This document contains an introduction to Modern OpenGL and some of the basics that you need to get started.

Table of Contents

Agenda:

1) What is OpenGL?
2) Initializing OpenGL.
3) Prepare your Environment.
4) First look on an OpenGL program.
5) OpenGL Syntax.
6) OpenGL Rendering Pipeline.
7) Primitive Drawing

Content:

What is OpenGL?

1) OpenGL is an API which is a merely software library for accessing features in graphics hardware.
2) OpenGL is designed as a streamlined, **hardware-independent** interface that can be implemented on many different types of graphics hardware systems, or entirely in **software independent** of a computer's OS or windowing software.
3) So you'll notice that **OpenGL doesn't contain functions that perform windowing tasks** (such as creating windows and specifying the window's size and title…).
4) OpenGL is implemented as **a client-server system**, try to think of it as if your application is the client, and the OpenGL implementation provided by the graphics card manufacturer is the server. So, the client will issue commands which will be executed by the server.

OpenGL Initialization

- OpenGL comes with the system.
- You will need to ensure that you have **downloaded** and installed a **recent driver for your graphics hardware**.
- To know the supported version of OpenGL by your graphics card try this program:
  http://www.realtech-vr.com/glview/index.html
- There are two phases for initialization:
  1. **Creating context**: It is like creating a window in an application. GLUT, FreeGLUT and GLFW are libraries for creating OpenGL context. We're going to use **GLFW**.
  2. **Getting functions**: For loading all available OpenGL extensions and functions automatically. We're going to use **GLEW** library.

Preparing the Development Environment

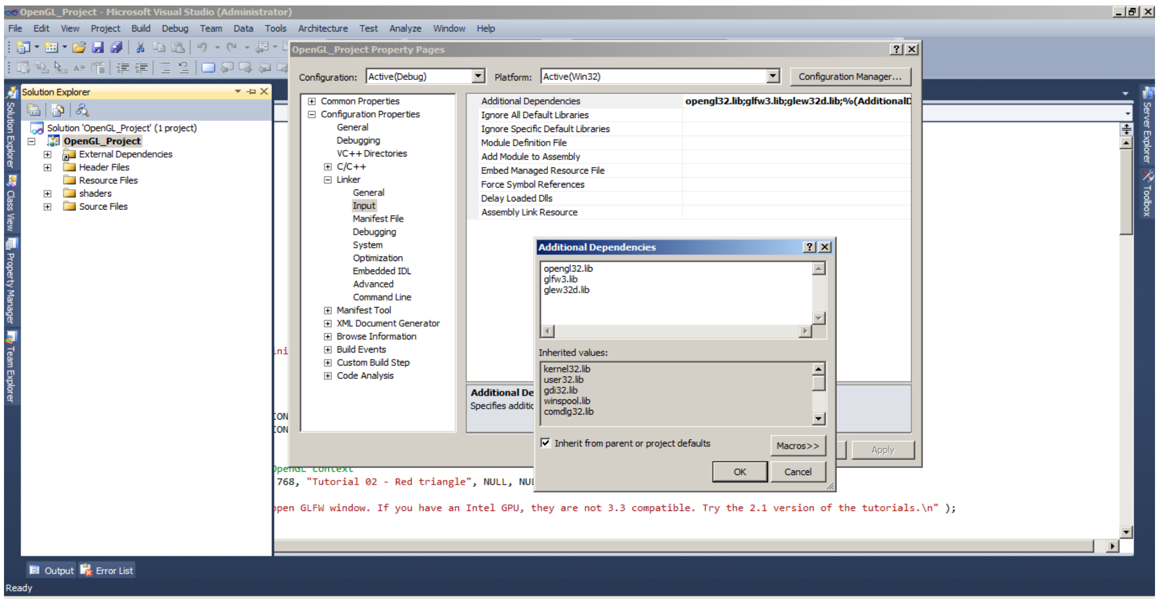
In this section, we describe how to prepare the development environment that we will use to create OpenGL programs. The steps described in this section need to be done only once on a given computer system to build OpenGL programs. You do not need to redo them every time you create a new OpenGL program.

We will be developing our OpenGL programs in C++ using the Visual Studio IDE. So, make sure that you have Visual Studio installed on your computer along with Visual C++.

The next step is to install the GLFW library. **GLFW** is an Open Source, multi-platform library for **creating windows** with OpenGL contexts and receiving **input** and **events**.

