

The Power to Create

Constructing a powerful asset and build pipeline using the UNIX toolset

Background Information

- Daniel P. Wright
- Computer Games Programming, 2007

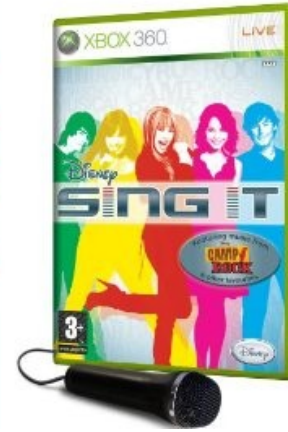
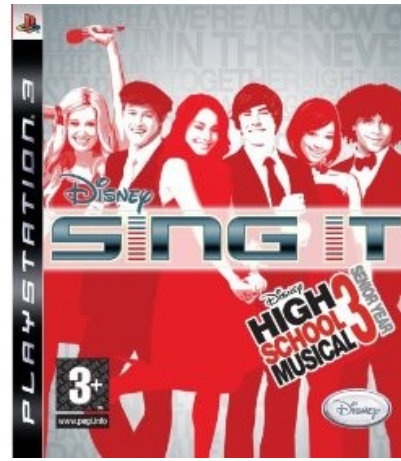
Zoë Mode

- Joined Zoë Mode in December of that year
- Worked in various areas, particularly gameplay, some parts of the singing tech, and asset/build management

The Problem

- Modern games development typically involves a number of different configurations of your product (called SKUs)
 - Different platforms
 - Different regions
 - Sometimes even different products!
- Each of these different configurations have subtly different code and data
- In addition, teams are getting bigger and bigger, and so is the codebase they're working on – and the number of assets

An Example



- “Disney Sing It”: released late 2008
- PS2, PS3, Xbox 360, Wii, and PC
- One general version, one specific to High School Musical
- EFIGS+D, EFS, ESwDaNo, ERussian
- Around 2TB of video data

A Solution: UNIX

- Not the full UNIX operating system, but a collection of tools common across UNIX systems
- Cygwin (<http://cygwin.org/>) on Windows
 - Natively supported on everything else
- The main tools we use are make and bash
 - Though really, these tools are just a way of automating the use of other tools
- Command-line tools make for easier automation

The Core Philosophy

- Many small programs working together
 - A program is anything that can be run
 - One line of an interpreted scripting language is a program, as far as we are concerned
- Each program performs one very specific function, but performs it well
- Programs can be linked together by piping, or by assigning either output or return values to a variable
- Can be confusing when you first start
 - grep, sed, awk, bash, cat, tac, tr, dd...

bash

- Bourne Again SHell: A progression from the old Bourne shell (sh)
- Default shell on many UNIX systems
 - Ubuntu, Fedora, Mac OS X, Cygwin
- Very powerful when compared with the DOS or NT command shells
 - Fully-featured interpreted language
 - Geared toward the running and interaction of other programs

bash: Some Basic Features

- If/then/else statements, for/while loops
 - As well as typical comparison operators, bash offers some file-specific queries, such as `if [-e filename]`
- Piping: `|`
 - Redirects the STDOUT of one program into the STDIN of the next
- Redirections: `<`, `>`, `>>`, etc.
 - Redirects STDIN, STDOUT, or STDERR to or from a file
- Inline evaluation: `$()`, `` ``
 - Evaluates the output of a program for use within your script

bash: Some Examples

- To find the longest (largest filesize) video in a directory and rename it “longest.avi”

```
mv $(ls -S | head -n 1) longest.avi
```

- To generate a .csv listing of file names, extensions, and modified dates, for importing into a spreadsheet

```
find . -type f | while read FILE
do
EXT=$(echo $FILE | sed 's:^.*\.([a-zA-Z0-9]*)$:\1:')
MODIFIED=$(ls -l --time-style=+%Y-%m-%d $FILE | awk '{print $7}')
echo $FILE,$EXT,$MODIFIED
done
```

make

- make is a program to manage *dependencies*
- As a result, it is very different from most programming languages you are likely to have encountered
 - Describe the problem, let it find the solution
 - No single, clear entry point
- It is one of many different solutions for managing dependencies
 - CMake, SCons, ant, Jam
 - make is one of the oldest
 - More multi-purpose than some, and interfaces very nicely with bash

