# VGUI Reference Manual

Karen McGaul
Robotics Research Group
Department of Engineering Science
University of Oxford

September 1, 2000

## Warranties

Robotics Research Group makes no warranties,either express or implied, regarding the enclosed computer software package, its merchantability, or its fitness for any particular purpose. This publication may be revised from time to time, without notice. While all care has been taken to ensure this document's accuracy, Robotics Research Group assumes no responsibility for any errors which may appear.

# Contents

# Chapter 1

# Introduction

This document describes VGUI, a library that provides 2D and 3D drawing in a way that is both simple and powerful. The basic design goals are:

- Provide drawing of images and geometric primitives using OpenGL, in a platform independent manner.
- Define the handling of user events such as keypresses and mouse movements.
- Do *not* deal with widgets (buttons, sliders, menus etc), but instead provide a single simple interface to whatever widget toolkit the user chooses. Current interfaces include ones to GTK, FLTK, glut, and glX.
- OpenGL should be usable “raw”, without any special wrapping by the library, so that existing documentation is used to best advantage.

The basic concept in the new GUI is that the user has access to a rectangular area (or areas) of the screen, on which OpenGL may be used to draw and from which events are received. Such an area is called a *tableau*. Once-off adaptor code is provided that plugs a tableau into any supported GUI toolkit.

# Chapter 2

# Tableaux

Currently VGUI provides several example tableaux, sufficient to construct complex user interfaces combining 2D and 3D with manipulators and zoom. So for example a "zoomer" tableau could provide a function to zoom an image in and out. We could also make a "deck" tableau to hold a deck of images and display one image at a time (this would be useful for playing a series of images as a movie).

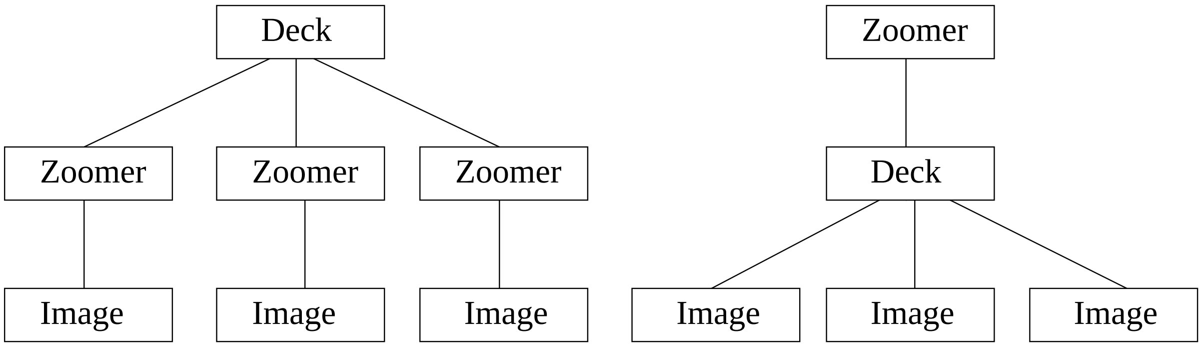


Figure 2.1: Two layouts for zoomer and deck tableaux

One tableau can be included as a child of another tableau. This creates a new tableau with the functionality of both tableaux making it up. So the new tableau could, for example, display a deck of images *and* provide zooming.

The order in which the inclusion is done will affect the functionality, so if the "zoomer" tableau contains the "deck" tableau then all images will change size together. If however the inclusion is done the other way around and each child tableau of the "deck" contains its own "zoomer" tableau then each image will change size independantly.

## 2.1 Example

As an example of a concrete task that will illustrate the use and power of the system, consider the simple case of displaying a 2D image. A tableau that displays an image looks something like:

```
class image_tableau
{
  // image data
  char* image_data;
  int width, height;

  // draw method
  void draw()
  {
    glRasterPos(0,0);
    glDrawPixels(..., image_data, ...);
   }

  // handle event
  bool handle_imple(vgui_event const &e)
  {
    if (e is a draw event)
      draw();
    return true;
  }
};
```

In fact, the draw method will probably be significantly more complex, in order to get maximum efficiency, overcome certain GL weirdnesses, and so on, but the basic code is there.

