GLV — Graphics Library of Views
Quick Tutorial (v. 0.96)

November 22, 2010

Media Arts and Technology and the AlloSphere Research Facility

University of California, Santa Barbara

Author: Lance Putnam
e-mail: ljputnam@umail.ucsb.edu

Contents

1 GLV Overview

GLV (Graphics Library of Views) is a GUI building toolkit written in C++ for Linux, OSX, and Win32. GLV is specially designed for creating interfaces to real-time, multimedia applications using hardware accelerated graphics. GLV has no dependencies on other libraries other than OpenGL which is provided by all modern operating systems. Although windowing is technically not a part of GLV, it does provide an abstraction layer for creating bindings to a particular windowing system for creating an OpenGL context and getting mouse and keyboard input. A binding to GLUT is currently provided.

1.1 Related Work

The following are other cross-platform C++ GUI libraries that use OpenGL for rendering:

	URL	License
AntTweakBar	www.antisphere.com/Wiki/tools:anttweakbar	zlib/libpng
aedGUI	aedgui.sourceforge.net	GPL
CEGUI	www.cegui.org.uk	LGPL or MIT
GiGi	gigi.sourceforge.net	LGPL
GLGooey	glgooey.sourceforge.net	zlib/libpng
GLUI	glui.sourceforge.net	LGPL
GLAM	glam.sourceforge.net	GPL
Guichan	guichan.sourceforge.net	BSD
LibUFO	libufo.sourceforge.net	LGPL
NUI	www.libnui.net	GPL
Turksa	jet.ro/turska	BSD/LGPL

2 Obtaining and Building

The GLV source code can be downloaded from

```
http://mat.ucsb.edu/glv/
```

or checked out through SVN here

```
svn co https://svn.mat.ucsb.edu/svn/glv-svn/trunk glv
```

2.1 Linux Compilation

The simplest way to build the library is to use GNU Make (see section 2.4 below). Use either Synaptic or apt-get to install the most recent developer versions of GLUT and GLEW.

2.2 Mac OS X Compilation

You can build the library using either GNU Make or Xcode. If you are using Make, see section 2.4 below. To use Xcode, you will need to install the developer tools from Apple. You can get them for free (after registration) from http://developer.apple.com/. The Xcode project is located at `osx/GLV.xcodeproj`.

2.3 Windows Compilation

You will need to either use Cygwin or MinGW with GNU make or create a new Visual Studio project. Visual Studio Express can be downloaded for free from Microsoft. Neither of these two methods have been tested, so you can make a contribution to GLV if you come up with a working solution.

2.4 Building With Make

If you had to install Make, test that it is working by opening a terminal and typing 'make –version'. Before running Make, ensure that the correct build options are set in the file `Makefile.config`. These can be set directly in `Makefile.config` or passed in as options to Make as OPTION=value.

To build the GLV library, simply run

```
make
```

and hope for the best.

3 GLV API

This tutorial is intended to be both a learning guide and reference to the basic functionality of GLV. Code examples are provided in-line with the text in many cases and related example files in the project `example/` folder are notated under headings inside of square brackets, e.g. *[exampleFile.cpp]*.

3.1 Core Objects

Rect

`Rect` is a geometry base class that all `Views` inherit from. It represents a rectangular object with data members to describe its position and extent in pixels. Its position is stored as the coordinate of its top-left corner on a cartesian grid with positive x going right and positive y going down.

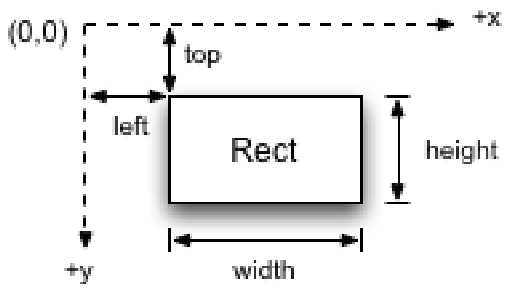


Figure 1: Rectangle model

A `Rect` can be constructed in the following ways:

```
Rect r1(10, 5, 100, 50);     // position (10,5), dimensions (100, 50)
Rect r2(100, 50);            // position  (0,0), dimensions (100, 50)
Rect r3(100);                // position  (0,0), dimensions (100,100)
```

`Rect` has methods for changing its position and extent and multiple methods for getting its right edge, bottom edge, center points, area. There is also a method to determine whether the `Rect` contains an x-y coordinate. This can be used, for instance, to check if a `Rect` has been clicked on in the window.

