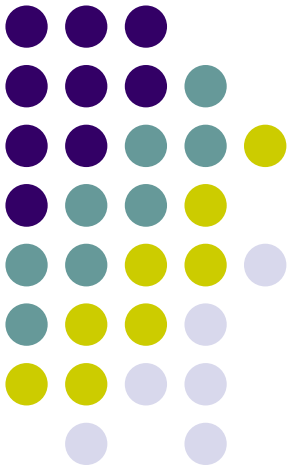
# Computer Graphics (CS 4731) Lecture 2: Introduction to OpenGL/GLUT (Part 1)

Prof Emmanuel Agu

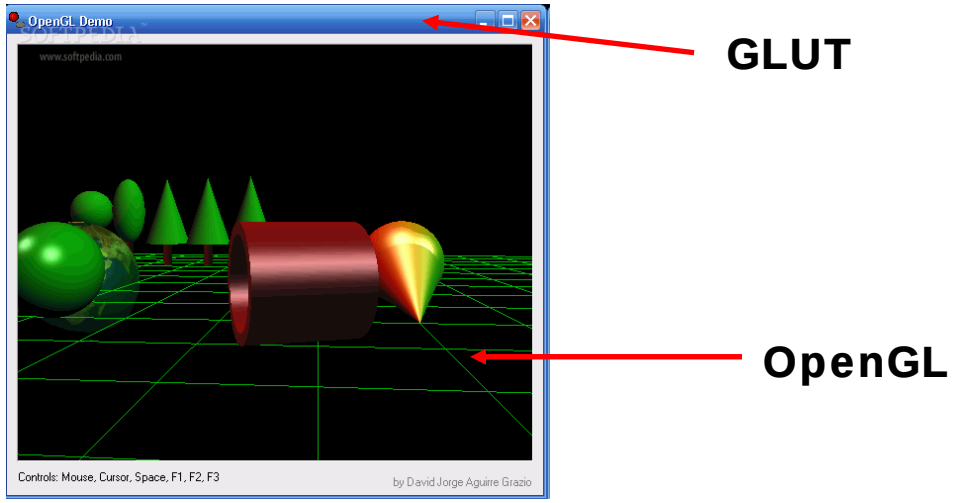
*Computer Science Dept.*

*Worcester Polytechnic Institute (WPI)*

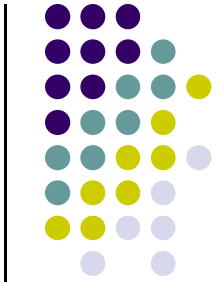


# Recall: OpenGL/GLUT Basics

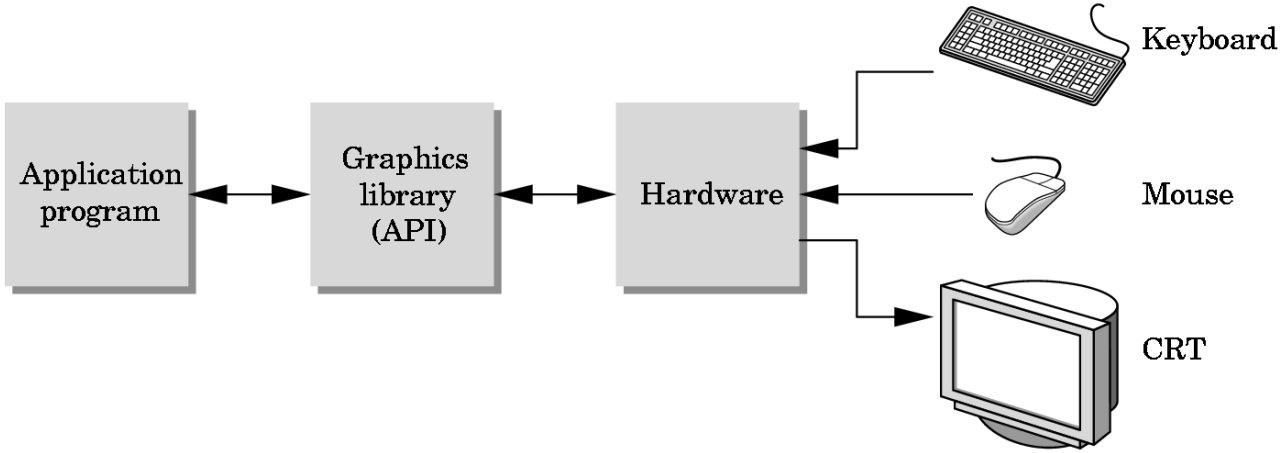
- OpenGL's function – Rendering (or drawing)
- OpenGL can render: 2D, 3D or images
- OpenGL does not manage drawing window
- Portable code!
- GLUT: Does minimal window management