This interface is abstracted in the obvious manner:

```
class tableau
{
public:
  virtual bool handle_event(vgui_event const&) = 0;
};
```

The key observation is that the tableau assumes that when its draw method is called, the OpenGL system is set up so that the tableau's window is current, the camera is in a sensible position and so on. It is the job of the parent window, in which the tableau is placed, to ensure that this applies, but as we shall see, with this responsibility also comes power...

## 2.2 Adding zoom

To add zooming and panning to the image, we do not need to modify the `image_tableau`. We create a new tableau which has the `image_tableau` as a child, and modifies the GL state before passing the draw request event down to the image.

```
class zoomer_tableau : public tableau
{
  vgui_slot child_tableau;
  float zoom_factor;

  void draw()
  {
    // Change state
    glScale3f(zoom_factor, zoom_factor, zoom_factor);
    // Ask child tableau to draw in new zoomed coordinates
    child->draw();
    // Undo the coordinate change
    glScale3f(1/zoom_factor, 1/zoom_factor, 1/zoom_factor);
  }

  // handle event
  bool handle_imple(vgui_event const &e)
  {
    if (e is a draw event)
      draw();
    return true;
  }
};
```

Now we have a zooming image, but the image tableau never needed to know... The maintainer of the image tableau can concentrate on the thorny issues of making `glDrawPixels` fast on Windows machines, overcoming compiler bugs, and generally writing a high-quality image display routine.

## 2.3 Multiple panes

We might then like to have two zoomed images side by side. Again, we can simply achieve this by having a tableau with two children

```
class composite_tableau : public tableau
  vgui_slot left_tableau;
  vgui_slot right_tableau;

  void draw()
    //  move origin, clip, viewport etc, so all drawing is
    //  in left pane
    glScale(...);
    ...
    glViewport(...);

    //  Draw the left child in the new coordinates
    left->draw();

    //  Reset for right pane
    glScale(...);
    ...
    glViewport(...);

    //  Draw the right child
    right->draw();
```

```
    //  Restore GL state
    ...

  // handle event
  bool handle_imple(vgui_event const &e)
  {
    if (e is a draw event)
      draw();
    return true;
  }
;
```

## 2.4 Event handling

Draw is an event rather than a virtual function so that all the work of the tableau is done in the handle_impl function. The reasoning here is that a tableau should generally be in the same state when drawing as when handling other events. For example, a scribble program which plots points under the mouse should draw the points in the same coordinate system as it recieved them. If draw and handle were different functions, the implementer would have to ensure that the state-handling code was identical in both cases.

The zoomer tableau must likely wishes to capture some mouse events in order to easily change the image pan and zoom. Each tableau implements an event handler which takes a `vgui_event` structure, and either uses the event that it defines, or passes it to its children. In the zoomer example, this might be written

```
class zoomer_tableau : public tableau
{
  vgui_slot child_tableau;
  float zoom_factor;

  // Event handling method.
  bool handle_impl(event e)
  {
    if ( e is control-mouse-drag)
    {
      //  This event interests us, use it to set the zoom factor
      zoom_factor = exp(e.x);
      //  Tell our parent we've used this event
      return true;
    }
    else
    {
      //  This event is uninteresting to us, pass it to the child.
      bool eaten = child->handle(e);

      //  Restore GL state
      ...

      //  Tell our parent whether the child used the event
      return eaten;
    }
};
```

## 2.5 Types of tableaux provided in VGUI

### 2.5.1 Graphics tableaux

- **viewer2D**: Manipulate a 2D tableau. Includes smooth zooming and zooming by sweeping out an area with same aspect ratio as viewport, toggling of anti-aliased points and lines.
- **viewer3D**: Manipulate a 3D tableau. includes dollyzoom (tm awf) toggles many params including headlight, lighting, shading, level of quality. you can also save/restore a position.
- **displaylist2D** and **displaylist3D**: Basic display lists that allow selection and highlighting of spatial object views (soviews). can use style factory to avoid repeated setting of glColor glPointSize etc.
- **easy2D** and **easy3D**: Display list interfaces with stateful colour and style parameters.

