ordered approach to classical
naturalism of information presentation. That is need for styling and
sizing comes about      not only to classify, but to order, and to provide
longevity, weight, and exposure of information into the view. I found a
discussion of color to be quite elegant according to some concepts of web
design standards. Such as that which leave anyone a nomad concepting a design
from mere tid bits of broken data structures within most document object
models. These tasks may be derrived from extensions of the tools used for
normlization of base platform. The concept of the "information system"
is present within most designs. By specifing a type of color schema and
perhaps direct methods for manulipting it by function may be too assumptive in
the approach to how the application can be built. So perhaps further definition
of these typese of traits can be refined with more investigation after
the base framework is assembled.

A style sheet that can apply itself to a document by classifying terms
of information system to that of information structure name within the
document heirarchy. This is typical of how style and the cascade works.
One method is to provide binary labeled build up routines that accept
a reduced parameter set of the explicit information to set. This would
be an extension that also provides base interaction of more advanced gui
elements such that defining elegant operating and interactive style is
accomplished as a utility function. here are numerious examples that
pop to mind, yet I would like to leave them out for later to understand
how this new extension should work. And perhaps in that provide an
efficient and flexible way these methods can be passed along. It is
interesting once tools are available how others use them. I have seen some
very pleaseing web designs, and therefore the ability to apply that content
and layout apprach is important. XML and XSL are some solutions which compete
and apply this in a grand parsed manner.

    a. naming for these forms are more strict and therefore more
       natural and pleasing
            1. the names are more specific
            2. information at times may have to be defaulted because it is
large set a. this information such as documentation, corporate id, etc. b.
personel c. software promoting license d. etc.
            3. some may required storage resources and
               processes that run in the background
                        a. all processes should be promoted
                        b. storage requirements and reasons

          I see this direction as an after approach to refining the
library for specific types of uses.

--- a type of style guide for std integration


*********************************************************************/
```

```
//! [test8a]
void test8a(Viewer &vm) {
  testStart(__func__);

  auto &dBook = vm.appendChild<DIV>(indexBy{"booklet5"});

  PARAGRAPH *pBooklet = nullptr;

  // a warning is issued which is what is required.
  // main.cpp:XXXXX: warning: format string is not a string
  // literal
  // [-Wformat-nonliteral]
  string s = "not literal";
  dBook.printf(s.c_str());

  dBook.printf("<div id=BookletParagraph>");
  dBook.printf("The paragraph content is here.");
  dBook.printf("<img id=imgData/>");
  dBook.printf("<ul id=chapterList>");

  for (int i = 0; i < 10; i++)
    dBook.printf("<li>Chapter List %i</li>", i);

  dBook.printf("</ul>");

  dBook.printf("</div>");

  vm.render();
}
//! [test8a]

/********************************************************************



********************************************************************/
//! [test8]
void test8(Viewer &vm) {
  testStart(__func__);

  int m = randomInt(5);

  auto &dBook = vm.appendChild<DIV>(indexBy{"booklet5"});
  vm.render();

  // std::reference_wrapper<PARAGRAPH> Booklet;

  for (int i = 0; i < m; i++) {
    // this line is like putting :
    // pBooklet=
    // in textual tag
    // dBook.printf("%s", referenceTag(Booklet));
    dBook.printf("<p id=BookletParagraph_%i>", i);
    dBook.printf("The paragraph content is here.");
    dBook.printf("  <ul id=notes_%i>%s", i, "text content ul");
    dBook.printf("      <ul id=guestSpeaker_%i>%s</ul>", i,
                 "<blue>guest speaker information <black>... ");
    dBook.printf("      <ul id=references_%i>%s</ul>", i,
                 " <green>reference information: <black>... ");
    dBook.printf("  </ul>");
    dBook.printf("</p>");

    // Booklet.setAttribute(textColor{"red"});
  }

  vm.render();
}
//! [test8]
//! [test10]
void test10(Viewer &vm) {
  testStart(__func__);

  int m = randomInt(5);

  auto dBook = vm.appendChild<DIV>(indexBy{"booklet5"});
  vm.render();

  PARAGRAPH *pBooklet = nullptr;

  for (int i = 0; i < m; i++) {
    // this line is like putting :
    // pBooklet=
    // in textual tag
    dBook << stringPointerTag(pBooklet) << "<p id=BookletParagraph_" << i << ">"
          << "The paragraph content is here."
          << "  <ul id=notes_" << i << ">text content ul"
          << "<ul id=guestSpeaker_" << i << ">"
          << "<blue>guest speaker information <black>... "
          << "</ul>"
          << "<ul id=references_" << i << ">"
          << "<green>reference information: <black>... "
          << "</ul>"
          << "</ul>"
          << "</p>";

    pBooklet->setAttribute(textColor{"red"});
  }

  vm.render();
}
//! [test10]

/*************************************************************************

*************************************************************************/
string_view randomString(int nChars) {
  unsigned int numChars =
      1u +
      (std::rand() / ((RAND_MAX + 1u) / static_cast<unsigned int>(nChars)));

  auto randchar = []() -> char {
    const char charset[] = "0123456789"
                           "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
                           "abcdefghijklmnopqrstuvwxyz";
    const size_t max_index = (sizeof(charset) - 1);
    return charset[rand() % max_index];
  };
  string s1(numChars, 0);
  ;
  string_view str(s1);
  //  std::generate_n(str.begin(), numChars, randchar);

  return str;
}

/*************************************************************************

*************************************************************************/
double randomDouble(double a, double b) {
  double ret =
      a + 1 +
      ((unsigned int)std::rand() / ((RAND_MAX + 1u) / static_cast<int>(b - a)));
  return ret;
}

/*************************************************************************
```

```
*************************************************************************/
int randomInt(int a) {
  int ret =
      1 + ((unsigned int)std::rand() / ((RAND_MAX + 1u) / static_cast<int>(a)));
  return ret;
}

/*************************************************************************

*************************************************************************/
void randomAttributeSettings(Element &e) {
  e.getAttribute<textIndent>().value = randomDouble(0.0, 10.5);

  e.getAttribute<objectTop>().value = randomDouble(0.0, 300);
  e.setAttribute(objectTop{randomDouble(0.0, 300), numericFormat::percent});

  e.getAttribute<objectLeft>().value = randomDouble(0.0, 300);
  e.setAttribute(objectLeft{randomDouble(0.0, 300), numericFormat::percent});

  e.getAttribute<objectWidth>().value = randomDouble(0.0, 300);
  e.setAttribute(objectWidth{randomDouble(0.0, 300), numericFormat::percent});

  e.getAttribute<objectHeight>().value = randomDouble(0.0, 300);
  e.setAttribute(objectHeight{randomDouble(0.0, 300), numericFormat::percent});

  e.getAttribute<textFace>().value = randomString(10);
  e.getAttribute<textWeight>().value = randomDouble(0.0, 1000.0);
  e.getAttribute<tabSize>().value = randomDouble(0.0, 10.0);
  e.getAttribute<focusIndex>().value = randomDouble(0.0, 1000.0);
  e.getAttribute<lineHeight>().value = randomDouble(0.0, 20.0);
  e.getAttribute<paddingTop>().value = randomDouble(0.0, 20.0);
  e.getAttribute<paddingBottom>().value = randomDouble(0.0, 20.0);
  e.getAttribute<paddingLeft>().value = randomDouble(0.0, 20.0);
  e.getAttribute<paddingRight>().value = randomDouble(0.0, 20.0);
  e.getAttribute<marginTop>().value = randomDouble(0.0, 50.0);
  e.getAttribute<marginBottom>().value = randomDouble(0.0, 50.0);
  e.getAttribute<marginLeft>().value = randomDouble(0.0, 50.0);
  e.getAttribute<marginRight>().value = randomDouble(0.0, 50.0);
  e.getAttribute<borderWidth>().value = randomDouble(0.0, 50.0);

  e.getAttribute<borderRadius>().value = randomDouble(0.0, 10.5);
}
```

viewManager.hpp

```
////  <file>viewManager.hpp</file>
///  <author>Anthony Matarazzo</author>
///  <date>1/13/19</date>
///  <version>1.0</version<
///
///  <summary>Cross platfrom GUI DOM model with c++ 17 templates. </summary>

#ifndef VIEW_MANAGER_HPP_INCLUDED
#define VIEW_MANAGER_HPP_INCLUDED

#include <fstream>
#include <iomanip>
#include <iostream>

#include <algorithm>
#include <any>
#include <array>
```

```
#include <future>
#include <iterator>
#include <map>
#include <optional>
#include <regex>
#include <sstream>
#include <string>
#include <string_view>
#include <tuple>
#include <unordered_map>
#include <variant>
#include <vector>

#include <cctype>
#include <climits>
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <cstring>

#include <charconv>
#include <functional>
#include <memory>
#include <type_traits>
#include <typeindex>
#include <typeinfo>
#include <utility>

/**************************************
                         OS SPECIFIC HEADERS
***************************************/
#if defined(__linux__)
#include <X11/keysymdef.h>
#include <xcb/xcb.h>
#include <xcb/xcb_keysyms.h>

#elif defined(_WIN64)

// Windows Header Files:
#include <d2d1.h>
#include <d2d1helper.h>
#include <wincodec.h>
#include <windows.h>

#ifndef HINST_THISCOMPONENT
EXTERN_C IMAGE_DOS_HEADER __ImageBase;
#define HINST_THISCOMPONENT ((HINSTANCE)&__ImageBase)
#endif

#endif
namespace ViewManager {

// forward declaration
class Element;
class StyleClass;
class event;

