OpenGL based fullscreen feature projector

December 4, 2013

Abstract

This report demostrates the standard OpenGL graphics pipeline through an example of a fullscreen checkerboard feature projector. This application was successfully deployed on distributed system for rendering on multiple workstations using *Chromium* framework. This work is a part of development done for indigenously developed seamless multiprojector display.

1 Problem statement

Develop an application which can be used for projecting checkerboard features onto multiple projectors one at a time. Application should allow for remote interactive control of projection.

2 Motivation

A multiprojector feature projector application was required for serially registering the projectors with respect to a common camera. This information is the basis for computing final vertex-texture mapping, chromium tile configuration and edge blending maps.

3 Introduction

Approach used for geometric alignment of multiprojector arrangement required registration of individual projector planes with respect to camera plane. Here, *registration* means to determine mapping between projector and camera pixels which can be used for computing mutually geometric aligned configuration for each individual projector.
In this section, we will first look at complete structure of the program(through a look at its *main() function*). Listing 1 will show the complete pipeline of operations(at function call level) performed in a typical OpenGL application with window(hence using GLUT as well) interface. As a small introduction to GLUT, it should be highlighted that OpenGL does not have any provision

for user interface, therefore OpenGL Graphics Utility(GLUT) was developed to provide support for window creation, manipulation, accessing input from mouse, keyboard, pop-up menu creation. Further, GLUT also provides routines for drawing primitives such as cubes, sphere, Utah teaspot etc. which are at higher abstraction level than the once provided by OpenGL. GLUT windows contain OpenGL contexts. Due to its licence limitations, modification and subsequent distribution of GLUT source was illegal. Therefore, *freeglut* was developed to be identical to GLUT and still provide possiblity to modify and then distribute the source.

Listing 1: Typical structure of GLUT based OpenGL application

```
//Initializing GLUT
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
glutInitWindowSize(proj_imagewidth*num_of_proj_in_row,
proj_imageheight*num_of_proj_in_col);
glutInitWindowSize(1024,768);
glutInitWindowPosition(0, 0);
glutCreateWindow(argv[0]);
glutKeyboardFunc(keyboard);
glutFullScreen();


//Initializing OpenGL
init();

//Registering OpenGL functions to GLUT
glutDisplayFunc(display);
glutReshapeFunc(reshape);


//Rendering starts...
glutMainLoop();
```

Now we will discuss the individual function calls in detail to help us understand the *typical* pipeline of operations performed in an interactive OpenGL application using GLUT.

3.1 Initializing GLUT

Listing 2 shows the function call for initializing GLUT library. This call will also negotiate a session with the windowing system. This function will return an error if it fails to connect with windowing system, or windowing system support for OpenGL is absent or if command line arguments are invalid. Typically, it requires *GLX*OpenGL Extenstion to X Window System library to be present

on the system to support windowing for OpenGL applications. In short, GLX is the interface to X-Window System for OpenGL applications.

Listing 2: Initializing GLUT

```
glutInit(&argc, argv);
```

3.2 Initializing Display mode

Display mode is used while creating and configuring window on which OpenGL application will run. It has window configuration options such as setting number of buffers for rendering on window, providing accumulation buffer, alpha component to displayed image etc. Output of OpenGL application will be mapped to this window. It determines the OpenGL display mode for rendering on the window to be created. Listing 3 shows the relevant function call.

Listing 3: Initializing OpenGL display mode

```
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
```

Here, we would like to elaborate on various buffers available in OpenGL. These buffers are enables during display mode configuration.

1. Accumulation buffer TO BE DONE??
2. Depth buffer
3. Stencil buffer

3.3 Creating and configuring a X-window

Listing 4 shows how we can create and configure X-window on which OpenGL application output is mapped.

Listing 4: Creating and configuring window

```
// typical window
glutInitWindowSize(1024,768);
glutInitWindowPosition(0, 0);
glutCreateWindow(argv[0]);

// or in fullscreen mode
glutCreateWindow(argv[0]);
glutFullScreen();
```

3.4 OpenGL initialization

Before describing specific details of OpenGL pipeline initialization, it is required to first understand the pipeline itself. So, here we first describe the standard OpenGL pipeline, followed by description of its initilization with the help of code snippet.

3.5 The OpenGL Standard Graphics Pipeline

OpenGL is an open graphics specification and it describes API for interacting with Graphics hardware in a system-independent manner.

3.6 Registering callbacks

Application can program interaction with input devices such as *mouse*, *keyboard* by registering them with GLUT framework. Further, it can also register handlers for window reshaping event, their core rendering handler. Listing 5 shows registration of such callbacks to GLUT framework along with their individual definitions.

Listing 5: Register event callbacks

```
////// Callback registration calls
glutDisplayFunc(display);
glutFullScreen();
glutReshapeFunc(reshape);
glutKeyboardFunc(keyboard);

////// Definitions of individual callbacks

//// The OpenGL rendering(texture mapping) callback
void display()
{
float vertex_coordinates[4][2];
float origin_x=0.0f,origin_y=0.0f;

//Compute vertex coordinates

origin_x=-1.0f+(2.0f/num_of_proj_in_row)*(proj_id%num_of_proj_in_row);
origin_y=-1.0f+(2.0f/num_of_proj_in_col)*(floor(proj_id/num_of_proj_in_row));
unsigned int i;
for(unsigned int vertex_id=0,i=0; vertex_id<4; vertex_id++)
  {
    if(vertex_id==2)
     i++;
    vertex_coordinates[vertex_id][0]=(2.0f/num_of_proj_in_row)*floor(vertex_id/2
```

```
      vertex_coordinates[vertex_id][1]=(2.0f/num_of_proj_in_col)*(i%2)+origin_y;
      i++;
     }


//Map the texture & vertex coordinates...
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glEnable(GL_TEXTURE_2D);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
glBindTexture(GL_TEXTURE_2D, texName);
glBegin(GL_QUADS);
glTexCoord2f(0.0, 0.0); //0,0
glVertex2f(vertex_coordinates[0][0],vertex_coordinates[0][1]);
glTexCoord2f(0.0, 1.0); // 0,1
glVertex2f(vertex_coordinates[1][0],vertex_coordinates[1][1]);
glTexCoord2f(1.0,1.0); // 1,1
glVertex2f(vertex_coordinates[2][0],vertex_coordinates[2][1]);
glTexCoord2f(1.0, 0.0);
glVertex2f(vertex_coordinates[3][0],vertex_coordinates[3][1]);
glEnd();
glFlush();
glDisable(GL_TEXTURE_2D);

return;

}

//// The window reshaping callback
void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);

}

//// Keyboard interaction callback
void keyboard (unsigned char key, int x, int y)
{
    switch (key)
    {
    case 27:
        exit(0);
        break;

    case 'n': // start projecting feature image on next projector
      proj_id++;
      glutPostRedisplay();
```

```
        break;

    default:
        break;
    }
}
```

4 Implementation details

5 Results