make: Some Syntax

- Variable assignment

`CC=gcc, CC:=gcc`

- Variable evaluation

`$a, $(CC)`

- Targets and Dependencies

`target: dependencies`

`steps to produce`

- Note the indentation!
- Target can be a filename, a directory name, or some arbitrary name you give it
- If a dependency can't be found, an equivalent rule with that as the target will be sought and run

make: More Useful Stuff

- Some more syntax:
 - `$@` evaluates to the target
 - `$<` evaluates to the first dependency
 - `$^` evaluates to a list of all dependencies
 - `$?` evaluates to a list of all dependencies which are newer than the target
- Some useful commands:
 - `$(warning var is $(var))` is a useful way to output variables
 - `$(subst .c,.o,$(sources))` generates a list of object files from a list of source files
 - `$(shell find . -name '*.c')` generates a list of source files by calling the shell command *find*

An Example makefile

```
sources:=$(shell find . -name '*.cpp')  
objects:=$(subst .cpp,.o, $(sources))  
output:=executable_name
```

```
.PHONY: all  
all: $(output)
```

```
$(executable_name): $(objects)  
    $(LD) $(LDFLAGS) -o $@ $^
```

```
%.o: %.cpp  
    $(CXX) $(CXXFLAGS) $(CPPFLAGS) $<
```

```
.PHONY: clean  
clean:  
    $(RM) $(output) $(objects)
```

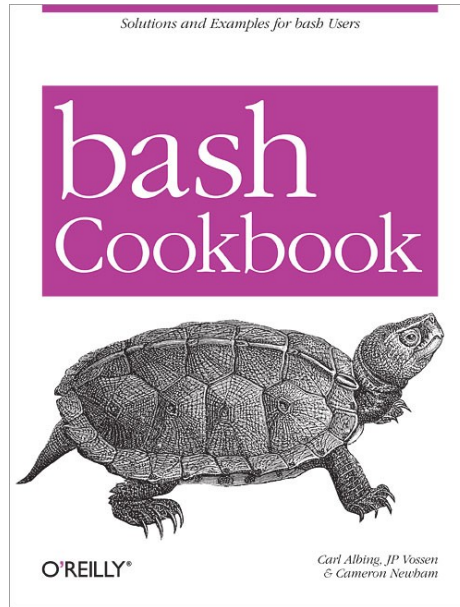
make's Not Just For Code!

- Since all make cares about is targets and the dependencies those targets require, it can be applied to many problems
 - A video requiring a video track and various audio tracks
 - Platform-specific texture and audio formats
 - Code documentation (!)
- It has two major strengths:
 - It can resolve complicated dependency mazes
 - It avoids unnecessary repetition
- These make it perfect for data conversion

make In Use

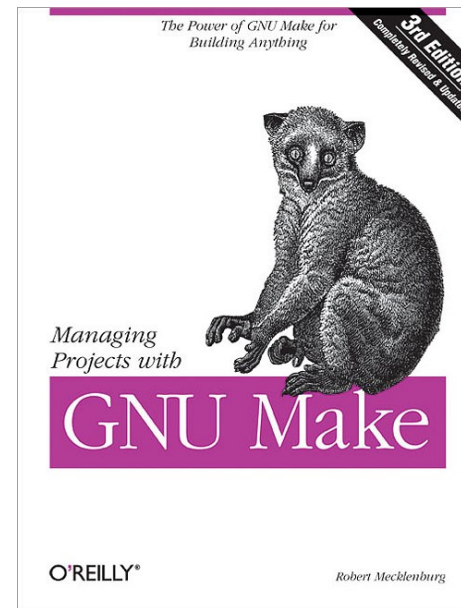
- On “Disney Sing It”, we had three make systems in place:
 - The Code Make
 - The Data Make
 - The Video Make
- The first two were run locally; the latter was run in a loop on a dedicated build machine
- Each could easily make a particular platform
 - Code: `make ps2/debug`, `make 360/release`
 - Data: `make platform=wii`, `make platform=ps3`
- Data could be changed locally, or updated by someone else
- Only edited files would be converted

Useful Resources



- bash Cookbook
- Carl Albing,
JP Vossen,
Cameron Newham
- ISBN: 0596526784

- Managing Projects with GNU Make
- Robert Mecklenburg
- ISBN: 0596006101



Contact Details

dani@dpwright.com
<http://www.dpwright.com/>