An Introduction to OpenGL Programming
Introducing OpenGL

Industry standard API for computer graphics
OpenGL (Open Graphics Library)

- OpenGL is a **cross-language, multi-platform** application programming interface (API) for rendering 2D and 3D computer graphics.

- Applications make calls to OpenGL, which then renders an image (by handling the graphics hardware) and displays it.

- The API contains about 150 commands.

- It is purely concerned with rendering, providing no APIs related to input, audio, or windowing.
Not the Only One

For Interactive Applications; not specific to hardware or OS
What is OpenGL?

• The standard specification defining an API that interfaces with the computer’s graphics system
  ✓ Cross-language
  ✓ Cross-platform
  ✓ Vendor-independent

• Introduced in 1992 by Silicon Graphics Inc
• OpenGL is purely concerned with rendering, providing no APIs related to input, audio, or windowing.
Development of OpenGL

- OpenGL is an evolving API. New versions of the OpenGL specification are regularly released by the Khronos Group, each of which extends the API to support various new features.

- **OpenGL 4.5** Release Date: August, 2014
Related APIs in the OpenGL Family
What OpenGL Does

- Allows definition of object shapes, material properties and lighting

- Arranges objects and interprets synthetic camera in 3D space

- Converts mathematical representations of objects into pixels (rasterization)

- Calculates the color of every object
OpenGL

- NO high-level rendering functions for complex objects
- build your shapes from primitives, points, lines, polygons, etc.
- The utility library GLU provides additional support
Simplified OpenGL Pipeline

- Display List
- Evaluator
- Per-Vertex Operations
  - Primitive Assembly
- Rasterization
- Per-Fragment Operation
- Frame Buffer
- Texture Memory
- Pixel Operations
Pieces of OpenGL Pipeline

Stores “subroutines”
Pieces of OpenGL Pipeline

Stores “subroutines”

Faster!
- Pre-compiled
- Store on GPU
- Pre-compute transformations
Pieces of OpenGL Pipeline

Construct geometric objects
Pieces of OpenGL Pipeline

Change meshed geometry

Per-Vertex Operations
Primitive Assembly

Store primitive shapes

Includes clipping!
Primitive \textit{[prim-i-tiv]}: A small piece of geometry that can be rendered; in OpenGL, triangles, quads, lines, points (usually).
Pieces of OpenGL Pipeline

Rasterization

Diagram showing the OpenGL pipeline with focus on the rasterization stage.
Pieces of OpenGL Pipeline

Modify and combine per-pixel information
Pieces of OpenGL Pipeline

Prepare image to be displayed

Frame Buffer
Related API

• **opengl32.lib (OpenGL Kernel Library)**
  ✅ Part of OpenGL
  ✅ Use the prefix of `gl` (ex: `glBegin()`)

• **GLU (OpenGL Utility Library)**
  ✅ Part of OpenGL
  ✅ Use the prefix of `glu` (ex: `gluLookAt()`)
Related API

- **GLUT (OpenGL Utility Toolkit)**
  - Not officially part of OpenGL
  - Provide common features for window system
  - Create window, mouse and keyboard, menu, event-driven
  - Lack of modern GUI support (e.g. scroller)
  - Use the prefix of `glut` (ex: `glutDisplayFunc()`)
Installing GLUT

- **On Windows:**
  - Download from Nate Robins’ website:
    - [http://www.xmission.com/~enate/glut.html](http://www.xmission.com/~enate/glut.html)
    - glut-3.7.6-bin has the dll/lib/header that are required
      - Copy glut.dll to `{windows/system32}\glut32.dll`
      - Copy glut.lib to `{VC++ lib path}\glut32.lib`
      - Copy glut.h to `{VC++ include path}\GL\glut.h`
Using GLUT

- Only need to include glut.h
  - `#include <GL/glut.h>`
    - Automatically includes gl.h and glu.h

- Lighthouse3D has a good GLUT tutorial
3 Stages in OpenGL

1. Define Objects in World Scene
2. Set Modeling and Viewing Transformations
3. Render the Scene
How OpenGL Works

- **OpenGL is a state machine**
  - You give it orders to set the current state of any one of its internal variables, or to query for its current status
  - The current state won’t change until you specify otherwise
  - Each of the system’s state variables has a default value
Functions of OpenGL

- Primitive - WHAT - Point, Edge, Polygon
- Attribute - HOW
- Transformation - Viewing & Modelling
- Input - provided by GLUT
- Control - provided by GLUT
- Query
Function Format of OpenGL

```
glVertex3f(x, y, z)
```

- `glVertex3f` is a function in the OpenGL library.
- It takes three parameters: `x`, `y`, and `z`.
- `x`, `y`, and `z` are of type `float`.

```
glVertex3fv(p)
```

- `glVertex3fv` is similar to `glVertex3f`, but it takes a pointer `p` as an additional parameter.
- `p` is a pointer to a `float` array.

