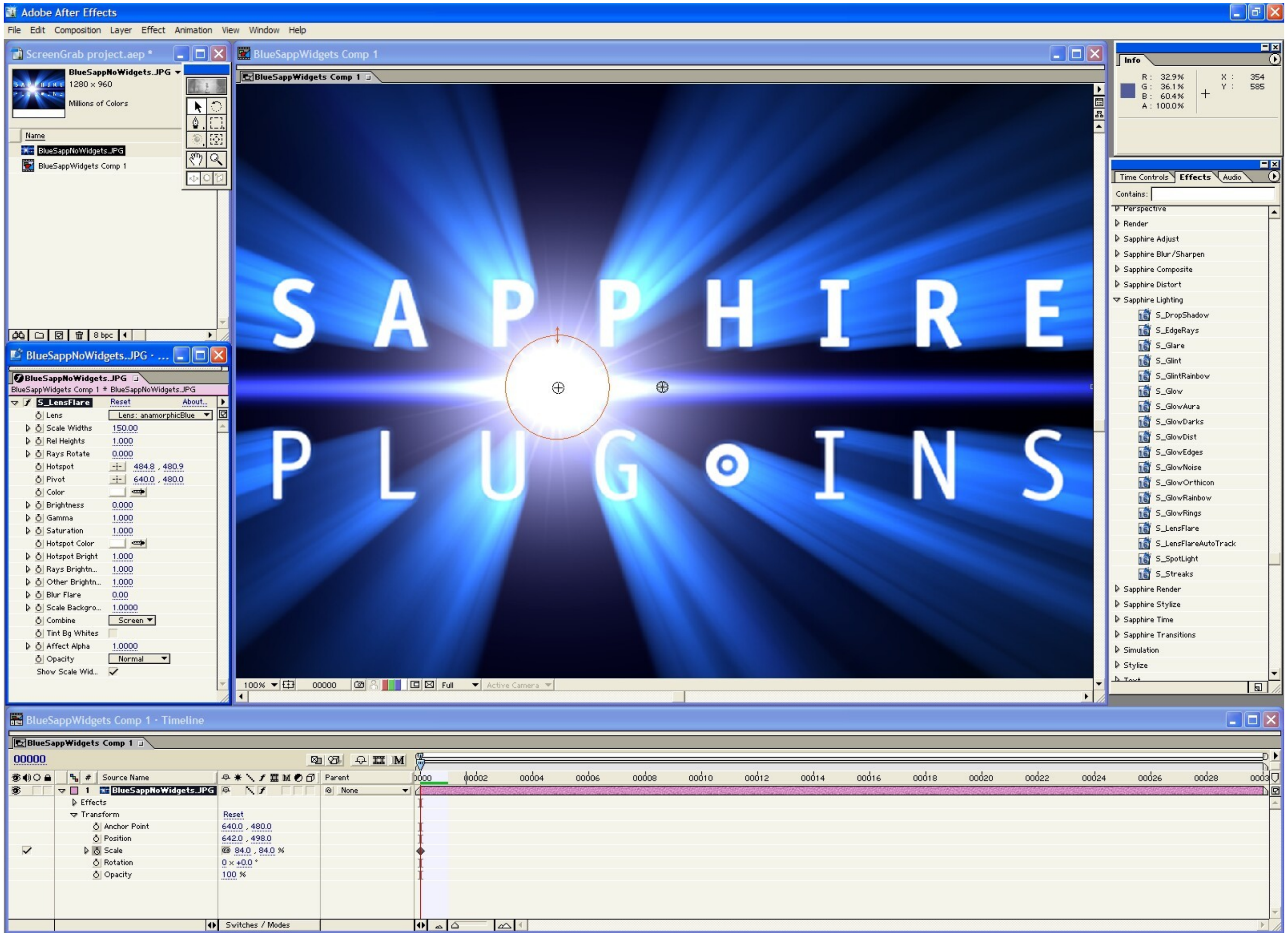
Builds in a Cross Platform Environment with SCons

Gary Oberbrunner
GenArts, Inc.
October, 2008

GenArts Company Info



BlueSappNoWidgets.JPG
1280 x 960
Millions of Colors

R : 32.9%
G : 36.1%
B : 60.4%
A : 100.0%
X : 354
Y : 585

BlueSappNoWidgets.JPG
BlueSappWidgets Comp 1 * BlueSappNoWidgets.JPG

S_LensFlare Reset About...