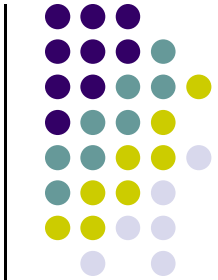
# Recall: OpenGL Programming Interface



- Programmer view of OpenGL?
  - Writes OpenGL Application programs
  - Uses OpenGL Application Programmer Interface (API)

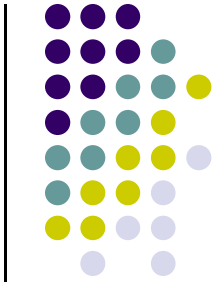


# Sequential Vs Event-driven

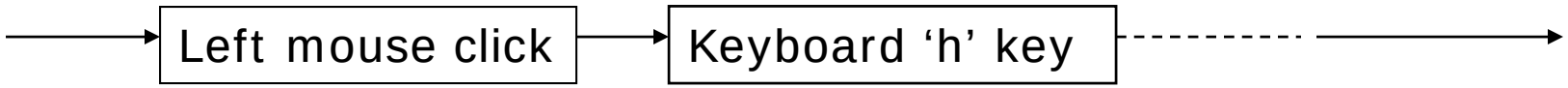


- OpenGL programs are event-driven
- Sequential program
  - **Start at main( )**
  - **Perform actions 1, 2, 3…. *N***
  - **End**
- Event-driven program
  - **Start at main( )**
  - **Initialize**
  - **Wait in infinite loop**
    - **Wait till defined event occurs**
    - **Event occurs => Take defined actions**
- What is World's most famous event-driven program?

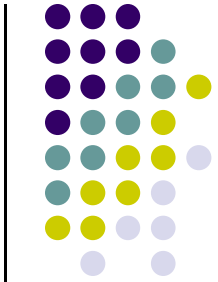
# OpenGL: Event-driven



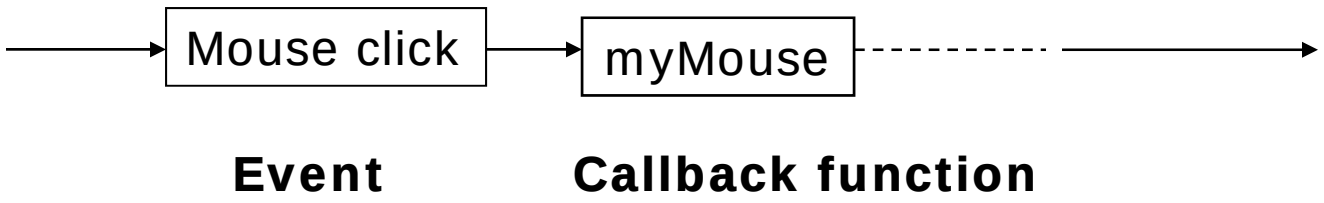
- Program only responds to events
- Do nothing until event occurs
- Example Events:
  - **mouse clicks,**
  - **keyboard stroke**
  - **window resize**
- Programmer:
  - defines events
  - Defines actions to be taken
- System:
  - maintains event queue
  - takes programmer-defined actions



