Grafică pe calculator (MLR5060)

Elemente de grafică 3_D

Programare *OpenGl*

- Introduction
- Rendering Primitives
- Rendering Modes
- Lighting
- Texture Mapping
- Additional Rendering Attributes
- Imaging

15 Dec. 2018
Goals

- Demonstrate enough *OpenGL* to write an interactive graphics program with
  - custom modeled 3D objects or imagery
  - lighting
  - texture mapping
- Introduce advanced topics for future investigation
OpenGL and GLUT

- **OpenGL**
  - Graphics rendering API
    - high-quality color images composed of geometric and image primitives
    - window system independent
    - operating system independent

Introduce advanced topics for future investigation

*OpenGL* is a library for doing computer graphics. By using it, you can create interactive applications which render high-quality color images composed of 3D geometric objects and images. *OpenGL* is window and operating system independent. As such, the part of your application which does rendering is platform independent. However, in order for *OpenGL* to be able to render, it needs a window to draw into. Generally, this is controlled by the windowing system on whatever platform you’re working on.
This is the most important diagram you will see today, representing the flow of graphical information, as it is processed from CPU to the frame buffer. There are two pipelines of data flow. The upper pipeline is for geometric, vertex-based primitives. The lower pipeline is for pixel-based, image primitives. Texturing combines the two types of primitives together. There is a pull-out poster in the back of the OpenGL Reference Manual ("Blue Book"), which shows this diagram in more detail.
OpenGL as a Renderer

- Geometric primitives
  - points, lines and polygons

- Image Primitives
  - images and bitmaps
  - separate pipeline for images and geometry
    - linked through texture mapping

- Rendering depends on state
  - colors, materials, light sources, etc.
As mentioned, OpenGL is a library for rendering computer graphics. Generally, there are two operations that you do with OpenGL:

- draw something
- change the state of how OpenGL draws

OpenGL has two types of things that it can render: geometric primitives and image primitives. Geometric primitives are points, lines and polygons. Image primitives are bitmaps and graphics images (i.e. the pixels that you might extract from a JPEG image after you’ve read it into your program.) Additionally, OpenGL links image and geometric primitives together using texture mapping, which is an advanced topic we’ll discuss this afternoon.

The other common operation that you do with OpenGL is setting state. “Setting state” is the process of initializing the internal data that OpenGL uses to render your primitives. It can be as simple as setting up the size of points and color that you want a vertex to be, to initializing multiple mipmap levels for texture mapping.
Related APIs

- AGL, GLX, WGL
  - glue between OpenGL and windowing systems

- GLU (OpenGL Utility Library)
  - part of OpenGL
  - NURBS, tessellators, quadric shapes, etc.

- GLUT (OpenGL Utility Toolkit)
  - portable windowing API
  - not officially part of OpenGL
As mentioned, OpenGL is window and operating system independent. To integrate it into various window systems, additional libraries are used to modify a native window into an OpenGL capable window. Every window system has its own unique library and functions to do this. Some examples are:

- GLX for the X Windows system, common on Unix platforms
- AGL for the Apple Macintosh
- WGL for Microsoft Windows

OpenGL also includes a utility library, GLU, to simplify common tasks such as: rendering quadric surfaces (i.e. spheres, cones, cylinders, etc.), working with NURBS and curves, and concave polygon tessellation.

Finally to simplify programming and window system dependence, we’ll be using the freeware library, GLUT. GLUT, written by Mark Kilgarrd, is a public domain window system independent toolkit for making simple OpenGL applications. It simplifies the process of creating windows, working with events in the window system and handling animation.
OpenGL and Related APIs

application program

OpenGL Motif widget or similar

GLX, AGL or WGL

X, Win32, Mac O/S

GLUT

GLU

GL

software and/or hardware
The above diagram illustrates the relationships of the various libraries and window system components.

Generally, applications which require more user interface support will use a library designed to support those types of features (i.e. buttons, menu and scroll bars, etc.) such as Motif or the Win32 API.

Prototype applications, or one which don’t require all the bells and whistles of a full GUI, may choose to use GLUT instead because of its simplified programming model and window system independence.
Preliminaries

- **Headers Files**
  - `#include <GL/gl.h>`
  - `#include <GL/glu.h>`
  - `#include <GL/glut.h>`

- **Libraries**

- **Enumerated Types**
  - OpenGL defines numerous types for compatibility
    - `GLfloat`, `GLint`, `GLenum`, etc.
All of our discussions today will be presented in the C computer language.

