Implementing the RGraphicsDevice package

Table of Contents

Motivation

The idea is to try to automate much of this as there is very little going on. Essentially, for a graphics device, we need to provide C routines that implement each of the function pointer fields. Each must accept its arguments, convert them to R objects, call the corresponding R function, converting the result to the corresponding C type and returning that to the R graphics engine. This is a job for the RGCCTranslationUnit package. In addition to implementing these simple C routines, we also want to provide R programmers with access to the fields of the data structures.

Exploring the data structures

We first start by generating a translation unit description for the R graphics device structure. We do this by creating a C file that includes Rinternals, R_ext/GraphicsEngine.h and R_ext/GraphicsDevice. Then we create the TU file from that by compiling that file with gcc and the flag -fdump-translation-unit. This is done in the TU directory of this package.

Given the TU file, let's read it into R and look at the data structures.

```
library(RGCCTranslationUnit)
tu = parseTU("TU/gd.c.t00.tu")
ds = getDataStructures(tu)
```

We know which structure(s) we want, namely the *DevDesc*. The following returns a *ResolvedTypeReference* so we resolve it again to get the *StructDefinition*:

```
DevDesc = resolveType(ds$DevDesc, tu)
DevDesc = resolveType(DevDesc, tu)
```

This contains a description of the fields, amongst other details.

Now, let's look at the function pointers which are like "methods" for an instance of a device.

```
i = sapply(DevDesc@fields, function(x) class(x@type)) == "FunctionPointer"
funs = DevDesc@fields[i]

 # ensure that the parameters, etc. are fully resolved
funs = lapply(funs, function(x) {
                      x@type = resolveType(x@type, tu)
                      x
```

```
                    })
```

The *lapply*() call ensures that the parameters are all fully resolved.

Let's look at the types of the parameters in these function pointers:

```
table(sapply(unlist(sapply(funs, function(x)
                                  lapply(x@type@parameters, slot, "type")),
                    recursive = FALSE),
      class))

          CString        doubleType           intType       PointerType
                5                23                 4                13
TypedefDefinition
               32
```

So we see the basic types (integers, doubles, character strings) and a typedef and pointers. We'll dig down further into these generic pointers and typedefs:

```
params = unlist(sapply(funs, function(x) lapply(x@type@parameters, slot, "type")),
f = function(x)  {
           if(is(x, "PendingType") || is(x, "ResolvedTypeReference"))
              x = resolveType(x, tu)
           if(is(x, "PointerType") || is(x, "TypedefDefinition"))
              f(x@type)
           else
              x
         }
tp = sapply(params, f)
table(sapply(tp, function(x)
                    if(is(x, "StructDefinition"))
                      x@name
                    else if(is(x, "PendingType"))
                      x@name
                    else class(x)))

     charType    doubleType       intType      pDevDesc R_GE_gcontext
            5            36             4            20            11
      SEXPREC
            1
```

So we see a pointer to a DevDesc, R_GE_gcontext and the regular R object type in C (SEXP or SEXPREC).

Let's also look at the non-function pointer fields:

```
names(DevDesc@fields[!i])

 [5] "clipLeft"                "clipRight"
 [7] "clipBottom"              "clipTop"
 [9] "xCharOffset"             "yCharOffset"
[11] "yLineBias"               "ipr"
[13] "cra"                     "gamma"
```

```
[15] "canClip"                  "canChangeGamma"
[17] "canHAdj"                  "startps"
[19] "startcol"                 "startfill"
[21] "startlty"                 "startfont"
[23] "startgamma"               "deviceSpecific"
[25] "displayListOn"            "canGenMouseDown"
[27] "canGenMouseMove"          "canGenMouseUp"
[29] "canGenKeybd"              "gettingEvent"
[31] "hasTextUTF8"              "wantSymbolUTF8"
[33] "useRotatedTextInContour" "reserved"

table(sapply(DevDesc@fields[!i], function(x)  class(x@type)))

ArrayType            doubleType EnumerationDefinition
        3                    14                    11
   intType           PointerType
        5                     1
```

These maintain the state of the graphics device. We will want access to these values, to both query and set them.

