Programming using OpenGL: A first Introduction

Introduction to Computer Graphics
Torsten Möller
Today

- Overview
- GL, GLU, GLUT, and GLUI
- First example
- OpenGL functions and states
- complete example: callbacks, simple viewing transformation, etc.
- input / output
Development of the OpenGL API
Early History of APIs

• IFIPS (1973) formed two committees to come up with a standard graphics API
  – Graphical Kernel System (GKS)
    • 2D but contained good workstation model
    – Core
      • Both 2D and 3D
  – GKS adopted as IS0 and later ANSI standard (1980s)

• GKS not easily extended to 3D (GKS-3D)
  – Far behind hardware development
PHIGS and X

• Programmers Hierarchical Graphics System (PHIGS)
  – Arose from CAD community
  – Database model with retained graphics (structures)

• X Window System
  – DEC/MIT effort
  – Client-server architecture with graphics

• PEX combined the two
  – Not easy to use (all the defects of each)
SGI and GL

• Silicon Graphics (SGI) revolutionized the graphics workstation by implementing the pipeline in hardware (1982)
• To access the system, application programmers used a library called GL
• With GL, it was relatively simple to program three dimensional interactive applications
OpenGL

• The success of GL lead to OpenGL (1992), a platform-independent API that was
  – Easy to use
  – Close enough to the hardware to get excellent performance
  – Focus on rendering
  – Omitted windowing and input to avoid window system dependencies
OpenGL Evolution

• Originally controlled by an Architectural Review Board (ARB)
  – Members included SGI, Microsoft, Nvidia, HP, 3DLabs, IBM, …
  – Now Khronos Group
  – Was relatively stable (through version 2.5)
    • Backward compatible
    • Evolution reflected new hardware capabilities
      – 3D texture mapping and texture objects
      – Vertex and fragment programs
  – Allows platform specific features through extensions
OpenGL Architecture
OpenGL is a …

Library for 2D and 3D graphics programming
• 200+ functions for building application programs
• Portable to many platforms (Win, Mac, Unix, Linux)
• Callable from many programming languages (C, Java, Perl, Python)
• Primarily concerned with modeling and rendering operations
  – Specify geometric primitives (lines, pixels, polygons ...)
  – Apply geometric transformations
  – Specify camera, light, color, texture information, etc.
• No windowing or (platform-specific) input/interaction functions — these are the jobs of GLUT
Modern OpenGL

• Performance is achieved by using GPU rather than CPU
• Control GPU through programs called shaders
• Application’s job is to send data to GPU
• GPU does all rendering
OpenGL 3.1

• Totally shader-based
  – No default shaders
  – Each application must provide both a vertex and a fragment shader

• No immediate mode
• Few state variables
• Most 2.5 functions deprecated
• Backward compatibility not required
Other Versions

- **OpenGL ES**
  - Embedded systems
  - Version 1.0 simplified OpenGL 2.1
  - Version 2.0 simplified OpenGL 3.1
    - Shader based
- **WebGL**
  - Javascript implementation of ES 2.0
  - Supported on newer browsers
- **OpenGL 4.1 and 4.2**
  - Add geometry shaders and tessellator
What About Direct X?

- Windows only
- Advantages
  - Better control of resources
  - Access to high level functionality
- Disadvantages
  - New versions not backward compatible
  - Windows only
- Recent advances in shaders are leading to convergence with OpenGL
OpenGL Libraries

• OpenGL core library
  – OpenGL32 on Windows
  – GL on most unix/linux systems (libGL.a)

• OpenGL Utility Library (GLU)
  – Provides functionality in OpenGL core but avoids having to rewrite code
  – Will only work with legacy code

• Links with window system
  – GLX for X window systems
  – WGL for Windows
  – AGL for Macintosh
OpenGL libraries

- **OpenGL**: GL (libGL) and GLU (libGLU)
  - Windows native implementations (opengl32)
  - Mesa 3D: freeware implementation for Linux
- **GLUT**: OpenGL Utility Toolkit (libglut)
  - Window and menu management
  - Mouse and keyboard interactions
  - Other nice utilities, e.g., glutSolidTeapot, etc.
GLUT

- OpenGL Utility Toolkit (GLUT)
  - Provides functionality common to all window systems
    - Open a window
    - Get input from mouse and keyboard
    - Menus
  - Event-driven
    - Code is portable but GLUT lacks the functionality of a good toolkit for a specific platform
    - No slide bars

Not usable anymore after OpenGL 3.2
Graphics libraries

- GLUI: GLUT-based user interface library (libglui)
- Offer controls to OpenGL applications, e.g.,
  - buttons,
  - checkboxes,
  - radio buttons,
  - etc.

