

DGL – The Diverse interface to OpenGL

Users/Developers Guide

By

Andrew A. Ray

Outline:

- 1) Introduction
- 2) Installation and configuration
- 3) DGL Usage guide
 - a) DSO listing
- 4) Porting guide and related API references
 - a) C callbacks
 - b) C++ interface
- 5) Explanation of the DGL run loop
- 6) Building an application with DGL
- 7) Toolkits supported by DGL
- 8) Explanation of the DGL Display System
 - a) Conceptual overview
 - b) API documentation
- 9) Textures and shared context support
 - a) Programmer explanation
 - b) Display designer explanation
- 10) Keyboard and mouse support
- 11) DGL Class API reference

Introduction

This is a toolkit that extends OpenGL to allow the user to develop applications that utilize raw OpenGL with all of the advantages that the DIVERSE API provides. With this toolkit navigation is provided by default, displays are interchangeable, and it is simple to port GLUT and other OpenGL programs. In short, if an application needs to run in a virtual environment, DGL may be the tool that you have been looking for.

DGL is a C++ toolkit that is based on top of DTK, Producer, and Open Threads. DGL runs on Linux, Mac OS X, IRIX, and Windows via Cygwin. It also can be used with the Diverse Adaptable Display System to power a CAVE™ type system.

Because this toolkit supports raw OpenGL there are also a number of different graphical toolkits that generate OpenGL that can be used. Currently Open Scene Graph, VTK, and Coin are currently the toolkits supported by DGL. In the future OpenVRML and OpenSG may be supported. Supported does not just mean you can take the toolkit and figure out how to make it work with DGL, we mean that you can use a predefined intuitive interface that allows you to take your pre-existing code and simply add it to a runtime that DIVERSE knows how to handle. For example, porting an Open Scene Graph application is as simple as adding the top node in your application to the root node that DIVERSE provides. Everything else necessary for the toolkit to work is done by DIVERSE. The purpose of DGL is to provide an environment to get work done. This applies to both developers and users. For users you get the simplistic experience that DIVERSE provides. This allows you to focus on what you want to do, not worry about how to make it work. For developers this means that you concentrate on the application that you are working on, not how to interface with X environment.

This toolkit was originally developed in a very short span of time and leverages several different technologies to help with its cross platform nature. Producer provides a set of cross platform display classes which allow for much easier cross platform development. Open Threads is a requirement for Producer. The Open Threads package

allows for simple cross platform threading.

The design goals for DGL were that it would just work, be simplistic, yet powerful, and not to get in the users way. DGL was originally started by Chris Logie, and the concept was turned into DIVERSE 3 by Daniel Larimer, Andrew Ray, Patrick Shinpaugh, and Mike Shore. This project died due to lack of support. From this project DPFGL was created by Andrew Ray to provide limited OpenGL support for DIVERSE users. Currently Andrew Ray has written the DGL system described in this document. Currently DGL is a cross-platform release that will in time grow mature and offer many of the same options that DPF provides.

All of the DIVERSE / DGL development has been funded by Dr. Ron Kriz of Virginia Tech through several different grants from both government and corporate sources. The full history of DIVERSE and list of contributors is available at diverse.sourceforge.net.

Installation and configuration guide:

Required software before install:

DTK 2.4.8 beta or later (available on diverse.sourceforge.net)

Producer (available on diverse.sourceforge.net)

Open Threads (available on diverse.sourceforge.net)

Optional recommended software version levels:

VTK 4.2

OpenScenGraph 0.96

Coin 2.1.0

Installation instructions for the impatient:

```
./configure
```

```
make
```

```
make install
```

Verbose section:

Building and installing DGL is similar to most Linux software. There is a standard configure script in the base directory of DGL and takes the following options:

```
--with-vtk [Build VTK support for DGL]
--with-coin [Build Coin support for DGL]
--with-osg [Build Open Scene Graph support for DGL]
--prefix=/your/directory/here [Install DGL into a different directory]
```

The default installation directory is `/usr/local`

To run the configure script simply type `./configure`. If you want to add different options run it by adding your options to the end of it. I.e. `./configure --with-vtk --with-coin`

After DGL is configured you need to build it. To do this simply type `make`. After this switch to root (unless you configured it to install to a different directory that does not require root privileges) and type `make install`.

