1 INSTALLING AND BUILDING PANDA

1.1 VR Studio View system

1.1.1 Overview

PANDA/DIRECT/Toontown source code is divided into multiple CVS source trees:

- **WINTOOLS/UNIXTOOLS**: Third party libraries/Software Development Kits (SDKs). These are operating system-specific installations of third party software. If you are running under windows, attach to WINTOOLS, if you are operating under Unix, attach to UNIXTOOLS. Includes Python, Netscape NSPR, and Miles audio
- **DTOOL**: Low-level libraries/tools for controlling CVS views, and interrogating C++ code.
- **PANDA**: Low-level 3D graphics engine code. Primarily C++. Includes code for graphics/scene graph setup/manipulation/rendering, graphic state guardians (interfaces to OpenGL, Direct X, Renderman), and source code for many PANDA systems: animation, audio, gui, input devices, particles, physics, shaders, etc.
- **DIRECT**: Mid-level tools/subsystems which supports show development, and scene-composition. Primarily Python with some C++. Includes code which sets up and initializes PANDA (using Python wrapper functions which call low-level C++ counterparts). Includes python modules for mid-level show coding systems: actors, directdevices (high-level wrappers around low-level input devices such as joysticks, magnetic trackers, etc.), finite state machines, 2D gui elements, intervals, tasks, and the DIRECT tk widget classes and panels.
- **OTP**: High-level show code that is generic enough to be shared across our MMP projects. Includes avatar, chat, friends, nametags, etc. Primarily Python code.
- **TOONTOWN**: High-level show-specific code. Primarily Python. Includes modules/classes for high-level toontown systems: battle, buildings, dna, installer, launcher, login, minigame, quest, safezone, shtiker, suits (cogs), toons, trolley, and tutorial.

The VR Studio view system enables users to quickly switch between using personal views (stored and compiled on one's local machine) and public versions which are compiled nightly and stored in public directories. The install version is built with OPTIMIZE level = 2. Release is built OPTIMIZE level=3. (See Config.pp section below)

With the view system, users only have to have personal copies of CVS trees that they are actively changing (e.g. show coders would only need a personal copy of Toontown, OTP, and Direct, 3D engine developers would only need personal copies of PANDA), for all other trees they can use the public copies.

In practice, many people choose to have their own copies of all trees for several reasons. If you take your work home on a laptop you can be self-contained without needing VPN access. Sometimes network access is slow, so loading the Panda dlls off the install

directory is slow. Sometimes you want to be buffered from the latest code so you might have your own CVS view of the tree so you can isolate yourself from changes.

1.1.2 Commands

- Personal copies of CVS trees are typically found in the directory:

  ~/player

  (where ~ is the unix shorthand for your home directory) Before executing the following commands, switch to that directory:

  ```
  > cd ~/player
  ```

- The first time you attach to a cvs view you will need to log in using:

  ```
  > cvs login
  ```

  Just hit return when prompted for a password.

- To attach to a CVS view use the cta command (the script used to attach to CVS views):

  ```
  > cta dtool install
  ```

  or

  ```
  > cta dtool personal
  ```

  The command

  ```
  > cta dtool
  ```

  will attach you to your default view (see the vspec section below).

  Attaching to a view initializes environment variables such as $DTOOL, $PANDA, and updates path variables to include important directories in the new views.

- To check out personal copies of the various trees:

  ```
  > cd /usr/$USERNAME/player

  > cvs co dtool
  ```

  or

  ```
  > cvs co panda
  > cvs co direct
  > cvs co otp
  > cvs co toontown
  ```

1.1.3 vspec Files: View Configuration

Vspec (view specification) files can be found in the directory: /usr/local/etc. These files specify the location of the various views (such as install, and release). They also specify the location and cvs server for the various personal views. The line:

mrmine:croot:path=/usr/mine/player/panda:server=,pserver,mine@dimbo,/fit/cvs

specifies that mrmine's personal panda tree can be found in /usr/mine/player/panda and the cvs server is dimbo with the code repository found in /fit/cvs.

