Introduction to GEL

An informal introduction to the GEL library with many examples of usage.

J. Andreas Bærentzen

July 12, 2010

Contents

1 Introduction **2**
- 1.1 GEL Homepage 2
- 1.2 Purpose of this document 2
- 1.3 Overview of GEL 2

2 Compilation and Installation **3**

3 CGLA **3**
- 3.1 Naming conventions 4
- 3.2 How to use CGLA 4
- 3.3 CGLA: Examples of Usage 5

4 HMesh **7**
- 4.1 Extended Example: Edge Flipping 9
- 4.2 Extended Example: Implicit Surface Polygonization 10

5 GLGraphics **11**

6 LinAlg **14**

7 Geometry **15**

8 Util **16**

9 Authors and License **17**

1 Introduction

GEL is an abbreviation of GEometry and Linear algebra. GEL is a C++ library with tools for manipulating digital representations of geometry such as polygonal meshes, triangle meshes, point clouds, and voxel grids. Since linear algebra tools are invariably needed for this, GEL also contains tools for linear algebra. Finally, GEL contains tools for drawing geometry using OpenGL and various utilities.

1.1 GEL Homepage

The online home of GEL is here: http://www.imm.dtu.dk/GEL/

1.2 Purpose of this document

GEL is not sufficiently documented, but a fair amount of Doxygen documentation is provided for those parts of GEL which are used a lot. The goal of this document is to provide more detail and to help people getting started. In particular, this document focuses on functionality in the CGLA and HMesh namespaces, but we shall also take a look at some other parts of GEL. Unfortunately, it is rather difficult to be complete: There is functionality in GEL not covered here. Consequently, if you are looking for something particular, we **entreat** you to look at the Doxygen web (start at the GEL homepage). Most functions and classes are in fact described in the auto-generated documentation.

1.3 Overview of GEL

GEL contains many tools for geometry processing and a number of data structures. Perhaps the most important is the Manifold class which resides in the HMesh namespace. A Manifold is a halfedge based representation of a polygonal mesh. There is also a simpler triangle mesh data structure, TriMesh, based on an indexed face set. There are several voxel grid data structures, a fairly efficient k-D tree for storing points, bounding box hierarchies etc.

A number of algorithms exist for manipulating these representations of geometric data. For instance, Manifold has a function for mesh simplification, several functions for smoothing (including anisotropic), mesh optimization, and mesh clean up. Another strong point of GEL is the fact that it contains good support for converting between representations; especially between meshes and voxel grids. There are several tools for polygonizing isosurfaces and also converting meshes to distance fields. The tools pertaining to the Manifold data structure are in the HMesh namespace. The other geometry related tools are in the Geometry namespace.

There are two packages for linear algebra: CGLA is strictly for small vectors and matrices (dimensions 2,3, and 4). LinAlg is a Lapack wrapper for slightly larger problems. At this point, it is fairly simple but includes a number of functions for solving linear systems, factoring matrices and finding eigensolutions for symmetric matrices.

GLGraphics contains facilities for drawing entities from other parts of GEL via OpenGL. Especially TriMesh and Manifold have good support. For instance, there is a nice wireframe drawing class. There are also two virtual trackballs for interactive programs and some tools for viewing in interactive programs and SOIL a small open source library for image loading by Jonathan Dummer.

Finally there are some utilities in Util: Getting command line arguments, hash table classes, a 2D grid class, a resource manager, an XML parser by Jeppe Frisvad and other tools.

Apart from the namespaces (sublibraries) we have a number of demo applications. Some of these might be useful in themselves. One example is a tool for converting a mesh to a signed distance field and an OBJ viewer which is also able to load X3D and PLY meshes. We are working on the MeshEdit application which is already a highly useful swiss knife for geometry processing.

2 Compilation and Installation

Life would be easier if one's library did not rely on dependencies. We try hard to have only a minimal set, but the following libraries *are* required in order to build GEL.

- OpenGL
- GLUT
- GLEW
- Lapack (and its depencies)
- GLConsole (only for MeshEdit).

OpenGL is typically installed on any platform. GLUT is typically available for any platform. A replacement such as FreeGLUT should be an option. Lapack is typically also available everywhere. The only problem is GLConsole. As of writing only MeshEdit depends on GLConsole which can be obtained from SourceForge:

http://sourceforge.net/projects/glconsole/

GLConsole is extremely convenient but a well kept, little documented secret. Please do not try to install it unless you need MeshEdit. We will update this text if we either include it in GEL (assuming the authors allow it) or change to some other library.

