OpenCascade Tcl vs. ACIS Scheme

eryar@163.com

摘要 Abstract：本文通过 OpenCascade 的 Tcl/Tk 和 ACIS 的 Scheme 的对比来说明脚本语言在程序中的重要作用。及通过在 Tcl 中实现自定义的命令来理解 Draw Test Harness 的实现，在此基础上更有利于对 OpenCascade 的理解，其中 Draw Test Harness 一些命令的实现可以做为程序实现的参考。

关键字 Key Words：OpenCascade, Tcl/Tk, ACIS, Scheme, Test

一、引言 Introduction

解释型的语言为程序开发提供了快速原型方法，由于是解释型的程序开发语言，所以用它编写的程序不需要编译和链接就可以在解释器中直接运行，用这种方式可以快速验证一些想法，用来测试程序等。

解释型的脚本开发语言有很多，流行的有 Python、Tcl/Tk、Perl、Lua、LISP、Javascript 等等。在应用程序中嵌入这些脚本语言，方便程序的扩展，用户可以根据需要对程序进行定制。像应力分析软件 Abaqus 中就使用了 Python 对其进行二次开发；绘图软件 AutoCAD 中可使用 AutoLISP 对其进行二次开发；造船软件 KCS Tribon 中也使用了 Python；AVEVA Marine/Plant 中使用了 PML；许多游戏程序中使用了 Lua 等等。程序有了脚本这个二次开发的强有力工具，软件开发者只需要关注核心领域，其它各种客户化的需求，可以通过这些脚本来实现，既能满足用户花样繁多的需求，也可确保程序质量。

ACIS 对 Scheme 的扩展为 Scheme 解释器提供了调用 ACIS 内部函数和访问 ACIS 内部数据的功能，其中，每个 Scheme 函数都调用相应的 ACIS C++函数。开发者可以利用 Scheme 来熟悉 ACIS 的基本功能、测试某些思想或者产生应用程序原型。Scheme 语言的初学者可以通过它来熟悉和掌握 ACIS 的 Scheme 程序开发方法。

OpenCascade 中使用 Tcl/Tk 来实现的 Draw Test Harness 提供了交互创建、显示和修改形状的功能。可以编写 Tcl 的脚本来自定义 Draw 或者实现程序的自动化测试。

本文通过 OpenCascade 的 Tcl/Tk 和 ACIS 的 Scheme 的对比来说明脚本在程序中的重要作用。及通过在 Tcl 中实现自定义的命令来理解 Draw Test Harness 的实现，在此基础上更有利于对 OpenCascade 的理解，其中一些命令的实现代码可以做为程序实现的参考。

二、ACIS Scheme

ACIS 中的 Scheme 接口是一些函数的集合，基于 Scheme 的应用程序可以通过这些函数调用 ACIS 的 API、类的公共成员函数及访问 ACIS 的数据。ACIS 系统中含有一个 Scheme 解释器，Scheme 应用程序就在这个解释器上运行，所以该解释器负责 Scheme 应用程序与 ACIS 系统之间的通信。ACIS 中的 Scheme 解释器是按 Elk 版的 Scheme 语言规则设计的，它被用来解释 Scheme 命令并调用相关的 C++代码。

