bindings-generator.lisp

Erik Winkels

August 13, 2009

Contents

1 Functions

This function converts the OGRE_CAPS_VALUE enum in the same way as is done in OgreRenderSystemCapabilities.h, namely: "#define CAPS_CATEGORY_SIZE 4", "#define OGRE_CAPS_BITSHIFT (32 - CAPS_CATEGORY_SIZE)" and "#define OGRE_CAPS_VALUE(cat, val) ((cat ¡¡ OGRE_CAPS_BITSHIFT) — (1 ¡¡ val))"

```
(defun enum-caps-value (string)
  (cl-ppcre:register-groups-bind (cat val)
      ("OGRE_CAPS_VALUE\\((\\S+), (\\d+)\\)" string)
    (logior (ash (gethash cat *enums*) 28)
            (ash 1 (parse-number val)))))
```

This function converts a C++ enum value to one we can use in Okra. It does this by setting several regular expressions lose on the input string to detect what kind of type it is (decimal, hexadecimal or constant) and executing different actions for different types.

```
(defun enum-value (string)
  (cond ((cl-ppcre:scan *regex-dec* string)
         (parse-number (first-match *regex-dec* string)))
        ((cl-ppcre:scan *regex-hex* string)
         (parse-number (mkstr "#x" (first-match *regex-hex* string))))
        ((cl-ppcre:scan *regex-con* string)
         (let ((constant (first-match *regex-con* string)))
           (cond ((gethash constant *enums*) (gethash constant *enums*))
                 ((equal (subseq constant 0 16) "OGRE_CAPS_VALUE(")
                  (enum-caps-value constant))
                 ;; hardcoding these enum values for now, since Doxygen
                 ;; doesn't put them in the xml
                 ((equal constant "HardwareBuffer::HBU_STATIC") 1)
                 ((equal constant "HardwareBuffer::HBU_DYNAMIC") 2)
                 ((equal constant "HardwareBuffer::HBU_WRITE_ONLY") 4)
                 ((equal constant "HardwareBuffer::HBU_STATIC_WRITE_ONLY") 5)
                 ((equal constant "HardwareBuffer::HBU_DYNAMIC_WRITE_ONLY") 6)
                 ((equal constant "HardwareBuffer::HBU_DYNAMIC_WRITE_ONLY_DISCARDABLE") 14
                 ((equal constant "TU_AUTOMIPMAP | TU_STATIC_WRITE_ONLY") 3)
                 (t (error "Could not parse enum constant: ~S" constant)))))
        (t (error "Could not parse enum: ~S" string))))
```

This function looks through FILE for memberdef-elements with attributes kind="enum" and prot="public" and puts their parsed representation in *ENUM-TYPES* and its enum values in *ENUMS*. (I think, I'm writing this a good time after having written this function ;-))

```
(defun parse-doxygen-enums-file (file)
  (iter (for enum in (node '(:|compounddef| :|sectiondef| :|memberdef|)
                           (parse-xml-file file)))
        (unless (and (equal (attribute :|kind| enum) "enum")
                     (equal (attribute :|prot| enum) "public"))
          (next-iteration))
        (as type-name = (value-of (first (node :|name| enum))))
        (iter (with previous-value = 0)
              (for value in (node :|enumvalue| enum))
              (as initializer = (value-of (first (node :|initializer| value))))
              (as name = (value-of (first (node :|name| value))))
              (push name (gethash type-name *enum-types*))
              (if (equal initializer "")
                  (progn (setf (gethash name *enums*) previous-value)
                         (incf previous-value))
                  (setf (gethash name *enums*) (enum-value initializer))))))
```