The last step in a DGL installation is to add the `dgl-config` variable to your `PATH` variable. If your path includes `/usr/local/bin` then this step is not necessary. If you installed DGL to a different location or `/usr/local/bin` is not in your path then this step is necessary. The easiest way to do this is done in your shell's configuration file. To do this with bash edit `~/.bashrc` and add `export PATH=/usr/local/bin:$PATH` to the end of the file.

IRIX installation instructions

IRIX is a very fickle operating system and does not perform well with the GNU auto tools at times. Because of this you may run into some problems with it. To combat these problems this section is here to provide information from known issues and solutions.

One of the main problems with IRIX is multiple binary types. The standard is

n32, but there is a 64 bit option you may want to use. In order to facilitate these different binary types the configure flags --enable-n32 and --enable-64 have been added. These flags force the usage of the IRIX compiler, not the GNU compiler. Only one of these flags can be specified at configure time. If you wish to have both binary types installed then you must configure, build, and install DGL into two different locations. For example the following commands will build and install DGL in two different locations:

```
./configure --enable-n32 --prefix=/usr/local/diversen32
make
make install
./configure --enable-64 --prefix=/usr/local/diverse64
```

If you build DGL successfully but get an error on the make install about not able to find libdgl.so then you will need to copy the libtool script from the DTK source directory into the DGL source directory, type make clean, and then make, make install, and it should fix that problem.

If you run into a problem with install.sh during make install then you will need to specify where the INSTALL variable before you type ./configure. For example:

```
INSTALL="$PWD/install-sh -c" ./configure --enable-n32
```

If all else fails then run the bootstrap script which will refresh all of the autotools files in the project. This should make everything work. You may have to copy the libtool script over from dtk if this command is run though. For some reason the GNU libtool program doesn't work properly.

Usage guide

The usage of DGL is meant to be as unobtrusive as possible. You simply build and run your applications as you would normally. In order to configure the application you simply set environmental variables in a terminal, or you can set up a script file to use these environmental variables as usual.

The main environmental variable that DGL uses is DGL_DSO_FILES. This environmental variable causes a DGL program to load DSO's (sometimes known as plugins) into the DGL runtime. The DSO's can add a wide range of functionality to your application. Simply by using DSO's you can change what display environment your application uses, what navigations your application uses, and add completely different functionality to an application.

Using DSO's is not required though. By default DGL selects intelligent defaults that provide a desktop display, keyboard, mouse, and navigation support.

Listing of DSOs provided with DGL:

desktopGroup – Provides navigation DSO's and keyboard/mouse functionality to DGL

desktopDisplay – Provides a simple display for the desktop, this display is automatically loaded if no other display is specified

desktopHyperSim – Simulate a six walled cave on your computer

desktopScreen – Use the DGL screen technology for a desktop display (provides a standard display)

desktopCaveEmulateGroup – Provides cave navigation to DGL

desktopScreenGroup – Provides a desktop screen display along with the functionality of the desktopCaveEmulateGroup.

dglKeyboardNav – Takes keyboard and mouse input from Producer and transfers it to the standard hooks in DTK. This performs the same as the DTK keyboard DSO.

vtCaveCluster* – Examples of how to use DGL in a DADS setup

vtCaveGroup – Example of how to use DGL on an SGI system to power a CAVE.

Example of how to run DGL applications with DSO's added:

Suppose you have an application called helix. After you build the helix application you run it by typing `./helix` in the source directory. By default DGL will provide a desktop display, keyboard support, mouse support, and a mouse based navigation that is identical what DPF provides.

If you want to test how your application would run in a six walled cave on your desktop you could do this by running the following command:

```
export DGL_DSO_FILES=desktopHypersim:desktopGroup  
./helix
```

If you wanted to power your application with the DTK Cave Device Simulator while using the hyper-sim you could type the following:

```
export DGL_DSO_FILES=desktopCaveEmulateGroup:desktopHyperSim  
./helix
```

Porting guide:

This is a guide to help you move your existing OpenGL code to DGL. This guide is composed of two parts. One is how to use existing C style callbacks and the other is how to use the C++ interface to DGL. The C callbacks are useful for glut style programs and the C++ interface is a way to harness the power of C++ in your program.

C style callbacks

There are two types of C callbacks in DGL. One set deals with controlling the DGL runtime and the other set deals with setting the callbacks that DGL calls within the runtime loop. The DGL runtime takes the standard frame by frame graphics approach and has hooks for flexibility.