View

[views.cpp, stretchAnchor.cpp]

A `View` is a `Rect` that has a drawing routine, methods for responding to various events, and properties for being a node in a tree of `View`s. Every `View` has a virtual `onDraw()` method that gets called every frame. This method contains the `View`'s specific OpenGL commands to display it on screen. A `View` contains four `View` references, parent, child, left sibling, and right sibling, that enable it to be a node in a tree structure. This is the main basis for how `View`s are organized spatially and behaviorally within a top-level `View`. `View`s are rendered by traversing the `View` tree pre-order calling each `View`'s `onDraw()` method, starting at the root `View`.

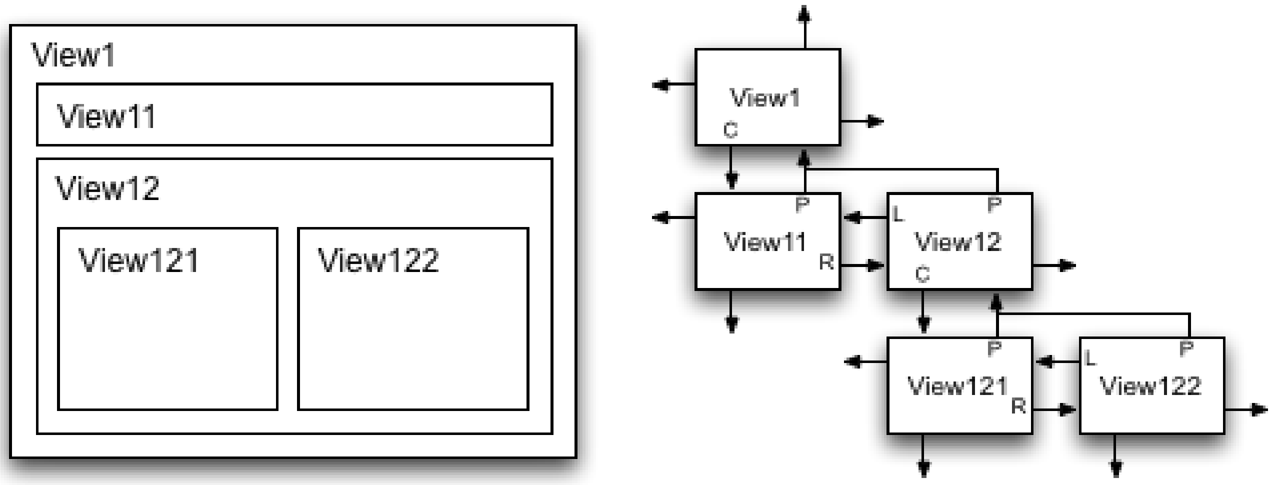


Figure 2: A hierarchical structure of `Views`

The code to produce the hierarchy shown in Fig. 2 is as follows:

```
// Create the Views
View v1;                // top parent
View View v11, v12;     // 2nd level
Views View v121, v122;  // 3rd level Views

// Create the tree hierarchy
v1 << v11 << (v12 << v121 << v122);
```

The << operator adds its right operand as a child to the left operand. The operator returns the parent view, so that multiple children can easily be added using method chaining.

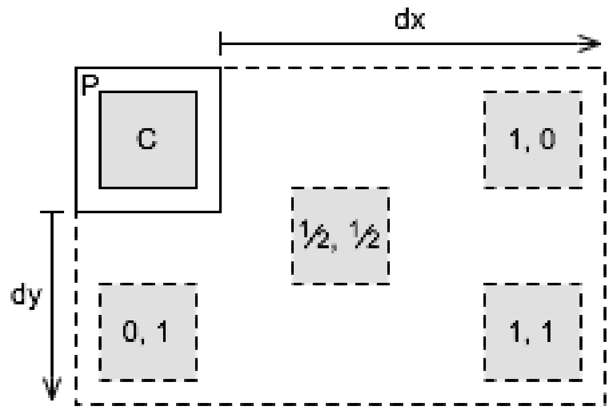
The `View` class has the following enumerated property bit flags:

```
Visible          // Whether to draw myself
DrawBack         // Whether to draw back rect
DrawBorder       // Whether to draw border
CropChildren     // Whether to crop children when drawing
CropSelf         // Whether to crop own drawing routine(s)
Focused          // Whether View is focused
FocusHighlight   // Whether to highlight border when focused
HitTest          // Whether View can be clicked
Controllable     // Whether View can be controlled through events
```

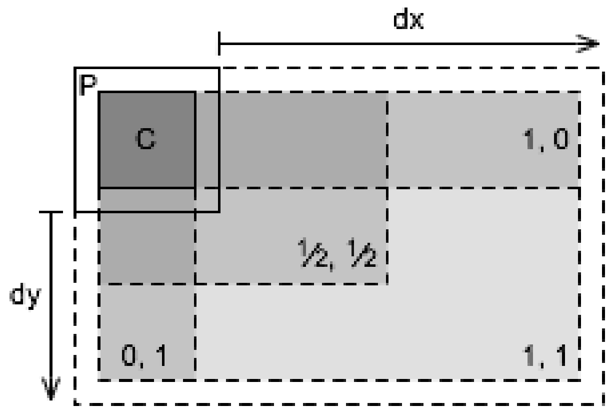
The properties can be modified and queried through several methods:

```
View v;
v.enable(Visible);                           // enable a property
v.disable(DrawBack);                         // disable a property
v.enable(DrawBorder | CropSelf | HitTest);   // enable multiple properties
v.toggle(Visible);                           // toggle a property
v.property(DrawBack, true);                  // set property using a boolean
if(v.enabled(Visible)){...}                  // check if a property is enabled
```