PARSE-DOXYGEN-TYPEDEFS-FILE looks through FILE for memberdef-elements with attribute kind="typedef", filters most of the results out and puts the rest in *TYPEDEFS*.

```
(defun parse-doxygen-typedefs-file (file)
  (iter (for typedef in (node '(:|compounddef| :|sectiondef| :|memberdef|)
                           (parse-xml-file file)))
        (unless (equal (attribute :|kind| typedef) "typedef")
          (next-iteration))
        (as name = (value-of (first (node :|name| typedef))))
        (as type = (value-of (first (node :|type| typedef))))
        ;; only very basic types are handled
        (when (or (and (not (equal type "UTFString"))
                       (cl-ppcre:scan *regex-caps* type))
                  (cl-ppcre:scan *regex-crap* type)
                  (cl-ppcre:scan *regex-stl* type)
                  (string-equal "real" name))
          (next-iteration))
        (setf (gethash name *typedefs*) type)))
```

PARSE-DOXYGEN-FILES is pretty big but also rather basic. It first clears all the hashes so we won't have old values from a previous run in them. After calling the functions for parsing the enums and typedefs it handle each file in FILES seperately.

For each sectiondef-element with attribute kind="public-func" it will iterate through each of its child memberdef-elements and filter out any contructors, deconstructors or names starting with an underscore. Then it'll get the memberdef's type and arguments and check if it blacklisted (see *BLACKLIST*). If not it'll get added to *MEMBERS*.

Lastly the member will get added to the class it belongs to in *CLASSES*.

```
(defun parse-doxygen-files (&optional (files *doxygen-files*))
  (clrhash *typedefs*)
  (format t "Parsing Doxygen files for typedefs...~%")
  (iter (for file in *doxygen-typedefs-files*)
        (parse-doxygen-typedefs-file file))
  (clrhash *enum-types*)
  (clrhash *enums*)
  (format t "Parsing Doxygen files for enums...~%")
  (iter (for file in *doxygen-enums-files*)
        (parse-doxygen-enums-file file))
  (clrhash *base-classes*)
  (clrhash *classes*)
  (clrhash *members*)
  (iter (for file in files)
        (as xml = (parse-xml-file file))
        (as base-class-name = (ogre-base-class-name xml))
        (as class-name = (ogre-class-name xml))
        (format t "Parsing Doxygen file for ~A class...~%" class-name)
        (setf (gethash class-name *base-classes*) base-class-name)
        (iter (for node in (node '(:|compounddef| :|sectiondef|) xml))
              ;; we're only interested in public methods
              ;(unless (or (equal (attribute :|kind| node) "public-static-func")
              ;            (equal (attribute :|kind| node) "public-func"))
              (unless (equal (attribute :|kind| node) "public-func")
                (next-iteration))
              (iter (for memberdef in (node :|memberdef| node))
                    (as name = (memberdef-name memberdef))
                    (when (or ;; skip constructors
                              (equal name class-name)
                              ;; skip deconstructors
                              (equal (elt name 0) #\~)
                              ;; skip member names starting with an underscore
                              (equal (elt name 0) #\_))
                      (next-iteration))
                    (as type = (memberdef-type memberdef))
                    (as args = (memberdef-args memberdef))
                    (as member = (list :name name :type type :args args))
                    (unless (blacklisted member)
                      ;; ":args" could be removed but it looks good in debug
                      (pushnew (list :args (if (listp args)
                                               (loop for arg in args
                                                  collect (if (consp arg)
                                                              (car arg)
                                                              arg))
                                              args))
                                (gethash name *members*)
                                :test #'equal))
                     (push member (gethash class-name *classes*))))))
```

2 Main Program

GENERATE-BINDINGS is the top-level functions called to generate all the Okra bindings. The name's pretty generic since our scope is just C++ and Common Lisp. If we'll ever add support for other languages we'll rename this function.

Sets *VERBOSE* by default to NIL which may be used by any other function to decide whether to print debug / verbose output to *STANDARD-OUTPUT*.

```
(defun generate-bindings (&optional (verbose nil))
  (setf *verbose* verbose)
  (initialise-templates)
  (parse-doxygen-files)
  (generate-cpp-bindings)
  (generate-lisp-bindings))
```

This is here for development so I can just do an "(asdf :okra-bindings-generator)" from Slime and be done with it.

```
;; for development
;(generate-bindings t)
(generate-bindings)
```