For C, there are a few required elements which an application must do:

- **Header files** describe all of the function calls, their parameters and defined constant values to the compiler. OpenGL has header files for GL (the core library), GLU (the utility library), and GLUT (freeware windowing toolkit).

- **Note:** glut.h includes gl.h and glu.h. On Microsoft Windows, including only glut.h is recommended to avoid warnings about redefining Windows macros.

- **Libraries** are the operating system dependent implementation of OpenGL on the system you’re using. Each operating system has its own set of libraries. For Unix systems, the OpenGL library is commonly named libGL.so and for Microsoft Windows, it’s named opengl32.lib.

- Finally, **enumerated types** are definitions for the basic types (i.e. float, double, int, etc.) which your program uses to store variables. To simplify platform independence for OpenGL programs, a complete set of enumerated types are defined. Use them to simplify transferring your programs to other operating systems.
GLUT Basics

- Application Structure
  - Configure and open window
  - Initialize OpenGL state
  - Register input callback functions
    - render
    - resize
    - input: keyboard, mouse, etc.
  - Enter event processing loop
Here’s the basic structure that we’ll be using in our applications. This is generally what you’d do in your own OpenGL applications.

The steps are:

1) Choose the type of window that you need for your application and initialize it.

2) Initialize any OpenGL state that you don’t need to change every frame of your program. This might include things like the background color, light positions and texture maps.

3) Register the *callback* functions that you’ll need. Callbacks are routines you write that GLUT calls when a certain sequence of events occurs, like the window needing to be refreshed, or the user moving the mouse. The most important callback function is the one to render your scene, which we’ll discuss in a few slides.

4) Enter the main event processing loop. This is where your application receives events, and schedules when callback functions are called.
Sample Program

Here’s an example of the main part of a GLUT based OpenGL application. This is the model that we’ll use for most of our programs in the course.

The `glutInitDisplayMode()` and `glutCreateWindow()` functions compose the window configuration step.

```c
void main( int argc, char** argv )
{
    int mode = GLUT_RGB | GLUT_DOUBLE;
    glutInitDisplayMode( mode );
    glutCreateWindow( argv[0] );
    init();
    glutDisplayFunc( display );
    glutReshapeFunc( resize );
    glutKeyboardFunc( key );
    glutIdleFunc( idle );
    glutMainLoop();
}
```

We then call the `init()` routine, which contains our one-time initialization. Here we initialize any OpenGL state and other program variables that we might need to use during our program that remain constant throughout the program’s execution.

Next, we register the callback routines that we’re going to use during our program.

Finally, we enter the event processing loop, which interprets events and calls our respective callback routines.
Set up whatever state you’re going to use

```c
void init( void )
{
    glClearColor( 0.0, 0.0, 0.0, 1.0 );
    glClearDepth( 1.0 );

    glEnable( GL_LIGHT0 );
    glEnable( GL_LIGHTING );
    glEnable( GL_DEPTH_TEST );
}
```
GLUT Callback Functions

- Routine to call when something happens
  - window resize or redraw
  - user input
  - animation

- “Register” callbacks with GLUT
  
  ```c
  glutDisplayFunc( display );
  glutIdleFunc( idle );
  glutKeyboardFunc( keyboard );
  ```
Do all of your drawing here

```c
void display( void )
{
  glClear( GL_COLOR_BUFFER_BIT );
  glBegin( GL_TRIANGLE_STRIP );
  glVertex3fv( v[0] );
  glVertex3fv( v[1] );
  glVertex3fv( v[2] );
  glVertex3fv( v[3] );
  glEnd();
  glutSwapBuffers();
}
```
Idle Callbacks

- Use for animation and continuous update

```c
glutIdleFunc( idle );

void idle( void )
{
    t += dt;
    glutPostRedisplay();
}
```
User Input Callbacks

- Process user input

```c
glutKeyboardFunc(keyboard);

void keyboard(char key, int x, int y)
{
    switch(key) {
        case 'q': case 'Q':
            exit(EXIT_SUCCESS);
            break;
        case 'r': case 'R':
            rotate = GL_TRUE;
            break;
    }
}
```
Elementary Rendering