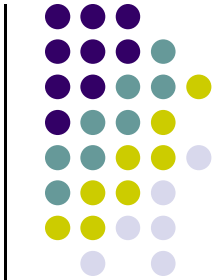
# OpenGL: Event-driven



- How in OpenGL?
  - Programmer registers callback functions (event handler)
  - Callback function called when event occurs
- Example: Programmer
  1. Declare function *myMouse*, called on mouse click
  2. Register it: glutMouseFunc(*myMouse*);
- OS receives mouse click, calls callback function **myMouse**

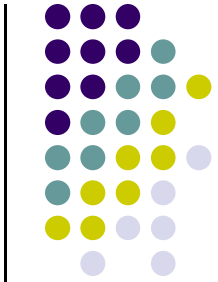


# glInfo: Finding out about your Graphics Card



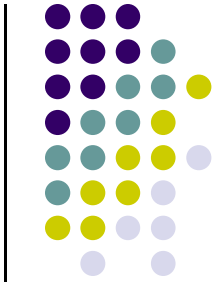
- Gives OpenGL version and extensions your graphics card supports
- Homework 0!

# Some OpenGL History



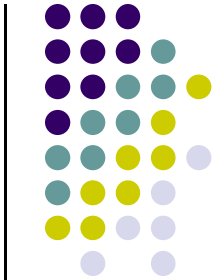
- OpenGL either on graphics card or in software (e.g. Mesa)
- Each graphics card supports specific OpenGL version
- OpenGL previously fixed function pipeline (up to version 1.x)
  - Pre-defined functions to generate picture
  - Programmer could not change steps, algorithms. Restrictive!!
- Shaders
  - allow programmer to write/load some OpenGL functions
  - proposed as *extensions* to version 1.4
  - part of core in OpenGL version 2.0 till date (ver 4.2)
- For this class you need: OpenGL version 3.3 or later

# Other OpenGL Versions

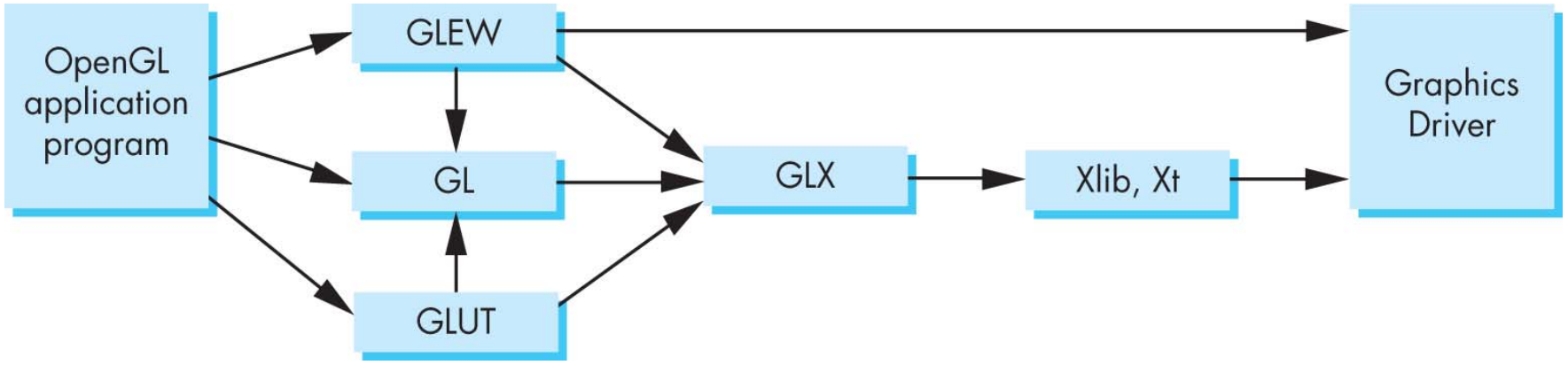


- OpenGL 4.1 and 4.2
  - Adds geometry shaders
- OpenGL ES: Mobile Devices
  - Version 2.0 shader based
- WebGL
  - Javascript implementation of ES 2.0
  - Supported on newer browsers