Runtime functions:

void dglInit();

Starts the DGL system, must be called once before any other DGL calls.

void dglStart();

Configures and draws the first frame, must be called once

bool dglIsRunning();

Tells you if the program is running. Useful for main loop control.

void dglRun();

Take over the run time loop and execute your program.

void dglFrame();

Performs the preFrame, frame, and postFrame callbacks. Essentially advances DGL one frame.

void dglSetData(void)*

Set a void pointer that can be accessed by your callback functions

void dglGetData();*

Get a void pointer that can be accessed inside your callback functions or in your main loop.

User callback functions:

void dglPreconfigCallback(functionpointer)

Called before your application is configured. Useful for initializing variables.

void dglPostconfigCallback(functionpointer)

Called after the application is configured. Can be used to set OpenGL state and other one time graphics issues.

void dglPreNavCallback(functionpointer)

This sets a draw callback before navigation transformations are applied.

void dglDisplayCallback(functionpointer)

Sets the draw callback for your program. This is the function that should draw your OpenGL code.

void dglOverlayCallback(functionpointer)

This function is called after your draw callback is called. It is useful for making sure that a specific draw function is called after others are called.

void dglPreFrameCallback(functionpointer)

This function is called before your draw function is called.

void dglPostFrameCallback(functionpointer)

This function is called after your draw function is called. It is useful for handling input, re-calculating values, etc...

void dglInitGLCallback(functionpointer)

This function is designed for initializing your OpenGL code

void dglSetupGLCallback(functionpointer)

This function is designed for loading textures and creating display lists

In the future this interface will be expanded to include more glut style functionality. This will include timers, mouse motion, keyboard input, etc... The reason for this is because the goal of DGL is to provide a simple way to get OpenGL code into a virtual environment.

C++ interface

To use the C++ interface to DGL simply derive off of the `dglAugment` class. This class is a dual purpose class as it can be used as a simple interface to DGL and it can be

loaded into the normal DIVERSE runtime as a normal DSO.

The virtual functions that dglAugment provides are:

int preConfig

Same as dtkAugments preConfig

int postConfig

Same as dtkAugments postConfig

int preFrame

Same as dtkAugments preFrame

int postFrame

Same as dtkAugments postFrame

int draw()

The draw callback added by DGL

int initGL()

This is function designed for use with initializing your OpenGL code

int setupGL()

This is a function designed for loading textures and creating display lists

All of these functions are designed to return values similar to DTK/DPF. Usually returning 0 means success and anything else is failure. See their documentation or examples for examples of how they are used.

Constructor:

dglAugment(char* name, DGL_AUGMENT_TYPE) – Pass in the name of the augment, and optionally what type of augment you would like it to be. All augments must be uniquely named due to the requirements of DTK. The DGL augment type can be one of three different types. The current types are PRENAV, BASE, and OVERLAY. By default you get a BASE DGL augment. The PRENAV is for draw callbacks applied before the navigation transformations are applied. The OVERLAY type is used for drawing objects after everything else has been drawn. A heads up display is an example of this.

Explanation of DGL run loop:

The DGL run loop is a standard DTK/DPF run loop with a draw callback added for OpenGL. DGL provides several methods for controlling the startup sequence. C callbacks are more limited than the C++ methods when it comes to controlling the run loop.

For the C callbacks here are the methods in which you can control the runtime loop:

first

```
dglInit();  
dglStart();
```

then

```
dglRun();
```

or

```
while (dglIsRunning())  
{  
    dglFrame();  
}
```

For the C++ interface there are two main steps, configuration and then execution. There are two different ways to do these steps. The manual or the automatic approach. DGL strives to make life as simple as possible by providing easy to use functions, or to allow you as much power as you need.

Configuration:

Completely manual

```
DGL::getApp()->preConfig();  
DGL::getApp()->config();  
DGL::getApp()->postConfig();
```

Completely automatic

```
DGL::start();
```

Runtime:

Completely manual:

```
DGL::preFrame();
```

```
DGL::frame();
```

```
DGL::postFrame();
```

Completely automatic:

```
DGL::run();
```

Recommended functions for your applications:

It is recommended that you develop using DGL with the following method. The other method explained above are available for flexibility in special situations, but in general the methods below are the easiest to use and understand.