**Parameters:**
- `b` - byte
- `ub` - unsigned byte
- `s` - short
- `us` - unsigned short
- `i` - int
- `ui` - unsigned int
- `f` - float
- `d` - double

**Note:**
- Pay attention to the case sensitivity in the parameters.
- `p` must be a pointer to a `float` array.
OpenGL Hello World

- Two Versions
  - GLUT Version (no shader) ← on class
  - GLEW Version (with shader) ← after class

- 2D demo
Prerequisite

• Head Files:

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

• Library Files:

- 编译系统库文件目录\opengl32.lib glu32.lib glut32.lib
- C:\windows\system32\opengl32.dll glu32.dll glut32.dll
Basic Structure Of OpenGL Program

- NOT Object-Oriented!!
- Use states to control
- Infinite Loop
Event Driven Programming

Display Handler

Keyboard Handler

Mouse Handler

Main Event Loop
2D demo

#include<gl/glut.h>

void renderScene(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glBegin(GL_TRIANGLES);
    glColor3f(0.0f, 1.0f, 0.0f);
    glVertex3f(-0.5,-0.5,0.0);
    glVertex3f(0.5,0.0,0.0);
    glVertex3f(0.0,0.5,0.0);
    glEnd();
    glFlush();
}

int main(int argc, char *argv[])
{
    glutInit(&argc, argv);
    glutCreateWindow("Hello OpenGL");
    glutDisplayFunc(renderScene);
    glutMainLoop();
    return 0;
}

Less than 20 lines!
Not that HARD
2D demo

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

void renderScene(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glBegin(GL_TRIANGLES);
    glColor3f(0.0f, 1.0f, 0.0f);
    glVertex3f(-0.5, -0.5, 0.0);
    glVertex3f(0.5, 0.0, 0.0);
    glVertex3f(0.0, 0.5, 0.0);
    glEnd();
    glFlush();
}

int main(int argc, char *argv[])
{
    glutInit(&argc, argv);
    glutCreateWindow("Hello OpenGL");
    glutDisplayFunc(renderScene);
    glutMainLoop();
    return 0;
}
```
2D demo

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

void renderScene(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glBegin(GL_TRIANGLES);
    glColor3f(0.0f, 1.0f, 0.0f);
    glVertex3f(-0.5, -0.5, 0.0);
    glVertex3f(0.5, 0.0, 0.0);
    glVertex3f(0.0, 0.5, 0.0);
    glEnd();
    glFlush();
}

int main(int argc, char *argv[])
{    glutInit(&argc, argv);
    glutCreateWindow("Hello OpenGL");
    glutDisplayFunc(renderScene);
    glutMainLoop();
    return 0;
}
```

clear the buffer

let’s draw a triangle

using RGB color green

this is the 3 points of the triangle

end of drawing

Do it!
Callbacks

• Wiki: In computer programming, a callback is a reference to a piece of executable code, that is passed as an argument to other code. This allows a lower-level software layer to call a subroutine (or function) defined in a higher-level layer.

• Usage
  – Callbacks allow the user of a function to fine-tune it at runtime, another use is in error signaling.
  – Callbacks may also be used to control whether a function acts or not.

• In C/C++: function pointer
Callbacks

• Typically, the main thread will just run in a loop, waiting for events to occur - for example, for the user to move his mouse in your window, or click one of your buttons. The GUI framework will provide a mechanism for you to pass it function pointers, which it will then associate with certain events. When an event occurs, the event loop will invoke any callback functions you've provided for that event. Often, the callback function will have parameters, and the event dispatcher will provide you with extra information about the event (perhaps the exact x,y coordinates of the mouse, for example) through the arguments it calls your callback function with.
Display Callback