# GLEW

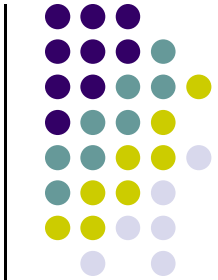


- OpenGL Extension Wrangler Library
- Makes it easy to access OpenGL extensions available on a particular system
- More on this later

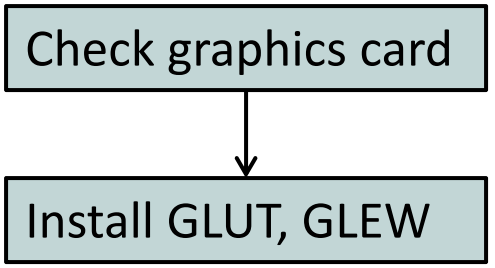



OpenGL/GLEW architecture on X Windows

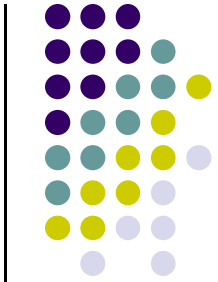
# Windows Installation of GLUT, GLEW



- Install Visual Studio (e.g 2010)
- Download freeglut 32-bit (GLUT implementation)
  - http://freeglut.sourceforge.net/
- Download GLEW
  - http://glew.sourceforge.net/
- Unzip => .lib, .h, .dll files
- Install
  - Put .dll files (for GLUT and GLEW) in C:\windows\system
  - Put .h files in Visual Studio…\include\ directory
  - Put .lib files in Visual Studio….\lib\ directory
  - **Note:** Use include, lib directories of highest VS version



# OpenGL Program?



- Usually has 3 files:
  - **Main .cpp file:** containing your main function
    - Does initialization, generates/loads geometry to be drawn
  - Two shader files:
    - **Vertex shader:** functions to manipulate (e.g. move) vertices
    - **Fragment shader:** functions to manipulate (e.g change color of) fragments/pixels

**.cpp file**

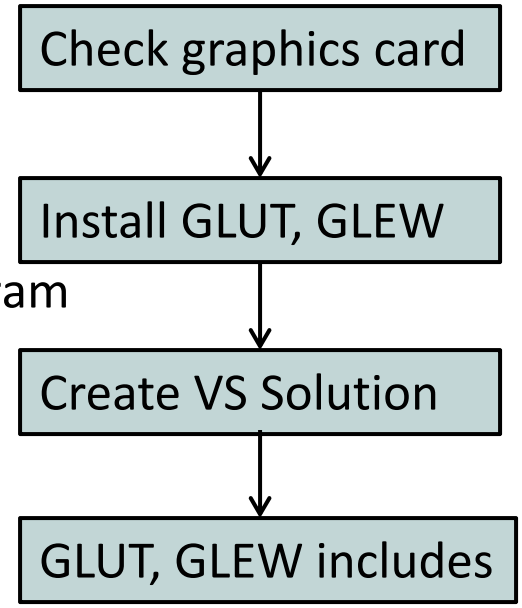
**Vertex shader**

**fragment shader**

**Next:** look at .cpp file

# Getting Started: Set up Visual studio Solution

1. Create empty project
2. Create blank console application (C program)
3. Add console application to project
4. Include **`glew.h`** and **`glut.h`** at top of your program

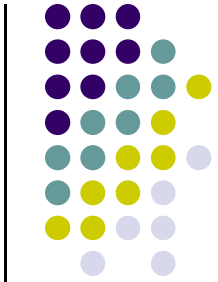



```
#include <glew.h>
#include <GL/glut.h>
```

**Note:** **`GL/`** is sub-directory of compiler **`include/`** directory

- **`glut.h`** contains GLUT functions, also includes **`gl.h`**
- OpenGL drawing functions in **`gl.h`**