GEL is compiled using CMake. Currently CMake is the official way of producing Visual Studio Solution files. A separate Mac OSX XCode project file is also maintained but may be removed if CMake makes it redundant. A set of Makefiles for various unices are also found in the package but no longer maintained.

To install GEL, put the libraries and header files in the logical place. CMake should instruct your build environment on how to do this. Then you can use GEL simply by including the header files and linking against the GEL libraries. On windows using Visual Studio it is important that you define SECURE_SCL to be 0. This is because Visual Studio is a bit pedantic about not allowing access to the raw memory of STL containers.

3 CGLA

CGLA is a set of numerical C++ vector and matrix classes and class templates designed with computer graphics in mind. CGLA stands for "Computer Graphics Linear Algebra". Let us get right down to the obvious question: Why create another linear algebra package? Well, CGLA evolved from a few matrix and vector classes because I didn't have anything better. Also, I created CGLA to experiment with some template programming techniques. This led to the most important feature of CGLA, namely the fact that all vector types are derived from the same template.

This makes it easy to ensure identical semantics: Since all vectors have inherited, say, the * operator from a common ancestor, it works the same for all of them.

It is important to note that CGLA was designed for Computer Graphics (not numerical computations) and this had a number of implications. Since, in computer graphics we mainly need small vectors of dimension 2,3, or 4 CGLA was designed for vectors of low dimensionality. Moreover, the amount of memory allocated for a vector is decided by its type at compile time. CGLA does not use dynamic memory. CGLA also does not use virtual functions, and most functions are inline. These features all help making CGLA relatively fast.

Of course, other libraries of vector templates for computer graphics exist, but to my knowledge none where the fundamental templates are parametrized w.r.t. dimension as well as type. In other words, we have a template (`ArithVec`)that gets both type (e.g. `float`) and dimension (e.g. 3) as arguments. the intended use of this template is as ancestor of concrete types such as `Vec3f` - a 3D floating point type.

The use of just one template as basis is very important, I believe, since it makes it extremely simple to add new types of vectors. Another very generic template is `ArithMat` which is a template for matrix classes. (and not necessarily NxN matrices).

From a users perspective CGLA contains a number of vector and matrix classes, a quaternion and some utility classes. In summary, the most important features are

- A number of 2, 3 and 4 d vector classes.
- A number of Matrix classes.
- A Quaternion class.
- Some test programs.
- Works well with OpenGL.

3.1 Naming conventions

Vectors in CGLA are named `VecDT` where `D` stands for dimension at `T` for type. For instance a 3D floating point vector is named `Vec3f`. Other types are d (double), s (short), i (int), uc (unsigned char), and usi (unsigned shourt int).

Matrices are similiarly named `MatDxDT`. For instance a 4D `double` matrix is called `Mat4x4d`.

3.2 How to use CGLA

If you need a given CGLA class you can find the header file that contains it in this document. Simply include the header file and use the class. Remember also that all CGLA functions and classes live in the CGLA namespace! Lastly, look at the example programs that came with the code.

An important point is that you should never use the Arith... classes directly. Classes whose names begin with Arith are templates used for deriving concrete types. It is simpler, cleaner, and the intended thing to do to only use the derived types.

In some cases, you may find that you need a vector or matrix class that I haven't defined. If so, it is fortunate that CGLA is easy to extend. Just look at, say, `Vec4f` if you need a `Vec5d` class.

For some more specific help look at the next section where some of the commonly used operations are shown.

3.3 CGLA: Examples of Usage

The examples below are by no means complete. Many things are possible but not covered below. However, most of the common usage is shown, so this should be enough to get you started. Note that in the following we assume that you are `using namespace CGLA` and hence don't prefix with `CGLA::`.

