Increasing [Effective and robust translation tables with zero based enums in C++](post-devel-cpp.html#menu)

05 january 2015, [Christophe Riccio](mailto:mail@g-truc.net)



[G-Truc Creation](http://www.g-truc.net)

# Table of contents

Table of contents 2

Introduction 3

1. Data accesses 3

1.1. Using constants for accesses 3

1.2. Using indexes for accesses 4

2. Translations 5

2.1. Definition 6

2.2. Translation implementations 6

2. Some scene complexity theory 10

3. Energy applied to scene complexity 10

3. Evidences justifying the multi draw mind frame 11

1.1. ARB\_sparse\_texture 12

1.2. ARB\_bindless\_texture 12

1.3. ARB\_seamless\_cubemap\_per\_texture 13

1.4. Immutable buffers (OpenGL 4.4, ARB\_buffer\_storage) 13

2.2. ARB\_indirect\_parameters 15

5. Better interactions with the fixed pipeline functionalities 16

6. Stronger shader interface between shader stages 17

6.1. Partial matching of shader interfaces using blocks for separate shader programs. 17

6.2. Packed shader inputs and outputs 19

6.3. Future directions and deprecation for the input and output interfaces 20

6.4. Offset and alignment in uniform and storage buffer. 20

6.5. Constant expressions for qualifiers. 21

6.6. Embedded transform feedback interface in shaders 21

6.7. More embedded shader interfaces for resources? 22

Conclusions 23

Is there room for an OpenGL 4.5 specification? Always! 23

Deprecation for OpenGL 5 hardware? 23

Features for OpenGL 5 hardware level? 24

Last words 24

# Introduction

It is often useful to use enumeration values as an identifier however all enumerations are not equal.

OpenGL is using weak-typed enumerations using assigned values where GLenum is nothing but an alias for unsigned int and all the constants are defines within a global scope.

|  |
| --- |
| #define GL\_FRAGMENT\_SHADER 0x8B30  #define GL\_READ\_ONLY 0x88B8  #define GL\_UNIFORM\_BUFFER 0x8A11  #define GL\_TEXTURE\_BASE\_LEVEL 0x813C  #define GL\_UNSIGNED\_INT\_10F\_11F\_11F\_REV 0x8C3B |

Such enumeration values are undesirable identifiers for many reasons:   
- It is very difficult to access data using such identifiers.  
- We can pass invalid values that result in runtime errors undetected at compile time.  
- They imply API dependence in cross graphics API code.  
- etc.

In this article, we propose to use **user-defined enums with translation tables** allowing detecting at compilation time translation issues and providing constantly fast performance.

We will illustrate this chapter working with code samples based on our experiences with OpenGL, generated assembly from multiple compilers and performance numbers.

Finally, after demonstration we will conclusion by a simple best practice summary with a reference code for every day use.

# 1. Data accesses

## 1.1. Using constants for accesses

In this section we are going to study a typical use case where we want to use an identifier to access [the programs of a graphics program pipeline](https://www.opengl.org/registry/specs/ARB/separate_shader_objects.txt) in OpenGL.

|  |
| --- |
| #define GL\_VERTEX\_SHADER 0x8B31  #define GL\_TESS\_CONTROL\_SHADER 0x8E88  #define GL\_TESS\_EVALUATION\_SHADER 0x8E87  #define GL\_GEOMETRY\_SHADER 0x8DD9  #define GL\_FRAGMENT\_SHADER 0x8B30 |

Listing 1.1.1 OpenGL defines the following constants for the shader stages

A native idea would be that we could use these constants to access the programs of a program pipeline. A first but not uncommon approach is to use a switch to return the matching OpenGL program name.

|  |
| --- |
| GLuint getProgramName(GLenum Stage) const  {  switch(Stage)  {  case GL\_VERTEX\_SHADER: return this->VertProgramName;  case GL\_TESS\_CONTROL\_SHADER: return this->ContProgramName;  case GL\_TESS\_EVALUATION\_SHADER: return this->EvalProgramName;  case GL\_GEOMETRY\_SHADER: return this->GeomProgramName;  case GL\_FRAGMENT\_SHADER: return this->FragProgramName;  default:  assert(0); // Invalid value for 'Stage'  return 0;  }  } |