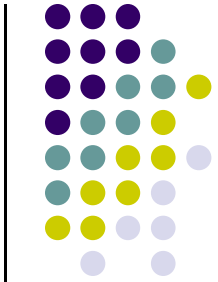
# Getting Started: More #includes



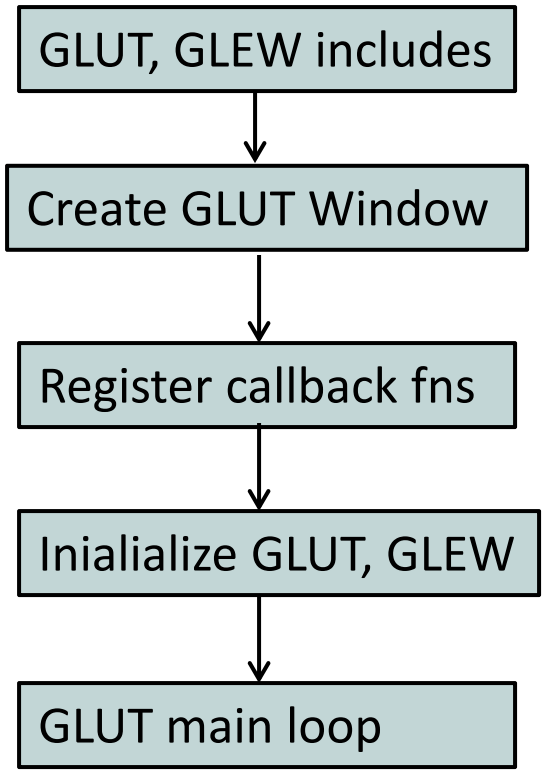
- Most OpenGL applications use standard C library (e.g for **`printf`**), so

```
#include <stdlib.h>
#include <stdio.h>
```

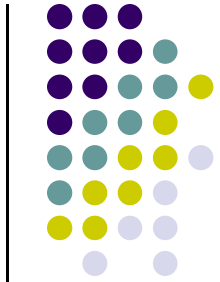
# OpenGL/GLUT Program Structure



- Configure and open window (GLUT)
  - Configure Display mode, Window position, window size
- Register input callback functions (GLUT)
  - Render, resize, input: keyboard, mouse, etc
- My initialization
  - Set background color, clear color, etc
  - Generate points to be drawn
  - Initialize shader stuff
- Initialize GLEW
- Register GLUT callbacks
- glutMainLoop( )
  - Waits here infinitely till event

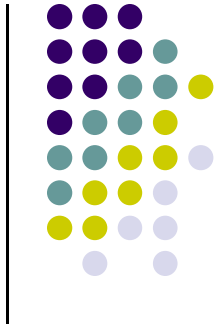


# GLUT: Opening a window

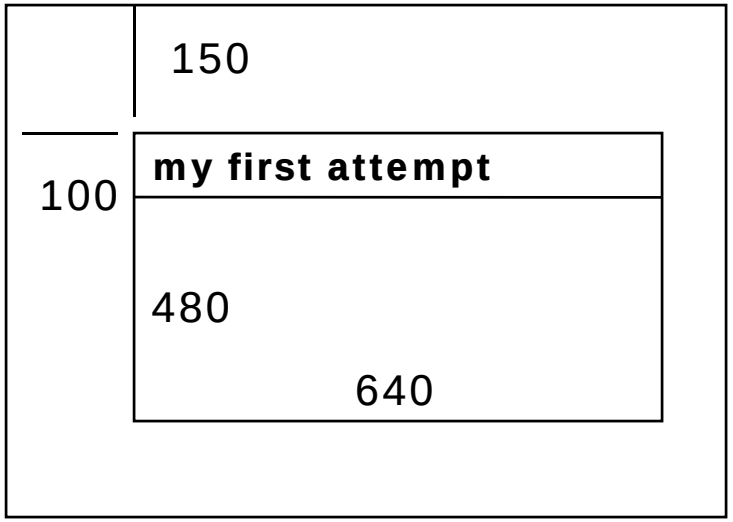


- GLUT used to create and open window
  - **`glutInit(&argc, argv);`**
    - Initializes GLUT
  - **`glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);`**
    - sets display mode (e.g. single buffer with RGB colors)
  - **`glutInitWindowSize(640,480);`**
    - sets window size (Width x Height) in pixels
  - **`glutInitPosition(100,150);`**
    - sets location of upper left corner of window
  - **`glutCreateWindow("my first attempt");`**
    - open window with title "my first attempt"
- Then also initialize GLEW
  - **`glewInit( );`**