In short, to compile the examples below you would need the following in the top of your file

```
#include <iostream> // For input output
#include "CGLA/Vec3f.h"
#include "CGLA/Quaternion.h"
#include "CGLA/Mat4x4f.h"

using namespace std; // For input output
using namespace CGLA;
```

To begin with let us create 3 3D vectors. This is done as follows:

```
  Vec3f p0(10,10,10);
  Vec3f p1(20,10,10);
  Vec3f p2(10,20,10);
```

if we had left out the arguments the three vectors would have been uninitialized, at least in release mode. If we compile in debug mode they would have been initialized to "Not a Number" to aid debugging. However, the $[10\ 10\ 10]^T$ vector could also have been created differently, using

```
  Vec3f p0(10);
```

A very common operation is to compute the normal of a triangle from the position of its vertices. Assuming the three vectors represent the vertices of a triangle, we can compute the normal by finding the vector from the first vertex to the two other vertices, taking the cross product and normalizing the result. This is a one-liner:

```
  Vec3f n = normalize(cross(p1-p0,p2-p0));
```

Quite a lot goes on in the snippet above. Observe that the - operator also works for vectors. In fact almost all the arithmetic operators work for vectors. You can also use assignment operators (i.e +=) which is often faster. Then there is the function `cross` which simply computes the cross product of its arguments. Another frequently used function is `dot` which takes the dot product. Finally the vector is normalized using the function normalize.

Of course, we can print all or at least most CGLA entities. For example

```
  cout << n << endl;
```

will print the normal vector just computed. We can also treat a vector as an array as shown below

```
float x = n[0];
```

here, of course, we just extracted the first coordinate of the vector.

CGLA contains a number of features that are not used very frequently, but which are used frequently enough to warrent inclusion. A good example is assigning to a vector using spherical coordinates:

```
Vec3f p;
p.set_spherical(0.955317, 3.1415926f/4.0f , 1);
```

CGLA also includes a quaternion class. Here it is used to construct a quaternion which will rotate the x axis into the y axis.

```
Quaternion q;
q.make_rot(Vec3f(1,0,0), Vec3f(0,1,0));
```

Next, we construct a 4×4 matrix `m` and assign a translation matrix to the newly constructed matrix. After that we ask the quaternion to return a 4×4 matrix corresponding to its rotation. This rotation matrix is then multiplied onto `m`.

```
Mat4x4f m = translation_Mat4x4f(Vec3f(1,2,3));
m *= q.get_mat4x4f();
```

Just like for vectors, the subscript operator works on matrices. However, in this case there are two indices. Just using one index will return the ith row as a vector as shown on the first line below. On the second line we see that using two indices will get us an element in the matrix.

```
Vec4f v4 = m[0];
float c = m[0][3];
```

There is a number of constructors for vectors. The default constructor will create a null vector as we have already seen. We can also specify all the coordinates. Finally, we can pass just a single number a. This will create the $[a\ a\ a]^T$ vector. For instance, below we create the $[1\ 1\ 1]^T$ vector. Subsequently, this vector is multiplied onto `m`.

```
Vec3f p(1);
Vec3f p2 = m.mul_3D_point(p);
```

Note though that `m` is a 4×4 matrix so ... how is that possible? Well, we use the function `mul_3D_point` which, effectively, adds a $w = 1$ coordinate to p making it a 4D vector. This w coordinate is stripped afterwards. In practice, this means that the translation part of the matrix is also applied. There is a similar function `mul_3D_vector` if we want to transform vectors without having the translation. This function, effectively, sets $w = 0$.

Finally, CGLA is often used together with OpenGL although there is no explicit tie to the OpenGL library. However, we can call the `get` function of most CGLA entities to get a pointer to the contents. E.g. `p.get()` will return a pointer to the first float in the 3D vector `p`. This can be used with OpenGL's "v" functions as shown below.

```
glVertex3fv(p.get());
```

4 HMesh

HMesh's `Manifold` class is a halfedge based mesh data structure. This type of data structure restricts you to manifolds, but allows for general N-gons for arbitrary N and makes manipulation somewhat simpler than other types of data structures for meshes. Note that this particular implementation does not allow more than one loop per face.

The central element of the data structure is the halfedge. It is not the goal here to discuss the halfedge data structure, but it is probably helpful that you see the syntax for e.g. going from a halfedge to the next. It may also be useful to understand the basics of the data structure. The goal is to explain the basics and show how to navigate around the mesh.

First of all, we only access elements via iterators. An iterator can be thought of as a pointer, but it is in fact an iterator to an STL list. Given a

```
Manifold m;
```

we can loop over all its faces using

```
for(FaceIter f = m.faces_begin(); f != m.faces_end(); ++f)
  f->touched = -1;
```

The loop above visited each face f of m and assigned -1 to a variable called touched. The touched variables are simply integers, and a touched variable is associated with each face.