A `View` can also reposition or resize itself automatically when its parent resizes. These capabilities are controlled through its `anchor()` and `stretch()` methods. In the most general case, anchor and stretch factors can be specified in each dimension as floating-point values between 0 and 1. The factors determine how much of the parent's resize amount is added to the view's position and extent. The following figures illustrate how a child view (C) behaves when its parent (P) is resized by dx and dy.



(a) Resize behavior with various x, y anchor factors



(b) Resize behavior with various x, y stretch factors

Figure 3

GLV

The `GLV` class acts as a top-level `View` as well as a bridge to a windowing system. This class handles the drawing of the `View` hierarchy and distributes mouse and keyboard events from the window to its descendents. A `GLV` object is usually constructed with a static drawing callback.

```
void drawCB(View * v){
    // v is a pointer to a GLV
}
GLV top(drawCB);
```

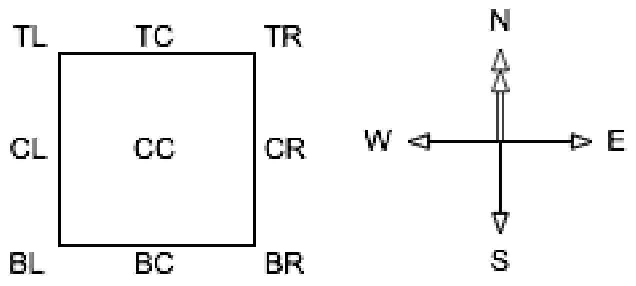
The `GLV` class contains objects with the current keyboard and mouse state that can be accessed from event and drawing callbacks.

Place

`Place` is an enumeration namespace for specifying specific points on a rectangle. The defined types are TL, TC, TR, CL, CC, CR, BL, BC, BR where T = top, B = bottom, L = left, R = right, and C = center. These are primarily used for positioning views and specifying parent anchoring points.

Direction

`Direction` is an enumeration namespace for specifying a direction. The defined types are N, E, S, and W. The convention used in GLV is that north points to the top of the screen and east to the right of the screen. `Direction`s are used primarily for specifying the placement flow of layout managers.



(a) `Place` (b) `Direction`

Figure 4: Illustrations of `Place` and `Direction` enums

Color

The `Color` class stores colors as their constituent red, green, blue, and alpha (RGBA) components. `Color`s can be created in several ways:

```
Color c1(0, 1, 0);                  // green (alpha = 1)
Color c2(0, 0, 1, 0.5);             // blue with alpha
Color c3(0.5);                      // grey scale
```

There is a lightweight HSV struct defined to allow working within the HSV color space. `Color`s can be constructed with this alternate syntax:

```
Color c(HSV(0.5, 1, 1));            // create a new color from HSV values
c.print();                          // prints [0.00 1.00 1.00 1.00]
```

Likewise, we can set and get a color's HSV values:

```
Color c;                            // create a new color
c = HSV(0.5, 1, 1);                 // set its RGB components from HSV values
c.print();                          // prints [0.00 1.00 1.00 1.00]
HSV h;                              // create a new HSV struct
h = c;                              // convert color's RGB components to HSV
printf("[%g␣%g␣%g]", h.h, h.s, h.v);    // prints [0.5 1 1]
```

`HSV` also has a special constructor to allow a short-hand method of getting a color's HSV values.

```
Color c(HSV(0.5, 1, 1));
HSV h(c);                               // create a new HSV struct from a color
printf("[%g␣%g␣%g]", h.h, h.s, h.v);    // prints [0.5 1 1]
```

Style

The `Style` object contains properties that determine the look of a `View`. Each `View` contains a pointer to a single `Style` object. By default, this is set to the global `Style` object `Style::standard()`.

```
View v1;
v1.style();         // reference to Style::standard()
v1.style().color.back   // the background color
v1.colors().back        // shorthand for style().color.back
```

Window/Application

The `Window` class provides the actual operating system window for drawing OpenGL. There are currently only bindings to GLUT, but others can easily be created by subclassing `Window`. The `Window` holds a pointer to a GLV context. The code for setting up a `Window` typically looks like this:

```
GLV top(drawCB);
Window win(800, 600, "GLV Window", &top);
```

The `Application` class is responsible for starting the main application event loop. `Application::run()` is called after setting up the GUI code.

