Robust and efficient translations in C++ using tables with zero-based enumerations

10 January 2015 – 25 January 2015, *Christophe Riccio*

G-Truc Creation

Table of contents

TABLE OF CONTENTS 2

INTRODUCTION 3

1. DATA ACCESSES 4

1.1. USING CONSTANTS FOR ACCESSES 4
1.2. USING INDEXES FOR ACCESSES 5

2. TRANSLATIONS 7

2.1. DEFINITION 7
2.2. TRANSLATION IMPLEMENTATIONS 7

3. PERFORMANCES 9

3.1. THE TESTS 9
3.2. VISUAL STUDIO 2013 INITIAL RESULTS 9
3.3. MORE VISUAL STUDIO VERSIONS RESULTS 10
3.4. CLANG, GCC, INTEL COMPILER RESULTS 10

4. ASSEMBLY ANALYSIS 13

4.1. STATIC CONST VS CONST TRANSLATION TABLE 13
4.2. VALUE SWITCH VS INDEX SWITCH 15

5. TRANSLATION TABLE ROBUSTNESS 19

5.1. DETECTING THE ADDITION OR REMOVAL OF ENUMERATION VALUES AT COMPILATION TIME 19
5 2. DETECTING TRANSLATION RUNTIME INPUT ERRORS 19
5 3. LIMITING THE TRANSLATION RANGE 21

6. EXTENDING TRANSLATION TABLE FUNCTIONALITY FOR RUNTIME DECISIONS 23

6.1. BAKING TRANSLATION TABLES AT RUNTIME 23
6 2. DETECTING TRANSLATION RUNTIME OUTPUT ERRORS 24

7. PITFALLS 26

7.1. FASTER THAN THE FASTEST TRANSLATION: NO TRANSLATION 26
7.2. REORDERING OF ENUMERATION VALUES 27

CONCLUSIONS 29

Introduction

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

OpenGL uses weak-typed enumerations and arbitrary 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
```

Example of OpenGL enumeration values.

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 a graphics API specific dependence.
- Etc.

In this article, we propose to use **zero-based enumerations with translation tables** allowing detecting at compilation time translation issues and providing constantly fast performance across multiple compilers, including Clang, GCC, Intel Compiler and Visual C++ tested for this article.

To support this proposal we will work with code samples, building from our experiences with OpenGL to provide examples of concrete usage. We will also provide an analysis of generated assembly from compilers and performance results.

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 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.

A common attitude with programmers is to blame the performance issue on **`std::map`**. This is missing the point entirely. We can write the fastest **`map`** ever, it will still be the wrong tool.

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
{
	STAGE_VERTEX = 0,
	STAGE_TESS_CONTROL,
	STAGE_TESS_EVALUATION,
	STAGE_GEOMETRY,
	STAGE_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 enumeration values or the compiler will complain.
- If the user submits **`'STAGE_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
{
	STAGE_VERTEX = 0,
	STAGE_TESS_CONTROL,
	STAGE_TESS_EVALUATION,
	STAGE_GEOMETRY,
	STAGE_FRAGMENT,
	STAGE_LAST = STAGE_FRAGMENT
};

std::array<GLuint, STAGE LAST + 1> ProgramNames;

GLuint GetProgramName(stage Stage) const
{
	return this->ProgramNames[Stage];
}
```

Listing 1.2.2: Automatically sized array following the number of enumeration values.

An alternative to the **stage** definition in listing 1.2.2 is the **stage** definition in listing 1.2.3. However, listing 1.2.2 is more reliable because it doesn't introduce an invalid index for **ProgramNames.**

```
enum stage
{
	STAGE_VERTEX = 0,
	STAGE_TESS_CONTROL,
	STAGE_TESS_EVALUATION,
	STAGE GEOMETRY,
	STAGE_FRAGMENT,
	STAGE_COUNT
};

std::array<GLuint, STAGE_COUNT> ProgramNames;
```

Listing 1.2.3: 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 2.1.1 shows an instance of translation:

```
STAGE_VERTEX => GL_VERTEX_SHADER
STAGE_TESS_CONTROL => GL_TESS_CONTROL_SHADER
STAGE_TESS_EVALUATION => GL_TESS_EVALUATION_SHADER
STAGE_GEOMETRY => GL_GEOMETRY_SHADER
STAGE_FRAGMENT => GL_FRAGMENT_SHADER
```

Listing 2.1.1: An instance of translation.

Performing this conversion in the other direction is still a translation even if it's questionable:

```
GL_VERTEX_SHADER => STAGE_VERTEX
GL_TESS_CONTROL_SHADER => STAGE_TESS_CONTROL
GL_TESS_EVALUATION_SHADER => STAGE_TESS_EVALUATION
GL_GEOMETRY_SHADER => STAGE_GEOMETRY
GL_FRAGMENT_SHADER => STAGE_FRAGMENT
```

Listing 2.1.2: Reverse translation.

We can also have multiple translations from a set of identifiers into N set of identifiers:

```
STAGE_VERTEX => GL_VERTEX_SHADER_BIT
STAGE_TESS_CONTROL => GL_TESS_CONTROL_SHADER_BIT
STAGE_TESS_EVALUATION => GL_TESS_EVALUATION_SHADER_BIT
STAGE_GEOMETRY => GL_GEOMETRY_SHADER_BIT
STAGE_FRAGMENT => GL_FRAGMENT_SHADER_BIT
```

Listing 2.1.3: Second translation from a unique enumeration.

Properties:

- Translations are surjection functions
- Translations may be bijective functions
- Multiple translation functions may be written for a set of identifiers as shown between listing 2.1.1 and 2.1.3.

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 STAGE_VERTEX: return GL_VERTEX_SHADER;
        case STAGE_TESS_CONTROL: return GL_TESS_CONTROL_SHADER;
        case STAGE_TESS_EVALUATION: return GL_TESS_EVALUATION_SHADER;
        case STAGE_GEOMETRY: return GL_GEOMETRY_SHADER;
        case STAGE_FRAGMENT: return GL_FRAGMENT_SHADER;
        }
}
```

Listing 2.2.1: Translation implementation based on switch.

Looking at the assembly we 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 GLenum const Table[] =
        {
                GL_VERTEX_SHADER,           // STAGE_VERTEX
                GL_TESS_CONTROL_SHADER,     // STAGE_TESS_CONTROL
                GL_TESS_EVALUATION_SHADER, // STAGE_TESS_EVALUATION
                GL_GEOMETRY_SHADER,         // STAGE_GEOMETRY
                GL_FRAGMENT_SHADER          // STAGE_FRAGMENT
        };

        return Table[Stage];
}
```

Listing 2.2.2: Translation implementation based on a static const table.