Called when window is redrawn

```c
void redraw()
{
    glClear(GL_COLOR_BUFFER_BIT);

    glBegin(GL_QUADS);
    glColor3f(1, 0, 0);
    glVertex3f(-0.5, 0.5, 0.5);
    glVertex3f( 0.5, 0.5, 0.5);
    glVertex3f( 0.5, -0.5, 0.5);
    glVertex3f(-0.5, -0.5, 0.5);
    glEnd(); // GL_QUADS

    glutSwapBuffers();
}
```
Called when the window is resized

```c
void reshape(int w, int h)
{
    glViewport(0.0, 0.0, w, h);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0, w, 0.0, h, -1.0, 1.0);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
```
Mouse Callback

Called when the mouse button is pressed

```c
void mousebutton(int button, int state, int x, int y)
{
    if (button==GLUT_LEFT_BUTTON && state==GLUT_DOWN)
    {
        rx = x; ry = winHeight - y;
    }
}
```

Called when the mouse is moved with button down

```c
void motion(int x, int y)
{
    rx = x; ry = winHeight - y;
}
```
Closing the program

- In previous programs, it was not possible to end the program through OpenGL. Now, we can use a simple callback function for the mouse button.

```c
void mouse(GLint btn, GLint state, GLint x, GLint y)
{
    if (btn == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
        exit(0);
}
```
Keyboard Callback

Called when a button is pressed

```cpp
void keyboardCB(unsigned char key, int x, int y)
{
    switch(key)
    {
        case 'a': cout << "a Pressed" << endl; break;
    }
}
```

Called when a special button is pressed

```cpp
void special(int key, int x, int y)
{
    switch(key)
    {
        case GLUT_F1_KEY:
            cout << "F1 Pressed" << endl; break;
    }
}
```
在屏幕上的位置通常是以像素为单位的，原点在左上角
- 因为显示器自顶向下刷新显示内容
- 在OpenGL中应用一个世界坐标系，其原点在左下角
- 在这个坐标系中的y坐标需要从窗口高度中减去回调函数返回的y值：
  - \( y := h - y \)
Get the height of window

为了完成y坐标的转换，需要知道窗口的尺寸

- 在程序执行过程中高度可能发生改变
- 需要利用一个全局变量跟踪其变化
- 新高度值返回给形状改变回调函数（见后）
- 也可以用查询函数`glGetIntv`和`glGetFloatv`获取，因为高度是状态的一部分
OpenGL - GLUT Example

```
#include <gl/glut.h>
#include <stdlib.h>
static GLfloat spin = 0.0;
void init( void )
{
    glClearColor( 0.0, 0.0, 0.0, 0.0 );
    glShadeModel( GL_FLAT );
    void display( void )
    {
        glClear( GL_COLOR_BUFFER_BIT );
        glPushMatrix();
        glRotatef( spin, 0.0, 0.0, 1.0 );
        glColor3f( 1.0, 1.0, 1.0 );
        glRectf( -25.0, -25.0, 25.0, 25.0 );
        glPopMatrix();
        glutSwapBuffers();
    }
```
void spinDisplay( void )
{
    spin += 2.0;
    if( spin > 360.0 )
        spin -= 360.0;
    glutPostRedisplay();
}

void reshape( int w, int h )
{
    glViewport( 0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    glOrtho( -50.0, 50.0, -50.0, 50.0, -1.0, 1.0 );
    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
}
void mouse( int button, int state, int x, int y )
{
    switch( button )
    {
    case GLUT_LEFT_BUTTON:
        if( state == GLUT_DOWN )
            glutIdleFunc( spinDisplay );
        break;
    case GLUT_RIGHT_BUTTON:
        if( state == GLUT_DOWN )
            glutIdleFunc( NULL );
        break;
    default: break;
    }
}
int main( int argc, char ** argv )
{
    glutInit( &argc, argv );
    glutInitDisplayMode( GLUT_DOUBLE | GLUT_RGB );
    glutInitWindowSize( 250, 250 );
    glutInitWindowPosition( 100, 100 );
    glutCreateWindow( argv[ 0 ] );

    init();
    glutDisplayFunc( display );
    glutReshapeFunc( reshape );
    glutMouseFunc( mouse );
    glutMainLoop();
    return 0;
}
Details of OpenGL Program
Contexts and Viewports?

