Curve and surface modeling

– a CAGD approach based on OpenGL and C++ –

Ágoston Róth

Department of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania

(agoston.roth@gmail.com)

Lecture 2 – March 7, 2022

Color styles

- keywords, built-in types, enumerations, constants and namespaces of C++
- keywords, built-in types, enumerations, constants and functions of OpenGL
- our types, constants, enumerations and namespaces
- comments

Color styles

- keywords, built-in types, enumerations, constants and namespaces of C++
- keywords, built-in types, enumerations, constants and functions of OpenGL
- our types, constants, enumerations and namespaces
- comments

Color styles

- keywords, built-in types, enumerations, constants and namespaces of C++
- keywords, built-in types, enumerations, constants and functions of OpenGL
- our types, constants, enumerations and namespaces
- comments

Color styles

- keywords, built-in types, enumerations, constants and namespaces of C++
- keywords, built-in types, enumerations, constants and functions of OpenGL
- our types, constants, enumerations and namespaces
- comments

Color styles

- keywords, built-in types, enumerations, constants and namespaces of C++
- keywords, built-in types, enumerations, constants and functions of OpenGL
- our types, constants, enumerations and namespaces
- comments

OpenGL geometric drawing primitives

Points – GL_POINTS

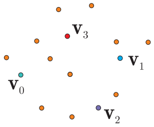

Fig. 1: Individual vertices.

```
glVertex{2|3|4}{s|i|f|d}[v]
```

```
#include <vector>

...

GLfloat x = ..., y = ..., z = ...;

GLfloat v[3] = {..., ..., ...};

class Vertex
{
public:
    GLfloat x, y, z;
    ...
};

GLuint vertex_count = ...;

std::vector<Vertex> p(vertex_count);

...

glBegin(GL_POINTS);

    // 1st possibility
    glVertex3f(x, y, z);

    // 2nd possibility
    glVertex3fv(v);

    // 3rd possibility
    for (std::vector<Vertex>::const_iterator it = p.begin()
         it != p.end(); ++it)
        glVertex3fv(&it->x);

glEnd();
```

OpenGL geometric drawing primitives

Points – GL_POINTS

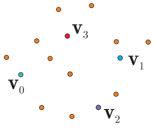

Fig. 1: Individual vertices.

```
glVertex{2|3|4}{s|i|f|d}[v]
```

```
#include <vector>

...

GLfloat x = ..., y = ..., z = ...;

GLfloat v[3] = {..., ..., ...};

class Vertex
{
public:
    GLfloat x, y, z;
    ...
};

GLuint vertex_count = ...;

std::vector<Vertex> p(vertex_count);

...

glBegin(GL_POINTS);

    // 1st possibility
    glVertex3f(x, y, z);

    // 2nd possibility
    glVertex3fv(v);

    // 3rd possibility
    for (std::vector<Vertex>::const_iterator it = p.begin();
         it != p.end(); ++it)
        glVertex3fv(&it->x);

glEnd();
```

OpenGL geometric drawing primitives

Segments – GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP

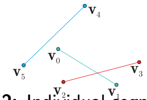

Fig. 2: Individual segments.

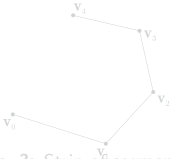

Fig. 3: Strip of segments.

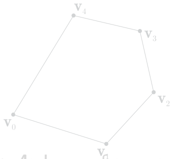

Fig. 4: Loop of segments.

```
#include <cstdlib>
#include <vector>

using std::vector;

class Vertex
{
public:
    GLfloat x, y, z;
    ...
};

...

GLuint segment_count = ...;
GLuint vertex_count = 2 * segment_count;

vector<Vertex> v(vertex_count);

...

for (vector<Vertex>::iterator it = v.begin(); it != v.end(); it++)
{
  it->x = -1.0f + 2.0f * rand() / static_cast<GLfloat>(RAND_MAX);
  it->y = -1.0f + 2.0f * rand() / static_cast<GLfloat>(RAND_MAX);
  it->z = -1.0f + 2.0f * rand() / static_cast<GLfloat>(RAND_MAX);
}

...

glBegin(GL_LINES);
 for (vector<Vertex>::const_iterator it = v.begin(); it != v.end(); it++)
    glVertex3fv(&it->x);
glEnd();
```

OpenGL geometric drawing primitives

Segments – GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP

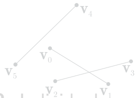

Fig. 2: Individual segments.

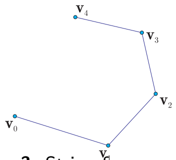

Fig. 3: Strip of segments.

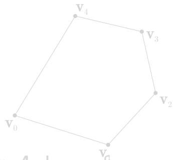

Fig. 4: Loop of segments.

```
#include <cmath>
#include <vector>

using namespace std;

class Vertex
{
public:
    GLfloat x, y, z;
    Vertex(): x(0.0f), y(0.0f), z(0.0f) {}
    ...
};

const GLfloat PI = 3.14159265358979323846264338327950f;

GLuint vertex_count = ...;
vector<Vertex> v(vertex_count);

...

// sine function on the interval [−π, π]
GLfloat du = 2.0 * PI / (vertex_count − 1), u = −PI;
for (vector<Vertex>::iterator it = v.begin(); it != v.end(); it++,u += du)
{
    it->x = u;
    it->y = sin(u);
}

...

glBegin(GL_LINE_STRIP);
  for (vector<Vertex>::const_iterator it = v.begin(); it < v.end(); ++it)
     glVertex2fv(&it->x);
glEnd();
```

OpenGL geometric drawing primitives

Segments – GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP

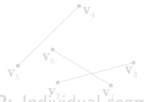

Fig. 2: Individual segments.

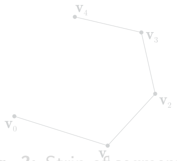

Fig. 3: Strip of segments.

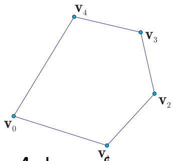

Fig. 4: Loop of segments.

```
#include <cmath>
#include <vector>

using namespace std;

class Vertex
{
public:
    GLfloat x, y, z;
    Vertex(): x(0.0f), y(0.0f), z(0.0f) {}
    ...
};

const GLfloat PI = 3.14159265358979323846264338327950f;

GLuint vertex_count = ...;
vector<Vertex> v(vertex_count);

...

// unit circle
GLfloat du = 2.0 * PI / vertex_count, u = 0.0;
for (vector<Vertex>::iterator it = v.begin(); it != v.end();it++,u += du)
{
    it->x = cos(u);
    it->y = sin(u);
}

...

glBegin(GL_LINE_LOOP);
 for (vector<Vertex>::const_iterator it = v.begin(); it != v.end(); it++)
    glVertex2fv(&it->x);
glEnd();
```

OpenGL geometric drawing primitives

Triangles – GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN

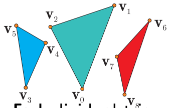

Fig. 5: Individual triangles.

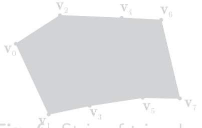

Fig. 6: Strip of triangles.

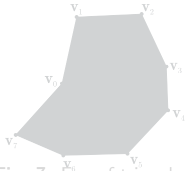

Fig. 7: Fan of triangles.

```
#include <vector>

using namespace std;

class Vertex
{
public:
    GLfloat x, y, z;
    ...
};

class Face
{
public:
    GLint node[3];
    ...
};

...

GLuint vertex_count = ...;
vector<Vertex> v(vertex_count);

GLuint triangle_count = ...;
vector<Face> f(triangle_count);

...

glBegin(GL_TRIANGLES);
  for (vector<Face>::const_iterator it = f.begin(); it != f.end(); it++)
      for (GLuint id = 0; id < 3; id++)
        glVertex3fv(&v[ it->node[id] ].x);
glEnd();
```

OpenGL geometric drawing primitives

Triangles – GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN

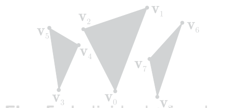

Fig. 5: Individual triangles.

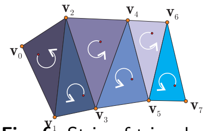

Fig. 6: Strip of triangles.

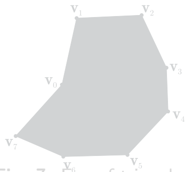

Fig. 7: Fan of triangles.

```
#include <cmath>
#include <vector>
using namespace std;

class Vertex
{
public:
        GLfloat x, y, z;
        Vertex(GLfloat x, GLfloat y, GLfloat z = 0.0f): x(x), y(y), z(z) {}
};

Vertex cylinder(GLfloat r, GLfloat alpha, GLfloat height)
{
        return Vertex(r * cos(alpha), r * sin(alpha), height);
}

const GLfloat PI = 3.14159265358979323846264338327950f;

GLuint slices = ...;
GLfloat r = 1.0f, min_height = −2.0f, max_height = 2.0f,
        dalpha = 2.0f * PI / slices, alpha = 0.0f;

vector<Vertex> v;
for (GLuint i = 0; i < slices; i++, alpha += dalpha)
{
        v.push_back( cylinder(r, alpha, max_height) );
        v.push_back( cylinder(r, alpha, min_height) );
}

...

glBegin(GL_TRIANGLE_STRIP);
 for (vector<Vertex>::const_iterator it = v.begin(); it != v.end(); it++)
        glVertex3fv(&it−>x);
glEnd();
```

OpenGL geometric drawing primitives

Triangles – GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN

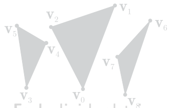

Fig. 5: Individual triangles.

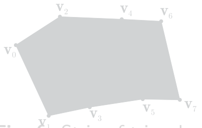

Fig. 6: Strip of triangles.