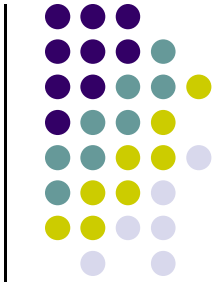
# OpenGL Skeleton



```
void main(int argc, char** argv){
    // First initialize toolkit, set display mode and create window

    glutInit(&argc, argv);   // initialize toolkit
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(640, 480);
    glutInitWindowPosition(100, 150);
    glutCreateWindow("my first attempt");
    glewInit( );

    // … then register callback functions,
    // … do my initialization
    // .. wait in glutMainLoop for events

}
```

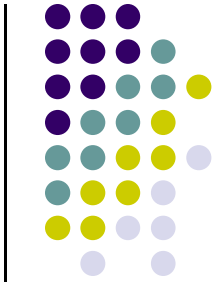


# GLUT Callback Functions

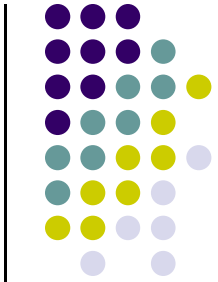


- Register all events your program will react to
- Callback: a function system calls when event occurs
- Event occurs  => system callback
- No registered callback = no action
- Example: if no registered keyboard callback function, hitting keyboard keys generates NO RESPONSE!!

# GLUT Callback Functions



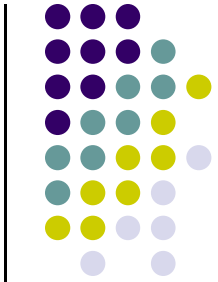
- GLUT Callback functions in skeleton
  - **`glutDisplayFunc(myDisplay);`** Image to be drawn initially
  - **`glutReshapeFunc(myReshape);`** called when window is reshaped
  - **`glutMouseFunc(myMouse);`** called when mouse button is pressed
  - **`glutKeyboardFunc(mykeyboard);`** called when keyboard is pressed or released
- **`glutMainLoop( );`**
  - program draws initial picture (by calling myDisplay function once)
  - Enters infinite loop till event



# OpenGL Skeleton

```
void main(int argc, char** argv){
    // First initialize toolkit, set display mode and create window
    glutInit(&argc, argv);   // initialize toolkit
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(640, 480);
    glutInitWindowPosition(100, 150);
    glutCreateWindow("my first attempt");
    glewInit( );

    // … now register callback functions
    glutDisplayFunc(myDisplay);
    glutReshapeFunc(myReshape);
    glutMouseFunc(myMouse);
    glutKeyboardFunc(myKeyboard);

    myInit( );
    glutMainLoop( );
}
```

# Example of Rendering Callback



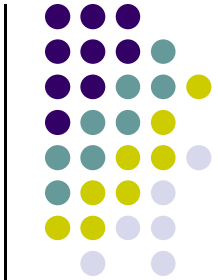
- Do all drawing code in display function
- Called once initially and when picture changes (e.g.resize)
- First, register callback in main( ) function

```
glutDisplayFunc( myDisplay );
```

- Then, implement display function

```
void myDisplay( void )
{
    // put drawing commands here

}
```

# Old way: Drawing Example



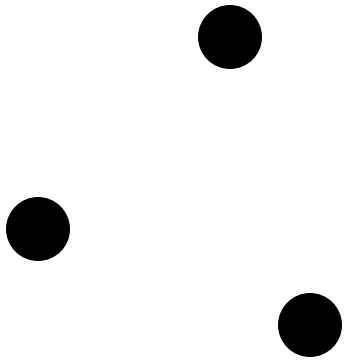
- Example: draw three dots. How?
  - Specify vertices between **glBegin** and **glEnd**
- Immediate mode
  - Generate points, render them (points not stored)
  - Compile scene with OpenGL program

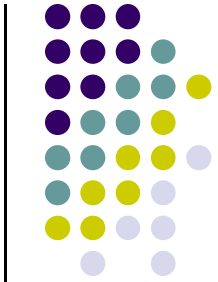
```
void myDisplay( void )
{
   ....

   glBegin(GL_POINTS)
        glVertex2i(100,50);
        glVertex2i(100,130);
        glVertex2i(150, 130);
       glFlush( );
   glEnd( )
```

Also GL_LINES, GL_POLYGON….