1. Let's assume that Visual Studio was installed at the directory "C:\Program File s (x86)\Microsoft Visual Studio 10.0\".
2. Locate **GLFW** library files as follows:
   - Put *glfw3.h* inside "C:\ Program File s (x86)\Microsoft Visual Studio 10.0\VC\include\gl\" (you may need to create the directory **gl** yourself).
   - Put *glfw3.lib* inside "C:\Program Files\Microsoft Visual Studio 10.0\VC\lib\".
   - **Windows 32-Bit Users**: Put *glfw3.dll* inside "C:\Windows\System32\".
   - **Windows 64-Bit Users**: Put *glfw3.dll* inside "C:\Windows\SysWOW64\".
3. Locate GLEW library files as step **2**.
4. Starting with a new empty project, you will need to do the following step to get OpenGL working:
- Specify the following lib files as additional linker inputs: *opengl32.lib, glfw3.lib, glew32.lib*. This can be performed by opening the properties of your project, and writing the names of the aforementioned

files in the field *Configuration Properties>Linker>Input>Additional Dependencies*. This is indicated in the following figure.


OpenGL_Project Property Pages

Configuration: Active(Debug) Platform: Active(Win32)

Additional Dependencies: opengl32.lib;glfw3.lib;glew32d.lib;%(AdditionalD

Additional Dependencies

```
opengl32.lib
glfw3.lib
glew32d.lib
```

Inherited values:

```
kernel32.lib
user32.lib
gdi32.lib
winspool.lib
comdlg32.lib
```

☑ Inherit from parent or project defaults

```
768, "Tutorial 02 - Red triangle", NULL, NU
pen GLFW window. If you have an Intel GPU, they are not 3.3 compatible. Try the 2.1 version of the tutorials.\n" );
```

Some terminology

1) **Rendering:** the process by which a computer creates an image from models.
2) **Models or Objects:** are constructed from geometric primitives (points, lines, triangles) that are specified by their vertices.
3) **Vertex:** the position of a point in space (note that a vertex has no size), can be 2D or 3D.
4) **Fragment:** for now we can loosely consider the fragment as a screen pixel. Although it's more complicated than that.
5) **Primitive:** it is a way that OpenGL interprets vertex streams, representing them as triangles, lines, points and so on.
6) **OpenGL context:** A context stores all of the state associated with this instance of OpenGL. A process (program) can have multiple contexts. In other words **we can consider each context to represent a separate viewable surface, like a window in an application.**
7) **Shaders:** simply put, shaders are special functions that are executed by the graphics card. Or we can say that shaders are little programs that are specifically compiled for your GPU. The OpenGL provided by your GPU manufacturer includes the compiler tools that will take the shader's source code and create the code that your GPU needs to execute. In OpenGL there are four shader stages (we will explain them later). The most common are, **vertex shaders**, which process the vertex data (ie: vertex positions, projection), and **fragment shaders**, which operates on fragments generated by the rasterizer (ie: pixel colors) (again, we will explain shaders in depth later). The point is BOTH vertex and fragment shaders are required in every OpenGL program.
8) **Buffer object:** is an object that holds a chunk (array) of unformatted memory and other attributes related to it, the OpenGL context manages the buffer object, which can store the vertex data, pixel data from images and a variety of other things. (ex: frame buffer).
9) **Framebuffer:** is a chunk of memory that the graphics hardware manages, which contains the pixels to be displayed on your monitor, and the graphics card feeds it to your monitor.
10) **Shader plumbing:** associating the data in the application (vertex specs, and vertex pos), with variables in the shader programs.

First look on an OpenGL program

1) Since you can do so many things with OpenGL. An OpenGL program can be very large and complicated. However, **the basic structure** of all OpenGL applications usually goes as follows:
    a. Initialize the ***state*** associated with **how the objects should be rendered** (drawn).
    b. Specify those **objects**.

OpenGL Syntax

1) All OpenGL library functions begin with the letters “gl”, followed by one or more words. (ex: glBindVertexArray())
2) **Functions:** Any functions starting with “glut”, “glew” or “glfw” are from other libraries that handle other things like: windowing, and user’s input.
3) **Constants:** Similar to the function-naming convention, constants start with “GL_”, and we use underscores to separate words. (Ex: GL_COLOR_BUFFER_BIT).
4) **Datatypes:** to aid the cross-platform ideology, OpenGL defines various types of data. (Ex: GLfloat) which is a floating-point value type can be used for declaring vertices for instance.