First:

```
DGL::init() or dglInit();
```

Second:

Configure the application

```
DGL::start() or dglStart();
```

Third:

Run your application. You can take control of the run time loop or you can hand over control of the runtime loop over to DGL. Both methods are described below.

If you need to control of the runtime loop:

```
while (DGL::isRunning() or dglIsRunning()) //Run while DGL has a good state
{
    DGL::frame(); or dglFrame();
}
```

If you do not need control of the run loop:

```
DGL::run(); //Run DGL
```

Access functions:

Use these functions if you need to gain access to the DGL augments at runtime.

```
vector<dglAugment*> DGL::getApp()->getStandardAugments()
```

Returns the augments called by the normal draw callback

```
vector<dglAugment*> DGL::getApp()->getOverlayAugments()
```

Returns the augments drawn after the standard draw callback

```
vector<dglAugment*> DGL::getApp()->getPreNavAugments()
```

Returns the augments drawn before the normal draw callback

Building an application with DGL:

In order to build an application with DGL simply include `<dgl.h>` in your application. That is all that is necessary to include all of the functionality of DGL. In order to link your application simply just use the `-ldgl` flag o your compiler. Unfortunately this doesn't work in the real world because of all of the different path problems on UNIX so `dgl-config` was invented.

To build your program the use of `dgl-config` is strongly recommended. `dgl-config` works by taking an argument and displaying data based on that argument. You can find out which directories you need to include in your application by calling `dgl-config --include`. The same is true for libraries(`--libs`), compiler flags (`--cflags`), version(`--version`), compiler(`--compiler`), and if the program can run (`--test`). You can also find out more information about DGL from looking at the output of `dgl-config`.

You can integrate `dgl-config` into your current makefiles or you can create new ones that use DGL. It is not recommended that you use the GNU autotools for your program unless you know what you are doing.

Toolkits supported by DGL:

Because of the OpenGL nature of several graphical toolkits it is possible to harness them inside of DGL. In order to provide the best experience for the end user (developer or user) we have tried to seamlessly integrate these graphical toolkits into DGL. We provide all of the hooks necessary to use these toolkits and do not require massive setup cost in order to use them. The three supported toolkits are listed below along with what files are necessary to include, how to add build support for them, api references, and sample code.

Usability instructions for all 3 APIS:

- 1) Initialize DGL
- 2) Initialize graphical API's adapter
- 3) Add content to world
- 4) Run application

VTK API reference:

Class DVtkRenderer:

static DVtkRenderer* New(dpf* app)

Construct a new DVtk Renderer

void AddActor(vtkProp* p)

Add an actor to the renderer.

void RotateSceneX(float deg)

rotate the scene in the x axis by the specified number of degrees

Class DVtkRenderWindow

static DVtkRenderWindow New()

Make a vtkRenderWindow

void AddRenderer(DVtkRenderer*)

Add the vtkRenderer to display in the window

Include file:

```
#include <dvtk.h>
```

Build options:

```
dgl-config --vtk-libs
```

```
dgl-config --vtk-include
```

Code example:

```
DGL::init();  
DVtkRenderer* ren1 = DvtkRenderer::new(DGL::getApp());  
DVtkRenderWindow* renWin = DVtkRenderWindow::New();  
renWin->AddRenderer(ren1);  
ren1->AddActor(youractorhere);  
ren1->RotateSceneX(90); //Flip world to DIVERSE coordinates  
DGL::start();  
DGL::run();
```

Open Scene Graph API Reference:

Class DOSG:

static void init()

Initialize the Open Scene Graph interface for DGL.

static osg::Group* getRoot()

Get the root node of the scene graph

static osg::Group* getEther()

Get the ether node of the scene graph. This is a non-navigated node.

static osg::Group* getWorld()

Get the world node of the scene graph. Children of this node will have navigation transformations performed on them.

Include file:

```
#include <dosg.h>
```

Build options:

```
dgl-config --osg-libs
dgl-config --osg-include
```

Code example:

```
DGL::init();
DOSG::init();
DOSG::getWorld()->addChild(readNodeFile("mymodel.osg"));
DGL::start();
DGL::run();
```

Coin API Reference:

Class Dcoin

static void init(dpf* app)

Initialize the Coin Interface to DGL. Requires a dpf pointer due to implementation method. An example of how to do this is listed below.

static SoGroup* getWorld()

Get the world node. Children of this node will have navigation transformations performed on them.

static SoGroup* getScene()

Get the scene node. This is the top node in the tree

static SoGroup* getEther()

Get the ether node. Children of this node do not have navigation transformations performed on them.

static SoSeparator* loadFile(const string& filename)

Load a file from disk and return a pointer to it.

