Increasing [Effective and robust translation tables with zero based enumerations 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 6

2.1. Definition 6

2.2. Translation implementations 6

3. Performances 7

3.1. The tests 7

3.2. Results 8

3.3. Value switch vs index switch 10

3.4. static const vs const translation table 12

4. Translation table robustness 15

4.1. Detecting the addition or removal of enumeration values at compilation time 15

4 2. “Future proof” code by failing at compilation time 16

5. Extending translation table functionality for runtime decisions 16

5.1. Baking translation tables at runtime 16

5.2. Dynamic translation tables at runtime 16

5.3. Performance comparison between static const table, fixed-size arrays and dynamic arrays. 16

6. Usages and abuses of translations 17

6.1. Use code context to avoid unnecessary translations 17

# 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 enumerations 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 everyday 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;  }  } |

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 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 2.2.2. 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](https://github.com/Groovounet/cpp-experiments/blob/master/test_translation_table.cpp) 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. 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. Results

Graph 3.2.1. Visual Studio 2010 results

Graph 3.2.2: Visual Studio 2013 results

Graph 3.2.3: Visual Studio 2015 results

Graph 3.2.4: Intel Compiler 2015 results

Graph 3.2.5: GCC 4.8.1 results

Graph 3.2.6: Clang 3.6.0 trunk results

## 3.4. Analyze

## 3.5. Value switch vs index switch

|  |
| --- |
| 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 2.4.1. 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 2.4.2. 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 2.4.3. 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 2.4.4. Compare the enumeration and the implicitly sized array sizes in a static assert to make sure the translation table handles all cases.

## 3.6. static const vs const translation table

A very interesting question regarding the translation table declaration is whether it should be declared static const or const.

To attempt to answer this question, an important factor to take into account is the semantic of 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 moved a data segment which listing 2.2.4 confirms. However, when we use const then then data if this table is supposed to remain with the instruction code, which effectively happens with Visual C++ 2013 as shown in listing 2.3.1.

|  |
| --- |
| translated static\_table\_translate(index Index)  {  static const translated Table[] =  {  TRANSLATED\_A, // INDEX\_A  TRANSLATED\_B, // INDEX\_B  TRANSLATED\_C, // INDEX\_C  TRANSLATED\_D // INDEX\_D  };  static\_assert(  sizeof(Table) / sizeof(translated) == INDEX\_COUNT,  "The translation table needs to be updated.");  assert(Index < INDEX\_COUNT);  return Table[Index];  } |

Listing 2.5.1. 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@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 2.5.2. Visual C++ 2013 assembly of a translation function based on a const table.

|  |
| --- |
| translated const\_table\_translate(index Index)  {  const translated Table[] =  {  TRANSLATED\_A, // INDEX\_A  TRANSLATED\_B, // INDEX\_B  TRANSLATED\_C, // INDEX\_C  TRANSLATED\_D // INDEX\_D  };  static\_assert(  sizeof(Table) / sizeof(translated) == INDEX\_COUNT,  "The translation table needs to be updated.");  assert(Index < INDEX\_COUNT);  return Table[Index];  } |

Listing 2.5.3. 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 2.5.4. 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 2.3.1, 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 evict 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.

|  |
| --- |
| ?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.

# 4. Translation table robustness

## 4.1. 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  {  VERTEX = 0,  TESS\_CONTROL,  TESS\_EVALUATION,  GEOMETRY,  FRAGMENT, STAGE\_LAST = FRAGMENT  };  GLenum translate(stage Stage)  {  static const GLenum Table[] = // Don’t set a size to be sure it’s implicitly sized.  {  GL\_VERTEX\_SHADER, // VERTEX  GL\_TESS\_CONTROL\_SHADER, // TESS\_CONTROL  GL\_TESS\_EVALUATION\_SHADER, // TESS\_EVALUATION  GL\_GEOMETRY\_SHADER, // GEOMETRY  GL\_FRAGMENT\_SHADER, // FRAGMENT  };  static\_cast(  sizeof(Table) / sizeof(GLenum) == LAST\_SHADER + 1,  "OPENGL ERROR: The translation table for 'stage' needs to be updated.");  return Table[Stage];  }; |

Listing 3.1.1. 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.

## 4 2. “Future proof” code by failing at compilation time

TODO

# 5. Extending translation table functionality for runtime decisions

## 5.1. Baking translation tables at runtime

|  |
| --- |
| 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;  }; |

Listing 4.1.1. Example of runtime statically sized array to perform translation

## 5.2. Dynamic translation tables at runtime

|  |
| --- |
| 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;  }; |

Listing 4.2.1. Example of runtime dynamically sized array to perform translation

## 5.3. Performance comparison between static const table, fixed-size arrays and dynamic arrays.

TODO

# 6. Usages and abuses of translations

## 6.1. Use code context to avoid unnecessary translations

|  |
| --- |
| enum target  {  UNIFORM\_BUFFER,  SHADER\_STORAGE\_BUFFER,  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 6.1.1. Example of useless last minute runtime decision

|  |
| --- |
| enum target  {  UNIFORM\_BUFFER,  SHADER\_STORAGE\_BUFFER,  TRANSFORM\_FEEDBACK\_BUFFER  };  void bindUniformBuffer(GLuint Unit, GLuint Buffer);  {  assert(Unit < this->UniformBufferBound.size());  if(this->UniformBufferBound[Unit] == Buffer)  return;  glBindBufferBase(GL\_UNIFORM\_BUFFER, Unit, Buffer);  this->UniformBufferBound[Unit] = Buffer;  }  void bindShaderStorageBuffer(GLuint Unit, GLuint Buffer);  {  assert(Unit < this-> ShaderStorageBufferBound.size());  if(this->ShaderStorageBufferBound[Unit] == Buffer)  return;  glBindBufferBase(GL\_SHADER\_STORAGE\_BUFFER, Unit, Buffer);  this->ShaderStorageBufferBound[Unit] = Buffer;  }  void bindTransformFeedbackBuffer(GLuint Unit, GLuint Buffer);  {  assert(Unit < this-> TransformFeedbackBufferBound.size());  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]);  ...  } |

Listing 6.1.2. Example of code context usage to avoid last minute runtime code decision