Programming with Legacy OpenGL
(Pre OpenGL 3.1)

Prof. George Wolberg
Dept. of Computer Science
City College of New York
Objectives

• Build a complete first program
  - Introduce a standard program structure using basic OpenGL (pre 3.1)
  - Why start with pre OpenGL 3.1 (legacy code)?
    • Easier learning curve
    • Familiarity with lots of existing code already written in it.

• Simple viewing
  - Two-dimensional viewing as a special case of three-dimensional viewing

• Initialization steps and program structure
OpenGL Camera

- OpenGL places a camera at the origin in object space pointing in the negative \( z \) direction.
- The default viewing volume is a box centered at the origin with sides of length 2.
Orthographic Viewing

In the default orthographic view, points are projected forward along the z axis onto the plane z=0.
Viewports

• Do not have to use the entire window for the image: `glViewport(x, y, w, h)`
• Values in pixels (window coordinates)
Program Structure

• Most OpenGL programs have a similar structure that consists of the following functions

  - **main()**:  
    • Opens main window with control panel and OpenGL canvas  
    • Enters event loop (last executable statement)

  - **initializeGL()**: sets the state variables  
    • Viewing  
    • Attributes

  - **resizeGL()**: handles window resizing event  
    • Sets viewport  
    • Sets viewing coordinates for orthographic or perspective projection

  - **paintGL ()**: render scene  
    • Clear framebuffer  
    • Call glVertex*() to draw primitives (triangles, polygons)
initializeGL()

```c
void initializeGL()
{
    glClearColor (0.0, 0.0, 0.0, 1.0);
    glColor3f(1.0, 1.0, 1.0);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();    glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();    glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
}
```

black clear color
opaque window
fill with white
viewing volume
Transformations and Viewing

• In OpenGL, the projection is carried out by a projection matrix (transformation)
• There is only one set of transformation functions so we must set the matrix mode first
  \texttt{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
  \texttt{glLoadIdentity ();}
  \texttt{glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);}
2D and 3D Viewing

- In `glOrtho(left, right, bottom, top, near, far)` the near and far distances are measured from the camera.
- 2D vertex commands place all vertices in the plane $z=0$.
- In 2D, the view or clipping volume becomes a *clipping window*.
void paintGL()
{
    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();
}
Pre-OpenGL 3.1 Primitives

GL_POINTS
GL_LINES
GL_LINE_STRIP
GL_LINE_LOOP
GL_TRIANGLES
GL_TRIANGLE_STRIP
GL_TRIANGLE_FAN
GL_QUAD_STRIP
GL_POLYGON
Example: Drawing an Arc

• Given a circle with radius $r$, centered at $(x,y)$, draw an arc of the circle that sweeps out an angle $\theta$.

$$(x, y) = (x_0 + r \cos \theta, y_0 + r \sin \theta),$$

for $0 \leq \theta \leq 2\pi$. 
Example Using Line Strip Primitive

```c
void drawArc(float x, float y, float r,
             float t0, float sweep)
{
    float t, dt; // angle
    int n = 30;   // # of segments
    int i;

    t = t0 * PI/180.0; // radians
    dt = sweep * PI/(180*n); // increment

    glBegin(GL_LINE_STRIP);
    for(i=0; i<=n; i++, t += dt)
        glVertex2f(x + r*cos(t), y + r*sin(t));
    glEnd();
}
Color and State

• The color as set by `glColor` becomes part of the state and will be used until changed
  - Colors and other attributes are not part of the object but are assigned when the object is rendered

• We can create conceptual *vertex colors* by code such as