- Geometric Primitives
- Managing OpenGL State
- OpenGL Buffers
OpenGL Geometric Primitives

- All geometric primitives are specified by vertices

- GL_LINES
- GL_POINTS
- GL_LINE_STRIP
- GL_LINE_LOOP
- GL_TRIANGLES
- GL_QUADS
- GL_TRIANGLE_STRIP
- GL_TRIANGLE_FAN
- GL_POLYGON
- GL_QUAD_STRIP
void drawRhombus( GLfloat color[] )
{
    glBegin( GL_QUADS );
    glColor3fv( color );
    glVertex2f( 0.0, 0.0 );
    glVertex2f( 1.0, 0.0 );
    glVertex2f( 1.5, 1.118 );
    glVertex2f( 0.5, 1.118 );
    glEnd();
}
OpenGL Command Formats

glVertex3fv( \( v \) )

**Number of components**
- 2 - (x,y)
- 3 - (x,y,z)
- 4 - (x,y,z,w)

**Data Type**
- b - byte
- ub - unsigned byte
- s - short
- us - unsigned short
- i - int
- ui - unsigned int
- f - float
- d - double

**Vector**
- omit “v” for scalar form
- glVertex2f( x, y )
Primitives are specified using

```c
GLuint red, greed, blue;
GLfloat coords[3];

GLuint primType;

GLuint glBegin(primType);
for (i = 0; i < nVerts; ++i) {
    glColor3f(red, green, blue);
    glVertex3fv(coords);
}

glEnd();
```
OpenGL Color Models

- RGBA or Color Index

**color index mode**

<table>
<thead>
<tr>
<th>Red</th>
<th>Green</th>
<th>Blue</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**RGBA mode**

Display
Shapes Tutorial

```
glBegin (GL_TRIANGLE_STRIP);
glColor3f (1.00 , 0.00 , 1.00 );
glVertex2f (0.0 , 25.0 );
glColor3f (0.00 , 1.00 , 1.00 );
glVertex2f (50.0 , 150.0 );
glColor3f (0.00 , 1.00 , 0.00 );
glVertex2f (125.0 , 100.0 );
glColor3f (1.00 , 1.00 , 0.00 );
glVertex2f (175.0 , 200.0 );
glEnd();
```
Controlling Rendering Appearance

- From Wireframe to Texture Mapped
OpenGL’s State Machine

- All rendering attributes are encapsulated in the OpenGL State
  - rendering styles
  - shading
  - lighting
  - texture mapping
Manipulating OpenGL State

- Appearance is controlled by current state
  for each (primitive to render) {
    update OpenGL state
    render primitive
  }

- Manipulating vertex attributes is most common way to manipulate state
  
  \texttt{glColor*()} / \texttt{glIndex*()}
  \texttt{glNormal*()}
  \texttt{glTexCoord*()}


Controlling current state

- **Setting State**
  
  ```
  glPointSize(size);
  glLineStipple(repeat, pattern);
  glShadeModel(GL_SMOOTH);
  ```

- **Enabling Features**
  
  ```
  glEnable(GL_LIGHTING);
  glDisable(GL_TEXTURE_2D);
  ```
Transformations in OpenGL

- Modeling
- Viewing
  - orient camera
  - projection
- Animation
- Map to screen
Camera Analogy

- 3D is just like taking a photograph (lots of photographs!)
Camera Analogy and Transformations

- Projection transformations
  - adjust the lens of the camera
- Viewing transformations
  - tripod–define position and orientation of the viewing volume in the world
- Modeling transformations
  - moving the model
- Viewport transformations
  - enlarge or reduce the physical photograph
Coordinate Systems and Transformations

- Steps in Forming an Image
  - specify geometry (world coordinates)
  - specify camera (camera coordinates)
  - project (window coordinates)
  - map to viewport (screen coordinates)
- Each step uses transformations
- Every transformation is equivalent to a change in coordinate systems (frames)
Affine Transformations

- Want transformations which preserve geometry
  - lines, polygons, quadrics

- Affine = line preserving
  - Rotation, translation, scaling
  - Projection
  - Concatenation (composition)
Homogeneous Coordinates

- each vertex is a column vector

\[ \vec{v} = \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \]

- \( w \) is usually 1.0
- all operations are matrix multiplications
- directions (directed line segments) can be represented with \( w = 0.0 \)
3D Transformations

