Programming using OpenGL
3.1: A First Introduction

Richard (Hao) Zhang

Introduction to Computer Graphics
CMPT 361 – Lecture 6
Outline

- OpenGL API overview and modern architecture
- OpenGL libraries: GL, GLUT, GLEW, and GLUI
- OpenGL functions and shaders (GLSL)
- Program structure through a simple program
  - Basic program structures: main, init, callbacks, etc.
  - Specifics on vertex and fragment shaders
  - Primitives (polygons), colors, and attributes
- Input and interactions

Readings: Chapter 2
OpenGL

- GL = Graphics Library
- Success of SGI’s GL led to OpenGL (1992), a platform-independent API that was
  - Easy to use
  - Close enough to the hardware for performance
  - Focus on rendering, very basic rendering
  - Omitted windowing and input interactions to avoid dependency on specific platform deployed
OpenGL is an API …

Application programmers’ interface: link between
- low-level: graphics hardware
- high-level: application program you write

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

WebGL: Javascript API for 2D/3D graphics within web browsers

January 29, 2018
Kind of a black box

- A system described only by input and output
- Internal working no really need to know
- For graphics functions

Creating primitives
Color control
Shading params
Keyboard events etc.

Primarily graphics sent to output devices
OpenGL functions

- **Primitive functions**
  - Low-level: points, line segments, triangles (polygons), text (stroke or raster)

- **Attribute functions**
  - Color, line thickness, material properties, etc.

- **Viewing and transformations**
  - Viewing (e.g., projection), transforming (e.g., rotation)

- **Control, e.g., of windows,** — **GL utility functions**

- **Input and interaction** — **GL utility functions**

- **Query functions, e.g., of certain OpenGL states**
OpenGL: state machine (SM)

- OpenGL is a state machine (more so pre-3.1)
- OpenGL functions mainly 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 or querying
    - Transformation functions
    - Attribute functions
- Fewer functions cause graphics output
Consequences of SM view

- Parameters are persistent
  - Once set, values remain, until changed next time
- Attributes are not really bound to primitives
  - They are bound to states
OpenGL interface libraries

- OpenGL core library (GL)
  - OpenGL32 on Windows
  - GL on most unix/linux systems (libGL.a)
- Shaders in OpenGL Shader Language (GLSL)
- OpenGL Utility Library (GLU)
  - Provides functionality to avoid rewriting code
  - Will now only work with legacy code
- Links with different window systems
  - E.g., WGL for Windows and AGL (Apple Graphics Library) for Macintosh
GLUT

- OpenGL Utility Toolkit (GLUT): provides functionality expected of all window systems
  - Opening a window, reshaping, adds name, etc.
  - Get input from mouse and keyboard; menus
  - Implements event-driven paradigm and **callbacks**
  - **Other nice utilities, e.g., glutSolidTeapot, etc.**
    
    ```
    #include <GL/glut.h>
    ```

- Code is portable but lacks the functionality of a complete toolkit (see freecglut)
  - No slide bars, etc.
GLUT was created long time ago (version 3.7 from 1998) and has been unchanged
- Amazing that it works mostly with OpenGL 3.1
- Some functionality are out-dated since it requires deprecated OpenGL functions

freeglut updates GLUT
- Same call format, e.g., glutMainLoop(), glutInit(), …
- Added capabilities
- See: http://freeglut.sourceforge.net/
GLEW

- OpenGL Extension Wrangler Library
- Makes it easy to access OpenGL extensions available on a particular system
- Application needs only to include glew.h and run a glewInit()
Graphics libraries

- GLUI: GLUT-based user interface library (libglui)

- Offer controls, e.g.,
  - buttons,
  - checkboxes,
  - radio buttons,
  - etc.

  to OpenGL applications

- But also old; now better choices include Qt & Tk
Pipelined architecture

- Operations lined up in a pipeline to allow parallelization

- Imagine many arithmetic operations flowing through the pipeline

- Parallel executions in different boxes

An arithmetic pipeline
Modern OpenGL

- Graphics performance is now achieved by using **GPU** rather than CPU
- Control GPU via programs called shaders
- Application’s job is to send data to GPU
- GPU does all rendering