3. Performances

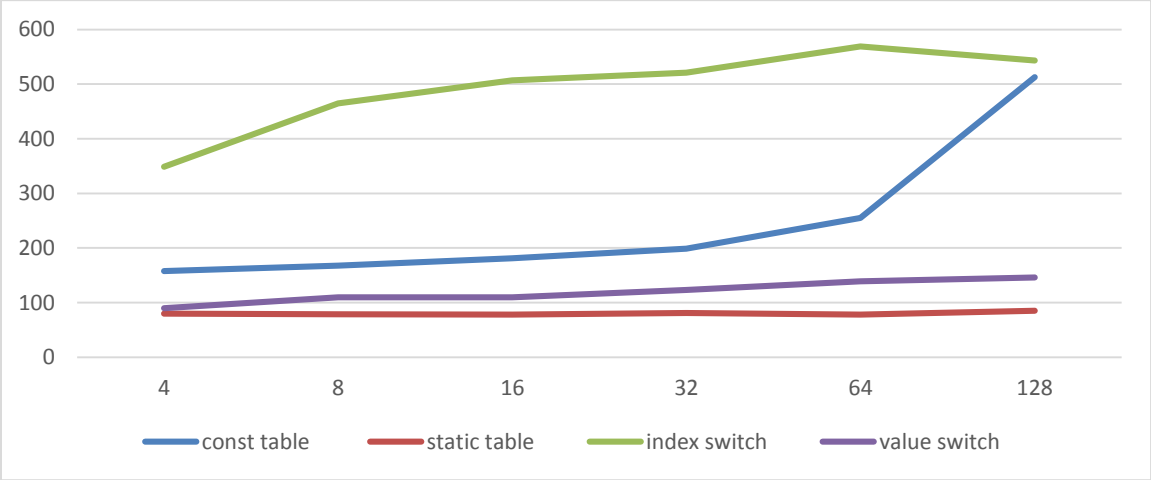
3.1. The tests

To evaluate our solution, we use an automatic test available on Github based on 4 different methods using enumerations containing between 4 to 128 enumeration values and multiple compilers: Visual Studio 2010, 2013 and 2015 preview; GCC 4.8.1; Intel Compiler 15; and Clang 3.5. The input set is generated ahead of measurement with pseudo random values including all the values of the input enumerations. Results are expressed in milliseconds on the ordinate axis. All the tests have been performed on a Haswell 4770K running Windows 7 64 bits.

We are studying four translations implementations:

- **`static table`**: This method is based on listing 2.2.2, indexing a table with a zero based enumeration.
- **`const table`**: This implementation varies from the **`static table`** case by declaring the table const only instead of static const.
- **`index switch`**: This method is based on listing 2.2.1, using a **`switch`** statement with a zero based enumeration.
- **`value switch`**: This implementation varies from the **`index switch`** case by using constants instead of a zero based enumeration.

3.2. Visual Studio 2013 initial results



Graph 3.2.1: Visual Studio 2013 results

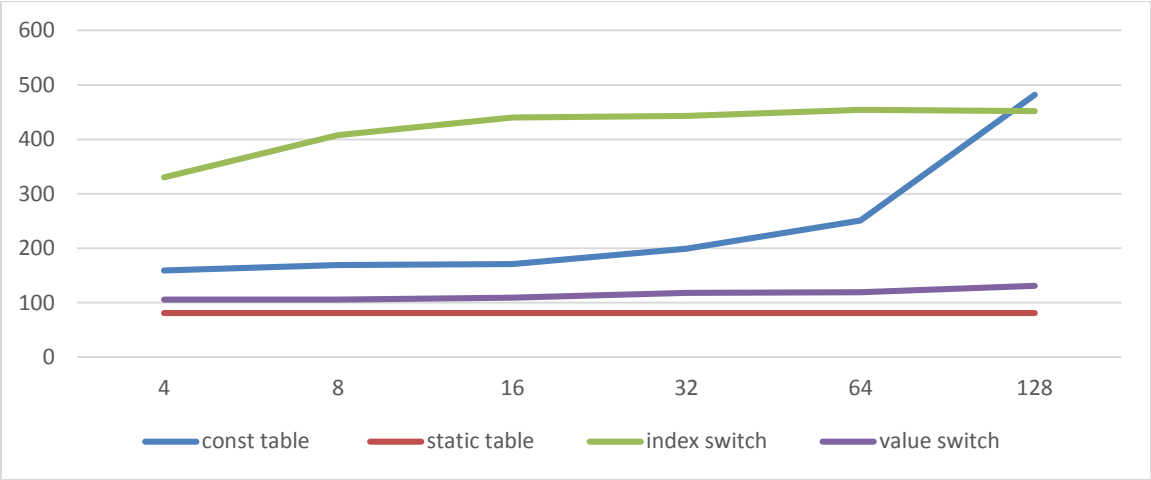
On Visual Studio 2013, the most efficient method is the static table method. Not only it is always faster but the performance are independent from the number of values in the enumeration.

A first surprise is that only changing the declaration of the translation table from **`static const`** to **`const`** only makes a huge performance difference. We will get back to this case in section 4.1.

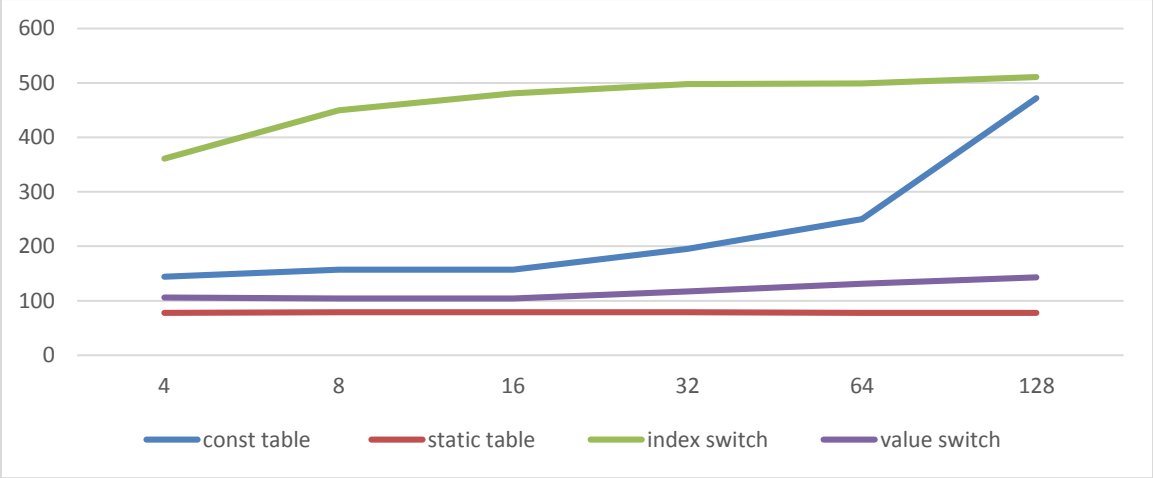
A second surprise is that the `index switch` and `value switch` cases perform very differently as well and zero based enumeration turns out to be a lot slower. We will study this case in depth in section 4.2.