Include file:

```
#include <dcoin.h>
```

Build options:

```
dgl-config --coin-libs
dgl-config --coin-include
```

Code example:

```
DGL::init();//Initialize DGL
DCoin::init();//Initialize Coin
DCoin::getWorld()->addChid(yourNodeHere);//Get the world node
DGL::start(); //Configure DGL
DGL::run(); //Run DGL
```

The DGL Display System

The DGL display system is modeled after standard graphics systems and is designed to give you a logical subdivision to the wide variety of information that is stored for each display. DGL assumes that each machine will have a certain number of pipes. Inside of each of these pipes there will be a certain number of windows (render surfaces), each of which has screens that can have multiple view ports built into them. Each part of the display subsystem has specific information about its specialty and is linked to the other parts of the display sub system. What this allows for is the ability to query the number of pipes available in each system and then go through each pipe and then access information all the way down to the viewports. This can be a somewhat tedious process, but it provides a logical decomposition to the large amount of information available.

Because DGL is based on top of Producer there are several producer classes that are used throughout the DGL display system. These components are publicly accessible so that experienced users can leverage the Producer API however they wish.

Programmer section:

This is a section that goes over the API that is provided with DGL. It is broken down from how to get a display object, then cascades down the long chain of the DGL display subsystem.

DGLPipe functions:

This class is just a simple wrapper that provides the concept of pipes on a computer. Below are the C++ functions that this class provides.

int GetNum() – Get the number of the pipe

int getNumWindows() – Get the number of windows in the pipe

DGLWindow* getWindow(int) – Get the specified window object pointer

void addWindow(DGLWindow* window) – Add a window to the pipe

DGLScreen functions:

This class deals with the ability to rotate a window to a specified offset, deal with interocular settings, near/far clipping planes, and the width / height issues. Some of these are VE abstractions on top of a RenderSurface so it was decided to separate them from the DGLWindow class. Below are the C++ functions that this class provides.

int addViewPort(DGLViewPort*) – Add a viewport to the screen

DGLViewPort* getViewport(int) – Get a specific viewport

int getNumViewports() – Get the number of viewports in the screen

int setNear(float) – Set the near clipping plane

float getNear() – Get the near clipping plane

int setFar(float) – Set the far clipping plane

float getFar() – Get the far clipping plane

int setOffset(dtkCoord) – Set the offset for the screen

dtkCoord getOffset() – Get the offset for the screen

int setFov(float) – Set the field of view

float getFov() – Get the field of view

DGLWindow* getWindow() – Returns the window the screen is associated to

int setWidth(float) – Set the width of the screen

float getWidth() – Get the width of the screen

int setHeight(float) – Set the height of the screen

float getHeight() – Get the height of the screen

void frame() – Update the screen

DGLWindow functions:

This class is primarily a wrapper on top of a Producer RenderSurface. It provides an abstraction on top of it while allowing for easily sharing contexts and allowing direct access to it if it is warranted. Below are the C++ functions that this class provides.

`int setPipe(bool)` – Set which pipe this window uses.

`int getNumScreens()` – Get the number of screens under this window.

`int addScreen(DGLScreen*)` – Add a screen to the window.

`int setStereo(bool)` – Tell the render surface to do stereo.

`bool getStereo()` – Is the window using stereo or not?

`int setFullScreen(bool)` – Set fullscreen mode ?

`bool getFullScreen()` – Is the window fullscreened?

`void setBorder(bool)` – Set if there will be a window border.

`bool getBorder()` – Is there a border on the window?

`int setCursor(bool)` – Set whether or not the cursor will appear over the display.

`bool getCursor()` – Does the mouse appear over the display?

`int setResizable(bool)` – Set the window so it can/cannot resize.

`bool getResizable()` – Is the window resizable?

`int setDepth(int)` – Set the bit depth producer uses.

`int getDepth()` – Get the bit depth used.

`int setWindowDimensions(int left,int bottom, int width, int height)` – Set the window dimensions.

`int realize()` – Create the window.

`int setName(char*)` – Name the window.