Geometric pipeline
GPU vs. CPU

Theoretical GFLOP/s
1500
1250
1000
750
500
250
0
NVIDIA GPU Single Precision
NVIDIA GPU Double Precision
Intel CPU Single Precision
Intel CPU Double Precision
GeForce GTX 480
GeForce GTX 280
GeForce 8800 GTX
GeForce 7800 GTX
GeForce 6800 Ultra
GeForce FX 5800
Pentium 4
Jan-03
Oct-05
Mar-07
Harpertown
Feb-01
Jun-04
Dec-09
Westmere
Bloomfield
Woodcrest
Tesla C 1060
Tesla C 2050

January 29, 2018
A more recent plot

Mac Pro CPU/GPU Performance Evolution

- Slowest GPU Option
- Fastest GPU Option
- Single Threaded CPU Perf (Fastest CPU)
- Multithreaded CPU Perf (Fastest CPU)
OpenGL 3.1 and beyond

- **Shader-based**
  - No default shaders
  - Each application must provide both a vertex shader and a fragment shader: two files
- No **immediate mode** --- next slide
- Fewer state variables (contrast to older OpenGL)
- Most OpenGL 2.5 functions deprecated
- No backward compatibility
Immediate vs. retained modes

- **Immediate**: display a primitive as soon as it is created at the CPU side
  - No memory, e.g., arrays, to store data
  - But lots of overhead switching among data processing, data transfer, and display handling

- **Retained mode**: store all primitive data in an array before sending data for display

- Modern GPU system: retained but send data to GPU and do processing/display there
OpenGL function format

- **function name**: `glUniform3f(x, y, z)`
- **belongs to GL library**: `glUniform3f` function
- **dimensions (2, 3, 4, or matrix)**
- **x, y, z are floats**
- **p is a pointer to an array**: `glUniform3fv(p)`

- `x`, `y`, `z` are floats
- `p` is a pointer to an array
glUniform

- Used to transfer data to shader
- `glUniform1f()`: transfers a floating point number such as a time value
- `glUniform3iv()`: transfers an integer position in 3D through pointer to 3D array of (int)’s
- `glUniformMatrix4fv()`: transfers a 4x4 matrix of floating point numbers (float)’s
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)
- glClearColor(GL_COLOR_BUFFER_BIT)

Include files also define OpenGL data types: GLfloat, GLdouble, ...
Shader-based OpenGL is based less on a state machine model and more on a data flow model.
- Data flow from CPU (application) to GPU
- Most state variables, attributes and related pre-3.1 functions have been deprecated
- Actions happen in shaders!
- Job of application program to send 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 and Microsoft HLSL
- Code sent to shaders as source code: *.glsl
- New OpenGL functions to compile, link and get information to shaders
Program structure

- Build a complete first program
  - Introduce shaders
  - Introduce a standard program structure
- Simple viewing
  - Two-dimensional viewing as a special case of three-dimensional viewing
- Initialization steps and program structure
Basic program structure

- Most OpenGL programs have a similar structure that consists of the following functions
  - **main()**:
    - The driver program
    - Enters event loop (last executable statement)
  - **init()**:
    - Sets the state variables
    - Specifies the callback functions (could also be in main())
    - Opens one or more windows with the required properties
    - Viewing parameters set up
    - Attribute specifications
    - Link app program and shaders by calling InitShader()
Most OpenGL programs have a similar structure that consists of the following functions:

- **InitShader**("vsource.glsl", "fsource.glsl"): Read shader source from files, e.g., "vsource.glsl"
  - Create geometry
  - Read, compile and link shaders
  - See text Section 2.8.6 and Appendix A.1 of 6th edition

- Registrating callbacks: what happens when triggered
  - Display function
  - Input, interaction, and window functions
Typical application