3.3. More Visual Studio versions results

In this section we propose to look at different version of Visual Studio to validate our results.



Graph 3.3.1: Visual Studio 2010 results

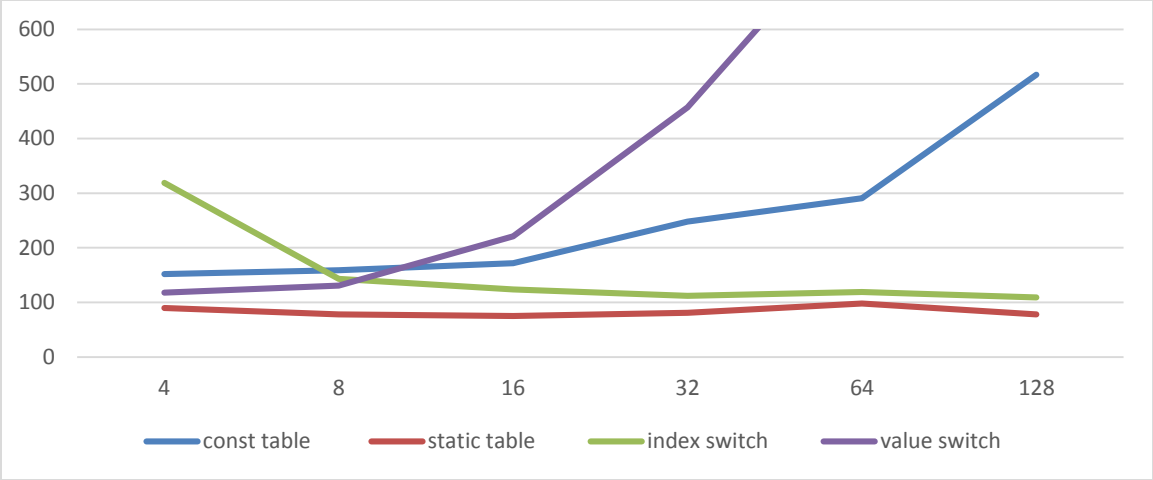


Graph 3.3.2: Visual Studio 2015 results

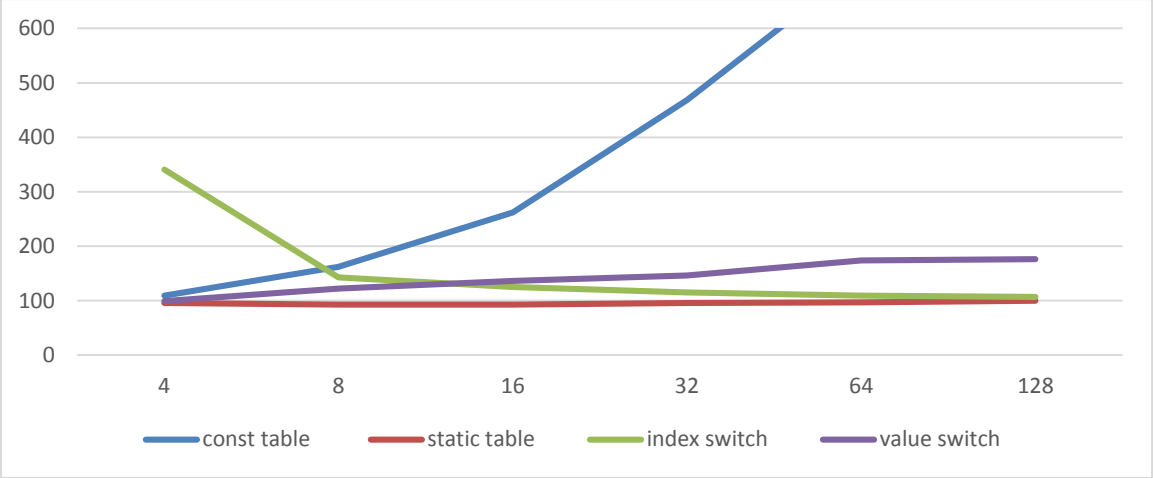
Certainly, we observe some performance variations but the performance characteristics are the same. Actually, by disabling the security check, `/GS-`, we can get back to close performance level across all Visual Studio versions.

We can conclude that these behaviors are not accidental and part of Visual Studio code debt and legacy.

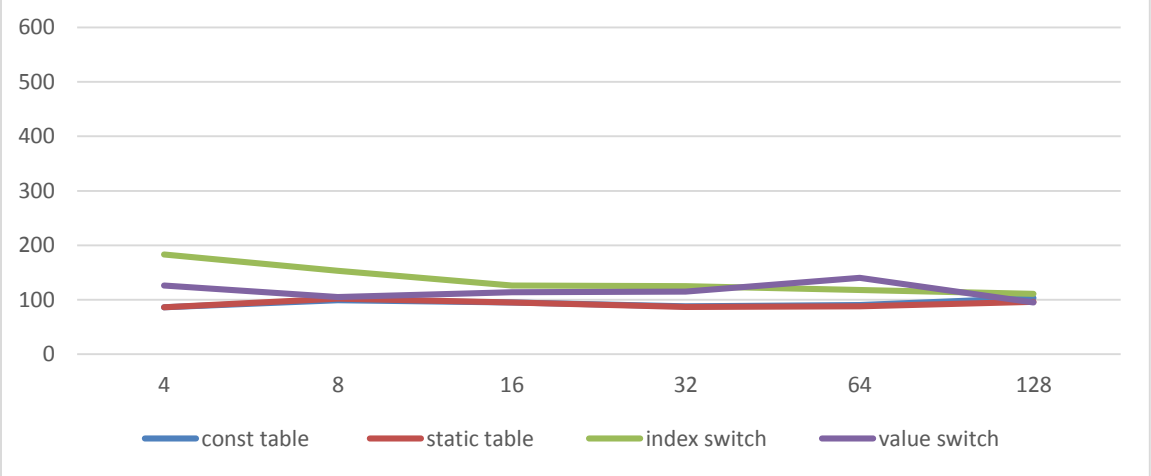
3.4. Clang, GCC, Intel Compiler results



Graph 3.4.1: Intel Compiler 2015 results



Graph 3.4.2: GCC 4.8.1 results



Graph 3.4.3: Clang 3.6.0 trunk results

A key observation from this article is that we can't rely on all compilers to behave the same way. Actually, only the **`static const`** table implementation displays the same

performance characteristic across compilers and equivalent performance levels. When considering performance, using a `static const` table implementation is the only valid choice.