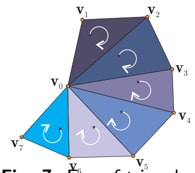

Fig. 7: Fan of triangles.

```
#include <cmath>
#include <vector>
using namespace std;

class Vertex
{
public:
    GLfloat x, y, z;
    Vertex(GLfloat x, GLfloat y, GLfloat z = 0.0f): x(x), y(y), z(z) {}
};

Vertex cone(GLfloat height, GLfloat alpha)
{
    return Vertex(height * cos(alpha), height * sin(alpha), height);
}

const GLfloat PI = 3.14159265358979323846264338327950f;

GLuint slices = ...;
GLfloat max_height = 2.0f, dalpha = 2.0f * PI / slices, alpha = 0.0f;

vector<Vertex> v;
v.push_back( cone(0.0f, 0.0f) );
for (GLuint i = 0; i < slices; i++, alpha += dalpha)
    v.push_back( cone(max_height, alpha) );

...

glBegin(GL_TRIANGLE_FAN);
 for (vector<Vertex >::const_iterator it = v.begin(); it != v.end(); it++)
        glVertex3fv(&it->x);
glEnd();
```

OpenGL geometric drawing primitives

Convex quadrilaterals – GL_QUADS, GL_QUAD_STRIP

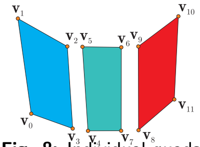

Fig. 8: Individual quads.

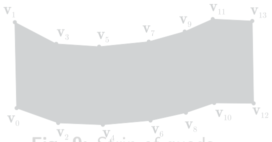

Fig. 9: Strip of quads.

```
#include <vector>

using namespace std;

class Vertex
{
public:
     GLfloat x, y, z;
     ...
};

class Face
{
public:
     GLint node[4];
     ...
};

...

GLuint vertex_count = ...;
vector<Vertex> v(vertex_count);

GLuint quad_count = ...;
vector<Face> f(quad_count);

...

glBegin(GL_QUADS);
  for (vector<Face>::const_iterator it = f.begin(); it != f.end(); it++)
      for (GLuint id = 0; id < 4; id++)
        glVertex3fv(&v[ it->node[id] ].x);
glEnd();
```

OpenGL geometric drawing primitives

Convex quadrilaterals – GL_QUADS, GL_QUAD_STRIP

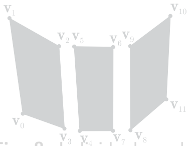

Fig. 8: Individual quads.

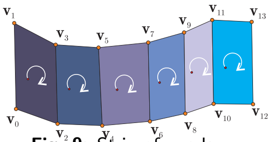

Fig. 9: Strip of quads.

```
#include <cmath>
#include <vector>
using namespace std;

class Vertex
{
public:
    GLfloat x, y, z;
    Vertex(GLfloat x, GLfloat y, GLfloat z = 0.0f): x(x), y(y), z(z) {}
};

Vertex cylinder(GLfloat r, GLfloat alpha, GLfloat height)
{
    return Vertex(r * cos(alpha), r * sin(alpha), height);
}

const GLfloat PI = 3.14159265358979323846264338327950f;

GLuint slices = ...;
GLfloat r = 1.0f, min_height = -2.0f, max_height = 2.0f,
        dalpha = 2.0f * PI / slices, alpha = 0.0f;

vector<Vertex> v;
for (GLuint i = 0; i < slices; i++, alpha += dalpha)
{
    v.push_back( cylinder(r, alpha, min_height) );
    v.push_back( cylinder(r, alpha, max_height) );
}

...

glBegin(GL_QUAD_STRIP);
 for (vector<Vertex >::const_iterator it = v.begin(); it != v.end(); it++)
     glVertex3fv(&it->x);
glEnd();
```

OpenGL geometric drawing primitives

Convex polygons – GL_POLYGON

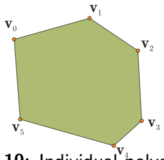

Fig. 10: Individual polygons.

```
#include <cmath>
#include <vector>

using namespace std;

class Vertex
{
public:
    GLfloat x, y, z;
    Vertex(): x(0.0f), y(0.0f), z(0.0f) {}
    ...
};

const GLfloat PI = 3.14159265358979323846264338327950f;

GLuint vertex_count = ...;
vector<Vertex> v(vertex_count);

...

// regular polygon on a unit circle
GLfloat du = 2.0 * PI / vertex_count, u = 0.0f;
for (vector<Vertex>::iterator it = v.begin(); it != v.end(); it++,u += du)
{
    it->x = cos(u);
    it->y = sin(u);
}

...

glBegin(GL_POLYGON);
 for (vector<Vertex>::const_iterator it = v.begin(); it != v.end(); it++)
    glVertex2fv(&it->x);
glEnd();
```

Colors, texture coordinates and normal vectors associated with vertices – part I

Example

```
#include <vector>

using namespace std;

class Color
{
public:
    GLfloat r, g, b, a; // r, g, b, a in [0, 1]
    Color(): r(0.0f), g(0.0f), b(0.0f), a(1.0f) {}
    ...
};

...

class TextureCoordinate
{
public:
    GLfloat u, v; // u, v in [0, 1] -- define the pixel (round(u * width), round(v * height))
                  // of an image of size width x height
    TextureCoordinate(): u(0.0f), v(0.0) {}
};

...

class Vertex
{
public:
    GLfloat x, y, z;
    Vertex(): x(0.0f), y(0.0f), z(0.0f) {}

    GLvoid Normalize();
    ...
};
```

Colors, texture coordinates and normal vectors associated with vertices – part II

Example

```
GLuint vertex_count = ...;

vector<Vertex>              v(vertex_count); // vertices
vector<Vertex>              n(vertex_count); // unit normal vectors of vertices
vector<Color>               c(vertex_count); // colors of vertices
vector<TextureCoordinate>   t(vertex_count); // texture coordinates of vertices

...

// creating/loading texture data and setting the texture parameters
// creating and enabling light objects

...

glBegin(...);
    for (GLuint i = 0; i < vertex_count; i++)
    {
        glColor4fv   (&c[i].r);
        glTexCoord2fv(&t[i].u);
        glNormal3fv  (&n[i].x);
        glVertex3fv  (&v[i].x);
    }
glEnd();
```

Display lists

What are and why use display lists?

- Display lists may improve performance since you can use them to store OpenGL commands for later execution. It is often a good idea to cache commands in a display list if you plan to redraw the same geometry multiple times, or if you have a set of state changes that need to be applied multiple times. Using display lists, you can define the geometry and/or state changes once and execute them multiple times.
- Some graphics hardware may store display lists in dedicated memory or may store the data in an optimized form that is more compatible with the graphics hardware or software.
- When a display list is invoked, the commands in it are executed in the order in which they were issued.
- Most OpenGL commands can be either stored in a display list or issued in immediate mode, which causes them to be executed immediately. One can freely mix immediate-mode programming and display lists within a single program.
- The code examples you have seen so far have used immediate mode.

Display lists

What are and why use display lists?

- Display lists may improve performance since you can use them to store OpenGL commands for later execution. It is often a good idea to cache commands in a display list if you plan to redraw the same geometry multiple times, or if you have a set of state changes that need to be applied multiple times. Using display lists, you can define the geometry and/or state changes once and execute them multiple times.
- Some graphics hardware may store display lists in dedicated memory or may store the data in an optimized form that is more compatible with the graphics hardware or software.
- When a display list is invoked, the commands in it are executed in the order in which they were issued.
- Most OpenGL commands can be either stored in a display list or issued in immediate mode, which causes them to be executed immediately. One can freely mix immediate-mode programming and display lists within a single program.
- The code examples you have seen so far have used immediate mode.

What are and why use display lists?

- Display lists may improve performance since you can use them to store OpenGL commands for later execution. It is often a good idea to cache commands in a display list if you plan to redraw the same geometry multiple times, or if you have a set of state changes that need to be applied multiple times. Using display lists, you can define the geometry and/or state changes once and execute them multiple times.
- Some graphics hardware may store display lists in dedicated memory or may store the data in an optimized form that is more compatible with the graphics hardware or software.
- When a display list is invoked, the commands in it are executed in the order in which they were issued.
- Most OpenGL commands can be either stored in a display list or issued in immediate mode, which causes them to be executed immediately. One can freely mix immediate-mode programming and display lists within a single program.
- The code examples you have seen so far have used immediate mode.

Display lists

What are and why use display lists?

- Display lists may improve performance since you can use them to store OpenGL commands for later execution. It is often a good idea to cache commands in a display list if you plan to redraw the same geometry multiple times, or if you have a set of state changes that need to be applied multiple times. Using display lists, you can define the geometry and/or state changes once and execute them multiple times.
- Some graphics hardware may store display lists in dedicated memory or may store the data in an optimized form that is more compatible with the graphics hardware or software.
- When a display list is invoked, the commands in it are executed in the order in which they were issued.
- Most OpenGL commands can be either stored in a display list or issued in immediate mode, which causes them to be executed immediately. One can freely mix immediate-mode programming and display lists within a single program.
- The code examples you have seen so far have used immediate mode.

Display lists

What are and why use display lists?

- Display lists may improve performance since you can use them to store OpenGL commands for later execution. It is often a good idea to cache commands in a display list if you plan to redraw the same geometry multiple times, or if you have a set of state changes that need to be applied multiple times. Using display lists, you can define the geometry and/or state changes once and execute them multiple times.
- Some graphics hardware may store display lists in dedicated memory or may store the data in an optimized form that is more compatible with the graphics hardware or software.
- When a display list is invoked, the commands in it are executed in the order in which they were issued.
- Most OpenGL commands can be either stored in a display list or issued in immediate mode, which causes them to be executed immediately. One can freely mix immediate-mode programming and display lists within a single program.
- The code examples you have seen so far have used immediate mode.