*Forces drawing to complete*

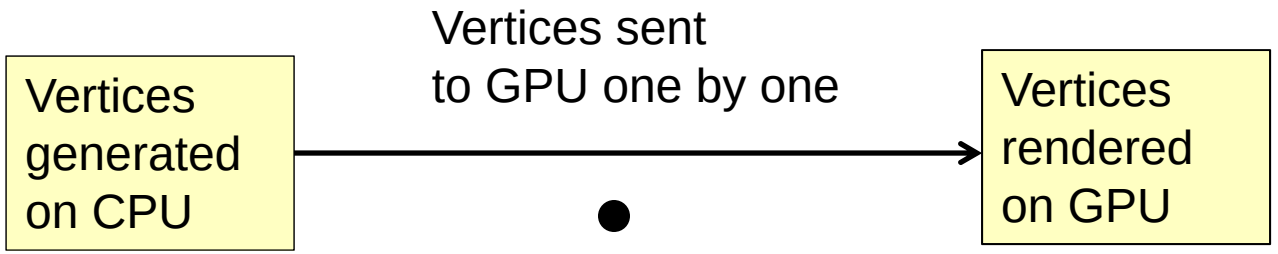
*x* *y*



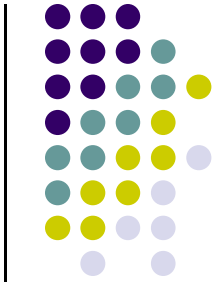


# Immediate Mode Graphics

- Geometry specified as sequence of vertices in application
- Immediate mode
  - OpenGL application receives input on CPU, moved to GPU, render!
  - Each time a vertex is specified in application, its location is sent to GPU
  - Creates bottleneck between CPU and GPU
  - Removed from OpenGL 3.1



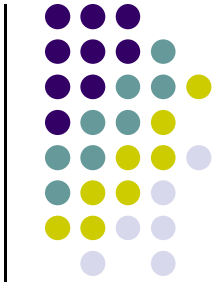
## New: Better Way of Drawing: Retained Mode Graphics



- **Retained mode:** generate all vertices in drawing, store in array, then move array of all points to GPU for drawing
- **Rendering steps:**

1. Generate points
2. Store all vertices into an array
3. Create GPU buffer for vetices
4. Move vertices from CPU to GPU buffer
5. Draw points from array on GPU using **`glDrawArray`**

## Better Way of Drawing: Retained Mode Graphics



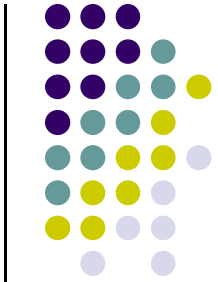
- Useful to declare types *point2* for <x,y> locations*, vec3* for <x,y,z> vector coordinates with their constructors
- put declarations in *header file vec.h*

```
#include “vec.h”

vec3 vector1;
```

- Can also do typedefs

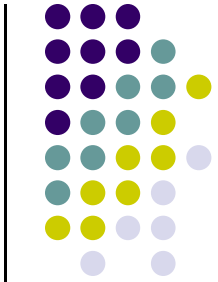
```
typedef vec2 point2;
```

# 1. Generate Points to be Drawn
# 2. Store in an array



- Generate points & store vertices into an array

```
point2 points[NumPoints];

points[0] = point2( -0.5, -0.5 );
points[1] = point2( 0.0, 0.5 );
points[2] = point2( 0.5, -0.5 );
```



# 3. Create GPU Buffer for Vertices

- Rendering from GPU memory significantly faster. Move data there
- Fast GPU (off-screen) memory for data called ***Buffer Objects***
- An array of buffer objects (called ***vertex array object***) are usually created
- So, first generate an array of names of vertex array objects