The `const` table implementation follows the same performance characteristics on GCC, ICC and Visual Studio. Only with Clang it behaves identically as the `static const` case which looking at the assembly we can confirm that both cases are compiled exactly the same way with Clang.

Whether Clang behaviors is right or not, the generated code is by far fastest than any other compiler in this experiment. Actually, if all compilers were behaving the same way, we could conclude that performance is not a relevant criterion to implement a translation.

However, the `const` table case, the `index switch` and the `value switch` are all performance cliff depending on the used compiler.

- `const table` cliffs on GCC, ICC and Visual Studio.
- `index switch` cliffs on Visual Studio but also all the compilers when the number of enumeration values is small.
- `value switch` cliffs on GCC and is generally a bad performer.

For performance, we need to implement translation using a `static const` table.

4. Assembly analysis

4.1. static const vs const translation table

We observed in section 3 that declaring the translation table **`static const`** or **`const`** makes a huge difference.

To attempt to understand this difference, an important factor to take into account is to understand the C++ semantic differences between **`static const`** and **`const`**. Anything declared **`static`** in C++ is nothing more than a global. Anything declared **`const`** is just another member of the function code.

Logically, if the compiler follows the C++ semantic, when we use **`static const`** the data of this table is placed into a data segment which listing 4.1.2 confirms. However, when we use **`const`** only then the table data is supposed to remain with the instruction code, which effectively happens with Visual C++ 2013 as shown in listing 4.1.4.

```
translated static_table_translate(index Index)
{
	static translated const Table[] =
	{
		TRANSLATED_A,		// INDEX_A
		TRANSLATED_B,		// INDEX_B
		TRANSLATED_C,		// INDEX_C
		TRANSLATED_D		// INDEX_D
	};

	static_assert(
		sizeof(Table) / sizeof(Table[0]) == INDEX_COUNT,
		"The translation table needs to be updated.");
	assert(Index < INDEX_COUNT);

	return Table[Index];
}
```

Listing 4.1.1: Compare the enumeration and the implicitly sized array sizes in a static assert to make sure the translation table handles all cases.

```
Index$ = 8
?static_table_translate@translation4@@YA?AW4translated@1@W4index@1@@Z PROC ;
translation4::static_table_translate, COMDAT

; 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@4QBW
432@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 4.1.2: Visual C++ 2013 assembly of a translation function based on a const table.

```
translated const_table_translate(index Index)
{
       translated const Table[] =
       {
              TRANSLATED_A,        // INDEX_A
              TRANSLATED_B,        // INDEX_B
              TRANSLATED_C,        // INDEX_C
              TRANSLATED_D         // INDEX_D
       };

       static_assert(
              sizeof(Table) / sizeof(Table[0]) == INDEX_COUNT,
              "The translation table needs to be updated.");
       assert(Index < INDEX_COUNT);

       return Table[Index];
}
```

Listing 4.1.3: Compare the enumeration and the implicitly sized array sizes in a static assert to make sure the translation table handles all cases.

```
Index$ = 48
?const_table_translate@translation4@@YA?AW4translated@1@W4index@1@@Z PROC ;
translation4::const_table_translate, COMDAT

; 34   :      {

$LN4:
       sub    rsp, 40                             ; 00000028H
       mov    rax, QWORD PTR __security_cookie
       xor    rax, rsp
       mov    QWORD PTR __$ArrayPad$[rsp], rax
       movdqa xmm0, XMMWORD PTR __xmm@00008aef00002c35000001c20000a0e7

; 35   :             const translated Table[] =
; 36   :             {
; 37   :                    TRANSLATED_A,        // INDEX_A
; 38   :                    TRANSLATED_B,        // INDEX_B
; 39   :                    TRANSLATED_C,        // INDEX_C
; 40   :                    TRANSLATED_D         // INDEX_D
; 41   :             };
; 42   :
; 43   :             static_assert(sizeof(Table) / sizeof(translated) == INDEX_COUNT,
"The translation table needs to be updated.");
; 44   :             assert(Index < INDEX_COUNT);
; 45   :
; 46   :             return Table[Index];

       movsxd rax, ecx
       movdqu XMMWORD PTR Table$[rsp], xmm0
       mov    eax, DWORD PTR Table$[rsp+rax*4]

; 47   :      }

       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 ;
translation4::const_table_translate
```

Listing 4.1.4: Visual C++ 2013 assembly of a translation function based on a const table.

As a result despite changing a single C++ key word, the assembly is really different because the logic is actually different too.

Reading listing 4.1.4, we realize that the **const** implementation is done relying on constants folding. The compiler is filling XMM registers with constants which is fine but it requires a complex instruction logic to access each constants.

When we enter a function there will be a stack allocation big enough so that all the variables in it could fit there. An additional issue with translation table declared **const** is that the function might get bigger so it will consume more CPU instructions cache, hence evicting more code resulting in more instruction cache misses.

Using **static const** is choosing to fight against instructions cache evictions. With static, the table goes into a data segment so effectively the function remains compact and evict less. The downside is that we may cache miss twice. Once on the function call (in the L1 instruction cache) and once on the table fetch (in L1 data cache).

We may be able to consider that it is fine to add pressure on the data cache in random plumbing code which is less likely to be very busy here while it is super busy in optimized data transformation code.

It could be tempting to jump into conclusions and assume that we should declare any constant static const. This is drawing conclusions too quickly! Modern processors (both CPUs and GPUs) make use of constants folding and it's typically a great strategy as long as the constants are not indexed.

For example, Haswell CPUs optimize throughput for constant folding:
- **MOVAPS/D xmm, xmm** latency: 1 throughput: 1
- **MOVAPS/D xmm, m128** latency: 3 throughput: 0.5

4.2. Value switch vs index switch

In section 3, we identified that **value switch** and **index switch** translation implementations were behaving very differently in an unexpected manner with Visual Studio as the **index switch** implementation is a lot slower.