- Each OpenGL application creates a **context** to issue rendering commands to.
- The application must also define a **viewport**, a region of pixels on the screen that can see the context.
  - Can be:
    - Part of a window
    - An entire window
    - The whole screen
Viewport

• The viewport is the part of the window your drawing is displayed to
  – By default, the viewport is the entire window

• Modifying the viewport is analogous to changing the size of the final picture
  – From the camera analogy

• Can have multiple viewports in the same window for a split-screen effect
Setting the Viewport

- `glViewport(int x, int y, int width, int height)`
  - `(x, y)` is the location of the origin (lower-left) within the window
  - `(width, height)` is the size of the viewport

- **The aspect ratio of the viewport should be the same as that of the viewing volume**
OpenGL as a State Machine

• Put a value into various states, then it will remain in effect until being changed.
  – e.g. glColor*()

• Many state variables are enabled or disabled with glEnable(), glDisable()
  – e.g. glEnable(GL_LIGHT0)
OpenGL State

• Some attributes of the OpenGL state
  – Current color
  – Camera properties (location, orientation, field of view, etc.)
  – Lighting model (flat, smooth, etc.)
  – Type of primitive being drawn
  – Line width, dotted line or full line,…
  – And many more…
OpenGL Input

- All inputs (i.e. geometry) to an OpenGL context are defined as vertex lists

- `glVertex*`
  - `* = nt OR ntv`
    - `n` - number (2, 3, 4)
    - `t` - type (i = integer, f = float, etc.)
    - `v` - vector
## OpenGL Types

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Data Type</th>
<th>Typical Corresponding C-Language Type</th>
<th>OpenGL Type Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>8-bit integer</td>
<td>signed char</td>
<td>GLbyte</td>
</tr>
<tr>
<td>s</td>
<td>16-bit integer</td>
<td>short</td>
<td>GLshort</td>
</tr>
<tr>
<td>i</td>
<td>32-bit integer</td>
<td>long</td>
<td>GLint, GLsizei</td>
</tr>
<tr>
<td>f</td>
<td>32-bit floating-point</td>
<td>float</td>
<td>GLfloat, GLclampf</td>
</tr>
<tr>
<td>d</td>
<td>64-bit floating-point</td>
<td>double</td>
<td>GLdouble, GLclampd</td>
</tr>
<tr>
<td>ub</td>
<td>8-bit unsigned integer</td>
<td>unsigned char</td>
<td>GLubyte, GLboolean</td>
</tr>
<tr>
<td>us</td>
<td>16-bit unsigned integer</td>
<td>unsigned short</td>
<td>GLushort</td>
</tr>
<tr>
<td>ui</td>
<td>32-bit unsigned integer</td>
<td>unsigned long</td>
<td>GLuint, GLenum, GLbitfield</td>
</tr>
</tbody>
</table>
OpenGL Input

- Examples:
  - `glVertex3f(.25, .25, .5);`
  - `double vertex[3] = {1.0, .33, 3.14159}; glVertex3dv(vertex);`
  - “v” tells the system to expect the coordinate list in a single data structure, instead of a list of n numbers
OpenGL Primitive Types

• All geometry is specified by vertex lists
  – But can draw multiple types of things
    • Points
    • Lines
    • Triangles
    • etc.

• The different things the system knows how to draw are the system **primitives**
OpenGL Primitive Types

- GL_POINTS
- GL_POLYGON
- GL_QUAD_STRIP
- GL_TRIANGLES
- GL_LINES
- GL_LINE_STRIP
- GL_LINE_LOOP

Note: Must be planner convex.
Specifying the OpenGL Primitive Type

• `glBegin(primitiveType);`
  // A list of glVertex* calls goes here
  // ...
  glEnd();

• `primitiveType` can be any of several things
  – See the next slide
OpenGL Primitives Example

```c
glBegin(GL_POLYGON);
    glVertex2f(0.0, 0.0);
    glVertex2f(0.0, 3.0);
    glVertex2f(3.0, 3.0);
    glVertex2f(4.0, 1.5);
    glVertex2f(3.0, 0.0);
 glEnd();
```
Color in OpenGL

- OpenGL colors are typically defined as RGB components each of which is a float in the range [0.0, 1.0]

- For the screen’s background:
  - `glClearColor( 0.0, 0.0, 0.0 );` // black color
  - `glClear( GL_COLOR_BUFFER_BIT );`

