Shader Buffers

Uniform & Shader Storage Buffer Objects

Francesco Andreussi

Bauhaus–Universität Weimar

20 December 2018

Bauhaus-Universität Weimar

Faculty of Media

Uniform Buffer Objects

Uniform Buffer Objects are used to make more compact and efficient the uploading of uniforms to the Shaders.

Instead of passing one variable at a time, developers can pass **multiple variables at once**. It is particularly useful when a group of data represent different characteristics of a same object/feature of the environment.

Furthermore, some values (as View and Projection Matrices) are used in multiple shaders; updating the values of a UBO, OpenGL automatically **updates all the shaders**, to which the UBO is bound.

Uniform Interface Blocks

In the **shader**, the buffer object is defined as an **Interface Block**. It's syntax is analogous to a `struct` in C:

every Block has *at least* a **layout specifier**, a **Block Name** (seen by OpenGL code), some **members** (the stored data) and (optionally) an **Instance Name** (seen by GLSL code); arrays of Blocks con be defined.

Simple

```
layout (std140) uniform CameraBlock {
  mat4 ViewMatrix;
  mat4 ProjectionMatrix;
};
...
pass_Position = ViewMatrix * ModelMatrix * ...
```

Instance Name

```
layout (std140) uniform CameraBlock {
  mat4 ViewMatrix;
  mat4 ProjectionMatrix;
} blockCam;
...
pass_Position = blockCam.ViewMatrix * blockCam.ModelMatrix * ...
```

Block Array

```
layout (std140) uniform CameraBlock {
  mat4 ViewMatrix;
  mat4 ProjectionMatrix;
} blockCam[3];
...
pass_Position = blockCam[1].ViewMatrix * blockCam[1].ModelMatrix
```

Interface Block (GLSL)