Lens	Lens: anamorphicBlue
Scale Widths	150.00
Rel Heights	1.000
Rays Rotate	0.000
Hotspot	484.8 , 480.9
Pivot	640.0 , 480.0
Color	
Brightness	0.000
Gamma	1.000
Saturation	1.000
Hotspot Color	
Hotspot Bright	1.000
Rays Brightn...	1.000
Other Brightn...	1.000
Blur Flare	0.00
Scale Backgro...	1.0000
Combine	Screen
Tint Bg Whites	□
Affect Alpha	1.0000
Opacity	Normal
Show Scale Wid...	✓

100% 00000 Full

00000

1 BlueSappNoWidgets.JPG None

Effects

Transform Reset

Anchor Point	640.0 , 480.0
Position	642.0 , 498.0
Scale	84.0 , 84.0 %
Rotation	0 x +0.0 °
Opacity	100 %

Plug-ins for:

- After Effects
- Final Cut Pro
- Avid
- Autodesk
- Nuke
- Quantel
- many more

GenArts' History With SCons

- Used predecessor, Cons, since before 2000
- Early user of SCons since 2001
- GenArts use of SCons:
    - Build software, documentation, tests, installers (.tardist, .rpm, .exe, .sitx, .dmg)
    - Cross-platform: IRIX, Windows 32 & 64, Linux 32 & 64, MacOS. CodeWarrior, gcc3, gcc4, MSVS (6/7/8), Intel compilers.
- Gradually got sucked into developing it...

What's Wrong With Make?

- Make considered harmful
- Repetitive

```
HelloWorld : HelloWorld.c
    gcc HelloWorld.c -o HelloWorld
```

- Recursion = nightmare
- Tightly coupled to system
- No automatic dependency tracking
- Can't track changes in Makefile
- Too low-level
    - need to use automake, imake etc. to help

Typical Automake Sample :-)

```
AM_CHECK_PYTHON_HEADERS(,[AC_MSG_ERROR(could not find Python headers)])

# get rid of the -export-dynamic stuff from the configure flags ...
export_dynamic=`(./libtool --config; echo eval echo \\
$export_dynamic_flag_spec) | sh`

# cairo
PKG_CHECK_MODULES(CAIRO, cairo >= cairo_required_version)
if test -n "$export_dynamic"; then
  CAIRO_LIBS=`echo $CAIRO_LIBS | sed -e "s/$export_dynamic//"`
fi

# cairo + cairo-xlib + gtk + pygtk
if test x"$with_pygtk" = xyes; then
  # was cairo compiled with cairo-xlib enabled?
  save_LIBS="$LIBS"
  LIBS="$CAIRO_LIBS"
  AC_CHECK_LIB([cairo], [cairo_xlib_surface_create], [], [with_pygtk=no])
  LIBS="$save_LIBS"
fi
```

Why SCons?

- SCons: whole-project view
- Readable and maintainable
- Automatic full dependency analysis
- Totally cross-platform
  - runs anywhere with Python
- Autotools tests built-in
- Customizable
- It's python! Full power of a real language

```
SConstruct:

Program('foo.c')

Program('bar',
  ['bar.c', 'funs.c'])
```

What about …

- What about Microsoft Visual Studio, XCode, Eclipse, etc.?
- These tools lock you into a single platform.
- Some are GUI-only and hard to drive from command line
  - which makes repeatable nightly buildbot runs hard
- However: SCons can export MSVS solution files for people who like IDEs
- XCode and Eclipse can also drive SCons as an external build tool.