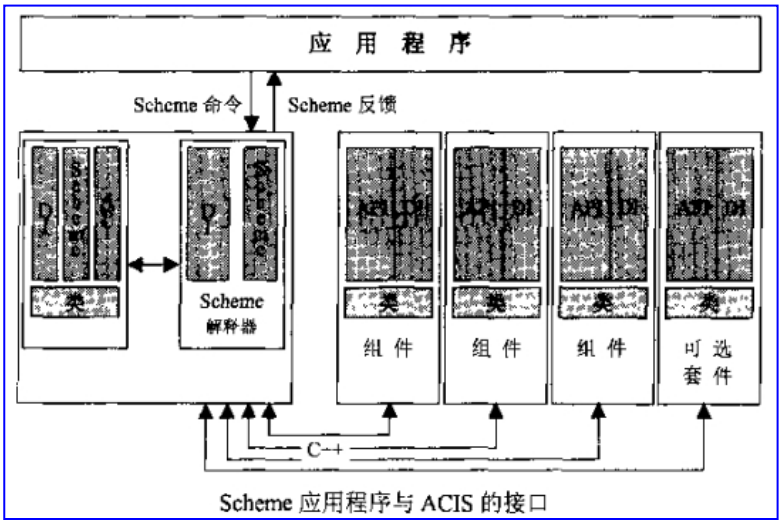

Figure 2.1 Scheme and ACIS

Scheme 的多功能性来自解释器语言的可扩展能力。应用程序开发者可以快速地将 Scheme 语言用于自己的开发任务，他们可以用 C++扩展 Scheme，然后就可以利用这些扩展写可解释执行的 Scheme 过程。开发者也可以构造特殊的 Scheme 数据类型和相应的操作方法，Scheme 应用程序和 C++函数可以使用这些自定义的 Scheme 数据类型。

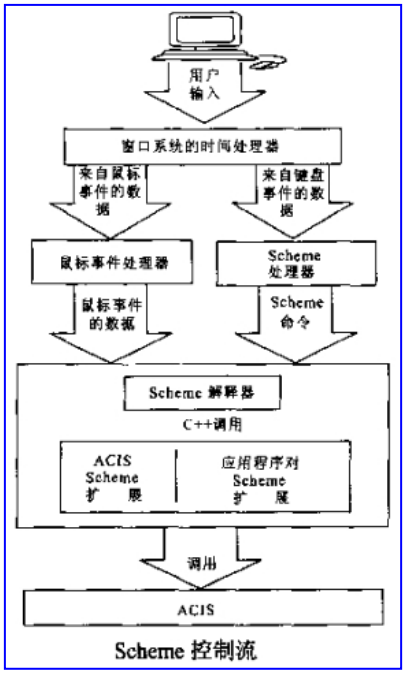

Figure 2.2 ACIS Scheme pipeline

与 C++相比 Scheme 是一种快速程序设计语言。尽管 C++语言有很多优点，但是也不得承认它的复杂性，例如要用 C++开发 ACIS 应用程序，需要包含所需的组件头文件并在程序中初始化这些组件，除此之外还要进行复杂的程序调试，所以使用 C++开发应用程序的速度

就受到较大影响。与其相反，Scheme 可以进行交互式程序开发，一段程序或一个命令的执行结果会及时反映出来，正是这种机制使我们可以利用 Scheme 语言进行快速的程序开发。虽然 Scheme 不能替代 C++成为主要的软件开发工具，但是在开发一个软件系统之前可以利用它进行算法的测试。

Scheme 语言与 C++语言相比有三个特点：没有指针、不需要头文件及可进行交互式程序设计，程序员可以在解释器中立即执行这些程序，从这个特点来看 Scheme 更像 Basic 和 Prolog 等程序设计语言。

在 ACIS 中使用 Scheme 来生成一个立方体的命令如下所示：

```
acis>(solid:block(position -10, -10, -10) (position 10, 10, 10))
# [entity 1 0]
```

上述命令生成一个正方体，总共调用两个 ACIS Scheme 过程 position 和 solid:block，#[entity 1 0]是该正方体的默认名称。

