Outcomes

Today you will learn the following

1. Use of vertex arrays to store vertex data in a single location and improve access efficiency
2. Use of Vertex Buffer Objects to store vertex data on the server side to improve transfer time
3. Use of Vertex Array Objects to encapsulate calls defining an object
4. Use of Display Lists to store frequently-made code
5. How to display text
6. Use the mouse in OpenGL
7. Programming Non-ASCII keys

Vertex Arrays (retained-mode rendering)

- Provide easy and efficient means to centralize and share data
- Using Vertex Arrays - A 3-step process

① Enable Vertex and Color arrays - in initialization

glEnableClientState(array) - where array is, successively, GL_VERTEX_ARRAY and GL_COLOR_ARRAY, for vertex coordinate and color values

② Specify Vertex data - in initialization

glVertexPointer(size, type, stride, *pointer)

glColorPointer(size, type, stride, *pointer)

where:
- pointer - address of start of data array
- type - declares data type
- size - number of values per vertex
- stride - offset between start of values for successive vertices

③ Draw!

glDrawElements(primitive, countIndices, type, *indices)

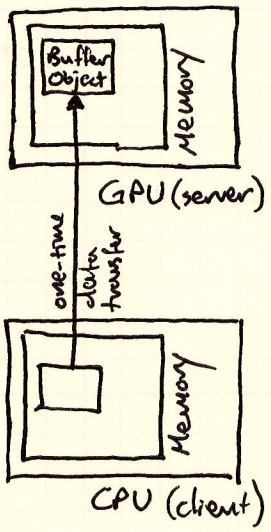
- primitive - geometric primitive
- indices - start address of an array of indices
- type - data type
- countIndices - number of indices to use

- We can also interleave color and vertex data into a single array
  - Still requires use of glVertexPointer and glColorPointer but stride offset should take into account offset between elements
  - Use glDrawArrays(primitive, first, countVertices)
- Use glMultiDrawElements(primitive, *countIndices, type, **indices, countPrimitives) to handle drawing multiple elements
  - more efficient than multiple calls to glDrawElements()
  - **indices → array of arrays' indices
  - countPrimitives → number of primitives to draw
- Similarly, use glMultiDrawArrays(primitive, *first, *countVertices, countPrimitives) which is analogous to glDrawArrays(...)

- Stop using glBegin() -- glEnd() which is for immediate-mode rendering

Vertex Buffer Objects (VBO)

- OpenGL Client-server Model



- Each time the server requires vertex data (coordinates, color, or to execute a glDrawElements()) it must be fetched from the client
- Thus the data must move across the bus from CPU to GPU (which is slower than direct memory access)
- If the data was previously sent this transfer may be redundant
- Buffer Objects allow us to specifically send a set of data to the server to be stored for future use.

- Using VBOs

① Get VBO Buffer ids

```
glGenBuffers(n, buffer)
```

② Reserve space for VBO

```
glBufferData(target, size, *data, usage)
```

Note: we can also use the following to update: glMapBuffer(...) with access being GL_READ_ONLY, GL_WRITE_ONLY, or GL_READ_WRITE we then update the buffer directly and call glUnmapBuffer(...) to release the VBO. The data is then drawn with glDrawElements(...)

③ Update VBO with vertex coordinate and color data

```
glBufferSubData(target, offset, size, *data)
```

~~offset* subData = (float*) glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY);~~

④ Activate VBO

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffer[INDICES]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(shapeIndices),
             shapeIndices, GL_STATIC_DRAW);
```

⑤ Activate Vertex Arrays

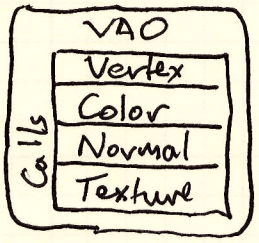
```
glEnableClientState(GL_VERTEX_ARRAY)
glEnableClientState(GL_COLOR_ARRAY)
```

⑥ Specify vertex pointers using offset of current VBO, rather than memory location

```
glVertexPointer(...)
glColorPointer(...)
```

Vertex Array Objects (VAO)

- A busy scene with many vertex array based objects will likely require switching multiple times between sets of arrays and buffers.
  - This will lead to numerous calls to glBindBuffer() and glVertexPointer()
- In OpenGL 3.0, vertex array objects were introduced to alleviate this
- VAOs act as a container for all the calls specifying one or more vertex arrays.
  - Thus one need then to only activate a VAO prior to drawing the object



- Using VAOs

① Initialize (setup())

glGenVertexArrays(n, vao) - returns n vertex array ids in array vao

② Create commands for each vertex array object (setup())

```
glBindVertexArray(vao[index])
glGenBuffers(n, buffer)
...
glVertexPointer(3, GL_FLOAT, 0, 0)
glColorPointer(3, GL_FLOAT, 0, (void *) (sizeof(vertices)))
```

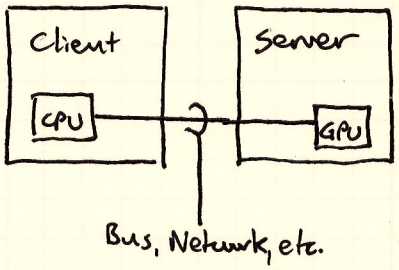
③ Draw! (drawScene())