The line:

default:ref:name= personal

specifies that the default view for this tree is the personal view. If you wish your default view to be some other tree (install, for example), edit the line to read:

default:ref:name=install

1.1.4 CVS: Concurrent Versions System

The VRStudio view system is closely coupled with an open source version control system called CVS. What the view system allows you to do is to quickly switch between different versions of CVS trees. For a detailed description of CVS (including FAQ and manual) go to www.cvshome.org. Some of the more commonly used cvs functions include:

COMMAND	**DESCRIPTION**
cvs checkout *module* cvs co *module*	Get a copy of the source code for the specified module
cvs commit *module*	Check changes into the repository
cvs update [-d –P] cvs up [-d –P]	Get your work tree in sync with the repository. Good options to use with this command are –d and –P which create new directories that have been added and prune away empty directories that are now obsolete.
cvs log [files …]	Print out history information for files
cvs diff [-r *rev1*] [files…]	Show differences between revisions. The –r option is used to specify a revision number to compare with the working version of the file.
cvs annotate [files…] cvs ann [files…]	Show last revision where each line in a file was modified (and by whom)
cvs add [files…]	Add a new file/directory to the repository. You need to do this every time you create a new file or directory
cvs remove [files…]	Remove an entry from the repository. You need to delete the file from your working directory (or use the –f option) before you can execute this command

Additionally you may want to setup a .cvsrc file in your home directory to specify default parameters for common CVS commands. Popular defaults are:

log -N

diff -uN

rdiff -u

update -Pd

checkout -P

release -d

1.2 Compilation

1.2.1 Config.pp: Compilation Customization

The Config.pp file is used to customize compilation of the various CVS trees.

- System-level Config.pp file is found in $WINTOOLS/panda/etc
- Personal Config.pp file is found in $HOME directory. This file overrides settings in the system level Config.pp file.
- Used to specify optimize level (amount of code optimization and debug information). For example, to set the optimize level to 3, include the following line in your personal Config.pp file:

  #define OPTIMIZE 3

  The different optimization levels are:

  - OPTIMIZE 1: Full debugging, no compiler optimizations
  - OPTIMIZE 2 (Install): Full debugging, with non-contradictory compiler optimizations
  - OPTIMIZE 3 (Release): Full compiler optimizations with whatever debugging is supported without conflicts with the optimization.
  - OPTIMIZE 4: (Publish) Full compiler optimizations, no debugging symbols. All assertions are removed.
- Used to include/exclude code subsystems such as the VRPN device layer, audio, etc. To compile with VRPN include the line:

  #define HAVE_VRPN yes

  or to include the stats feature:

  #define DO_PSTATS yes
- To turn off a feature, define a variable to no value:

  #define DO_PSTATS

1.2.2 ppremake: generation

The ppremake command is used to generate makefiles that are consistent (i.e. optimize levels, locations of libraries, packages to build, etc.) with the settings specified in the Config.pp file. You must ppremake after updating your tree with CVS and before compiling that tree. To use ppremake change to the root of the tree and execute the command ppremake. To update Panda for instance:

```
> cd $PANDA
> ppremake
```

You will see a lot of information describing which makefiles are being generated.

If you have a compilation error (and in particular a linking error), a good rule of thumb is to execute ppremake and then try the build again. This will clear up many problems. If that doesn't work, try making clean (see next section) before rebuilding the tree

1.2.3 make: Compilation

To build code on Windows platforms, use Visual C++'s make command-line complier command.

To compile and install (e.g. move libraries to the lib directory) run:

```
> make install
```

To remove files from the install directories:

```
> make uninstall
```

To clean out intermediate files, previous versions of compiled libraries, etc:

```
> make clean
```

1.2.4 genPyCode: Generating the Python/C++ Foreign Function Interface

Every time you change and compile C++ code (in particular, when you change a C++ class API) it is necessary to generate python code. This builds the python wrappers around C++ classes and methods (automatically generating the files found in the $DIRECT/lib/py directory). The files in the lib/py directory represent shadow classes in Python for C++ classes. This allows the programmer to treat the C++ classes as first-class Python classes. You can use and inherit from C++ classes directly in Python, just like any regular Python class.

When generating python code, run

```
> genPyCode
```

2 RUNNING PANDA APPLICATIONS

2.1 Execution Setup

2.1.1 Configrc: Execution Customization

- Found in $HOME or ~/ directory
- Overrides/customizes default PANDA settings

- Read in during PANDA startup, must restart PANDA for changes to take effect
- Allows users to configure display settings, graphics libraries, application behavior, etc.