Fig. 1: Uniform Interface Blocks Examples

UBO Bindings

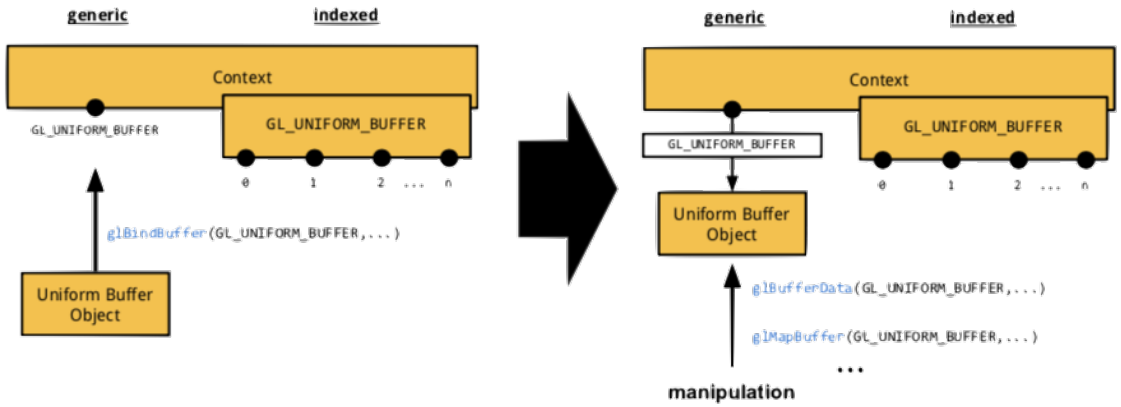

Fig. 2: Generic UBO Binding

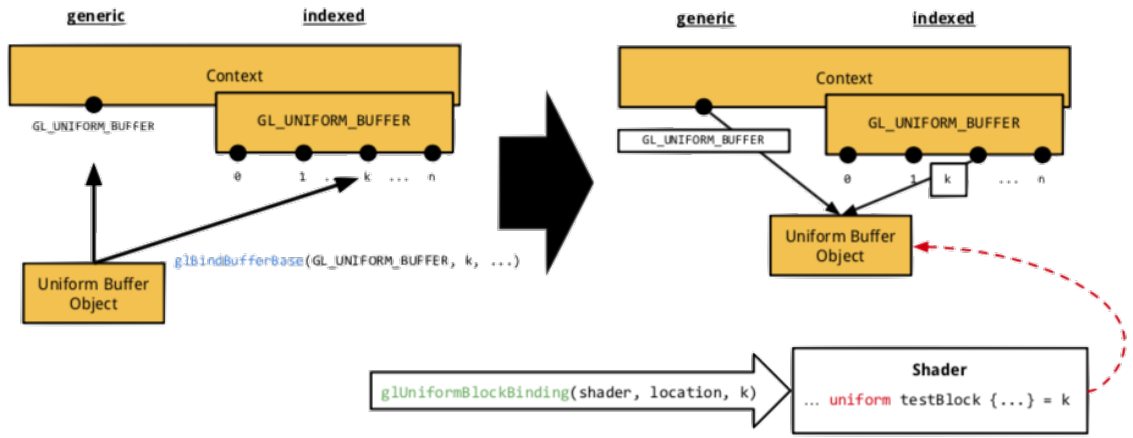

Fig. 3: Indexed UBO Binding

Buffer Block Memory Layouts

There are 4 alternatives Memory Layouts:

- **packed**: performance–oriented and implementation–determined, the layout is determined by the internals and the actual exchanged data may vary.
- **shared**: loosened version of `packed`; no variation on the exchanged data format, a certain type of Block has the same layout in different programs.
- **std140**: fixed layout at declaration time, wastes space introducing paddings to 16–bits multiples.
- **std430**: similar to `std140`, but without the additional paddings (to be used **only** with Shader Storage Blocks).

UBO Usage

Create Uniform Buffer Object

```
glGenBuffers(1, &ubo_handle)
glBindBufferBase(GL_UNIFORM_BUFFER, buff_idx, ubo_handle)
glBufferData(GL_UNIFORM_BUFFER, size, nullptr, usage)
```

Bind UBO to shader Interface Block

```
GLuint loc=glGetUniformBlockIndex(pr_handle, ‘‘BlkName’’)
glUniformBlockBinding(pr_handle, loc, buff_idx)
```

Update Buffer Data (when necessary)

```
glBindBuffer(GL_UNIFORM_BUFFER, ubo_handle)
void* buff_ptr=glMapBuffer(GL_UNIFORM_BUFFER, GL_WRITE_ONLY)
std::memcpy(data_ptr, buffer_ptr, buffer_size)
glUnmapBuffer(GL_UNIFORM_BUFFER)
```

Shader Storage Buffer Object

They are similar to UBO, but there are some differences:

- SSBOs can be larger (up to 128MB instead of 16KB).
- SSBOs are writable from the shaders (UBO are not).
- SSBOs can have variable size and so they can store arrays of arbitrary length.

Because they provide more flexibility, they are less efficient than UBOs.

They are supported **only by OpenGL v4.3**.

Shader Storage Interface Blocks

SSIBs declaration has the same structure of UIBs'.

However, the storage qualifier here **must** be `buffer` instead of `uniform`, the accepted layouts are **only std140 and std430** and it is possible to use arrays of arbitrary size (preferably located as last data).

The array size can be accessed with the GLSL command `myArray.lenght()`.

Constant Size

```
const uint NUM_LIGHTS = 10;
layout (std430) buffer LightBlock {
  vec4 ViewPosition;
  vec4 LightPositions[NUM_LIGHTS];
};
...
for (uint i = 0; i < LightPositions.length(); ++i) {
    ...
}
```

Dynamic Size

```
layout (std430) buffer LightBlock {
  vec4 ViewPosition;
  vec4 LightPositions[];
};
...
for (uint i = 0; i < LightPositions.length(); ++i) {
    ...
}
```

Fig. 4: Shader Storage Interface Blocks Examples

SSBO Usage

Create Shader Storage Buffer Object

```
glGenBuffers(1, &ssbo_handle)
glBindBufferRange(GL_SHADER_STORAGE_BUFFER, buff_idx,
ssbo_handle, offset, used_data_size)
glBufferData(GL_SHADER_STORAGE_BUFFER, size, nullptr,
usage)
```

Bind UBO to shader Interface Block

```
GLuint loc=glGetProgramResourceIndex(pr_handle,
GL_SHADER_STORAGE_BLOCK, ‘‘BlkName’’)
glShaderStorageBlockBinding(pr_handle, loc, buff_idx)
```

Update Buffer Data (when necessary)

```
glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo_handle)
void* buff_ptr=glMapBuffer(GL_SHADER_STORAGE_BUFFER,
GL_WRITE_ONLY)
std::memcpy(data_ptr, buffer_ptr, buffer_size)
glUnmapBuffer(GL_SHADER_STORAGE_BUFFER)
```

Thanks for the Attention and Happy Holidays!