Note that we only refer to faces via their iterators. The same is true of vertices and halfedges. Vertices and halfedges also have touched variables, so the above code could be repeated exactly for vertices and halfedges. Here goes:

```
for(VertexIter v = m.vertices_begin();
     v != m.vertices_end(); ++v)
  v->touched = -1;

for(HalfEdgeIter h = m.halfedges_begin();
     h != m.halfedges_end(); ++h)
  h->touched = -1;
```

What would be the use of setting the touched values to -1? Well, if we want to keep track of whether we have visited a given halfedge, we can initially set the touched values of all halfedges to -1 and then as we visit a halfedge set its value to something different. The same idea can be applied to faces and vertices.

Another common use of the touched values is as indices into an array. In this way, you can store auxiliary data with the mesh. In fact, there is a function `enumerate_vertices()` which enumerates all vertices (so each has a unique positive (or zero) integer. Similar functions exist for faces and halfedges.

So far, we have only discussed touched, but halfedges, faces, and vertices contain other data members. Most of these are iterators pointing to other edges, faces, or vertices. In the following, we briefly mention each member of the datastructures and show how to get to it. To get the opposite halfedge, use

```
HalfEdgeIter h2 = h1->opp;
```

To get the next halfedge (in counterclockwise cycle around the vertex),

```
HalfEdgeIter h2 = h1->next;
```

the previous halfedge,

```
HalfEdgeIter h2 = h1->prev;
```

the vertex they point at,

```
VertexIter v = h->vert
```

and the face whose counter clockwise halfedge loop they are one segment of.

```
FaceIter f = h->face;
```

A vertex knows its position which is a Vec3f. To get the position, use

```
Vec3f pos = v->pos;
```

A vertex also knows one outgoing halfedge:

```
HalfEdgeIter h = v->he;
```

A face knows one halfedge in its counterclockwise cycle

```
HalfEdgeIter h = face->last;
```

To move around a face, we can repeatedly ask for the next halfedge. Likewise, to move around a vertex, we can repeatedly ask for `h->opp->next` but it is not elegant, and in both cases we need a stop condition. Circulators encapsulate the task of visiting all edges delimiting a face or radiating from a vertex in a nice(r) way.

Given

```
VertexIter v
```

we can go:

```
VertexCirculator vc(v);
Vec3f p(0.0);
for( ; !vc.end(); ++vc)
  p += vc.get_vertex()->pos;
p/= vc.no_steps();
```

So what does the code above do? It visits each neighboring vertex to a given vertex and computes the average position which is useful in a number of situations.

We can do the same thing with a FaceCirculator

```
FaceCirculator fc(f);
Vec3f p(0.0);
for( ; !fc.end(); ++fc)
  p += fc.get_vertex()->pos;
p/= fc.no_steps();
```

Internally, both the vertex and face circulators simply keep track of a single halfedge. So `get_vertex()` returns the vertex pointed to by that halfedge. There are similar functions for getting the halfedge itself, the face it points to or its opposite halfedge.

Use circulators whenever you need to go around a face or vertex, but not when you need more complex navigation which is not easily expressed as circulation. In that case it is probably easier to just use the next, prev, and opp to move around. To work with the functions and classes just described, you will need the following header files:

```
#include <HMesh/Manifold.h>
#include <HMesh/FaceCirculator.h>
#include <HMesh/VertexCirculator.h>
```

4.1 Extended Example: Edge Flipping

The following example demonstrates how to create a Manifold and add polygons (in this case triangles) and finally how to flip edges of a manifold. First, let us define some vertices

```
vector<Vec3f> vertices(3);
vertices[0] = p1;
vertices[1] = p2;
vertices[2] = p3;
```

and then a vector of faces. The faces vector contains the number of vertices of each face in the mesh we want tor produce. Initially, we want to make a mesh with just one single triangle, so we produce a vector of one element and set that element to 3.

```
vector<int> faces(1);
faces[0] = 3;
```

Next, we need the index vector. This vector is a long list of indices to vertices. For each face, we store the indices to its vertices in this vector. Since we just have a triangle, we store three vertices in the vector.

```
vector<int> indices(3);
indices[0]=0;
indices[1]=1;
indices[2]=2;
```

