Computer Graphics Programming I

- Agenda:
  - Quiz #4
  - Faster geometry:
    - Vertex arrays
    - Vertex buffer objects
  - Work on term project

Why vertex arrays?

- Immediate mode is slow
  - Using a function call per data item carries significant overhead
  - Flexibility of interface make it more difficult for driver to optimize
- Immediate mode is cumbersome
  - Model data is typically stored as arrays of positions, normals, etc.
  - Application developers have to write code to convert array data to repeated function calls

Vertex Array Overview

- Three step process:
  - Provide pointer *client memory* containing data
    - Must also describe the layout of the data
    - Analogous to `glTexImage2D`
  - Enable arrays that will be used
  - Specify which array data to use to draw each primitive

Providing Array Data

- Each data element that can be specified between begin / end has an array
  - Examples:

    `glVertex` → `glVertexPointer`

    `glColor` → `glColorPointer`

    `glNormal` → `glNormalPointer`

    `glTexCoord` → `glTexCoordPointer`

    `glFogCoord` → `glFogCoordPointer`
- **No** array entry-point for glMultiTexCoord
  - Use `glActiveTexture` and `glTexCoordPointer`

Providing Array Data (cont.)

- Each function provides same data to GL:
  - Number of data components
    - Most data will have 2, 3, or 4 components
    - May be implicit → normals always have 3 components
  - Type of data
  - Array stride
    - Number of bytes from one element to the next
    - Specifying zero implies that the data is packed
  - Pointer to the array

Array Stride

- ➲ Consider this data:

```
const GLfloat my_data[] = {
    /* position    normal */
     1.0,  1.0, 1.0, 0.0, 0.0, 1.0,
     1.0, -1.0, 1.0, 0.0, 0.0, 1.0,
    -1.0, -1.0, 1.0, 0.0, 0.0, 1.0,
    -1.0,  1.0, 1.0, 0.0, 0.0, 1.0,
    ...
};
```

- ➲ From one normal to the next there are 6 floats
  - The stride is 6 `* sizeof(GLfloat)`

Array Stride (cont.)

- Data need not be homogeneous:

```
struct data {
    GLfloat position[4];
    GLfloat normal[3];
    GLubyte color[3];
};
```

- Here stride is just `sizeof(struct data)`
  - This is useful for loading data directly from disk (or network) into a buffer

Example

```
struct data {
    GLfloat position[4];
    GLfloat normal[3];
    GLubyte color[3];
};

struct data *model;

void setup_arrays(void)
{
    glVertexPointer(4, GL_FLOAT, sizeof(struct data),
                    & model->position);
    glNormalPointer(GL_FLOAT, sizeof(struct data),
                    & model->normal);
    glColorPointer(3, GL_UNSIGNED_BYTE,
                   sizeof(struct data), & model->color);
}
```

Enabling Arrays

- Each array that will be used must be enabled
  - Arrays are in client memory, and the enables are client state
  - Use `glEnableClientState` instead of `glEnable`
- Each array has a name
  - `GL_VERTEX_ARRAY`, `GL_COLOR_ARRAY`, `GL_NORMAL_ARRAY`, etc.

Drawing with a Vertex Array

- There are 3 common ways to draw:
  - Blocks of vertices in order
  - Arbitrary vertices, one at a time
  - Arbitrary vertices, en masse

glDrawArrays

- Draw a group of primitives using a range of vertices in order

```
glDrawArrays(GLenum mode,
    GLint first_element, GLsizei count);
```

- Directly from the manual page:

> "...uses `count` sequential elements from each enabled array to construct a sequence of geometric primitives, beginning with element `first`. mode specifies what kind of primitives are constructed, and how the array elements construct those primitives."

glArrayElement

- Specify one array element to use with one call
  - Used like immediate mode functions

```
glArrayElement(GLint i);
```

  - Example:

```
glBegin(GL_TRIANGLES);
while (!done) {
    done = get_next_triangle(indices);
    glArrayElement(indices[0]);
    glArrayElement(indices[1]);
    glArrayElement(indices[2]);
}
glEnd();
```

glDrawElements

- The mostly commonly used drawing function

```
glDrawElements(GLenum mode, GLsizei count,
    GLenum type, const GLvoid *indices);
```