- For objects:
  - `glColor3f( 1.0, 1.0, 1.0 );` // white color
Color in OpenGL

- GLUT_RGB and GLUT_RGBA
- alpha channel

- glColor3f (1.0, 1.0, 1.0);
- glColor3i (0, 255, 255);
- glColor3fv (colorArray);
Polygon Display Modes

- `glPolygonMode( GLenum face, GLenum mode );`
  - Faces: GL_FRONT, GL_BACK, GL_FRONT_AND_BACK
  - Modes: GL_FILL, GL_LINE, GL_POINT
  - By default, both the front and back face are drawn filled

- `glFrontFace( GLenum mode );`
  - Mode is either GL_CCW (default) or GL_CW

- `glCullFace( GLenum mode );`
  - Mode is either GL_FRONT, GL_BACK, GL_FRONT_AND_BACK
  - You must enable and disable culling with
    - `glEnable( GL_CULL_FACE )` or `glDisable( GL_CULL_FACE );`
Drawing Other Objects

- GLU contains calls to draw cylinders, cones and more complex surfaces called NURBS
- GLUT contains calls to draw spheres and cubes
Finishing Up Your OpenGL Program

• OpenGL commands are not executed immediately
  – They are put into a command buffer that gets fed to the hardware

• When you’re done drawing, need to send the commands to the graphics hardware
  – glFlush() or glFinish()
glFlush vs. glFinish

- **glFlush();**
  - Forces all issued commands to begin execution
  - Returns immediately (asynchronous)

- **glFinish();**
  - Forces all issued commands to begin execute
  - Does not return until execution is complete (synchronous)
Matrices in OpenGL

• Vertices are transformed by 2 matrices:
  – ModelView
    • Maps 3D to 3D
    • Transforms vertices from object coordinates to eye coordinates
  – Projection
    • Maps 3D to 2D (sort of)
    • Transforms vertices from eye coordinates to clip coordinates
Matrix in OpenGL

• There are two matrix stacks.
  – ModelView matrix (GL_MODELVIEW)
  – Projection matrix (GL_PROJECTION)

• When we call functions of transformation, we should change to the appropriate matrix stack first.

```c
glMatrixMode(GL_MODELVIEW);
//now we are in modelview matrix stack!
//do modelview transformation here.....

glMatrixMode(GL_PROJECTION);
//now we are in projection matrix stack!
//do projection transformation here....
```
Matrix in OpenGL

- Matrix multiplications always apply to the top of matrix stack.

Top matrix
In the stack

Translation matrix (glTranslatef)
ModelView Matrix

• **Modeling Transformation**
  - Perform rotate, translate, scale and combinations of these transformations to the object.

• **Viewing Transformation**
  - To positioning and aiming the camera
The ModelView Matrix

• In OpenGL, the viewing and modeling transforms are combined into a single matrix - the modelview matrix
  – Viewing Transform - positioning the camera
  – Modeling Transform - positioning the object

• Why?
  – Consider how you would “translate” a fixed object with a real camera
Modeling Transformations

- `glTranslatef(x, y, z)`
  - Multiplies current matrix by a matrix that moves an object by x,y,z

```plaintext
glTranslatef( 0, 0, -1 )
```
Modeling Transformations

- `glRotatef(angle, x, y, z)`
  - Multiplies current matrix by a matrix that rotates an object in a counterclockwise direction about the ray from origin to \((x, y, z)\) with angle as the degrees

```
glRotatef( 45.0, 0, 0, 1)
```
Modeling Transformations

- **glScale{fd} (x, y, z)**
  - Multiplies current matrix by a matrix that scales an object along axes.

```
GLfloat scale[3] = {2.0, -0.5, 1.0};
glScalef(scale[0], scale[1], scale[2]);
```
Viewing Transformations

- `gluLookAt (eyex, eyey, eyez, atx, aty, atz, upx, upy, upz );`
- By default the camera is at the origin, looking down negative z, and the up vector is the positive y axis
Using OpenGL Matrices

• Use the following function to specify which matrix you are changing:
  – glMatrixMode(whichMatrix): whichMatrix = GL_PROJECTION | GL_MODELVIEW

• To guarantee a “fresh start”, use glLoadIdentity();
  – Loads the identity matrix into the active matrix
Using OpenGL Matrices

- To load a user-defined matrix into the current matrix:
  - `glLoadMatrix{fd}(TYPE *m)`
- To multiply the current matrix by a user defined matrix
  - `glMultMatrix{fd}(TYPE *m)`
- SUGGESTION: To avoid row-/column-major confusion, specify matrices as `m[16]` instead of `m[4][4]`
Transforms in OpenGL