Not usable anymore after OpenGL 3.2
freeglut

• GLUT was created long ago and has been unchanged
  – Amazing that it works with OpenGL 3.1
  – Some functionality requires deprecated functions

• freeglut updates GLUT
  – Added new capabilities
  – Context checking

Not usable anymore after OpenGL 3.2
Easy package for I/O

- GLFW: http://www.glfw.org
- Pez: https://github.com/prideout/pez
GLEW

• OpenGL Extension Wrangler Library
• Makes it easy to access OpenGL extensions available on a particular system
• Avoids having to have specific entry points in Windows code
• Application needs only to include glew.h and run a glewInit()
Software Organization

- OpenGL application program
- GLEW
- GL
- GLX
- GLUT
- Xlib, Xt
- Graphics Driver
OpenGL is an API …

- **Application Programmers’ Interface:** a link between
  - low-level: graphics hardware
  - high-level: application program you write

OpenGL ES (for embedded systems): a subset of desktop OpenGL, providing lightweight interface for 2D/3D graphics on mobile and hand-held devices, etc.
OpenGL Functions
OpenGL Functions

• Primitives
  – Points
  – Line Segments
  – Triangles

• Attributes

• Transformations
  – Viewing
  – Modeling

• Control (GLUT, GLFW, Pez)

• Input (GLUT, GLFW, Pez)

• Query
OpenGL State

• OpenGL is a state machine
• OpenGL functions are of two types
  – Primitive generating
    • Can cause output if primitive is visible
    • How vertices are processed and appearance of primitive are controlled by the state
  – State changing
    • Transformation functions
    • Attribute functions
    • Under 3.1 most state variables are defined by the application and sent to the shaders
Lack of Object Orientation

• OpenGL is not object oriented so that there are multiple functions for a given logical function
  – glVertex3f
  – glVertex2i
  – glVertex3dv

• Underlying storage mode is the same

• Easy to create overloaded functions in C++ but issue is efficiency
OpenGL function format

\texttt{glVertex3f(x,y,z)}
\begin{itemize}
  \item \texttt{x, y, z} are floats
  \item \texttt{glVertex3f} belongs to GL library
\end{itemize}

\texttt{glVertex3fv(p)}
\begin{itemize}
  \item \texttt{p} is a pointer to an array
  \item \texttt{p} is a pointer to an array
OpenGL #defines

- Most constants are defined in the include files `gl.h`, `glu.h` and `glut.h`
  - Note `#include <GL/glut.h>` should automatically include the others
  - Examples
    - `glEnable(GL_DEPTH_TEST)`
    - `glClear(GL_COLOR_BUFFER_BIT)`

- Include files also define OpenGL data types: `GLfloat`, `GLdouble`, ...
OpenGL and GLSL

- Shader based OpenGL is based less on a state machine model than a data flow model
- Most state variables, attributes and related pre 3.1 OpenGL functions have been deprecated
- Action happens in shaders
- Job of application is to get data to GPU
GLSL

• OpenGL Shading Language
• C-like with
  – Matrix and vector types (2, 3, 4 dimensional)
  – Overloaded operators
  – C++ like constructors
• Similar to Nvidia’s Cg/CUDA and Microsoft HLSL
• Code sent to shaders as source code
• New OpenGL functions to compile, link and get information to shaders
First Simple Code
First example

• Display 2D points drawing the Sierpinski gasket

1. Choose $p_0$ randomly inside the triangle
2. Choose a triangle vertex randomly
3. $p_1$ is the midpoint
4. Repeat from step 2, replacing $p_0$ by $p_1$

