Visualization with OpenGL

Essential approaches to programming computer graphics with Open Graphics Language (OpenGL) graphics library are described. This document serves as the basis for exercises in PRACE Summer of HPC Visualization training. Rationale for giving introduction to OpenGL is that such knowledge is important when developing codes that require some specific visualization for which OpenGL can be handy. Programming Graphical Processing Unit (GPU) through *shaders* is an important technique to accelerate graphics and other embarrassing parallel problems. OpenGL evolved from immediate mode to GPU only processing with the advent of OpenGL Shading Language (GLSL). GLSL is used for tutorial without the tendency to introduce *photo-realism*as output but rather useful colors for scientific data exploration. Introduction to the subject is given by recipes to follow, discussing important techniques for visualization that can also be extended to general GPU programming for parallel computing. Instead of jumping to the latest OpenGL specification we use minimum required OpenGL 2.1 with the extensions currently available on modest hardware and still be able to use modern OpenGL 3.1+ programming principles.

Running this tutorial on Linux desktop one requires at least the OpenGL 2.1 graphics with the GLSL 1.2 and supporting libraries GL, GLU, GLUT, GLEW. This can be verified with the following commands:

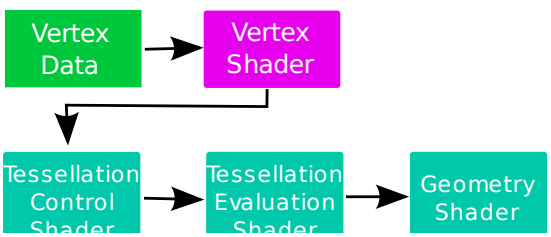
```
$ glxinfo |grep OpenGL.*version
OpenGL version string: 2.1 Mesa 8.0.5
OpenGL shading language version string: 1.20
$ ls /usr/include/GL/{glut.h,glew.h,gl.h,glu.h}
/usr/include/GL/glew.h  /usr/include/GL/glu.h
/usr/include/GL/gl.h    /usr/include/GL/glut.h
```

Introduction

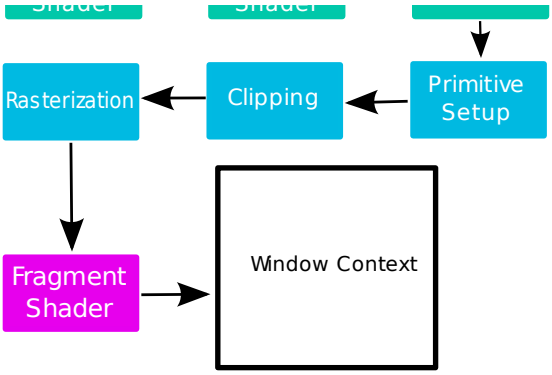
For the visualization of specific phenomena is usually not possible to use a general purpose visualization tools. Such cases occur especially in the visualization of engineering and physics problems. The modeling results are usually not only simple function plots but complex objects such as graphs, hierarchical structure, animation, motion mechanism, control channels, volume models of specific forms, ...

Through the time different standards were effective for computer graphics. This is mainly due to the complexity of implementation and closed code in the past. OpenGL remains the only widely accepted open standard, which was first introduced on Silicon Graphics workstations (SGI). There exist also a Microsoft Direct3D, which is limited to PCs with Windows and is not as easy to use as OpenGL, which is due to its openness and capacity provided on all operating systems and hardware platforms. OpenGL stagnated for some time with upgrades to the original SGI specification. Many extensions previously available from hardware vendors are now standardized with OpenGL 3+ where things dramatically changed. *Immediate mode* programming where communication from OS to GPU was regular practice and major obstacle to graphics performance. Programming knowledge of OpenGL 1.x is therefore not recommended for nowadays and can simply be forgotten and treated as legacy.

Modern OpenGL changed previously fixed rendering pipeline to fully programmable graphics pipeline as shown in Fig.1 Processors that transform input vertex data to the window context at the end are called *shaders*. The Vertex shader and the Fragment shader are most important in the rendering pipeline. To use rendering pipeline



as shown in Fig.1 one has to provide program for them as there is no default because they are essential part of every OpenGL program. Programming shaders is done in GLSL (OpenGL Shading Language) that is similar to C language with some predefined variables and reserved keywords that help describing communication between OS and GPU. Programs (for shaders) written in GLSL are compiled on-the-fly, assembled and transferred to GPU for execution.



OpenGL is designed as a hardware-independent interface between the program code and graphics accelerator. Hardware independence of OpenGL means also that in the language specification there is no support for control of window system events that occur with interactive programming. For such interactive control for each operating system were designed interfaces that connect the machine with the OpenGL system. Due to the specifics of different window systems (Windows, XWindow, MacOS, iOS, Android) it is required that for each system tailored techniques are used to call OpenGL commands in hardware. Portability is thus limited by graphical user interface (GUI) that handles OpenGL context (window). In order to still be able to write *portable programs* with otherwise limited functionality of the user interface, GLUT library (OpenGL Utility Toolkit) was created. It compensates all the differences between operating systems and introduces a unified methods of manipulating *events*. With the GLUT library it is possible to write portable programs that are easy to write and have sufficient capacity for simple user interfaces.

Legacy OpenGL

Basics of the OpenGL language are given in the (core) GL library. More complex primitives can be build by the GLU library (GL Utility) which contain the routines that use GL routines only. GLU routines contain multiple GL commands that are generally applicable and have therefore been implemented to ease OpenGL programming.

To get quickly introduced into OpenGL it is better to start with legacy (short and simple) program that will be later replaced with modern OpenGL after discussion that caused replacement with OpenGL 3.x. Unfortunately FORTRAN support for modern OpenGL is lacking bindings for GLEW. Legacy OpenGL programming in Fortran is still possible. Before we can dive in OpenGL we need to revise windowing systems and how they interact with users.

Events

All window interfaces (GUI) are designed to operate on the principle of *events*. Events are signals from the Window system to our program. Our program is fully responsible for the content of the window. Windowing system only assigns area (window). The contents of the window area must then be fully controlled. In addition to the window assignment the windowing system to sends messages (events) to our program. The most common messages are:

display
: The command asks for presentation of window contents. There are several possible occasions when this happens. For example, when another window reveals part of our window or when window is moved on the screen. Another example is when window is re-displayed after icon is being pressed at the taskbar. Interception of such events is mandatory, because every program must ensure that the contents of the window is restored window, when such event occurs.

reshape
: Command to our program that occurs when the size and/or shape of the window changes. In this case the content of the window must be provided for a new window size. Event occurs, inter alia, when the mouse resizes the window. Immediately after reshape, display event is sent.

keyboard

Commands coming from the keyboard.

mouse
: Describes the mouse buttons at their change when user pressed or released one of the buttons.

motion
: This command defines the motion tracking of the moving mouse with pressed button.

timer
: Program requests message after a certain time in order to change the contents of the window. The function is suitable for timed simulation (animation).