Function overloading: Since OpenGL is written in C, so it doesn’t support function overloading. To overcome this, you will find functions in numerous forms, such as **glUniform*(),** can be found as **glUniform2f()** and **glUniform3fv()**. The suffixes at the end provide information about the arguments passed to the function. The following table illustrates the suffixes meaning:

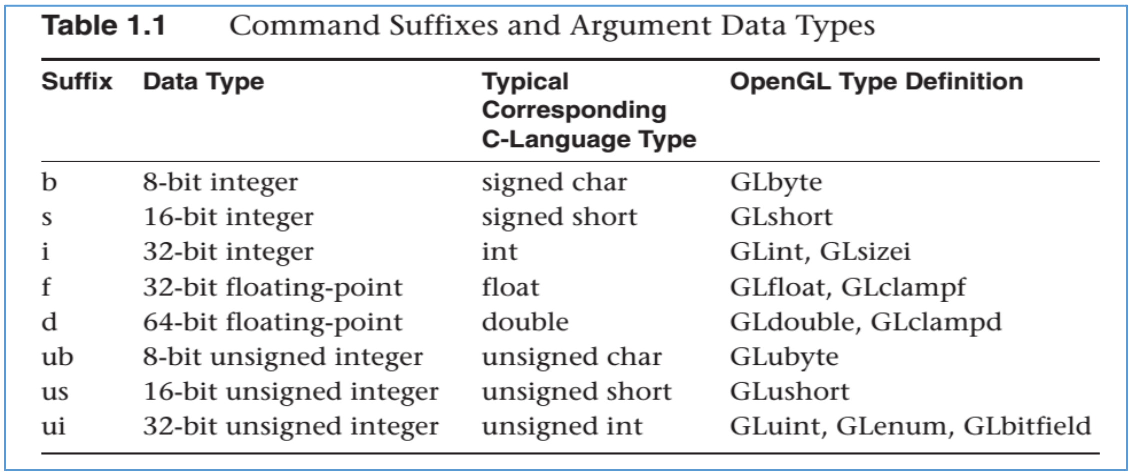


Table 1.1 Command Suffixes and Argument Data Types

Suffix	Data Type	Typical Corresponding C-Language Type	OpenGL Type Definition
b	8-bit integer	signed char	GLbyte
s	16-bit integer	signed short	GLshort
i	32-bit integer	int	GLint, GLsizei
f	32-bit floating-point	float	GLfloat, GLclampf
d	64-bit floating-point	double	GLdouble, GLclampd
ub	8-bit unsigned integer	unsigned char	GLubyte
us	16-bit unsigned integer	unsigned short	GLushort
ui	32-bit unsigned integer	unsigned int	GLuint, GLenum, GLbitfield

Rendering Pipeline

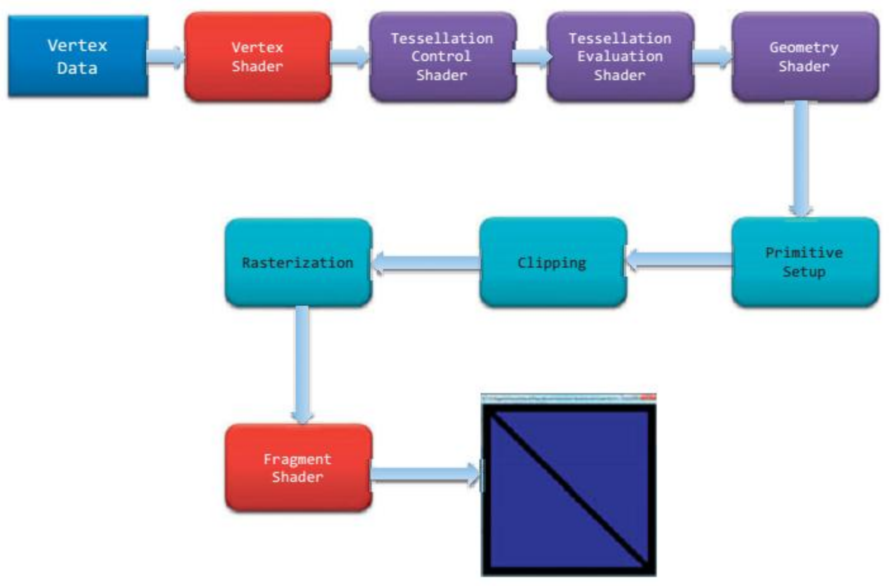


Figure 1.2 The OpenGL pipeline

The rendering pipeline is a **sequence of processing stages for converting the data your application provides to OpenGL into a final rendered image**. The pipeline has changed very much throughout different OpenGL versions. Your application's pipeline will be like the following:

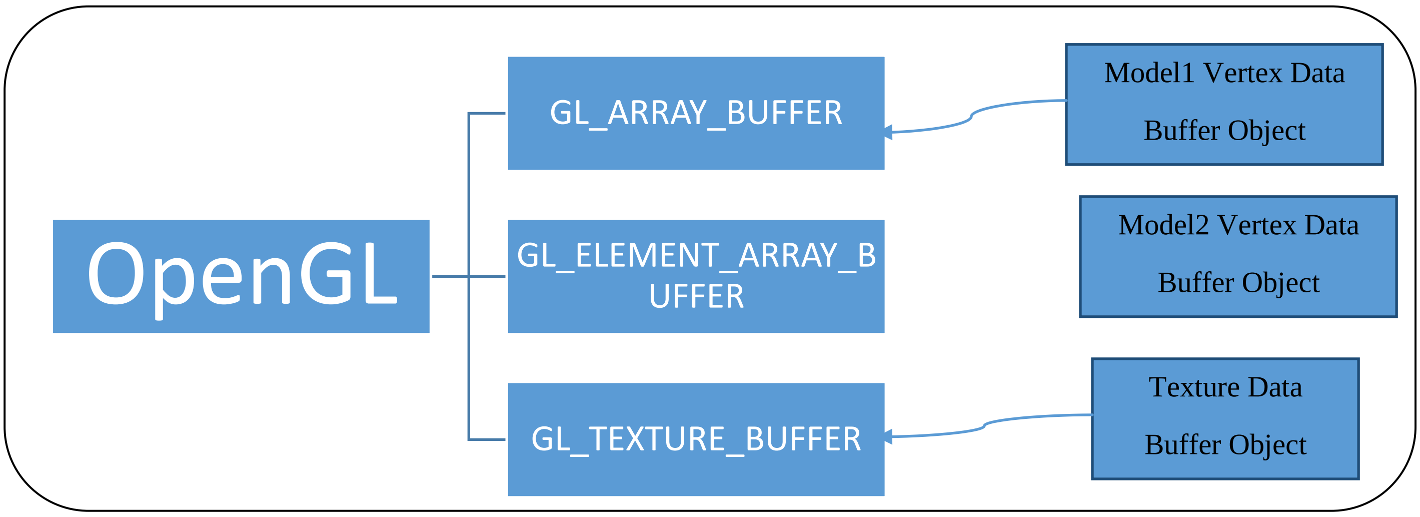
1) **Preparing data:**
    i. <u>OpenGL requires that all data be stored in buffer objects</u>. Filling these buffers with data can happen in different ways. (discussed later)
2) **Sending data to OpenGL:**
    i. After initializing our buffers, we can request rendering our primitives by calling OpenGL's drawing commands. (discussed later)
    ii. <u>The vertex data is sent to the OpenGL server, to start the pipeline.</u>
3) **The programmable rendering pipeline: [**the illustration is for OpenGL 4.3 rendering pipeline]
    a. Vertex Shading:
        i. For each vertex that is issued by a drawing command, a vertex shader will be called to process the data associated with that vertex. Depending on the application, vertex shaders may be very simple, perhaps just copying the data to pass it through to the next shading stage, we will call that a **pass-through shader**. Or it can be a very complex shader that performs many computations (do transformations, or determine the vertex color using lighting computations for example)
        ii. Usually in most applications you will need no more than one vertex shader, but note that **only one vertex shader can be active at any moment in time.**
    b. **Tessellation Shading:**
        i. After the vertex shader processed the vertex data. The tessellation subdivides the vertex data into small primitives (ex: lines, triangles) called **patches** to describe the object's shape.
        ii. We are not interested in them in this course.
    c. **Geometry Shading:**
        i. It is an optional shader which allows additional processing of individual primitives.
        ii. We are not interested in it in this course.
    d. **Primitive Assembly:**
        i. Organize the vertices into their associated primitives to be passed for clipping and rasterization.
    e. **Clipping:**
        i. The primitives outside/inside/in and out, the viewport (window) are clipped based on their position and type. (clipping will be discussed later in the lectures)
        ii. This operation is handled automatically by OpenGL.
    f. **Rasterization:**
        i. The updated (clipped) primitives are sent to the rasterizer for fragment generation. You can consider a fragment as a "candidate pixel", or a pixel that we may or may not draw in the framebuffer depending on the next two stages.
    g. **Fragment Shading:**
        i. Is the stage in which you have programmable control over the color of a screen location (certain pixel), and potentially its depth value. Fragment shaders are very powerful as they often employ texture mapping to augment the colors provided by the vertex processing stages.
    h. **Per Fragment Operations:**
        i. During this stage a fragment's visibility is determined using depth testing aka z-buffering (will be discussed later).
        ii. After that the fragment may be written directly to the framebuffer, which will be passed to the display device.