// interface specific communication types
typedef std::function<void(const event &)> eventHandler;
typedef std::function<bool(const Element &)> ElementQuery;

typedef std::vector<std::reference_wrapper<Element>> ElementList;

using tableData = std::vector<std::vector<std::string_view>>;

template <std::size_t I, typename T>
using dataTransformMap =
    std::unordered_map<const typename std::tuple_element<I, T>::type &,
                       std::function<Element &(T &)>>;
```

```
/**
These namespace global lists contain the objects as a system ownership.
**/
extern std::vector<std::unique_ptr<Element>> elements;
extern std::unordered_map<std::string_view, std::reference_wrapper<Element>>
    indexedElements;
extern std::vector<std::unique_ptr<StyleClass>> styles;

/*******************************************************
                                  Events
*******************************************************/
enum class eventType : uint8_t {
  paint,
  focus,
  blur,
  resize,

  keydown,
  keyup,
  keypress,

  mouseenter,
  mousemove,
  mousedown,
  mouseup,
  click,
  dblclick,
  contextmenu,
  wheel,
  mouseleave
};

using event = class event {
public:
  event(const eventType &et) {
    evtType = et;
    bUnicodeEvent = false;
  }

  event(const eventType &et, const char &k) {
    evtType = et;
    key = k;
    bUnicodeEvent = false;
  }

  event(const eventType &et, const std::wstring &uniK) {
    evtType = et;
    unicodeKeys = uniK;
    bUnicodeEvent = true;
  }

  event(const eventType &et, const short &mx, const short &my,
        const short &mb_dis) {
    evtType = et;
    mousex = mx;
    mousey = my;

    if (et == eventType::wheel)
      wheelDistance = mb_dis;
    else
      mouseButton = mb_dis;
    bUnicodeEvent = false;
  }
  event(const eventType &et, const short &w, const short &h) {
    evtType = et;
    width = w;
    height = h;
    bUnicodeEvent = false;
  }
  event(const eventType &et, const short &distance) {
    evtType = et;
    wheelDistance = distance;
    bUnicodeEvent = false;
  }
  ~event(){};

public:
  eventType evtType;
  bool bUnicodeEvent;

  union {
    struct {
      ///  <summary>contains the character of the key pressed.</summary>
      char key;
    };

    struct {
      std::wstring unicodeKeys;
    };

    ///  <summary>contains the values of the mouse actions.</summary>
    struct {
      short mousex;
      short mousey;
      char mouseButton;
    };
    struct {
      short width;
      short height;
    };
    struct {
      ///  <summary>contains the distance the wheel traveled.</summary>
      short wheelDistance;
    };
  };
};

/* communication enum options.
these are provided as public, and used as a type
on the input of the class constructor.*/

using numericFormat = enum option { px, pt, em, percent, autoCalculate };
using colorFormat = enum colorFormat { rgb, hsl, name };

/*
Classes used by complex attributes.
*/
class doubleNF {
public:
  double value;
  numericFormat option;
  doubleNF(const doubleNF &_val) : value(_val.value), option(_val.option) {}
  doubleNF(const double &_val, const numericFormat &_nf)
      : value(_val), option(_nf) {}
};

class colorNF {
public:
  std::array<double, 4> value;
  colorFormat option;
  colorNF(const colorFormat &_opt, const std::array<double, 4> &_val)
      : option(_opt), value(_val) {}
  colorNF(const std::string_view &_colorName) { option = colorFormat::name; }
  void lighter(const double &step = 0.1) {}
  void darker(const double &step = 0.1) {}
  void monochromatic(const double &step = 0.1) {}
  void triad(void) { /*hsl rotate 120*/
  }
  void neutralCooler(void) { /* hsl rotate -30 */
  }
  void neutralWarmer(void) { /* hsl rotate 30 */
  }
  void complementary(void) { /* hsl rotate 180*/
  }
  void splitComplements(void) { /*hsl rotate 150 */
  }
};

/*
        INTERNEL MACROS to reduce code needed for these attribute storage
        implementations. These macros develop the storage class or
   inheiritance to implement a complex attribute. Within the system, the
   "using" provides an alias to name a specific type. Therefore, all classes
   declared using these macros will establish their storage space securely at
   its position within the attribute storage container. When a getAttribute
   with a specific attribute is requested, the reference to this class is
   returned. Therefore, values and such can be read or edited using only the
   memory. getAttribute is templated so that the most modified values attain
   their own specific function. Therefore, newer or specific interfaces can be
   developed.

*/

#define _NUMERIC_ATTRIBUTE(NAME)                                              \
  using NAME = class NAME {                                                   \
  public:                                                                     \
    double value;                                                             \
    NAME(const double &_val) : value(_val) {}                                 \
  }

#define _STRING_ATTRIBUTE(NAME)                                               \
  using NAME = class NAME {                                                   \
  public:                                                                     \
    std::string_view value;                                                   \
    NAME(const std::string_view &_val) : value(_val) {}                       \
    NAME(const NAME &_val) : value(_val.value) {}                             \
  }

#define _NUMERIC_WITH_FORMAT_ATTRIBUTE(NAME)                                  \
  using NAME = class NAME : public doubleNF {                                 \
  public:                                                                     \
    NAME(const double &_val, const numericFormat &_nf)                        \
        : doubleNF(_val, _nf) {}                                              \
    NAME(const doubleNF &_val) : doubleNF(_val) {}                            \
  }

#define _ENUMERATED_ATTRIBUTE(NAME, ...)                                      \
  using NAME = class NAME {                                                   \
  public:                                                                     \
    enum optionEnum : uint8_t { __VA_ARGS__ };                                \
    optionEnum value;                                                         \
                                                                              \
  public:                                                                     \
    NAME(const optionEnum &val) : value(val) {}                               \
  }

#define _NUMERIC_WITH_ENUMERATED_ATTRIBUTE(NAME, ...)                         \
  using NAME = class NAME {                                                   \
  public:                                                                     \
    enum optionEnum : uint8_t { __VA_ARGS__ };                                \
    double value;                                                             \
    optionEnum option;                                                        \
    NAME(const double &_val, const optionEnum &_opt)                          \
        : value(_val), option(_opt) {}                                        \
    NAME(NAME &&) = default;                                                  \
  }
```

```
#define _COLOR_ATTRIBUTE(NAME)                                              \
  using NAME = class NAME : public colorNF {                                \
  public:                                                                   \
    NAME(const double &_v1, const double &_v2, const double &_v3)           \
        : colorNF(colorFormat::rgb, {_v1, _v2, _v3, 0}) {}                  \
    NAME(const std::string_view &_colorName) : colorNF(_colorName) {}       \
    NAME(const colorFormat &_opt, const std::array<double, 4> &_val)        \
        : colorNF(_opt, _val) {}                                            \
    NAME(const colorFormat _opt, const double &_v1, const double &_v2,      \
         const double &_v3)                                                 \
        : colorNF(_opt, {_v1, _v2, _v3, 0}) {}                              \
    NAME(const colorFormat &_opt, const double &_v1, const double &_v2,     \
         const double &_v3, const double &_v4)
                              \
        : colorNF(_opt, {_v1, _v2, _v3, _v4}) {}                            \
  }

#define _VECTOR_ATTRIBUTE(NAME)                                             \
  using NAME = class NAME {                                                 \
  public:                                                                   \
    std::vector<std::string_view> value;                                    \
    NAME(std::vector<std::string_view> _val) : value(std::move(_val)) {}    \
    NAME(NAME &&) = default;                                                \
  }

// namespace ViewManager

/*
Attributes are classes with registered name aliases. The underlying
architecture reduces the use of templates but promotes the passing of data.
The attributes are contained within the internel structure, within the
Element base class. The attribute storage is indexed and stored within an
unordered map. References are returned so that callee communication is
minimal.
*/

_STRING_ATTRIBUTE(indexBy);
_ENUMERATED_ATTRIBUTE(display, in_line, block, none);
_ENUMERATED_ATTRIBUTE(position, absolute, relative);

_NUMERIC_WITH_FORMAT_ATTRIBUTE(objectTop);
_NUMERIC_WITH_FORMAT_ATTRIBUTE(objectLeft);
_NUMERIC_WITH_FORMAT_ATTRIBUTE(objectHeight);
_NUMERIC_WITH_FORMAT_ATTRIBUTE(objectWidth);

_COLOR_ATTRIBUTE(background);
_NUMERIC_ATTRIBUTE(opacity);

_STRING_ATTRIBUTE(textFace);
_NUMERIC_WITH_FORMAT_ATTRIBUTE(textSize);
_NUMERIC_ATTRIBUTE(textWeight);
_COLOR_ATTRIBUTE(textColor);
_ENUMERATED_ATTRIBUTE(textAlignment, left, center, right, justified);

_NUMERIC_WITH_FORMAT_ATTRIBUTE(textIndent);
_NUMERIC_WITH_FORMAT_ATTRIBUTE(tabSize);
_NUMERIC_WITH_ENUMERATED_ATTRIBUTE(lineHeight, normal, numeric);

_NUMERIC_WITH_FORMAT_ATTRIBUTE(marginTop);
_NUMERIC_WITH_FORMAT_ATTRIBUTE(marginLeft);
_NUMERIC_WITH_FORMAT_ATTRIBUTE(marginBottom);
_NUMERIC_WITH_FORMAT_ATTRIBUTE(marginRight);

_NUMERIC_WITH_FORMAT_ATTRIBUTE(paddingTop);
_NUMERIC_WITH_FORMAT_ATTRIBUTE(paddingLeft);
_NUMERIC_WITH_FORMAT_ATTRIBUTE(paddingBottom);
_NUMERIC_WITH_FORMAT_ATTRIBUTE(paddingRight);
```

```
_NUMERIC_WITH_ENUMERATED_ATTRIBUTE(borderStyle, none, dotted, dashed, solid,
                                   doubled, groove, ridge, inset, outset);