Display lists – an example, part I

- Consider the class Object that stores some kind of renderable static geometry that must be viewed from different angles.

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

class Object
{
protected:
    GLuint _display_list_name;

    // data structures of the geometry
    ...

public:
    // default constructor
    Object();

    // copy constructor
    Object(const Object& object);

    // assignment operator
    Object& operator =(const Object& rhs);

    // geometry handling methods
    ...

    // display list handling methods
    void DeleteDisplayList();
    bool Render() const;
    virtual bool UpdateDisplayList();

    // destructor
    virtual ~Object();
};
```

Display lists – an example, part II

- Implementation details:

```
// intializing the display list name with 0
Object::Object(): _display_list_name(0), ...
{
    ...
}


// when copying an existing instance of the class Object,
// instead of copying the display list name we need to generate a new display list
Object::Object(const Object& object): _display_list_name(0), ... // copying geometry
{
    // if the display list of the copied object exists, then we clone it
    if (object._display_list_name)
        UpdateDisplayList();
}


// in case of the assignment operator, first, we need to delete the old/existing display list
// and, finally, we have to create a new display list based on the copied geometric properties
Object& Object::operator =(const Object& rhs)
{
    if (this != &rhs)
    {
        // copying geometry
        ...

        // if the display list of the object on the right hand side exists, then we clone it
        if (rhs._display_list_name)
            UpdateDisplayList();
    }
    return *this;
}
```

Display lists – an example, part III

```
void Object::DeleteDisplayList()
{
    // if the display list exists, then we can delete it
    if (_display_list_name)
    {
        glDeleteLists(_display_list_name, 1);
        _display_list_name = 0;
    }
}



bool Object::Render() const
{
    // if the display list exists, the we can invoke it
    if (_display_list_name)
    {
        glCallList(_display_list_name);
        return true;
    }
    return false;
}



bool Object::UpdateDisplayList()
{
    // deleting the old/current display list
    DeleteDisplayList();

    // generating the display list name
    _display_list_name = glGenLists(1);

    // if the display list name is still 0, then there was not enough memory
    if (!_display_list_name)
        return false;
```

Display lists – an example, part IV

```
    // defining/compiling OpenGL commands that form the display list
    glNewList(_display_list_name, GL_COMPILE);

        // pixel, line, material, light, blending, texture, fog, and other properties
        // possible transformations
        glBegin(...);
            ...
        glEnd();

        ...

        // pixel, line, material, light, blending, texture, fog, and other properties
        // possible transformations
        glBegin(...);
            ...
        glEnd();

        ...

        // other state variables, transformations and glBegin —— glEnd blocks
        ...

    glEndList();

    return true;
}

Object::~Object()
{
    // we need to free the display list
    DeleteDisplayList();

    // and we also have to delete the data structures of the geometry
    ...
}
```

Hierarchial display lists – an example, part I

- Consider the derived classes Wheel, SteeringWheel, WindScreen of the previous base class Object. These objects are parts of the class Car that is also derived from the class Object. It is possible that the virtual method

```
bool Object::UpdateDisplayList();
```

  must be redeclared/redefined in all derived classes.

```
class Wheel: public Object
{
private:
    // wheel parameters
    ...

public:
    Wheel(...);
    Wheel(const Wheel& w);
    Wheel& operator =(const Wheel& rhs);

    ...

    bool UpdateDisplayList();

    ...

};

class SteeringWheel: public Object
{
private:
    // steering wheel parameters
    ...
```

Hierarchial display lists – an example, part II

```
public:
    SteeringWheel(...);
    SteeringWheel(const SteeringWheel& sw);
    SteeringWheel& operator =(const SteeringWheel& rhs);

    ...

    bool UpdateDisplayList();

    ...

};

class Windscreen: public Object
{
private:
    // wind screen parameters
    ...

public:
    Windscreen(...);
    Windscreen(const Windscreen& ws);
    Windscreen& operator =(const Windscreen& rhs);

    ...

    bool UpdateDisplayList();

    ...

};


// other classes that represent other parts of the car
...
```

Hierarchial display lists – an example, part III

```
class Car: public Object
{
private:
    Wheel           _wheel;
    SteeringWheel   _steering_wheel;
    Windscreen      _windscreen;

    // other parts of the car
    ...

public:
    Car(...);
    Car(const Car& c);
    Car& operator =(const Car& rhs);

    ...

    bool UpdateDisplayList();

    ...

};

// updating the objects that form the car
Car::Car(...): _wheel(...), _steering_wheel(...), _wind_screen(...), ...
{
}

// creating the hierarchical display list of the car
bool Car::UpdateDisplayList()
{
    if (!_wheel.UpdateDisplayList())
        return false;
```

Hierarchial display lists – an example, part IV

```
    if (!_steering_wheel.UpdateDisplayList())
        return false;

    if (!_windscreen.UpdateDisplayList())
        return false;

    DeleteDisplayList();
    _display_list_name = glGenLists(1);

    if (!_display_list_name)
        return false;

    glNewList(_display_list_name, GL_COMPILE);

        // transformations, material properties, texture properties, etc.
        _wheel.Render();

        // transformations, material properties, texture properties, etc.
        _wheel.Render();

        // transformations, material properties, texture properties, etc.
        _wheel.Render();

        // transformations, material properties, texture properties, etc.
        _wheel.Render();

        // transformations, material properties, texture properties, etc.
        _steering_wheel.Render();

        // transformations, material properties, texture properties, etc.
        _windscreen.Render();

    glEndList();

    return true;
}
```

Commands that cannot be stored in display lists

```
GLvoid glDeleteLists(GLuint list, GLsizei range);
GLuint glGenLists(GLsizei range);

GLvoid glPixelStorei(GLenum pname, GLint param);
GLvoid glPixelStoref(GLenum pname, GLfloat param);

GLvoid glFeedbackBuffer(GLsizei size, GLenum type, GLfloat *buffer);
GLvoid glSelectBuffer(GLsizei size, GLuint *buffer);

GLvoid glGet*(..., ...);

GLvoid glReadPixels(
        GLint x, GLint y,
        GLsizei width, GLsizei height,
        GLenum format, GLenum type, GLvoid *pixels);

GLvoid glFinish();
GLvoid glFlush();

GLboolean glIsEnabled(GLenum cap);

GLboolean glIsList(GLuint list);

GLint glRenderMode(GLenum mode);
```

Saving/restoring state variables and modelview matrices

```
/*
    possible OpenGL bitfields:

    GL_ALL_ATTRIB_BITS,
    GL_ACCUM_BUFFER_BIT, GL_COLOR_BUFFER_BIT, GL_CURRENT_BIT, GL_DEPTH_BUFFER_BIT,
    GL_ENABLE_BIT, GL_EVAL_BIT, GL_FOG_BIT, GL_HINT_BIT, GL_LIGHTING_BIT, GL_LINE_BIT,
    GL_LIST_BIT, GL_PIXEL_MODE_BIT, GL_POINT_BIT, GL_POLYGON_BIT, GL_POLYGON_STIPPLE_BIT,
    GL_SCISSOR_BIT, GL_STENCIL_BUFFER_BIT, GL_TEXTURE_BIT, GL_TRANSFORM_BIT, GL_VIEWPORT_BIT
*/

GLvoid glPushAttrib(GLbitfield mask); // e.g. mask = GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT
GLvoid glPopAttrib();

GLvoid glPushMatrix();
GLvoid glPopMatrix();
```

- Here you can find more information on state values associated with different bitfields.

Saving/restoring state variables and modelview matrices

```
/*
    possible OpenGL bitfields:

    GL_ALL_ATTRIB_BITS,
    GL_ACCUM_BUFFER_BIT, GL_COLOR_BUFFER_BIT, GL_CURRENT_BIT, GL_DEPTH_BUFFER_BIT,
    GL_ENABLE_BIT, GL_EVAL_BIT, GL_FOG_BIT, GL_HINT_BIT, GL_LIGHTING_BIT, GL_LINE_BIT,
    GL_LIST_BIT, GL_PIXEL_MODE_BIT, GL_POINT_BIT, GL_POLYGON_BIT, GL_POLYGON_STIPPLE_BIT,
    GL_SCISSOR_BIT, GL_STENCIL_BUFFER_BIT, GL_TEXTURE_BIT, GL_TRANSFORM_BIT, GL_VIEWPORT_BIT
*/

GLvoid glPushAttrib(GLbitfield mask); // e.g. mask = GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT
GLvoid glPopAttrib();