In addition to these events there exist some other too. In general it is not necessary that all events to a window are implemented in our program. It is our responsibility to decide which events will be used in the application. Usually program must notify windowing system which events will took over and for that window will receive events.

GLUT

For an abstraction of events (commands from the windowing system) we will use GLUT library (OpenGL Utility Toolkit). Many other GUI libraries are available (native and portable). GLUT falls into the category of simple operating/windowing system independent GUIs for OpenGL. An example of a minimal program that draws a single line is shown in Listing 1 (first.c). \lstinputlisting[caption=Drawing a line with OpenGL and GLUT., label=first.c]{first.c} Program in C language consists of two parts: the subroutine display and the main program. Program runs from the start in `main()` and at the end falls into endless loop `glutMainLoop` that calls registered subroutines when event occurs. Before falling into `glutMainLoop` we need to prepare drawing context.

Listing 1: `first.c`

```c
#include <GL/glut.h>

void display()
{
  glClear(GL_COLOR_BUFFER_BIT);
  glColor3f(1.0, 0.4, 1.0);
  glBegin(GL_LINES);
    glVertex2f(0.1, 0.1);
    glVertex3f(0.8, 0.8, 1.0);
  glEnd();
  glutSwapBuffers();
}

int main(int argc, char *argv[])
{
  glutInit(&argc,argv);

glutInitDisplayMode(GLUT_DOUBLE);
  glutCreateWindow("first.c GL
code");
  glutDisplayFunc(display);
  glutMainLoop();
  return 0;
}
```

Listing 2: `first.py`

```python
from OpenGL.GLUT import *
from OpenGL.GL import *
import sys

def display():
    glClear(GL_COLOR_BUFFER_BIT)
    glColor3f(1.0, 0.4, 1.0)
    glBegin(GL_LINES)
    glVertex2f(0.1, 0.1)
    glVertex3f(0.8, 0.8, 1.0)
    glEnd()
    glutSwapBuffers()

if __name__ == "__main__":
    glutInit(sys.argv)

glutInitDisplayMode(GLUT_DOUBLE)
    glutCreateWindow("first.py
GL code")
    glutDisplayFunc(display)
    glutMainLoop()
```

Structure of the program is usually very similar for all languages. Confer Listing 2 (first.py) rewritten in Python. All GLUT programs include commands in the following order:

- Include definitions of constants and functions for OpenGL and GLUT with the include statement.
- Initialize GLUT and setup other variables that are not directly related to OpenGL but rather to the object that is being visualized.

- Set window parameters such as initial position, size, type, bit plane memory.
- Create the window and name it.
- Setup the features of the OpenGL machine. These are usually commands `glEnable` for setup of lighting, materials, lists, and non-default behavior of OpenGL machine.
- Register call-back routines which will be called at events. Mandatory registration is just for `glutDisplayFunc(display)`. The rest are optional.
- The last command in `main` is a call to `glutMainLoop`, from which the program returns when the window is closed. At the same time the `main` program ends.

The command `glutInit` initializes GLUT library routines. It is followed by a request for window creation of a certain type. The constant `GLUT_DOUBLE` and the default `GLUT_RGB` suggests that we want a double-buffered window with a RGB space. Variable `window` keeps reference of window returned by `glutCreateWindow` and at the same time instructs the OS to set the window title. We have to tell to the window system which events the program will intercept. For example given, this is only `display` of the contents of the window. Call of the subroutine `glutDisplayFunc` instructs the `glutMainLoop` that whenever requests from OS for window redisplay occurs subroutine `display` should be called. Routines for handling events are usually called call-back *routines as it reside in program as standalone code snippets that are called auto-magically at certain events from the windowing system. When some event occurs is up to the windowing system that follows user interaction. The main point to emphasize here is that registered call-back routines do get additional information on the kind of event. For example of keyboard event we can get also mouse (x,y) coordinates besides the key pressed.*

We have seen that the subroutine `display` includes commands responsible for drawing in the window. All routines or functions there are OpenGL and have prefix `gl` to the name. Prefix is necessary to distinguish them and prevent name clash with other libraries. To understand the language one can interpret function names without prefixes and suffixes as the OpenGL is designed so, that the types of the arguments for all programming languages are similar. Subroutine `display` is therefore responsible for drawing the contents of the window. The `glClear` command clears the entire area of the window. When clearing we need to define precisely what we want to clear by argument given. In our case, this is `GL_COLOR_BUFFER_BIT`, which means clearing of all pixels in the color buffer.

The `glColor` command to sets the current color of graphic elements that will be drawn in subsequent commands. As an argument RGB color components are passed. Usually commands with multiple arguments are provide for different data types (integer, float, double) and some command can have different number of arguments for the same command. To distinguish them suffix is added. For the `glColor3f` suffix `3f` therefore means that the subroutine has three arguments of type float. Choice of the arguments type depends on application requirements. Programmer can freely choose data type that suits most without the need of data type conversion. In our example we have two variants for vertex command with different number of arguments of the same type. `glVertex2f` means that we are specifying just two coordinates while the third is by default z=0. Types of the arguments specified as the suffix letter are as follows:

f
: float in C language and `real*4` in Fortran.

d
: double for C and `real*8` in Fortran.

i
: integer (4 bytes).

s
: short integer in C and `integer*2` in Fortran.

Besides fixed number of arguments there are also functions that take as an argument vector (as a pointer to memory). For these the suffix contains letter `v` at the end. Below are some interpretations of suffixes:

3f
: Three arguments of `real`s follow as arguments.

3i

Three arguments of `integer`s follow as arguments.

3fv
: One argument as a vector that contains three `float`s follows.

Variety of different arguments for the same command can be in `glVertex` command where we can find

```
  glVertex2d,  glVertex2f,  glVertex2i, glVertex2s,  glVertex3d,
glVertex3f,
  glVertex3i,  glVertex3s,  glVertex4d, glVertex4f,  glVertex4i,
glVertex4s,
  glVertex2dv, glVertex2fv, glVertex2iv,glVertex2sv, glVertex3dv,
glVertex3fv,
  glVertex3iv, glVertex3sv, glVertex4dv,glVertex4fv, glVertex4iv,
glVertex4sv.
```

Large number of routines for the same function is performance and language related in order to waive the default conversion and thus provide a more comprehensive and faster code. For languages with name mangling like C++ one can find simpler OpenGL wrapped functions (eg. just `glVertex`) that don't affects performance. But as many languages does not have name mangling built into compiler such practise is not widespread. Besides specifying single vertex each time one can use `glVertexPointer` and points to memory where number of vertices of specified type exist. This can save us of some looping, but as this is essentially copying of system memory into OpenGL hardware engine, the performance is not really improved.

Drawing of graphic primitives in OpenGL occurs between two commands `glBegin(primitive type)` and `glEnd()`. Primitive type given as argument at the beginning specifies how subsequent vertices will be used for primitive generation. Instead of giving primitive type as number several predefined constant are provided within `include` directive to ease readability and portability of the OpenGL programs. Before providing vertex position one can change OpenGL engine primitive state such as current drawing `glColor3f` or `glNormal` that is per vertex property.