_NUMERIC_WITH_FORMAT_ATTRIBUTE(borderWidth);
_COLOR_ATTRIBUTE(borderColor);
_NUMERIC_ATTRIBUTE(borderRadius);

_NUMERIC_ATTRIBUTE(focusIndex);
_NUMERIC_ATTRIBUTE(zIndex);

_ENUMERATED_ATTRIBUTE(listStyleType, none, disc, circle, square, decimal, alpha,
                      greek, latin, roman);

/*******************************************************
                                          StyleClass
*******************************************************/
class StyleClass {
public:
  std::unordered_map<std::type_index, std::any> attributes;
  StyleClass *self;

public:
  template <typename... Args> StyleClass(const Args &... args) : self(this) {
    //((void)setValue(std::forward<Args>(args)), ...);
  }

  template <typename T> void setValue(const T &attr) {
    std::type_index ti = std::type_index(typeid(attr));
    attributes[ti] = attr;
  }
};

/*******************************************************
                                          Visualizer
********************************************************/

namespace Visualizer {
typedef struct {
  double t;
  double l;
  double w;
  double h;
} rectangle;
extern std::vector<rectangle> items;

std::size_t allocate(Element &e);

void deallocate(const std::size_t &token);

void openWindow(Element &e);
void closeWindow(Element &e);

/******************************
                            Platform object
*******************************/
class platform {
public:
  platform(eventHandler evtDispatcher, unsigned short width,
           unsigned short height);
  ~platform();

  void messageLoop(void);
  void flip(void);
#if defined(_WIN64)
  LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam,
                           LPARAM lParam);
#endif

public:
```

```
#if defined(__linux__)
  xcb_connection_t *m_connection;
  xcb_screen_t *m_screen;
  xcb_drawable_t m_window;
  xcb_pixmap_t m_offScreen;

  // xcb -- keyboard
  xcb_key_symbols_t *m_syms;

#elif defined(_WIN64)
  HWND m_hwnd;
  ID2D1Factory *m_pDirect2dFactory;
  ID2D1HwndRenderTarget *m_pRenderTarget;
  ID2D1BitmapRenderTarget *m_pOffscreen;
  ID2D1Bitmap *m_offScreenBitmap;

#endif

private:
  eventHandler dispatchEvent;
}; // class platform

}; // namespace Visualizer

template <typename TYPE>
TYPE &_createElement(const std::vector<std::any> &attr);

/*******************************************************
                                                                Element
*******************************************************/
class Element {
public:
public:
  Element(const std::vector<std::any> &attribs = {})
      : m_self(this), m_parent(nullptr), m_firstChild(nullptr),
        m_lastChild(nullptr), m_nextChild(nullptr), m_previousChild(nullptr),
        m_nextSibling(nullptr), m_previousSibling(nullptr), m_childCount(0) {

    setAttribute(attribs);
  }
  ~Element() { Visualizer::deallocate(surface); }
  Element(const Element &other) {
    m_self = other.m_self;
    m_parent = other.m_parent;
    m_firstChild = other.m_firstChild;
    m_lastChild = other.m_lastChild;
    m_nextChild = other.m_nextChild;
    m_previousChild = other.m_previousChild;
    m_nextSibling = other.m_nextSibling;
    m_previousSibling = other.m_previousSibling;
    m_childCount = other.m_childCount;
    attributes = other.attributes;
    styles = other.styles;
  }
  Element(Element &&other) noexcept {
    m_self = other.m_self;
    m_parent = other.m_parent;
    m_firstChild = other.m_firstChild;
    m_lastChild = other.m_lastChild;
    m_nextChild = other.m_nextChild;
    m_previousChild = other.m_previousChild;
    m_nextSibling = other.m_nextSibling;
    m_previousSibling = other.m_previousSibling;
    m_childCount = other.m_childCount;
    attributes = std::move(other.attributes);
    styles = std::move(other.styles);
  }

  Element &operator=(const Element &other) {
```

```
  // Self-assignment detection
  if (&other == this)
    return *this;
  m_self = other.m_self;
  m_parent = other.m_parent;
  m_firstChild = other.m_firstChild;
  m_lastChild = other.m_lastChild;
  m_nextChild = other.m_nextChild;
  m_previousChild = other.m_previousChild;
  m_nextSibling = other.m_nextSibling;
  m_previousSibling = other.m_previousSibling;
  m_childCount = other.m_childCount;
  attributes = other.attributes;
  styles = other.styles;
  return *this;
}
Element &operator=(Element &&other) noexcept {
  if (&other == this)
    return *this;
  m_self = other.m_self;
  m_parent = other.m_parent;
  m_firstChild = other.m_firstChild;
  m_lastChild = other.m_lastChild;
  m_nextChild = other.m_nextChild;
  m_previousChild = other.m_previousChild;
  m_nextSibling = other.m_nextSibling;
  m_previousSibling = other.m_previousSibling;
  m_childCount = other.m_childCount;
  attributes = std::move(other.attributes);
  styles = std::move(other.styles);
  return *this;
}
// move assignment

/// <summary> overload of the stream insertion operator.
/// Simply puts the data into the stream. It should be
/// noted that flush should be called.
/// </summary>
template <typename T> Element &operator<<(const T &data) {
  std::stringstream s;
  // s << data;
  return *this;
}

/*
Data interface.
The data<> templated function provides a standard interface for data
 projection and building for ease of interface communication. That is,
 at most times, the default operation for these implementation details
 will be provided as a devoted comprehension of the element's
 purpose. Data input into the system will be owned by the
 internal storage. This is a typical case use for
 these types of objects as visual presentors. However, a refererence to
 the std::vector<T> & can be attained by using the data function. Adding
       or modify records of this attained reference is direct to c++ standard
       library.


The usageAdaptorState structure holds information about what is in the
possibly large amount of unknown data. It holds also the information that
is not clipped from view, a memory range, and an index range. From this
standpoint, it should be kept current periodiocally and or logically.
      It can be updated by the hinting system since the system knows about
      the addresses of the data. In proper structure, vectors offer a range
      based approach to hinting as each element is stored contiguiogusly in
memory. Therefore, any numerical address given within a hint can reveal
several aspects about an element. For example, inclusion within the set as a
vector element simply by using a numerical less than and also greater than.
```

```
  Once set inclusion is established, further investigation about the quality
  of the hint information and other numerical parameters can be used to find
  if that particular is reflected currently within the
  document object model proper. That is most of these items may be visited by
  the renderer. When the element is not within dom visibility, its
  textual visual surface area allocated for pixel data is deallocated.
  Other aspects such as graphic objects, audio, video, medium may be
  handled externally for better cache management according to principles
  of modern hardware operation.

  Relocation and geometric allocation           provided within c++ std::vector
  will provide also effective use of this as applications operate
  within well planned c++ memory subsystems. It appears as if the vector
  memory reserve grows logically according to necessity of only demand.
  The worst case scenerio happens rather
 quickly where visible items
  are stripped of their pointer reference data.

  Always keeping in mind of the large data possibility, keeping the
  dom cache tight around the visible contained area will provide
  quicker recovery when the worst case happens. And after the
  vector memory has grown geometrically because of all the data movement,
  data movement not impeded at all by this design, it is assumed to be put off
  for larger amounts of time. New geometric changes occuring will
  be recorded in the _lastWorkLoad to indicate locations or other information
  needed to make the new remapping just a numerical range insertion for
  the visible element's data and gui elements. Index information to the
  element's address within the data adaptor is saved and used to compute their
  new data pointers or - reference wrappers-good stuff.

  */
private:
  typedef struct _usageAdaptorState {
    std::size_t _size;
    std::size_t _visibleBegin;
    std::size_t _visibleEnd;
    std::size_t _lastWorkLoad;
  } usageAdaptorState;

  /*
        usageAdaptor<> - INTERNAL a templated class used only internally,
  holds the data of a templated type within a vector.
  Noted that vectors have sequential memory for all of their elements.
  This property is used by the hinting system to deduce set inclusion.
  They can be established just by the type leaveing
  data and fnTransform empty.

  When just the data is passed, and a default transform function
  does not exist, the data is saved leaving the fnTransform as a default
  formatter according to the type.

  */
private:
  template <typename T> class usageAdaptor {
  public:
    usageAdaptor(void) {}
    usageAdaptor(std::vector<T> &_d) : _data(_d) { saveState(); }
    usageAdaptor(std::function<Element &(T &)> &_fn) : fnTransform(_fn) {
      saveState();
    }
    usageAdaptor(std::vector<T> _d, std::function<Element &(T &)> _fn)
        : _data(_d), fnTransform(_fn) {
      saveState();
    }

    auto data(void) -> std::vector<T> & {
      saveState();
      return _data;
    }
```

```
   auto data(std::vector<T> &_input) -> std::vector<T> & {
     _data = std::move(_input);
     saveState();
     return _data;
   }
   std::function<Element &(T &)> &transform(void) { return fnTransform; }
   // analyze hint data and deduce states
   void hint(void *hint1, std::size_t hint2, std::size_t hint3) {}

  private:
   std::vector<T> _data;
   std::function<Element &(T &)> fnTransform;
   usageAdaptorState state;

  private:
   void saveState(void) { state._size = _data.size(); }
  };

  /* three templated functions provide the data interface,
       dataTransform<>(), data<>(), and dataHint<>(). All of these
       templated functions work with the internal data storage mechnism.
  */
public:
  template <typename R, typename T>
  void dataTransform(const std::string_view &txtFn) {}

  template <typename R, typename T>
  void dataTransform(const std::string_view &txtFn,
                  const std::function<bool(T &)> &_fn) {}

  template <typename R, typename T>
  void dataTransform(const std::function<R &(T &)> &_fn) {
   std::function<R &(T &)> fn = _fn;
   std::type_index tIndex = std::type_index(typeid(std::vector<T>));

   auto it = m_usageAdaptorMap.find(tIndex);