- Application program: *.cpp
  - main(), init(), callbacks, geometry processing
- A vertex shader: *.glsl
- A fragment shader: *.glsl
- Include files in include/, e.g., header files and InitShader.cpp
- See example on Serpinski gasket app
```c
#include <GL/glew.h>
#include <GL/glut.h>  // includes gl.h

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE|GLUT_RGB);
    glutInitWindowSize (500,500);
    glutInitWindowPosition (0,0);
    glutCreateWindow("simple");
    glutDisplayFunc (mydisplay);
    glewInit();
    init();
    glutMainLoop();
}
```

- specify window properties
- display callback
- set OpenGL state and initialize shaders
- enter event loop
GLUT functions

- **glutInit** allows application to get command line arguments and initializes system.
- **gluInitDisplayMode** requests properties for the window (the rendering context), e.g., RGB color, single buffering: properties logically ORed together.
- **glutInitWindowSize** in pixels.
- **glutInitWindowPosition** from top-left corner of display.
- **glutCreateWindow** create window with title “simple”.
- **glutDisplayFunc** display callback.
- **glutMainLoop** enter infinite event loop.
Display callback

- Display callback invoked when the window is redrawn, e.g., first opening, resized, etc.
- Once we get data onto the GPU (how? Later), we can initiate the rendering with a simple callback

```c
void mydisplay()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glDrawArrays(GL_TRIANGLES, 0, 3);
    glFlush();
}
```

- Arrays are **buffer objects** that contain per-vertex info
- `glDrawArrays` draws from the **active object** --- SM concept
Vertices can have many attributes
- Position
- Color
- Texture Coordinates
- Application data

A vertex array holds these data
Bundles all vertex data (positions, colors, ..,)

Generate **name(s)** for vertex array(s) then bind

```c
GLuint abuffer;
glGenVertexArrays(1, &abuffer);
glBindVertexArray(abuffer);
```

At this point we have a current vertex array with name only but no contents (next slide)

First time `glBindVertexArray` is call for a name, the object with that name is created

Later use of `glBindVertexArray` lets us switch between multiple VAOs --- **activate** a VAO for `glDrawArrays`
Buffer objects (VBOs)

- Buffer objects (on the GPU) allow us to transfer large amounts of data from the app program to the GPU.
- Create (buffer ID); bind (buffer to ID); input data to buffer.

```c
GLuint buffer;
glGenBuffers(1, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(points), points, GL_STATIC_DRAW);
```

- Data in current vertex buffers/arrays is sent to GPU.

Array defined in app program.
Vertex array objects and buffer objects can be set up in `init()`

- Set clear color and other OpenGL parameters
- Set up shaders also as part of initialization
  - Read
  - Compile
  - Link

- First let us consider a few other issues
The units in **points** are determined by the application and are called
- *object* (or *model*) coordinates
- *world* coordinates

Viewing specifications usually are also in object coordinates transformed through
- *eye* (or *camera*) coordinates
- *clip* coordinates
- normalized *device* coordinates
- *window* (or *screen*) coordinates

OpenGL also uses some internal reps that usually are not visible to the app but are important in the shaders
OpenGL places a camera at the origin in object space pointing in the negative z direction.

The default viewing volume:
- a box centered at the origin with sides of length 2.
Orthographic viewing

- **Default** orthographic view: points are projected 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)
In OpenGL, projection is carried out by a projection matrix (transformation)

Transformation functions are also used for changes in coordinate systems

Pre-3.0 OpenGL had a set of transformation functions, e.g., glRotate(), which have been deprecated

Some choices now:
- Perform in application code
- GLSL functions in vertex shader
OpenGL shaders

- Simple Shaders
  - Vertex shader
  - Fragment shader
- Vertex and fragment shaders in separate files and must be included
- Linked to app program after compilation
Execution model
Use of vertex shaders

- When executing `glDrawArray()`, each of the N vertices invokes execution of the vertex shader
  
  `glDrawArray(GL_POINTS, 0, N)`

- Moving vertices: morphing, wave motion, general mesh deformation, etc.