Listing 1.1.2. Trivializing the programs accesses issue using a C++ switch

The code shown is listing 1.1.2 is an abomination for the following reasons.  
- The function user may submit any integer input value, the compiler won’t complain.  
- Just looking at the prototype, the user can’t know that GL\_COMPUTE\_SHADER is not a valid value.  
- If the class adds support to GL\_COMPUTE\_SHADER, the compiler won’t help the programmer to update getProgramName by throwing a compiler time error.  
- The constants in listing 1.1.1 have a different semantic from the Stage input variable.  
- The code is inefficient, basically compiled into a series of if instructions.   
- The function performance is dependent of the Stage value.  
- The more values we add, the slower the function becomes.  
- The function generates a lot of CPU instructions most of which are never used, polluting the instruction cache and causing previous code in cache to be evicted.  
- Etc.

Another very common solution but just as bad, is to design an over engineered solution based on a std::map.

|  |
| --- |
| GLuint getProgramName(GLenum Stage) const  {  std::map<GLenum, GLuint>::const\_iterator it = this->ProgramNames.find(Stage);  assert(it != this->ProgramNames.end()); // Invalid value for 'Stage'  return it->second;  } |

Listing 1.1.3. Over-engineering the solution with a std::map

The aesthetic of this code may look better than the code in listing 1.1.2 but the code suffers the same issues and querying a program name is even a lot slower because we will suffer many cache misses jumping from node to node in the find function before returning the requested OpenGL program name.

## 1.2. Using indexes for accesses

When we take the time to think about how with are going to access some data, we quickly figure out that the easiest, the most robust, and the most efficient way to access data is to index an array.

Once we chose to access the data through a table we need to index that table and a zero-based enumeration comes logically to mind for that purpose.

|  |
| --- |
| enum stage  {  VERTEX = 0,  TESS\_CONTROL,  TESS\_EVALUATION,  GEOMETRY,  FRAGMENT  };  GLuint GetProgramName(stage Stage) const  {  return this->ProgramNames[Stage];  } |

Listing 1.2.1. Table access using a zero based enumeration

Typically, the desire of reusing existing values is motivated by avoiding the duplication of values. However, as shown in listing 1.2.1, creating additional values is vastly superior.

- The function user can only submit one of the enum values or the compiler will complain.  
- If the user submits ‘COMPUTE’, the compiler will throw an error.   
- The code is efficient, basically compiled into addressing an array.  
- The function performance is independent from the Stage value.  
- The performance is roughly independent from the number of value in the enumeration.   
- The constants in listing 1.1.1 have a different semantic from the Stage input variable.  
- The function code is compact and entirely executed making good use of the CPU instruction cache.

We can still improve the reliability of this code by adding a value to identify the number of elements in the stage enumeration. Using this value we can size the ProgramNames variable automatically when new enumerations are added.

|  |
| --- |
| enum stage  {  VERTEX = 0,  TESS\_CONTROL,  TESS\_EVALUATION,  GEOMETRY,  FRAGMENT, STAGE\_LAST = FRAGMENT  };  std::array<GLuint, STAGE\_LAST + 1> ProgramNames;  GLuint GetProgramName(stage Stage) const  {  return this->ProgramNames[Stage];  } |

Listing 2.2. Automatically sized array following the number of enumeration values.

An alternative to the stage definition in listing 2.2 is the stage definition in listing 2.3. However, listing 2.2 is more reliable because it doesn’t introduce an invalid index for ProgramNames.

|  |
| --- |
| enum stage  {  VERTEX = 0,  TESS\_CONTROL,  TESS\_EVALUATION,  GEOMETRY,  FRAGMENT, STAGE\_COUNT  };  std::array<GLuint, STAGE\_COUNT> ProgramNames; |

Listing 2.3. Alternative to listing 2.2 but that introduces an invalid value to the enumeration.

# 2. Translations

## 2.1. Definition

A motivation to use existing enumerations for addressing data is that it we create a new and better fitting enumeration, then we will need to convert that new enumeration into the original enumeration or we will need to store both values.