An example of a **fractal**: self-similar geometric structure
Display routine

```c
void display() {
    GLfloat vertices[3][2] = {{0.0,0.0},{25.0,50.0},{50.0,0.0}}; /* A triangle */

    #define GLfloat float

    int j, k, rand(); /* standard random number generator */
    GLfloat p[2] = {7.5, 5.0}; /* arbitrary initial point inside triangle */
    glClear(GL_COLOR_BUFFER_BIT);

    glBegin(GL_POINTS); /* compute and plot 5000 new points */
    for( k=0; k<5000; k++) {
        j=rand()%3; /* pick a vertex at random */

        /* compute point halfway between selected vertex and old point */
        p[0] = (p[0]+vertices[j][0])/2.0;
        p[1] = (p[1]+vertices[j][1])/2.0;

        /* plot new point */
        glVertex2fv(p);
    }
    glEnd();
    glFlush(); /* clear buffers */
}
```

#define GLfloat float

Clear the window to a color specified by a call to glClearColor()

GL_POINTS specify what geometric primitives the vertex calls define.
Other primitives: GL_LINES, GL_POLYGON, etc.

Draw a 2D vertex given as a vector
Also possible, glVertex2i(x, y), glVertex3f(x, y, z), etc.

Force points to be rendered as soon as possible
The Sierpinski gasket
Unresolved issues

• In what colors are we drawing?
• Where on the screen does the image appear?
• How large will the image be?
• How do we create an area of the screen – a window – for our image?
• How long will the image remain on the screen?
• Where is the camera?
Answers

• In what colors are we drawing?
  
  \texttt{glColor3f(1.0, 0.0, 0.0); /* draw in red */}

• Where on the screen does the image appear?
  
  \texttt{glutInitWindowPosition(0,0); /* place window top left on display */}

• How large will the image be?
  
  \texttt{glutInitWindowSize(500,500); /* 500 x 500 pixel window */}

• How do we create an area of the screen – a window – for our image?
  
  \texttt{glutCreateWindow("Sierpinski Gasket"); /* window title */}

• How long will the image remain on the screen?
  
  \texttt{glutMainLoop(); /* enter (infinite) event loop */}
OpenGL functions

• Primitive functions: e.g., points, lines or polygons
• Attribute functions: e.g., color, light, material
• Viewing functions: e.g., camera position
• Input functions (GLUT): e.g., mouse, keyboard
• Control functions: e.g., OpenGL features, start/end
• Query functions, to acquire OpenGL states

• Read text or OpenGL primer and online OpenGL and glut function references (from course web page)
OpenGL as a state machine

- **Input from user program**
  - Description of geometric primitives – scene-changing functions
  - Camera positions and orientations – state-changing functions
  - Light, surface material, color, etc. – state-changing functions

- **Output to display: image pixels**
Partial list of OpenGL states

- Color: glColor3f()
- Line width: glLineWidth()
- Light source parameters: glLight()
- Camera position and orientation: gluLookAt()
- Transformation matrix: glLoadIdentity(), glRotate()
- Etc. – see OpenGL manual

- State variables have default values
- State values are applied to all functions that follow until the next change of state
First Simple Code
A simple program

- Generate a square on a solid background
```
#include <GL/glut.h>
void mydisplay(){
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_POLYGON);
        glVertex2f(-0.5, -0.5);
        glVertex2f(-0.5, 0.5);
        glVertex2f(0.5, 0.5);
        glVertex2f(0.5, -0.5);
    glEnd();
    glFlush();
}

int main(int argc, char** argv){
    glutInit(&argc, argv);
    glutCreateWindow("Simple");
    glutDisplayFunc(mydisplay);
    glutMainLoop();
}
```
Display callback and event loop

- Main program defines a display callback
  - Every glut program must have a display callback
  - Display callback (mydisplay) is executed whenever OpenGL decides the display must be refreshed, e.g., when the window is opened or resized or moved

- Main ends by entering an infinite, interactive event loop
  - Read mouse & keyboard from GLUT/GLUI
  - Use input to change something in the “scene”
  - Redisplay the scene with OpenGL (clear window & redraw)
Compilation