- `indices` points to an array of elements that are used to draw primitives
  - `type` specifies what type of data `indices` is
    - Can be `GL_UNSIGNED_INT`, `GL_UNSIGNED_SHORT`, or `GL_UNSIGNED_BYTE`

glDrawElements (cont.)

```
void fake_glDrawElements(GLenum mode, GLsizei count,
                         GLenum type, const GLvoid *indices)
{
    glBegin(mode);
    for (GLsizei i = 0; i < count; i++) {
        switch(type) {
        case GL_UNSIGNED_BYTE:
            glArrayElement(((GLubyte *)indices)[i]);
            break;
        case GL_UNSIGNED_SHORT:
            glArrayElement(((GLshort *)indices)[i]);
            break;
        case GL_UNSIGNED_INT:
            glArrayElement(((GLuint *)indices)[i]);
            break;
        }
    }
    glEnd();
}
```

glMultiDrawArrays

- Specify multiple `glDrawArrays`-like draw calls with a single call:

```
glMultiDrawArrays(GLenum mode,
    GLint *first, GLsizei *count,
    GLsizei primcount);
```

- `primcount` specifies the number of values pointed to by `first` and `count`.

glMultiDrawElements

- Specify multiple `glDrawElementss`-like draw calls with a single call:

```
glMultiDrawElements(GLenum mode,
    const GLsizei *count, GLenum type,
    const GLvoid **indices,
    GLsizei primcount);
```

- `primcount` specifies the number of values pointed to by `count` and `indices`.

References

http://www.opengl.org/sdk/docs/man/xhtml/glVertexPointer.xml

http://www.opengl.org/sdk/docs/man/xhtml/glDrawArrays.xml

http://www.opengl.org/sdk/docs/man/xhtml/glDrawElements.xml

Break

Client memory?

- Unlike textures, vertex arrays are not kept
  - The GL copies the data during the drawing call, uses it, then forgets it
  - Allows easy changing of data between drawing calls
  - Prevents optimizations of static data
    - Data must be re-uploaded to the card on every draw call!

Compiled Vertex Arrays

- Original solution:
  - Application can “lock” a range of data

    ```
    glLockArraysEXT(GLint first, GLsizei count);
    glUnlockArraysEXT(void);
    ```

    - Agreement between application an driver that the application will not modify locked data
    - Allows driver to copy data to card once
    - Driver can also cache transformed data
- Very limited: can only lock one range at a time
  - Want something that works like texture objects

Buffer Objects

- Generic objects that can hold data in *server memory*
  - Similar to textures, but *without* formating semantics
- Data in these objects can be used in place of client memory data
  - Originally intended for vertex data, but can be used for other things as well
- `GL_ARB_vertex_buffer_object` extension
  - Part of core in 1.4

Creating Buffer Objects

- Intentionally very similar to textures

```
void glBindBuffer(GLenum target, GLuint buffer);
void glDeleteBuffers(GLsizei n,
    const GLuint *buffers);
void glGenBuffers(GLsizei n, GLuint *buffers);
GLboolean glIsBuffer(GLuint buffer);
```

- Initially only two targets:
  - `GL_ARRAY_BUFFER` – Data used for vertex arrays
  - `GL_ELEMENT_ARRAY_BUFFER` – Data used for element data
- Bind buffer 0 to disable buffer object for that target

Filling Buffers

- Writes data to the currently bound buffer object
  - Analogous to `glTexImage2D` / `glTexSubImage2D`

```
void glBufferData(GLenum target, GLsizeiptr size,
    const GLvoid *data, GLenum usage);
void glBufferSubData(GLenum target,
    GLintptr offset, GLsizeiptr size,
    const GLvoid *data);
```