   // if the requested data adaptor does not exist,
   // create its position within the adaptor member vector
   // return this to the caller.
   if (it == m_usageAdaptorMap.end()) {

     // create a default data display for the type here.
     std::function<Element &(T &)> fnDefault;
     m_usageAdaptorMap[tIndex] = usageAdaptor<T>(fnDefault);

   } else {
     const usageAdaptor<T> &adaptor =
       std::any_cast<usageAdaptor<T> &>(m_usageAdaptorMap[tIndex]);
   }
  }

  /*
  This data transform uses the named value of the numerical column.
  While being a constant, this reliance is that of an itm within a tuple
  column. This tuple column is used as a index into the natural long list
  of vector data. AS well, the parameter essential is an unordered map
  reference. Simply that the match index as the key and a std::function
  as the found item. THe item is used to process a requested data element
  based upon this match. So, in effect this allows the unordered may to change,
  over time, and selectively modify traites or visual composition in
  a strict binary fashion rather than css conteplated sheets. While both are
  often necessary, this type of solid work code really becomes the marking
  of complexity and specific layouts that may be dynamically loaded.
  And also provide effective space for description. I find that simple
  and complex may go together, yet for usage one must create the sceneros
  that elvolve in delicate use of competent technology.

  */
  template <std::size_t I, typename T>
  void dataTransform(
     const std::unordered_map<const typename std::tuple_element<I, T>::type &,
                              std::function<Element &(T &)>> &_transformList) {
  }

  /************** data access ***************/
  template <typename T = std::string_view> auto &data(void) {
   std::type_index tIndex = std::type_index(typeid(std::vector<T>));

   // if the requested data adaptor does not exist,
   // create its position within the adaptor member vector
   // return this to the caller.
   auto it = m_usageAdaptorMap.find(tIndex);
   if (it == m_usageAdaptorMap.end()) {
     // create a default data display for the type here.
     std::function<Element &(T &)> fnDefault;
     m_usageAdaptorMap[tIndex] = usageAdaptor<T>(fnDefault);
     return std::any_cast<usageAdaptor<T>>(m_usageAdaptorMap[tIndex]).data();
   } else {
     return std::any_cast<usageAdaptor<T> &>(it->second).data();
   }
  }

  template <typename T>
  void dataHint(const T &hint1, std::size_t hint2 = 0, std::size_t hint3 = 0) {}

  template <typename T>
  void dataHint(int hint1 = 0, std::size_t hint2 = 0, std::size_t hint3 = 0) {
   auto it = m_usageAdaptorMap.find(std::type_index(typeid(std::vector<T>)));

   // save input signal ? valid ?
   // check saved state from getAdaptor.

   if (it != m_usageAdaptorMap.end()) {
     const usageAdaptor<T> &adaptor =
       std::any_cast<const usageAdaptor<T> &>(it->second);

     // adaptor.hint(hint1, hint2, hint3);
   }
  }

  /* query at this level must allow traversal of types, and direction.
  siblings, parent, children...
  */
  auto query(const std::string_view &queryString) -> ElementList{};
  auto query(const ElementQuery &queryFunction) -> ElementList{};

  // private data menbers to hold the information of the class
private:
  Element *m_self;
  Element *m_parent;
  Element *m_firstChild;
  Element *m_lastChild;
  Element *m_nextChild;
  Element *m_previousChild;
  Element *m_nextSibling;
  Element *m_previousSibling;
  std::size_t m_childCount;

  // interface access ponits for the interface
public:
#define _REF_INTERFACE(NAME, xNAME)                                            \
  std::optional<std::reference_wrapper<Element>> NAME(void) {                  \
   return (xNAME ? std::optional<std::reference_wrapper<Element>>{*xNAME}     \
                 : std::nullopt);                                             \
  }
  _REF_INTERFACE(parent, m_parent);
  _REF_INTERFACE(firstChild, m_firstChild);
  _REF_INTERFACE(lastChild, m_lastChild);
  _REF_INTERFACE(nextChild, m_nextChild);
  _REF_INTERFACE(previousChild, m_previousChild);
  _REF_INTERFACE(nextSibling, m_nextChild);
  _REF_INTERFACE(previousSibling, m_previousSibling);

  inline std::size_t &childCount(void) { return m_childCount; }

  std::vector<std::reference_wrapper<StyleClass>> styles;

private:
  std::unordered_map<std::type_index, std::any> attributes;
  std::unordered_map<std::type_index, std::any> m_usageAdaptorMap;
  std::size_t surface;

public:
  auto appendChild(const std::string_view &sMarkup) -> Element & {
    return ingestMarkup(*this, sMarkup);
  }
  auto appendChild(Element &newChild) -> Element & {
    newChild.m_parent = this;
    newChild.m_previousSibling = m_lastChild;
    if (m_lastChild)
      m_lastChild->m_nextSibling = newChild.m_self;
    m_lastChild = newChild.m_self;
    m_childCount++;
    return newChild;
  }
  auto appendChild(const ElementList &elementCollection) -> Element & {
    for (Element &e : elementCollection)
      appendChild(e);
    return *this;
  }

  template <typename TYPE, typename... ATTRS>
  auto appendChild(const ATTRS &... attrs) -> Element & {

#if 0
    std::vector<std::any> attrvector{{attrs...}};
    TYPE &e = _createElement<TYPE>(attrvector);
    appendChild(e);
    return e;
#endif
  }

public:
  auto append(const std::string_view &sMarkup) -> Element & {
    Element *base = this->m_parent;
    if (base == nullptr)
      base = this;

    return ingestMarkup(*base, sMarkup);
  }
  auto append(Element &sibling) -> Element & {
    m_nextSibling = sibling.m_self;
    sibling.m_parent = this->m_parent;
    sibling.m_previousSibling = this;
    this->m_parent->m_childCount++;
    return sibling;
  }
  template <typename TYPE, typename... ATTRS>
  auto append(const ATTRS &... attrs) -> Element & {

#if 0
    std::vector<std::any> attrvector{{attrs...}};
    TYPE &e = _createElement<TYPE>(attrvector);
    append(e);
    return e;
```

```
#endif
  }
  auto append(const ElementList &elementCollection) -> Element & {
    for (Element &e : elementCollection)
      append(e);
    return *this;
  }

public:
  /* setAttribute

          The set attribute accepts an std::any, typesafe and convertable.
          There are several predetermined types that are recoginized through
          the type stored with the std::any parameter.

          The filter allows compact input for vector and simple c++ based
     types direction from the creation functions (createElement<>, append<>, and
          appendChild<> which there are
 typically few              entries. */
  Element &setAttribute(const std::any &setting) {
    // filter list
    enum _enumTypeFilter {
      dt_char,
      dt_double,
      dt_float,
      dt_int,
      dt_std_string_view,

      dt_vector_char,
      dt_vector_double,
      dt_vector_float,
      dt_vector_int,
      dt_vector_string_view,

      dt_vector_vector_string_view,
      dt_vector_pair_int_string_view,
      dt_indexBy,
      dt_nonFiltered
    };

    // filter map
    static std::unordered_map<size_t, _enumTypeFilter> _umapTypeFilter = {
        {std::type_index(typeid(char)).hash_code(), dt_char},
        {std::type_index(typeid(double)).hash_code(), dt_double},
        {std::type_index(typeid(float)).hash_code(), dt_float},
        {std::type_index(typeid(int)).hash_code(), dt_int},
        {std::type_index(typeid(std::string_view)).hash_code(),
         dt_std_string_view},
        {std::type_index(typeid(std::vector<char>)).hash_code(),
         dt_vector_char},
        {std::type_index(typeid(std::vector<double>)).hash_code(),
         dt_vector_double},
        {std::type_index(typeid(std::vector<float>)).hash_code(),
         dt_vector_float},
        {std::type_index(typeid(std::vector<int>)).hash_code(), dt_vector_int},
        {std::type_index(typeid(std::vector<std::string_view>)).hash_code(),
         dt_vector_string_view},
        {std::type_index(typeid(std::vector<std::vector<std::string_view>>))
             .hash_code(),
         dt_vector_vector_string_view},
        {std::type_index(
             typeid(std::vector<std::vector<std::pair<int, std::string_view>>>))
             .hash_code(),
         dt_vector_pair_int_string_view},
        {std::type_index(typeid(indexBy)).hash_code(), dt_indexBy}};

    // set search result defaults for not found in filter
    _enumTypeFilter dtFilter = dt_nonFiltered;
    bool bSaveInMap = false;
```

```
std::unordered_map<size_t, _enumTypeFilter>::iterator it =
    _umapTypeFilter.find(setting.type().hash_code());
if (it != _umapTypeFilter.end())
  dtFilter = it->second;