The last command in the `display` subroutine is `glutSwapBuffers()`. For applications in which the contents of the display changes frequently, it is most appropriate to use windows dual graphics buffers, which is setup by using the `GLUT_DOUBLE` at window initialization. The advantage of such drawing strategy is in the fact that while one buffer is used for current drawing the other is shown. Drawing thus occurs in the background and when buffer is ready for display we simply flip the buffers. In particular it should be noted that such behaviour is system dependent and once upon a time when the `GLUT_SINGLE` (without double buffers) with the `glFlush()` at the end was used instead. Nowadays `GLUT_DOUBLE` is usually used, which is most helpful with high frame-rate applications such as animation. Only simple primitives are used within OpenGL. Reason for that is mainly due to the requirement of performance and possible hardware acceleration. There are three types of simple primitives: points, lines, and triangles. Higher level primitives (like quadrilaterals) can be assembled from simple ones. Curves can be approximated by lines. Large surfaces can be tessellated with triangles. For complex surfaces (like NURBS) GLU library can be used to calculate vertices. The following line primitives are possible:

GL_LINES
: Pairs of vertices in a vertex stream create line segments.

GL_LINE_STRIP
: Vertex stream builds connected lines (polyline).

GL_LINE_LOOP
: Same as polyline above except that last vertex is connected by a line to the first.

Every surface can be assembled with triangles.

GL_TRIANGLES
: For each triangle three vertices are required from vertex stream.

GL_TRIANGLE_STRIP

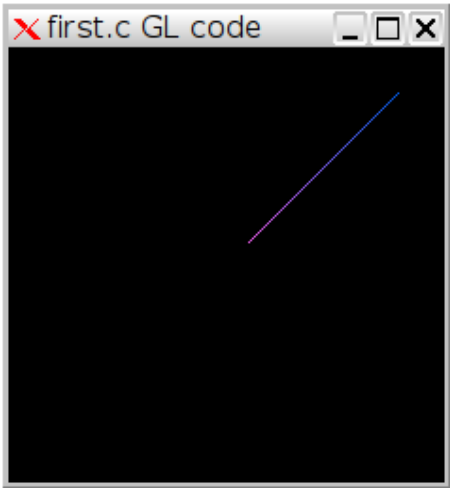
Strip of triangles. For first triangle three vertices are needed. For every additional vertex new triangle is created by using last two vertices.

GL_TRIANGLE_FAN
Triangles are added to the first one by using first and last vertex to create a triangle fan.

Exercises #1:

1. Create the following `first.c` using your favorite editor.

```
#include <GL/glut.h>

void display()
{
  glClear(GL_COLOR_BUFFER_BIT);
  glColor3f(1.0, 0.4, 1.0);
  glBegin(GL_LINES);
    glVertex2f(0.1, 0.1);
    glVertex3f(0.8, 0.8, 1.0);
  glEnd();
  glutSwapBuffers();
}

int main(int argc, char *argv[])
{
  glutInit(&argc,argv);
  glutInitDisplayMode(GLUT_DOUBLE);
  glutCreateWindow("first.c GL code");
  glutDisplayFunc(display);
  glutMainLoop();
  return 0;
}
```



Create the `Makefile` to build your program.

```
CFLAGS=-Wall -g
LDFLAGS=-lGL -lGLU -lglut -lGLEW

ALL=first
default: $(ALL)

first : first.o

clean:
	rm -rf  *.o *~ [!m]*.obj core* $(ALL)
```

Beware that Makefile is TAB aware. So the last line should contain TAB indentation and not spacing.

Make and run the program with

```
make
./first
```

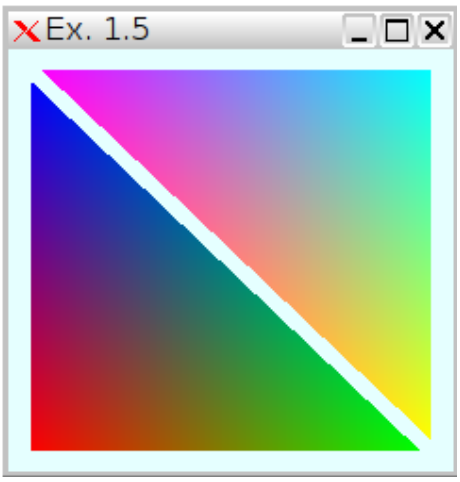
2. Add RGB color to vertices with `glColor3f(0.0, 0.4, 1.0);`.
3. Replace single line drawing in `display()` with the following snippet

```
   GLfloat vertices[][2] = {
    { -0.90, -0.90 }, // Triangle 1
    {  0.85, -0.90 },
    { -0.90,  0.85 },
    {  0.90, -0.85 }, // Triangle 2
```

```
    {  0.90,  0.90 },
    { -0.85,  0.90 }
  };
```

and try to draw two wireframe triangles in a loop. Change primitive to `GL_LINE_LOOP`.

4. Draw two primitives with `GL_TRIANGLES`.
5. Add different color to each vertex.




```
GLfloat color[][3] = {
 {1, 0, 0}, {0, 1, 0}, {0, 0, 1},
 {1, 1, 0}, {0, 1, 1}, {1, 0, 1}};
```

6. Replace loop with the following

```
  glVertexPointer(2, GL_FLOAT, 0,
&vertices[0][0]);

glEnableClientState(GL_VERTEX_ARRAY);
  glDrawArrays(GL_TRIANGLES, 0, 6);

glDisableClientState(GL_VERTEX_ARRAY);
```

How can we add color to vertices? See glColorPointer and glEnableClientState.

7. Change background to `glClearColor(0.9,1,1,1.0);` and suggest initial window in `main()`

```
glutInitWindowSize(512, 512);
glutInitWindowPosition((glutGet(GLUT_SCREEN_WIDTH)-512)/2,
                       (glutGet(GLUT_SCREEN_HEIGHT)-512)/2);
```

8. Add keyboard event to quit the program when pressing ESCape key with keycode 27 by adding callback function

```
void keyboard(unsigned char key, int x, int y)
{
  if (key == 27)
        exit(0);
}
```

and registering event within `main()` by `glutKeyboardFunc(keyboard);`. Some prefer `key == 'q'`, though.