- A vertex is transformed by 4 x 4 matrices
  - all affine operations are matrix multiplications
  - all matrices are stored column-major in OpenGL
  - matrices are always post-multiplied
  - product of matrix and vector is \( \mathbf{M} \mathbf{v} \)

\[
\mathbf{M} = \begin{bmatrix}
m_0 & m_4 & m_8 & m_{12} \\
m_1 & m_5 & m_9 & m_{13} \\
m_2 & m_6 & m_{10} & m_{14} \\
m_3 & m_7 & m_{11} & m_{15}
\end{bmatrix}
\]
Specifying Transformations

- Programmer has two styles of specifying transformations
  - specify matrices (`glLoadMatrix`, `glMultMatrix`)
  - specify operation (`glRotate`, `glOrtho`)
- Programmer does not have to remember the exact matrices
  - check appendix of Red Book (Programming Guide)
Prior to rendering, view, locate, and orient:
- eye/camera position
- 3D geometry

Manage the matrices
- including matrix stack

Combine (composite) transformations
Transformation Pipeline

object → eye → clip → normalized device → window

- other calculations here
  - material & color
  - shade model (flat)
  - polygon rendering mode
  - polygon culling
  - clipping
Matrix Operations

- Specify Current Matrix Stack
  \texttt{glMatrixMode( GL\_MODELVIEW \textit{or} GL\_PROJECTION )}

- Other Matrix or Stack Operations
  \texttt{glLoadIdentity() \hspace{1em} glPushMatrix()}
  \texttt{glPopMatrix()}

- Viewport
  - usually same as window size
  - viewport aspect ratio should be same as projection transformation or resulting image may be distorted
  \texttt{glViewport( x, y, width, height )}
Projection Transformation

- Shape of viewing frustum
- Perspective projection
  \[ \text{gluPerspective}( \text{fovy}, \text{aspect}, \text{zNear}, \text{zFar}) \]
  \[ \text{glFrustum}( \text{left}, \text{right}, \text{bottom}, \text{top}, \text{zNear}, \text{zFar}) \]
- Orthographic parallel projection
  \[ \text{glOrtho}( \text{left}, \text{right}, \text{bottom}, \text{top}, \text{zNear}, \text{zFar}) \]
  \[ \text{gluOrtho2D}( \text{left}, \text{right}, \text{bottom}, \text{top}) \]
  \[ \text{calls glOrtho with z values near zero} \]
Applying Projection Transformations

- Typical use (orthographic projection)
  ```c
  glMatrixMode( GL_PROJECTION );
  glLoadIdentity();
  glOrtho( left, right, bottom, top, zNear, zFar );
  ```
Viewing Transformations

- Position the camera/eye in the scene
  - place the tripod down; aim camera
- To “fly through” a scene
  - change viewing transformation and redraw scene
- `gluLookAt( eye_x, eye_y, eye_z, aim_x, aim_y, aim_z, up_x, up_y, up_z )`
  - up vector determines unique orientation
  - careful of degenerate positions
Projection Tutorial

Command manipulation window:

```plaintext
fovy aspect zNear zFar

```gluPerspective( 60.0 , 1.00 , 1.0 , 10.0 );
```
```
gluLookAt( 0.00 , 0.00 , 2.00 , <- eye
```
```
0.00 , 0.00 , 0.00 , <- center
```
```
0.00 , 1.00 , 0.00 ); <- up
```
```
Click on the arguments and move the mouse to modify values.
```
Modeling Transformations

- Move object
  \[ \text{glTranslate}\{fd\}(x, y, z) \]
- Rotate object around arbitrary axis
  \[ \text{glRotate}\{fd\}(\text{angle}, x, y, z) \]
  - angle is in degrees
- Dilate (stretch or shrink) or mirror object
  \[ \text{glScale}\{fd\}(x, y, z) \]
Transformation Tutorial

World-space view

Screen-space view

Command manipulation window

```c
glTranslatef( 0.00, 0.00, 0.00 );

glRotatef( -52.0, 0.00, 1.00, 0.00 );

glScalef( 1.00, 1.00, 1.00 );

glBegin( ... );

... 