What are Buffer Objects (BO) and Binding Buffers

- Remember that OpenGL requires that all data for all primitive data(lines, triangles,… etc.) to be stored in buffer objects.
- Try to think of OpenGL as a box with some ports in that box and you want to connect a certain Buffer Object to one of these ports. The connecting process is called ***buffer binding***.
- In the following figure you can see that we can have many buffer objects that can represent different things.
- Also you can **note that model1 is the currently bound BO**, this means when the OpenGL will start drawing or painting, the GPU will use the vertex data of model1 from the BO currently bound to GL_ARRAY_BUFFER.
    - If you noticed, only model 1 will be drawn now, so what do we do?
    - It is simple, first we bind model1 draw it, then bind model 2 and draw it.
    - Example: binding model1 vertex data,

```
// glBindBuffer(Glenum target, GLuint buffer)

glBindBuffer(GL_ARRAY_BUFFER, model1BufferObject);
```


OpenGL
GL_ARRAY_BUFFER
GL_ELEMENT_ARRAY_BUFFER
GL_TEXTURE_BUFFER
Model1 Vertex Data Buffer Object
Model2 Vertex Data Buffer Object
Texture Data Buffer Object

- In the figure, the GPU will use the texture data (will be discussed later) bound to the GL_TEXTURE_BUFFER.

OpenGL Syntax related Notes

1) In order to allocate some memory (array) in OpenGL you need two steps:
    a. Generate the name using one of the **glGen*** functions, these functions return the currently unused names of the object to be allocated (it is something similar to creating pointer-type in C), **note that you have reserved the name** (or created the pointer in C) **but you haven't actually allocated the memory yet.**
    b. **The allocation** process is called **binding an object**, and is done by a collection of functions that have the form **glBind***. When you bind an object for the first time, OpenGL will internally allocate the memory it needs and make that object *current*, which means that any operations relevant to the bound object, will affect its state. Also, you can think of binding an object like setting a track switch in a railroad yard. Once a track switch has been set, all trains go down that set of tracks.
2) Make sure that you free the allocated memory using **glDelete***.
3) To load data into a buffer object after initialization and binding, we use **glBufferData()** which does two things:
    a. Allocate a memory to hold the data.
    b. Copy the data from the array to the OpenGL server's memory.
4) Normalized-Device Coordinates (NDC), OpenGL works in a coordinate space from [-1,1]. While it may look like a limitation, later we will handle this and discuss the mathematics required.

Vertex-Data (Attributes)

1) A vertex can have some characteristics that describe it, which can be called **attributes**.
2) These **attributes can be** for example vertex **position**, vertex **color**, vertex **normal** (for lighting).
3) Vertex data (the array that holds the vertices' attributes) can contain one or more attributes.

4) Example for a triangle with 1 attribute (2D vertices' positions):

```
GLfloat verts[] =
{
        +0.0f, +1.0f,
        -1.0f, -1.0f,
        +1.0f, -1.0f
};
```

5) Example for a red triangle with 2 attributes (3D vertices' positions, and RGB color):

```
GLfloat verts[] =
{
        +0.0f, +1.0f, +0.0f, 1.0f, 0.0f, 0.0f, // Red vertex
        -1.0f, -1.0f, +0.0f, 0.0f, 1.0f, 0.0f, // Green vertex
        +1.0f, -1.0f, +0.0f, 0.0f, 0.0f, 1.0f  // Blue vertex
};
```

6) Note that you have to tell OpenGL how you organized your attributes in the array. (discussed later)

How to pass vertex data to OpenGL using Vertex-Buffer Objects (VBO)

To pass your vertices (sometimes called vertex data, because it might contain more information such as vertices normal and colors for example) to OpenGL you will need two steps:

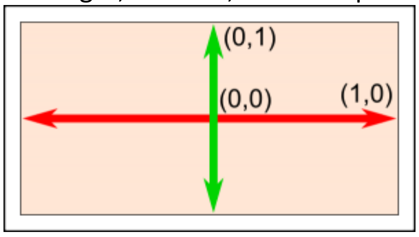
1) During Initialization:
    1. Create an **array** and write your **vertices** in it, or load it from a file. (the array is allocated in RAM)
    2. **Create a Buffer Object** name (it's just a number that represents the ID of the BO not a string, but it is called *name* in OpenGL references).
    3. **Reserve a BO** name.
    4. **Bind** (allocate if it is the first time to use this name) the reserved name.
    5. **Allocate the needed space for the vertex data**, and pass it to the OpenGL (**now the vertex data is in the GPU's memory**).
    6. **Enable the desired attributes** (see the previous section).
    7. **Describe** the vertex data **format**.
2) During Drawing:
    8. Will give the OpenGL the command to draw, by specifying the type of primitive, the vertex data represents. And the number of vertices will be used.