Modern OpenGL

Immediate mode programming with `glBegin` and `glEnd` was removed from OpenGL 3.x as such transmission of vertex streams and its attributes (colors, normals, ...) from system memory to GPU is considered as a major performance drawback. Display lists were previously used to save stream of OpenGL calls that also included vertex data and was just replayed at redraw. But this is inherently sequential operation that blocked parallel vertex processing. Requirement to store vertex arrays to GPU directly as an *object* can solve problem described. Storing vertex arrays into GPU also means that manipulation on them to build the model should be inside the GPU. Legacy OpenGL included many modelling *utilities for transforming world coordinates into viewport. Transformations of coordinate systems in 3D space allowed manipulate model stack easily with `glPushMatrix` and `glPopMatrix` commands. But similarly to `glBegin`/`glEnd` such manipulations are not used outside GPU anymore. Instead all operations on vertex data is transferred to* vertex shader. *There operations on data can be performed with standard vector math in homogeneous coordinates.*

We extend previous exercise with example that introduces OpenGL 3.x techniques:

- OpenGL Shading Language (GLSL 1.2) where simple vertex and fragment shader are required.
- Vertex Aray Objects (VAOs) and vertex buffers (VBOs) stored in GPU.

Create `triangle.c` and update `Makefile` with new target

```
#include <stdio.h>
#include <stdlib.h>
#include <GL/glew.h>
#include <GL/glut.h>

GLuint program;
GLuint vbo_vertices;
GLint attribute_coord2d;

static const GLchar * vertex_shader[] = {
  "attribute vec2 coord2d;" // input vertex position
  "void main()"
  "{"
  "  gl_Position = gl_ModelViewProjectionMatrix*vec4(coord2d, 0.0,
1.0);"
  "}"
};
static const GLchar * fragment_shader[] =
  {"void main()"
   "{"
   " gl_FragColor = vec4(0.4,0.4,0.8,1.0);"
   "}"
  };

void create_shaders()
{
  GLuint v, f;

  v = glCreateShader(GL_VERTEX_SHADER);
  f = glCreateShader(GL_FRAGMENT_SHADER);
  glShaderSource(v, 1, vertex_shader, NULL);
  glShaderSource(f, 1, fragment_shader, NULL);
  glCompileShader(v);
  glCompileShader(f);
  program = glCreateProgram();
  glAttachShader(program, f);
  glAttachShader(program, v);
  glLinkProgram(program);
  glUseProgram(program);

  attribute_coord2d = glGetAttribLocation(program, "coord2d");
  if (attribute_coord2d == -1) {
    fprintf(stderr, "Could not bind attribute coord2d\n");
  }
  glEnableVertexAttribArray(attribute_coord2d);
}

void send_buffers_to_GPU(void)
{
  GLuint vertex_array_object;
  glGenVertexArrays(1, &vertex_array_object);
  glBindVertexArray(vertex_array_object);

  GLfloat vertices[][2] = {
    { -0.90, -0.90 }, // Triangle 1
    {  0.85, -0.90 },
```

```
    { -0.90,  0.85 },
    {  0.90, -0.85 }, // Triangle 2
    {  0.90,  0.90 },
    { -0.85,  0.90 }
  };

  glGenBuffers(1, &vbo_vertices);
  glBindBuffer( GL_ARRAY_BUFFER, vbo_vertices);
  glBufferData( GL_ARRAY_BUFFER, sizeof(vertices), vertices,
GL_STATIC_DRAW);
}


void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT);
  glBindBuffer(GL_ARRAY_BUFFER, vbo_vertices);
  glVertexAttribPointer(attribute_coord2d, 2, GL_FLOAT, GL_FALSE, 0,
NULL);
  glDrawArrays(GL_TRIANGLES, 0, 6); // Draw 6 vertices
  glutSwapBuffers();
}

int main(int argc, char **argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
  glutCreateWindow("GLSL Intro");
  glutDisplayFunc(display);
  glewInit();
  if (!glewIsSupported("GL_VERSION_2_0"))
   {
     printf("GLSL not supported\n");
     exit(EXIT_FAILURE);
   }
  glClearColor(0.9,1.0,1.0,1.0);
  send_buffers_to_GPU();
  create_shaders();
  glutMainLoop();
  return EXIT_SUCCESS;
}
```

Exercises #2

1. To be able to continue and not get lost introduce shader compiler logs in case of compilation errors by adding the following code into `create_shaders()` right at after vertex shader compilation:

```
GLint compiled;
glGetShaderiv(v, GL_COMPILE_STATUS, &compiled
);
if ( !compiled ) {
  GLsizei  maxLength, length;
  glGetShaderiv( v, GL_INFO_LOG_LENGTH,
&maxLength );
  GLchar* log = malloc(sizeof(GLchar)*
(maxLength+1));
  glGetShaderInfoLog(v,  maxLength, &length,
log);
  printf("Vertex Shader compilation failed:
%s\n", log);
```

```
  free(log);
}
```

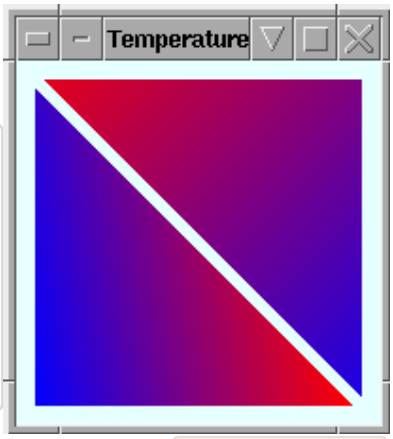
Do not forget to repeat the same thing for fragment shader. Add linker debugging

```
GLint linked;
glGetProgramiv(program, GL_LINK_STATUS, &linked );
if ( !linked ) {
  GLsizei len;
  glGetProgramiv(program, GL_INFO_LOG_LENGTH, &len );
  GLchar* log = malloc(sizeof(GLchar)*(len+1));
  glGetProgramInfoLog(program, len, &len, log );
  printf("Shader linking failed: %s\n", log);
  free(log);
}
```

Create some error to verify if it works. For general (core) OpenGL errors we can use the following `glcheck()` utility at suspicious places.

```
#define glcheck() {GLenum s; if ((s=glGetError()) != GL_NO_ERROR)
\
                fprintf (stderr, "OpenGL Error: %s at line %d\n",
\
                         gluErrorString(s),  __LINE__);}
```

2. Copy `triangle.c` into `temperature.c` and introduce vertex temperature with additional array and buffer at the end of `send_buffers_to_GPU()`




```
GLfloat vertex_temperature[] = {0, 1, 0.2,
0.1, 0.5, 0.9};
glGenBuffers(1, &vbo_temperature);
glBindBuffer(GL_ARRAY_BUFFER,
vbo_temperature);
glBufferData(GL_ARRAY_BUFFER,
sizeof(vertex_temperature),
           vertex_temperature,
GL_STATIC_DRAW);
```

and adding corresponding global attribute and VBOs IDs at the top of `temperature.c` so that global section reads:

```
GLuint program;
GLuint vbo_vertices;
GLuint vbo_temperature;
GLint attribute_coord2d;
GLint attribute_temperature;
```

Replace shaders with

```
static const GLchar * vertex_shader[] = {
  ""
  "attribute float temperature;" // custom variable along with
vertex position
  "varying float  t;" // communicate between the vertex and the
fragment shader
  "void main()"
  "{"
  "  t = temperature;"
  "  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;"
  "}"
```