### 2.5.2 Grouping tableaux

- **polytab**: Draw several tableaus in order and toggle them on/off. This is a "stack of acetates" model, like the Parmesan StackManager.

### 2.5.3 Coordinate system changes

- **manipulator**: Combined 2D/3D zooming and panning.
- **blackbox**: Event record and playback.

### 2.5.4 Debugging

- **tview**: Tableau graph viewer. Click on tableau icon to get its type/name.
- **event_echo**: Display all events as they pass through the tableau.

# Chapter 3

# Adaptors

An adaptor can be written for any GUI toolkit which has the ability to display an OpenGL area, for example, Gimp Tool Kit (GTK), Qt, Microsoft Foundation Classes (MFC).

## 3.1 Adaptors provided with VGUI

For some of the more commonly used GUI toolkits VGUI provides an easy way to create menus, dialogue boxes and add tableaux. Using this VGUI pattern to create your applications also means that they are easily ported from one GUI toolkit to another. Thus it is possible to compile the same application to run with a different GUI on different platforms.

GUI toolkits that are supported or are intended to be supported in the near future are:

- **GTK** (GIMP Toolkit):
  Information at **http://www.gtk.org/**
- **FLTK** (Fast Light Toolkit)
  Information at **http://fltk.easysw.com/**
- **Motif**
- **Qt**
  Information at **http://www.troll.no/developer/**
- **GLUT** (OpenGL Utility Toolkit)
  Information at **http://www.opengl.org/Documentation/GLUT.html**
- **glX**

- **MFC** (Microsoft Foundation Classes)

## 3.2 Using a particular GUI toolkit

If you write your code using the pattern provided in VGUI then it will run on any supported GUI toolkit.

If you wish to run your application using a particular GUI toolkit, Foo say, then you will need to satisfy these conditions:

- There will need to exist a specialisation of vgui for Foo (usually called `vgui_Foo`).
- You will have to have Foo installed on your machine.
- You will have to configure Target to know where Foo is installed.

If Foo satisfies these conditions then when `vgui_Foo` is compiled, Foo will be registered as available for use by your applications.

If no preference given on the command line for a GUI toolkit then application will look for a suitable GUI toolkit that is registered. If more than one GUI toolkit is registered then one will be selected at random.

If you wish to specify a GUI toolkit (for example Foo) then you will need to add `--factory=Foo` to the end of the command line when you run your application.

## 3.3 Adding a new GUI toolkit

One of the key ideas of VGUI is that it can be used with any widget set or user interface that can embed a GL drawing area. If the GUI toolkit you wish to use is not supported by VGUI then you can either: just go ahead and write your application in that that toolkit, or provide a new class derived from vgui that interfaces your toolkit to VGUI. The advantage of this second option is that the code you write will be portable to any other supported toolkit and you will be able to compile other people's VGUI applications on your favourite GUI toolkit.

In order to port VGUI to a new interface there are several areas where the underlying interface must implement VGUI functionality. These are:

1. Events for the OpenGL drawing area must be translated into `vgui_event`'s that will be used by the tableaux for that drawing area. This will be done by a subclass of `vgui_adaptor` e.g. `vgui_Xm_adaptor`.

2. Functions for manipulating a window should be provided. This will normally be done by a subclass of `vgui_window` e.g. `vgui_Xm_window`.
3. VGUI functions for controlling the event loop should be implemented. These functions include running the event loop, running one event at a time, flushing the event loop and adding user produced events. This will be provided in a class derived from `vgui` e.g. `vgui_Xm`. The vgui class also has factory methods for producing `vgui_windows` and `vgui_dialog_impls` that will need to be overriden
4. If the interface supports menus then it should be able to build a menu bar or popup menu from the `vgui_menu` description. Popup menus are handled in the `vgui_adaptor` subclass while a menubar is handled in the `vgui_window` subclass.
5. If the interface has text entry fields, option menus and checkboxes then it should be able to build a popup dialog from a `vgui_dialog` description. The class that provides the base for dialog implementations is `vgui_dialog_impl`.