Keyboard

This is a class used to store the current state of the keyboard. The state includes the key code of the last key pressed, whether it was pressed or released, and the state of modifier keys shift, control, and alt.

Printable keyboard keys can be referenced by their non-shifted ASCII character. For non-printable keys, there exists a `Key::t` type with the following types:

```
Enter       F{1-12}     PageDown
BackSpace   Insert      PageUp
Tab         Left        End
Return      Up          Home
Escape      Right
Delete      Down
```

Here is a skeleton code illustrating how to map key events to various actions, such as inside a callback:

```
switch(keyboard.key()){
    case ’b’:           break;
    case ’ ’:           break;
    case Key::Escape:   break;
    case Key::F5:       break;
};
```

Mouse

This is a class used to store the current state of the mouse. The state includes its buttons' state, coordinates relative to the window, coordinates relative to its listener, coordinates relative to the window when each button was pressed, mouse wheel position, and velocity and acceleration in the x and y directions.

Data

`Data` is a dynamically typed, multidimensional array that `View`s use to store model data according to the standard model-view-controller paradigm. The types of data are specified by the enumeration types `BOOL`, `INT`, `FLOAT`, `DOUBLE`, and `STRING`. The special type `VOID` designates no type. The array supports up to four dimensions of arbitrary size.

```
Data d(Data::INT, 5,4,3,2); // allocate array with dimensions 5 x 4 x 3 x 2 of type int
d.type();                   // returns INT
d.order();                  // returns number of dimensions (=4)
d.size();                   // returns total number of elements (=120)
d.size(0);                  // returns size of dimension 0 (=5)
d.size(1);                  // returns size of dimension 1 (=4)
d.size(1,2,3);              // returns product of sizes of dimensions 1, 2, and 3 (=24)
d.shape(2,3,4,5);           // reverse dimensions to be 2 x 3 x 4 x 5

d.assign(123, 0,0,0,0);     // sets element at index (0,0,0,0) to 123
d.at<int>(0,0,0,0);         // gets element at index (0,0,0,0) (=123)
```

One handy feature of `Data` is the ability to create subsets of data called slices. A slice points to the same memory as its originator, but can have a different memory offset, index stride, and shape. By default, whenever a new slice is created it turns into a one-dimensional array with a size equal to the total number of elements of its originator.

```
Data x(Data::INT, 4);
x.assign(10, 0);
x.assign(20, 1);
x.assign(30, 2);
x.assign(40, 3);

Data a = x.slice();                 // [10, 20, 30, 40]
Data b = x.slice(0, d.size()/2, 2); // [10, 30]
Data c = x.slice(1, d.size()/2, 2); // [20, 40]
```

3.2 Messaging

There are two message passing subsystems in GLV— default GUI event callbacks and customizable notifications between objects.

Event Callbacks

A `View` can handle callbacks through virtual methods and/or function pointers. Upon receipt of an event, a `View`'s virtual methods are called, then its list of function pointers, if populated. A `View` can receive any of the following events of type of `Event::t`:

```
Quit    WindowCreate    FocusGained MouseDown   KeyDown
        WindowDestroy   FocusLost   MouseUp     KeyUp
        WindowResize                MouseMove   KeyRepeat
                                    MouseDrag
                                    MouseWheel
```

A `View`'s virtual methods are:

```
virtual void onAnimate(double dsec, GLV& g);
virtual void onDraw(GLV& g);
virtual bool onEvent(Event::t e, GLV& g);
virtual void onResize(space_t dx, space_t dy);
```

The function pointer types are:

```
typedef bool (*eventCallback)(View * v, GLV& g);
typedef void (*drawCallback)(View * v, GLV& g);
```

The event callbacks return a boolean value designating whether the event should be propagated onward to the parent view. A rule of thumb is that the event callback should only return false if it responds to the incoming event, otherwise it should return true so that the event can be handled from a more global scope. Virtual methods can be overridden by subclasses to customize their behavior. The following code example demonstrates how to create a `View` subclass that defines its own drawing callback and adds additional event handling.

```
class SubView : public BaseView{
public:
    virtual void onDraw(GLV& g){
        // drawing commands within local pixel space
    }

    virtual bool onEvent(Event::t e, GLV& g){
        bool r = true;

        switch(e){
        case Event::MouseDrag:  return false;
        case Event::MouseDown:  return false;
        case Event::KeyDown:    return false;
        default:;
        }

        return r && BaseView::onEvent(e, g);
    }

    virtual void onResize(space_t dx, space_t dy){
        // dx and dy are the changes in dimensions
    }
};
```