  - Like textures, the targets must match

Buffer Usage

- The usage parameter tries to convey the application's intention for the buffer
  - Data “frequency”:
    - Stream – data is specified once and used a few times
    - Static – data is specified ones and used many times
    - Dynamic – data is specified and used many times
  - Data “usage”:
    - Draw – data used as source for drawing
    - Read – data copied from GL and read back to client
    - Copy – data copied from GL and used as source for drawing

GL_STREAM_*

- From the spec:
  - `GL_STREAM_DRAW` – The data store contents will be specified once by the application, and used at most a few times as the source of a GL (drawing) command.
  - `GL_STREAM_READ` – The data store contents will be specified once by reading data from the GL, and queried at most a few times by the application.
  - `GL_STREAM_COPY` – The data store contents will be specified once by reading data from the GL, and used at most a few times as the source of a GL (drawing) command.

*GL_STATIC_**

- From the spec:
  - `GL_STATIC_DRAW` – The data store contents will be specified once by the application, and used many times as the source for GL (drawing) commands.
  - `GL_STATIC_READ` – The data store contents will be specified once by reading data from the GL, and queried many times by the application.
  - `GL_STATIC_COPY` – The data store contents will be specified once by reading data from the GL, and used many times as the source for GL (drawing) commands.

*GL_DYNAMIC_**

- From the spec:
  - `GL_DYNAMIC_DRAW` – The data store contents will be respecified repeatedly by the application, and used many times as the source for GL (drawing) commands.
  - `GL_DYNAMIC_READ` – The data store contents will be respecified repeatedly by reading data from the GL, and queried many times by the application.
  - `GL_DYNAMIC_COPY` – The data store contents will be respecified repeatedly by reading data from the GL, and used many times as the source for GL (drawing) commands.

Using Buffer Object Data

- When a buffer is bound, the pointer parameters various functions have new meanings
  - The `pointer` parameter to `glVertexPointer` and friends becomes an *offset* into the currently bound `GL_ARRAY_BUFFER`.
  - The `indices` parameter to `glDrawElements` and friends becomes an *offset* into the currently bound `GL_ELEMENT_ARRAY_BUFFER`.

Buffer Mapping

- Unlike textures, can get a pointer to the memory of the buffer
  - Functionality exists to make it easier to port vertex arrays to buffer objects
  - Cannot use a mapped buffer for rendering
  - Cannot pass the mapped pointer back into the GL

```
GLvoid *glMapBuffer(GLenum target, GLenum access);
void glUnmapBuffer(GLenum target);
```

- `access` must be one of `GL_READ_ONLY`, `GL_WRITE_ONLY`, or `GL_READ_WRITE`

Buffer Access

- Use the correct access mode!
  - `GL_READ_ONLY` buffers may be mapped in a way that writing will cause the application to crash
  - `GL_WRITE_ONLY` buffers may not be loaded with the contents of the buffer (they may be filled with garbage)
  - `GL_READ_WRITE` buffers may force the driver to copy the buffer from the card and copy the data back on unmap

Buffer Mapping Woes

- Do *not* map a large buffer for writing and only modify a small portion
  - Some drivers implement mapping by copying data off the card into system memory, then copy the system memory back on unmap
    - Radeon drivers work this way
  - Mapping a 16MiB buffer to modify 4 bytes causes 32MiB to be copied (16MiB down and up)
    - Use `glBufferSubData` instead

Buffer Subrange

- Apple has an extension to work around this
  - Before unmapping a buffer, tell the GL what regions were modified.

    ```
    void glFlushMappedBufferRangeAPPLE(GLenum target,
        GLintptr offset, GLsizeiptr size);
    ```

  - `GL_APPLE_flush_buffer_range` extension
    - Supported on all drivers in OS X 10.3 and later
  - Similar functionality will exist in OpenGL 3.0

Next week...

- Discuss final
- Work on term projects

Legal Statement

- This work represents the view of the authors and does not necessarily represent the view of IBM or the Art Institute of Portland.
- OpenGL is a trademark of Silicon Graphics, Inc. in the United States, other countries, or both.
- Khronos and OpenGL ES are trademarks of the Khronos Group.
- Other company, product, and service names may be trademarks or service marks of others.