Lets suppose that the GUI toolkit you want to use is the (fictitious) "Foo" toolkit. The following is a step-by-step guide to all the classes and functions you will need to make in order to implement VGUI in the new interface system.

### 3.3.1 vgui_Foo_adaptor

Note that this is the only class that you have to implement in order to use VGUI tableaux. The other classes are needed if you want to use the VGUI pattern to create menus and dialog boxes.

This class is derived from `vgui_adaptor`. It translates events for the OpenGL drawing area into `vgui_events`. The most important functions to provide are :

- a function or functions dealing with mapping events from Foo's event system into `vgui_events` e.g. in FLTK an `Fl_Gl_Window` has a single handle method that can be overriden while in Motif several callback functions must be provided for the drawing widget including exposure, resize and input.

  NOTE. If "Foo" is some interface derived from X such as Motif or the Athena widget set then the functions in `vgui_X/vgui_X_utils` may be useful for converting between XEvents and `vgui_events`.

  Once a `vgui_event` has been constructed it should be sent to the non-virtual `vgui_adaptor` function `::dispatch`.
- `::make_current`: Override the `vgui_adaptor` function `::make_current`.
- `::swap_buffers`: Override the `vgui_adaptor` function `::swap_buffers`.

- `::update`: Override the `vgui_adaptor::update` method so that it looks like:

```
void vgui_Foo_adaptor::update(vgui_message const &e)
{
  if (e.user == vgui_tableau::msg_redraw)
    foo_draw();
  else
    vgui_adaptor::update(e);
}
```

foo_draw could either be a member function of Foo's OpenGL widget that needs overriding like FLTK's `Fl_Gl_Window::draw` method or just a convenient function like `vgui_Xm_adaptor::draw`.

```
void vgui_Xm_adaptor::draw()
{
  make_current();

  vgui_tableau *tableau = get_tableau();

  glDrawBuffer(GL_BACK);

  vgui_matrix_state::clear_gl_matrices();
  if (tableau)
  {
    do_clear();

    vgui_event e;
    e.type = vgui_DRAW;
    dispatch(e);

    swap_buffers();
  }
}
```

### 3.3.2 vgui_Foo_window

This is the next class to implement and the amount of work is very dependant on the interface system you are using. Later a menubar widget can be added to this class but to start with the only features that need to be implemented are embedding the vgui_Foo_adaptor widget and getting the window to display.

### 3.3.3 vgui_Foo

There may be some setup for the widget system that should take place in this class in the vgui::init_impl function. The only other features to worry about are providing the singleton methods/data

from `vgui_Xm.h`

```
class vgui_Xm : public vgui
```

```
{
public:
  static vgui_Xm* instance();

protected:
  vgui_Xm();
  static vgui_Xm* instance_;
};
```

from `vgui_Xm.C`

```
vgui_Xm* vgui_Xm::instance_ = 0;

static vgui_Xm* vgui_Xm_instantiation = vgui_Xm::instance();

vgui_Xm* vgui_Xm::instance()
{
  if (instance_ == 0)
    instance_ = new vgui_Xm;

  return instance_;
}
```

The line containing vgui_Xm_instantiation causes the (sole) instance of vgui_Xm to be created and due to the ctor vgui::vgui being called, vgui_Xm is then registered with VGUI under the name given in the virtual function `::name` e.g. `string vgui_Fl::name() const { return "fltk"; }`

The other important function to override is `vgui::run_impl` which basically calls the main event loop for the interface e.g.

```
void vgui_Xm::run_impl()
{
  XtAppMainLoop(app_context);
}

void vgui_Fl::run=_impl()
{
  Fl::run();
}
```

The other functions to override are the factory methods returning vgui_window subclasses of the correct type. e.g.

```
vgui_window* vgui_Fl::produce_window_impl(int width, int height, char const* title)
{
  return new vgui_Fl_window(width, height, title);
}
```

### 3.3.4 Supporting menu bars

### 3.3.5 Supporting pop-up menus

### 3.3.6 Supporting dialog boxes