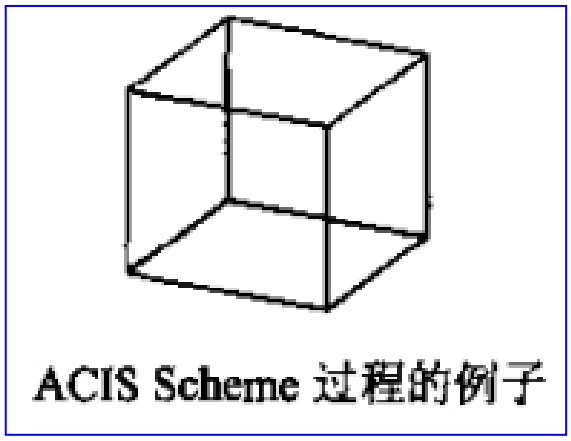

Figure 2.3 The block made by ACIS Scheme

三、OpenCascade Tcl

Tcl 是一种用于控制和扩展应用程序的动态语言，也称为脚本语言。它的名字代表“工具命令语言”Tool Command Language。Tcl 提供的通用编程能力可以满足大多数应用程序的需要。而且 Tcl 既是可嵌入的（embeded），也是可扩展的（extensible）。它的解释器是一个 C 函数库，可以很容易地整合到应用程序中；而任何一个应用程序都可以通过增加命令来扩展 Tcl 内核的功能。Tcl 最有用的一个扩展就是 Tk，这是一个用于开发图形用户界面（Graphical User Interface, GUI）应用程序的工具集。Tk 扩展了 Tcl 内核的功能，增加了构建用户界面的命令，使您可以使用 Tcl 脚本来构建图形用户界面，而不必写 C 代码。

Tcl/Tk 一起为应用程序开发者和使用者提供了很多好处。首先是快速开发。很多有意思的程序完全可以用 Tcl 脚本编写。这使您可以在比 C/C++或 Java 更高的层次上进行开发，Tk 隐藏了 C 或 Java 程序员必须关心的很多细节。与低级工具相比，使用 Tcl/Tk 所需要学习的知识更少，需要编写的代码更少。通过几个小时的学习，Tcl/Tk 新手用户就可以创建有意思的用户界面，很多开发人员从其他工具集转而使用 Tcl/Tk 工具集后，应用程序开发所需的代码数量和开发时间都减少了 90%。

Tcl/Tk 适于快速开发的另一个原因在于 Tcl 是解释型语言。使用 Tcl 应用程序时，可以在运行中生成和使用新的脚本，而无需重新编译和重启程序。这使您可以迅速尝试新的想法，迅速修正程序中的错误。因为 Tcl 是解释型语言，它的运行速度比 C 程序慢。但是通过内部优化，与编译语言相比的大部分性能差距都可以消除。例如，您可以运行有数百条 Tcl 命令的脚本，鼠标的每一次移动都不会有能感知的延迟。在一些特别的场合，当性能成为重要问题时，可以把 Tcl 脚本中影响性能的部分替换为 C 代码。

Tcl 的第二个好处是在于它是跨平台的语言，它的大多数扩展包括 Tk 也是如此。这意味着在一个平台（如 Linux）上开发的程序，在大多数情况下可以不加改动地在另一个平台上运行，如在 Macintosh 或 Windows 上运行。

Tcl 还是第一种拥有原生 Unicode 支持的动态语言。因此，Tcl 可以处理这个世界上几乎所有的书面语言。Tcl 无需扩展就可以处理 Unicode 支持的所有文本。

使用 Tcl 的另一个显著优点在于它和它的大多数扩展都是免费的开源软件。Tcl 和 Tk 遵循 BSD 授权，允许所有人免费下载、查看、修改及再发布。

Tcl 是一种绝妙的“胶合语言”，可以让应用程序很容易地拥有强大的脚本语言功能。例如，要为一个已经存在的应用程序添加脚本能力，只需要实现几条新的 Tcl 命令，用来为应用程序提供相应的基本功能。然后再把您的新命令和 Tcl 库链接起来生成全功能的脚本语言，该语言就包含了 Tcl 提供的命令（称为 Tcl 内核）和您编写的那些命令。

Tcl 还为用户提供了方便。一旦学习了 Tcl/Tk，就能为任何 Tcl/Tk 应用程序编写脚本，只需要学习该应用程序特有的少数几条命令即可。这使得更多的用户有能力对应用程序进行个性化改造和强化。

在 Tcl 中实现自定义命令很方便，只需要按 Tcl 的格式定义一个命令函数。基于对象的命令函数的声明如下：

```
typedef int (Tcl_ObjCmdProc) _ANSI_ARGS_((ClientData clientData,
    Tcl_Interp *interp, int objc, struct Tcl_Obj * CONST * objv));
```