We call translation the conversion a set of identifiers to a different set of identifiers.

Building on the OpenGL shaders example, listing 2.1 shows an instance of translation:   
VERTEX => GL\_VERTEX\_SHADER  
TESS\_CONTROL => GL\_TESS\_CONTROL\_SHADER  
TESS\_EVALUATION => GL\_TESS\_EVALUATION\_SHADER  
GEOMETRY => GL\_GEOMETRY\_SHADER  
FRAGMENT => GL\_FRAGMENT\_SHADER

Performing this conversion in the other direction is still a translation even it’s questionable:  
GL\_VERTEX\_SHADER => VERTEX  
GL\_TESS\_CONTROL\_SHADER => TESS\_CONTROL  
GL\_TESS\_EVALUATION\_SHADER => TESS\_EVALUATION  
GL\_GEOMETRY\_SHADER => GEOMETRY  
GL\_FRAGMENT\_SHADER => FRAGMENT

We can also have multiple translations from a set of identifiers into N set of identifiers:  
VERTEX => GL\_VERTEX\_SHADER\_BIT  
TESS\_CONTROL => GL\_TESS\_CONTROL\_SHADER\_BIT  
TESS\_EVALUATION => GL\_TESS\_EVALUATION\_SHADER\_BIT  
GEOMETRY => GL\_GEOMETRY\_SHADER\_BIT  
FRAGMENT => GL\_FRAGMENT\_SHADER\_BIT

Properties:  
- Translations are surjection functions  
- Translations may be bijective functions  
- Multiple translations function may be written for a set of identifiers

## 2.2. Translation implementations

A first possible implementation is to build a special case of listing 1.1.2 to implement the translation.

|  |
| --- |
| GLenum translate(stage Stage)  {  switch(Stage)  {  case VERTEX\_SHADER: return GL\_VERTEX\_SHADER;  case TESS\_CONTROL\_SHADER: return GL\_TESS\_CONTROL\_SHADER;  case TESS\_EVALUATION\_SHADER: return GL\_TESS\_EVALUATION\_SHADER;  case GEOMETRY\_SHADER: return GL\_GEOMETRY\_SHADER;  case FRAGMENT\_SHADER: return GL\_FRAGMENT\_SHADER;  case COMPUTE\_SHADER: return GL\_COMPUTE\_SHADER;  }  } |

Listing 2.2.1. Translation implementation based on switch.

|  |
| --- |
| ?index\_switch\_translate@translation4@@YA?AW4translated@1@W4index@1@@Z PROC ; translation4::index\_switch\_translate  test ecx, ecx  je SHORT $LN4@index\_swit  dec ecx  je SHORT $LN3@index\_swit  dec ecx  je SHORT $LN2@index\_swit  dec ecx  jne SHORT $LN5@index\_swit  mov eax, 35567 ; 00008aefH  ret 0  $LN2@index\_swit:  mov eax, 11317 ; 00002c35H  ret 0  $LN3@index\_swit:  mov eax, 450 ; 000001c2H  ret 0  $LN4@index\_swit:  mov eax, 41191 ; 0000a0e7H  $LN5@index\_swit:  ret 0  ?index\_switch\_translate@translation4@@YA?AW4translated@1@W4index@1@@Z ENDP ; translation4::index\_switch\_translate |

Listing 2.2.2.Visual C++ 2013 assembly for listing 2.2.1.

Looking at the assembly with see that the generated code for this function is particularly slow with a lot of jumps. Worse, the more values the enumerations contain, the longer and slower the code is going to be. Finally, the performance of a function depends on the input value because the code path will differ according to the input value.

|  |
| --- |
| GLenum translate(stage Stage)  {  static const GLenum Table[] =  {  GL\_VERTEX\_SHADER, // VERTEX\_SHADER  GL\_TESS\_CONTROL\_SHADER, // TESS\_CONTROL\_SHADER  GL\_TESS\_EVALUATION\_SHADER, // TESS\_EVALUATION\_SHADER  GL\_GEOMETRY\_SHADER, // GEOMETRY\_SHADER  GL\_FRAGMENT\_SHADER // FRAGMENT\_SHADER  };  return Table[Stage];  } |