So, if we had had two triangles, we would have stored six indices. Finally, we call `build_manifold` with the indexed face set data, we have compiled.

```
build_manifold(mani,          // The triangle mesh.
 3,            // Number of vertices.
 &vertices[0],// Pointer to vertices.
 1,            // Number of faces.
 &faces[0],   // Pointer to faces.
 &indices[0]); // Pointer to indices.
```

The result of the above is a mesh with a single triangle. Note that `build_manifold` is in

```
#include <HMesh/build_manifold.h>
```

Next, we try to split an edge. We grab the first halfedge and split it: Now we have two triangles. Splitting the halfedge changes the containing triangle to a quadrilateral, and we call triangulate to divide it into two triangls again.

```
HalfEdgeIter h = m.halfedges_begin();
m.split_edge(h);
m.triangulate(h->face);
```

Now, we obtain an iterator to the first of our (two) triangles.

```
FaceIter f1 =m.faces_begin();
```

We create a vertex, `v`, at centre of `f1` and insert it in that face:

```
VertexIter v=m.create_vertex(centre(f1));
m.face_insert_point(f1,v);
```

The code below, marks one of a pair of halfedges (with the value 1).

```
for(HalfEdgeIter h = m.halfedges_begin(); h!=m.halfedges_end(); ++h)
  h->touched =0;
for(HalfEdgeIter h = m.halfedges_begin(); h!=m.halfedges_end(); ++h)
  if(h->opp->touched == 0)
    h->touched = 1;
```

Next, we set the `flipper` variable to point to the first of these halfedges:

```
flipper = m.halfedges_begin();
```

The long piece of code below examines a halfedge pointed to by `flipper`, and if it is not a boundary halfedge it tries to flip it. Then it increments the `flipper` variable until it lands on a halfedge marked with 1. If `flipper` reaches the end of the list of halfedges, we start over.

```
if(is_boundary(flipper))
  cout << "boundary edge: could not flip" << endl;
else
  if(!m.flip(flipper))
    cout << "could not flip" << endl;
do
{
  ++flipper;
  if(flipper==m.halfedges_end())
    flipper = m.halfedges_begin();
}
while(flipper->touched == 0);
```

4.2 Extended Example: Implicit Surface Polygonization

This example demonstrates how we can produce an HMesh Manifold from an implicitly defined object. Say, we have some function, `f`, which represents an object by what we sometimes call a characteristic function (`f<0` inside the object, `f>0` outside, `f=0` on the surface). We wish to extract the 0 set as a Manifold.

First, we create a 3D grid of floating point values as shown below.

```
RGrid<float> grid(Vec3i(N,N,N)); // Voxel grid on which to sample
```

We sample `f` on that grid

```
for(int i=0;i<N;++i)
  for(int j=0;j<N;++j)
    for(int k=0;k<N;++k)
      grid[Vec3i(i,j,k)] = f(i,j,k);
```

Next, we call a function which polygonizes the 0 set. The first two arguments are the grid and the manifold, the third argument is set to 0, since we want the zero level set, and the final argument tells the polygonizer to push the vertices onto the zero set (as opposed to leaving them in their initial position which is close to but not actually on the zero set).

```
cuberille_polygonize(grid, mani, 0.0, true);
```

In post-processing, we triangulate `mani` and remove ugly triangles. Caps are triangles with one big angle (close to 180 degrees). Needles are triangles with one very short edge.

```
triangulate(mani);
remove_caps_from_trimesh(mani, M_PI * 0.85);
remove_needles_from_trimesh(mani, 1e-2);
```

Note that you need the following header files:

```
#include <Geometry/RGrid.h>
#include <HMesh/volume_polygonize.h>
```

to use the grid and the polygonizer, respectively.

5 GLGraphics

GLGraphics contains a set of tools needed for visualization of geometrical objects. In particular, this namespace contains mesh rendering facilities, and in the following, a very simple example is given. This section is basically a walk through of the simplest GEL program that draws a mesh. Apart from GEL, we also use OpenGL and that requires a toolkit for connecting with the window system. With GEL programs, one generally uses GLUT, and this example is not an exception.

For starters, we need to include some files mostly from GLGraphics but also the header file for the mesh load function of HMesh and the GLEW header. Why incude glew.h rather than just gl.h? Well, HMesh/draw.h contains some draw functions which rely on shaders, and since gl.h is normally not up to date, it makes sense to use glew.h instead of gl.h even though it inflicts a dependency. We include GLGraphics/gel_glut.h rather than the normal glut.h. That is because the GEL version works the same on Windows, OSX, Linux and probably other platforms.