The Desired Code

Ideally, we would simply call generateStructInterface. This will handle generating much of the interface, but there are different considerations we want to explore. Firstly, for each of our function pointers, we will have a C routine that provides a proxy. Each of these will find the corresponding R function for the device. If this is **`NULL`**, the C routine does nothing. If there is an R function, it invokes that.

How do we find the corresponding R function for a C routine? We can either have a list of R functions or, alternatively, a C-level structure with a SEXP for each individual function. The benefit of having a separate structure is slightly more direct access relative to calling

```
VECTOR_ELT(funs, i)
```

and having to use numbers/enums to index into that list, e.g. CIRCLE. So we would define a structure such as

```
typedef struct {
  SEXP text;
  SEXP strWidth;
  SEXP size;
    ..
} RDevDescMethods;
```

We could even use an array of SEXP objects rather than a struct as the offsets are fixed.

We would then create an instance of this and store it in the `deviceSpecific` field of the *pDevDesc* object. We will populate it with R functions. We have an R function that has a parameter for each of the methods and creates an instance of this *RDevDescMethods* structure and fills in each value that is provided by the caller. We should test the number of parameters each function accepts to ensure they are compatible.

So the top-level R function to create an R-level graphics device would look something like

```
createGraphicsDevice =
```

```
function(text = NULL, line = NULL, circle = NULL, ...)
{
  devMethods = .Call("R_create_RDevDesc", text, line, circle, ....)
  dev = .Call("R_create_DevDesc")
  dev$deviceSpecific = .Call("R_createGraphicsDevice", devMethods)
  dev$canClip = FALSE
  dev$canHAdj = TRUE
    ...

  dev
}
```

There may be an issue with registering the device before its fields are initialized, so we may need to separate creation and registration in R_createGraphicsDevice.

The C routines that we set for the DevDesc function pointers will be something along the lines

```
void
R_circle(double x, double y, double r, const pGEcontext gc, pDevDesc dd)
{
  RDevDescMethods *methods = (RDevDescMethods *) dd->deviceSpecific;
  SEXP e, cur;

  if(!methods->circle || methods->circle == R_NilValue)
    return;

  PROTECT(cur = e = allocVector(LANGSXP, 6));
  SETCAR(cur, methods->circle); cur = CDR(cur);
  SETCAR(cur, ScalarReal(x)); cur = CDR(cur);
  SETCAR(cur, ScalarReal(y)); cur = CDR(cur);
  SETCAR(cur, ScalarReal(r)); cur = CDR(cur);
  SETCAR(cur, mkRef(gc, "GEcontext")); cur = CDR(cur);
  SETCAR(cur, mkRef(dd, "DevDesc"));

  Rf_eval(e, R_GlobalEnv);
}
```

Note that we could create each of these calls just once and store those in the RDevDescMethod structure. This is just a minor modification to how we generate the code and would make the code more efficient.

So we need a class definition for a reference to a DevDesc and also for a reference to a GEcontext.

Generating the Code

The first thing we can do is to create code for the *RDevDescMethods* type that we introduced earlier. This does not exist in the C code for the graphics device and is a construct for our interface. We can write the definition and the accessor code for the fields. However, it is more convenient to programmatically generate this by "pretending" it was in the C code definitions we read. We can programmatically construct a description of such a C-level structure. This is a structure with a SEXP for each of the function pointers in the *DevDesc* structure. So one thing we can do is to copy the **`funs`** list describing the function pointers and change the type of each to be a SEXP.

```
sexp = resolveType(ds$SEXP, tu)
fields = lapply(funs, function(x) {
                           x@type = sexp
                           x
                         })
```

Then we can create a StructDefinition object with these fields and put that within a TypedefDefinition to be able to refer to this via the name *RDevDescMethods*:

```
devDescMethods = new("TypedefDefinition",
                     name = "RDevDescMethods",
                     type = new("StructDefinition",
                                name = "RDevDescMethods",
                                fields = fields))
```

We can now use all the code generation facilities in RGCCTranslationUnit to create an interface to this data type.

