DPFGL – The Diverse graphics interface to opengl PerFormer for openGL

Outline:

1) Introduction
2) Installation and configuration
3) Porting guide and api references
   a) C callbacks
   b) C++ interface
4) Explanation of the DPFGL run loop
5) Building an application with DPFGL
6) Toolkits supported by DPFGL
7) Example programs supplied with DPFGL

Introduction

This is an a toolkit that extends DPF 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.

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 DPFGL. 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 DPFGL, 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 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 DPFGL 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.

Note: In most of the software the PF has been dropped from DPFGL so it can be shortened to DGL. DGL is easier to type and represents what this software package really provides, OpenGL support. In the future the same interface may be applied to a different backend for OpenGL rendering so that a proprietary package will not be required. There is no timeframe for this right now due to funding issues.

Installation and configuration guide:

Required software before install:

DTK 2.3.2

DPF 2.3.2

Optional recommended software version levels:

VTK 4.2

OpenScenGraph 0.96

Coin 2.1.0

For the impatient:

./configure

make

make install

Verbose section:

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

--with-vtk -Build VTK support for DPFGL

--with-coin –Build Coin support for DPFGL

--with-osg - Build Open Scene Graph support for DGLPF.

--prefix=/your/directory/here –Install DPFGL into a different directory

The default installation directory is /usr/local/diverse/dgl-X.Y.Z

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 DPFGL 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 DPFGL installation is to add the dgl-config variable to your PATH variable. 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/diverse/dgl-X.Y.Z/bin:$PATH to the end of the file(where X,Y,Z are the version numbers DPFGL).

Another step you may want to take is to create a symlink from dgl-X.Y.Z to dgl in /usr/local/diverse or your installation location. Then include the symlink in your shell's configuration file. This way you will not have to change your shell's configuration file

each time you install a different version of DPFGL.

Porting guide:

This is a guide that 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 DPFGL calls.

Runtime functions:

void dglInit() - Starts the DPFGL system, must be called before any other DPFGL calls.

void dglStart() - Configures and draws the first frame

bool dglIsRunning()- Tells you if the program is running. Useful for loop control.

void dglRun() - Take over the run time loop and execute your program.

void dglFrame() - Performs the preFrame, frame, and postFrame callbacks. Essentially advance DPFGL one frame.

void* dglSetData()-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...

In the future this interface will expanded to include more glut style functionality. This will include timers, mouse motion, keyboard input, etc...

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 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

All of these functions are designed to return values similar to DTK/DPF. See their documentation or examples on how they are used.

Constructor:

dglAugment(dpf* app, char* name, DGL_AUGMENT_TYPE) – Pass in the dpf app that will be used with DPFGL system and the name of the augment. All augments must be uniquely named due to the requirements of DTK. The last argument is the type of DGL augment you want it to be. 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.

Explanation of DPFGL run loop:

The DPFGL run loop is a standard DTK/DPF run loop with a draw callback added for OpenGL. DPFGL 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:

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

then

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

For the C++ interface there are two main steps, configuration and then execution. Here are the different methods in order to do this.

Configuration:

Completely manual

```
DGL::preConfig();
DGL::postConfig();
```

Completely automatic

```
DGL::config();
```

Runtime:

Completely manual:

```
DGL::preFrame();
DGL::draw();
DGL::postFrame();
```

Completely automatic:

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

Hybrid functions:

DGL::start() //Configures and draws the first frame of the DPFGL application
DGL::run() //Takes over the runtime loop and runs the DPFGL system.

Recommended functions for your applications:

It is recommended that you use DPFGL in the following way. The other ways 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 DGL::init(dpf* app) -Initialize DPFGL to run by itself or inside of an existing DPF application.
Second:
If you need control of the run loop:

```
DGL::config(); //Configure DPFGL
while (DGL::isRunning()) //Run while DPFGL has a good state
{
	DGL::frame();
}
```

If you do not need control of the run loop:

```
DGL::start(); //Configure DPFGL and render the first frame

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

Access functions:

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

vector<dglAugment*> getStandardAugments() - Returns the augments called by the normal draw callback

vector<dglAugment*> getOverlayAugments() - Returns the augments drawn after the standard draw callback

vector<dglAugment*> getPreNavAugments() - Returns the augments drawn before the normal draw callback

Building an application with DPFGL:

In order to build an application with DPFGL simply include <dgl.h> in your application. That is all that is necessary to include all of the functionality of DPFGL.

To build your program the use of dgl-config is strongly recommended. Examples on its usage are listed in the examples directory of DPFGL. 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).

Toolkits supported by DPFGL:

Because of the OpenGL nature of several graphical toolkits it is possible to harness them inside of DPFGL. In order to provide the best experience for the end user (developer or user) we have tried to seamlessly integrate these graphical toolkits into DPFGL. 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.

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

Open Scene Graph:

API Reference:

Class DOSG:

static void init(dpf* app) – Initialize the Open Scene Graph interface for DPFGL. Requires a dpf pointer due to implementation method. An example of how to do this is listed below.

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

Coin:

API Reference:

Class Dcoin

static void init(dpf* app) - Initialize the Coin Interface to DPFGL. 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

Usability instructions:
1) Initialize DGL
2) Initialize DCoin
3) Add coin content to world
4) Run application

Code example:
DGL::init();//Initialize DGL

DCoin::init(DGL::getApp())//Initialize Coin

DCoin::getWorld()->addChid(yourNodeHere);//Get the world node

DGL::config(); //Configure DPFGL

DGL::run(); //Run DPFGL

Example programs supplied with DPFGL:

There are several example programs supplied with DPFGL and they illustrate both the build process and how to use different features of DPFGL. Below is a list of examples and what they do. All of the examples are available in the examples directory off of the DPFGL root directory.

CCallbackHelix: This is a simple double helix written with OpenGL that uses the C callbacks for interface with DPFDGL.

banner: A picture of a mountain that acts like a flag in the wind. This example also uses the C callbacks in DPFDGL. This example is from Nehe's openGL site.

coinfly: This is a program that takes one argument, that argument is the file to load and display in the virtual world. This program uses the Coin interface for DPFGL and requires that Coin be installed in order to work properly.

dosgfly: This program uses the Open Scene Graph interface to DPFGL in order to

provide model loading. It takes one command line parameter and loads that model and lets you view it in a virtual world.

genericDglAugment: This program is a skeleton example of how to create your own dglAugment. It does nothing, but can act as a starting point for your code.

helixDglOverlayAugment: This example shows how you can use the C++ interface for the helix example listed above and make it an overlay that is drawn after the regular draw callback. You can run this example by exporting DPF_DSO_FILES=desktopGroup:helix and then running diversifly NULL.

helixDglAugment: This example shows how you can use the C++ interface for the helix example listed above. You can run this example by exporting DPF_DSO_FILES=desktopGroup:helix and then running diversifly NULL

helixDglPreNavAugment: This example shows how you can use the C++ interface for the helix example listed above except the helix is drawn before navigation is applied. You can run this example by exporting DPF_DSO_FILES=desktopGroup:helix and then running diversifly NULL

quad: This program uses the VTK interface that DPFGL provides. It is an example adapted from the VTK distribution.

structure: This is an openGL wire frame of a block. Example written by Stephen Manley.