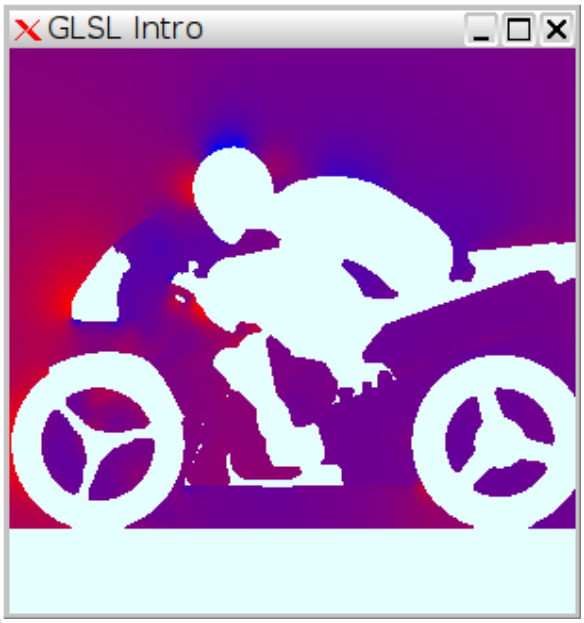
```
};
static const GLchar * fragment_shader[] = {
  "vec3 Cool = vec3(0, 0, 1);" // Red
  "vec3 Hot  = vec3(1, 0, 0);" // Blue
  "varying float t;" // Interpolated by fragment
  "void main()"
  "{"
  "  vec3 color = mix(Cool, Hot, t);"  // use the built-in mix()
function
  "  gl_FragColor = vec4(color, 1.0);" // append alpha channel
  "}"
};
```

Bind temperature buffer and specify data layout within `display()` just before `glDrawElements()` with

```
glBindBuffer(GL_ARRAY_BUFFER, vbo_temperature);
glVertexAttribPointer(attribute_temperature, 1, GL_FLOAT,
GL_FALSE, 0, NULL);
```

What happens if we don't enable temperature vertex array? Confer temperature.c attached if having troubles.

3. Add additional custom vertex array for the pressure. Change the temperature array to have values in Celsius for water boiling range [0-100]°C. Pressure should be in the range of [0-1] MPa. Scaling to color range [0-1] should be done in shaders. Toggle between them with the keyboard event by using the keys `'p'` and `'t'` that `glEnableVertexAttribArray()` and `glDisableVertexAttribArray()` corresponding vertex attribute arrays.
4. Fetch pressure data as 2D y-slice in file p_yNormal.vtk formatted in VTK that can be quickly read by the following code:




```
GLfloat *point;   int points;
GLuint *triangle; int
triangles;
GLfloat *pressure;

void read_VTK_pressure(const
char *filename)
{
  char line[80];
  int i; FILE *f;
  f = fopen(filename, "r");
  while(fgets(line, 80, f))
    {
      if (strstr(line,
"POINTS"))
        {
          float dummy_y;
          points =
atof(line+7);
          point =
malloc(points*2*sizeof(float));
          pressure =
malloc(points*sizeof(float));
          assert(point !=
NULL && pressure != NULL);
          for(i = 0; i <
points; ++i)
            fscanf(f, "%f %f
%f", &point[i*2], &dummy_y,
&point[i*2+1]);
```

```
        }
      else if (strstr(line, "POLYGONS"))
        {
          triangles = atof(line+9);
          triangle = malloc(triangles*3*sizeof(GLuint));
          for (i = 0; i < triangles; ++i)
            {
              int n;
              fscanf(f, "%d %d %d %d", &n, &triangle[i*3],
                     &triangle[i*3+1], &triangle[i*3+2]);
            }
        }
      else if (strstr(line, "FIELD"))
        {
          fgets(line, 80, f); // skip: p 1 27582 float
          for (i = 0; i < points; ++i)
            fscanf(f, "%f", &pressure[i]);
        }
    }
  fclose(f);
  printf("Read %d points for %d triangles for field %s\n",
         points, triangles, filename);
}
```

Insert this code into temperature.c and rename temperature with pressure everywhere. We will be drawing indexed so the last lines of the `display()` now reads

```
  glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo_elements);
  glDrawElements(GL_TRIANGLES, triangles*3, GL_UNSIGNED_INT, 0);
  glutSwapBuffers();
```

Sending element data buffers to GPU is slightly changed by using `GL_ELEMENT_ARRAY_BUFFER` instead of `GL_ARRAY_BUFFER` in subroutine

```
void send_buffers_to_GPU(void)
{
  GLuint vertex_array_object;
  glGenVertexArrays(1, &vertex_array_object);
  glBindVertexArray(vertex_array_object);

  glGenBuffers(1, &vbo_vertices);
  glBindBuffer(GL_ARRAY_BUFFER, vbo_vertices);
  glBufferData(GL_ARRAY_BUFFER, points*2*sizeof(GLfloat), point,
GL_STATIC_DRAW);

  glGenBuffers(1, &vbo_pressure);
  glBindBuffer(GL_ARRAY_BUFFER, vbo_pressure);
  glBufferData(GL_ARRAY_BUFFER, points*sizeof(GLfloat), pressure,
GL_STATIC_DRAW);

  glGenBuffers(1, &ibo_elements);
  glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo_elements);
  glBufferData(GL_ELEMENT_ARRAY_BUFFER,
triangles*3*sizeof(GLuint),
               triangle, GL_STATIC_DRAW);
}
```

Positioning (translation) of the motorbike and scaling of the pressure [-300..200] to color mix range [0-1] can be done in shaders directly. Confer final pressure.c if having trubles with coding.

