Hologram Graphics API (HAPI)

Mark Green

Faculty of Science

UOIT

Introduction

This document describes the design and implementation of the graphics API for holographic displays. From the application programmer's perspective the API is similar to a traditional graphics scene graph with a few extensions dictated by the nature of holographic displays. Internally the software architecture is quite different from that of a traditional graphics API. The output of this package is an interference pattern that when illuminated by a laser produces a hologram. While interference patterns can be viewed as images, they are quite different from the images normally encountered in graphics. In general the image produced by a single point light source is not spatially restricted and contributes to every pixel in the interference pattern. The interference pattern for two point light sources can be produced by summing the interference patterns for the individual light sources. These two properties of interference patterns have a significant impact on the software architecture of our API. Since most hologram generation algorithms are point based, geometry must eventually be converted into points, a process that doesn't appear in most graphics packages. Traditional graphics packages produce pixels, but these are points in a 2D space and not the 3D points that we need for our algorithms. Pixels can be converted into 3D points by retrieving the corresponding depth buffer values, but by this point in the graphics pipeline most of the 3D structure has been lost due to projection and other operations which could make it difficult to use these values in our algorithms.

Another issue is that most hologram generation algorithms don't deal with the hidden surface problem. Due to computational complexity, points and lines are favoured over polygons and other surface based primitives. Since lines are a common primitive some type of hidden surface algorithms is required. This is complicated by the fact that holograms can be viewed from different angles which could change the points that are visible.

Why are we doing this? Couldn't we just use a standard scene graph with a few modifications in the later stages to produce holograms? There are several answers to these questions:

- We have a holographic display and need some way of generating interference patterns for it. We could produce a separate program for each interference pattern, but a general approach seems to make more sense.
- We want to develop and evaluate different hologram generation algorithms and having a standard framework or software package facilitates this.
- We feel that there is enough difference between the generation of images for a raster display and a holographic display to justify the development of a new software architecture.

The next section of this document describes the desired feature set for our hologram generation API. The third section describes the software architecture of our API and presents a simple example program. The fourth section is a detailed description of the main objects in the API and how they are used.

Features

The following features have guided our design of the API:

- A relatively standard scene graphics structure from the application developer's point of view. A programmer who is already familiar with scene graphs should be able to learn our API is a short period of time.
- The possibility of importing geometry from one or more of the standard formats. The desire to use only line and point based primitives will have some influence on the selection of file format.
- The ability to use different hologram generation algorithms without impacting application programs that use the API. One of the main reasons for developing this API is to support the development and evaluation of new hologram generation algorithms. The ability to plug a new algorithm into the system without impacting any of the other components simplifies this greatly and provides an even play field for algorithm evaluation.
- The ability to reuse interference patterns that don't change during part or all of the execution of a program. The computation of interference patterns is time consuming and since they can be summed it makes sense to save output of parts of the scene graph that don't change so they don't need to be recomputed on each update. The programmer should have control over the parts of the scene graph that are considered to be static.
- The ability to take advantage of GPU computation when it's available. The interference pattern computations are highly parallelizable and the use of a GPU could significantly reduce the time to compute them.
- A direct interface to our holographic display that supports real time interaction and animation. The interferece patterns can be saved as images, which would be useful for long computation times. It is hoped that we can develop real time algorithms and in this case we would like to directly drive our display device.

Some of these features are relatively easy to realize, while others are more of an ongoing research project.

Software Architecture

At the highest level our software architecture consists of three main components:

1. The scene graph that is used by application programmers to describe holograms to be generated. This is the part of the API that is most visible to the typical programmer.
2. Data conversion from the format provided by the scene graph to the format required by the hologram generation algorithms. This involves converting geometry into the point light sources that approximate it and converting the data into the units required by the hologram generation algorithms, which is typically μm.

3. The hologram generation algorithms that produce the interference patterns. In some cases these interference patterns will be stored in the scene graph where the geometry won't be changing on each update. The final interference pattern for the entire scene graph is converted into an image that can be sent to the display device.

Scene Graph Architecture

The scene graph is similar to that of a standard graphics scene graph. It consists of nodes that are arranged in a tree or directed graph structure. There are three main types of nodes in our scene graph architecture:

1. Geometry nodes that represent the points, lines and polygons in the model.
2. Transformation nodes that represent geometrical transformations.
3. Attribute nodes that set attributes such as colour and whether the subtree below this node is static.