GLvoid glPushMatrix();
GLvoid glPopMatrix();
```

- Here you can find more information on state values associated with different bitfields.

Vertex arrays

- OpenGL requires many function calls to render geometric primitives, e.g. drawing a 10-sided polygon requires at least 12 function calls: one call to glBegin(), one call for each of the vertices, and a final call to glEnd().
- Moreover, additional information (like colors, surface normals, texture coordinates) added function calls for each vertex. This can quickly double or triple the number of function calls required for one geometric object.
- For some systems, function calls have a great deal of overhead and hinder performance.
- An additional problem is the redundant processing of vertices that are shared between adjacent polygons.
- OpenGL has vertex array routines that allow you to specify a lot of vertex-related data with just a few arrays and to access that data with equally few function calls. E.g. using such routines, all 10 vertices in a 10-sided polygon can be put in one array and called with one function; if each vertex also has a normal vector, all 10 normal vectors can be put into another array and also called with one function.

Vertex arrays

- OpenGL requires many function calls to render geometric primitives, e.g. drawing a 10-sided polygon requires at least 12 function calls: one call to glBegin(), one call for each of the vertices, and a final call to glEnd().
- Moreover, additional information (like colors, surface normals, texture coordinates) added function calls for each vertex. This can quickly double or triple the number of function calls required for one geometric object.
- For some systems, function calls have a great deal of overhead and hinder performance.
- An additional problem is the redundant processing of vertices that are shared between adjacent polygons.
- OpenGL has vertex array routines that allow you to specify a lot of vertex-related data with just a few arrays and to access that data with equally few function calls. E.g. using such routines, all 10 vertices in a 10-sided polygon can be put in one array and called with one function; if each vertex also has a normal vector, all 10 normal vectors can be put into another array and also called with one function.

Vertex arrays

- OpenGL requires many function calls to render geometric primitives, e.g. drawing a 10-sided polygon requires at least 12 function calls: one call to glBegin(), one call for each of the vertices, and a final call to glEnd().
- Moreover, additional information (like colors, surface normals, texture coordinates) added function calls for each vertex. This can quickly double or triple the number of function calls required for one geometric object.
- For some systems, function calls have a great deal of overhead and hinder performance.
- An additional problem is the redundant processing of vertices that are shared between adjacent polygons.
- OpenGL has vertex array routines that allow you to specify a lot of vertex-related data with just a few arrays and to access that data with equally few function calls. E.g. using such routines, all 10 vertices in a 10-sided polygon can be put in one array and called with one function; if each vertex also has a normal vector, all 10 normal vectors can be put into another array and also called with one function.

Vertex arrays

- OpenGL requires many function calls to render geometric primitives, e.g. drawing a 10-sided polygon requires at least 12 function calls: one call to glBegin(), one call for each of the vertices, and a final call to glEnd().
- Moreover, additional information (like colors, surface normals, texture coordinates) added function calls for each vertex. This can quickly double or triple the number of function calls required for one geometric object.
- For some systems, function calls have a great deal of overhead and hinder performance.
- An additional problem is the redundant processing of vertices that are shared between adjacent polygons.
- OpenGL has vertex array routines that allow you to specify a lot of vertex-related data with just a few arrays and to access that data with equally few function calls. E.g. using such routines, all 10 vertices in a 10-sided polygon can be put in one array and called with one function; if each vertex also has a normal vector, all 10 normal vectors can be put into another array and also called with one function.

Vertex arrays

- OpenGL requires many function calls to render geometric primitives, e.g. drawing a 10-sided polygon requires at least 12 function calls: one call to glBegin(), one call for each of the vertices, and a final call to glEnd().
- Moreover, additional information (like colors, surface normals, texture coordinates) added function calls for each vertex. This can quickly double or triple the number of function calls required for one geometric object.
- For some systems, function calls have a great deal of overhead and hinder performance.
- An additional problem is the redundant processing of vertices that are shared between adjacent polygons.
- OpenGL has vertex array routines that allow you to specify a lot of vertex-related data with just a few arrays and to access that data with equally few function calls. E.g. using such routines, all 10 vertices in a 10-sided polygon can be put in one array and called with one function; if each vertex also has a normal vector, all 10 normal vectors can be put into another array and also called with one function.

Enabling/disabling vertex arrays

- One can activate/enable up to eight arrays, each storing a different type of data: vertex coordinates, surface normals, RGBA colors, secondary colors, color indices, fog coordinates, texture coordinates, or polygon edge flags.

```
/*
    acceptable parameters are:
    GL_VERTEX_ARRAY,
    GL_NORMAL_ARRAY,
    GL_COLOR_ARRAY, GL_SECONDARY_COLOR_ARRAY, GL_INDEX_ARRAY,
    GL_FOG_COORDINATE_ARRAY,
    GL_TEXTURE_COORD_ARRAY,
    GL_EDGE_FLAG_ARRAY
*/

GLvoid glEnableClientState(GLenum array);
GLvoid glDisableClientState(GLenum array);

// e.g. if you use lighting, you need to activate
// both the surface normal and vertex coordinate arrays
glEnableClientState(GL_NORMAL_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);

...

// now, suppose that you want to deactivate the lighting,
// in this case you also want to stop changing the values of the surface normal state,
// which is wasted effort; to do this, you need to call
glDisableClientState(GL_NORMAL_ARRAY);
```

- The specification of vertex arrays cannot be stored in a display list and the data remains on the side of the client.

Enabling/disabling vertex arrays

- One can activate/enable up to eight arrays, each storing a different type of data: vertex coordinates, surface normals, RGBA colors, secondary colors, color indices, fog coordinates, texture coordinates, or polygon edge flags.

```
/*
    acceptable parameters are:
    GL_VERTEX_ARRAY,
    GL_NORMAL_ARRAY,
    GL_COLOR_ARRAY, GL_SECONDARY_COLOR_ARRAY, GL_INDEX_ARRAY,
    GL_FOG_COORDINATE_ARRAY,
    GL_TEXTURE_COORD_ARRAY,
    GL_EDGE_FLAG_ARRAY
*/

GLvoid glEnableClientState(GLenum array);
GLvoid glDisableClientState(GLenum array);

// e.g. if you use lighting, you need to activate
// both the surface normal and vertex coordinate arrays
glEnableClientState(GL_NORMAL_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);

...

// now, suppose that you want to deactivate the lighting,
// in this case you also want to stop changing the values of the surface normal state,
// which is wasted effort; to do this, you need to call
glDisableClientState(GL_NORMAL_ARRAY);
```

- The specification of vertex arrays cannot be stored in a display list and the data remains on the side of the client.

Specifying data for arrays, part I

- There are eight different routines for specifying arrays – one routine for each kind of array.

```
/*
 - specifies where spatial coordinate data can be accessed;
 - pointer is the memory address of the first coordinate of the first vertex in the array;
 - type specifies the data type (GL_SHORT, GL_INT, GL_FLOAT, or GL_DOUBLE) of each coordinate
   in the array;
 - size is the number of coordinates per vertex, which must be 2, 3, or 4;
 - stride is the byte offset between consecutive vertices, e.g. if stride is 0, the vertices
   are understood to be tightly packed in the array
*/
GLvoid glVertexPointer(GLint size, GLenum type, GLsizei stride, const GLvoid* pointer);

// To access the other seven arrays, there are seven similar routines:

/*
   - sizes: 2, 3, 4;
   - types: GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT,
            GL_FLOAT, GL_DOUBLE
*/
GLvoid glColorPointer(GLint size, GLenum type, GLsizei stride, const GLvoid* pointer);

/*
   - size: 3;
   - types: GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, GL_UNSIGNED_INT,
            GL_FLOAT, GL_DOUBLE
*/
GLvoid glSecondaryColorPointer(GLint size, GLenum type, GLsizei stride, const GLvoid* pointer);

/*
   - size: 1;
   - types: GL_UNSIGNED_BYTE, GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE
*/
GLvoid glIndexPointer(GLenum type, GLsizei stride, const GLvoid* pointer);
```

Specifying data for arrays, part II

```
/*
    − size: 3;
    − types: GL_BYTE, GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE
*/
GLvoid glNormalPointer(GLenum type, GLsizei stride, const GLvoid* pointer);

/*
    − size: 1;
    − types: GL_FLOAT, GL_DOUBLE
*/
GLvoid glFogPointer(GLenum type, GLsizei stride, const GLvoid* pointer);

/*
    − sizes: 1, 2, 3, 4;
    − types: GL_SHORT, GL_INT, GL_FLOAT, GL_DOUBLE
*/
GLvoid glTexCoordPointer(GLint size, GLenum type, GLsizei stride, const GLvoid* pointer);

/*
    − size: 1;
    − type: no type argument, i.e. type of data must be GLboolean
*/
GLvoid glEdgeFlagPointer(GLsizei stride, const GLvoid* pointer);
```

Specifying data for arrays, part III

```
// a simple example
static GLint vertices[] = { 25,  25,
                           100, 325,
                           175,  25,
                           175, 325,
                           250,  25,
                           325, 325};

static GLfloat colors[] = {1.00, 0.20, 0.20,
                           0.20, 0.20, 1.00,
                           0.80, 1.00, 0.20,
                           0.75, 0.75, 0.75,
                           0.35, 0.35, 0.35,
                           0.50, 0.50, 0.50};

glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);

glColorPointer(3, GL_FLOAT, 0, colors);
glVertexPointer(2, GL_INT, 0, vertices);


...
// an intertwined example:      colors            vertices
static GLfloat mixed[]   =  {1.0, 0.2, 1.0,    100.0, 100.0, 0.0,
                             1.0, 0.2, 0.2,      0.0, 200.0, 0.0,
                             1.0, 1.0, 0.2,    100.0, 300.0, 0.0,
                             0.2, 1.0, 0.2,    200.0, 300.0, 0.0,
                             0.2, 1.0, 1.0,    300.0, 200.0, 0.0,
                             0.2, 0.2, 1.0,    200.0, 100.0, 0.0};

glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);

glColorPointer(3, GL_FLOAT, 6 * sizeof(GLfloat), &mixed[0]);
glVertexPointer(2, GL_FLOAT, 6 * sizeof(GLfloat), &mixed[3]);
```

Dereferencing and rendering of arrays

- Until the contents of the vertex arrays are dereferenced, the arrays remain on the client side, and their contents are easily changed.
- When the vertex array are derefenced, contents of the arrays are obtained, sent to the server, and then sent down the graphics processing pipeline for rendering.
- One can obtain data from a single array element (indexed location), from an ordered list of array elements (which may be limited to a subset of the entire vertex array data), or from a sequence of array elements.

```
// dereferencing a single array element
GLvoid glArrayElement(GLint ith);

// dereferencing a list of array elements
GLvoid glDrawElements(GLenum mode, GLsizei count, GLenum type, const GLvoid* indices);

GLvoid glMultiDrawElements(GLenum mode, GLsizei* element_count_of_each_list,
                           GLenum type, GLvoid** addresses_of_indices_lists, GLsizei list_count);

GLvoid glDrawRangeElements(GLenum mode, GLuint start, GLuint end,
                           GLsizei count, GLenum type, GLvoid* indices);

// dereferencing a sequence of array elements
GLvoid glDrawArrays(GLenum mode, GLint first, GLsizei count);
```

Dereferencing and rendering of arrays

- Until the contents of the vertex arrays are dereferenced, the arrays remain on the client side, and their contents are easily changed.
- When the vertex array are derefenced, contents of the arrays are obtained, sent to the server, and then sent down the graphics processing pipeline for rendering.
- One can obtain data from a single array element (indexed location), from an ordered list of array elements (which may be limited to a subset of the entire vertex array data), or from a sequence of array elements.