The next task is to create the C routines that will act as proxies and pass the C-level arguments to the corresponding R functions (see ??? (dd) (page 4**)**. We'll do this for just one (circle) and look at the result. We call *`createProxyRCall`*() and give it the FunctionPointer that will be called. We also give it the name of the C routine to define. And the third argument is code C code to access the R function corresponding to this function.

```
createProxyRCall(funs[["circle"]]@type, "R_circle",
                   "((RDevDescMethods*) (r5->deviceSpecific))->circle")
```

So we do this for all of the function pointers with a "simple" loop. Unfortunately, the function pointer getEvent does not have a parameter providing a DevDesc reference. This means we cannot readily access the device specific data and hence the corresponding R function. So for the moment, we'll just exclude that.

```
funs = funs[ - match("getEvent", names(funs)) ]
```

Now we can generate the code for all the function pointers.

```
proxyNames = structure(paste("R", names(funs), sep = "_"),
                       names = names(funs))
code = mapply(createProxyRCall,
              lapply(funs, slot, "type"),
              proxyNames,
              paste("((RDevDescMethods*) ( r",
                    sapply(funs, function(x) length(x@type@parameters)),
                    "->deviceSpecific))->", names(funs), sep = ""))
```

Note that we have to know how many parameters there are for each routine to be able to correctly create the C code to access the *pDevDesc* parameter. It would be simpler if the compiler emitted the parameter names for a routine and we could refer to this as `dd` as it is in the header files. Unfortunately, this only happens for routine definitions, not declarations.

The next thing to create is the code that initializes the DevDesc with these routines. We have the names in **`proxyNames`**, so we can generate this code easily as (assuming a variable named `dd` representing the *pDevDesc*)

```
init = c("dd->deviceSpecific = calloc(1, sizeof(RDevDescMethods));",
         "if(!dd->deviceSpecific) return(0);",
         paste("dd ->", names(proxyNames), "=", proxyNames, ";"),
```

```
         "return(1);")
```

We can put this in code we write or specify it as the body of a new C routine which we can call:

```
init = CRoutineDefinition("initializeDevDescMethods",
                          c("int", "initializeDevDescMethods(pDevDesc dd)", "{", i
```

We'll manually write the code to create the graphics device and call this initialization routine as this is specialized code that depends on knowing R internals. See *R_createGraphicsDevice* in Rgd.c.

(Need writeCode method for devDescMethods. Done now?)

```
con = file("src/proxy.c", "w")
writeIncludes(c("<Rdefines.h>", "<R_ext/GraphicsEngine.h>", "<R_ext/GraphicsDevice

writeCode(devDescMethods@type, "C", con)
writeCode(devDescMethods, "C", con)

idevDescMethods = generateStructInterface(devDescMethods, DefinitionContainer(tu))
writeCode(idevDescMethods, "C", con)
writeCode(idevDescMethods, "R", "R/devDescMethods.R")

cat(paste(sapply(code, as, "character"), collapse = "\n\n"), file = con)
writeCode(init, "C", file = con)
close(con)
```

Next we need an R function and C routine that sets the fields of the *RDevDescMethods*. This is used to initialize the fields of the RDevDescMethods in the `deviceSpecific` field of the *DevDesc* structure. We can have a single function that accepts values for all the fields and sets them in a single operation, or we can have individual routines to set each field. It is easy to create the latter using *`generateStructInterface`*() (at least it is intended to be!) and this generates the corresponding class definition. C routine to instantiate a C version, provide methods for getting and setting individual elements via the $ operatoretc, ... We do this with the *`generateStructInterface`*() call in the code above.

After this, we need to be able to access and set the fields in the DevDesc. This is a run-time facility so that the R programmer can query and modify a particular graphics device. (See TU/tu.R for details.)

```
devDesc.sub = DevDesc
devDesc.sub@fields = devDesc.sub@fields[!i]
idevDesc = generateStructInterface(devDesc.sub, DefinitionContainer(tu))
```