• Unix/linux
  – Compile with –lglut –lGLU –lGL loader flags
  – May have to add –L flag for X libraries
  – Mesa implementation part of most linux distributions
  – Check web for latest on Mesa 3D and glut

• Visual C++ under windows
  – Get glut.h, glut32.lib and glut32.dll from web
  – Create a console application
  – Add opengl32.lib, glu32.lib, glut32.lib to project settings (under link tab)
Programming structure

• Most OpenGL programs have a similar structure that consists of the following functions
  – `main()`:
    • Opens one or more windows with the required properties
    • Defines the callback functions
    • Enters event loop (last executable statement)
  – `init()`: sets the state variables
    • Viewing
    • Attributes
  – Callbacks:
    • Display function
    • Input and window functions
simple.c revisited

• In this version, we shall see the same output but we define all the relevant state values through appropriate function calls

• In particular, we set
  – Colors
  – Viewing conditions
  – Window properties
Main function

#include <GL/glut.h>

int main(int argc, char **argv) {
    glutInit(&argc, argv); /* init GLUT, call before anything */
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(250, 250);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Simple");
    init(); /* initialize display parameters */
    glutDisplayFunc(mydisplay); /* register display callback */
    glutMainLoop();
}

© Machiraju/Zhang/Möller
GLUT functions

- `glutInit` allows application to get command line arguments and initializes system
- `glutInitDisplayMode` requests properties for the window (the rendering context)
  - RGB color
  - Single buffering
  - Properties logically OR-ed together
- `glutWindowSize` in pixels
- `glutWindowPosition` from top-left corner of display
- `glutCreateWindow` create window with title “simple”
- `glutDisplayFunc` display callback
GLUT functions

• `glutMainLoop` enter infinite event loop
• Events include operations such as
  – keyboard press
  – mouse movement
  – mouse button press
  – mouse button release
  – reshape window
  – expose window, etc.
• Event-based programming method
Callback functions

• **What happens when some event occurs?**
  – A callback function is called
  – Programmer can
    • define (register) own callback functions
    • rely on default callback functions
    • do nothing – events ignored

• **GLUT handles definition of callbacks**
  – For display: `glutDisplayFunc()`
  – For mouse movement:
    `glutMouseFunc()`, `glutMotionFunc()`
  – For reshaping of window: `glutReshapeFunc()`
  – For keyboard press: `glutKeyboardFunc()`
init() function

```c
void init()
{
    /* select clearing (background) color */
    glClearColor(0.0, 0.0, 0.0, 1.0);  /* RGBA (alpha for transparency) */
    /* A = 1: fully opaque */

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-1, 1, -1, 1, -1, 1);  /* left, right, bottom, top, znear, zfar */
    /* orthographic projection for viewing */
    /* for 2D, gluOrtho2D(x_min, x_max, y_min, y_max); */
}
```
Projection

• In OpenGL, the projection is carried out by a projection matrix

• Need to set the matrix mode first to know which transformation matrix is being modified (there are others, e.g., GL_MODELVIEW)

   `glMatrixMode(GL_PROJECTION)`

• Transformation functions are incremental so we start with an identity matrix and alter it with a projection matrix that gives the view volume

   `glLoadIdentity();`
   `glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);`
View volume

- In `glOrtho(left, right, bottom, top, near, far)`,
  - Left, ..., top specify left, ..., top clipping planes
  - near and far specify distances from the camera (center of projection)
  - –1 indicates that the near clipping plane is behind the camera

- Two-dimensional vertex commands, e.g., `glVertex2i()`, place all vertices in the plane $z = 0$

- If the application is in two dimensions, we can use the function
  `gluOrtho2D(left, right, bottom, top)`

- In two dimensions, the view or clipping volume becomes a 2D clipping window
mydisplay() function

```c
void mydisplay() {
    glColor3f(1.0, 1.0, 1.0); /* draw primitives in white */

    glBegin(GL_POLYGON);
    /* or GL_LINES or GL_POINTS ... */

    glVertex2f(-.5, -.5);
    glVertex2f(.5, -.5);
    glVertex2f(.5, .5);
    glVertex2f(-.5, .5);
    glEnd();
    glFlush(); /* do not wait, draw it now */
}
```
Output
OpenGL Primitives