为了能在 Tcl 中调用一个命令函数，必须先调用 Tcl_CreateObjCommand 注册它，格式如下所示：

```
EXTERN Tcl_Command      Tcl_CreateObjCommand (Tcl_Interp * interp,
                        CONST char * cmdName, Tcl_ObjCmdProc * proc,
                        ClientData clientData,
                        Tcl_CmdDeleteProc * deleteProc);
```

这就是把 Tcl 中的字符串与实现它的 C 函数关联起来“魔术”。更详细内容请参考：《**Create New Commands in Tcl**》。下面主要来分析一下 Draw Test Harness 代码实现：

首先在 Draw_PInterp.hxx 中将 Tcl_Interp 定义为*Draw_PInterp，并在类 Draw_Interpretor 中对 Tcl 解释器进行了简单封装，在 Init 时创建一个新的 Tcl_Interp，并可管理自定义的命令；

然后，在 Draw_Main 中来加载 Tcl 的解析器。程序代码如下所示：

```
// Declarations of macros DRAW_MAIN to be used in executables instead of explicit
main/WinMain
#ifndef WNT
// main()
#define DRAW_MAIN int main (Standard_Integer argc, char* argv[])\
{return _main_ (argc, argv, Draw_InitAppli);}
#else
// WinMain() and main()
#define DRAW_MAIN Standard_Integer PASCAL WinMain (HINSTANCE hInstance, HINSTANCE
hPrevinstance, LPSTR lpCmdLine, Standard_Integer nCmdShow)\
{return _WinMain_ (hInstance, hPrevinstance, lpCmdLine, nCmdShow,
Draw_InitAppli);}\
\
int main (int argc, char* argv[], char* envp[])\
{return _main_ (argc, argv, envp, Draw_InitAppli);}
#endif
```

自定义了 main 函数，实现如下所示：

```
//=======================================================================
//function : _main_
//purpose  :
//=======================================================================
Standard_Integer _main_ (int argc, char* argv[], char* envp[], const
FDraw_InitAppli fDraw_InitAppli)
{
  Draw_IsConsoleSubsystem = Standard_True;
  //return _WinMain_(::GetModuleHandle(NULL), NULL, GetCommandLine(), SW_SHOW,
fDraw_InitAppli);
  theDraw_InitAppli = fDraw_InitAppli;
  //ParseCommandLine(GetCommandLine());

  // MKV 01.02.05
#if ((TCL_MAJOR_VERSION > 8) || ((TCL_MAJOR_VERSION == 8) && (TCL_MINOR_VERSION >=
4)))
  Tcl_FindExecutable(argv[0]);
#endif

  Draw_Appli(::GetModuleHandle(NULL), NULL, GetCommandLine(), SW_SHOW,
fDraw_InitAppli);
  return 0;

}
```

Draw Test Harness 中自定义的命令是通过类 Draw_Commands 来添加的，程序如下所示：

```
void Draw::Commands(Draw_Interpretor& theCommands)
{
  Draw::BasicCommands(theCommands);
  Draw::VariableCommands(theCommands);
  Draw::GraphicCommands(theCommands);
```

```
  Draw::PloadCommands(theCommands);
  Draw::UnitCommands(theCommands);
}
```

分别对应的定义文件为：

- Draw_BasicCommands.cxx：基本命令；
- Draw_VariableCommands.cxx：变量命令；
- Draw_GraphicCommands.cxx：图形命令；
- Draw_PloadCommands.cxx：加载命令；
- Draw_UnitCommands.cxx：单位命令；

如可以打开 Draw_BasicCommands 可以看到 OpenCascade 定义了哪些基本命令，我发现一个查看 OpenCascade 编译时的配置信息的命令 dversion：

```
static Standard_Integer dversion(Draw_Interpretor& di, Standard_Integer, const char**)
{
  // print OCCT version and OCCTY-specific macros used
  di << "Open CASCADE Technology " << OCC_VERSION_STRING_EXT << "\n";
#if defined(DEB) || defined(_DEBUG)
  di << "Debug mode\n";
#endif
#ifdef HAVE_TBB
  di << "TBB enabled (HAVE_TBB)\n";
#else
  di << "TBB disabled\n";
#endif
#ifdef HAVE_GL2PS
  di << "GL2PS enabled (HAVE_GL2PS)\n";
#else
  di << "GL2PS disabled\n";
#endif
#ifdef HAVE_FREEIMAGE
  di << "FreeImage enabled (HAVE_FREEIMAGE)\n";
#else
  di << "FreeImage disabled\n";
#endif
#ifdef No_Exception
  di << "Exceptions disabled (No_Exception)\n";
#else
  di << "Exceptions enabled\n";
#endif
```