- Lighting: more realistic models, advanced shading operations, e.g., cartoon shader, etc.
Use of fragment shader

- Vertex shader causes vertices to be generated and passed to clipper and primitive assembly
- Primitive assembly generates fragments
- Each fragment generation invokes a call to fragment shader: per-fragment (pixel) operation

per vertex lighting per fragment lighting
Fragment shade applications

- **Image-space** techniques, e.g., texture mapping and variations

- smooth shading
- environment mapping
- bump mapping
Simplest vertex shader

- Placed in a vShader file, “vsSource.glsI”
- Include into program by creating a “program object” (see initShader() in text)

```glsl
in vec4 vPosition;
void main(void) {
    gl_Position = vPosition;
}
```
A “pass-through” vertex shader

- Input vertex positions are in vPosition
- “in” signifies its values are input to shader when the shader is initiated
- gl_Position: special state variable
  - gives positions to be passed to the rasterizer
  - Must be specified by every vertex shader
- Typically, vertex shader applies transformations to vertices: coordinate system transforms
Simplest fragment shader

- Placed in a fShader file, “fsource.glsl”
- Include into program by creating a “program object” (see initShader() in text)

```c
void main(void)
{
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```
Setting color

- At the minimum, fragment shader outputs a color for each fragment
- `gl_FragColor`: a built-in state variable
  - Specifies color of fragment
  - 4D vector gives RGBA color specification (later)
  - $R = 1.0$ means red, $A = 1.0$ means opaque
Link shader with app program

- Read shaders (from shader source files)
- Compile shaders (see InitShader() in appendix)
- Create a program object
- Link everything together (see appendix)
- **Link variables in application with variables in shaders** (see text Section 2.8.6)
  - Vertex attributes
  - Uniform variables (pass info from app to shader)
- There are other variables pass info between shaders
Summary

- Three components
  - Application program: main() + any other files
  - Vertex shader: own file, must be provided
  - Fragment shader: own file, must provide
- Shaders initiated and linked: InitShader(), from within the main(), typically init(), program
- Variables linked among three components
- Callbacks and then main event loop
OpenGL geometric primitives

- GL_POINTS
- GL_LINES
- GL_LINE_STRIP
- GL_LINE_LOOP
- GL_TRIANGLES
- GL_TRIANGLE_STRIP
- GL_TRIANGLE_FAN
Polygon issues

- OpenGL will only display triangles!
  - **simple**: edges cannot cross
  - **convex**
  - **Flat**: all vertices are in the same plane

- Application program must tessellate a polygon into triangles (triangulation)
- OpenGL 4.1 contains a tessellator
Conceptually simple to test for simplicity and convexity, but time consuming

Earlier OpenGL versions have GL_POLYGON, assumed both and left testing to the application

Present version only renders triangles

Need algorithm to triangulate an arbitrary polygon (or surface)
Good and bad triangles

- Long thin triangles rendered badly
- Equilateral triangles render well
- Maximize minimum angle ($\neq$ MinMax angle)
- Delaunay triangulation for unstructured points
Attributes determine the appearance of objects

- Color (points, lines, triangles)
- Size and width (points, lines)
- Stipple pattern (lines, triangles)
- Triangle mode
  - Display as filled: solid color or stipple pattern
  - Display edges
  - Display vertices

Only a few (glPointSize) as built-in OpenGL, rest by app programs and passed to shaders
RGB color and indexed color

- Each color component stored separately in frame buffer
- Usually 8 bits per component in buffer (bit-planes)
- Color values range from 0.0 (none) to 1.0 (all) as float or from 0 to 255 as unsigned bytes
- **Indexed colors:** Colors are indices into tables of RGBs
- Requires less memory 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
  - Handled in shader
Setting colors

- Colors are ultimately set in fragment shader but can be determined in either shader or app.
- Application colour: pass to vertex shader as a variable or as a vertex attribute.
- Vertex shader color: pass to fragment shader as a variable.
- Fragment color: can alter via shader code.
Input and interactions

- Basic paradigm for interactive computer graphics
- Physical vs. logical view of input devices
- Event-driven input and callbacks
- Double buffering for smooth animation
- Programming interactions with GLUT

Readings: Sections 3.2 – 3.4
Physical input devices

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

Older APIs (GKS, PHIGS, not OpenGL) defined six specific types of logical input

- **Locator**: returns a position, e.g., clicked at by a mouse pointer
- **Choice**: returns one of \( n \) discrete items, e.g., a menu item
- **String**: returns strings of characters, e.g., via key presses
- **Stroke**: returns array of positions
- **Valuator**: as a means to provide analog input, e.g., a slider
- **Pick**: returns ID of an object

OpenGL and GLUT provide functions to handle all these
OpenGL: event-driven

- OpenGL handles interactions via **event-driven** approach

- Each input device can be triggered at any time by user

- Each trigger generates an **event**
  - Event input to the application program is put in an **event queue**
  - Even queue is examined by application program repeatedly

- Examples of event triggers: mouse click, release, key stroke, ....
Some event types

- Window: resize, expose, iconify
- Mouse: click one or more buttons
- Motion: mouse movement
- Keyboard: press or release of a key
- Timer: triggered when the timer has counted down
- Idle: nonevent
  - Define what should be done if no other event is in the event queue, e.g., useful for generating animation
Callback functions

- Event-driven paradigm is implemented using **callbacks**
- Define a callback function for each type of event the graphics system recognizes
- User-supplied function: executed when the event occurs
- Event ignored if no callback defined
- **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

- Last line in `main.c` is typically