A `View` also has a single pointer to a `drawCallback` type and a map of `eventCallback` lists. A `View`'s function pointer type callbacks are called after its virtual callbacks. The event callbacks are special in that one or more can be added per event type. `View` has an overloaded, chainable function operator defined to simplify appending event callbacks.

```
View view;
```

```
// append mouse dragging behaviors to view
view    (Event::MouseDrag, mouseMove)
        (Event::MouseDrag, mouseResize);
```

Notifications

[notification.cpp]

All GLV Views inherit a Notifier class so they can operate as the subject of one or more observers. The principle of operation is for a Notification object to be passed from the sending object (subject) to the receiving objects (observers). The Notification holds pointers to the sender, receiver, and optional data and is handled by a user-defined callback function. This is used primarily for sending out notifications when a widget changes its value. For instance, we may want to update a label according to the value of a slider:

```
// Notification callback
void sliderUpdateLabel(const Notification& n){
    Label& l = *n.receiver<Label>();
    Slider& v = *n.sender<Slider>();
    l.setValue(v.getValue());
}

int main(){
    Slider slider;
    Label sliderLabel;

    // Notify the label whenever the slider value is updated.
    slider.attach(sliderUpdateLabel, Update::Value, &sliderLabel);
}
```

3.3 Drawing Commands

Drawing commands can be issued either directly through OpenGL or by using the commands found in the draw namespace (in glv_draw.h). The drawing namespace consists of a minimal wrapper around the OpenGL API, graphics buffers, an assortment of shape drawing routines, and vector-based text rendering.

GraphicsData

[graphicsData.cpp]

GLV uses buffers versus "immediate mode" for all drawing. This has several advantages: 1) rendering is more efficient, both in terms of data bandwidth and rendering speed, 2) vertices and colors do not have to be sent in a particular order, 3) data can be produced once and rendered multiple times, e.g., to multiple viewports, and 4) geometry can rendered using indices. In GLV, an abstraction called GraphicsData stores separate buffers of vertices, colors, and indices that can be rendered through the draw::paint function. The buffers contained in GraphicsData grow dynamically as elements are added to them, so explicit memory management is generally not required. The following code example illustrates how to draw a very simple mesh.

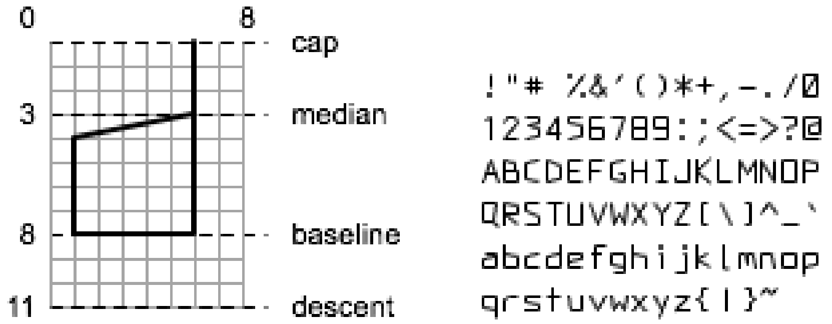
```
GraphicsData gd;        // create a new set of graphics buffers
```

```
// inside a drawing routine...
{
    // reset internal "taps" of buffers to beginning
    gd.reset();

    // generate rainbow heptagon
    for(int i=0; i<7; ++i){
        float p = float(i)/7;
        gd.addColor(HSV(p));
        gd.addVertex(cos(p*2*M_PI), sin(p*2*M_PI));
    }

    // draw heptagon
    draw::paint(draw::LineLoop, gd);

    // generate indices 0, 3, 6, 2, 5, 1, 4
    for(int i=0; i<7; ++i){
        gd.addIndex(i*3 % 7);
    }

    // draw star heptagon
    draw::paint(draw::LineLoop, gd);
}
```

Text

[drawText.cpp]

The `draw` namespace has a function for rendering simple text to the screen. The text renderer uses an internal fixed-width vector font with each character defined on an 8x11 unit grid. The cap lies at 0 units and the baseline at 8 units. In general, characters are rendered with the minimal number of vertices that does not sacrifice their readability and distinction from others. The text rendering mechanism is not meant to be comprehensive, but to be quick and easy without requiring dependencies on external libraries or font files.



(a) Font metrics (b) Font glyphs

Figure 5: Default GLV vector font

A text string can be rendered using one line of code:

```
draw::text("Amazingly␣few␣discotheques␣provide␣jukeboxes.");
```

In this example, the text is drawn with its left edge at 0 and letter cap at 0. Text strings with new lines, \n, and tabs, \t, are also handled properly by the renderer.