Click on the arguments and move the mouse to modify values.
```
Connection: Viewing and Modeling

- Moving camera is equivalent to moving every object in the world towards a stationary camera.
- Viewing transformations are equivalent to several modeling transformations.

`gluLookAt()` has its own command can make your own polar view or pilot view.
Projection is left handed

- Projection transformations (*gluPerspective*, *glOrtho*) are left handed
  - think of \( z_{\text{Near}} \) and \( z_{\text{Far}} \) as distance from view point
- Everything else is right handed, including the vertexes to be rendered
3 examples of `resize()` routine

- restate projection & viewing transformations

Usually called when window resized

Registered as callback for `glutReshapeFunc()`
void resize( int w, int h )
{
    glViewport( 0, 0, (GLsizei) w, (GLsizei) h );
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    gluPerspective( 65.0, (GLfloat) w / h, 1.0, 100.0 );
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    gluLookAt( 0.0, 0.0, 5.0, 
               0.0, 0.0, 0.0, 
               0.0, 1.0, 0.0 );
}
Same effect as previous LookAt

```c
void resize( int w, int h )
{
    glViewport( 0, 0, (GLsizei) w, (GLsizei) h );
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    gluPerspective( 65.0, (GLfloat) w/h, 1.0, 100.0 );
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    glTranslatef( 0.0, 0.0, -5.0 );
}
```
void resize( int width, int height )
{
    GLdouble aspect = (GLdouble) width / height;
    GLdouble left = -2.5, right = 2.5;
    GLdouble bottom = -2.5, top = 2.5;
    glViewport( 0, 0, (GLsizei) w, (GLsizei) h );
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    ... continued ...
if ( aspect < 1.0 ) {
    left /= aspect;
    right /= aspect;
} else {
    bottom *= aspect;
    top *= aspect;
}
glOrtho( left, right, bottom, top, near, far );
glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
}
Compositing Modeling Transformations

- **Problem 1**: hierarchical objects
  - one position depends upon a previous position
  - robot arm or hand; sub-assemblies

- **Solution 1**: moving local coordinate system
  - modeling transformations move coordinate system
  - post-multiply column-major matrices
  - OpenGL post-multipplies matrices
Problem 2: objects move relative to absolute world origin
- my object rotates around the wrong origin
  - make it spin around its center or something else

Solution 2: fixed coordinate system
- modeling transformations move objects around fixed coordinate system
- pre-multiply column-major matrices
- OpenGL post-multiplies matrices
- must reverse order of operations to achieve desired effect
At least 6 more clipping planes available

- Good for cross-sections
- Modelview matrix moves clipping plane

- $Ax + By + Cz + D < 0$ clipped

- `glEnable( GL_CLIP_PLANEi )`
- `glClipPlane( GL_CLIP_PLANEi, GLdouble* coeff )`
Screen space back to world space

```c
glGetDoublev(GL_MODELVIEW_MATRIX, GLdouble mvmatrix[16] )
glGetDoublev(GL_PROJECTION_MATRIX,
    GLdouble projmatrix[16] )
gluUnProject(GLdouble winx, winy, winz,
           mvmatrix[16], projmatrix[16],
           GLint viewport[4],
           GLdouble *objx, *objy, *objz )
```

- **gluProject** goes from world to screen space
On-Line Resources

- http://www.opengl.org
  - start here; up to date specification and lots of sample code
- news:comp.graphics.api.opengl
- http://www.sgi.com/software/opengl
- http://www.mesa3d.org/
  - Brian Paul’s Mesa 3D
  - very special thanks to Nate Robins for the OpenGL Tutors
  - source code for tutors available here!
Books

- OpenGL Programming for the X Window System
  - includes many GLUT examples
Referințe


2. *Intro to 3D Graphics using Tao.OpenGL*, Erika Troll, Josh Lavinder, Alton Ng, Andrew Padilla
   - [http://www.math.ucla.edu/~wittman/10c.1.11s/Lectures/Raids/Graphics3D.pdf](http://www.math.ucla.edu/~wittman/10c.1.11s/Lectures/Raids/Graphics3D.pdf)


5. *An Interactive Introduction to OpenGL Programming*, Dave Shreiner, Ed Angel, Vicki Shreiner
   - [http://www.vis.uky.edu/~ryang/teaching/cs535-2012spr/Lectures/OpenGL.pptx, Lectures](http://www.vis.uky.edu/~ryang/teaching/cs535-2012spr/Lectures/OpenGL.pptx, Lectures)