Interactivity

Assemble the following Utah teapot model and attached virtual trackball.h and trackball.c sources from SGI. Teapot is built-in model for testing purposes in GLUT and uses legacy `glBegin()`/`glEnd()` commands and surface normals. Similarly deprecated GLSL usage of `gl_Vertex` and `gl_Normal` built-in input vertex attributes is used in `vertex_shader[]`. Nevertheless it is a good starting point for viewing applications.


```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <GL/glew.h>
#include <GL/glut.h>
#include "trackball.h"

GLuint program;

static const GLchar *
vertex_shader[] ={"\
varying vec3 normal,
lightDir;\
uniform mat4 RotationMatrix;\
void main()\
{          \

lightDir=normalize(vec3(gl_Lig

normal=normalize(gl_NormalMatr
  gl_Position =
gl_ProjectionMatrix * \

RotationMatrix*gl_ModelViewMat
}"};

static const GLchar *
fragment_shader[] ={"\
/* simple toon fragment
shader */\
/*
www.lighthouse3d.com
*/\
\
varying vec3 normal,
lightDir;\
\
void main()\
{\
        float intensity;\
        vec3 n;\
        vec4 color;\
\
        n =
```

```
normalize(normal);\
        intensity = max(dot(lightDir,n),0.0);\
        if (intensity > 0.98)\
                color = vec4(0.8,0.8,0.8,1.0);\
        else if (intensity > 0.5)\
                color = vec4(0.4,0.4,0.8,1.0);\
        else if (intensity > 0.25)\
                color = vec4(0.2,0.2,0.4,1.0);\
        else\
                color = vec4(0.1,0.1,0.1,1.0);\
        gl_FragColor = color;\
}"};

void create_shaders()
{
  GLuint v, f;

  v = glCreateShader(GL_VERTEX_SHADER);
  f = glCreateShader(GL_FRAGMENT_SHADER);
  glShaderSource(v, 1, vertex_shader, NULL);
  glShaderSource(f, 1, fragment_shader, NULL);
  glCompileShader(v);
  GLint compiled;
  glGetShaderiv(v, GL_COMPILE_STATUS, &compiled );
  if ( !compiled ) {
    GLsizei  maxLength, length;
    glGetShaderiv( v, GL_INFO_LOG_LENGTH, &maxLength );
    GLchar* log = malloc(sizeof(GLchar)*(maxLength+1));
    glGetShaderInfoLog(v,  maxLength, &length, log);
    printf("Vertex Shader compilation failed: %s\n", log);
    free(log);
  }
  glCompileShader(f);
  glGetShaderiv(f, GL_COMPILE_STATUS, &compiled );
  if ( !compiled ) {
    GLsizei  maxLength, length;
    glGetShaderiv( f, GL_INFO_LOG_LENGTH, &maxLength );
    GLchar* log = malloc(sizeof(GLchar)*(maxLength+1));
    glGetShaderInfoLog(f,  maxLength, &length, log);
    printf("Fragment Shader compilation failed: %s\n", log);
    free(log);
  }
  program = glCreateProgram();
  glAttachShader(program, f);
  glAttachShader(program, v);
  glLinkProgram(program);
  GLint linked;
  glGetProgramiv(program, GL_LINK_STATUS, &linked );
  if ( !linked ) {
    GLsizei len;
    glGetProgramiv(program, GL_INFO_LOG_LENGTH, &len );
    GLchar* log = malloc(sizeof(GLchar)*(len+1));
    glGetProgramInfoLog(program, len, &len, log );
    printf("Shader linking failed: %s\n", log);
    free(log);
  }
  glUseProgram(program);
}


float lpos[4] = {1, 0.5, 1, 0};
GLfloat m[4][4]; // modelview rotation matrix
```

```
float last[4], cur[4]; // rotation tracking quaternions
int width, height, beginx, beginy;
float p1x, p1y, p2x, p2y;

void display(void) {
  GLuint location = glGetUniformLocation(program, "RotationMatrix");
  build_rotmatrix(m, cur);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glLightfv(GL_LIGHT0, GL_POSITION, lpos);
  if( location >= 0 )
    glUniformMatrix4fv(location, 1, GL_FALSE, &m[0][0]);
  glutSolidTeapot(0.6);
  glutSwapBuffers();
}

void reshape (int w, int h)
{
  double l = 1;
  width=w;  height=h;
  glViewport (0, 0, w, h);
  glMatrixMode (GL_PROJECTION);
  glLoadIdentity();
  glOrtho(-l, l, -l, l, -l, l);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
}

void keys(unsigned char key, int x, int y)
{
   if (key == 27 || key == 'q')
         exit(0);
}

void mouse(int button,int state, int x, int y)
{
  beginx = x;
  beginy = y;
}

void motion(int x,int y)
{
  p1x = (2.0*beginx - width)/width;
  p1y = (height - 2.0*beginy)/height;
  p2x = (2.0 * x - width) / width;
  p2y = (height - 2.0 * y) / height;
  trackball(last, p1x, p1y, p2x, p2y);
  add_quats(last, cur, cur);
  beginx = x;
  beginy = y;
  glutPostRedisplay();
}

int main(int argc, char **argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);
  glutInitWindowSize(512, 512);
  glutInitWindowPosition((glutGet(GLUT_SCREEN_WIDTH)-512)/2,
                         (glutGet(GLUT_SCREEN_HEIGHT)-512)/2);
  glutCreateWindow("Use mouse to rotate");

  trackball(cur, 0.0, 0.0, 0.0, 0.0);
```

```
  glutDisplayFunc(display);
  glutReshapeFunc(reshape);
  glutMouseFunc(mouse);
  glutMotionFunc(motion);
  glutKeyboardFunc(keys);

  glEnable(GL_DEPTH_TEST);
  glClearColor(1.0,1.0,1.0,1.0);
  glewInit();
  if (!glewIsSupported("GL_VERSION_2_0"))
   {
     printf("GLSL not supported\n");
     exit(EXIT_FAILURE);
   }
  create_shaders();
  glutMainLoop();
  return EXIT_SUCCESS;
}
```

To build two sources we add the following line to `Makefile`:

```
teapot: teapot.o trackball.o
```

Exercises #3

1. OK, it rotates. But how come the light rotates with the teapot? I'm pretty sure that the light is not rotated while the teapot vertices are. Answer: Take a look into the `normal`. You need to rotate the vertex normal in the `vertex_shader[]` too! With a code like:

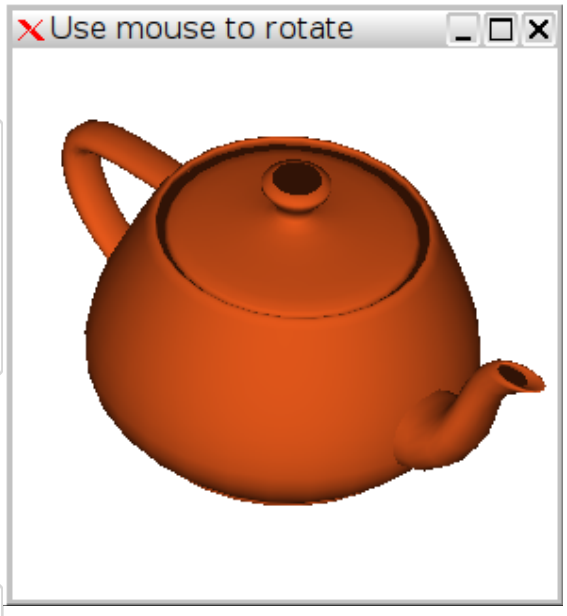
   ```
   vec4 n = RotationMatrix*vec4(gl_NormalMatrix*gl_Normal, 1);\
   normal = normalize(n.xyz); \
   ```

2. Introduce zoom in/out functionality of the viewer by adding mouse wheel events to the end of `mouse()`

   ```
   if (button == 3 && state ==
   GLUT_DOWN)
     { zoom *= 1.1;
   glutPostRedisplay(); }
   else if (button == 4 && state
   == GLUT_DOWN)
     { zoom /= 1.1;
   glutPostRedisplay(); }
   ```

   and introduction of global variable `float zoom = 1.0;` that is communicated to GPU by additional `uniform float Zoom;` in the `vertex_shader[]`. Last line is replaced by