  ```c
  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
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 redispays

- 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
  
glutIdleFunc(myIdle)

- Useful for **animations**, e.g.,

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

void myDisplay() {
    glClear(…);
    /* 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.
- Timing for screen refresh (one cycle of an event loop) may not synchronize with drawing of display in the buffer.
- Complex displays take longer than one refresh cycle to draw:
  - Partially drawn screen may be cleared by refresh.
  - 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(GL_RGB | GL_DOUBLE)`
  - At the end of the display callback buffers are **swapped**

```c
void mydisplay()
{
    glClear(GL_COLOR_BUFFER_BIT|....)
    /* draw graphics here */
    glutSwapBuffers()
}
```
Using global variables

- The form of all GLUT callbacks is fixed, e.g.,

  - void mydisplay()
  
  - void mymoused(GLint button, GLint state, GLint x, GLint y)

- Must use global variables to pass other necessary information to callbacks

  float t; /*global */
  
  void mydisplay()
  {
    /* draw something that depends on t */
  
  

January 29, 2018
Mouse callback

glutMouseFunc(mymouse)

```c
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`
  - Can also use query functions
    - `glGetIntegerv(state_name, variable)`
    - `glGetFloatv(state_name, variable)`
  
  to obtain any value that is part of the OpenGL state
Using the mouse position

- Example, 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

```c
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 */
    quad[0] = points2(x+size, y+size);
    quad[1] = points2(x-size, y+size);
    quad[2] = points2(x+size, y-size);
    quad[3] = points2(x-size, y-size);
    glBufferSubData( GL_ARRAY_BUFFER, 0, sizeof(quad), quad );
    glutPostRedisplay();
}
```
Keyboard events

- When mouse pointer is in the window, key press and release generate events
  
  ```c
  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(myreshape)`

`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, even before the window’s display callback
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)
{
    mat4 p;
    glViewport(0, 0, w, h);
    if (w <= h)
        p = Ortho2D(-1.0, 1.0, -1.0 * (GLfloat) h / (GLfloat) w,
                     1.0 * (GLfloat) h / (GLfloat) w);
    else
        p = Ortho2D(-1.0 * (GLfloat) w / (GLfloat) h,
                     1.0 * (GLfloat) w / (GLfloat) h, -1.0, 1.0);
    glUniformMatrix4fv( projection, 1, GL_TRUE, p );
}
```
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_id = glutCreateMenu(mymenu);
glutAddmenuEntry("clear Screen", 1);
gluAddMenuEntry("exit", 2);
glutAttachMenu(GLUT_RIGHT_BUTTON);
```

Menu identifier; can be used to set current menu

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

- **Menu callback, registered by the call to glutCreateMenu()**

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

- **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()**
Test example (see Appendix A)

- 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
The Sierpinski gasket