/* filter these types specifically and do not store them in the map.
 these items change the dataAdaptor. This creates a more usable
 syntax for population of large and small data within the
 simple initializer list format given within the attribute list.*/
switch (dtFilter) {
case dt_char: {
  auto v = std::any_cast<char>(setting);
  data<char>() = std::vector<char>{v};
} break;
case dt_double: {
  auto v = std::any_cast<double>(setting);
  data<double>() = std::vector<double>{v};
} break;
case dt_float: {
  auto v = std::any_cast<float>(setting);
  data<float>() = std::vector<float>{v};
} break;
case dt_int: {
  auto v = std::any_cast<int>(setting);
  data<int>() = std::vector<int>{v};
} break;
case dt_std_string_view: {
  auto v = std::any_cast<std::string_view>(setting);
  data<std::string_view>() = std::vector<std::string_view>{v};
} break;
case dt_vector_char: {
  auto v = std::any_cast<std::vector<char>>(setting);
  data<char>() = v;
} break;
case dt_vector_double: {
  auto v = std::any_cast<std::vector<double>>(setting);
  data<double>() = v;
} break;
case dt_vector_float: {
  auto v = std::any_cast<std::vector<float>>(setting);
  data<float>() = v;
} break;
case dt_vector_int: {
  auto v = std::any_cast<std::vector<int>>(setting);
  data<int>() = v;
} break;
case dt_vector_string_view: {
  auto v = std::any_cast<std::vector<std::string_view>>(setting);
  data<std::string_view>() = v;
} break;
case dt_vector_vector_string_view: {
  auto v =
      std::any_cast<std::vector<std::vector<std::string_view>>>(setting);
  data<std::vector<std::string_view>>() = v;
} break;
case dt_vector_pair_int_string_view: {
  auto v =
      std::any_cast<std::vector<std::pair<int, std::string_view>>>(setting);
  data<std::pair<int, std::string_view>>() = v;
} break;
  // attributes stored in map but filtered for processing.
case dt_indexBy: {
  updateIndexBy(std::any_cast<indexBy>(setting));
  bSaveInMap = true;
} break;
  // other items are not filtered, so just pass through to storage.
case dt_nonFiltered: {
  bSaveInMap = true;
```

```
    } break;
    }

    if (bSaveInMap)
      attributes[std::type_index(setting.type())] = setting;

    return *this;
  }

  template <typename... TYPES> Element &setAttribute(const TYPES... settings) {
    setAttribute(settings...);
    return *this;
  }

  Element &setAttribute(const std::vector<std::any> &attribs) {
    for (auto n : attribs)
      setAttribute(n);
    return *this;
  }

public:
  template <typename ATTR_TYPE> ATTR_TYPE &getAttribute(void) {
    ATTR_TYPE *ret = nullptr;
    std::unordered_map<std::type_index, std::any>::iterator it =
        attributes.find(std::type_index(typeid(ATTR_TYPE)));
    if (it != attributes.end())
      ret = &std::any_cast<ATTR_TYPE &>(it->second);
    return *ret;
  }

public:
  Element &clear(void) { return *this; }

public:
  std::vector<eventHandler> onfocus;
  std::vector<eventHandler> onblur;
  std::vector<eventHandler> onresize;
  std::vector<eventHandler> onkeydown;
  std::vector<eventHandler> onkeyup;
  std::vector<eventHandler> onkeypress;
  std::vector<eventHandler> onmouseenter;
  std::vector<eventHandler> onmouseleave;
  std::vector<eventHandler> onmousemove;
  std::vector<eventHandler> onmousedown;
  std::vector<eventHandler> onmouseup;
  std::vector<eventHandler> onclick;
  std::vector<eventHandler> ondblclick;
  std::vector<eventHandler> oncontextmenu;
  std::vector<eventHandler> onwheel;

private:
  std::vector<eventHandler> &getEventVector(eventType evtType);

public:
  auto move(const double t, const double l) -> Element &;
  auto resize(const double w, const double h) -> Element &;
  auto addListener(eventType evtType, eventHandler evtHandler) -> Element &;
  auto removeListener(eventType evtType, eventHandler evtHandler) -> Element &;
  void render(void);

  auto insertBefore(Element &newChild, Element &existingElement) -> Element &;

  auto insertAfter(Element &newChild, Element &existingElement) -> Element &;

  void remove(void);
  auto removeChild(Element &childElement) -> Element &;
  auto removeChildren(Element &e) -> Element &;
  auto replaceChild(Element &newChild, Element &oldChild) -> Element &;
```

```
#if defined(__clang__)
  void printf(const char *fmt, ...)
      __attribute__((__format__(__printf__, 2, 0)));
#else
  void printf(const char *fmt, ...);
#endif

  auto ingestMarkup(Element &node, const std::string_view &markup) -> Element &;

private:
  void updateIndexBy(const indexBy &setting);

}; // class Element

//
auto operator""_pt(unsigned long long int value) -> doubleNF {
  return doubleNF{static_cast<double>(value), numericFormat::pt};
}
auto operator""_pt(long double value) -> doubleNF {
  return doubleNF{static_cast<double>(value), numericFormat::pt};
}
auto operator""_em(unsigned long long int value) -> doubleNF {
  return doubleNF{static_cast<double>(value), numericFormat::em};
}
auto operator""_em(long double value) -> doubleNF {
  return doubleNF{static_cast<double>(value), numericFormat::em};
}
auto operator""_px(unsigned long long int value) -> doubleNF {
  return doubleNF{static_cast<double>(value), numericFormat::px};
}
auto operator""_px(long double value) -> doubleNF {
  return doubleNF{static_cast<double>(value), numericFormat::px};
}
auto operator""_percent(unsigned long long int value) -> doubleNF {
  return doubleNF{static_cast<double>(value), numericFormat::percent};
}
auto operator""_percent(long double value) -> doubleNF {
  return doubleNF{static_cast<double>(value), numericFormat::percent};
}
auto operator""_pct(unsigned long long int value) -> doubleNF {
  return doubleNF{static_cast<double>(value), numericFormat::percent};
}
auto operator""_pct(long double value) -> doubleNF {
  return doubleNF{static_cast<double>(value), numericFormat::percent};
}
auto operator""_normal(unsigned long long int value) -> lineHeight {
  return lineHeight{static_cast<double>(value), lineHeight::normal};
}
auto operator""_normal(long double value) -> lineHeight {
  return lineHeight{static_cast<double>(value), lineHeight::normal};
}
auto operator""_numeric(unsigned long long int value) -> lineHeight {
  return lineHeight{static_cast<double>(value), lineHeight::numeric};
}
auto operator""_numeric(long double value) -> lineHeight {
  return lineHeight{static_cast<double>(value), lineHeight::numeric};
}

using BR = class BR : public Element {
public:
  BR(const std::vector<std::any> &attribs) : Element() {
    setAttribute(attribs);
  }
};

using H1 = class H1 : public Element {
public:
  H1(const std::vector<std::any> &attribs)
      : Element({display::block, marginTop{.67_em}, marginLeft{.67_em},
              marginBottom{0_em}, marginRight{0_em}, textSize{2_em},
              textWeight{800}}) {
    setAttribute(attribs);
  }
};

using H2 = class H2 : public Element {
public:
  H2(const std::vector<std::any> &attribs)
      : Element({display::block, marginTop{.83_em}, marginLeft{.83_em},
              marginBottom{0_em}, marginRight{0_em}, textSize{1.5_em},
              textWeight{800}}) {
    setAttribute(attribs);
  }
};

using H3 = class H3 : public Element {
public:
  H3(const std::vector<std::any> &attribs)
      : Element({display::block, marginTop{1_em}, marginLeft{1_em},
              marginBottom{0_em}, marginRight{0_em}, textSize{1.17_em},
              textWeight{800}}) {
    setAttribute(attribs);
  }
};

using PARAGRAPH = class PARAGRAPH : public Element {
public:
  PARAGRAPH(const std::vector<std::any> &attribs)
      : Element({listStyleType::disc, marginTop{1_em}, marginLeft{1_em},

          marginBottom{0_em}, marginRight{0_em}}) {
    setAttribute(attribs);
  }
};

using DIV = class DIV : public Element {
public:
  DIV(const std::vector<std::any> &attribs) : Element({display::block}) {
    setAttribute(attribs);
  }
};

using SPAN = class SPAN : public Element {
public:
  SPAN(const std::vector<std::any> &attribs) : Element() {
    setAttribute(attribs);
  }
};

using UL = class UL : public Element {
public:
  UL(const std::vector<std::any> &attribs)
      : Element({listStyleType::disc, display::block, marginTop{1_em},
              marginLeft{1_em}, marginBottom{0_em}, marginRight{0_em},
              paddingLeft{40_px}}) {
    setAttribute(attribs);
  }
};

using OL = class OL : public Element {
public:
  OL(const std::vector<std::any> &attribs)
      : Element({listStyleType::decimal, display::block, marginTop{1_em},
              marginLeft{1_em}, marginBottom{0_em}, marginRight{0_em},
              paddingLeft{40_px}}) {
    setAttribute(attribs);
  }
};
```

```
using LI = class LI : public Element {
public:
  LI(const std::vector<std::any> &attribs) : Element() {
    setAttribute(attribs);
  }
};

// self understood attribute communication
_VECTOR_ATTRIBUTE(tableColumns);
using TABLE = class TABLE : public Element {
public:
  TABLE(const std::vector<std::any> &attribs) : Element() {
    setAttribute(attribs);
  }
};

using MENU = class MENU : public Element {
public:
  MENU(const std::vector<std::any> &attribs) : Element() {
    setAttribute(attribs);
  }
};

using GUI_OBJECT = class GUI_OBJECT : public Element {
public:
  GUI_OBJECT(const std::vector<std::any> &attribs) : Element() {
    setAttribute(attribs);
  }
};

using IMAGE = class IMAGE : public Element {
public:
  IMAGE(const std::vector<std::any> &attribs) : Element() {
    setAttribute(attribs);
  }
};

class textNode : public Element {
public:
  std::string_view &value;
  textNode(std::string_view &s) : value(s), Element() {}
};

/// USE_LOWER_CASE_ENTITY_NAMES
/// <summary>Options for compiling that provide recognition of
/// lower case entity names.</summary>
#define USE_LOWER_CASE_ENTITY_NAMES

// map lower case terms as well for flexibility
#ifdef USE_LOWER_CASE_ENTITY_NAMES
using element = Element;
using paragraph = PARAGRAPH;
using br = BR;
using h1 = H1;
using h2 = H2;
using h3 = H3;
using dblock = DIV;
using span = SPAN;
using ul = UL;
using ol = OL;
using li = LI;
using menu = MENU;
using gui_object = GUI_OBJECT;
using image = IMAGE;
using textnode = textNode;

#endif
```

```
/********************************************************************************
Templated factory implementation for reference based callee and
 viewManager ownership of the object. The vector deconstructor will cause
proper deallocation.