  ```
  glColor
  glVertex
  glColor
  glVertex
  ```
First Assignment:  
Tessellation and Twist

• Consider rotating a 2D point about the origin

\[ x' = x \cos \theta - y \sin \theta \]

\[ y' = x \sin \theta + y \cos \theta \]

• Now let amount of rotation depend on distance from origin giving us twist

\[ x' = x \cos(d\theta) - y \sin(d\theta) \]

\[ y' = x \sin(d\theta) + y \cos(d\theta) \]

\[ d \propto \sqrt{x^2 + y^2} \]
Example

triangle

triangle

tessellated triangle

twist without tessellation

twist after tessellation

Angel and Shreiner: Interactive Computer Graphics 7E © Addison-Wesley 2015
void initializeGL()
{
    // init vertex and color buffers
    initBuffers();

    // init state variables
    glClearColor (0.0, 0.0, 0.0, 1.0);
    glColor3f(1.0, 1.0, 1.0);
}
void resizeGL(int w, int h) {
    // compute aspect ratio
    float ar = (float) w / h;

    // set xmax, ymax
    float xmax, ymax;
    if(ar > 1.0) { // wide screen
        xmax = ar;
        ymax = 1.;
    } else { // tall screen
        xmax = 1.;
        ymax = 1/ar;
    }

    // set viewport to occupy full canvas
    glViewport(0, 0, w, h);

    // init viewing coordinates for orthographic projection
    glLoadIdentity();
    glOrtho(-xmax, xmax, -ymax, ymax, -1.0, 1.0);
}
typedef QVector2D vec2;
typedef QVector3D vec3;
std::vector<vec2> m_points;
std::vector<vec3> m_colors;

void paintGL()
{
    // clear canvas with background values
    glClear(GL_COLOR_BUFFER_BIT);

    // draw all points in m_points
    for(uint i=0, j=0; i<m_colors.size(); ++i) {
        // set color
        glColor3f(m_colors[i][0], m_colors[i][1], m_colors[i][2]);

        glBegin(GL_TRIANGLES);
            glVertex2f(m_points[j][0], m_points[j][1]); j++;
            glVertex2f(m_points[j][0], m_points[j][1]); j++;
            glVertex2f(m_points[j][0], m_points[j][1]); j++;
        glEnd();
    }
}
void initBuffers()
{
    // init triangle vertices
    const vec2 v[] = {
        vec2( 0.0 ,  0.75),
        vec2( 0.65, -0.375),
        vec2(-0.65, -0.375)
    };

    // recursively subdivide triangle;
    // store vertices and colors in m_points[] and m_colors[]
    divideTriangle(v[0], v[1], v[2], m_subdivisions);
}
```c
void divideTriangle(vec2 a, vec2 b, vec2 c, int count)
{
    if(count > 0) {
        vec2 ab = vec2((a[0]+b[0]) / 2.0, (a[1]+b[1]) / 2.0);
        vec2 ac = vec2((a[0]+c[0]) / 2.0, (a[1]+c[1]) / 2.0);
        vec2 bc = vec2((b[0]+c[0]) / 2.0, (b[1]+c[1]) / 2.0);
        divideTriangle( a, ab, ac, count-1);
        divideTriangle( b, bc, ab, count-1);
        divideTriangle( c, ac, bc, count-1);
        divideTriangle(ab, ac, bc, count-1);
    } else triangle(a, b, c);
}
```
void triangle(vec2 a, vec2 b, vec2 c)
{
    if(m_updateColor) {
        m_colors.push_back(vec3((float) rand()/RAND_MAX,
                        (float) rand()/RAND_MAX,
                        (float) rand()/RAND_MAX));
    }

    // init geometry
    m_points.push_back(rotTwist(a));
    m_points.push_back(rotTwist(b));
    m_points.push_back(rotTwist(c));
}
rotTwist()

```c
void rotTwist(vec2 p) {
    float d = m_twist ? sqrt(p[0][0]*p[0][0] + p[1][0]*p[1][0]) : 1;
    float sinTheta = sin(d*m_theta);
    float cosTheta = cos(d*m_theta);
    return vec2(p[0]*cosTheta - p[1]*sinTheta,
                p[0]*sinTheta + p[1]*cosTheta);
}
```
Programming with OpenGL: Sierpinski Gasket Example (3D)

Prof. George Wolberg
Dept. of Computer Science
City College of New York
Objectives

- Develop a more sophisticated 3D example
  - Sierpinski gasket: a fractal
- Introduce hidden-surface removal
Three-dimensional Applications

• In OpenGL, two-dimensional applications are a special case of three-dimensional graphics

• Going to 3D
  - Not much changes
  - Use `vec3`, `glUniform3f`
  - Have to worry about the order in which primitives are rendered or use hidden-surface removal
Sierpinski Gasket (2D)

- Start with a triangle

- Connect bisectors of sides and remove central triangle

- Repeat
Example

• Five subdivisions
The gasket as a fractal

• Consider the filled area (black) and the perimeter (the length of all the lines around the filled triangles)

• As we continue subdividing
  - the area goes to zero
  - but the perimeter goes to infinity

• This is not an ordinary geometric object
  - It is neither two- nor three-dimensional

• It is a fractal (fractional dimension) object
Gasket Program

// initial triangle
vec2 v[3] = {vec2(-1.0, -0.58),
            vec2( 1.0, -0.58),
            vec2( 0.0,  1.15)};

int n; // number of recursive steps
Draw one triangle

// display one triangle
void triangle(vec2 a, vec2 b, vec2 c)
{
    static int i = 0;

    points[i] = a;
    points[i+1] = b;
    points[i+2] = c;
    i += 3;
}

// triangle subdivision using vertex numbers
void divide_triangle(vec2 a, vec2 b, vec2 c, int m) {
    vec2 ab, ac, bc;

    if (m > 0) {
        ab = (a + b)/2;
        ac = (a + c)/2;
        bc = (b + c)/2;
        divide_triangle(a, ab, ac, m-1);
        divide_triangle(c, ac, bc, m-1);
        divide_triangle(b, bc, ab, m-1);
    } // else, draw triangle at end of recursion
    else {triangle(a,b,c);}
}
display and init Functions