运行结果如下图所示：

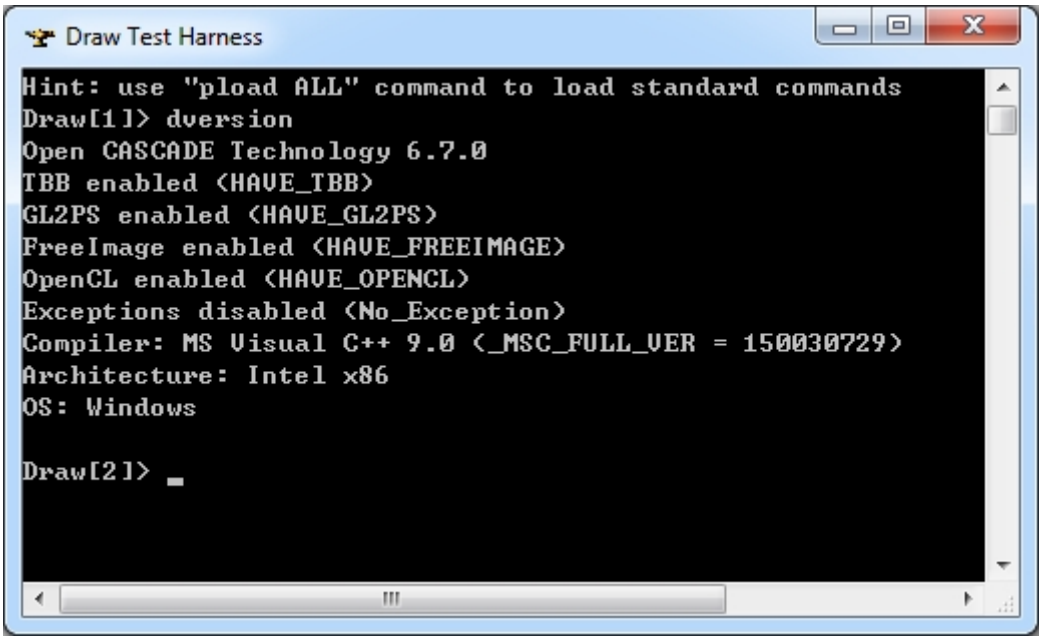

Figure 3.1 dversion command in Draw Test Harness

由上图可知编译 OpenCascade 时的相关配置信息，如版本号、使用了哪些第三方库、编译器类型及版本、系统类型 x86/AMD64、操作系统等等。

```
  // OS
#if defined(_WIN32) || defined(__WINDOWS__) || defined(__WIN32__)
  di << "OS: Windows\n";
#elif defined(__APPLE__) || defined(__MACH__)
  di << "OS: Mac OS X\n";
#elif defined(__sun)
  di << "OS: SUN Solaris\n";
#elif defined(__ANDROID__) /* must be before Linux */
  #include <android/api-level.h>
  di << "OS: Android (__ANDROID_API__ = " << __ANDROID_API__ << ")\n";
#elif defined(__linux__)
  di << "OS: Linux\n";
#elif defined(__FreeBSD__) || defined(__OpenBSD__) || defined(__NetBSD__) || defined(__DragonFly__)
  #include <sys/param.h>
  di << "OS: BSD (BSD = " << BSD << ")\n";
#else
  di << "OS: unrecognized\n";
#endif
```