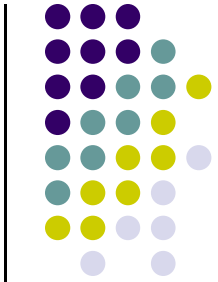
```
GLuint vao;
glGenVertexArrays( 1, &vao );
```

**Number of Buffer Object names to generate**

**Array in which vertex buffer names are stored**

- Then bind the vertex array object

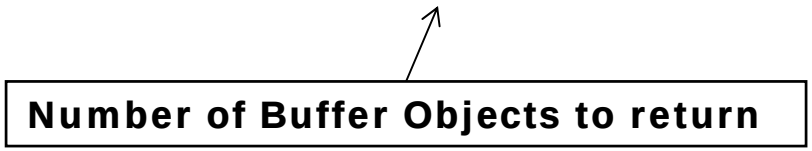
```
glBindVertexArray( vao );
```



# 3. Create GPU Buffer for Vertices

- Next, create a buffer object in two steps
  1. Create VBO and give it name (unique ID number)

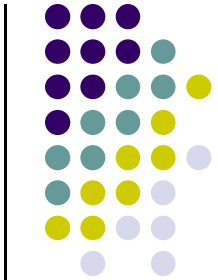
```
GLuint buffer;
glGenBuffers(1,  &buffer);    // create one buffer object
```




  2. Make created VBO currently active one

```
glBindBuffer(GL_ARRAY_BUFFER, buffer);    //data is array
```

# 4. Move points GPU memory

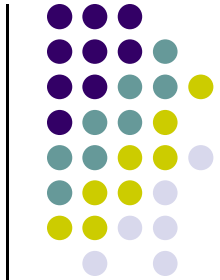


3. Move **`points`** generated earlier to VBO

```
glBufferData(GL_ARRAY_BUFFER, buffer, sizeof(points),
points, GL_STATIC_DRAW ); //data is array
```

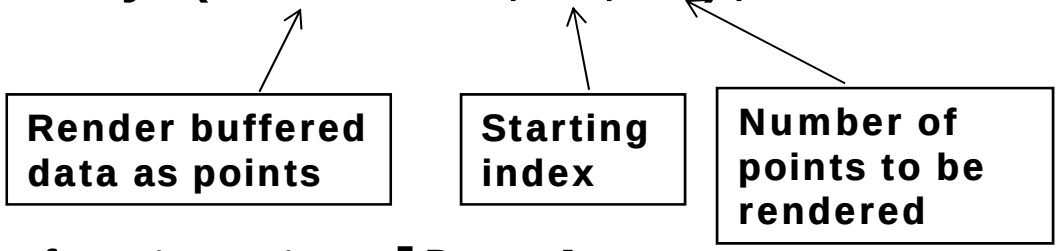
**Data to be transferred to GPU memory (generated earlier)**

- **`GL_STATIC_DRAW:`** buffer object data will be specified once by application and used many times to draw
- **`GL_DYNAMIC_DRAW:`** buffer object data will be specified repeatedly and used many times to draw



# 5. Draw points (from VBO)

```
glDrawArrays(GL_POINTS, 0, N);
```

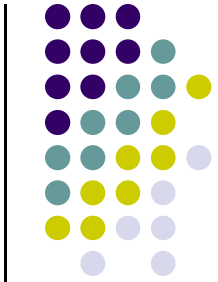



- Display function using **`glDrawArrays:`**

```
void mydisplay(void){
      glClear(GL_COLOR_BUFFER_BIT); // clear screen
      glDrawArrays(GL_POINTS, 0, N);
      glFlush( );        // force rendering to show
}
```

- Other possible arguments to **`glDrawArrays`** instead of **`GL_POINTS?`**

# References



- Angel and Shreiner, Interactive Computer Graphics, 6th edition, Chapter 2
- Hill and Kelley, Computer Graphics using OpenGL, 3rd edition, Chapter 2