Shaders & OpenGL Shading Language (GLSL)

1) Every OpenGL program that uses OpenGL 3.1 or greater must provide at least two shaders: a vertex shader and a fragment shader.
2) For us at this point, a **shader** is a **small function written in OpenGL Shading Language** (aka. GLSL), **which is a special language similar to C++ for constructing OpenGL shaders that run on the GPU.**
3) Your GLSL shader is provided to OpenGL as a string of characters.
4) The process of reading the shader files and preparing them for OpenGL is done in **LoadShaders()** function, for simplification we will not go into it.
5) More on shaders and GLSL will be discussed later.

Screen coordinates

The screen origin is in the middle, X is on the right, as usual, and Y is up. This is what it gives on a wide screen:



This is something built in your graphics card. So (-1,-1) is the bottom left corner of your screen. (1,-1) is the bottom right, and (0,1) is the middle top.

Example: (Drawing a Triangle)

1) Initialization:

```
////////////////////////////////////////////////
// 1) create the data array (stored in RAM)
GLfloat verts[] =
{
   +0.0f, +1.0f, +0.0f,
   -1.0f, -1.0f, +0.0f,
   +1.0f, -1.0f, +0.0f
};
////////////////////////////////////////////////
// 2) create a buffer object name(ID) holder.
GLuint myBufferID;
////////////////////////////////////////////////
// 3) reserve/generate a buffer object name(ID).
// void glGenBuffers(GLsizei n, GLuint * buffers);
// n: number of names to be generated. (you can generate more than one name)
// buffers: names generated.
glGenBuffers(1, &myBufferID);
////////////////////////////////////////////////
// 4) set myBufferID as the current GL_ARRAY_BUFFER.
//   note that since this is the first time we bind myBufferID,
//   in this step OpenGL will both allocate and bind the buffer object.
// void glBindBuffer(GLenum target,GLuint buffer);
// target: Specifies the target to which the buffer object is bound.
// buffer: Specifies the name of a buffer object.
glBindBuffer(GL_ARRAY_BUFFER, myBufferID);
////////////////////////////////////////////////
// 5) allocate the mem in the GPU and copy the data from the RAM to the GPU,
//   in the current GL_ARRAY_BUFFER, which in our case is myBufferID.
// void glBufferData(GLenum target,GLsizeiptr size,
//                    const GLvoid * data,GLenum usage);
// target: Specifies the target buffer object.
// size:  Specifies the size in bytes of the buffer object's new data store.
// data:  Specifies a pointer to data that will be copied into the data store for
initialization, or NULL if no data is to be copied.
// usage: Specifies the expected usage pattern of the data store.
// Usage can make performance difference, because it affects the place where the data is
allocated,
// but since we won't change the data so we can make it GL_STATIC_DRAW.
glBufferData(GL_ARRAY_BUFFER, sizeof(verts), verts,GL_STATIC_DRAW);
////////////////////////////////////////////////
//6) enable the desired attributes. (Please go to this section for more information about
vertex attributes:
//   the attributes are 0 indexed, and here we have only one attribute.
// void glEnableVertexAttribArray( GLuint index);
glEnableVertexAttribArray(0);
////////////////////////////////////////////////
//7) specify the vertex-data format.
// void glVertexAttribPointer(
// GLuint index, (the index of the attribute you are describing)
// GLint size,          (the number of elements in that attribute)
// GLenum type,  (the type of each element in that attribute)
// GLboolean normalized, (do you want to normalize the data?)
// GLsizei stride,      (the offset between each instance of that attribute)
// const GLvoid * pointer (the offset of the first component of the first instance of the
attribute) );
glVertexAttribPointer(0,3,GL_FLOAT,GL_FALSE,0,0);
```

2) Drawing:

```
// void glDrawArrays(GLenum  mode, GLint  first, GLsizei  count);
// note that glDrawArrays, uses the currently bound BO in GL_ARRAY_BUFFER.
glDrawArrays(GL_TRIANGLES,0,3);
```

Multiple Triangles

We will discuss two ways for drawing multiple objects:

1. Using a single VBO.
2. Using element array buffers (VBO Indexing).
3. Using multiple VBOs.

1. Using a single VBO

- If we wanted to draw more than one triangle with single VBO, there should be two differences:
  a. The vertices positions of course.

```
GLfloat verts[] =
{
	+0.0f, +0.0f, +0.0f,
	+1.0f, +1.0f, +0.0f,
	-1.0f, +1.0f, +0.0f,

	+0.0f, +0.0f, +0.0f,
	-1.0f, -1.0f, +0.0f,
	+1.0f, -1.0f, +0.0f
};
```

  b. The glDrawArrays function.

```
glDrawArrays(GL_TRIANGLES,0,6); // 6 because we have 6 vertices.
```