GL_POINTS
GL_LINES
GL_LINE_STRIP
GL_LINE_LOOP
GL_TRIANGLES
GL_TRIANGLE_STRIP
GL_TRIANGLE_FAN
GL_POLYGON
GL_QUAD_STRIP
Polygon issues

- OpenGL will only display polygons correctly if
  - simple: edges cannot cross
  - convex
  - flat: all vertices are in the same plane
- User program can check if above true
  - OpenGL will produce output if these conditions are violated but it may not be what is desired
- But note that triangles satisfy all conditions

© Machiraju/Zhang/Möller
Attributes

- Attributes are part of the OpenGL state and determine the appearance of objects
  - Color (points, lines, polygons)
  - Size and width (points, lines)
  - Stipple pattern (lines, polygons)
  - Polygon mode
    - Display as filled: solid color or stipple pattern
  - Light and material properties for shading
RGB color and indexed color

- Each color component is stored separately in the frame buffer
- Usually 8 bits per component in buffer (bit-planes)
- Note in `glColor3f` the color values range from 0.0 (none) to 1.0 (all), whereas in `glColor3ub` (unsigned bytes) the values range from 0 to 255
- Indexed colors: Colors are indices into tables of RGB values
- Requires less memory (indices typically 8 bits) but not as important now — memory is cheap
Smooth color

- Default is **smooth shading**
  - OpenGL interpolates vertex colors across visible polygons
- Alternative is **flat shading**
  - Color of first vertex determines fill color
- OpenGL function:
  ```
  glShadeModel(GL_SMOOTH)  
  or GL_FLAT
  ```
Flat vs. Gouraud Shading

Flat Shading   Gouraud Shading

© Machiraju/Zhang/Möller
Input and Interaction
Interact

Ivan Sutherland, Sketchpad, 1963

Doug Engelbart, 1968

© Machiraju/Zhang/Möller
Paradigm for interactive CG

• “Sketchpad project” by Ivan Sutherland in 1963

• Basic interactive paradigm that characterizes interactive computer graphics (used for CAD in Sketchpad):
  – User sees an image on the display
  – User interacts with the image through an input device (light pen, mouse, trackball)
  – Image changes (object moves, rotates, morphs, etc.)
  – Then repeat

• This is still the basic paradigm we follow today
Input devices

• User inputs control the program through input devices

• Input devices can be viewed as
  – Physical devices: described by their physical properties
    • e.g., mouse, keyboard, trackball, etc.
  – Logical devices: characterized by how they influence the application program
    • e.g., What is returned to program via API
      – An (x, y) position on the screen?
      – An object identifier for a menu item chosen?

• Both are performed by the same physical device, the mouse
Physical input devices

- mouse
- trackball
- light pen
- data tablet
- joy stick
- space ball
Logical input devices

• Older APIs (GKS, PHIGS, not OpenGL) defined six specific types of logical input
  – Locator: return a position, e.g., clicked at by a mouse pointer
  – Choice: return one of n discrete items, e.g., a menu item
  – String: return strings of characters, e.g., via key presses
  – Stroke: return array of positions
  – Valuator: return floating point number
  – Pick: return ID of an object

• OpenGL and GLUT provide functions to handle all these
Event queue

- OpenGL handles input interactions following the event-driven approach.
- Most systems have more than one input device, each of which can be triggered at an arbitrary time by a user.
- Each trigger generates an event whose input to the application program is put in an event queue which is examined by the application program.
- Examples of a trigger: mouse click, release, key stroke, ….
Some event types

- Window: resize, expose, iconify
- Mouse: click one or more buttons
- Motion: move mouse
- Keyboard: press or release a key
- Timer: triggered when the timer has counted down
- Idle: nonevent
  - Define what should be done if no other event is in the even queue
Callbacks