```
translated index_switch_translate(index Index)
{
        switch(Index)
        {
        case INDEX_A: return TRANSLATED_A;
        case INDEX_B: return TRANSLATED_B;
        case INDEX_C: return TRANSLATED_C;
        case INDEX_D: return TRANSLATED_D;
        }
}
```

Listing 4.2.1: Compare the enumeration and the implicitly sized array sizes in a static assert to make sure the translation table handles all cases.

```
Index$ = 8
?index_switch_translate@translation4@@YA?AW4translated@1@W4index@1@@Z PROC ;
translation4::index_switch_translate, COMDAT

; 100  :             switch(Index)
       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

; 105  :             case INDEX_D: return TRANSLATED_D;
       mov    eax, 35567                          ; 00008aefH

; 106  :             }
; 107  :      }
       ret    0
$LN2@index_swit:

; 104  :             case INDEX_C: return TRANSLATED_C;
       mov    eax, 11317                          ; 00002c35H

; 106  :             }
; 107  :      }
       ret    0
$LN3@index_swit:

; 103  :             case INDEX_B: return TRANSLATED_B;
       mov    eax, 450                            ; 000001c2H

; 106  :             }
; 107  :      }
       ret    0
$LN4@index_swit:

; 101  :             {
; 102  :             case INDEX_A: return TRANSLATED_A;
       mov    eax, 41191                          ; 0000a0e7H
$LN5@index swit:

; 106  :             }
; 107  :      }
       ret    0
?index_switch_translate@translation4@@YA?AW4translated@1@W4index@1@@Z ENDP ;
translation4::index_switch_translate
```

Listing 4.2.2: Compare the enumeration and the implicitly sized array sizes in a static assert to make sure the translation table handles all cases.

```
index value_switch_translate(translated Value)
{
       switch(Value)
       {
       case TRANSLATED_A: return INDEX_A;
       case TRANSLATED_B: return INDEX_B;
       case TRANSLATED_C: return INDEX_C;
       case TRANSLATED_D: return INDEX_D;
       }
}
```

Listing 4.2.3: Compare the enumeration and the implicitly sized array sizes in a static assert to make sure the translation table handles all cases.

```
Value$ = 8
```

```
?value_switch_translate@translation4@@YA?AW4index@1@W4translated@1@@Z PROC ;
translation4::value_switch_translate, COMDAT

; 111  :             switch(Value)
       cmp    ecx, 450                        ; 000001c2H
       je     SHORT $LN3@value_swit
       cmp    ecx, 11317                      ; 00002c35H
       je     SHORT $LN2@value_swit
       cmp    ecx, 35567                      ; 00008aefH
       je     SHORT $LN1@value_swit
       cmp    ecx, 41191                      ; 0000a0e7H
       jne    SHORT $LN5@value_swit

; 112  :             {
; 113  :             case TRANSLATED_A: return INDEX_A;
       xor    eax, eax

; 117  :             }
; 118  :      }
       ret    0
$LN1@value_swit:

; 116  :             case TRANSLATED_D: return INDEX_D;
       mov    eax, 3

; 117  :             }
; 118  :      }
       ret    0
$LN2@value_swit:

; 115  :             case TRANSLATED_C: return INDEX_C;
       mov    eax, 2

; 117  :             }
; 118  :      }
       ret    0
$LN3@value_swit:

; 114  :             case TRANSLATED_B: return INDEX_B;
       mov    eax, 1
$LN5@value_swit:

; 117  :             }
; 118  :      }
       ret    0
?value_switch_translate@translation4@@YA?AW4index@1@W4translated@1@@Z ENDP ;
translation4::value_switch_translate
```

Listing 4.2.4: Compare the enumeration and the implicitly sized array sizes in a static assert to make sure the translation table handles all cases.

Listing 4.2.2 and 4.2.4 shows that Visual Studio implements the switch statement very similarly. First we have section of code testing which case we are at and then we have a section of code handling either case. The junction between the two sections is made using a jump instruction.

In fact, the only difference is in the testing section as highlighted in listing 4.2.5.

```
; index switch
       test   ecx, ecx                        ; latency:1 thoughtput:0.25
       je     SHORT $LN4@index_swit
       dec    ecx                             ; latency:6 thoughtput:1
       je     SHORT $LN3@index swit
       dec    ecx                             ; latency:6 thoughtput:1
       je     SHORT $LN2@index_swit
```

```
        dec     ecx                                     ; latency:6 thoughtput:1
        jne     SHORT $LN5@index_swit

; value switch
        cmp     ecx, 450                                ; latency:1 thoughtput:0.25
        je      SHORT $LN3@value_swit
        cmp     ecx, 11317                              ; latency:1 thoughtput:0.25
        je      SHORT $LN2@value_swit
        cmp     ecx, 35567                              ; latency:1 thoughtput:0.25
        je      SHORT $LN1@value_swit
        cmp     ecx, 41191                              ; latency:1 thoughtput:0.25
        jne     SHORT $LN5@value_swit
```

Listing 4.2.5: Test assembly for value switch and index switch with latencies and throughput on Haswell

Looking at Agner's instruction table for Haswell, we see that the code may look like the same but the latencies and throughputs are very different. Furthermore, the assembly generated for index switch has a result dependency chain on **ecx** which makes it unfriendly for instructions parallelism.

5. Translation table robustness

5.1. Detecting the addition or removal of enumeration values at compilation time

One common bug with OpenGL is that the API evolves and the software follows this evolution. For example, new shader stages and new texture formats have been added to OpenGL.

When implementing these changes a first step is to add a new entry in the associated software zero-based enumeration. This change implies that everywhere this enumeration is used; we introduced a runtime bug where the new entry is not correctly handle.

With translation tables, a solution is to generate compile time errors everywhere this modified enumeration is used. This is accomplished thanks to **`static_assert`** and implicitly sized arrays.