********************************************************************************/
template <class TYPE>
auto _createElement(const std::vector<std::any> &attrs) -> TYPE & {
  std::unique_ptr<TYPE> e = std::make_unique<TYPE>(attrs);
  elements.push_back(std::move(e));
  return static_cast<TYPE &>(*elements.back().get());
}

template <class TYPE, typename... ATTRS>
auto createElement(const ATTRS &... attribs) -> TYPE & {
  std::vector<std::any> attrvector{{attribs...}};
  return _createElement<TYPE>(attrvector);
}

template <typename... Types> auto createStyle(Types... args) -> StyleClass & {
  std::unique_ptr<StyleClass> newStyle = std::make_unique<StyleClass>(args...);
  styles.push_back(std::move(newStyle));
  return *styles.back().get();
}

auto query(const std::string_view &queryString) -> ElementList;
auto query(const ElementQuery &queryFunction) -> ElementList;

template <class T = Element &>
auto getElement(const std::string_view &key) -> T & {

#if 0
  auto &it = indexedElements.find(key);
  T ret = it->second.get();

  return ret;
#endif
}

bool hasElement(const std::string_view &key);

#define CREATE_OBJECT_ALIAS(OBJECT_TEXT, OBJECT_TYPE)                          \
  {                                                                            \
#OBJECT_TEXT,                                                                  \
        [](const std::vector <std::any> &attrs)                                \
            -> Element & { return _createElement<OBJECT_TYPE>(attrs); }        \
  }

#define CREATE_OBJECT(OBJECT_TYPE) CREATE_OBJECT_ALIAS(OBJECT_TYPE, OBJECT_TYPE)

static std::unordered_map<
    std::string_view,
    std::function<Element &(const std::vector<std::any> &attr)>>
    objectFactoryMap = {CREATE_OBJECT(BR),        CREATE_OBJECT(H1),
                        CREATE_OBJECT(H2),        CREATE_OBJECT(H3),
                        CREATE_OBJECT(PARAGRAPH), CREATE_OBJECT(DIV),
                        CREATE_OBJECT(SPAN),      CREATE_OBJECT(UL),
                        CREATE_OBJECT(OL),        CREATE_OBJECT(LI),
                        CREATE_OBJECT(MENU),      CREATE_OBJECT(GUI_OBJECT),
                        CREATE_OBJECT(IMAGE)};

/*****************************************************************

Viewer
*****************************************************************/

class Viewer : public Element {
public:
  Viewer(const std::vector<std::any> &attribs);
  ~Viewer();
  Viewer(const Viewer &) {}
  Viewer(Viewer &&other) noexcept {}
  Viewer &operator=(const Viewer &) {}
  Viewer &operator=(Viewer &&other) noexcept {} // move assignment

  void render(void);
  void processEvents(void);

  Viewer &clear(void) {}

  // event implementation
  void dispatchEvent(const event &e);

private:
  std::unique_ptr<Visualizer::platform> m_device;
};

/// <summary>returns a tag string for the pointer. The tag is a double
/// pointer so that it can be changed. These macros are useful when
/// element creation is being accomplished using parsing while the
/// software needs the pointer to the created object.</summary>
#define stringPointerTag(a) (symbolicPointer(&a))

/// <summary>useful for a parameter to printf which accepts a char
/// ///.</summary>
///
/// <remarks>
/// The macro returns a tag string for the pointer. The tag is a double
/// pointer so that it can be changed. These macros are useful when
/// element creation is being accomplished using parsing while the
/// software needs the pointer to the created object. The macro creates
/// readable printf statements.
/// </remarks>
#define charPointerTag(a) (symbolicPointer(&a).c_str())

/// <summary> The function returns a string identity of the point within
/// tags. This enables easy integration between textual markup build and c++
/// pointer facilities. Example symbolic pointer:</summary> <paragraphPtr
/// typeHash 0x11111 memoryLocation 0x3333adbc>
///
/// <param name="e">[in] e</param> should be a double pointer that has not
/// been populated yet. That is, it should be nullptr. The value is stored
/// within a tag.</param>
///
template <typename ElementType> std::string symbolicPointer(ElementType **e) {
  std::string elementName = std::type_index(typeid(e)).name();

  /* build meaningful string representative of the object's name.
  The namestring is built differently by the compier depending
  upon the compiler.
  */

#if defined(__linux__)
  std::size_t found = elementName.find_first_of("_0123456789");

  if (found != std::string::npos) {
    elementName = elementName.substr(found + 1);
  }

#elif defined(_WIN64)

#define PREFIX_ID "class "
  std::size_t found = elementName.find_first_of(PREFIX_ID);

  if (found != std::string::npos) {
    elementName = elementName.substr(found + strlen(PREFIX_ID));
    found = elementName.find_first_of(" ");
    if (found != std::string::npos)
```

```
      elementName = elementName.substr(0, found);
  }
#endif

  std::transform(elementName.begin(), elementName.end(), elementName.begin(),
                 ::tolower);

  char *buffer = nullptr;
  int len;

  len = _scprintf("<%sPtr typeHash %zx memoryLocation %zx/>",
                  elementName.c_str(), std::type_index(typeid(*e)).hash_code(),
                  (unsigned long long)e) +
        1;

  buffer = (char *)malloc(len * sizeof(char));
  sprintf_s(buffer, len, "<%sPtr typeHash %zx memoryLocation %zx/>",
            elementName.c_str(), std::type_index(typeid(*e)).hash_code(),
            (unsigned long long)e);

  std::string sFormatted;
  sFormatted.reserve((size_t)len);
  sFormatted = buffer;
  free(buffer);

  return sFormatted;
}
}; // namespace ViewManager

#endif // VIEW_MANAGER_HPP_INCLUDED
```

viewManager.cpp

```
///  <file>viewManagement.cpp</file>
///  <author>Anthony Matarazzo</author>
///  <date>6/16/18</date>
///  <version>1.0</version>
///
///  <summary>class that implements that main creation interface.
/// The file contains initialization, termination.</summary>
///
#include "viewManager.hpp"

using namespace std;
using namespace ViewManager;

/**********
These namespace global lists contain the objects as a system ownership.
********/
std::vector<std::unique_ptr<Element>> ViewManager::elements;
std::unordered_map<std::string_view, std::reference_wrapper<Element>>
    ViewManager::indexedElements;
std::vector<std::unique_ptr<StyleClass>> ViewManager::styles;

Viewer::Viewer(const vector<any> &attrs) : Element(attrs) {
  // setAttribute(indexBy{"_root"});
  Visualizer::openWindow(*this);
}

/// <summary>deconstructor for the view manager object.
/// </summary>
```

```
Viewer::~Viewer() { Visualizer::closeWindow(*this); }

/// <summary>The main entry point for clients after their initial
/// buildup. The function simply calls the traversing function
/// with the main root as the starting point.
///
///
/// </summary>
void Viewer::render(void) {}

/// <summary>
/// This is the only entry point from the platform for the event
/// dispatching system. The routine expects only certain types
/// of messages from the platform. The other events, that are computed,
/// are developed by this routine as needed. Each of these events, whether
/// passed through as the same message, or developed is placed into the
/// viewManager message queue. The background event dispatching fetches
/// messages from this queue, and calls the element's event processor
/// routines. The main thing to rememeber is that the information is
/// processed from a queue and using a background thread.
/// </summary>
void Viewer::dispatchEvent(const event &evt) {

  switch (evt.evtType) {
  case eventType::resize:
    break;
  case eventType::keydown:
    break;
  case eventType::keyup:
    break;
  case eventType::keypress:
    break;

  case eventType::mousemove:
    break;
  case eventType::mousedown:
    break;
  case eventType::mouseup:
    break;

  case eventType::wheel:
    break;
  }

  /* these events do not come from the platfrom. However,
   they are spawned from conditions based upon the platform events.
            */

#if 0
                eventType::focus
                                eventType::blur

                                eventType::mouseenter
                                eventType::click
                                eventType::dblclick
                                eventType::contextmenu
                                eventType::mouseleave

#endif
}

///< summary>The entry point that processes messages from the operating
/// system application level services. Typically on linux this is a
/// coupling of xcb and keysyms library for keyboard. Previous
/// incarnations of techology such as this typically used xserver.
/// However, XCB is the newer form. Primarily looking at the code of such
/// programs as vlc, the routine simply places pixels into the memory
/// buffer. while on windows the direct x library is used in combination
/// with windows message queue processing. </summary>
```

```
void Viewer::processEvents(void) { m_device->messageLoop(); }

/// <summary>given the string id, the function returns
/// the element.
///
/// jquery shows that handling large document models can be
/// effectively managed using searching and iterators. Here,
/// elements or groups of elements can be accessed through query strings.
/// #, or *, or partial matches. Change attributes?
/// @ for style,
/// </summary>

auto ViewManager::query(const std::string_view &queryString) -> ElementList {
  ElementList results;

  if (queryString == "*") {
    for (const auto &n : elements) {
      results.push_back(*n);
    }