- The event-driven paradigm is implemented using callback functions
- Define a callback function for each type of event the graphics system recognizes
- This user-supplied function is executed when the event occurs
- GLUT example:
  
  glutMouseFunc(mymouse)

mouse callback function
GLUT callbacks

• GLUT recognizes a subset of the events recognized by any particular window system (Windows, X, Macintosh)
  – glutDisplayFunc
  – glutIdleFunc
  – glutMouseFunc
  – glutReshapeFunc
  – glutKeyboardFunc
  – glutMotionFunc
  – glutTimerFunc
  – etc.
GLUT event loop

• Recall that the last line in `main.c` must be `glutMainLoop();`

• which puts the program in an infinite event loop

• In each pass through the event loop, GLUT
  – Looks at the events in the event queue
  – For each event in the queue, GLUT executes the appropriate callback function if one is defined
  – If no callback is defined for the event, the event is ignored
Display callback

- Executed whenever GLUT determines that the window should be refreshed, e.g.,
  - When the window is first opened
  - When the window is reshaped
  - When a window is exposed
  - When the user program decides it wants to change the display

- In `main.c`
  - `glutDisplayFunc(mydisplay)` identifies the function to be executed
  - Every GLUT program must have a display callback
Posting redisplay

- Many events may change the image and thus invoke the display callback function
  - Can lead to multiple executions of the display callback on a single pass through the event loop
- We can avoid this problem by calling
  
  ```
  glutPostRedisplay();
  ```
- from within each such event callback, which sets a flag
- GLUT checks the flag at the end of the event loop
- If flag set, then the display callback function is executed; this ensures that display is redrawn once through a loop
Using the idle callback

• The idle callback is executed whenever there are no events in the event queue

\[
\text{glutIdleFunc}(\text{myidle});
\]

• Useful for animations, e.g.,

```c
void myidle() {
    t += dt;  /* e.g., a rotation angle */
    glutPostRedisplay();
}

void mydisplay() {
    glutClear(...);
    /* draw something that depends on t */
    glutSwapBuffers();
}
```
Animation in a display

• In single-buffer mode, the buffer being drawn to is the buffer being displayed

• Two unsatisfying situations
  – The display is complex and takes longer than one refresh cycle to complete, e.g., partially drawn screen will be cleared
  – The scene needs to be repeatedly cleared and redrawn

• The timing for screen refresh does not synchronize with drawing of display in the buffer

• Partial displays may be seen, generating artifacts
Double buffering

• Instead of one color buffer, we use two
  – Front Buffer: one that is displayed but not written to
  – Back Buffer: one that is written to but not displayed

• Program then requests a double buffer in `main.c`
  – `glutInitDisplayMode(GLUT_RGB|GLUT_DOUBLE)`
  – At the end of the display callback buffers are swapped

```c
void mydisplay()
{
  glutSwapBuffers();
}
```
Using global variables

- The form of all GLUT callbacks is fixed
  - void mydisplay()
  - void mymouse(GLint button, GLint state, GLint x, GLint y)
- Must use global variables to pass information to callbacks

```c
float t; /*global */
void mydisplay()
{
    /* draw something that depends on t */
}
```
The mouse callback

- `glutMouseFunc(mymouse)`
- `void mymouse(GLint button, GLint state, GLint x, GLint y)`
- Returns
  - which button (`GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON`, `GLUT_RIGHT_BUTTON`) caused event
  - state of that button (`GLUT_UP`, `GLUT_DOWN`)
  - position of mouse pointer in window
Positioning

- The position in the screen window is usually measured in pixels with the origin at the top-left corner
  - Consequence of refresh done from top to bottom
- OpenGL uses a world coordinate system with origin at the bottom left
  - Must invert y coordinate returned by callback by height of window:
    \[ y := h - y \]
Obtaining the window size

• To invert the y position we need the window height
  – Height can change during program execution
  – Track with a global variable
  – New height returned to reshape callback that we will look at in detail soon
  – Can also use query functions
    • `glGetIntegerv(state_name, variable)`
    • `glGetFloatv(state_name, variable)`
Using the mouse position

• In the next example, we draw a small square at the location of the mouse each time the left mouse button is clicked.