We also open the namespaces we will need and declare two globals: The view controller `view_ctrl` and the mesh `mani`. The view controller class is rather complex. It more or less insulates you from having to deal directly with the projections and transformations of OpenGL: It sets up a perspective projection and also the modelview transformation needed to view the object. From a user interface perspective, it defines a virtual trackball which allows you to rotate the object.

```
#include <GL/glew.h>
#include <GLGraphics/gel_glut.h>
#include <GLGraphics/draw.h>
#include <GLGraphics/GLViewController.h>
#include <HMesh/load.h>

using namespace std;
using namespace HMesh;
using namespace CGLA;
using namespace GLGraphics;

GLViewController* view_ctrl;
Manifold mani;
```

Below is the display function which is called from the GLUT event loop whenever an event that requires redrawing is received. All drawing takes place inside this function. In this case, it is simple since all we need to do is clear the screen (and depth buffer), set up a new modelview transformation with the view controller and then call the draw function followed by a swap of buffers.

The draw function sends the polygons to the graphics card. It is very inefficient and uses the fixed function pipeline. At a minimum one should draw the polygons to a display list, but for a simple example, this works. The buffer swap is because we use double buffering; in other words, we draw to the back buffer.

```
void display() {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    view_ctrl->set_gl_modelview();
    draw(mani);
    glutSwapBuffers();
}
```

The mouse callback below is called when a mouse event has occurred. This is where user input for the virtual trackball is registered. We check whether a mouse is pressed or released. If it is pressed, we detect which button and then “grab” the virtual trackball in order to rotate, zoom, or pan depending on which button is pressed. If the button is released, we inform the view controller of this.

```
void mouse(int button, int state, int x, int y)  {
    Vec2i pos(x,y);
    if (state==GLUT_DOWN)
    {
        if (button==GLUT_LEFT_BUTTON)
            view_ctrl->grab_ball(ROTATE_ACTION,pos);
        else if (button==GLUT_MIDDLE_BUTTON)
            view_ctrl->grab_ball(ZOOM_ACTION,pos);
        else if (button==GLUT_RIGHT_BUTTON)
            view_ctrl->grab_ball(PAN_ACTION,pos);
    }
    else if (state==GLUT_UP)
        view_ctrl->release_ball();
}
```

If the mouse moves with a button depressed, we register motion in the function below. The new mouse position is sent to the view controller which then, e.g., rotates the trackball accordingly. Note that this function ends by posting a redisplay event, i.e. informs GLUT that display should be called at the earliest convenience.

```
void motion(int x, int y) {
    Vec2i pos(x,y);
    view_ctrl->roll_ball(Vec2i(x,y));
    glutPostRedisplay();
}
```

Finally, in the main function, we first load the mesh and then setup glut. This mostly involves defining the callback function just described. We set up the view controller passing it window dimensions and the size and position of the bounding sphere of the object just loaded. Finally, we do some minimal OpenGL set up: Essentially clearing the depth buffer and enabling lights. Finally, we pass control to the GLUT event loop.

```
int main(int argc, char** argv) {
    string file = "bunny-little.x3d";
    load(file, mani);

    glutInitDisplayMode(GLUT_RGBA|GLUT_DOUBLE|GLUT_DEPTH);
    glutInitWindowSize(800, 800);
    glutInit(&argc, argv);
    glutCreateWindow("GEL Example Program");
    glutDisplayFunc(display);
    glutMouseFunc(mouse);
    glutMotionFunc(motion);

    Vec3f bsphere_center(0,0,0);
    float bsphere_radius = 3;
    mani.get_bsphere(bsphere_center,bsphere_radius);
    view_ctrl = new GLViewController(800,800,
        bsphere_center,bsphere_radius*2);

    glClearColor(0.0f, 0.0f, .5f, 0.f);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);

    glutMainLoop();
}
```

Note that the mesh can be either an X3D, OBJ, PLY, or OFF files. However, only the geometry is loaded not the textures. In the case of OBJ files, we also convert all polygons to triangles, because the TriMesh loader (see the geometry class) is in fact used.