```
// dereferencing a single array element
GLvoid glArrayElement(GLint ith);

// dereferencing a list of array elements
GLvoid glDrawElements(GLenum mode, GLsizei count, GLenum type, const GLvoid* indices);

GLvoid glMultiDrawElements(GLenum mode, GLsizei* element_count_of_each_list,
                           GLenum type, GLvoid** addresses_of_indices_lists, GLsizei list_count);

GLvoid glDrawRangeElements(GLenum mode, GLuint start, GLuint end,
                           GLsizei count, GLenum type, GLvoid* indices);

// dereferencing a sequence of array elements
GLvoid glDrawArrays(GLenum mode, GLint first, GLsizei count);
```

Dereferencing and rendering of arrays

- Until the contents of the vertex arrays are dereferenced, the arrays remain on the client side, and their contents are easily changed.
- When the vertex array are derefenced, contents of the arrays are obtained, sent to the server, and then sent down the graphics processing pipeline for rendering.
- One can obtain data from a single array element (indexed location), from an ordered list of array elements (which may be limited to a subset of the entire vertex array data), or from a sequence of array elements.

```
// dereferencing a single array element
GLvoid glArrayElement(GLint ith);

// dereferencing a list of array elements
GLvoid glDrawElements(GLenum mode, GLsizei count, GLenum type, const GLvoid* indices);

GLvoid glMultiDrawElements(GLenum mode, GLsizei* element_count_of_each_list,
                           GLenum type, GLvoid** addresses_of_indices_lists, GLsizei list_count);

GLvoid glDrawRangeElements(GLenum mode, GLuint start, GLuint end,
                            GLsizei count, GLenum type, GLvoid* indices);

// dereferencing a sequence of array elements
GLvoid glDrawArrays(GLenum mode, GLint first, GLsizei count);
```

Dereferencing and rendering of arrays

Dereferencing a single array element

Example

```
static GLint vertices[] = { 25,  25,    100, 325,   175,  25,   175, 325,   250, 25,
325, 325};

static GLfloat colors[] = {1.00, 0.20, 0.20,    0.20, 0.20, 1.00,    0.80, 1.00, 0.20,
                           0.75, 0.75, 0.75,   0.35, 0.35, 0.35,   0.50, 0.50, 0.50};
...

glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);

    glColorPointer(3, GL_FLOAT, 0, colors);
    glVertexPointer(2, GL_INT, 0, vertices);

    ...

    glBegin(GL_TRIANGLES);
        glArrayElement(2);
        glArrayElement(3);
        glArrayElement(5);
    glEnd();

    ...

glDisableClientState(GL_COLOR_ARRAY);
glDisableClientState(GL_VERTEX_ARRAY);

// when executed, the glBegin(GL_TRIANGLES) -- glEnd() block has the same effect as
glBegin(GL_TRIANGLES);
    glColor3fv(colors + (2 * 3));
    glVertex2iv(vertices + (2 * 2));
    glColor3fv(colors + (3 * 3));
    glVertex2iv(vertices + (3 * 2));
    glColor3fv(colors + (5 * 3));
    glVertex2iv(vertices + (5 * 2));
glEnd();
```

- Since

  ```
  GLvoid glArrayElement(GLint ith);
  ```

  is only a single function call per vertex, it may reduce the number of function calls, which increases overall performance.
- If the vertex coordinate array is enabled, the glVertex*v() routine is executed last, after the execution (if enabled) of up to seven corresponding array values.
- Be warned that if the contents of the array are changed between glBegin() and glEnd(), there is no guarantee that you will receive original or changed data for your requested element. To be safe, do not change the contents of any array element that might be accessed until the primitive is completed.

Dereferencing and rendering of arrays

Dereferencing a single array element

- Since

  ```
  GLvoid glArrayElement(GLint ith);
  ```

  is only a single function call per vertex, it may reduce the number of function calls, which increases overall performance.
- If the vertex coordinate array is enabled, the glVertex*v() routine is executed last, after the execution (if enabled) of up to seven corresponding array values.
- Be warned that if the contents of the array are changed between glBegin() and glEnd(), there is no guarantee that you will receive original or changed data for your requested element. To be safe, do not change the contents of any array element that might be accessed until the primitive is completed.

- Since

  ```
  GLvoid glArrayElement(GLint ith);
  ```

  is only a single function call per vertex, it may reduce the number of function calls, which increases overall performance.
- If the vertex coordinate array is enabled, the glVertex*v() routine is executed last, after the execution (if enabled) of up to seven corresponding array values.
- Be warned that if the contents of the array are changed between glBegin() and glEnd(), there is no guarantee that you will receive original or changed data for your requested element. To be safe, do not change the contents of any array element that might be accessed until the primitive is completed.

Dereferencing and rendering of arrays

Dereferencing a list of array elements

```
/*
  Defines a sequence of geometric primitives using count number of elements, whose indices
  are stored in the array indices. type must be one of GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT,
  or GL_UNSIGNED_INT, indicating the data type of the indices array. mode specifies what
  kind of primitives are constructed and is one of the same values that is accepted by glBegin();
  e.g. GL_POINTS, GL_LINES, GL_LINE_LOOP, GL_TRIANGLES, and so on.
*/
GLvoid glDrawElements(GLenum mode, GLsizei count, GLenum type, const GLvoid* indices);
```

Example

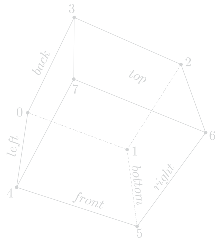

Fig. 11: A cube with numbered vertices.

```
static GLubyte front_indices[] = {4, 5, 6, 7};
static GLubyte right_indices[] = {1, 2, 6, 5};
static GLubyte bottom_indices[] = {0, 1, 5, 4};
static GLubyte back_indices[] = {0, 3, 2, 1};
static GLubyte left_indices[] = {0, 4, 7, 3};
static GLubyte top_indices[] = {2, 3, 7, 6};

glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, front_indices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, right_indices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, bottom_indices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, back_indices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, left_indices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, top_indices);

// or, alternatively, you may compact all indices as follows
static GLubyte all_indices[] = {4, 5, 6, 7, 1, 2, 6, 5,
                                0, 1, 5, 4, 0, 3, 2, 1,
                                0, 4, 7, 3, 2, 3, 7, 6};

glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, all_indices);
```

It is an error to encapsulate glDrawElements() between a glBegin()/glEnd() pair.

Dereferencing and rendering of arrays

Dereferencing a list of array elements

```
/*
  Defines a sequence of geometric primitives using count number of elements, whose indices
  are stored in the array indices. type must be one of GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT,
  or GL_UNSIGNED_INT, indicating the data type of the indices array. mode specifies what
  kind of primitives are constructed and is one of the same values that is accepted by glBegin();
  e.g. GL_POINTS, GL_LINES, GL_LINE_LOOP, GL_TRIANGLES, and so on.
*/
GLvoid glDrawElements(GLenum mode, GLsizei count, GLenum type, const GLvoid* indices);
```

Example

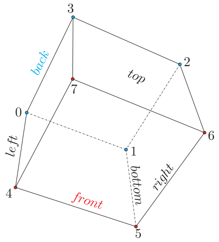

Fig. 11: A cube with numbered vertices.

```
static GLubyte front_indices[] = {4, 5, 6, 7};
static GLubyte right_indices[] = {1, 2, 6, 5};
static GLubyte bottom_indices[] = {0, 1, 5, 4};
static GLubyte back_indices[] = {0, 3, 2, 1};
static GLubyte left_indices[] = {0, 4, 7, 3};
static GLubyte top_indices[] = {2, 3, 7, 6};

glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, front_indices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, right_indices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, bottom_indices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, back_indices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, left_indices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, top_indices);

// or, alternatively, you may compact all indices as follows
static GLubyte all_indices[] = {4, 5, 6, 7, 1, 2, 6, 5
                                0, 1, 5, 4, 0, 3, 2, 1,
                                0, 4, 7, 3, 2, 3, 7, 6};

glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, all_indices);
```

It is an error to encapsulate glDrawElements() between a glBegin()/glEnd() pair.

Dereferencing and rendering of arrays

Dereferencing a list of array elements

```
/*
  Defines a sequence of geometric primitives using count number of elements, whose indices
  are stored in the array indices. type must be one of GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT,
  or GL_UNSIGNED_INT, indicating the data type of the indices array. mode specifies what
  kind of primitives are constructed and is one of the same values that is accepted by glBegin();
  e.g. GL_POINTS, GL_LINES, GL_LINE_LOOP, GL_TRIANGLES, and so on.
*/
GLvoid glDrawElements(GLenum mode, GLsizei count, GLenum type, const GLvoid* indices);
```

Example

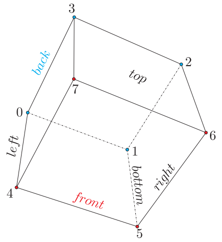

Fig. 11: A cube with numbered vertices.