More reasons to use SCons

- Always-correct minimal builds
  - No more “need to rebuild from scratch”
- Simultaneous variant builds
  - debug, release, profile, …
- Repositories for sharing build products
- Supports test-driven development
- SCons can tell you *why* it builds something
- Makes easy things simple, and complex things possible

Who's Using SCons?

- Autodesk (Toxik etc.)
- Blender (3d modeler)
- id Software (Doom3)
- Nullsoft NSIS
- GenArts
- Google: Steven Knight
  - Chrome, Google Earth, etc.

- Intel
- 35+ packages in Debian
- 500 member users list
- 175 member dev list
- 200+ downloads/day

"It was long past time for autotools to be replaced, and SCons has won the race to become my build system of choice. Unified builds and extensibility with Python — how can you beat that?"
— Eric S. Raymond, author of "The Cathedral and the Bazaar"

SCons Example

- Cross-platform builds, simple program

main.c:

```
main(int argc, char **argv)
{
  printf(“hi, world\n”);
}
```

SConstruct:

```
Program('hello.c')
```

Win32:

```
C:\>scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
cl /nologo /c hello.c /Fohello.obj
link /nologo /OUT:hello.exe hello.obj
scons: done building targets.
```

POSIX:

```
% scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
cc -o hello.o -c hello.c
cc -o hello hello.o
scons: done building targets.
```

SCons Scripts Are Python

- SConstruct Files Are Python Scripts
  - comments, functions, looping, conditionals, OS interface, etc.
  - keyword arguments

```
# Different source file sets per product
if product_a:
  src_files = Split('main.c file1.c file2.c')
else:
  src_files = Split('main.c file3.c'
Program(target='program', source= src_files)
```

```
# list of obj files for all sources which pass
# is_debug() filter, using python list comprehension
dbgobjs = \
  [Object(x) for x in srclist if is_debug(x)]
```

SCons is a *Declarative* System

- SCons Functions Are Order-Independent
  - First, SCons reads all scripts
  - It builds a complete dependency graph
  - Then it executes that graph, building only what is needed

SCons Example: dependencies

- Simple program
- Change header:
  - auto rebuilds
- Change source
  - rebuilds only what's needed
- MD5 signatures for change detection

```
main.c:
#include “prog.h”
main(int argc, char **argv) {
  printf(“hi,” NAME “\n”);
}
```

```
prog.h:
#define NAME “Gary”
```

```
% scons -Q
gcc -o main.o main.c
gcc -o main main.o
% vi prog.h
% scons -Q
gcc -o main.o main.c
gcc -o main main.o
% : add comment in main.c
% scons -Q
gcc -o main.o main.c
%
```

Environment

- All builds are done in a Construction Environment
- It's a python dictionary
- Builders can expand keywords

Standard environment on Linux (excerpt):

```
env['CXXCOM']= '$CXX -o $TARGET -c $CXXFLAGS $CCFLAGS $_CCCOMCOM $SOURCES'
env['CXX']= 'g++'
env['CXXFLAGS']= []  # nothing, null, empty list
env['_CCCOMCOM']= '$CPPFLAGS $_CPPDEFFLAGS $_CPPINCFLAGS'
env['_CPPDEFFLAGS']= '${_defines(CPPDEFPREFIX, CPPDEFINES, CPPDEFSUFFIX,
  __env__)}'
```

Common Customizations

- CPPPATH: finding headers
- CPPDEFINES: #defines
- CFLAGS, CCFLAGS, CXXFLAGS
- LINKFLAGS, SHLINKFLAGS
- CCCOM, LINKCOM

Libraries

- static/shared
- LIBS=
- LIBPATH=
- Understands lib pre/suffixes (libfoo.so, foo.dylib, etc.)