Apart from the drawing functions and the viewcontroller, GLGraphics also contains other useful pieces: A few functions which greatly simplify shader loading and SOIL. SOIL is a small library for loading images. It is designed for OpenGL and makes it easy to e.g. save screen shots or load an image for use as a texture. While the need for image loading is fairly obvious, it might be more difficult to see the need for shader loading functions. However, it *is* a little tricky to make shader programs in C++ using GLSL but the problems are almost all silly little things. For instance, how do you robustly read from a text file? How do you compile a shader and check for errors? In OpenGL What is the difference between a GLSL program and a shader anyway?

The short answer to the last question is this: A ”shader” can be either a vertex shader, a geometry shader (in OpenGL 2.0) or a fragment shader. A ”program” is a linked combination of these three types of shaders. Note that you don’t have to have a geometry shader.

These functions attempt to obviate the need for an answer to the first two questions by providing an API for loading and compiling shaders and checking for errors. However, I don’t include functions for creating the program, attaching shaders and linking. Why not? Because the need for flexibility means that the API would be just as complex as just using the OpenGL functions directly! However, loading shaders and checking for errors in compiled shaders is different. It makes sense to wrap that. It also makes sense to wrap the error checking for programs that are linked, so there is a function for that too.

Since shader loading and error check sandwhich the two calls needed for compilation, the most important function in this API, `create_glsl_shader`, loads a shader, compiles it, checks for errors and returns the shader handle. There is also a version which creates a shader from a string.

There is some code below to illustrate usage.

```
GLuint vs = create_glsl_shader(GL_VERTEX_SHADER,
  shader_path, "tri.vert");
GLuint gs = create_glsl_shader(GL_GEOMETRY_SHADER_EXT,
  shader_path, "tri.geom");
GLuint fs = create_glsl_shader(GL_FRAGMENT_SHADER,
  shader_path, "tri.frag");

prog_P0 = glCreateProgram();

if(vs) glAttachShader(prog_P0, vs);
if(gs) glAttachShader(prog_P0, gs);
if(fs) glAttachShader(prog_P0, fs);

// Specify input and output for the geometry shader.
// Note that this must be done before linking the program.
glProgramParameteriEXT(prog_P0,GL_GEOMETRY_INPUT_TYPE_EXT,
  GL_TRIANGLES);
glProgramParameteriEXT(prog_P0,GL_GEOMETRY_VERTICES_OUT_EXT,3);
glProgramParameteriEXT(prog_P0,GL_GEOMETRY_OUTPUT_TYPE_EXT,
  GL_TRIANGLE_STRIP);

// Link the program object and print out the info log
glLinkProgram(prog_P0);
print_glsl_program_log(prog_P0);

// Install program object as part of current state
glUseProgram(prog_P0);

// Set the value of a uniform
glUniform2f(glGetUniformLocation(prog_P0,"WIN_SCALE"),
  win_size_x/2.0, win_size_y/2.0);
```

6 LinAlg

Sometimes the simple 2,3,4 dimensional vectors from CGLA just don't cut it. We often need to solve large linear systems. The LinAlg namespace contains some vector and matrix classes and an interface to Lapack. This provides fairly easy way to do many numerical computations.

In the example below, we find coefficients for $ax^2 + by^2 + cxy + dx + ey + f$ such that the surface contains six specific points. The coefficients are found by solving the linear system $\mathbf{Ax} = \mathbf{b}$ where the rows of the $\mathbf{A}$ matrix contains $x^2\ y^2\ xy\ x\ y\ 1$ computed from the xy positions of each point, $\mathbf{x}$ contains the unknown coefficients, and $\mathbf{b}$ on the right hand side contains the z values of the six points. Using LinAlg, we implement it as follows:

```
CMatrix A(6,6);
CVector b(6);

for(int i=0;i<6;++i)
{
    A.set(i,0,pts[i][0]*pts[i][0]);
    A.set(i,1,pts[i][1]*pts[i][1]);
    A.set(i,2,pts[i][0]*pts[i][1]);
    A.set(i,3,pts[i][0]);
    A.set(i,4,pts[i][1]);
    A.set(i,5,1);
    b[i] = pts[i][2];
}
CVector x;
LinearSolve(A,b,x);
```

While this example shows only the function for solving a system which should have a solution, there are also functions for solving overdetermined systems in the least square sense and finding the minimum norm solution to underdetermined systems using singular value decomposition. There is also a function for finding eigenvalues and eigenvectors of symmetric matrices and more.