`char* getName()` – Get the name of the window.

`Producer::RenderSurface* getRenderSurface()` – Get the render surface that is used to display information.

`void updateKM()` – Update the KeyboardMouseCallback option.

`int setKMCallback(Producer::KeyboardMouseCallback*)` – Set the callback that is used to handle keyboard and mouse feedback.

`bool getSharedParent()` – Is this window a shared parent?

`int setSharedParent(bool)` – Used to make this window a shared parent.

bool getSharedChild() – Is this window a shared child window?
int setSharedChildWindow(DGLWindow*) – Set the parent window for the child
DGLWindow *getSharedParentWindow() – Get the parent window of this object.

DGLViewport:

This class sets whether or not screen or perspective technology is used. Perspective is for a desktop, screens are used with non-desktop type displays. Below are the functions for the class.

int setGeometry(float x, float y, float w, float h) – Set the geometry of the viewport
int getGeometry(float& x, float& y, float& w, float& h) – Get the viewport data
int setScreen(DGLScreen*) – Set the screen used to display on
DGLScreen* getScreen() – Get the screen to display on
int setType(SCREEN||PERSPECTIVE) – Set the type of screen to use
int create() – Create the viewport
void frame() – Update the viewport

Display designer section:

This is a section that explains how to make your own displays with DGL. The first question that needs to be asked when making a display is which pipes are the application going to use? Next how many windows are going to be inside each pipe, are they going to be full screen, use stereo, and do they share contexts? After this you need to decide on whether or not you are going to use a perspective or a screen type of display. You then will need to add in information such as inter-ocular distance, and many other options explained later. The API reference above will come in handy when creating your own display.

First obtain a DGLDisplay object by calling DGL::getApp()->display(). Next you will need to call the addPipe function on the display object and tell it what pipe to use. Next get a pointer to the pipe from the display object. After this create a DGLWindow object. Configure it to your specifications then set up the associations between pipe and

window. Next create a DGLScreen object and set its properties. Associate the screen and window to each other. Lastly create a DGLViewport object, set its type(Perspective or Screen), and associate it with the previously created screen object.

Also of note is the ability to share contexts between displays. This can save some precious resources if used properly. It is not on by default except with the desktopHyperSim, but can be added to different displays if need be. This is described with more detail in the next section.

Texture and shared context support

As explained earlier the DGL display system supports sharing of contexts so that textures and display lists can be reused between different displays. What this means to a programmer is that they don't have to load a texture for each display that DGL is using, or even have to worry about the displays used in DGL when using textures. DGL accomplishes this by the use of parent / child displays explained earlier in the document. In order for the programmer to harness the power of these sections two specific callback functions must be used. These are the setup and init callbacks. The setup callback is used for loading textures and creating display lists while the init callback is used for binding textures and setting up your OpenGL environment. These callbacks are called once before the draw callback in your application is called. If the need arises to call these functions again (in case you want to dynamically load textures or create display lists) you can tell DGL to call the setup or init callbacks at the proper time.

NOTE: DGL calls setupGL then initGL once by default then will not call them unless you tell DGL to call them. After you tell DGL to call them it will and then will not call them until you tell DGL to call them again. These callbacks are not meant to be called each frame, unless there is a very specific goal that you are looking to accomplish.

API explanation:

As stated earlier in the document DGL has functions available to do loading/initialization functions. Control for these functions being called is in the main DGL class. The callbacks are either part of a DGL augment or are a C style callback.

Functions for controlling the initialization callback:

int DGL::getApp()->setInitGL(bool value)

bool DGL::getApp()->getInitGL()

Functions for controlling the setup callback:

int DGL::getApp()->setSetupGL(bool value)

bool DGL::getApp()->getSetupGL()

If the value passed to either init or setup GL is true then before the next frame is drawn DGL will call the appropriate callback functions.

C Interface functions:

void dglInitGLCallback(functionpointer)

void dglSetupGLCallback(functionpointer)

As stated earlier these two functions allow for C Callbacks for the setup and init GL function

C++ Interface functions:

These functions are part of the DGL Augment class. They are called for each augment when DGL is told to call either the set or init callbacks. They are simply called initGL or setupGL. Derive from the augment class and override these functions and DGL will take care of the rest for you.

Display designer consideration:

In order to effectively use the setup and init GL functions the displays have to be properly shared. This is done by having one window be designated as the parent window and all of the other windows need to be designated as child windows. Once this is done DGL takes care of the rest.