  } else {

    std::regex matchExpression(queryString.data(),
                               std::regex_constants::ECMAScript |
                                   std::regex_constants::icase);

    for (const auto &n : elements) {
      if (std::regex_match(n->getAttribute<indexBy>().value.data(),
                           matchExpression))
        results.push_back(*n);
    }
  }
  return results;
}

auto ViewManager::query(const ElementQuery &queryFunction) -> ElementList {
  ElementList results;

  for (const auto &n : elements) {
    if (queryFunction(*n))
      results.push_back(*n);
  }
  return results;
}

bool ViewManager::hasElement(const std::string_view &key) {
  auto it = indexedElements.find(key);
  return it != indexedElements.end();
}

/*******************************************************************************************
                              Element

*******************************************************************************************/
/**
 *  updateElementIdMap
 *  updates the id within the
 * index if the item has
 * changed.
 *
 */
void Element::updateIndexBy(const indexBy &setting) {

  // changing id just changes
  // the key in elementById
  // map
  std::string_view oldKey = "";
  const std::string_view &newKey = setting.value;
  std::unordered_map<std::type_index, std::any>::iterator it =
      attributes.find(std::type_index(typeid(indexBy)));

  // get the key of the old id
  if (it != attributes.end()) {
    oldKey = std::any_cast<indexBy>(it->second).value;
  }

  // case a. key is not blank,
  // yet it is the same value
  // as the old key therefore
  // there is no change.
  if (!oldKey.empty() && oldKey == newKey) {
    return;

    // case b. remap just the
    // key
  } else if (!oldKey.empty() && !newKey.empty()) {
    auto nodeHandler = indexedElements.extract(oldKey);
    nodeHandler.key() = newKey;
    indexedElements.insert(std::move(nodeHandler));

    // case c. remove key from
    // map
  } else if (!oldKey.empty() && newKey.empty()) {
    indexedElements.erase(oldKey);

    // case d. did not exist
    // before, add key to map
  } else if (!newKey.empty()) {
    indexedElements.insert({newKey, std::ref(*this)});
  }

  return;
}

auto Element::insertBefore(Element &newChild, Element &existingElement)
    -> Element & {
  Element &child = newChild;

  // maintain tree structure
  child.m_parent = existingElement.m_parent;
  child.m_nextSibling = existingElement.m_self;
  child.m_previousSibling = existingElement.m_previousSibling;
  existingElement.m_previousSibling = child.m_self;

  // provide linkage
  if (child.m_previousSibling)
    child.m_previousSibling->m_nextSibling = child.m_self;

  if (child.m_nextSibling)
    child.m_nextSibling->m_previousSibling = child.m_self;

  // case where insert is at the first
  if (existingElement.m_self == m_firstChild) {
    m_firstChild = child.m_self;
  }

  m_childCount++;

  return child;
}

auto Element::insertAfter(Element &newChild, Element &existingElement)
    -> Element & {

  return newChild;
}
```

```
auto Element::removeChild(Element &oldChild) -> Element & { return *this; }

auto Element::replaceChild(Element &newChild, Element &oldChild) -> Element & {

  return *this;
}

auto Element::move(const double t, const double l) -> Element & {
  getAttribute<objectTop>().value = t;
  getAttribute<objectLeft>().value = l;
  return *this;
}

auto Element::resize(const double w, const double h) -> Element & {
  getAttribute<objectWidth>().value = w;
  getAttribute<objectHeight>().value = h;
  return *this;
}

void Element::remove(void) { return; }

auto Element::removeChildren(Element &e) -> Element & { return *this; }

/// <summary>The function mapps the event id to the appropiate vector.
/// This is kept statically here for resource management.</summary>
///
vector<eventHandler> &Element::getEventVector(eventType evtType) {
  static unordered_map<eventType, vector<eventHandler> &> eventTypeMap = {
      {eventType::focus, onfocus},
      {eventType::blur, onblur},
      {eventType::resize, onresize},
      {eventType::keydown, onkeydown},
      {eventType::keyup, onkeyup},
      {eventType::keypress, onkeypress},
      {eventType::mouseenter, onmouseenter},
      {eventType::mouseleave, onmouseleave},
      {eventType::mousemove, onmousemove},
      {eventType::mousedown, onmousedown},
      {eventType::mouseup, onmouseup},
      {eventType::click, onclick},
      {eventType::dblclick, ondblclick},
      {eventType::contextmenu, oncontextmenu},
      {eventType::wheel, onwheel}};

  unordered_map<eventType, vector<eventHandler> &>::iterator it;
  it = eventTypeMap.find(evtType);
  return it->second;
}

/// <summary>
/// The function will return the address of a std::function for the purposes
/// of equality testing. Function from
/// https://stackoverflow.com/questions/20833453/comparing-stdfunctions-for-equality
/// </summary>
template <typename T, typename... U>
size_t getAddress(std::function<T(U...)> f) {
  typedef T(fnType)(U...);
  fnType **fnPointer = f.template target<fnType *>();
  return (size_t)*fnPointer;
}

auto Element::addListener(eventType evtType, eventHandler evtHandler)
    -> Element & {
  getEventVector(evtType).push_back(evtHandler);

  return *this;
}
/// <summary>The function will remove an event listener from the list of
/// events to receive messages.</summary>
```

```
///
/// <param name="evtType"> is the type of event to remove.</param>
/// <param name="evtHandler"> is the event to remove.?</param>
///
///
auto Element::removeListener(eventType evtType, eventHandler evtHandler)
    -> Element & {

  auto eventList = getEventVector(evtType);
  auto it = eventList.begin();

  while (it != eventList.end()) {
    if (getAddress(*it) == getAddress(evtHandler))
      it = eventList.erase(it);
    else
      it++;
  }

  return *this;
}

///
/// <summary>This is the main function which invokes drawing of the item and
/// its children. It is called recursively when painting needs to occur.
/// This function is used internally and is not necessary to invoke. That
/// is, system already invokes this as part of the processing stack. The
/// work performed by this routine is accomplished using the surface image.
///
 </summary>
///
void Element::render(void) {}

/// <summary>Uses the vasprint standard function to format the given
/// parameters with the format string. Inserts the named
/// elements within the markup into the document.</summary>
///
/// <param name="fmt"> is a printf format string. It should be a literal
/// string.</param> <param name="..."> is a variable argument parameter
/// passed to the standard printf function. </param>
///`
///
/// <code>
///
///  vector<string> movies={"The Hulk",
///                                "Super Action Hero",
///                                "Star Invader Eclipse"};
///
/// auto& e=createElement();
///
/// e.printf("The movie theatre is <blue>opened</blue> today for matinee.");
/// e.printf("Here is a list of movies: <ul>");
/// for(auto n : movies)
///              e.prinf("<li>%s</li>",n.c_str());
/// e.printf("</ul>");
///
/// </code>
///
/// <note>
/// <para>If a literal is not used for the first parameter, a warning will
/// be issued. This warning is effective because at times, it reminds the
/// developer that if the format string comes from a foreign source
/// and is not controlled, the stack may be violated.</para>
///
/// <para>To prevent this, the function has a static attribute type check
/// that is used during compile time. The actual parameter format
/// is associated with the 2 parameter. That is
/// this is actually in the 1 slot due to it being a
/// class member. So when a literal is not used for the first
/// parameter, a warning is issued.</para>
/// </note>
void Element::printf(const char *fmt, ...) {
#if defined(__linux__)
  va_list ap;
  va_start(ap, fmt);
  char *buffer = nullptr;

  /* These are checked with the __attribute__ setting on the function
   * declare above. Turning them off here makes it only report warnings  to
   * calls of this member function and not the asprintf call. As well,
   * buffer is passed as a double pointer and is used to note the allocated
   * memory. It is also freed within this routine. */
#pragma clang diagnostic ignored "-Wformat-security"
#pragma clang diagnostic ignored "-Wformat-nonliteral"

  vasprintf(&buffer, fmt, ap);

#pragma clang diagnostic warning "-Wformat-security"
#pragma clang diagnostic warning "-Wformat-nonliteral"

  ingestMarkup(*this, buffer);
  free(buffer);

  va_end(ap);

#elif defined(__WIN64)
  va_list ap;
  va_start(ap, fmt);
  char *buffer = nullptr;
  int len;

  len = _vscprintf(fmt, ap) + 1;

  buffer = (char *)malloc(len * sizeof(char));
  vsprintf(buffer, fmt, ap);

  ingestMarkup(*this, buffer);
  free(buffer);

  va_end(ap);
#endif
}

Element &Element::ingestMarkup(Element &node, const std::string_view &markup) {
  return *this;
}

/****************************************************************************
                                    Visualizer  module

****************************************************************************/

std::unordered_map<std::size_t, std::reference_wrapper<Element>> nodes;

std::size_t ViewManager::Visualizer::allocate(Element &e) {
  static std::size_t token = 0;
  std::size_t ret = token;
  nodes.emplace(ret, std::ref<Element>(e));
  token++;

  return ret;
}

void ViewManager::Visualizer::deallocate(const std::size_t &token) {}

void ViewManager::Visualizer::openWindow(Element &e) {}
void ViewManager::Visualizer::closeWindow(Element &e) {}

/****************************************************************************
```

```
                                    Visualizer Platform module

****************************************************************************/
ViewManager::Visualizer::platform::platform(eventHandler evtDispatcher,
                                       unsigned short width,
                                       unsigned short height) {

  dispatchEvent = evtDispatcher;

#if defined(__linux__)

  /* Open the connection to the X server */
  m_connection = xcb_connect(nullptr, nullptr);

  /* Get the first screen */
  m_screen = xcb_setup_roots_iterator(xcb_get_setup(m_connection)).data;
  m_syms = xcb_key_symbols_alloc(m_connection);

  /* Create black (foreground) graphic context */
  m_window = m_screen->root;
  m_foreground = xcb_generate_id(m_connection);

  uint32_t mask = XCB_GC_FOREGROUND | XCB_GC_GRAPHICS_EXPOSURES;
  uint32_t values[2] = {m_screen->black_pixel, 0};
  xcb_create_gc(m_connection, m_foreground, m_window, mask, values);