2. Using Element Array Buffers (VBO Indexing)

- But notice that the vertices 0 and 3 are the same vertex (in the previous example). Which can be **considered a redundancy**, imagine if a vertex is shared between many triangles, then we will have to repeat it for each triangle.
- In order to **optimize** our vertex data, **we will describe our triangle** not only with vertices but **with indices** as well:
  a. We will remove the duplicated vertex, so now we only have 5 unique vertices, indexed from 0 to 4.

```
GLfloat verts[] =
{
	+0.0f, +0.0f, +0.0f,
	+1.0f, +1.0f, +0.0f,
	-1.0f, +1.0f, +0.0f,
	-1.0f, -1.0f, +0.0f,
	+1.0f, -1.0f, +0.0f
};
```

  b. Now we want to describe the **connectivity** between those vertices, **using something called the element array buffer**, which contains the indices of the vertices making our triangles. We can simply imagine that it will be something like this [0,1,2,0,3,4]. As you can see the first 3 indices will make the first triangle, and the second 3 indices will make the second triangle.

```
GLushort indices[] = {0,1,2,0,3,4}; //unsigned short to save memory.
```

  c. Now we need to create a new buffer object in the GPU and send the indices to it.

```
GLuint myIndicesBufferID;
glGenBuffers(1, &myIndicesBufferID);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, myIndicesBufferID);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
```

  d. In the drawing function, instead of calling glDrawArrays, we will call glDrawElements.

```
glDrawElements(GL_TRIANGLES, 0, 6, GL_UNSIGNED_SHORT, 0);
```

3. Using multiple VBOs

1) We want to see what should we do if we want to draw more than one model (in our case a triangle and square), and each model has its own data array. We need to create **four** BOs, two for each model, one for the vertices and one for the indices for each model.
2) Now let's dive into the code and explain it after:
   a. Initialization:

```
//create the triangle BOs.
GLuint triangleVertexBufferID;
GLuint triangleIndexBufferID;
glGenBuffers(1, &triangleVertexBufferID);
glBindBuffer(GL_ARRAY_BUFFER,triangleVertexBufferID);
glBufferData(GL_ARRAY_BUFFER, numberOfTriVertices*sizeof(GLfloat),
triangleVertexData,GL_STATIC_DRAW);

glGenBuffers(1, &triangleIndexBufferID);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,triangleIndexBufferID);

glBufferData(GL_ELEMENT_ARRAY_BUFFER, numberOfTriIndices*sizeof(GLushort),
triangleIndexData,GL_STATIC_DRAW);

glEnableVertexAttribArray(0);

//create the square's BOs
GLuint squareVertexBufferID;
GLuint squareIndexBufferID;
glGenBuffers(1, &squareVertexBufferID);
glBindBuffer(GL_ARRAY_BUFFER,squareVertexBufferID);
glBufferData(GL_ARRAY_BUFFER, numberOfSqrVertices*sizeof(GLfloat),
squareVertexData,GL_STATIC_DRAW);

glGenBuffers(1, &squareIndexBufferID);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,squareIndexBufferID);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, numberOfSqrIndices*sizeof(GLushort),
squareIndexData,GL_STATIC_DRAW);
```

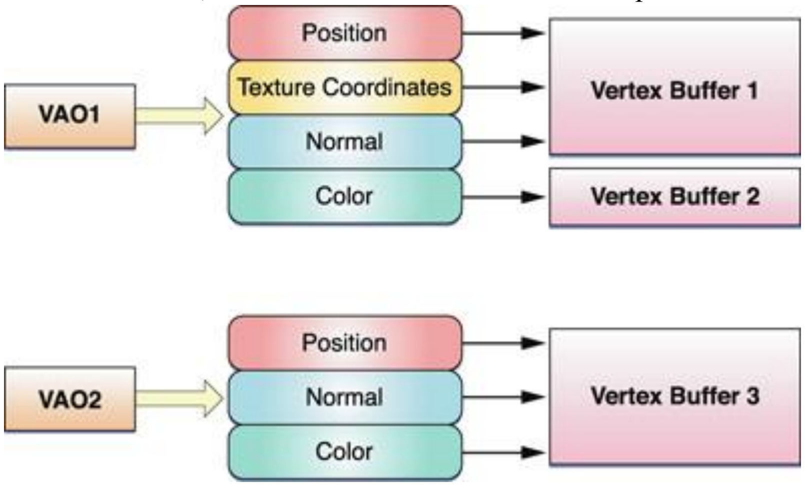
   b. Drawing:

```
//draw the currently bound data in GL_ARRAY_BUFFER, and GL_ELEMENT_ARRAY_BUFFER
glBindBuffer(GL_ARRAY_BUFFER,triangleVertexBufferID);
glVertexAttribPointer(0,2,GL_FLOAT,GL_FALSE,0,0);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,triangleIndexBufferID);
glDrawElements(GL_TRIANGLES, numberOfTriIndices,GL_UNSIGNED_SHORT,0);

glBindBuffer(GL_ARRAY_BUFFER,squareVertexBufferID);
glVertexAttribPointer(0,2,GL_FLOAT,GL_FALSE,0,0);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,squareIndexBufferID);
glDrawElements(GL_TRIANGLES, numberOfSqrIndices,GL_UNSIGNED_SHORT,0);
```