SConstruct for building lib:

```
env = Environment()
lib = env.Library('foo', ['f1.c',
'f2.c', 'f3.c'])
shlib = env.SharedLibrary('sharedfoo',
  ['f1.c', 'f2.c', 'f3.c'])
```

SConstruct for linking with lib:

```
env.Program('prog.c',
  LIBS=['foo', 'bar'], LIBPATH='.')
```

```
% scons -Q
cc -o f1.o -c f1.c
cc -o f2.o -c f2.c
cc -o f3.o -c f3.c
ar rc libfoo.a f1.o f2.o f3.o
ranlib libfoo.a
cc -o prog.o -c prog.c
cc -o prog prog.o -L. -lfoo -lbar
```

```
C:\>scons -Q
cl /nologo /c f1.c /Fof1.obj
cl /nologo /c f2.c /Fof2.obj
cl /nologo /c f3.c /Fof3.obj
lib /nologo /OUT:foo.lib f1.obj f2.obj
f3.obj
cl /nologo /c prog.c /Foprog.obj
link /nologo /OUT:prog.exe /LIBPATH:.
foo.lib bar.lib prog.obj
```

Builders

- SCons comes with many built-in builders:
  - C/C++ (many)
  - Yacc/Lex
  - Java
  - FORTRAN (f77, g77, ifort, etc.)
  - Assemblers (various)
  - LaTeX, SWIG, Qt (Qt4 coming)...
- It's easy to add your own

File Nodes

- All files and dirs are represented internally by Nodes
- Cross-platform
- Path methods (.path, .abspath, etc.)
- All Builders return a list of Nodes

```
p = Program('prog.c') # p is a Node
Install('/usr/bin', p)
```

Scanners

- Scanners are used to find implicit dependencies
- cpp_scanner finds C/C++ headers
- Others for resource files, FORTRAN, LaTeX, etc.
- Can add your own

Aliases

- Aliases are a name for a set of Nodes, usually build targets
- Can be used on command line or as source for further build targets

```
SConstruct:
Alias('all', Install('/tmp', env.Program('prog.c')))
```

```
% scons all
… builds and installs prog.exe
```

Parallel builds

- scons -jN …
- Always keeps N jobs running if possible
- Uses dependency graph
- Can be hooked into distcc for distributed build farming (great for nightly builds)

SConf: Autoconf functionality

- In a cross-platform world, you need to know what your system looks like
- sizeof(int)? memcpy/bcopy? size_t? libm?

SConstruct:

```
env = Environment()
conf = Configure(env)
if conf.CheckType('size_t'):
  conf.env.Append(CPPDEFINES, 'HAS_SIZE_T')
env = conf.Finish()
```

- Configure calls are cached for speed

The *Command* Builder

- Command() can run any shell command(s)
- Perfect for one-off commands

```
SConstruct:
env.Command(target='in', source=None,
  “echo 'hi there' > $TARGET”)

env.Command(target=”out”, source=”in”,
  “sed -e 's/hi/bye' < $SOURCE > $TARGET”)
```

```
% scons
echo 'hi there' > in
sed -e 's/hi/bye' < in > out
%
```

- Command can even be a python function

Adding Builders

- Anything you can do in a shell or in python, you can do in a Builder.

SConstruct:

```
env = Environment()
bld = Builder(action = 'foobuild < $SOURCE > $TARGET')
env['BUILDERS']['Foo'] = bld
env.Foo('file.c', 'file.c.in') # build file.c from .in
env.Program('hello', ['hello.c', 'file.c'])
```

```
% scons -Q
foobuild < file.c.in > file.c
cc -o hello.o hello.c
cc -o file.o file.c
cc -o hello hello.o file.o
%
```

Issues

- Slow to start for big projects
  - No-op build times can be an issue
  - getting better
  - recipes on wiki for speedups
- Memory usage
  - also getting better
- Complex builds may still require a fair amount of python code
- Java support not as good as Ant/Maven (specialized Java build tools)

Links

- SCons site: www.scons.org
  - man page, Users Guide
  - downloads
- SCons wiki: www.scons.org/wiki
  - Community contributions
  - Developer info
- Mailing list: users@scons.tigris.org