```
enum stage
{
	STAGE_VERTEX = 0,
	STAGE_TESS_CONTROL,
	STAGE_TESS_EVALUATION,
	STAGE_GEOMETRY,
	STAGE_FRAGMENT,
	STAGE_LAST = STAGE_FRAGMENT
};

GLenum translate(stage Stage)
{
	// Don't set a size to be sure it's implicitly sized.
	static GLenum const Table[] =
	{
		GL_VERTEX_SHADER,		// STAGE_VERTEX
		GL_TESS_CONTROL_SHADER,		// STAGE_TESS_CONTROL
		GL_TESS_EVALUATION_SHADER,	// STAGE_TESS_EVALUATION
		GL_GEOMETRY_SHADER,		// STAGE_GEOMETRY
		GL_FRAGMENT_SHADER		// STAGE_FRAGMENT
	};

	static_assert(sizeof(Table) / sizeof(Table[0]) == STAGE_LAST + 1,
		"OPENGL ERROR: The translation table for 'stage' needs to be updated.");

	return Table[Stage];
};
```

Listing 5.1.1: Compare the enumeration and the implicitly sized array sizes in a static assert to make sure the translation table handles all cases.

One issue with this design is that it may happen that a change will remove an enumeration value and add a new one at the same time. With OpenGL it's not a common case but translation tables have a larger scope.

5 2. Detecting translation runtime input errors

It is often useful to introduce an invalid value to an enumeration to avoid using a valid value as an invalid value that would be misleading.

```
enum stage
{
```

```
        STAGE_INVALID = -1,
        STAGE_VERTEX = 0,
        STAGE_TESS_CONTROL,
        STAGE_TESS_EVALUATION,
        STAGE_GEOMETRY,
        STAGE_FRAGMENT,
        STAGE_LAST = STAGE_FRAGMENT
};

class programPoll
{
        struct program
        {
                GLuint Name;
                stage Stage;
                …
        };

        std::vector<program> Polls;

        void release(std::size_t ProgramIndex)
        {
                assert(ProgramIndex < this->Polls.size());
                program & Program = this->Polls[];
                glDeletePrograms(1, &Program.Name);
                Program.Name = 0;
                Program.Stage = STAGE_INVALID;
        }
};
```

Listing 5.2.1: Example of usage of an explicit invalid enumeration value.

An argument against this idea is that in listing 5.2.1, we could rely on **`Program.Name`** to be equal to zero to detect invalid cases. This is overloading the semantic of zero for a program name with an additional semantic. In fact, with OpenGL compatibility profile, zero is a perfectly valid program name that enables the fixed function pipeline. Hence, using an explicit invalid value makes the code easier to understand.

A good invalid value is **`-1`** which is equivalent to **`0xFFFFFFFF`** in hexadecimal. First, it's the last possible value in an enumeration so the entire range remains available. Second, when working with zero based enumeration, it's conceptually an invalid value. This property makes it likely to fail, generate an error, throw an assert or crash when incorrectly used which is exactly what we should look for an invalid value so that we can detect the problem early.

```
GLuint GetProgramName(stage Stage) const
{
        assert(Stage != STAGE_INVALID);
        return this->ProgramNames[Stage];
}
```

Listing 5.2.2: Table access using a zero based enumeration

In listing 5.2.2, in debug build we will fail on assert if we use an invalid value. In release build, it's very likely that we violently crash on **`ProgramNames`** access: This is a good thing! If we didn't had an invalid value, then Stage would necessarily be a valid value and the program would fail down later in the code execution or worse it would not fail which makes fixing bugs a lot harder.

```
enum stage
{
        STAGE_INVALID = -1,
```

```
        STAGE_VERTEX = 0,
        STAGE_TESS_CONTROL,
        STAGE_TESS_EVALUATION,
        STAGE_GEOMETRY,
        STAGE_FRAGMENT,
        STAGE_LAST = STAGE_FRAGMENT
};

GLenum translate(stage Stage)
{
        static GLenum const Table[] =
        {
                GL_VERTEX_SHADER,           // STAGE_VERTEX
                GL_TESS_CONTROL_SHADER,     // STAGE_TESS_CONTROL
                GL_TESS_EVALUATION_SHADER,  // STAGE_TESS_EVALUATION
                GL_GEOMETRY_SHADER,         // STAGE_GEOMETRY
                GL_FRAGMENT_SHADER,         // STAGE_FRAGMENT
        };

        static_assert(sizeof(Table) / sizeof(Table[0]) == STAGE_LAST + 1,
                "OPENGL ERROR: The translation table for 'stage' needs to be updated.");

        assert(Stage != STAGE_INVALID); // OpenGL ERROR: Invalid Stage value
        return Table[Stage];
};
```

Listing 5.2.3: Failing on an invalid value in a translation.

5 3. Limiting the translation range

It is possible that a zero-based enumeration is going to be used for multiple contexts but the contexts vary enough that not all the values would be valid for all contexts.

A first possibility is to create separated zero-based enumerations per context but this idea requires additional translation between zero-based enumerations.

An alternative is to mark ranges within the enumerations using **_FIRST** and **_LAST** enumeration value aliases as shown in listing 5.3.1.

```
enum stage
{
        STAGE_COMPUTE = 0,
        STAGE_VERTEX,
        STAGE_GFX_FIRST = STAGE_VERTEX
        STAGE_TESS_CONTROL,
        STAGE_TESS_EVALUATION,
        STAGE_GEOMETRY,
        STAGE_FRAGMENT,
        STAGE_GFX_LAST = STAGE_FRAGMENT
};

GLenum translateGraphicsStage(stage Stage)
{
        static GLenum const Table[] =
        {
                GL_VERTEX_SHADER,           // STAGE_VERTEX
                GL_TESS_CONTROL_SHADER,     // STAGE_TESS_CONTROL
                GL_TESS_EVALUATION_SHADER,  // STAGE_TESS_EVALUATION
                GL_GEOMETRY_SHADER,         // STAGE_GEOMETRY
                GL_FRAGMENT_SHADER,         // STAGE_FRAGMENT
        };

        static_assert(
```

```
                sizeof(Table) / sizeof(Table[0]) == STAGE_GFX_LAST - STAGE_GFX_FIRST + 1,
                "OPENGL ERROR: The translation table for 'stage' needs to be updated.");

        // OpenGL ERROR: Invalid range for Stage value
        assert(Stage >= STAGE_GFX_FIRST && Stage <= STAGE_GFX_LAST);

        return Table[Stage - STAGE_GFX_FIRST];
};
```

Listing 5.3.1: Limiting the valid range of a zero based enumeration.

An additional value for this design in the example listing 5.3.1 is that if OpenGL adds a new graphics shader stage on the future, then almost all the code will remain correct, arrays will scale automatically to the correct size and the compile will fail on the **`static_assert`** showing what code needs to be updated.

6. Extending translation table functionality for runtime decisions

6.1. Baking translation tables at runtime

A nice and pretty compile time **`static const`** table is just not a solution that is going to fit all scenarios. It should be the default solution to choose but in many scenarios, it's just not flexible enough.

For instance, a real life scenario is that we would want to use the same code base for OpenGL and OpenGL ES but while OpenGL 4.5 supports tessellation and geometry shader stages, OpenGL ES 3.1 doesn't. Listing 6.1.1 shows a native implementation to support this behavior.