```
static GLubyte front_indices[] = {4, 5, 6, 7};
static GLubyte right_indices[] = {1, 2, 6, 5};
static GLubyte bottom_indices[] = {0, 1, 5, 4};
static GLubyte back_indices[] = {0, 3, 2, 1};
static GLubyte left_indices[] = {0, 4, 7, 3};
static GLubyte top_indices[] = {2, 3, 7, 6};

glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, front_indices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, right_indices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, bottom_indices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, back_indices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, left_indices);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, top_indices);

// or, alternatively, you may compact all indices as follows
static GLubyte all_indices[] = {4, 5, 6, 7, 1, 2, 6, 5
                                0, 1, 5, 4, 0, 3, 2, 1,
                                0, 4, 7, 3, 2, 3, 7, 6};

glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, all_indices);
```

It is an error to encapsulate glDrawElements() between a glBegin()/glEnd() pair.

Dereferencing and rendering of arrays

Dereferencing a list of array elements

- The effect of

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

  is almost the same as this command sequence:

  ```
  glBegin(mode);
      for (GLsizei i = 0; i < count; i++)
          glArrayElement(indices[i]);
  glEnd();
  ```

- Additionaly checks to make sure mode, count and type are valid.
- Also, unlike the preceding sequence of commands, executing glDrawElements() leaves several states indeterminate.
- After its execution, current RGB color, secondary color, color index, normal coordinates, fog coordinates, texture coordinates, and edge flags are indeterminate if the corresponding array has been enabled.

Dereferencing and rendering of arrays

Dereferencing a list of array elements

- The effect of

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

  is almost the same as this command sequence:

  ```
  glBegin(mode);
      for (GLsizei i = 0; i < count; i++)
          glArrayElement(indices[i]);
  glEnd();
  ```

- Additionaly checks to make sure mode, count and type are valid.
- Also, unlike the preceding sequence of commands, executing glDrawElements() leaves several states indeterminate.
- After its execution, current RGB color, secondary color, color index, normal coordinates, fog coordinates, texture coordinates, and edge flags are indeterminate if the corresponding array has been enabled.

Dereferencing and rendering of arrays

Dereferencing a list of array elements

- The effect of

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

  is almost the same as this command sequence:

  ```
  glBegin(mode);
      for (GLsizei i = 0; i < count; i++)
          glArrayElement(indices[i]);
  glEnd();
  ```

- Additionaly checks to make sure mode, count and type are valid.
- Also, unlike the preceding sequence of commands, executing glDrawElements() leaves several states indeterminate.
- After its execution, current RGB color, secondary color, color index, normal coordinates, fog coordinates, texture coordinates, and edge flags are indeterminate if the corresponding array has been enabled.

Dereferencing and rendering of arrays

- The effect of

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

  is almost the same as this command sequence:

  ```
  glBegin(mode);
      for (GLsizei i = 0; i < count; i++)
          glArrayElement(indices[i]);
  glEnd();
  ```

- Additionaly checks to make sure mode, count and type are valid.
- Also, unlike the preceding sequence of commands, executing glDrawElements() leaves several states indeterminate.
- After its execution, current RGB color, secondary color, color index, normal coordinates, fog coordinates, texture coordinates, and edge flags are indeterminate if the corresponding array has been enabled.

Dereferencing and rendering of arrays

Dereferencing a list of array elements

```
/*
  Calls a sequence of list_count number of glDrawElements() commands.
  addresses_of_indices_lists is an array of pointers to lists of array elements.
  index_count_of_each_list is an array of how many vertices are found in each respective array
  element list.
  mode (primitive type) and type (data type) are the same as in case of glDrawElements().
*/

GLvoid glMultiDrawElements(GLenum mode, GLsizei* index_count_of_each_list,
                           GLenum type, GLvoid** addresses_of_indices_lists, GLsizei list_count);
```

Example

```
static GLsizei list_count = 2;

static GLubyte indices_list_1[] = {0, 1, 2, 3, 4, 5, 6};
static GLubyte indices_list_2[] = {7, 1, 9, 8, 10, 11};

static GLsizei index_count_of_each_list[list_count] = {7, 6};

static GLvoid* addresses_of_indices_lists[list_count] = {indices_list_1, indices_list_2};

glMultiDrawElements(/*primitive type*/,
                    index_count_of_each_list, GL_UNSIGNED_BYTE,
                    addresses_of_indices_lists, list_count);
```

Dereferencing and rendering of arrays

Dereferencing a list of array elements

```
/*
  Calls a sequence of list_count number of glDrawElements() commands.
  addresses_of_indices_lists is an array of pointers to lists of array elements.
  index_count_of_each_list is an array of how many vertices are found in each respective array
  element list.
  mode (primitive type) and type (data type) are the same as in case of glDrawElements().
*/

GLvoid glMultiDrawElements(GLenum mode, GLsizei* index_count_of_each_list,
                           GLenum type, GLvoid** addresses_of_indices_lists, GLsizei list_count);
```

Example

```
static GLsizei list_count = 2;

static GLubyte indices_list_1[] = {0, 1, 2, 3, 4, 5, 6};
static GLubyte indices_list_2[] = {7, 1, 9, 8, 10, 11};

static GLsizei index_count_of_each_list[list_count] = {7, 6};

static GLvoid* addresses_of_indices_lists[list_count] = {indices_list_1, indices_list_2};

glMultiDrawElements(/*primitive type*/,
                    index_count_of_each_list, GL_UNSIGNED_BYTE,
                    addresses_of_indices_lists, list_count);
```

Dereferencing and rendering of arrays

Dereferencing a list of array elements

- The effect of

```
GLvoid glMultiDrawElements(GLenum mode, GLsizei* index_count_of_each_list,
                           GLenum type, GLvoid** addresses_of_indices_lists, GLsizei list_count);
```

  is the same as

```
for (GLsizei i = 0; i < list_count; i++)
{
  if (index_count_of_each_list[i] > 0)
  {
    glDrawElements(mode, index_count_of_each_list[i], type, addresses_of_indices_lists[i]);
  }
}
```

  i.e. combines the effects of several glDrawElements() calls into a single one.

Dereferencing and rendering of arrays

Dereferencing a list of array elements

```
/*
    Creates a sequence of geometric primitives that is similar to, but more restricted than,
    the sequence created by glDrawElements().
    Several parameters of glDrawRangeElements() are the same as counterparts in glDrawElements(),
    including mode (kind of primitives), count (number of elements), type (data type), and indices
    (array locations of vertex data).
    glDrawRangeElements() introduces two new parameters: start and end, which specify a range of
    acceptable values for indices. To be valid, values in the array indices must lie between
    start and end, inclusive.
*/

GLvoid glDrawRangeElements(GLenum mode,
                           GLuint start, GLuint end,
                           GLsizei count, GLenum type, GLvoid* indices);
```

Dereferencing and rendering of arrays

Dereferencing a sequence of array elements

```
/*
    Constructs a sequence of geometric primitives using array elements starting at
    first and ending at first + count − 1 of each enabled array.
    mode specifies what kind of primitives are constructed and accepts the same values
    as the command glBegin().
*/

GLvoid glDrawArrays(GLenum mode, GLint first, GLsizei count);
```

- Its effect is almost the same as the command sequence

```
glBegin(mode);
    for (GLsizei i = 0; i < count; i++)
        glArrayElement(first + i);
glEnd();
```

```
/*
    Calls a sequence of primcount (a number of) glDrawArrays() commands.
    mode specifies the primitive type with the same values as accepted by glBegin().
    first and count contain lists of array locations indicating where to process each list of array
    elements.
    Therefore, for the ith list of array elements, a geometric primitive is constructed starting at
    first[i] and ending at first[i] + count[i] − 1.
*/

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

- Its effect is the same as

```
for (GLsizei i = 0; i < primcount; i++)
{
    if (count[i] > 0)
        glDrawArrays(mode, first[i], count[i]);
}
```

Dereferencing and rendering of arrays

Dereferencing a sequence of array elements

```
/*
    Constructs a sequence of geometric primitives using array elements starting at
    first and ending at first + count − 1 of each enabled array.
    mode specifies what kind of primitives are constructed and accepts the same values
    as the command glBegin().
*/

GLvoid glDrawArrays(GLenum mode, GLint first, GLsizei count);
```

- Its effect is almost the same as the command sequence

```
glBegin(mode);
    for (GLsizei i = 0; i < count; i++)
        glArrayElement(first + i);
glEnd();
```

```
/*
    Calls a sequence of primcount (a number of) glDrawArrays() commands.
    mode specifies the primitive type with the same values as accepted by glBegin().
    first and count contain lists of array locations indicating where to process each list of array
    elements.
    Therefore, for the ith list of array elements, a geometric primitive is constructed starting at
    first[i] and ending at first[i] + count[i] − 1.
*/

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

- Its effect is the same as

```
for (GLsizei i = 0; i < primcount; i++)
{
    if (count[i] > 0)
        glDrawArrays(mode, first[i], count[i]);
}
```

Interleaved arrays

```
/*
    Initializes all eight arrays, disabling arrays that are not specified in format,
    and enabling the arrays that are specified.

    format is one of 14 symbolic constants, which represent 14 data configurations:
    GL_V2F, GL_V3F,
    GL_C4UB_V2F, GL_C4UB_V3F,
    GL_C3F_V3F, GL_N3F_V3F, GL_C4F_N3F_V3F, GL_T2F_V3F,
    GL_T4F_V4F, GL_T2F_C4UB_V3F, GL_T2F_C3F_V3F, GL_T2F_N3F_V3F,
    GL_T2F_C4F_N3F_V3F, GL_T4F_C4F_N3F_V4F.

    stride specifies the byte offset between consecutive vertices. If stride is 0, the vertices
    are understood to be tightly packed in the array.
    pointer is the memory address of the first coordinate of the first vertex in the array.
    If multitexturing is enabled, the command affects only the active texture unit.
*/

GLvoid glInterleavedArrays(GLenum format, GLsizei stride, const GLvoid* pointer);
```

Example