7 Geometry

The Geometry namespace contains many different classes and functions. Roughly, we can divide the contents into

- Spatial data structures such as kd tress, bounding box hierarchies, and BSP trees.
- A Triangle mesh data structure called TriMesh. TriMesh is different from HMesh in that it only contains triangles. It is much more an API for real time rendering which also facilitates material properties and not just geometry.
- Classes and functions for dealing with volume data. HGrid and RGrid are, respectively, a class for hierarchically stored voxel grids (sparse grids) and regular grids. There is a number of functions for traversing voxel grids systematically and along rays.
- Surprisingly, you will also find my C++ port of Jules Bloomenthals isosurface polygonizer and an implementation of Luiz Velho's parametric surface tiler in this namespace.

One of the most frequently used tools in Geometry is the kD tree class which merits an example. In the code below, we create a kD tree which had `Vec3fs` as keys and integers as data. Note that since the kD tree is a template, we could use any other type as both key and data, however, only CGLA vectors have been tested as key types, but there are no restrictions on data types.

```
KDTree<Vec3f,int> tree;
for(int i=0;i<N;++i)
    {
        Vec3f p0;
        make_ran_point(p0);
        tree.insert(p0, i);
    }
tree.build();
```

Note also, that before we use the tree it needs to be built. This is because the internal representation is a balanced binary heap which it is easier to build at the end rather than maintain during insertion of new points. To look for the points near a given point, we can call

```
Vec3f p0;
// ...
std::vector<Vec3f> keys;
std::vector<int> vals;
float radius = 1.95f;
int N = tree.in_sphere(p0, radius , keys, vals);
```

This will find all points in a radius of 1.95 units from `p0`. The keys and values are stored in `keys` and `vals` when the functions returns. The return value of the function is the number of points found within the radius.

Sometimes, we simply want the point closest to a given point `p0` which is done using `tree.closest_point(p0, d, pos, x)` where the two first arguments indicate the point and the maximum search radius. The two last arguments are set to the key and data of the point found to be closest to `p0`. The function returns true if a closest point was found.

8 Util

The Util namespace contains a rather mixed bag of utilities many of which are quite useful and very diverse.

The XML parser for instance is written from scratch and the basis for our X3D loader although it will load any XML file.

`Grid2D` is often useful when we want to deal with e.g. simulation on a 2D grid.

The `Timer` class is nearly indispensable when we need to time something, and it is very easy to use.

`ArgExtracter` is a class for getting the command line arguments to a program.

There is more - for instance a resource manager template, some string utility templates and a hash table implementation.

Arguably, it would be more pretty if these pieces had been more thematically linked, but many are used a lot, and it seemed reasonable to have somewhere to stick various functions and classes that did not naturally belong elsewhere.

9 Authors and License

Many other people contribute, but the core of GEL was written (mostly) by Andreas Bæentzen (jab@imm.dtu.dk), and any bug fixes, contributions, or questions should be addressed to me, unless you know precisely who could take care of the problem.

I was considering putting GEL under the LGPL. But it is a long complex text. The longer any kind of document, the more chances for loopholes in my opinion. Instead, I list a few rules below. The most important one is that if you want to use GEL for some purpose, and it is not crystal clear whether it is against the rules, contact me. As for the rules:

You are allowed to use GEL for academic or commercial purposes. In particular, you are welcome to give GEL to students as a basis for academic work or to use it for your own applications.

The biggest limitation that I want to impose is that you cannot repackage GEL in any way. You are not allowed to distribute another library which contains GEL or parts of GEL unless you make it clear that this other library contains GEL. You are also not allowed to redistribute GEL in a changed form. If you want changes to be made, contact me.

Of course, neither I nor my employer will give you any money or be held responsible in any way, under any circumstances what so ever - no matter what sort of damage GEL might inflict upon you. Not that I can foresee GEL causing you any damage :-)

If anything is unclear, please contact me. In fact, if you want to use GEL for a bigger project, I'd appreciate an email to jab@imm.dtu.dk

In a project such as this, it is almost impossible to completely avoid building upon fragments of other peoples source code. GEL includes an obj loader based on work by Nate Robins. Some pieces of source code from Graphics Gems have also been used. Moreover, I have included rply by Diego Nehab, Princeton University, and the Simple OpenGL Image Loader by Jonathan Dummer. All of this amounts to only a fraction of the GEL source code and it should not be in violation of any license. In particular, SOIL and RPLY are under the MIT license and it is acceptable to include these packages as long as the copyright notice is retained (which it is.)