OpenGL Essentials

Context, Objects & Primitives

Francesco Andreussi

Bauhaus–Universität Weimar

28 November 2019

Bauhaus-Universität Weimar

Faculty of Media

OpenGL Context (1)

The context represents an OpenGL **instance** and its **state**, holding all the related attributes. The OpenGL **commands** can be executed **only** in the **current** context, affecting its **state**.

The current context is a **thread–local variable**, then a **process** can have multiple threads and, hence, **multiple current contexts**, but the same context cannot be the current in different threads.

A context can be associated with some OpenGL **objects** through the `glBind` operation. These objects are context–dependent, but can be shared with other contexts.

There are two ways to modify the state of a context, we can call some dedicated OpenGL commands (*direct*) or change one (or more) of the bound object(s) (*indirect*).

When the context is **destroyed**, **OpenGL is closed**.

OpenGL Context (2)

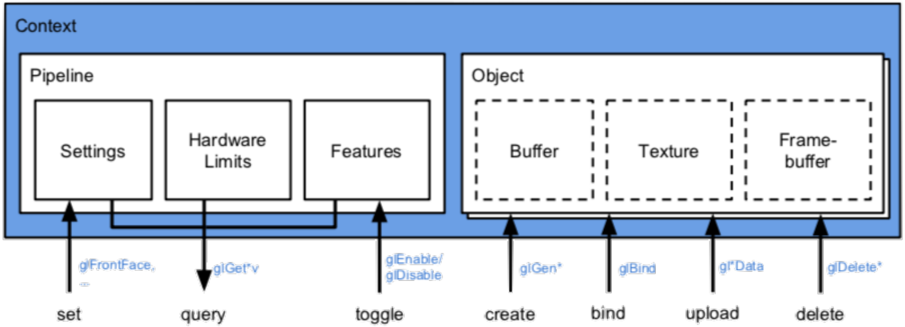


Fig. 1: OpenGL Context Scheme

OpenGL Objects (1)

Definition and Main Properties

An OpenGL Object is an OpenGL construct that contains some state. When they are bound to the context, the state that they contain is mapped into the context's state.

Thus, changes to context state will be stored in this object, and functions that act on this context state will use the state stored in the object.

[OpenGL Wiki]

There is **NO direct access** to the objects, only their handles (IDs) are visible to the user. These IDs are used to bind the objects to the *binding points* of the context.

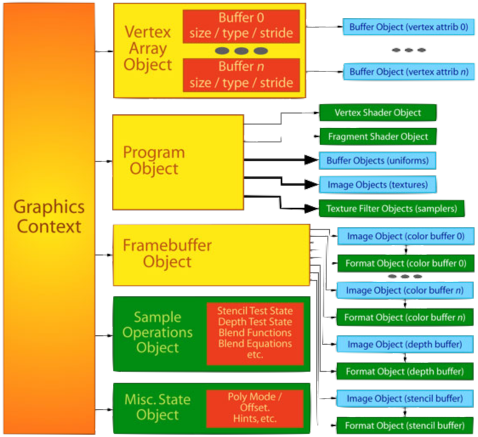
Each binding point supports a specific kind of object. A binding operation **erases** every previous bound on the specified binding point.

State manipulations affect an object **iff** this object is currently bound to a binding point.

OpenGL Objects (2)

Objects Hierarchy Overview

The objects can be grouped by category and interact with each other (within the same context).



Graphics Context
Vertex Array Object
Buffer 0 size / type / stride
Buffer n size / type / stride
Buffer Object (vertex attrib 0)
Buffer Object (vertex attrib n)
Program Object
Vertex Shader Object
Fragment Shader Object
Buffer Objects (uniforms)
Image Objects (textures)
Texture Filter Objects (samplers)
Framebuffer Object
Image Object (color buffer 0)
Format Object (color buffer 0)
Image Object (color buffer n)
Format Object (color buffer n)
Image Object (depth buffer)
Format Object (depth buffer)
Image Object (stencil buffer)
Format Object (stencil buffer)
Sample Operations Object
Stencil Test State
Depth Test State
Blend Functions
Blend Equations
etc.
Misc. State Object
Poly Mode / Offset.
Hints, etc.

Fig. 2: OpenGL Objects Hierarchy

OpenGL Objects (3)

VAOs, VBOs & IBOs

A **Vertex Array Object** is an OpenGL Object with all the **states** necessary for rendering. It encapsulates and organises the vertex data, but does not really allocate memory. It cannot be shared between different contexts.