• This example does not use the display callback but one is required by GLUT; We can use the empty display callback function
  – mydisplay() {}
Example: drawing square

void mymouse(int btn, int state, int x, int y)
{
    if (btn == GLUT_RIGHT_BUTTON && state==GLUT_DOWN)
        exit(0);
    if (btn == GLUT_LEFT_BUTTON && state==GLUT_DOWN)
        drawSquare(x, y);
}

void drawSquare(int x, int y)
{
    y = h – y; /* invert y position */
    glColor3ub( (char) rand()%256, (char) rand()%256, (char) rand()%256);
    /* a random color */
    glBegin(GL_POLYGON);
    glVertex2f(x+size, y+size);
    glVertex2f(x-size, y+size);
    glVertex2f(x-size, y-size);
    glVertex2f(x+size, y-size);
    glEnd();
}
Using the motion callback

• We can draw squares (or anything else) continuously as long as a mouse button is depressed by using the motion callback
  – \texttt{glutMotionFunc(drawSquare)}

• We can draw squares without depressing a button using the passive motion callback
  – \texttt{glutPassiveMotionFunc(drawSquare)}
Keyboard events

• When the mouse is in the window, key press and release generate events
  
  – glutKeyboardFunc (mykey);
  – glutKeyboardUpFunc (mykeyup);

  void mykey(unsigned char key, int x, int y) {
    if (key == ‘Q’ | key == ‘q’) exit(0);
  }

• Returns ASCII code of key pressed and location of mouse pointer

• Note that special keys (F1, arrow) and modifiers (shift) can also be handled
Reshaping the window

• We can reshape and resize the OpenGL display window by pulling the corner of the window

• What happens to the display?
  – Must redraw from application
  – Two possibilities
    • Display part of world
    • Display whole world but force to fit in new window, e.g., may alter aspect ratio
Reshape possibilities

original

reshaped
The reshape callback

- `glutReshapeFunc(myrreshape)`
- `void myreshape(int w, int h)`
  - Returns width and height of new window (in pixels)
  - A redisplay is posted automatically at end of execution of the callback
  - GLUT has a default reshape callback but you probably want to define your own

- Reshape callback is good place to put viewing functions because it is invoked when the window is first opened
Reshape example

- This callback preserves shapes by making the viewport and world window have the same aspect ratio

```c
void myReshape(int w, int h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION); /* switch matrix mode */
    glLoadIdentity();
    if (w <= h)
        gluOrtho2D(-1.0, 1.0, -1.0 * (GLfloat) h / (GLfloat) w, 1.0 * (GLfloat) h / (GLfloat) w);
    else
        gluOrtho2D(-1.0 * (GLfloat) w / (GLfloat) h, 1.0 * (GLfloat) w / (GLfloat) h, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW); /* return to modelview mode */
}
```
GLUT menus

- GLUT supports pop-up menus
  - A menu can have submenus

- Three steps
  - Define entries for the menu
  - Define action for each menu item
    - Action carried out if entry selected
  - Attach menu to a mouse button
Defining a simple menu

Menu identifier; can be used to set current menu

```c
menu_id = glutCreateMenu(mymenu);
glutAddMenuEntry("clear Screen", 1);
glutAddMenuEntry("exit", 2);
glutAttachMenu(GLUT_RIGHT_BUTTON);
```

Menu item identifiers

<table>
<thead>
<tr>
<th>clear screen</th>
</tr>
</thead>
<tbody>
<tr>
<td>exit</td>
</tr>
</tbody>
</table>
Menu actions

- Menu callback

```c
void mymenu(int id)
{
    if (id == 1) glClear();
    if (id == 2) exit(0);
}
```

- Each menu has an id that is returned when it is created; the id can be used to set current menu, using `glutSetMenu(…)`

- Submenus can also be created: `glutAddSubMenu(…)`
Other functions in GLUT

• Dynamic Windows: glutDestroyWindow()
  – Create and destroy during execution
• Sub-windows: glutCreateSubWindow()
• Multiple Windows
• Changing callbacks during execution, e.g., set to NULL
• Timers: glutTimerFunc()
• OpenGL texts and portable fonts
  – glRasterPos*() for positioning and
    glutBitmapCharacter() for writing
  – glutStrokeCharacter()