Listing 2.2.3. Translation implementation based on a static const table.

|  |
| --- |
| ?static\_table\_translate@translation4@@YA?AW4translated@1@W4index@1@@Z PROC ; translation4::static\_table\_translate  ; 51 : static const translated Table[] =  ; 52 : {  ; 53 : TRANSLATED\_A, // INDEX\_A  ; 54 : TRANSLATED\_B, // INDEX\_B  ; 55 : TRANSLATED\_C, // INDEX\_C  ; 56 : TRANSLATED\_D // INDEX\_D  ; 57 : };  ; 58 :  ; 59 : static\_assert(sizeof(Table) / sizeof(translated) == INDEX\_COUNT, "The translation table needs to be updated.");  ; 60 : assert(Index < INDEX\_COUNT);  ; 61 :  ; 62 : return Table[Index];  movsxd rax, ecx  lea rcx, OFFSET FLAT:?Table@?1??static\_table\_translate@translation4@@YA?AW4translated@2@W4index@2@@Z@4QBW432@B  mov eax, DWORD PTR [rcx+rax\*4]  ; 63 : }  ret 0  ?static\_table\_translate@translation4@@YA?AW4translated@1@W4index@1@@Z ENDP ; translation4::static\_table\_translate |

Listing 2.2.4. Visual C++ 2013 assembly for listing 2.2.3.

|  |
| --- |
| GLenum translate(stage Stage)  {  const GLenum Table[] =  {  GL\_VERTEX\_SHADER, // VERTEX\_SHADER  GL\_TESS\_CONTROL\_SHADER, // TESS\_CONTROL\_SHADER  GL\_TESS\_EVALUATION\_SHADER, // TESS\_EVALUATION\_SHADER  GL\_GEOMETRY\_SHADER, // GEOMETRY\_SHADER  GL\_FRAGMENT\_SHADER // FRAGMENT\_SHADER  };  return Table[Stage];  } |

Listing 2.2.5. Translation implementation based on a const table.

|  |
| --- |
| ?static\_table\_translate@translation4@@YA?AW4translated@1@W4index@1@@Z PROC ; translation4::static\_table\_translate  ; 51 : static const translated Table[] =  ; 52 : {  ; 53 : TRANSLATED\_A, // INDEX\_A  ; 54 : TRANSLATED\_B, // INDEX\_B  ; 55 : TRANSLATED\_C, // INDEX\_C  ; 56 : TRANSLATED\_D // INDEX\_D  ; 57 : };  ; 58 :  ; 59 : static\_assert(sizeof(Table) / sizeof(translated) == INDEX\_COUNT, "The translation table needs to be updated.");  ; 60 : assert(Index < INDEX\_COUNT);  ; 61 :  ; 62 : return Table[Index];  movsxd rax, ecx  lea rcx, OFFSET FLAT:?Table@?1??static\_table\_translate@translation4@@YA?AW4translated@2@W4index@2@@Z@4QBW432@B  mov eax, DWORD PTR [rcx+rax\*4]  ; 63 : }  ret 0  ?static\_table\_translate@translation4@@YA?AW4translated@1@W4index@1@@Z ENDP ; translation4::static\_table\_translate |

Listing 2.2.6. Visual C++ 2013 assembly for listing 2.2.5.

## 2.3. static const vs const translation table

My instinct would be that it should be static and not only const because when we enter the function there will be a stack allocation big enough so that all the variables in it could fit there. The issue with only const is that the function might get bigger so it will consume more CPU instructions cache, hence evict more stuff resulting in more instruction cache misses. So, my choice here is to fight against instructions cache evictions. With static, the table should go into the data part of the ASM so effectively the function remains compact and evict less. The downside is that we may cache miss twice. Once on the function call (in L1 inst. cache) and once on the table fetch (in L1 data cache which is not that busy in "random code" while is super busy in optimize data transformation code). It possible that in practice the compilers put the table at the same place. I can check that and see, I don't know. Benchmarking might be a little hard as it might depend on the benchmark. My felling is that our call stack is pretty big so we are probably bound by the instruction cache evictions rather than cache miss latencies but it's possible that we end up evicting all the time anyway in which case it's better to optimize for number of cache misses. I am pretty doubtful on this because I would be super surpris that the data cache is a bottleneck is random plumbing code. It's definitely an interesting topic which I am happy to study more. In any case, this is magnitudes more efficient and robust than doing translations with switch statements! A rule on using switch statement? Hell yes! I would be up for something like that: "Stop pretending with 'switch' that the code is 'future proof', make your code break everywhere X is used if your enums changed using indirection tables instead of switches".