A **Vertex Buffer Object** is a Buffer Object that is used as source data for Vertex Array Objects, it is bound to `GL_ARRAY_BUFFER` and carries all the vertex information (e.g., position, normal, colour...). It is necessary for sending the vertex data from the RAM to the VRAM of the GPU for processing.

A **Index Buffer Object** is a (optional) Buffer Object used for defining a reading order for the vertices. It is bound to a `GL_ELEMENT_ARRAY_BUFFER`.

Vertex Specification

A "generic" `initializeGeometry()` function

Initialise Vertex Array Object

```
glGenVertexArrays(1, &vao_handle)
glBindVertexArray(vao_handle)
```

Initialise Vertex Buffer Object and load data

```
glGenBuffer(1, &vbo_handle)
glBindBuffer(GL_ARRAY_BUFFER, vbo_handle)
glBufferData(GL_ARRAY_BUFFER, size, dataptr, usage)
```

Specify (activate, connect and set format) the Attributes

```
glEnableVertexAttribArray(index)
glVertexAttribPointer(index, size, dtype, norm, str, ptr)
```

Define Vertex Indices (optional)

```
glGenBuffers(1, &ibo_handle)
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo_handle)
glBufferData(GL_ELEMENT_ARRAY_BUFFER, size, dataptr, usage)
```

Attributes Formatting

`glVertexAttribPointer(index, size, dtype, norm, str, ptr)`

- `index`: index of the vertex attribute to format → `unsigned int`
- `size`: number of components of the attribute → `int (1-4)`
- `dtype`: type of the attributes → `enum` (e.g., `GL_FLOAT`, `GL_BYTE`...)
- `norm`: whether the value should be normalised → `bool`
- `str`: byte distance between values of consecutive vertices → `GLsizei` (non-negative integer)
- `ptr`: offset in bytes of first component of the first attribute → `GLvoid*` (casted from `GLsizei`)

Primitive Assembly

Executed as last step of the **Geometry Stage**.
It is **fixed** and the only control over the process is given by the possibility of specifying the kind of primitive we want OpenGL to read and render from the vertex data loaded in the VBOs. The primitives OpenGL knows are:

- `GL_POINTS`: every vertex is a separate point
- `GL_LINES`: segments connecting a vertex with the following
- `GL_LINE_STRIP`: series of segments connecting all the vertices
- `GL_LINE_LOOP`: idem + segment between the last and the first vertex
- `GL_TRIANGLES`: triangles defined by three consecutive vertices
- `GL_TRIANGLE_STRIP`: series of triangles sharing in pairs two vertices
- `GL_TRIANGLE_FAN`: series of triangles with one common vertex

Vertex Indexing

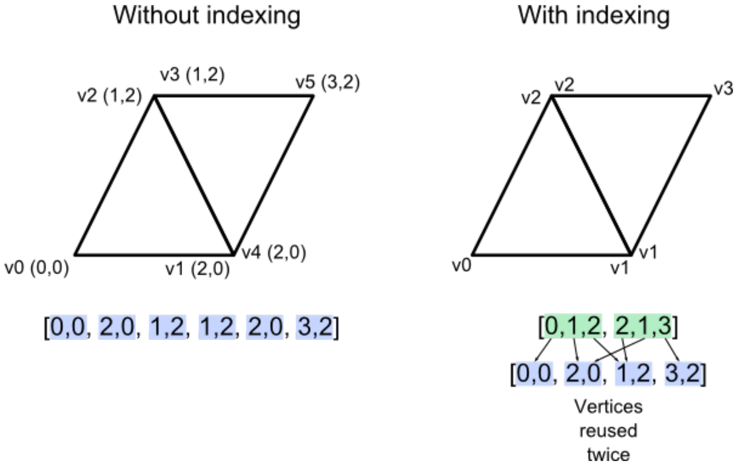


Fig. 3: Vertex representations

More info *here*

Drawing

(Re)bind VAO of Geometry

`glBindVertexArray(vao_handle)`

Bind Shader Program(s)

`glUseProgram(program_handle)`

Draw Geometry

Using indices and parameters

`glDrawElements(prim_type, count, index_type, index_ptr)`

Using parameters but NO indices

`glDrawArrays(prim_type, start_index, count)`

Thanks for the Attention!