```
enum stage
{
	STAGE_VERTEX = 0,
	STAGE_TESS_CONTROL,
	STAGE_TESS_EVALUATION,
	STAGE_GEOMETRY,
	STAGE_FRAGMENT,
	STAGE_LAST = STAGE_FRAGMENT
};

// Making an initialization time decision in the rendering loop
GLenum translate(stage Stage, bool IsProfileES)
{
	static GLenum const Table[] =
	{
		GL_VERTEX_SHADER,                                  // STAGE_VERTEX
		ProfileES ? GL_NONE : GL_TESS_CONTROL_SHADER,      // STAGE_TESS_CONTROL
		ProfileES ? GL_NONE : GL_TESS_EVALUATION_SHADER,   // STAGE_TESS_EVALUATION
		ProfileES ? GL_NONE : GL_GEOMETRY_SHADER,          // STAGE_GEOMETRY
		GL FRAGMENT SHADER                                 // STAGE FRAGMENT
	};

	static_assert(sizeof(Table) / sizeof(Table[0]) == STAGE_LAST + 1,
		"OPENGL ERROR: The translation table for 'stage' needs to be updated.");

	return Table[Stage];
};
```

Listing 6.1.1: Compare the enumeration and the implicitly sized array sizes in a static assert to make sure the translation table handles all cases.

Listing 6.1.1 illustrates a bad practice: In this OpenGL scenario, translations are going to happen in the rendering loop when precisely we need to most performance. Adding the **`ProfileES`** parameter implies cost in the rendering loop despite this decision being made during the OpenGL context creation. An alternative is to bake the translation table at initialization time, in this scenario, right after creating the OpenGL context.

```
GLenum translate(std::array<GLenum, LAST SHADER + 1> const & Table, stage Stage)
{
	return Table[Stage];
};
```

Listing 6.1.2: Example translation using a runtime baked table

In listing 6.1.2, we pass the translation table through a parameter but the source of table doesn't really matter. We measured no performance difference between using an

std::array or a **static const** implicitly sized array on all compilers. At the assembly level, in all cases accessing the table is passing a memory address.

6 2. Detecting translation runtime output errors

As soon as we introduce runtime decisions for translation table creation, we may output invalid values. In listing 6.2.1, the output of the translation is checked using **assert(Translated != INVALID_ENUM)** allowing to detect issues as early as possible.

```
enum stage
{
	STAGE_VERTEX = 0,
	STAGE_TESS_CONTROL,
	STAGE_TESS_EVALUATION,
	STAGE_GEOMETRY,
	STAGE_FRAGMENT,
	STAGE_LAST = STAGE_FRAGMENT
};

static GLenum const INVALID_ENUM = static_cast<GLenum>(0xDEADF00D);

GLenum translate(std::array<GLenum, LAST_SHADER + 1> const & Table, stage Stage)
{
	GLenum const Translated = Table[Stage];
	assert(Translated != INVALID_ENUM); // OpenGL ERROR: Invalid output enum value
	return Translated;
};
```

Listing 6.2.1: Compare the enumeration and the implicitly sized array sizes in a static assert to make sure the translation table handles all cases.

The definition of the invalid output value needs to be done carefully. Initially, we were using **GL_NONE** for this value however ever **GL_NONE** is potentially a valid value and other OpenGL values are also defined with the same value as **GL_NONE** which is zero: **GL_ZERO**, **GL_NO_ERROR**, **GL_FALSE**, **GL_POINTS**. A second idea was to use -1 which translates in **0xFFFFFFFF** however this value is also pretty common in OpenGL: **GL_INVALID_INDEX**, **GL_ALL_BARRIER_BITS**, **GL_ALL_SHADER_BITS** and **GL_TIMEOUT_IGNORED**.

We finally fall backed to the magic debug value **0xDEADF00D** because:

- The value is unused in OpenGL.
- It's a popular invalid value among programmers.
- It's a 32 bits values while OpenGL enumeration values are 16 bits values except for flags.
- It's big enough that if the value is used to access an array, it's very likely going to generate a bad memory access error.

A good invalid output value depends on the output type.

In listing 6.2.1, we lost the **static_assert** hence the capability to check that the translation table is up to date with the zero base enumeration. We can rescue this capability by moving it at the translation table initialization as shown in listing 6.2.2.

```
void init(std::array<GLenum, LAST_SHADER + 1> & Dest, bool IsProfileES)
{
	static GLenum const Table[] =
	{
```

```
                GL_VERTEX_SHADER,                                   // STAGE_VERTEX
                ProfileES ? GL_NONE : GL_TESS_CONTROL_SHADER,       // STAGE_TESS_CONTROL
                ProfileES ? GL_NONE : GL_TESS_EVALUATION_SHADER,    // STAGE_TESS_EVALUATION
                ProfileES ? GL_NONE : GL_GEOMETRY_SHADER,           // STAGE_GEOMETRY
                GL_FRAGMENT_SHADER                                  // STAGE_FRAGMENT
        };

        static_assert(sizeof(Table) / sizeof(Table[0]) == STAGE_LAST + 1,
                "OPENGL ERROR: The translation table for 'stage' needs to be updated.");

        memcpy(&Dest[0], &Table, sizeof(Table));

        return Table[Stage];
};
```

Listing 6.2.2: Compare the enumeration and the implicitly sized array sizes in a static assert to make sure the translation table handles all cases.

7. Pitfalls

7.1. Faster than the fastest translation: no translation

It happens that we write complex functions with multiple code paths where only one path is executed despite that branch being decided at compilation time as shown in listing 7.1.