The following table lists some of the more commonly-used Configrc variables. Default values are the value used if the variable is not defined in your Configrc file.

Variable	Values	Default	Comments
load-display	pandagl pandadx7 pandadx8 pandadx9		Specifies which graphics GSG to use for rendering (openGL or DirectX)
win-width	number of pixels	640	Specifies width of PANDA window in pixels
win-height	number of pixels	480	Specifies height of PANDA window in pixels
fullscreen	#t or #f	#f	Enables full screen mode
show-fps-meter	#t or #f	#f	Shows frame rate meter
model-path	a path string		Adds specified path to the list of paths searched when loading a model
texture-path	a path string		Adds specified path to the list of paths searched when loading a texture
sound-path	a path string		Add specified path to the list of paths searched when loading a sound
audio-library-name	miles_audio, null	miles_audio	Set to null if you wish to run without sound.
notify-level-[package]	fatal, error, warning, info, debug, spam		Sets notify level for different panda packages (such as dxgsg, graph, etc.) to control amount of information printed out during execution (fatal being least, spam the most)
want-tk	#t or #f	#f	Enables support for using Tkinter/PMW (python's wrappers around Tk)
want-directtools	#t or #f	#f	Enables directtools, a suite of interactive object/camera

			manipulation tools
direct-gui-edit	#t or #f	#f	Enables direct manipulation of direct gui widgets (buttons, labels, frames, etc)

2.2 Emacs Python Mode

The emacs distribution includes a python mode list file:

emacs-[version]/lisp/progmodes/python-mode.el

This is a lisp file which is used to configure the python major mode for editing python programs. It includes useful functions for automatically formatting python code. We have customized this file to utilize the ppython script described above (instead of just running python). The modified version of the file can be found in:

$DIRECT/src/directscripts/python-mode.el

This should be copied to the lisp/progmodes directory in your emacs installation.

- To switch to python mode execute:

```
M-x python-mode  M-x is actually <alt-x>
```

- To start up a python shell execute:

```
M-x py-shell
```

  To start up a **debug** python shell (if you compiled your C++ code with optimize levels 1 or 2 and generated python code using genPyCode win-debug) execute:

```
M-x pyd-shell
```

  If you start up a python debug shell you will see the following lines in your emacs buffer:

  Adding parser accelerators …
  Done.

- When using python mode in emacs, you start your simulation (the application's main loop – which includes updating the graphics window and the application) using the command:

```
run()
```

  This has a keyboard shortcut ^D (Control-D)

- To interrupt the simulation, hit the Enter key, this should return you to the Python prompt:
  >>>

- If you are running Python in one buffer and editing Python source code in another, you can redefine classes on the fly by placing your cursor within the source code

for that class (somewhere between the class keyword and the end of the class definition) and typing the command:

C-c C-v (Control-c then Control-v)

This should (in most cases) update that class within the running python application. There are certain cases which do not work (such as commands bound to tk widgets).

3 USING PANDA

3.1 Utilities

The following sections include sample code demonstrating the use of PANDA3D's more important features. When learning to use PANDA there are several important utilities to help you out:

- To get a detailed listing of a classes' methods and attributes, use the pdir command:

```
>>> pdir(NodePath)
```

  or, if you have a node path object:

```
>>> pdir(camera)
```

- Alternately, you can use the inspect utility

```
>>> inspect(camera)
```

  this is more interesting if you have a Python object (and not a handle to a C++ object):

```
>>> inspect(base)
```

  as you can see, the inspect viewer allows one to see the current values of a classes' attributes. If the attribute is dynamically updating, you may have to click on a value to refresh its value. If an attribute is a structure, double clicking on it will dive the inspector down one level to inspect the contents of the structure.