Figure 3.2 The OS OpenCascade supported

由上图可知 OpenCascade 可识别的操作系统涵盖发当今主流的操作系统，如 Windows、Mac OS、Android、Linux 等。

最后以一个实现光线追踪的脚本来演示一下使用脚本的灵活性，Tcl 脚本如下所示：

```
# Script reproducing creation of bottle model as described in OCCT Tutorial
pload MODELING VISUALIZATION
puts "Writing \"eryar@163.com\"..."
box body 0 0 0 130 20 8
# define text Courier Consolas
text2brep text2d eryar@163.com Times-Roman 18 bold composite=0
#text2brep text2d eryar@163.com Consolas 18 bold composite=0
#text2brep text2d eryar@163.com Courier 18 bold composite=0
prism text text2d 0 0 2
ttranslate text 4 5 6

# cut operation
bcut bodytext body text
ttranslate text 0 20 0

puts "Showing result..."
# display result
vdisplay bodytext
vdisplay text
vzfit
vfit
vsetdispmode 1
# set ray tracing
if { [regexp {HAVE_OPENCL} [dversion]] } {
    puts "Trying raytrace mode..."
    if { ! [catch {vraytrace 1}] } {
        vtextureenv on 1
        vfit
    }
}
```

脚本实现了在一个 box 上 cut 掉文本后的光线跟踪效果，如下图所示：

Figure 3.3 Ray Tracing Rendering by Tcl

理解了 Tcl 的作用之后，我认为 OpenCascade 提供的最快速的的开发方式应该就是全部使用 Tcl/Tk 来开发程序。利用 OpenCascade 已经定义好的命令，包括操作 ApplicationFramework 的自定义命令。在此基础上，使用 Tk 来实现 GUI 及用户交互操作，这样开发程序事办功倍，而且还可提供用户二次开发功能，即使用 Tcl/Tk 来对程序进行二次开发，扩展能力强。

四、结论 Conclusion

造型程序中引入脚本语言的共同优点有：脚本语言是解释执行，不需要编译链接，可以即时验证一些想法。还可使用脚本来执行自动化测试，保证算法质量。便于扩展，提供用户二次开发的工具。

与 ACIS 的 Scheme 相比，OpenCascade 的 Tcl/Tk 有着明显的优势。其中最明显的就是源代码开放，可以通过查看源程序，来完全理解脚本的实现。而这在 ACIS 中是不可能的。

理解如何在 Tcl 实现自定义的命令后，可以查看 OpenCascade 在 Draw Test Harness 中相关程序的实现方法，对学习一些造型算法的使用还是很有帮助的。

五、后记 Postscript

在曹金凤《Python 语言在 Abaqus 中的应用》的序言一中看到这样一段话：“我还想顺便谈一个体会：同 50 年前有限元方法出现的时代相比，现在有了大量有效的计算力学软件。既然已经有了大家公认的很优秀的分析平台，研究者不应再奋力去开发具有竞争力的新软件了，而应当把精力花在基于这些平台进行二次开发上面了。从科学技术共同体的角度去思考，这应当是如今计算力学软件研发的最佳策略。”——隋允康于北京工业大学。

对于这段话的大部分内容是赞同的，即站在巨人的肩上才能看得更远。但纵观当今力学分析的 CAE 软件优秀的大都是国外产品，如 ANSYS、Abaqus、COADE.CAESAR、MSC/PATRAN、ADINA 等等，而国内同类产品鲜有耳闻。原因可能是 CAE 软件的开发涉及的知识很多，需要计算机图形学、软件工程、力学、数学等等多学科知识的结合，所以开发具有自主知识产权的 CAE 软件也是国家科技实力的一个表现。若只是在外国人的平台上二次开发，总是会受制于人。也许有些狭隘，如今世界已然是个地球村，科学无国界。

六、参考资料 References

1. Tcl and the Tk Toolkit
2. Practical Programming in Tcl and Tk
3. Tcl/Tk A Developer’s Guide
4. http://sourceforge.net/projects/tcl/
5. http://www.tcl.tk/
6. 詹海生等, 基于 ACIS 的几何造型技术与系统开发, 清华大学出版社, 2002
7. 曹金凤, 王旭春, 孔亮. Python 语言在 Abaqus 中的应用. 机械工业出版社, 2011