```
glBindVertexArray(vao[index])
glDrawElements(GL_TRIANGLE_STRIP, 10, GL_UNSIGNED_INT, 0)
```

Note: VBOs and VAOs are necessary when working with shaders

Display Lists

- Used to define and cache a set of commands to be flushed repeatedly
- stored on the machine which runs the display unit (called the server)
  - often in a pre-compiled and optimized form
  - When needed, the display list is called to invoke the contained commands
- Invoked by the machine ~~~~ running the program → (called the client)



- Display Lists also provide a convenient and logical way to encapsulate 3D objects
  - Think Classes!

- Using Display Lists

- There are 3 steps to use a display list:

① Create the display list

```
list = glGenLists(1); //glGenLists returns a block of size range of available
                      // display list indices where range is the parameter
```

~~② Specify the list and commands~~

② Specify the list and commands

```
glNewList(list, GL_COMPILE);  //glNewList(listName, mode) and glEndList()
  *commands                   // contains the commands. The list name param
glEndList();                  // is from a created display list and mode
                              // param can be GL_COMPILE (used to store)
                              // and GL_COMPILE_AND_EXECUTE (used to
                              // store and run)
```

③ Call the list

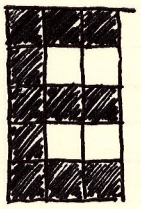
```
glCallList(list); //executes the list provided as a parameter
```

- Calling Multiple Lists

- You can call multiple lists using glCallLists(n, type, *offsets) which causes n display list executions. Indices are obtained by adding offsets of type type to the current display list base obtained by glListBase(base)

Drawing Text

- Two types of text can be created
  - bitmapped - text defined as a pattern of on and off bits in a rectangular block



  - Stroke - characters created using line primitives

- FreeGLUT provides both types.
  - glutBitmapCharacter(*font, character)
  - glutStrokeCharacter(*font, character)

- Stroke fonts have the following advantages
  - can scale in size
  - can be rotated

- Bitmapped fonts have the following disadvantages
  - always aligned with the axes
  - cannot change in size

- Canonical string writing routines:

```
void writeBitmapString(void *font, char *string){
   char *c;
   for(c=string; *c != '\0'; c++)
      glutBitmapCharacter(font, *c);
}
```

```
void writeStrokeString(void *font, char *string){
   char *c;
   for(c=string; *c != '\0'; c++)
      glutStrokeCharacter(font, *c);
}
```

- Called With
  - glRasterPos3f(p, q, r); - positions string at (p, q, r) for start
  - writeBitmapString(font, string) - writes string
  - coordinates can be changed with glTranslatef() and glRotatef()
  - Thickness of stroke characters can be adjusted with glLineWidth()

Bitmap Fonts Available

GLUT_BITMAP_8_BY_13
GLUT_BITMAP_9_BY_15
GLUT_BITMAP_TIMES_ROMAN_10
GLUT_BITMAP_TIMES_ROMAN_24
GLUT_BITMAP_HELVETICA_10
GLUT_BITMAP_HELVETICA_12
GLUT_BITMAP_HELVETICA_18

Stroke Fonts Available

GLUT_STROKE_ROMAN
GLUT_STROKE_MONO_ROMAN

Programming the Mouse

- The mouse can respond to three types of events:
  - button clicks
  - mouse motion
  - wheel turning
- Handling Events: Clicks

```
void mouseControl(int button, int state, int x, int y)
{
    if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
        points.push_back(Point(x, height - y, pointSize));
    if (button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
        exit(0);
    glutPostRedisplay();
}
```

register this callback using
- glutMouseFunc(...);

in main()

  - Button = can be any of the following: GLUT_LEFT_BUTTON, GLUT_RIGHT_BUTTON, GLUT_MIDDLE_BUTTON
  - State = can be any of the following: GLUT_UP, GLUT_DOWN
  - x, y = location in the OpenGL window where the event occurred
- Handling Events: Motion

```
void mouseMotion(int x, int y)
{
    currentPoint.setCoords(x, height-y);
    glutPostRedisplay();
}
```

register this callback using
- glutMotionFunc(...) → if button pressed
- glutPassiveMotionFunc(...) → otherwise

in main()

- Handling Events: Wheel

```
void mouseWheel(int wheel, int direction, int x, int y)
{
    (direction > 0) ? tempSize++ : tempSize--;
    points.back().setSize(tempSize);
    glutPostRedisplay();
}
```

register this callback using
- glutMouseWheelFunc(...)

in main()

  - Wheel Number: 0 if a single wheel
  - Direction: +1 or -1
  - x, y: location of mouse in screen coordinates

Programming Non-ASCII Keys

- Previously, we showed that we can process ASCII keyboard input by assigning a keyboard handling function in setup() using

```
glutKeyboardFunc(keyboard_handling_func);
```

- But, as noted, this only works for ASCII keys.
- For non-ASCII keys (such as the arrow keys) we need to register a special keyboard handling callback in setup() using

```
glutSpecialFunc(special_key_handling_func)
```

For Next Time

1. Review prior lectures
2. Read Ch 3 sect 8-13
3. When Assignment 01 is released begin working on it
4. Come to class!

Additional Notes