   ```
     gl_Position =
   gl_ProjectionMatrix *
   RotationMatrix \
      *
   gl_ModelViewMatrix*vec4(Zoom*gl_
   1.0); \
   ```



In the `display()` we send `zoom` to GPU before drawing the `glutSolidTeapot()` by adding

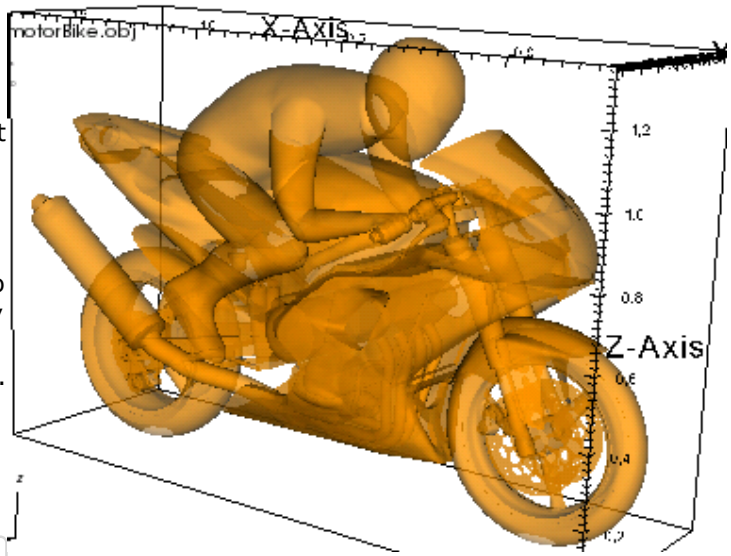
```
location = glGetUniformLocation(p, "Zoom");
if (location >= 0)  glUniform1f(location, zoom);
```

3. Simplify the cartoon shader to Gouraud shader by using just greyscale `gl_FragColor = vec4(intensity);` or copper mix

```
 vec4 copper_ambient = vec4(0.191250, 0.073500, 0.022500,
1.000000); \
 vec4 copper_diffuse = vec4(0.703800, 0.270480, 0.082800,
1.000000); \
 gl_FragColor = copper_ambient + intensity*copper_diffuse;\
```

Reading Objects

Sometimes we hit limitations of the visualisation tools for the data that we want to visualize. For example motorBike.obj from OpenFOAM contains object groups that we want to show colored separately and not as whole. Neither VisIt and ParaView can read Wavefront OBJ file with group separation. We are forced to convert motorBike.obj into bunch of files and read them one by one. The following `wavefront.c` converts motorBike.obj into 67 files. Try to open them in VisIt and ParaView. Note that we need to compensate vertex counting that starts with 1 and not with 0.

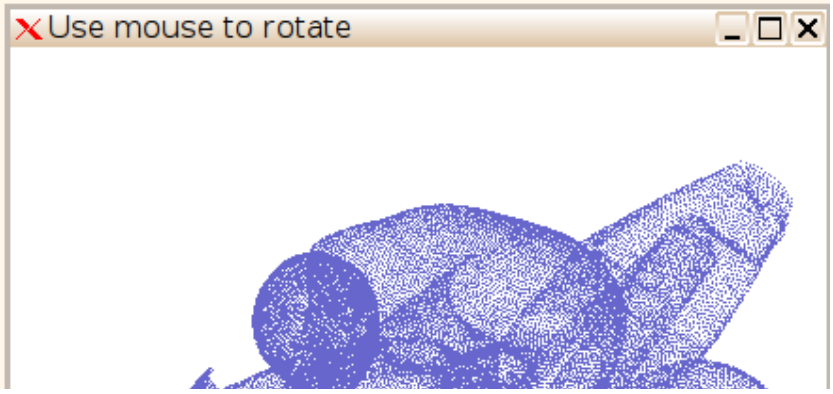



```
#include <stdio.h>

#define MaxVertices 400000
#define MaxFaces
400000
#define MaxGroups   100

float
vertex[MaxVertices*3];
unsigned int
face[MaxFaces*3];
char
group_name[MaxGroups][80];
int
start_face[MaxGroups];

int vertices = 0;
int faces    = 0;
int groups   = 0;

void read_wavefront(const
char *filename)
{
  char line[80];
  FILE *f =
fopen(filename, "r");
```

```
  while(fgets(line, sizeof(line), f))
    switch(line[0])
      {
      case 'v':
        sscanf(&line[1],  "%f %f %f", &vertex[vertices*3],
               &vertex[vertices*3+1], &vertex[vertices*3+2]);
        ++vertices;
        break;
      case 'g':
        sscanf(&line[1], "%s", group_name[groups]);
        start_face[groups++] = faces;
        break;
      case 'f':
        sscanf(&line[1],  "%d %d %d", &face[faces*3],
               &face[faces*3+1], &face[faces*3+2]);
        --face[faces*3]; --face[faces*3+1];
        --face[faces*3+2]; ++faces;
        break;
      }
  fclose(f);
  start_face[groups] = faces;
  printf("Read %d vertices and %d faces within %d groups from %s\n",
         vertices, faces, groups, filename);
}

void write_wavefront(int group_number)
{
  int i = 0; char n[80], *p = group_name[group_number];
  while (*p != '%' && *p != '\0') n[i++] = *p++; // remove % from name
  n[i++] = '.'; n[i++] = 'o'; n[i++] = 'b'; n[i++] = 'j'; n[i] = '\0';
  FILE *f = fopen(n, "w"); fprintf(f, "# Wavefront OBJ file\n");
  for (i = 0; i < vertices; i++)
    fprintf(f, "v %g %g %g\n", vertex[i*3], vertex[i*3+1],
vertex[i*3+2]);
  fprintf(f, "g %s\n", group_name[group_number]);
  for (i = start_face[group_number]; i < start_face[group_number+1];
++i)
    fprintf(f, "f %d %d %d\n", face[i*3]+1, face[i*3+1]+1,
face[i*3+2]+1);
  fclose(f);
}

int main(int argc, char **argv)
{
  int i;
  read_wavefront("motorBike.obj");
  for(i = 0; i < groups; i++) write_wavefront(i);
  return 0;
}
```

Exercises #4

1. Insert wavefront.c into extended teapot.c interactivity example above and save it as `motorbike.c`. Verify that there are no compile problems and that the `main()` contains



`read_wavefront("motorBike.obj");`. Disable lengthy wavefront saving in `main()`.

2. Instead of drawing of the teapot in `display()` we will draw single vertex array object as a point cloud with adding

```
//glutSolidTeapot(0.6);
glVertexPointer(3, GL_FLOAT, 0, vertex);
glEnableClientState(GL_VERTEX_ARRAY);
glDrawArrays(GL_POINTS, 0, vertices);
glDisableClientState(GL_VERTEX_ARRAY);
```

that pushes 132871 vertices (1.5MB) from client memory to GPU on every redraw. Better approach would be to follow VBOs principles by generating vertex buffer in GPU as `temperature.c` example.