Each geometry nodes represents a collection of one type of geometrical primitive. The types that are currently supported a point, line and polygon. The type of the node is set when the node is created and cannot be changed. Each node can have an arbitrary number of primitives in it, but they must all be of the same type.

The transformation nodes consist of one or more the three standard are transformations; rotate, scale and translate. If more than one transformation is added to the node their transformation matrices are multiplied. It is also possible to add an arbitrary 4x4 matrix to a transformation node. Note there is not a separate node type for each type of transformation.

The attribute nodes are used for anything that is not geometry or a transformation. The colour attribute node is used to set the colour of all the geometry that is below it in the tree. Our current display only has a red laser, so this is of limited utility. The static node is used to indicate that everything below it doesn't change from one update to the next. After the first update the diffraction pattern corresponding to the subtree is stored at this node so it doesn't need to be computed on subsequent updates.

Data Conversion

Data conversion sits between the scene graph and the hologram generation algorithms. Line and polygon primitives must be converted into point light sources for the hologram generation algorithms. Note, point primitives don't need to be converted since they are already in the correct format.

The hologram generation algorithms operate in physical units that are relevant to the display device. In most cases this will be µm, which are not the most convenient application coordinates. Therefore, a mapping from application coordinates to µm is required. This is similar to the window/viewport transformation in most graphics packages. The application or world coordinates can be set to any convenient values for the programmer. This is specified in terms of minimum and maximum values along the x, y and z values. The device coordinates are constrained by the actual display device. This is the part of the physical space that the hologram will occupy. Again this can be specified in terms of minimum and maximum values along the x, y and z axis, but in this case the units are cm.

Hologram Generation Algorithms

The library can support multiple hologram generation algorithms, therefore instead of describing the algorithms the interface to the algorithms will be described instead. There are two inputs to a generation algorithm; the point light sources and an interference pattern object. The algorithms will add the contribution of each point light source to the interference pattern object. Thus, the interference pattern object is also the output from the algorithm. The details of the algorithms and how the application programmer can select them are covered in a subsequent section.

Device Interface

The application programmer has the choice of storing the computed interference patterns in a file or send them directly to the device. An API call is used to select the destination. This function is called at the start of the application and the destination cannot be changed while the application is running. In order to send the results directly to the display device, it must be connected by a USB cable to the computer running the application. A TCP/IP connection to the device is established over the USB connection and command packets are sent to the devices to send the patterns and instructions on how they are to be displayed. Most of this is hidden from the application programmer, but must be implemented in our API.

Detailed API Description

This section contains a detailed description of the API from the programmer's perspective. It describes the API functions that are required by most programs and how a HAPI program should be structured.

In general a program will be divided into three sections. The first section establishes the initial conditions for the program such as the number of colour channels, the wavelengths of the lasers used and the coordinate spaces. The second section constructs the scene graph for the application and the third section displays the scene graph and modifies it based on any user interaction.

Initialization Section

The initialization part of the program must specify the colour channels that are used by the program. The setChannels() function is used for this and the parameter to this function is the or of RED, GREEN, or BLUE constants. For each of the colour channels the wavelength of the laser that is used for that channel must be specified. The setWavelength() function is used for this purpose. The first parameter to this procedure is the channel (RED, GREEN, or BLUE) and the second parameter is the laser wavelength in nm. Both of these parameters are integers. For example, for a typical red laser the following function call would be used:

```
setWavelength(RED, 650);
```

Note, for each colour there are lasers with slightly different wavelengths and the hologram computation must match the wavelength of the laser that is used. For example, another common wavelength for red laser is 635 nm.

The package can save the computed diffraction patterns to a file or they can be directly sent to the display device. The setTarget() function is used to specify which of these alternative is used. The parameter this function is either FILE or DEVICE. In the case of a file target the base file name for the diffraction patterns must also be selected using the setBaseFileName() function. The single parameter to this procedure is a text string that specifies the location where the files should be saved and a prefix for the file name. Each colour channel is saved on a separate file and Red, Green, or Blue is append to the file name to indicate the channel.

The mapping for application coordinates to the physical location of the hologram are handled by the setWorldSpace() and setDeviceSpace() functions. There are six parameters to both of these functions, with the first three parameters being the coordinates of the front lower left corner of the space and the next three parameters being the coordinates of the back upper right corner of the space.

Scene Graph Construction

Display and Interaction

The traverse function traverses a scene graph, converts it into a diffraction pattern and sends the result to the target. The single parameter to this function is the root of the scene graph tree to be displayed.

Example Program

The test program that is distributed with the HAPI library is shown below. This program constructs 8 interference patterns that when displayed show a rotating truncated pyramid.

```
/*********************************************
 *
 *            test
 *
 *  Test program for the HAPI library. This
 *  program has three optional parameters that
 *  can be used to move the hologram on the
 *  display surface and depth.  The x and y
 *  displacements are used to avoid artifacts
 *  caused by the interaction between the laser
 *  and the DMD pixels.  The z displacement can
 *  be used to control the distance between the
 *  DMD surface and where the hologram appears.
 *
 *********************************************/
#define _USE_MATH_DEFINES
#include <math.h>
#include "HAPI.h"
#include <stdio.h>
#include "Matrix.h"
#include <vector>

int main(int argc, char **argv) {
```

```
Node *node = new Node(NONE);
GeometryNode *gnode;
TransformationNode *tnode;
int i;
char buffer[256];
double t1;
double dx, dy, dz;

dx = dy = dz = 0.0;
if (argc == 4) {
       dx = atof(argv[1]);
       dy = atof(argv[2]);
       dz = atof(argv[3]);
       printf("dz: %f\n", dz);
}

t1 = getSeconds();
/*
 *  Compute only the red channel and
 *  use a wavelenght of 650 nm for red
 */
setChannels(RED);
setWavelength(RED, 650.0);
/*
 *  Store the resulting interference
 *  pattern in a file, the base name
 *  for the files constructed is "test"
 */
setTarget(FILE);
setBaseFileName("test");
/*
 *  The world space defines the coordinate system
 *  used in the application.  Use whatever units
 *  you like.
 */
setWorldSpace(-1.0, -1.0, -1.0, 1.0, 1.0, 1.0);
/*
*  The device space specifies where the hologram
*  will appear in the real world.  The units used
*  are cm.  These values are a good start for our
*  prototype display
*/
setDeviceSpace(-0.4+dx, -0.4+dy, 17.75+dz, 0.4+dx, 0.4+dy, 18.25+dz);

/*
 *  Create the lines for the truncated pyramid
 */
gnode = new GeometryNode(LINE);
/* Front face*/
gnode->addLine(-0.5, -0.5, 1.0, 0.5, -0.5, 1.0);
gnode->addLine(0.5, -0.5, 1.0, 0.5, 0.5, 1.0);
gnode->addLine(0.5,0.5, 1.0, -0.5, 0.5, 1.0);
gnode->addLine(-0.5, 0.5, 1.0, -0.5, -0.5, 1.0);

/* Back Face */
gnode->addLine(-1.0, -1.0, -1.0, 1.0, -1.0, -1.0);
gnode->addLine(1.0, -1.0, -1.0, 1.0, 1.0, -1.0);
gnode->addLine(1.0, 1.0, -1.0, -1.0, 1.0, -1.0);
```

```
        gnode->addLine(-1.0, 1.0, -1.0, -1.0, -1.0, -1.0);

        /* Lines connecting the faces*/
        gnode->addLine(-0.5, -0.5, 1.0, -1.0, -1.0, -1.0);
        gnode->addLine(0.5, -0.5, 1.0, 1.0, -1.0, -1.0);
        gnode->addLine(0.5, 0.5, 1.0, 1.0, 1.0, -1.0);
        gnode->addLine(-0.5, 0.5, 1.0, -1.0, 1.0, -1.0);

        /*
        *  Set up the transformation for rotating the
        *  truncated pyramid
        */
        tnode = new TransformationNode();
        tnode->rotateZ(0.0);

        /*
        *  Link up the nodes in the scene graph
        */
        node->addChild(tnode);
        tnode->addChild(gnode);
        /*
         *  Generate the interference patterns
         */
        for(i=0; i<8; i++) {
               sprintf(buffer,"ptest%d",i);
               setBaseFileName(buffer);
               display(node);
               tnode->rotateZ(0.1);
        }
        printf("total time: %f\n", getSeconds() - t1);
}
```