3.4 Widgets

[widgets.cpp]

Widget

[attachVariable.cpp]

`Widget` is the base class for widgets that control one or more numerical values, such as sliders and buttons. It contains an array of values and an interval to which those values are restricted.

```
Sliders w(Rect(100,20), 8); // make a bank of 8 sliders
w.interval(100, 0);         // set value interval to [0,100]
w.min();                    // get lower bound of interval (=0)
w.max();                    // get upper bound of interval (=100)
w.setValue(50, 0);          // set slider 0 to 50
w.setValue(90, 1);          // set slider 1 to 90
w.getValue(0);              // get slider 0 value (=50)
```

For widgets having more than one value, there are methods for accessing specific widget elements.

```
Buttons w(Rect(100), 4,2);  // make a 4x2 grid of buttons
w.size();                   // get total number of elements (=16)
w.sizeX();                  // get number of elements across x dimension (=4)
w.sizeY();                  // get number of elements across y dimension (=2)
w.select(1,1);              // select button at x,y position (1,1)
w.select(5);                // select button at x,y position (5/4, 5 mod 4) = (0, 1)
w.selected();               // get last selected button (=5)
```

It is possible to attach variables to a `ValueWidget` that get updated whenever the widget changes value.

```
Slider2D w;
float x,y;                  // some variables to control

w.attachVariable(x, 0);     // attach variable to the first widget value
w.attachVariable(y, 1);     // atach variable to the second widget value

w.setValue(0.5, 0);         // x is set to 0.5
w.setValue(0.1, 1);         // y is set to 0.1
```

When variables are attached to a `ValueWidget` they will be polled from the GLV draw loop to see if they differ from the mapped to value in the widget. If the variable differs, then the widget will synchronize its value to the attached variable and send out any notifications to observers.

Label

A `Label` is a text string that is typically attached as a child to another view. By default, `Label`s do not render a background or border allowing them to 'float' on top of other views. Hit testing is also disabled by default preventing them from stealing focus from a parent view. `Label`s can also be specified to render vertically which can be useful for notating the y-axis of graphs.

Button

A `Button` is a widget that has two possible states- on or off. By default, a `Button` is toggleable meaning that its state changes when it is clicked on. A `Button` can also be made non-toggleable so that it becomes active on a `MouseDown` event and inactive on a `MouseUp` event. `Button` has two drawing function pointers for specifying which graphical symbols to use for on and off states.

Buttons

The `Buttons` widget is an extremely flexible N x M button array. It can be used to create menus, radio buttons, and button matrices, for example. The Buttons widgets can be either in mutually exclusive mode or not. When in mutually exclusive mode, only one button can be on at a time. Otherwise, any number of buttons can be on or off at a time.

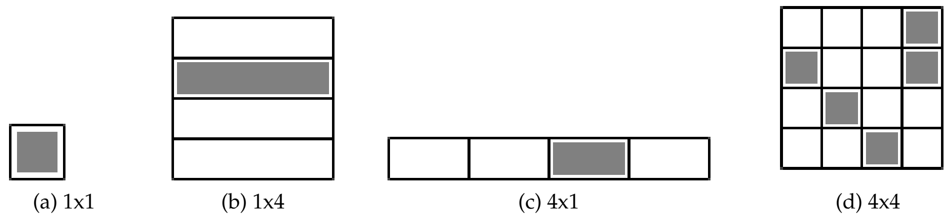


Figure 6: `Button` and `Buttons`

Plot

The `Plot` object permits various types of graphs to be produced from numerical data. A `Plot` contains one or more `Plottable` objects which define a particular drawing behavior. The default behavior is for a `Plottable` to graph whatever data is present in the `Plot` it is attached to. A `Plottable` can also contain its own data so that multiple graphs can be displayed on a single `Plot`. When a `Plottable` has its own data defined, it ignores any data that may be contained in the `Plot`.

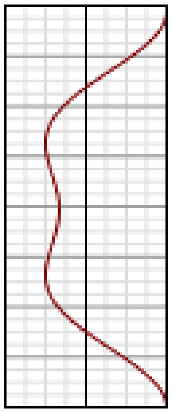
`Plottable` inherits a `GraphicsMap`. `GraphicsMap` defines a virtual method called `onMap` that defines how model data is mapped into graphics data for plotting.

NumberDialer

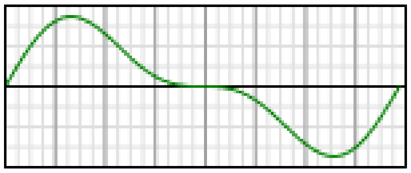
The `NumberDialer` allows one to treat each digit of a number as a dial. The number of digits in the integer and fractional parts is specified upon construction. The `NumberDialer` can also be specified to have a minimum or maximum value. The default amounts are the largest displayable values. When a digit is dragged up or down with the mouse, the number is incremented or decremented by an amount proportional to the place of the digit. The currently selected digit (indicated by a lightly colored box) can also be modified through the number keys on the keyboard. The sign of the number can be toggled by clicking it or by pressing the '-' key.