|  |
| --- |
| ?const\_table\_translate@translation4@@YA?AW4translated@1@W4index@1@@Z PROC  $LN4:  sub rsp, 40 ; 00000028H  mov rax, QWORD PTR \_\_security\_cookie  xor rax, rsp  movsxd rax, ecx  movdqu XMMWORD PTR Table$[rsp], xmm0  mov eax, DWORD PTR Table$[rsp+rax\*4]  mov rcx, QWORD PTR \_\_$ArrayPad$[rsp]  xor rcx, rsp  call \_\_security\_check\_cookie  add rsp, 40 ; 00000028H  ret 0  ?const\_table\_translate@translation4@@YA?AW4translated@1@W4index@1@@Z ENDP |

Listing 2.3.1. Visual C++ 2013 of a translation function based on a const table.

## 2.4. Clang and GCC behaviors

## 2.5. Translation performances

*Performance comparison between translation using 'switch' and a table. 4 to 128 values per enumeration*

# 3. Translation table robustness

## 3. 1. Detecting the addition or removal of enumeration values at compilation time

Translation from user-defined enum to OpenGL enum:

* enum stage
* {
* VERTEX\_SHADER,
* TESS\_CONTROL\_SHADER,
* TESS\_EVALUATION\_SHADER,
* GEOMETRY\_SHADER,
* FRAGMENT\_SHADER,
* COMPUTE\_SHADER,
* LAST\_SHADER = COMPUTE\_SHADER
* };
* GLenum translate(stage Stage)
* {
* static const GLenum Table[] =
* {
* GL\_VERTEX\_SHADER, // VERTEX\_SHADER
* GL\_TESS\_CONTROL\_SHADER, // TESS\_CONTROL\_SHADER
* GL\_TESS\_EVALUATION\_SHADER, // TESS\_EVALUATION\_SHADER
* GL\_GEOMETRY\_SHADER, // GEOMETRY\_SHADER
* GL\_FRAGMENT\_SHADER, // FRAGMENT\_SHADER
* GL\_COMPUTE\_SHADER // COMPUTE\_SHADER
* };
* static\_cast(
* sizeof(Table) / sizeof(GLenum) == LAST\_SHADER + 1,
* "OPENGL ERROR: The translation table for 'stage' needs to be updated.");
* return Table[Stage];
* };

## 3. 2. Detecting invalid values at runtime

Translation from user-defined enum to OpenGL enum:

* enum stage
* {
* VERTEX\_SHADER,
* TESS\_CONTROL\_SHADER,
* TESS\_EVALUATION\_SHADER,
* GEOMETRY\_SHADER,
* FRAGMENT\_SHADER,
* COMPUTE\_SHADER,
* LAST\_SHADER = COMPUTE\_SHADER
* };
* GLenum translate(stage Stage)
* {
* static const GLenum Table[] =
* {
* GL\_VERTEX\_SHADER, // VERTEX\_SHADER
* GL\_NONE, // TESS\_CONTROL\_SHADER
* GL\_NONE, // TESS\_EVALUATION\_SHADER
* GL\_GEOMETRY\_SHADER, // GEOMETRY\_SHADER
* GL\_FRAGMENT\_SHADER, // FRAGMENT\_SHADER
* GL\_COMPUTE\_SHADER // COMPUTE\_SHADER
* };
* static\_cast(
* sizeof(Table) / sizeof(GLenum) == LAST\_SHADER + 1,
* "OPENGL ERROR: The translation table for 'shader\_stage' needs to be updated.");
* GLenum Translation = Table[Stage];
* assert(Translation != GL\_NONE); // Invalid value for 'Stage'
* return Translation;
* };