- To interactively adjust the position, orientation, and scale of a node path use the placer panel:

```
>>> camera.place()
```

- To explore a NodePath's hierarchy use the scene graph explorer:

```
>>> render.explore()
```

- To adjust a NodePath's color use the rgb panel:

```
>>> smiley.rgbPanel()
```

3.2 Examples

```
################################################
# STARTING AND RUNNING
################################################

runPythonEmacs
<alt-x> py-shell
# Note: use pyd-shell if running a debug build like install

# To get a standard Panda window
from ShowBaseGlobal import *

# Or to start toontown
from ToontownStart import *

In Emacs:
Press <enter> to break out of the main loop and get a python prompt
Press <ctrl-d> to run again


################################################
# NODEPATHS
################################################

# Load a model. Loading a model returns a NodePath
# The full path includes $TTMODELS variable, which is a lengthy path.
smiley = loader.loadModel('models/misc/smiley')

# Loaded models by default are not drawn because they are under a special
# node called hidden.  Hidden is like storage for objects you do not want
# drawn presently, but will use in the future.  To put smiley in the
# rendered scene graph, use the reparentTo command specifying the top node that
# is drawn, called render.
smiley.reparentTo(render)

# Position smiley to be 10 feet in front of the camera so we can see
# it. Position is specified as x,y,z or Left, forward, up.
smiley.setPos(camera, 0, 10, 0)

# To position relative to its parent's coordinate system
smiley.setPos(0,10,10)

# Turn smiley to face us. Rotation is specified as h,p,r or heading, pitch,
# and roll
smiley.setHpr(180,0,0)
# or
smiley.lookAt(camera)

# Make smiley turn red. Color is specified as r,g,b,a
smiley.setColor(1,0,0,1)

# To see everything a NodePath can do, use the pdir function.  pdir is like
# Python's builtin dir() but has some special functionality to look in
# super-classes and print nicely.
pdir(smiley)
```

```
# You may also move the camera around using the left, middle, and right
# mouse buttons

##################################################
# EVENTS
##################################################

# Events (for example, mouse clicks) are all managed through the Messenger
# class ($DIRECT/src/showbase/Messenger.py).  There is one global messenger
# object called messenger.  Events are used for broadcasting capability
# through code and as a guarding mechanism to control when responses will
# be executed.  The messenger system is object-oriented, so you must have
# an object to listen for an event. To inherit the messaging API, subclass
# from DirectObject.

import DirectObject
class Hello(DirectObject.DirectObject):
    def __init__(self):
        self.accept('mouse1', self.printHello)
    def printHello(self):
        print 'Hello, mouse1 was clicked'
    def printEscape(self):
        print 'Escape was pressed'
    def printMyEvent(self):
        print 'My Event'

h = Hello()

# To accept an event:
h.accept('escape-up', h.printEscape)

# To accept an event only once, then automatically ignore future events of
# the same name:
h.acceptOnce('escape-up', h.printEscape)

# To ignore an event:
h.ignore('escape-up')

# To ignore all events this object may have been accepting:
self.ignoreAll()

# To send an event:
h.accept('myEvent', h.printMyEvent)
messenger.send('myEvent')

# For debugging, you can tell the messenger to show every event it sends
# and accepts. Toggle it again to turn this off.
messenger.toggleVerbose()

# Mouse and keyboard events propagate through the messenger from C++ and
# are sent just like any other event you send by hand. Mouse events are of the
# form "mouse1", "mouse1-up". Keyboard events are simply the letter, for
# example, "k", "k-up".

# To see who is accepting which events:
print messenger
```

```
################################################
# INTERVALS
################################################