Slider

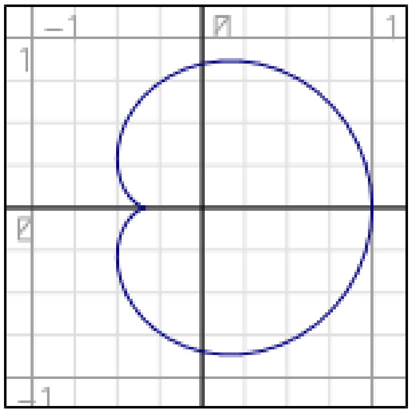
The `Slider` widget allows one to change a value by sliding a bar across the screen. Depending on its dimensions, the slider will automatically be displayed as vertical (w < h) or horizontal (w >= h). The slider



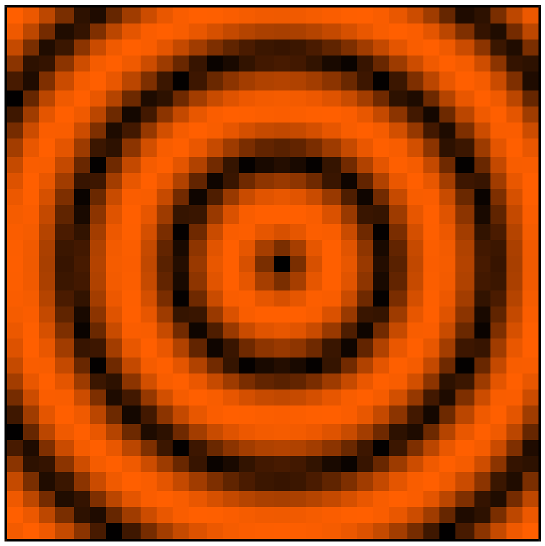
(a) 1D y-axis plot using PlotFunction1D



(b) 1D x-axis plot using PlotFunction1D



(c) 2D plot using PlotFunction2D



(d) 2D plot using PlotFunction2D

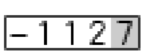
Figure 7: Using various Plottables with Plot

value is 0 at the left or bottom edge and 1 at the top or right edge. In its signed mode, the value is -1 at the left or bottom edge.

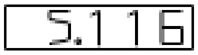
Clicking on the slider with the left mouse button will set its absolute position. Dragging with the left or right mouse button will increment the slider's value. Normally, the amount the slider moves is in a one-to-one correspondence to the mouse cursor position. However, if both mouse buttons are held while dragging, the slider is incremented by a quarter of the distance the mouse moves.

Sliders

The Sliders widget is a group of several sliders. In general, it will be easier and more efficient to control a set of values using a single Sliders widget rather than multiple Slider widgets. The sliders can be displayed either horizontally or vertically. It also supports a special drag set mode where sliders can be set by dragging across the widget as if drawing a curve. Mouse interaction with individual sliders is the same as the Slider widget.

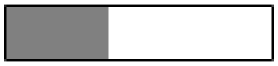


(a) Integer

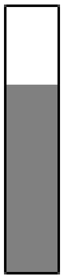


(b) Real

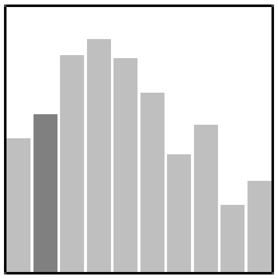
Figure 8: `NumberDialer`



(a) Horizontal `Slider`



(b) Vertical `Slider`



(c) Vertical `Sliders`

Figure 9: `Slider` and `Sliders`

SliderRange

The `SliderRange` widget allows one to control the minimum and maximum values of an interval. The entire interval can be translated by dragging the bar. Clicking and dragging near the ends of the bar changes the associated extremum. Clicking on a blank region causes the whole interval to translate in the direction of the click.

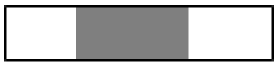


Figure 10: Horizontal `SliderRange`

Slider2D

The `Slider2D` widget is used to control two numeric values simultaneously, such as an x-y position. A knob indicates the current values of the slider. The slider values are 0 at the left or bottom edge and 1 at the top or right edge. Mouse interaction along each axis of `Slider2D` is the same as with the `Slider` widget.

SliderGrid

The `SliderGrid` widget is used to control two or more parameters either alone or in pairs. Each cell in the grid represents one permutation of parameter pairs. The pairs are controlled just like in `Slider2D`, with the exception of the cells along the diagonal. The diagonal cells allow exclusive control over individual parameters. The parameters are layed out on the grid with the first parameter at the bottom-left and the last parameter at the top-right. Within each cell, the slider values are 0 at the left or bottom edge and 1 at the top or right edge. Mouse interaction within cells is the same as with the `Slider2D` widget.