# 4. Extending translation table functionality for runtime decisions

## 4.1. Baking translation tables at runtime

Translation from user-defined enum to OpenGL enum:

* enum stage
* {
* VERTEX\_SHADER,
* TESS\_CONTROL\_SHADER,
* TESS\_EVALUATION\_SHADER,
* GEOMETRY\_SHADER,
* FRAGMENT\_SHADER,
* COMPUTE\_SHADER,
* LAST\_SHADER = COMPUTE\_SHADER
* };
* GLenum translate(std::array<stage, LAST\_SHADER + 1> const & Table, stage Stage)
* {
* static\_cast(
* sizeof(shader) \* Table.size() / sizeof(GLenum) == LAST\_SHADER + 1,
* "OPENGL ERROR: The translation table for 'stage' needs to be updated.");
* GLenum Translation = Table[Stage];
* assert(Translation != GL\_NONE); // Invalid value for 'Stage'
* return Translation;
* };

## 4.2. Dynamic translation tables at runtime

Translation from user-defined enum to OpenGL enum:

* typedef std::size\_t stage;
* GLenum translate(std::vector<stage> const & Table, stage Stage)
* {
* assert(Stage < Table.size()); // Invalid value for 'Stage'
* GLenum Translation = Table[Stage];
* assert(Translation != GL\_NONE); // Invalid value for 'Stage'
* return Translation;
* };

# 5. Usages and abuses of translations

## 5.1. Use code context to avoid unnecessary translations

Example of useless last minute translation:

* enum target
* {
* UNIFORM\_BUFFER,
* SHADER\_STORAGE\_BUFFER,
* TRANSFORM\_FEEDBACK\_BUFFER
* };
* void bindBuffer(target Target, GLuint Unit, GLuint Buffer)
* {
* switch(Target)
* {
* case UNIFORM\_BUFFER:
* if(this->UniformBufferBound[Unit] == Buffer)
* return;
* this->UniformBufferBound[Unit] = Buffer;
* break;
* case SHADER\_STORAGE\_BUFFER:
* if(this->ShaderStorageBufferBound[Unit] == Buffer)
* return;
* this->ShaderStorageBufferBound[Unit] = Buffer;
* break;
* case TRANSFORM\_FEEDBACK\_BUFFER:
* if(this->TransformFeedbackBufferBound[Unit] == Buffer)
* return;
* this->TransformFeedbackBufferBound[Unit] = Buffer;
* break;
* }
* glBindBufferBase(translate(Target), Unit, Buffer);
* }
* void foo()
* {
* ...
* for(GLuint i = 0; i < Buffers.size(); ++i)
* bindBuffer(UNIFORM\_BUFFER, i, Buffers[i]);
* ...
* }

Using the code context to avoid translation:

* void bindUniformBuffer(GLuint Unit, GLuint Buffer);
* {
* if(this->UniformBufferBound[Unit] == Buffer)
* return;
* glBindBufferBase(GL\_UNIFORM\_BUFFER, Unit, Buffer);
* this->UniformBufferBound[Unit] = Buffer;
* }
* void bindShaderStorageBuffer(GLuint Unit, GLuint Buffer);
* {
* if(this->ShaderStorageBufferBound[Unit] == Buffer)
* return;
* glBindBufferBase(GL\_SHADER\_STORAGE\_BUFFER, Unit, Buffer);
* this->ShaderStorageBufferBound[Unit] = Buffer;
* }
* void bindTransformFeedbackBuffer(GLuint Unit, GLuint Buffer);
* {
* if(this->TransformFeedbackBufferBound[Unit] == Buffer)
* return;
* glBindBufferBase(GL\_TRANSFORM\_FEEDBACK\_BUFFER, Unit, Buffer);
* this->TransformFeedbackBufferBound[Unit] = Buffer;
* }
* void foo()
* {
* ...
* for(GLuint i = 0; i < Buffers.size(); ++i)
* bindUniformBuffer(i, Buffers[i]);
* ...
* }