```
enum target
{
	UNIFORM_BUFFER,
	SHADER_STORAGE_BUFFER,
	TRANSFORM_FEEDBACK_BUFFER,
	TARGET_LAST = TRANSFORM_FEEDBACK_BUFFER
};

void bindBuffer(target Target, GLuint Unit, GLuint Buffer)
{
	switch(Target)
	{
	case UNIFORM_BUFFER:
		assert(Unit < this->UniformBufferBound.size());
		if(this->UniformBufferBound[Unit] == Buffer)
			return;
		this->UniformBufferBound[Unit] = Buffer;
		break;
	case SHADER_STORAGE_BUFFER:
		assert(Unit < this->ShaderStorageBufferBound.size());
		if(this->ShaderStorageBufferBound[Unit] == Buffer)
			return;
		this->ShaderStorageBufferBound[Unit] = Buffer;
		break;
	case TRANSFORM_FEEDBACK_BUFFER:
		assert(Unit < this->TransformFeedbackBufferBound.size());
		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]);
	...
}
```

Listing 7.1: Example of useless last minute runtime decision

Issues:

- Each time we call **`bindBuffer`**, the function code is fetched in the CPU L1 instruction cache causing the eviction of the oldest code and a potential code cache miss. However, it's very possible that the rendering code will never use shader storage buffer or transform feedback buffer so fetching such code is pure instruction cache pollution.
- This code design require a switch, hence an expensive runtime decision while really the decision is taken in the foo function code already.

- Exposing API dependent code (**GL_UNIFORM_BUFFER**) is ugly as it makes the API less reusable.

We can create dedicated functions per code paths avoiding CPU L1 instruction cache pollution and avoiding runtime decisions as illustrated in listing 7.2.

```
void bindUniformBuffer(std::size_t Unit, std::uint32_t Buffer);
{
	assert(Unit < this->UniformBufferBound.size());

	if(this->UniformBufferBound[Unit] == Buffer)
		return;
	this->UniformBufferBound[Unit] = Buffer;

	glBindBufferBase(GL_UNIFORM_BUFFER,
		static_cast<GLuint>(Unit), static_cast<GLuint>(Buffer));
}

void bindShaderStorageBuffer(std::size_t Unit, std::uint32_t Buffer);
{
	assert(Unit < this-> ShaderStorageBufferBound.size());

	if(this->ShaderStorageBufferBound[Unit] == Buffer)
		return;
	this->ShaderStorageBufferBound[Unit] = Buffer;

	glBindBufferBase(GL_SHADER_STORAGE_BUFFER,
		static_cast<GLuint>(Unit), static_cast<GLuint>(Buffer));
}

void bindTransformFeedbackBuffer(std::size_t Unit, std::uint32_t Buffer);
{
	assert(Unit < this-> TransformFeedbackBufferBound.size());

	if(this->TransformFeedbackBufferBound[Unit] == Buffer)
		return;
	this->TransformFeedbackBufferBound[Unit] = Buffer;

	glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER,
		static_cast<GLuint>(Unit), static_cast<GLuint>(Buffer));
}

void foo()
{
	...
	for(std::size_t i = 0; i < Buffers.size(); ++i)
		bindUniformBuffer(i, Buffers[i]);
	...
}
```

Listing 7.2: Using the code context to avoid translation

Typically, when the result of the translation could be determine at compilation time or should be optimized by the compiler then translation is the wrong tool to resolve the problem.

Tip: Consider inlining! Not only inline may remove the function calls but by making the functions smaller, they become better candidates for inlining.

7.2. Reordering of enumeration values

A major pitfall of translation tables is that reordering or renaming enumeration values will produce incorrect translation tables that will remain unnoticed. Unfortunately, there is no trivial and reliable solution to detect reordering or renaming.

However, the use of implicitly sized array and static assert allows to easily find across the code all the cases where translation tables are used how should be updated. For example, a good search string is '**STAGE_LAST + 1,**' or '**== STAGE_LAST**'.

Additionally, using a source enumeration value in comment on front of each translated values improve the readability of the translation and the ability of detecting mismatches with the original enumeration.

```
enum stage
{
	STAGE_VERTEX = 0,
	STAGE_TESS_CONTROL,
	STAGE_TESS_EVALUATION,
	STAGE GEOMETRY,
	STAGE_FRAGMENT,
	STAGE_LAST = STAGE_FRAGMENT
};

GLenum translate(stage Stage)
{
	static GLenum const Table[] =
	{
		GL_VERTEX_SHADER,		// STAGE_VERTEX
		GL_TESS_CONTROL_SHADER,		// STAGE_TESS_CONTROL
		GL_TESS_EVALUATION_SHADER,	// STAGE_TESS_EVALUATION
		GL_GEOMETRY_SHADER,		// STAGE_GEOMETRY
		GL_FRAGMENT_SHADER,		// STAGE_FRAGMENT
	};

	static_assert(sizeof(Table) / sizeof(Table[0]) == STAGE_LAST + 1,
		"OPENGL ERROR: The translation table for 'stage' needs to be updated.");

	return Table[Stage];
};
```

Listing 7.2.1: Comments referencing the enumeration values source.

Conclusions

It is always pretty hard to conclude without falling into excessive generalizations.

Nevertheless, translation using table with a zero-based enumeration is a solid base to build robust code failing early and providing constant cross-compiler and execution performances.

As discussed in this article, a lot of tweaks are possible to the basic implementation to adapt it to specific scenarios and keep constant performances and ensure robustness in code context.

```
enum stage
{
	STAGE_VERTEX = 0,
	STAGE_TESS_CONTROL,
	STAGE_TESS_EVALUATION,
	STAGE_GEOMETRY,
	STAGE_FRAGMENT,
	STAGE_LAST = FRAGMENT
};

GLenum translate(stage Stage)
{
	static GLenum const Table[] =
	{
		GL_VERTEX_SHADER,			// STAGE_VERTEX
		GL_TESS_CONTROL_SHADER,		// STAGE_TESS_CONTROL
		GL_TESS_EVALUATION_SHADER,	// STAGE_TESS_EVALUATION
		GL_GEOMETRY_SHADER,			// STAGE_GEOMETRY
		GL_FRAGMENT_SHADER,			// STAGE_FRAGMENT
	};

	static_assert(
		sizeof(Table) / sizeof(GLenum) == STAGE_LAST - STAGE_FIRST + 1,
		"OPENGL ERROR: The translation table for 'stage' needs to be updated.");

	return Table[Stage];
};
```

Basic translation implementation using a table and a zero-based enumeration.

In OpenGL alone, there is a lot of use cases for translations: Texture formats, texture targets, sampler filtering, sampler wrapping, buffer usages, stencil operations, blend operations, cull modes, vertex attribute formats, primitive types, etc.

Obviously, translations are not limited to OpenGL and they have many more use cases; future work that we leave for our imaginations.

Special thanks to Stephanie Hurlburt for the review of this article.

Simplicity is the ultimate sophistication.
Leonardo Da Vinci