• OpenGL uses **4x4** matrices for all its transforms
  – But you don’t have to build them all by hand!
• `glRotate(fd)(angle, x, y, z)`
  – Rotates counter-clockwise by `angle` degrees about the vector `(x, y, z)`
• `glTranslate(fd)(x, y, z)`
• `glScale(fd)(x, y, z)`
• In OpenGL, the last transform in a list is applied **FIRST**
  – Think back to right-multiplication of transforms

```cpp
glTranslatef(1,0,0);
glRotatef(45.0, 0,0,1);
drawObject();
```

```cpp
glRotatef(45.0, 0,0,1);
glTranslatef(1,0,0);
drawObject();
```
Projection Transforms

• The projection matrix defines the viewing volume
  – Used for 2 things:
    • Projects an object onto the screen
    • Determines how objects are clipped

• The viewpoint (the location of the “camera”) that we’ve been talking about is at one end of the viewing volume
Projection Transform

- **Perspective**
  - Viewing volume is a truncated pyramid
    - aka *frustum*

- **Orthographic**
  - Viewing volume is a box
Perspective Projection

• The most noticeable effect of perspective projection is foreshortening
• OpenGL provides several functions to define a viewing frustum
  – glFrustum(...)
  – gluPerspective(...)

![Diagram of perspective projection](image)
glFrustum

- \text{glFrustum(GLdouble } \text{left, GLdouble } \text{right, GLdouble } \text{bottom, GLdouble } \text{top, GLdouble } \text{near, GLdouble } \text{far})$
  - \text{(left, bottom, } -\text{near) and (right, top, } -\text{near) are the bottom-left and top-right corners of the near clip plane}
  - far is the distance to the far clip plane
  - near and far should always be positive
gluPerspective

• This GL Utility Library function provides a more intuitive way (I think) to define a frustum

• gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble near, GLdouble far)
  – fovy - field of view in y (in degrees)
  – aspect - aspect ratio (width / height)
  – near and far - same as with glFrustum()
Orthographic Projection

- With orthographic projection, there is no foreshortening
  - Distance from the camera does not change apparent size
- Again, there are several functions that can define an orthographic projection
  - glOrtho()
  - gluOrtho2D()
glOrtho

- \texttt{glOrtho(GLdouble \textit{left}, GLdouble \textit{right}, GLdouble \textit{bottom}, GLdouble \textit{top}, GLdouble \textit{near}, GLdouble \textit{far})}
  - Arguments are the same as \texttt{glPerspective()}
  - \((\textit{left}, \textit{bottom}, -\textit{near})\) and \((\textit{right}, \textit{top}, -\textit{near})\) are the bottom-left and top-right corners of the near clip plane
  - \textit{near} and \textit{far} can be any values, but they should not be the same
gluOrtho2D

- This GL Utility Library function provides a more intuitive way (I think) to define a frustum
- `gluOrtho2D(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top)`
  - `(left, bottom)` and `(right, top)` define the (x, y) coordinates of the bottom-left and top-right corners of the clipping region
  - Automatically clips to between -1.0 and 1.0 in z
- **In 2D mode, frustum is equal to viewport**
OpenGL Transformations

glTranslatef  gluPerspective
glRotatef      gluOrtho2D
glScalf        gluFrustum
gluLookAt      gluOrtho

Vertex

Modelview Matrix

Projection Matrix

Perspective Division

Viewport Transformation

X
Y
Z
W

eye coordinates
clip coordinates
normalized device coordinates
window coordinates

glViewport()
References

- OpenGL officially website:  
  - [http://www.opengl.org](http://www.opengl.org)
- NeHe (useful installation guides included)  
  - [http://nehe.gamedev.net/](http://nehe.gamedev.net/)
- Nate Robins (for demonstration)  
  - [http://www.xmission.com/~nate/tutors.html](http://www.xmission.com/~nate/tutors.html)
- The Red Book (OpenGL Programming Guide)
The “Red Book”

An older version is available (free!) online:

http://fly.cc.fer.hr/~unreal/therebook/
• Fast Light Toolkit
• Cross-Platform C++ GUI Toolkit
• Provides more full-featured UI functionality than GLUT
  – Also supports GLUT code through emulation
• Download from http://www.fltk.org