```
// an intertwined example:       colors               vertices
static GLfloat mixed[]   =  {1.0, 0.2, 1.0,     100.0, 100.0, 0.0,
                             1.0, 0.2, 0.2,       0.0, 200.0, 0.0,
                             1.0, 1.0, 0.2,     100.0, 300.0, 0.0,
                             0.2, 1.0, 0.2,     200.0, 300.0, 0.0,
                             0.2, 1.0, 1.0,     300.0, 200.0, 0.0,
                             0.2, 0.2, 1.0,     200.0, 100.0, 0.0};

// the command sequence
glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);

glColorPointer(3, GL_FLOAT, 6 * sizeof(GLfloat), &mixed[0]);
glVertexPointer(2, GL_FLOAT, 6 * sizeof(GLfloat), &mixed[3]);

// is equivalent to the single function call
glInterleavedArrays(GL_C3F_V3F, 0, mixed);
```

Interleaved arrays

```
/*
    Initializes all eight arrays, disabling arrays that are not specified in format,
    and enabling the arrays that are specified.

    format is one of 14 symbolic constants, which represent 14 data configurations:
    GL_V2F, GL_V3F,
    GL_C4UB_V2F, GL_C4UB_V3F,
    GL_C3F_V3F, GL_N3F_V3F, GL_C4F_N3F_V3F, GL_T2F_V3F,
    GL_T4F_V4F, GL_T2F_C4UB_V3F, GL_T2F_C3F_V3F, GL_T2F_N3F_V3F,
    GL_T2F_C4F_N3F_V3F, GL_T4F_C4F_N3F_V4F.

    stride specifies the byte offset between consecutive vertices. If stride is 0, the vertices
    are understood to be tightly packed in the array.
    pointer is the memory address of the first coordinate of the first vertex in the array.
    If multitexturing is enabled, the command affects only the active texture unit.
*/

GLvoid glInterleavedArrays(GLenum format, GLsizei stride, const GLvoid* pointer);
```

Example

```
// an intertwined example:      colors             vertices
static GLfloat mixed[]    =  {1.0, 0.2, 1.0,    100.0, 100.0, 0.0,
                              1.0, 0.2, 0.2,      0.0, 200.0, 0.0,
                              1.0, 1.0, 0.2,    100.0, 300.0, 0.0,
                              0.2, 1.0, 0.2,    200.0, 300.0, 0.0,
                              0.2, 1.0, 1.0,    300.0, 200.0, 0.0,
                              0.2, 0.2, 1.0,    200.0, 100.0, 0.0};
// the command sequence
glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);

glColorPointer(3, GL_FLOAT, 6 * sizeof(GLfloat), &mixed[0]);
glVertexPointer(2, GL_FLOAT, 6 * sizeof(GLfloat), &mixed[3]);

// is equivalent to the single function call
glInterleavedArrays(GL_C3F_V3F, 0, mixed);
```

Vertex buffer objects. Additional dependencies

GL Extension Wrangler library

GLEW – setting dependencies in your project

- Download and unzip the directory structure Dependencies aside your project file (⋆.pro) and into your project-build-desktop folder. The corresponding bit-variant glew32.dll must be also copied either near to your executable files, or into the folder Windows/system32.
- Append your project file with

```
win32 {
    message("Windows␣platform...")
    INCLUDEPATH += $$PWD/Dependencies/Include
    DEPENDPATH += $$PWD/Dependencies/Include
    LIBS += -lopengl32 -lglu32

    CONFIG(release, debug|release): {
        contains(QT_ARCH, i386) {
            message("x86␣(i.e.,␣32-bit)␣release␣build")
            LIBS += -L"$$PWD/Dependencies/Lib/GL/x86/" -lglew32
        } else {
            message("x86_64␣(i.e.,␣64-bit)␣release␣build")
            LIBS += -L"$$PWD/Dependencies/Lib/GL/x64/" -lglew32
        }
    } else: CONFIG(debug, debug|release): {
        contains(QT_ARCH, i386) {
            message("x86␣(i.e.,␣32-bit)␣debug␣build")
            LIBS += -L"$$PWD/Dependencies/Lib/GL/x86/" -lglew32
        } else {
            message("x86_64␣(i.e.,␣64-bit)␣debug␣build")
            LIBS += -L"$$PWD/Dependencies/Lib/GL/x64" -lglew32
        }
    }
}
```

Vertex buffer objects. Additional dependencies

GL Extension Wrangler library

GLEW – setting dependencies in your project

- Download and unzip the directory structure Dependencies aside your project file (⋆.pro) and into your project-build-desktop folder. The corresponding bit-variant glew32.dll must be also copied either near to your executable files, or into the folder Windows/system32.
- Append your project file with

```
win32 {
    message("Windows␣platform...")
    INCLUDEPATH += $$PWD/Dependencies/Include
    DEPENDPATH += $$PWD/Dependencies/Include
    LIBS += -lopengl32 -lglu32

    CONFIG(release, debug|release): {
        contains(QT_ARCH, i386) {
            message("x86␣(i.e.,␣32-bit)␣release␣build")
            LIBS += -L"$$PWD/Dependencies/Lib/GL/x86/" -lglew32
        } else {
            message("x86_64␣(i.e.,␣64-bit)␣release␣build")
            LIBS += -L"$$PWD/Dependencies/Lib/GL/x64/" -lglew32
        }
    } else: CONFIG(debug, debug|release): {
        contains(QT_ARCH, i386) {
            message("x86␣(i.e.,␣32-bit)␣debug␣build")
            LIBS += -L"$$PWD/Dependencies/Lib/GL/x86/" -lglew32
        } else {
            message("x86_64␣(i.e.,␣64-bit)␣debug␣build")
            LIBS += -L"$$PWD/Dependencies/Lib/GL/x64" -lglew32
        }
    }
}
```

Vertex buffer objects. Additional dependencies

GL Extension Wrangler library

GLEW – setting dependencies in your project

- Notice that the header file glew.h must be included before QtOpenGL.h or before any header file that includes gl.h and does not includes glew.h.

Vertex buffer objects

- While vertex arrays help reduce the number of function calls to specify geometric objects, the data still resides in the memory of the client.
- Every time you want to utilize the enabled vertex arrays for rendering, the data is transferred from the client to the OpenGL server. While that may be as simple as sending data from memory to the graphics card in your computer, sending data in each frame can be a hindrance to performance.
- To address this situation, OpenGL 1.5 introduced buffer objects that allow your data to be stored in the server and transferred only once, assuming there are enough memory resources to hold your data. Moreover, you can update the stored values in the server.
- In order to represent/render curves and surfaces, we will use vertex buffer objects.

Vertex buffer objects

- While vertex arrays help reduce the number of function calls to specify geometric objects, the data still resides in the memory of the client.
- Every time you want to utilize the enabled vertex arrays for rendering, the data is transferred from the client to the OpenGL server. While that may be as simple as sending data from memory to the graphics card in your computer, sending data in each frame can be a hindrance to performance.
- To address this situation, OpenGL 1.5 introduced buffer objects that allow your data to be stored in the server and transferred only once, assuming there are enough memory resources to hold your data. Moreover, you can update the stored values in the server.
- In order to represent/render curves and surfaces, we will use vertex buffer objects.

Vertex buffer objects

- While vertex arrays help reduce the number of function calls to specify geometric objects, the data still resides in the memory of the client.
- Every time you want to utilize the enabled vertex arrays for rendering, the data is transferred from the client to the OpenGL server. While that may be as simple as sending data from memory to the graphics card in your computer, sending data in each frame can be a hindrance to performance.
- To address this situation, OpenGL 1.5 introduced buffer objects that allow your data to be stored in the server and transferred only once, assuming there are enough memory resources to hold your data. Moreover, you can update the stored values in the server.
- In order to represent/render curves and surfaces, we will use vertex buffer objects.

Vertex buffer objects

- While vertex arrays help reduce the number of function calls to specify geometric objects, the data still resides in the memory of the client.
- Every time you want to utilize the enabled vertex arrays for rendering, the data is transferred from the client to the OpenGL server. While that may be as simple as sending data from memory to the graphics card in your computer, sending data in each frame can be a hindrance to performance.
- To address this situation, OpenGL 1.5 introduced buffer objects that allow your data to be stored in the server and transferred only once, assuming there are enough memory resources to hold your data. Moreover, you can update the stored values in the server.
- In order to represent/render curves and surfaces, we will use vertex buffer objects.

Vertex buffer objects

Creating buffer objects for vertex data

- You can call glGenBuffers() to allocate buffer object identifiers.

```
/*
    Returns n currently unused names for buffer objects in the array buffers.
    The names returned in buffers do not have to form a contiguous set of integers.

    The names returned are marked as used for the purposes of allocating additional
    buffer objects, but only acquire a valid state once they have been found.

    Zero is a reserved buffer object name and is never returned as a buffer
    object by glGenBuffers().
*/
GLvoid glGenBuffers(GLsizei n, GLuint* buffers);
```

- You can also determine whether an identifier is a currently used buffer object identifier by calling glIsBuffer().

```
/*
    Returns GL_TRUE if buffer is the name of a buffer object that has been
    bound, but has not been subsequently deleted.

    Returns GL_FALSE if buffer is zero of if buffer is a nonzero value that
    is not the name of a buffer object.
*/

GLboolean glIsBuffer(GLuint buffer);
```

Vertex buffer objects

Creating buffer objects for vertex data

- You can call glGenBuffers() to allocate buffer object identifiers.

```
/*
    Returns n currently unused names for buffer objects in the array buffers.
    The names returned in buffers do not have to form a contiguous set of integers.

    The names returned are marked as used for the purposes of allocating additional
    buffer objects, but only acquire a valid state once they have been found.

    Zero is a reserved buffer object name and is never returned as a buffer
    object by glGenBuffers().
*/
GLvoid glGenBuffers(GLsizei n, GLuint* buffers);
```

- You can also determine whether an identifier is a currently used buffer object identifier by calling glIsBuffer().