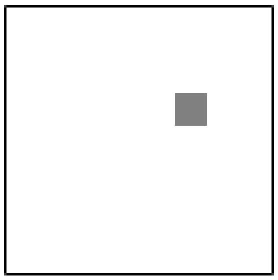
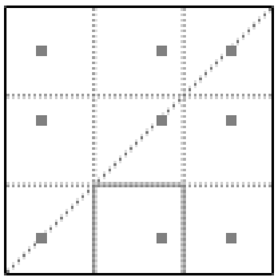
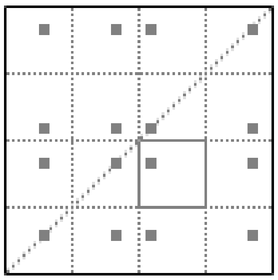


Figure 11: `Slider2D`



(a) Three element `SliderGrid`



(b) Four element `SliderGrid`

Figure 12: `SliderGrid`

View3D

`View3D` allows a 3D scene to be rendered within a View. Custom drawing code is implemented by subclassing `View3D` and overriding the virtual onDraw3D() method. `View3D` has options for setting the near and far clipping planes and the field of view angle in the y direction. By default, depth testing is enabled and blending is disabled.

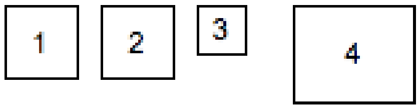
3.5 Layout

Placer

The `Placer` object is used to position a sequence of elements relative to each other, at absolute positions, or a combination of both. Placer holds relative increment factors and absolute increment amounts for both the x and y directions, thus making it an extremely flexible layout positioner.

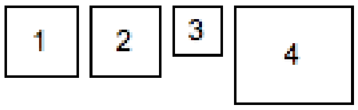
In the simplest case, elements can be arranged so that their absolute positions are a fixed distance apart. This is done by using relative increment factors of zero and absolute increment amounts equal to the distance.

```
View top;
View v1(Rect(6)), v2(v1), v3(Rect(4)), v4(Rect(10,8));

// args: (absolute x-increment, absolute y-increment)
Placer placer(8, 0);
placer << v1 << v2 << v3 << v4;
```

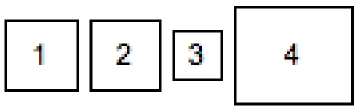
This results in the arrangement: 

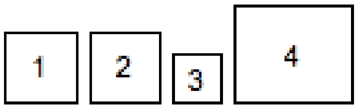
Another use of `Placer` is to position a sequence of views with a fixed amount of padding between them. In this case, the relative increment factor is set to 1, while the absolute increment is set to the desired padding amount. The following example positions elements going east with a padding of 1.

```
View top; View v1(Rect(6)), v2(v1), v3(Rect(4)), v4(Rect(10,8));

// args: (parent view, flow direction, align place, initial x, initial y, padding)
Placer placer(top, Direction::E, Place::TL, 0,0, 1);
placer << v1 << v2 << v3 << v4;
```

This results in the arrangement: 

In the previous example, the alignment was set to the top-left corner. By setting the alignment property of `Placer`, views can be aligned from different places.

Arrangement using center-left alignment: 

Arrangement using bottom-left alignment: 

Table

[tableLayout.cpp]

The `Table` object is used for doing general rectangular layouts on an N x M dimensional grid. A 2D arrangement string is used to specify how `Views` will be arranged when added to the table, starting from the top-left corner and going left-to-right, top-to-bottom. The arrangement string consists of single character codes that specify either existence of elements and their alignments or table cell structure (i.e. spanning, dimensions).

The element alignment codes are:

```
p ^ q   top-left top-center top-right
< x >   center-left center-center center-right
b v d   bottom-left bottom-center bottom-right
```

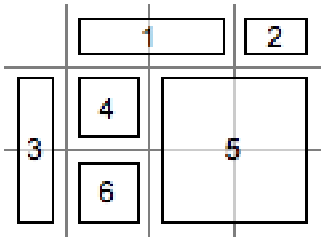
The structural codes are:

```
.   empty region
-   span first leftward neighboring element rightward
|   span first upward neighboring element downward
,   end of row
```

The arrangement string makes it easy to specify complex layouts. The string can be written in a two-dimensional fashion to give a one-to-one visualization of the arrangement.

```
const char * layout =
    ". x - x,"
    "x x x -,"
    "| x | . "

Table table(layout);
```

```
table << v1 << v2 << v3 << v4 << v5 << v6;
table.arrange();
```

This produces: 

If the arrangement string is shorter than the number of `Views` added, then it will be repeated. This feature can be used to define a layout pattern using only a single row specification.

```
Table table("><");
table << v1 << v2 << v3 << v4 << v5 << v6;
table.arrange();
```

This produces: