The Uber Graphics Library: An Overview

Nathan Cournia
nathan@cournia.com

Key Words and Phrases: Uber, Graphics, Demo, Library

---

1. INTRODUCTION

This article presents the Uber graphics library. The Uber graphics library is a small C++ library designed to aid in the creation of graphics demos. The goal of Uber is to provide a base of commonly used code (such as methods to open window, perform matrix math, etc) so that the programmer can focus on polishing his/her demo, rather than worring about how to load a TARGA file.

2. FEATURES

Features of the library include:

—**Portablility.** With the help of `autoconf`, portability is achieved via the C prepoccessor and portable coding practices. Uber is know to successfully compile and run on both Linux and Windows platforms.

—**OpenGL.** All graphics are handled through the tried and tested OpenGL graphics library.

—**Image Manipulation.** Low level functions such as blitting, blending, and seeking are support through Uber's image interface.

—**File Format Support.** Uber natively supports the Portable Pixmap (PNM) and TrueVision TARGA (TGA) formats. BMP, XPM, LBM, PCX, GIF, JPEG, and PNG image formats are also support via 3rd party libraries.

—**Texturing** Building on top of Uber's image object and OpenGL, tedious operations such as texture loading, can now be achieved in a single function call.

—**Timers.** Uber's `clocker` object can be used for such things as performance measurement and physics calculations.

—**Utility Functions.** Uber features a small library of cross-platform portable utility functions such as `mkdir( )` and `chdir( )`. Other useful functions include PERL like string manipulation functions and C++ based file IO access functions.

—**Math Library.** Uber sports a full featured math library which supports mathematical concepts suchs as vectors, quaternions, and matrices. Each of these objects is represented as a C++ class, thereby affording the programmers all of the benfits of the C++ object system, such as type safety, encapsulation, polymorphism, and ease of use.

—**Multiple Backends.** Uber is design to support multiple backends, such as SDL, GLUT, X, and GTK+. At this time only the SDL backend is supported.

—**True Type Font Rendering** High quality text rendering is a breeze with Uber's true type font rendering object.

—**Graphical User Interfaces.** Tedious operations such as opening windows, initializing OpenGL, and switching between fullscreen contexts, are some of the features that Uber's video subsystem is capable of easily handling.

2.1 What Uber Doesn't Do

Uber is not designed for the following operations:

—**Input Handling.** The Uber library is designed to support multiple backends (such as SDL, GLUT, X, GTK+, etc.). Input handling in each of these backends is designed differently. Abstracting the input from these backends into a common format would be difficult. As such, Uber makes no attempt to do so. Input handling must be written in a backend specific manner. To aid the programmer in this task, C preprocessor symbols are provided to determine backend capabilities.

—**Sound.** Uber has no sound interface. Sound must be handled either via the backend, or an external libarary.

3. GETTING STARTED

3.1 Obtaining Sources

Before you start using the Uber graphics library, you must first obtain it. The Uber graphics libary can only be downloaded from CVS. You can download the library through CVS on Clemson University's VR network as follows:

```
cvs -d/pub/research/uber/cvsroot co uber
```

Off-campus via SSH:

```
cvs -dusername@rafiki.vr.clemson.edu:/pub/research/uber/cvsroot co uber
```

Where username is your login on Clemson University's VR network.

3.2 Building Uber

Once you have obtained the Uber source code, you must build it. To build the Uber library, you must first initialize autoconf:

```
./bootstrap
```

This will generate the configure script, which must be ran next:

```
./configure
```

The final step in building the Uber library is compiling and linking the source code. This is done via make:

```
make
```

At this point the Uber library should be built.

3.3 Viewing Demos

All demos are contained in the demos directory. Within the demos directory, each demo is located in a self-contained directory. To run a demo enter at the shell:

```
cd demos
cd mydemo
./mydemo
```

Where mydemo is the name of the demo you wish to view.

4. USING UBER

One of Uber's design goals is ease-of-use. As such, an effort has been made to keep Uber's interface small and simple. In this section we give an overview of this small and simple interface, resulting in several example programs.

4.1 Creating a Window

Opening a window is a simple and straight forward process. To open a window, simply initalize a video object. To initialize the video object, create the object followed by a call to the objects video_base::init method.

```
void video_base::init(
  unsigned int w,   //width of the window
  unsigned int h,   //height of the window
  unsigned int bpp, //bits per pixel
  bool fs )        //true for fullscreen mode
```

Listing 1 illustrates an example on how to initalize the video object.

Listing 1. Opening a Window

```
//create a video context with a SDL backend
video_sdl video;
if( !video.init( 640, 480, 32, false ) ) {
  //something went wrong while initalizing the video object
  std::cerr << "error: " << video.get_error( ) << std::endl;
  return 1;
}
```

Note that the programmer has a choice as to what type of video backend he/she wishes to use. In Listing 1 a SDL backend was choosen by creating a video_sdl object. Other backends can be choosen by simply creating an object of the appropriate type. For example, to create a GTK+ backend, instantiate a video_gtk object.

Refer to demos/overview/window.cpp for a complete example illustrating how to open a window and handle backend specific events.

4.2 Creating a Simple Image Viewer

Now that we know how to open a window, we must do something with it. In this section we overview how to load a texture into the OpenGL subsystem, and display that image on the screen.

4.2.1 *Loading File Textures.* Loading a texture into OpenGL is a straightforward process. Simply include the file texture.h in your source code and call gl::texture::load.

```
gl::texture gl::texture::load( const std::string& filename,
  bool build_mipmaps = true,
  GLfloat min_filter = GL_LINEAR_MIPMAP_LINEAR,
  GLfloat max_filter = GL_LINEAR )
```

gl::texture::load returns a texture object suitable for passing to OpenGL functions such as glBindTexture. Note that by default gl::texture::load will build mipmaps for filename and enable trilinear filtering. gl::texture::load is capable of handle non power of two images. Listing 2 shows an example of loading a file into the OpenGL subsystem.

Listing 2. Loading a File Texture

```
//load texture from file
gl::texture texture = gl::texture::load( "mrt.jpg" );
if( !texture ) {
  std::cerr << "error: could not load 'mrt.jpg'" << std::endl;
  return 1;
}
```

See demos/overview/simple_img.cpp for a complete example showing how to load a texture from a file.

4.2.2 *Writing to the Screen in 2D.* Uber has the capablility to easily draw textures in OpenGL's orthographic mode. This capability aids in the creation of such things such as heads up displays. To render a texture to the screen in orthographic mode call video_base::render_image.

```
void vidoe_base::render_image(
  const rectf& rect,
  const gl::texture& texture )
```

All 2D drawing via Uber is done in a resolution independent 1024x768 virtual window. Listing 3 demonstrates drawing a texture to this virtual window.

Listing 3. Drawing to Uber's Virtual Window

```
  //create a drawing rectangle to tell uber where to
  //draw the image on the screen, these coordinates
  //will be set in uber's 1024x768 virtual 2d window.
  rectf draw_rect;

  //find the greatest dimension of the image (w or h)
  //and set it to 700 pixels (in uber's virtual
  //space).  scale the other dimension according to
  //the texture's aspect ratio.
  if( texture.get_aspect( ) > 1.0 ) {
    //width of the image dominates
    draw_rect.set_wh( 700.0, 700.0 / texture.get_aspect( ) );
```

```
  } else {
    //height of the image dominates
    draw_rect.set_wh( 700.0 * texture.get_aspect( ), 700.0 );
  }

  //determine the coordinates to draw the image in the
  //center of the screen
  draw_rect.set_xy( 512.0 - draw_rect.w / 2.0,
    384.0 - draw_rect.h / 2.0 );

  //. . .

  //draw the image to the screen
  video.render_image( draw_rect, texture );
```

See demos/overview/simple_img.cpp for an example on how to draw to Uber's virtual window. Figure 1 shows the result of this example program.

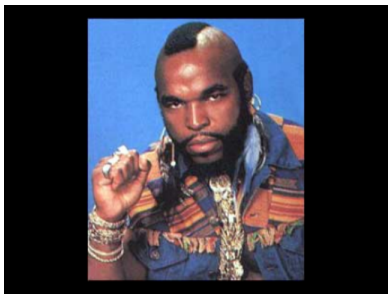

Fig. 1. Example of Drawing to Uber's Virtual Window

4.3 Image Blitting/Blending

Uber contains a facility to blit images onto images. Uber's blitting interface is capable of operating in the traditional since (a 1 to 1 pixel copy) and also capable of performing alpha blending.

4.3.1 *Loading Images.* Before one can blit to an image, an image must be loaded into memory. The is done through Uber's image::image_t object and the image::load function.

```
image::image_t* image::load( const std::string& filename )
```

In order to use image::load the prorgammer must include image_loader.h. Listing 4 contains an example of loading images into memory.

4.3.2 *Blitting Images.* Once images have been loaded into memory, blitting can occurr. Blitting is performed by calling the destination image's image::image_t::blit method.

```
bool image::image_t::blit(
  const image_t& img,
```

```
  const recti& region )
```

Listing 4 shows an example of loading an image into memory followed by blitting one image onto the other.

Listing 4. Blitting Images

```
//load base image
  image::image_t *base = image::load( "mrt.jpg" );
  if( !base ) {
    std::cerr << "error: could not load 'mrt.jpg'"
      << std::endl;
    return 1;
  }

  //load image that will be blitted on the base image
  image::image_t *decal = image::load( "karl.png" );
  if( !decal ) {
    std::cerr << "error: could not load 'karl.png'"
      << std::endl;
    return 1;
  }

  //define a blitting region on the base image
  recti blit_rect;
  blit_rect.set_xy( 70, 0 );

  //blit the decal onto the base image
  base->blit( *decal, blit_rect );
```

4.3.3 *Alpha Blending Images.* Figure 2 shows the resulting image produced from Listing 4. image::image_t::blit performs a 1 to 1 pixel blit. Notice that the source image has completly overwritten all pixels in the blitting region. As a result, Figure 2 looks less than stellar. To fix this we can blend the source image onto the destination image (using the alpha information stored in the source image). This is called alpha blitting/blending. In order to perform alpha blitting, call image::image_t::blend.

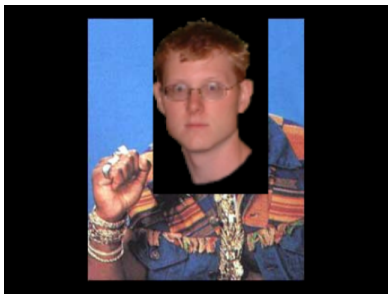

Fig. 2. Blitting an image onto another.

```
bool image::image_t::blend(
  const image_t& img,
  const recti& region )
```

Figure 3 displays the results of Listing 4 with the call to image::image_t::blit replaced with image::image_t::blend.

Fig. 3. Blending an image onto another.

4.3.4 *Loading Textures from Memory.* Once an image has been loaded into memory, sending the image to the OpenGL subsystem is achieved by calling gl::texture::load.

```
gl::texture gl::texture::load( const image::image_t *img
  bool build_mipmaps = true,
  GLfloat min_filter = GL_LINEAR_MIPMAP_LINEAR,
  GLfloat max_filter = GL_LINEAR )
```

Once the image has been loaded into OpenGL, the programmer can safely delete the image. See demos/overview/simple_img.cpp for an example on how to load images residing in memory into the OpenGL subsystem.

4.4 Text Rendering

Building on top of the Freetype 2.0 library, Uber is capable of rendering high quality bitmap, Type 1, and True Type fonts to the screen.

4.4.1 *Loading Fonts.* Loading a font is as simple as a call to gl::ttf::load.

```
gl::ttf* load(
  const std::string& filename,
  unsigned int font_height ) //height in px of tallest glyph
```

Listing 5 contains an example of loading a font.

4.4.2 *Rendering Strings.* Rendering text to the screen is similar to rendering images to Uber's virtual window. video_base::render_string renders given text to the screen.

```
void video_base::render_string(
  real_t &x, //x coord to render string (in virtual window)
  real_t &y, //y coord to render string (in virtual window)
  const gl::ttf *font, //font to use
```

```
  real_t size, //height of line in (virtual window) pixels
  const std::string& text )
```

After calling video_base::render_string, x and y will be updated with the coordinate of the last pixel rendered. This pixel will always be the upper right hand pixel of the last character in the string. Listing 5 demonstrates how to load a font, and then render text to the screen using the loaded font.

Listing 5. Loading/Rendering Fonts

```
//load the font
  gl::ttf *font = gl::ttf::load( "ariblk.ttf", 32 );
  if( !font ) {
    std::cerr << "error: could not load 'ariblk.ttf'"
    << std::endl;
    return 1;
  }

  //set the location of where we want our text rendered
  real_t text_x = 310.0;
  real_t text_y = 0.0;

  //. . .

  //render the text
  video.render_string( text_x, text_y, font, 32,
    "I PITTY THE FOOL!" );
```

See demos/overview/font.cpp for a more in-depth example on how to load and use fonts. Figure 4 shows the result of this example program.

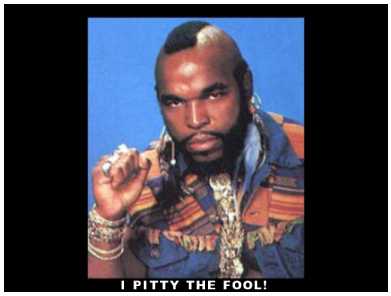

Fig. 4. Rendering text to the screen.

4.5 Mathematics

All mathematical objects and functions reside in the math namespace. Each object in the math namespace has been designed to take full advantage of the capabilities of C++. Objects such as real_t, math::matrix4, math::vector, and math::quaternion have been design to work seamlessly with each other and OpenGL. Overloaded

operators have been defined for each object to aid in producing code that is both easy to read, and easy to write.

In the following sections we provide code snippets to help the programmer get a feel as to how Uber's math facililties work. For a more in-depth view of Uber's math interface, refer to Uber's API documentation.

4.5.1 *Vectors.* Vectors in Uber are represents as 3-vectors and are defined in the object math::vector. Overloaded operators for vectors include -, +, =, ==, *, and /. Overloaded operators have been defined for such operations such as scalar addition, scalar, subtractions, and scalar multiplication. Listing 6 shows math::vector in action. Refer to Uber's API documentation for more information.

Listing 6. Uber's Vector Object

```
//declaring a vector
math::vector u;

//declaring a vector, passing initializers for x, y, z
math::vector v( 2.0, 4.0, -10 );

//setting the value of a vector
u.set( -1.0, 7.0, 5.0 );

//setting the value of a component of the vector
u.x( 6.0 );
u[ 0 ] = 6.0;

//vector multiplication
math::vector q = u * v;

//vector multiplication (with assignment)
q *= u;

//vector addition
q = u + v;

//vector subtraction
q = u - v;

//scalar multiplication
real_t s;
q = u * s;

//scalar addition
q = u + s;

//scalar subtraction
q = u - s;

//scalar division
q = u / s;
```

```
//normalizing a vector
q.normalize( );

//length of a vector
s = q.length( );

//negate (flip) a vector
s.negate( );

//cross product
q = math::cross( u, v );

//dot product
s = math::dot( u,v );

//outputting a vector
std::cout << v << std::endl;

//sending the vector to OpenGL
glVertex3v( u );

//assignment of predefined vectors
u = math::vector::IDENTITY;
u = math::vector::X_AXIS;
u = math::vector::Y_AXIS;
u = math::vector::Z_AXIS;
```

4.5.2 *Matrices.* Matrices in Uber are represents as 4x4 matrices and are defined in the object math::matrix4. Overloaded operators for matrices include -, +, =, ==, *, and /. Overloaded operators have been defined for such operations such as scalar addition, scalar, subtractions, and scalar multiplication. Note that data in the math::matrix4 is stored in row major format. Since OpenGL expects matrices passed into functions such as glMultMatrix* to be in column major format, the programmer must transpose the matrix before sending the matrix to OpenGL. Listing 7 shows math::matrix4 in action. Refer to Uber's API documentation for more information.

Listing 7. Uber's Matrix Object

```
//declaring a matrix
math::matrix4 ma;

//declaring a matrix, passing initializers for
//rotation matrix (3x3)
math::matrix4 mb(
  0.0, 0.0, 0.0,
  0.0, 0.0, 0.0,
  0.0, 0.0, 0.0
);
```

```
//declaring a matrix, passing initializers for
//the entire matrix (4x4)
math::matrix4 mc(
  0.0, 0.0, 0.0, 0.0,
  0.0, 0.0, 0.0, 0.0,
  0.0, 0.0, 0.0, 0.0,
  0.0, 0.0, 0.0, 0.0
);

//setting the values of the rotation part of the matrix
ma.set(
  0.0, 0.0, 0.0,
  0.0, 0.0, 0.0,
  0.0, 0.0, 0.0
);

//setting the values of the entire matrix
ma.set(
  0.0, 0.0, 0.0, 0.0,
  0.0, 0.0, 0.0, 0.0,
  0.0, 0.0, 0.0, 0.0,
  0.0, 0.0, 0.0, 0.0
);

//setting the value of a cell in the matrix
//(0 indexed)
ma.set( 2, 3, 666.0 );

//matrix multiplication
ma = mb * mc;

//matrix addition
ma = mb + mc;

//matrix subtraction
ma = mb - mc;

//vector multiplication
math::vector v;
ma = mb * v;

//outputting a matrix
std::cout << ma << std::endl;

//transposing the matrix
ma.transpose( );

//setting the matrix to the identity matrix
ma.set_identity( );
ma = math::matrix4::IDENTITY;
```

```
//setting the translation part of the matrix
ma.set_translation( v );

//getting the translation part of the matrix
v = ma.get_translation( );

//compute the inverse of the rotation matrix
ma = mb.get_inverted_matrix( );

//create a rotation matrix n radians around an arbitrary axis
math::vector axis;
real_t angle; //in radians
ma.from_angle_axis( angle, axis );

//get the axis and raidans around the axis the rotation
//matrix describes
ma.get_angle_axis( angle, axis );

//send the matrix to OpenGL
//note that you must first transpose the matrix
ma.transpose( );
glMultMatrixf( ma );
```

4.5.3 *Quaternions.* Quaternions in Uber are represented in the object math::quaternion. Listing 8 shows math::quaternion in action. Refer to Uber's API documentation for more information.

Listing 8. Uber's Quaternion Object

```
//declaring a quaternion
math::quaternion qa;

//declaring a quaternion, initalizing x y z w
math::quaternion qb( 0.0, 0.0, 0.0, 1.0 );

//declaring a quaternion, initializing with a vector and w
math::vector v;
real_t w;
math::quaternion qb( v, w );

//setting the data in a quaternion
qa.set( v, w );
qa.set( 0.0, 0.0, 0.0, 1.0 );

//finding the conjugate of a quaternion
qb = qa.conjugate( );

//find the inverse of a quaternion
qb = !qa;
```

```
//normalizing a quaternion
qa.normalize( );

//finding the magnitude of a quaternion
real_t s = qa.magnitude( );

//setting a quaterion to the identity quaternion
qa.set_identity( );
qa = math::quaternion::IDENTITY;

//setting parts of the quaternion
qa.x( 4.0 );
qa.v( v );

//getting parts of the quaternion
real_t s = qa.x( );
v = qa.v( );

//matrix multiplication
v = qa * v;

//create a quaternion n radians around an arbitrary axis
math::vector axis;
real_t angle; //in radians
qa.from_angle_axis( angle, axis );

//get the axis and raidans around the axis the
//quaternion describes
qa.get_angle_axis( angle, axis );

//interpolating between two quaternions
real_t percent; //between 0.0 and 1.0
qa = math::quaternion::slerp( percent, qb, qc );

//outputting a quaternion
std::cout << q << std::endl;

//sending the quaternion to OpenGL
glMultMatrixf( qa.get_transposed_rotation_matrix( ) );
```

4.5.4 *Helpers.* Listing 9 demonstrates some of Uber's useful math helper function. Refer to mathematics.h and Uber's API documentation for further information on Uber's helper functions.

Listing 9. Uber's Math Helper Functions

```
//return a random number between 0 and 1
real_t s;
s = math::rand01( );

//convert from radians to degrees
```

```
real_t deg, rad;
deg = math::to_degrees( rad );

//convert from degrees to radians
real_t deg, rad;
rad = math::to_radians( deg );

//determine if 2 numbers are equal,
//accounting for precision error
double a, b;
if( math::equals( a, b ) ) {
  //equals
} else {
  //not equals
}

//swap the value of 2 objects
double a, b;
math::swap( a, b );

//determine sign of a value
if( math::sign( a ) < 0 ) {
  //a is negative
} else {
  //b is not negative
}

//determine if two values hold the same sign
if( math::same_sign( a, b ) ) {
  //a and b have same sign
} else {
  //a and b's signs differ
}

//check if a value is negative
if( math::negative( a ) {
  //a is negative
} else {
  //a is not negative
}

//check if a value is positive
if( math::positive( a ) {
  //a is positive
} else {
  //a is negative
}
```

4.6 Utilities

Located in the utils namespace (utils.h) is a slew of useful helper functions. In the following sections, we will cover some of those functions. Refer to Uber's API documentation for further information on Uber's utility functions.

4.6.1 *String Manipulation.* Refer to Uber's API documentation for further information on Uber's utility functions.

4.6.2 *File System Access.* Refer to Uber's API documentation for further information on Uber's utility functions.

4.6.3 *File IO.* Uber contains a simple mechanism for writing and reading from binary streams. Given an object, struct, or type, Uber is capable of loading the given object, struct or type from a binary stream or reading data from a binary stream into the object, struct, or type in a single function call. Reading from a binary stream is achieved through utils::read.

```
template <class T>
std::istream& utils::read(
  std::istream& stream,
  T& in )
```

Listing 10 contains an example of reading data from a binary stream into an arbitrary structure.

Listing 10. Uber's File IO: Reading

```
//an arbitrary struct
typedef unsigned char byte;
struct tga_header
{
  byte id_length;
  byte color_map_type;
  byte image_type;
  byte color_map[ 5 ];
  short x_origin;
  short y_origin;
  short width;
  short height;
  byte bpp;
  byte image_descriptor;
};

// . . .

//open a file
std::ifstream mystream( "myfile.tga" );
// . . . perform error checking

//instantiate the header we want to read data into
tga_header header;
```

```
//read from the file
if( !utils::read( mystream, header ) ) {
  //error.  something went wrong while reading
  //from the file.
} else {
  //header has now been loaded with data from the file
}
```

Writing to a binary stream is achieved through utils::write.

```
template <class T>
std::ostream& utils::write(
  std::ostream& stream,
  T& out )
```

Listing 11 contains an example of writing data to a binary stream from an arbitrary structure.

Listing 11. Uber's File IO: Writing

```
//an arbitrary struct
typedef unsigned char byte;
struct tga_header
{
  byte id_length;
  byte color_map_type;
  byte image_type;
  byte color_map[ 5 ];
  short x_origin;
  short y_origin;
  short width;
  short height;
  byte bpp;
  byte image_descriptor;
};

// . . .

//open a file for writing
std::ofstream mystream( "myfile.tga" );
// . . . perform error checking

//instantiate the header we want to read data from
tga_header header;

// . . . load header full of the data we want to write

//write heaader to the file
if( !utils::write( mystream, header ) ) {
  //error.  something went wrong while writing
  //to the file.
} else {
```

```
  //header has now been written to the file
}
```

5. CREATING A DEMO

This section covers creating a demo in the Uber Graphics Library heirarchy. After obtaining the the Uber source code from CVS, change directories to demos as follows:

```
cd demos
```

Here create a directory to store the files for your demo. For example, if you wish to create a demo called mydemo type:

```
mkdir mydemo
```

Change your current directory to the directory you just created. You should now place all files neccessary to run your demo in this directory. For example, if your demo is only one file, mydemo.cpp, you would copy mydemo.cpp into the demos/mydemo directory.

5.1 Setting Up the Build System

Now that all files are in the proper place, you must tell the build system, autoconf, which files to build. Using our mydemo example, first create an automake file in your demo directory. The name of the automake file should be Makefile.am. Listing 12 shows an example automake file for the mydemo example demo.

Listing 12. Example Demo automake File

```
## require automake 1.6
AUTOMAKE_OPTIONS = 1.6

## what follows bin_PROGRAMS are the names of
## binaries that should be built
bin_PROGRAMS=mydemo

## where we can find the uber source dir
UBER_SRC = ../../src

## include uber headers
INCLUDE = -I$(UBER_SRC)

## each binary that is to be built has a _SOURCES line
## this line contains all of the sources needed to build
## the binary
mydemo_SOURCES = mydemo.cpp

## pass flags from autoconf to automake
AM_CXXFLAGS = @CXXFLAGS@ @WERROR@
```

```
AM_LDFLAGS = @LDFLAGS@

## each binary that is to be built has a _CXXFLAGS
## this line contains all of the c++ flags
## that will be sent to the c++ compiler as
## each source file is compiled
mydemo_CXXFLAGS = $(INCLUDE) $(GL_CFLAGS) \
  $(SDL_CFLAGS) $(FT2_CFLAGS)

## each binary that is to be built has a _LDFLAGS
## this line contains all of the linker flags
## that will be sent to the liner
## when the program is linked together
mydemo_LDADD = \
  $(UBER_SRC)/libuber.a \
  @LIBS@ \
  $(GL_LIBS) \
  $(SDL_LIBS) \
  $(FT2_LIBS)
```

The next step is to tell autoconf where it can find your new sources. To do this first add your demo's directory to the file demos/Makefile.am. Listing 13 shows an example of this.

Listing 13. Telling automake to proccess your demo.

```
SUBDIRS = overview mydemo
```

We then tell autoconf to generate a makefile for the new demo. This is done by editing configure.in in Uber's root directory. In configure.in, add the demo as an argument to AC_OUTPUT (located near the bottom of the file.) Listing 14 shows an example of this.

Listing 14. Telling autoconf to generate a Makefile for your demo.

```
AC_OUTPUT([
  Makefile
  src/Makefile
  demos/Makefile
  demos/overview/Makefile
  demos/mydemo/Makefile
  ])
```

Everything is now in place for your code to be built. To do so, simply change your current directory to Uber's root directory and type the following:

```
./bootstrap
./configure
make
```

Note that we ran bootstrap and configure because we made changes to those systems' configuration files. This step is usually not neccessary during the normal course of developing your software.

5.2 Adding Demo Files to CVS

The final step to adding your demo to the Uber library is upload your additions. To reach this goal change your current directory to demos and add your demo's directory to the CVS repository as follows:

```
cvs add mydemo
```

This will create the directory CVS in the mydemo directory. Next, change your current directory to your demo's directory and upload your new files to the CVS repository as follows:

```
cd mydemo
cvs add Makefile.am mydemo.cpp
```

Only add Makefile.am and the source code of your demo! **DO NOT** run `cvs add *` in your demo's directory, as this may add unwanted files produced by the build system.

The final step in uploading your demo is to commit your changes. To do this change your current directory to Uber's root directory, and type the following:

```
cvs commit
```

This will bring up a text editor asking you to descibe your additions to the CVS repository. Type in a short description of your additions in the following format:

```
- First comment here.
- Second comment here.
```

Note that if you are on Clemson's VR network, you may be asked to describe your changes more than once. This is due to the fact that changes were made in multiple directories (Uber's root directory, demos, and mydemo).

6. USING UBER'S CVS

6.1 Downloading Uber

The Uber graphics libary can only be downloaded from CVS. You can download the library through CVS on Clemson University's VR network as follows:

```
cvs -d/pub/research/uber/cvsroot co uber
```

Off-campus via SSH:

```
cvs -dusername@rafiki.vr.clemson.edu:/pub/research/uber/cvsroot co uber
```

Where username is your login on Clemson University's VR network.

6.2 Updating Your Working Directory

Once a working copy of Uber has been check out of the CVS repository, getting the latest updates from the repository is as simple as:

```
cvs update -d
```

6.3 Upload (Committing) Changes to the CVS Repository

Once changes have been made to the local working copy of Uber, changes can be uploaded to the CVS repository as follows:

```
cvs commit
```

Note that this command is sensitive to where it is being run from. If it is ran from Uber's root directory, CVS will recursively descend into subdiretories, committing any changes.

6.4 Adding Files

Add plain text files and directories can be acheived with:

```
cvs add <filename>
```

Binary files should be added with:

```
cvs add -kb <filename>
```

Note that this only schedules the file(s) to be added. To add the file(s) to the repository run:

```
cvs commit
```

7. CONCLUSION

We have presented a brief overview of the Uber graphics library. We have by no means covered all of its features. For a more in-depth look at the Uber library, refer to Uber's API documentation.