3. We see that rotation of the motorbike around the front wheel is not really nice. To compensate we translate all points in the `vertex_shader[]` by adding `vec3(-0.75, 0, -0.7);` to every vertex in world coordinates and thus moving motor bike to origin. Last part of the `vertex_shader[]` now reads:

```
vec3 position = gl_Vertex.xyz + vec3(-0.75, 0, -0.7); \
gl_Position = gl_ProjectionMatrix * RotationMatrix \
 * gl_ModelViewMatrix*vec4(Zoom*position, 1.0); \
```

4. Instead of drawing points use indexed drawing of faces by using

```
// glDrawArrays(GL_POINTS, 0, vertices);
glDrawElements(GL_TRIANGLES, faces*3, GL_UNSIGNED_INT, face);
```

Result is silhouette as we did't provided vertex normals.

5. Calculate vertex normals by averaging nearby faces normals that are calculated with cross product. We need to add normal array as a global variable

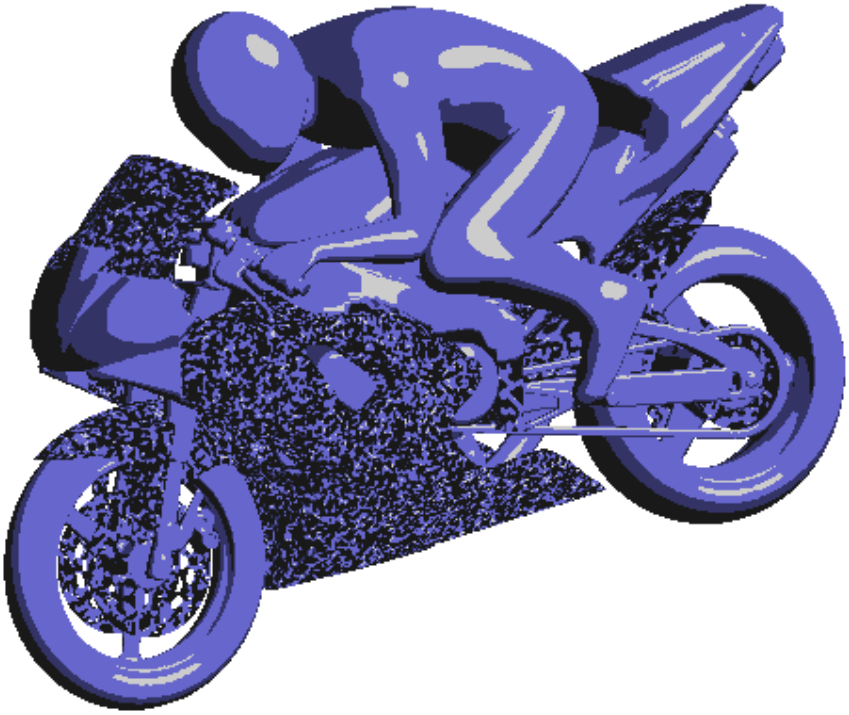
```
float       normal[MaxVertices*3];
```

and add glNormalPointer with glEnableClientState(GL_NORMAL_ARRAY) to the last part of `display()` that now reads:

```
//glutSolidTeapot(0.6);
glNormalPointer(GL_FLOAT, 0, normal);
glVertexPointer(3, GL_FLOAT, 0, vertex);
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_NORMAL_ARRAY);
glDrawElements(GL_TRIANGLES, faces*3, GL_UNSIGNED_INT, face);
glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_NORMAL_ARRAY);
```

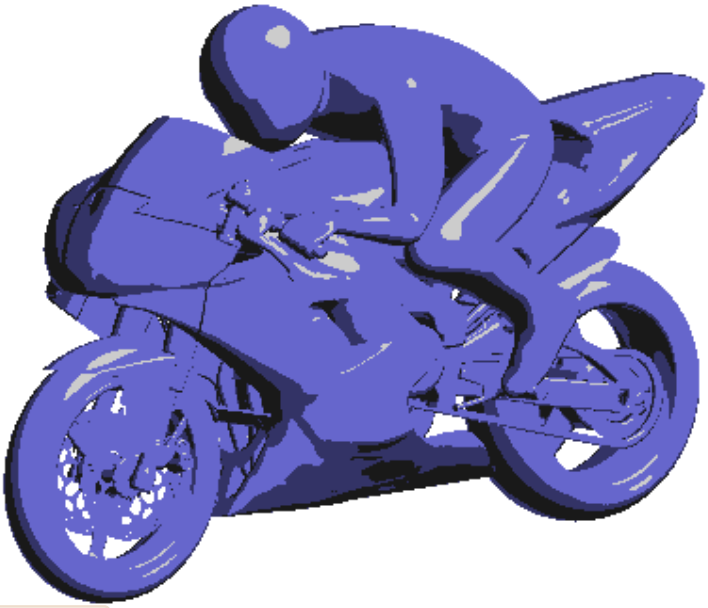
For calculation of normals we quickly invent the following subroutine:

```
void
calculate_nor
{
  int i;
  for(i =
0; i <
vertices*3;
++i)

normal[i] =
0.0;
  for(i =
0; i <
faces; ++i)
    {
      int
p1 =
face[i*3]*3;
      int
p2 =
face[i*3+1]*3
      int
p3 =
face[i*3+2]*3
      float
ux =
vertex[p3]-ve
      float
uy =
vertex[p3+1]-
      float
uz =
vertex[p3+2]-
      float
vx =
vertex[p2]-ve
      float
vy =
vertex[p2+1]-
      float
vz =
vertex[p2+2]-
      float
nx = uy*vz
- uz*vy;
      float
ny = uz*vx
- ux*vz;
      float
nz = ux*vy
- uy*vx;
      float
length =
sqrt(nx*nx+ny

normal[p1]
+=
nx/length;

normal[p1+1]
```



```
+= ny/length;
      normal[p1+2] += nz/length;
      normal[p2] += nx/length;
      normal[p2+1] += ny/length;
      normal[p2+2] += nz/length;
      normal[p3] += nx/length;
      normal[p3+1] += ny/length;
      normal[p3+2] += nz/length;
    }
}
```

called in `main()` right after



`read_wavefront("motorBike.obj");`. Now mistery occurs with garbled part of motorbike. Where the problem is? Possible suggestions: pointer problems, memory leackage, normals calculation, OpenGL bug, shader, data, ... Use debugger, Valgrind and VisIt (normals) to locate the problem. Hint: Observe number of triangles versus number of vertices. Quick fix is given as final motorbike.c source.

Homework

1. Colorize the model by groups
2. Add opacity (alpha)
3. Find (or create) nice shader source
4. Add scientific data such as pressure at the surface
5. Draw streamlines with velocity colorization
6. Convert example into GPU buffered (VBOs).
7. Combine sliced pressure data(VTK) and model (OBJ) together as custom CFD visualization not available in visualization tools to date.

Attachments (24)

Last modified on 07/04/13 20:02:44