```
/*
    Returns GL_TRUE if buffer is the name of a buffer object that has been
    bound, but has not been subsequently deleted.

    Returns GL_FALSE if buffer is zero of if buffer is a nonzero value that
    is not the name of a buffer object.
*/

GLboolean glIsBuffer(GLuint buffer);
```

Vertex buffer objects

Making a buffer object active

- To make a buffer object active, it needs to be bound.
- Binding selects which buffer object future operations will affect, either for initializing data or using that buffer for rendering.
- The corresponding command is glBindBuffer() and you will likely call it multiple times: once to initialize the object and its data, and then subsequent times either to select that object for use in rendering, or to update its data.

```
/*
    Specifies the current active buffer object.

    target must be set to either GL_ARRAY_BUFFER or GL_ELEMENT_ARRAY_BUFFER.

    buffer specifies the buffer object to be bound to.

    This function does 3 things.
    1)  When using buffer of an unsigned integer other than zero for the first time,
        a new buffer object is created and assigned that name.
    2)  When binding a previously created buffer object, that buffer object becomes the
        active buffer object.
    3)  When binding to a buffer value of zero, OpenGL stops using buffer objects.
*/

GLvoid glBindBuffer(GLenum target, GLuint buffer);
```

- To disable the use of buffer objects, call glBindBuffer() with zero as the buffer identifier.

Vertex buffer objects

Making a buffer object active

- To make a buffer object active, it needs to be bound.
- Binding selects which buffer object future operations will affect, either for initializing data or using that buffer for rendering.
- The corresponding command is glBindBuffer() and you will likely call it multiple times: once to initialize the object and its data, and then subsequent times either to select that object for use in rendering, or to update its data.

```
/*
    Specifies the current active buffer object.

    target must be set to either GL_ARRAY_BUFFER or GL_ELEMENT_ARRAY_BUFFER.

    buffer specifies the buffer object to be bound to.

    This function does 3 things.
    1)  When using buffer of an unsigned integer other than zero for the first time,
        a new buffer object is created and assigned that name.
    2)  When binding a previously created buffer object, that buffer object becomes the
        active buffer object.
    3)  When binding to a buffer value of zero, OpenGL stops using buffer objects.
*/

GLvoid glBindBuffer(GLenum target, GLuint buffer);
```

- To disable the use of buffer objects, call glBindBuffer() with zero as the buffer identifier.

Vertex buffer objects

Making a buffer object active

- To make a buffer object active, it needs to be bound.
- Binding selects which buffer object future operations will affect, either for initializing data or using that buffer for rendering.
- The corresponding command is glBindBuffer() and you will likely call it multiple times: once to initialize the object and its data, and then subsequent times either to select that object for use in rendering, or to update its data.

```
/*
    Specifies the current active buffer object.

    target must be set to either GL_ARRAY_BUFFER or GL_ELEMENT_ARRAY_BUFFER.

    buffer specifies the buffer object to be bound to.

    This function does 3 things.
    1)  When using buffer of an unsigned integer other than zero for the first time,
        a new buffer object is created and assigned that name.
    2)  When binding a previously created buffer object, that buffer object becomes the
        active buffer object.
    3)  When binding to a buffer value of zero, OpenGL stops using buffer objects.
*/

GLvoid glBindBuffer(GLenum target, GLuint buffer);
```

- To disable the use of buffer objects, call glBindBuffer() with zero as the buffer identifier.

Vertex buffer objects

Making a buffer object active

- To make a buffer object active, it needs to be bound.
- Binding selects which buffer object future operations will affect, either for initializing data or using that buffer for rendering.
- The corresponding command is glBindBuffer() and you will likely call it multiple times: once to initialize the object and its data, and then subsequent times either to select that object for use in rendering, or to update its data.

```
/*
    Specifies the current active buffer object.

    target must be set to either GL_ARRAY_BUFFER or GL_ELEMENT_ARRAY_BUFFER.

    buffer specifies the buffer object to be bound to.

    This function does 3 things.
    1)  When using buffer of an unsigned integer other than zero for the first time,
        a new buffer object is created and assigned that name.
    2)  When binding a previously created buffer object, that buffer object becomes the
        active buffer object.
    3)  When binding to a buffer value of zero, OpenGL stops using buffer objects.
*/

GLvoid glBindBuffer(GLenum target, GLuint buffer);
```

- To disable the use of buffer objects, call glBindBuffer() with zero as the buffer identifier.

Vertex buffer objects

Allocating and initializing buffer objects with vertex data

- Once you have bound a buffer object, you need to reserve space for storing your vertex data. This operation is done by calling glBufferData().

```
/*
    Allocates size storage units (usually bytes) of OpenGL server memory for storing vertex
    array data or indices. Any previous data associated with the currently bound object will
    be deleted.

    target may be either GL_ARRAY_BUFFER, for vertex data, or GL_ELEMENT_ARRAY_BUFFER, for
    index data.

    size is the amount of storage required for storing the respective data.
    This value is generally the number of elements in the data multiplied by their respective
    storage size.

    data is either a pointer to a client memory that is used to initialize the buffer object
    or NULL.
    If a valid pointer is passed, size units of storage are copied from the client to the server.
    If NULL is passed, size units of storage are reserved for use, but are left uninitialized.

    usage provides a hint as to how the data will be read and written after allocation.
    Valid values are:
        GL_STREAM_DRAW,  GL_STREAM_READ,  GL_STREAM_COPY,
        GL_STATIC_DRAW,  GL_STATIC_READ,  GL_STATIC_COPY,
        GL_DYNAMIC_DRAW, GL_DYNAMIC_READ, GL_DYNAMIC_COPY.

    glBufferData() will return GL_OUT_OF_MEMORY if the requested size exceeds what the server
    is able to allocate, or a GL_INVALID_VALUE if usage is not one of the permitted values.
*/

GLvoid glBufferData(GLenum target, GLsizeiptr size, const GLvoid* data, GLenum usage);
```

Vertex buffer objects

Updating data values in buffer objects, part I

- There are two methods for updating data stored in a buffer object.
- The first one assumes that you have data of the same type prepared in a buffer in your application. glBufferSubData() will replace some subset of the data in the bound buffer object with the data you provide.

```
/*
    Updates size bytes starting at offset (also measured in bytes) in the currently bound
    buffer object associated with target using the data pointed to by data.

    target must be either GL_ARRAY_BUFFER or GL_ELEMENT_ARRAY_BUFFER.

    The routine will generate a GL_INVALID_VALUE error if size is less than zero, or
    if size + offset is greater than the original size specified when the buffer object
    was created.
*/

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

Vertex buffer objects

Updating data values in buffer objects, part I

- There are two methods for updating data stored in a buffer object.
- The first one assumes that you have data of the same type prepared in a buffer in your application. glBufferSubData() will replace some subset of the data in the bound buffer object with the data you provide.

```
/*
    Updates size bytes starting at offset (also measured in bytes) in the currently bound
    buffer object associated with target using the data pointed to by data.

    target must be either GL_ARRAY_BUFFER or GL_ELEMENT_ARRAY_BUFFER.

    The routine will generate a GL_INVALID_VALUE error if size is less than zero, or
    if size + offset is greater than the original size specified when the buffer object
    was created.
*/

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

Vertex buffer objects

Updating data values in buffer objects, part II

- The second method allows you to update more selectively which data values are replaced. glMapBuffer() returns a pointer to the the buffer object memory into which you can write new values (or simply read the data, depending upon your choice of memory access permissions), just as if you were assigning values to an array. By calling glUnmapBuffer() you signify that you have completed updating the data.

```
/*
    Returns a pointer to the data storage for the currently bound buffer object associated
    with target, which may be either GL_ARRAY_BUFFER or GL_ELEMENT_ARRAY_BUFFER.

    access must be either GL_READ_ONLY, GL_WRITE_ONLY, or GL_READ_WRITE, indicating the
    operations that a client may do on the data.

    glMapBuffer() will return NULL either if the buffer cannot be mapped (setting the OpenGL
    error state to GL_OUT_OF_MEMORY) or if the buffer was already mapped previously (when the
    OpenGL error state will be set to GL_INVALID_OPERATION).
*/

GLvoid* glMapBuffer(GLenum target, GLenum access);

/*
    Indicates that updates to the currently bound buffer object are complete, and the buffer
    may be released.

    target must be either GL_ARRAY_BUFFER or GL_ELEMENT_ARRAY_BUFFER.
*/

GLboolean glUnmapBuffer(GLenum target);
```

Vertex buffer objects

Updating data values in buffer objects, part III

Example for mapping and unmapping vertex buffer objects

```
GLfloat* data = (GLfloat*) glMapBuffer(GL_ARRAY_BUFFER, GL_READ_WRITE);

if (data != (GLfloat*)0)
{
    // modify third components, e.g. z values of vectors
    for (GLint i = 0; i < ...; ++i)
        data[3 * i + 2] *= rand() / (GLfloat) RAND_MAX;

    glUnmapBuffer(GL_ARRAY_BUFFER);
}
else
{
    // error: not being able to update data
}
```

Vertex buffer objects
Deleting buffer objects

- When you do not need a buffer object, you can release its resources and make its identifier available by calling glDeleteBuffers(). Any bindings to currently bound objects that are deleted are reset to zero.

```
/*
    Deletes n buffer objects, named by elements in the array buffers.
    The freed buffer objects may be reused (e.g. by glGenBuffers()).

    If a buffer object is deleted while bound, all bindings to that object
    are reset to default buffer object, as if glBindBuffer() had been called
    with zero as the specified buffer object.

    Attempts to delete nonexistent buffer objects or the buffer object named zero
    are ignored without generating an error.
*/

GLvoid glDeleteBuffers(GLsizei n, const GLuint* buffers);
```