Optimizing the use of multiple VBOs using Vertex Array Objects (VAOs)

- If you noticed the drawing part of the previous example, you will note that we need to re-specify the vertex attributes for each object in every frame, and all that code while we only have one attribute. Imagine if we have 10 attributes, how the code will look.
- Here is where Vertex Arrays comes into place, the name might be confusing but vertex arrays are different than vertex-data. Since OpenGL 3 Core, they are compulsory, but you may use only one and modify it permanently
- Vertex Array Objects basically track all the state that is defined while binding the buffers and the vertexattrib calls, so we won't have to feed it to the OpenGL each frame.



- The question is, why do we use VAOs? Using multiple VAOs and spreading the data on multiple buffers causes a substantial performance difference. Check this interesting article http://www.openglsuperbible.com/2013/12/09/vertex-array-performance/

- The changes in our code:
    a. Initialization: note that we’ve moved the enabling and specifying the attributes part after binding the VAO.

```
//create the triangle BOs.
	GLuint triangleVertexBufferID;
	GLuint triangleIndexBufferID;
	glGenBuffers(1, &triangleVertexBufferID);
	glBindBuffer(GL_ARRAY_BUFFER,triangleVertexBufferID);
	glBufferData(GL_ARRAY_BUFFER, numberOfTriVertices*sizeof(GLfloat), 
triangleVertexData,GL_STATIC_DRAW);

	glGenBuffers(1, &triangleIndexBufferID);
	glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,triangleIndexBufferID);
	glBufferData(GL_ELEMENT_ARRAY_BUFFER, numberOfTriIndices*sizeof(GLushort), 
triangleIndexData,GL_STATIC_DRAW);

	//create the square's BOs
	GLuint squareVertexBufferID;
	GLuint squareIndexBufferID;
	glGenBuffers(1, &squareVertexBufferID);
	glBindBuffer(GL_ARRAY_BUFFER,squareVertexBufferID);
	glBufferData(GL_ARRAY_BUFFER, numberOfSqrVertices*sizeof(GLfloat), 
squareVertexData,GL_STATIC_DRAW);

	glGenBuffers(1, &squareIndexBufferID);
	glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,squareIndexBufferID);
	glBufferData(GL_ELEMENT_ARRAY_BUFFER, numberOfSqrIndices*sizeof(GLushort), 
squareIndexData,GL_STATIC_DRAW);

	//////////////////////////////////////////////////////////////////////////
	//vertex arrays.
	GLuint triangleVertexArrayObjectID;
	GLuint squareVertexArrayObjectID;

	//this will reserve one vertex array object
	glGenVertexArrays(1, &triangleVertexArrayObjectID);
	glGenVertexArrays(1, &squareVertexArrayObjectID);

	// allocate and bind the VAO.
	glBindVertexArray(triangleVertexArrayObjectID);
	//at this point, all the coming bind buffers and vertex attribs, will be stored in 
the triangleVertexArrayObjectID
	glEnableVertexAttribArray(0);
	glBindBuffer(GL_ARRAY_BUFFER,triangleVertexBufferID);
	glVertexAttribPointer(0,2,GL_FLOAT,GL_FALSE,0,0);
	glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,triangleIndexBufferID);

	glBindVertexArray(squareVertexArrayObjectID);
	glEnableVertexAttribArray(0);
	glBindBuffer(GL_ARRAY_BUFFER,squareVertexBufferID);
	glVertexAttribPointer(0,2,GL_FLOAT,GL_FALSE,0,0);
	glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,squareIndexBufferID);
```

    b. Drawing:

```
//drawing
//setup transformations if exist.

glBindVertexArray(triangleVertexArrayObjectID);
glDrawElements(GL_TRIANGLES, numberOfTriIndices,GL_UNSIGNED_SHORT,0);

glBindVertexArray(squareVertexArrayObjectID);
glDrawElements(GL_TRIANGLES, numberOfSqrIndices,GL_UNSIGNED_SHORT,0);
```