Keyboard and Mouse Support

This section is designed to explain the differences in keyboard and mouse input between DTK/DPF and DGL, explain how to access this information, and how to extend

keyboard/mouse support if what is currently there is not flexible enough for you.

At the time DTK/DPF were created it did not make sense to have a VR API supported on non Linux/IRIX platforms. Graphics support and VE support was simply not there at this time. The only feasible platform to run a VE was an SGI IRIX system and UNIX type machines were used for desktop development. This situation has changed dramatically since the creation of DIVERSE. This bit of history is included to explain why the handling for keyboard and mouse input has been changed with DGL. Because of the UNIX style background of DTK/DPF, these two API's depend on X11 to make them work properly. This does not work well with non-UNIX platforms. Thankfully Producer handles all of these problems by abstracting them away. By using Producer one can get keyboard and mouse input in a consistent manner across platforms.

Keyboard input howto:

Keyboard information for DGL can be obtained by reading from a shared memory segment named keyboard. This segment is the size of two characters, and is in the format of [plr] [character]. P stands for pressed, r stands for released. The only caveat is that only one character is written each frame so it can be quite easy to miss characters if a user bangs on the keyboard.

Mouse input howto:

Mouse input can be obtained one of two different ways. The first way is to use a `dtkInValuator` with the name `pointer` to get the X/Y information of the mouse, a `dtkInButton` with the name `button` will allow you to get the mouse clicks. The second way is to use a shared memory segment named `mouse` of size two floats and a shared memory segment named `mousebutton` of size `char + int` to read information about the buttons. The `mouse` shared memory segment simply holds the xy coordinates of the mouse. The `mousebutton` shared memory operates like the keyboard shared memory segment in the previous paragraph except it contains which button was either pressed or released.

Extending DGL to provide more than it already does:

In order to extend the keyboard/ mouse functionality of DGL it is necessary to derive off of the KeyboardMouseCallback class provided with Producer. Once this is done then you need to call the setKMCallback method in the DGLWindow class before the config step in the program.

DGL Class API Reference:

DGL solves several different problems, and because of this it has several functions designed for utility and control. Some of the base DGL functions have been explained before but the DGL class itself has not been discussed. The DGL class is a singleton and is static. You initialize it by calling `DGL::init()` and can get a pointer to it by typing `DGL::getApp()`. This is all that is necessary to start using the DGL class. Below are some of the functions that the DGL class provides.

NOTE: The DGL class is derived off of the `dtkManager` class. This provides several important DTK functions.

`int addPreNavAugment(dglAugment*)` – Add a pre-nav object

`int removePreNavAugment(dglAugment*)` – Remove a pre-nav object

`int addAugment(dglAugment*)` – Add an augment

`int removeAugment(dglAugment*)` – Add an augment

`int addOverlay(dglAugment*)` – Add an overlay augment

`int removeOverlay(dglAugment*)` – Removes an overlay augment

`vector<dglAugment*> getStandardAugments()` – Get the standard augments

`vector<dglAugment*> getOverlayAugments()` – Get the overlay augments

`vector<dglAugment*> getPreNavAugments()` – Get the pre-nav augments

`int setHeadTracking(bool value)` – Use or do not use head tracking

`bool getHeadTracking()` – Get the current status of the head tracker

`int setSetupGL(bool)` – Turn on / off the setupGL callback

`bool getSetupGL()` – Return the status flag for the setupGL callback

`int setInitGL(bool)` – Turn on / off the initGL callback

bool getInitGL() – Get the status flag for the initGL callback

int setDiverseCoord(bool) – Use the DIVERSE coordinate system or not (default = yes)

bool getDiverseCoord() - Check if the DIVERSE coordinate system is used

int setOrigin(dtkCoord) – Set the origin

dtkCoord getOrigin() – Get the origin

float getScale() – Get the scale

int setScale(float) – Set the scale

int dglToWorld(dtkCoord&) – convert a set of DGL coordinates to world coordinates

int WorldToDgl(dtkCoord&) – convert a set of world coordinates to DGL coordinates

NOTE: These functions mimic the DPF functions and operate identically. Written by

John Kelso

int lock() – Lock the DGL class using an open threads mutex

int unlock() – Unlock the DGL class using an open threads mutex