```c
void paintGL()
{
    glClearColor(GL_COLOR_BUFFER_BIT);
    glDrawArrays(GL_TRIANGLES, 0, NumVertices);
}

void initializeGL()
{
    ...
    // v: initial triangle vertices
    // n: number of recursive steps
    divide_triangle(v[0], v[1], v[2], n);
    ...
}
```
Moving to 3D

• We can easily make the program three-dimensional by using

  \texttt{vec3 \ v[3]}

and starting with a tetrahedron
3D Gasket

• We can subdivide each of the four faces

• Appears as if we remove a solid tetrahedron from the center leaving four smaller tetrahedra

• Code almost identical to 2D example
Triangle code

// display one triangle
void triangle(vec3 a, vec3 b, vec3 c) {
    static int i = 0;

    points[i] = a;
    points[i+1] = b;
    points[i+2] = c;
    i += 3
}

Subdivision code

// triangle subdivision using vertex numbers
void divide_triangle(vec3 a, vec3 b, vec3 c, int m) {
    vec3 ab, ac, bc;

    if(m > 0) {
        ab = (a + b)/2;
        ac = (a + c)/2;
        bc = (b + c)/2;
        divide_triangle(a, ab, ac, m-1);
        divide_triangle(c, ac, bc, m-1);
        divide_triangle(b, bc, ab, m-1);
    } else { // else, draw triangle at end of recursion
        triangle(a, b, c);
    }
}
Tetrahedron code

```c
void tetrahedron(int m)
{
    glColor3f(1.0,0.0,0.0);
    divide_triangle(v[0], v[1], v[2], m);

    glColor3f(0.0,1.0,0.0);
    divide_triangle(v[3], v[2], v[1], m);

    glColor3f(0.0,0.0,1.0);
    divide_triangle(v[0], v[3], v[1], m);

    glColor3f(0.0,0.0,0.1.0);
    divide_triangle(v[0], v[3], v[1], m);

    glColor3f(0.0,0.0,0.0);
    divide_triangle(v[0], v[2], v[3], m);
}
```
Almost Correct

• Because the triangles are drawn in the order they are specified in the program, the front triangles are not always rendered in front of triangles behind them.
Hidden-Surface Removal

• We want to see only those surfaces in front of other surfaces

• OpenGL uses a *hidden-surface* method called the z-buffer algorithm that saves depth information as objects are rendered so that only the front objects appear in the image
Using the Z-buffer algorithm

• The algorithm uses an extra buffer, the z-buffer, to store depth information as geometry travels down the pipeline
• It must be
  - Enabled in `initializeGL()`
    • `glEnable(GL_DEPTH_TEST)`
  - Cleared in the `paintGL()`
    • `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)`
Surface vs. Volume Subdivision

- In our example, we divided the surface of each face.
- We could also divide the volume using the same midpoints.
- The midpoints define four smaller tetrahedrons, one for each vertex.
- Keeping only these tetrahedrons removes a volume in the middle.
Volume Subdivision
Input and Interaction

Prof. George Wolberg
Dept. of Computer Science
City College of New York
Objectives

- Event-driven input
- Callback functions / slot functions
- Window resize functions
  - Alter aspect ratio
  - Preserve aspect ratio
Event Mode

• Most systems have more than one input device, each if which can be triggered at an arbitrary time by a user

• Each trigger generates an event whose measure is put in an event queue which can be examined by the user program
Event Types

- Window: resize, expose, iconify
- Mouse: click one or more buttons
- Motion: move mouse
- Keyboard: press or release a key
Callbacks

• Programming interface for event-driven input
• Define a callback function for each type of event the graphics system recognizes
• In Qt, this function is known as the slot function
• This user-supplied function is executed when the event occurs

• Qt example:

```cpp
QSlider m_sliderTheta;
QSpinBox m_spinBoxTheta;
connect(m_sliderTheta, SIGNAL(valueChanged(int)), this, SLOT(changeTheta(int)));
cconnect(m_spinBoxTheta, SIGNAL(valueChanged(int)), this, SLOT(changeTheta(int)));
```
void changeTheta(int angle)
{
    // update slider and spinbox
    m_sliderTheta->blockSignals(true);
    m_sliderTheta->setValue(angle);
    m_sliderTheta->blockSignals(false);

    m_spinBoxTheta->blockSignals(true);
    m_spinBoxTheta->setValue(angle);
    m_spinBoxTheta->blockSignals(false);

    m_theta = angle * (M_PI/180.);  // convert to radians
    m_points.clear();  // clears points vector
    initBuffers();  // recalculates points
    updateGL();  // redraw: invokes paintGL()
}
Qt Event Loop

• Remember that the last line in main.c for a program using Qt must be \texttt{return app.exec();}

\texttt{#include “MainWindow.h”}  // UI window header

\texttt{int main(int argc, char **argv)}

\{ 
  \texttt{QApplication app(argc, argv);}  // create application
  \texttt{MainWindow window;}  // create UI window
  \texttt{window.showMaximized();}  // display window
  \texttt{return app.exec();}  // infinite processing loop
\}
Infinite Event Loop

- In each pass through the event loop, Qt
  - looks at the events in the queue
  - for each event in the queue, Qt executes the appropriate slot function if one is defined
  - if no slot is defined for the event, the event is ignored
paintGL()

- The paintGL() function is executed whenever Qt determines that the window should be refreshed, for example
  - 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
- Every Qt/OpenGL program must have a paintGL()
Posting Displays

• Many events may invoke paintGL()
  - Can lead to multiple executions of the display callback on a single pass through the event loop

• We can avoid this problem by instead using
  
  ```
  updateGL();  // if using QGLWidget
  update();   // if using QOpenGLWidget
  ```
  which sets a flag.

• Qt checks to see if the flag is set at the end of the event loop

• If set, then the paintGL() function is executed
Animating a Display

• When we redraw the display through the display callback, we usually start by clearing the window
  - `glClear(GL_COLOR_BUFFER_BIT)`
  then draw the altered display

• Problem: the drawing of information in the frame buffer is decoupled from the display of its contents
  - Graphics systems use dual ported memory

• Hence we can see partially drawn display
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
• Handled automatically by QGLWidget() in Qt.
In `initializeGL()`:

```cpp
QBasicTimer timer;  // faster than QTimer
timer.start(12, this);  // countdown 12 msec
```

In `MainWidget` class:

```cpp
void MainWidget::timerEvent(QTimerEvent *)
{
    angularSpeed *= 0.99;  // decrease angular speed (friction)
    // stop rotation when speed goes below threshold
    if(angularSpeed < 0.01) angularSpeed = 0.0;
    else {
        // update rotation
        rotation = QQuaternion::fromAxisAndAngle(rotationAxis,
                                                 angularSpeed) * rotation;
        update();  // request an update
    }
}
```
Positioning

- Positions in the screen window are usually measured in pixels with the origin at the top-left corner. The 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 resizeGL()
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
      – Can alter aspect ratio
Reshape possibilities

original

reshaped
The `resizeGL()` function is a good place to put camera functions because it is invoked when the window is first opened.

```c
void resizeGL(int w, int h)
```
Pre-OpenGL 3.0: Reshape Example #1

- This reshape fct *does not preserve* shapes; it ignores the aspect ratio between the viewport and world window

```c
void resizeGL(int w, int h)
{
    // set viewport to occupy full window
    glViewport(0, 0, w, h);

    // init viewing coordinates for orthographic projection
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-1., 1., -1., 1., -1., 1.);
}
```
Pre-OpenGL 3.0: Reshape Example #2

• This reshape fct *preserves* shapes by making the viewport and world window have the same aspect ratio

```c
void resizeGL(int w, int h)
{
    // compute aspect ratio
    float ar = (float) w / h;

    // set xmax, ymax
    float xmax, ymax;
    if(ar > 1.0) {  // wide screen
        xmax = ar;
        ymax = 1;
    } else {        // tall screen
        xmax = 1;
        ymax = 1 / ar;
    }
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-xmax, xmax, -ymax, ymax, 1., 1.);
}
```
Modern OpenGL: Reshape Example (with Qt)

- This reshape fct *preserves* shapes by making the viewport and world window have the same aspect ratio. Uses Qt.

```cpp
QString m_projection;
void resizeGL(int w, int h)
{
    // compute aspect ratio
    float ar = (float) w / h;

    // set xmax, ymax
    float xmax, ymax;
    if(ar > 1.0) {  // wide screen
        xmax = ar;
        ymax = 1;
    } else {  // tall screen
        xmax = 1;
        ymax = 1 / ar;
    }

    glViewport(0, 0, w, h);
    m_projection.setToIdentity();
    m_projection.ortho(-xmax, xmax, -ymax, ymax, 1., 1.);
}
```