  /* Create a window */
  m_window = xcb_generate_id(m_connection);
  mask = XCB_CW_BACK_PIXEL | XCB_CW_EVENT_MASK;
  values[0] = m_screen->white_pixel;
  values[1] = XCB_EVENT_MASK_EXPOSURE | XCB_EVENT_MASK_KEY_PRESS |
              XCB_EVENT_MASK_KEY_RELEASE | XCB_EVENT_MASK_POINTER_MOTION |
              XCB_EVENT_MASK_BUTTON_MOTION | XCB_EVENT_MASK_BUTTON_PRESS |
              XCB_EVENT_MASK_BUTTON_RELEASE;

  xcb_create_window(m_connection, XCB_COPY_FROM_PARENT, m_window,
                    m_screen->root, 0, 0, m_clientWidth, m_clientHeight, 10,
                    XCB_WINDOW_CLASS_INPUT_OUTPUT, m_screen->root_visual, mask,
                    values);

  /* Map the window on the screen and flush*/
  xcb_map_window(m_connection, m_window);
  xcb_flush(m_connection);

  return;

#elif defined(_WIN64)
  HRESULT hr;

  // Create a Direct2D factory.
  hr =
    D2D1CreateFactory(D2D1_FACTORY_TYPE_SINGLE_THREADED, &m_pDirect2dFactory);

  if (FAILED(hr))
    return;

  // Register the window class.
  WNDCLASSEX wcex = {sizeof(WNDCLASSEX)};
  wcex.style = CS_HREDRAW | CS_VREDRAW;
  // ******** wcex.lpfnWndProc = &Visualizer::platform::WndProc;
  wcex.cbClsExtra = 0;
  wcex.cbWndExtra = sizeof(LONG_PTR);
  wcex.hInstance = HINST_THISCOMPONENT;
  wcex.hbrBackground = NULL;
  wcex.lpszMenuName = NULL;
  wcex.hCursor = LoadCursor(NULL, IDI_APPLICATION);
  wcex.lpszClassName = "ViewManagerApp";

  RegisterClassEx(&wcex);
  // Create the window.
  m_hwnd = CreateWindow("ViewManagerApp", "ViewManager Application",
                        WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT,
                        static_cast<UINT>(width), static_cast<UINT>(height),
                        NULL, NULL, HINST_THISCOMPONENT, 0L);
  hr = m_hwnd ? S_OK : E_FAIL;
  if (FAILED(hr))
    return;

  SetWindowLongPtr(m_hwnd, GWLP_USERDATA, (long long)this);

  ShowWindow(m_hwnd, SW_SHOWNORMAL);
  UpdateWindow(m_hwnd);

  RECT rc;
  GetClientRect(m_hwnd, &rc);

  // Create a Direct2D render target
  hr = m_pDirect2dFactory->CreateHwndRenderTarget(
      D2D1::RenderTargetProperties(),
      D2D1::HwndRenderTargetProperties(
          m_hwnd, D2D1::SizeU(rc.right - rc.left, rc.bottom - rc.top)),
      &m_pRenderTarget);

  if (FAILED(hr))
    return;

#endif
}

#if defined(_WIN64)
///< summary>The default window message procesor for the application.
/// This is the version ofr the Microsoft Windows operating system.
///</summary>
LRESULT CALLBACK ViewManager::Visualizer::platform::WndProc(HWND hwnd,
                                                            UINT message,
                                                            WPARAM wParam,
                                                            LPARAM lParam) {
  LRESULT result = 0;

  bool handled = false;

  LONG_PTR lpUserData = GetWindowLongPtr(hwnd, GWLP_USERDATA);
  platform *platformInstance = (platform *)lpUserData;

  switch (message) {
  case WM_SIZE:
    platformInstance->dispatchEvent(event{eventType::resize,
                                          static_cast<short>(LOWORD(lParam)),
                                          static_cast<short>(HIWORD(lParam))});

    result = 0;
    handled = true;
    break;

  case WM_KEYDOWN: {
    UINT scandCode = (lParam >> 8) & 0xFFFFFF00;
    WCHAR lBuffer[10];
    BYTE State[256];

    GetKeyboardState(State);
    ToUnicode(wParam, scandCode, State, lBuffer, wcslen(lBuffer), 0);

    platformInstance->dispatchEvent(
        event{eventType::keydown, wstring(lBuffer)});
  } break;

  case WM_KEYUP: {
```

```
  UINT scandCode = (lParam >> 8) & 0xFFFFFF00;
  WCHAR lBuffer[10];
  BYTE State[256];

  GetKeyboardState(State);
  ToUnicode(wParam, scandCode, State, lBuffer, wcslen(lBuffer), 0);

  platformInstance->dispatchEvent(event{eventType::keyup, wstring(lBuffer)});
} break;

case WM_CHAR: {
  WCHAR tmp[2];
  tmp[0] = wParam;
  tmp[1] = 0x00;

  platformInstance->dispatchEvent(event{eventType::keypress, wstring(tmp)});
} break;

case WM_LBUTTONDOWN:
  platformInstance->dispatchEvent(
      event{eventType::mousedown, static_cast<short>(LOWORD(lParam)),
            static_cast<short>(HIWORD(lParam)), 1});
  break;

case WM_LBUTTONUP:
  platformInstance->dispatchEvent(
      event{eventType::mouseup, static_cast<short>(LOWORD(lParam)),
            static_cast<short>(HIWORD(lParam)), 1});
  break;

case WM_MBUTTONDOWN:
  platformInstance->dispatchEvent(
      event{eventType::mousedown, static_cast<short>(LOWORD(lParam)),
            static_cast<short>(HIWORD(lParam)), 2});
  break;

case WM_MBUTTONUP:
  platformInstance->dispatchEvent(
      event{eventType::mouseup, static_cast<short>(LOWORD(lParam)),
            static_cast<short>(HIWORD(lParam)), 2});
  break;

case WM_RBUTTONDOWN:
  platformInstance->dispatchEvent(
      event{eventType::mousedown, static_cast<short>(LOWORD(lParam)),
            static_cast<short>(HIWORD(lParam)), 3});
  break;

case WM_RBUTTONUP:
  platformInstance->dispatchEvent(
      event{eventType::mouseup, static_cast<short>(LOWORD(lParam)),
            static_cast<short>(HIWORD(lParam)), 3});
  break;

case WM_MOUSEMOVE:
  platformInstance->dispatchEvent(event{eventType::mousemove,
                                        static_cast<short>(LOWORD(lParam)),
                                        static_cast<short>(HIWORD(lParam))});
  result = 0;
  handled = true;
  break;

case WM_MOUSEWHEEL:
  platformInstance->dispatchEvent(event{

      eventType::wheel, static_cast<short>(LOWORD(lParam)),
      static_cast<short>(HIWORD(lParam)), GET_WHEEL_DELTA_WPARAM(wParam)});
  break;
  case WM_DISPLAYCHANGE:
    InvalidateRect(hwnd, NULL, FALSE);
    result = 0;
    handled = true;
    break;

  case WM_PAINT:
    platformInstance->dispatchEvent(event{eventType::paint});
    ValidateRect(hwnd, NULL);
    result = 0;
    handled = true;
    break;

  case WM_DESTROY:
    PostQuitMessage(0);
    result = 1;
    handled = true;
    break;
  }

  if (!handled)
    result = DefWindowProc(hwnd, message, wParam, lParam);

  return result;
}
#endif

/// <summary>terminates the xserver connection
/// and frees resources.
/// </summary>
ViewManager::Visualizer::platform::~platform() {

#if defined(__linux__)

  xcb_free_gc(m_connection, m_foreground);
  xcb_key_symbols_free(m_syms);

#elif defined(_WIN64)
  m_pDirect2dFactory->Release();
  m_pRenderTarget->Release();
  m_pOffscreen->Release();
  m_offScreenBitmap->Release();
#endif
}

void ViewManager::Visualizer::platform::messageLoop(void) {
#if defined(__linux__)

  while ((event = xcb_wait_for_event(m_connection))) {
    switch (event->response_type & ~0x80) {
    case XCB_MOTION_NOTIFY: {
      xcb_motion_notify_event_t *motion = (xcb_motion_notify_event_t *)event;
      dispatchEvent(
          event{eventType::mousemove, motion->event_x, motion->event_y});
    } break;
      .case XCB_BUTTON_PRESS : {
        xcb_button_press_event_t *bp = (xcb_button_press_event_t *)event;
        dispatchEvent(
            event{eventType::mousedown, bp->event_x, event_y, bp->detail});
      }
      break;

    case XCB_BUTTON_RELEASE: {
      xcb_button_release_event_t *br = (xcb_button_release_event_t *)event;
      dispatchEvent(
          event{eventType::mouseup, bp->event_x, event_y, bp->detail});
    } break;

    case XCB_KEY_PRESS: {
      xcb_key_press_event_t *kp = (xcb_key_press_event_t *)event;
      xcb_keysym_t sym = xcb_key_press_lookup_keysym(m_syms, kp, 0);
      dispatchEvent(event{eventType::keydown, sym});
    } break;

    case XCB_KEY_RELEASE: {
      xcb_key_release_event_t *kr = (xcb_key_release_event_t *)event;
      xcb_keysym_t sym = xcb_key_press_lookup_keysym(m_syms, kr, 0);
      dispatchEvent(event{eventType::keyup, sym});
    } break;
    }
    free(event);
  }

#elif defined(_WIN64)
  MSG msg;

  while (GetMessage(&msg, NULL, 0, 0)) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }

#endif
}

void ViewManager::Visualizer::platform::flip() {}
```