## BACKGROUND ##
## Intervals are used to create timelines of dynamic behavior (objects
## moving, colors changing, etc.).  The advantage of intervals is that an
## object's behavior is defined for the entire duration of the interval.
## This means one can jump into an interval at any time and the behavior of
## the objects controlled by the interval is completely defined.  A
## detailed set of interval examples can be found in
## $DIRECT/src/intervals/IntervalTest.py

# There are several different types of intervals:
# ActorInterval - for playing back animations
# FunctionInterval - for executing functions
# LerpInterval - for linearly interpolating nodePaths or functions
# MopathInterval - for moving an object along a motion path
# ParticleInterval - for playing back a particle effect
# SoundInterval - for playing back a sound
# WaitInterval - a no-op to fill time
#
# These intervals are grouped using container classes:
# Sequence - grouping several intervals to be played in sequence, one after
#           the other
# Parallel - grouping several intervals to be played in parallel

## USING INTERVALS ##
# The following intervals will lerp an object's position, move it along a
# path, play an animation and play some sounds

# This loads in the different interval class definitions
from IntervalGlobal import *

## LERP INTERVAL ##
# Load up a model to move along the motion path using a MopathInterval
smiley1 = loader.loadModel('models/misc/smiley')
smiley1.reparentTo(render)
# Create interval.  Arguments are:
# nodePath.posInterval(duration, pos, startPos=None, other=None, blendType =
None)
# If startPos is not set, lerp is from current pos
# If blendType is not set, linear interpolation is used
smiley1LerpInterval = smiley1.posInterval(3, Point3(0,0,5),
                                          startPos=Point3(0,0,0),
                                          blendType = 'easeInOut')

# You can play, stop, loop, jump to any time, and popup an interactive
# control panel on this interval
smiley1LerpInterval.play()
# To stop playback
smiley1LerpInterval.stop()
# To play in a looping mode
smiley1LerpInterval.loop()
# To jump to a particular time
```

```
smiley1LerpInterval.setT(5.0)
# To pop up interactive controls
smiley1LerpInterval.popupControls()

## MOPATH INTERVALS ##
# Load up a model to move along the motion path using a MopathInterval
smiley2 = loader.loadModel('models/misc/smiley')
smiley2.reparentTo(render)

# Load up a motion path
import Mopath
# This directory must be on your model-path
motionPath = Mopath.Mopath()
motionPath.loadFile('phase_6/paths/dd-e-w')

# Create the MopathInterval
smiley2MopathInterval = MopathInterval(motionPath, smiley2, 'smiley2path')

## ACTOR INTERVALS ##
# Intervals can be created to control animated characters
# Load up an animated character to attach to smiley2
import Actor
donald = Actor.Actor()
donald.loadModel("phase_6/models/char/donald-wheel-1000")
donald.loadAnims({"steer":"phase_6/models/char/donald-wheel-wheel"})
donald.reparentTo(smiley2)

# This will create an anim interval that plays through the animation once
donaldSteerInterval = donald.actorInterval('steer')

# This will create an anim interval that is started at t = 0 and then
# loops for 10 seconds
donaldLoopInterval = donald.actorInterval('steer', loop=1, duration = 10.0)

## SOUND INTERVALS ##
# Intervals can be used to play back sounds
# Load up a sound
shipbell = base.loadSfx('phase_6/audio/sfx/SZ_DD_shipbell.mp3')
# Create a sound interval
shipbellSoundInterval = SoundInterval(shipbell, name='shipbell')

# Again with the seagull sound
seagull = base.loadSfx('phase_6/audio/sfx/SZ_DD_Seagull.mp3')
seagullSoundInterval = SoundInterval(seagull, name = 'seagull')

## FUNCTION INTERVALS
# FunctionIntervals can be used to execute arbitrary functions
from DirectUtil import setBackgroundColor

def func1():
    base.setBackgroundColor(0,0,1)

def func2():
    base.setBackgroundColor(0,0,0)

# Create the function Intervals
functionInterval1 = Func(func1)
```

```
functionInterval2 = Func(func2)

# FunctionIntervals can be used to throw events.
donaldDone = Func(messenger.send, 'donald-is-done')
# Create a function to execute on receiving event
def handleDonaldDone():
    print 'Donald is done'

# Since Intervals inherit from DirectObject, they can handle their own events
donaldDone.accept('donald-is-done', handleDonaldDone)

## WAIT INTERVALS ##
# Wait Intervals can be used to add arbitrary delays to Sequences
waitInterval1 = Wait(1.0)

## SEQUENCES ##
# Intervals can be played in sequence using Sequences. Intervals can be played
# one right after the other, or a delay can be specified between subsequent
# intervals
donaldSeq = Sequence(donaldSteerInterval,
                     shipbellSoundInterval,
                     functionInterval1,
                     donaldSteerInterval,
                     shipbellSoundInterval,
                     functionInterval2,
                     waitInterval1,
                     donaldDone,
                     name = 'donaldSeq')
# Like Intervals, Sequences can be played, stopped, looped, and controlled via
# an interactive panel
donaldSeq.play()

# Create a few more sequences
smiley1Seq = Sequence(smiley1LerpInterval, name = 'smiley1Seq')
smiley2Seq = Sequence(smiley2MopathInterval,
                      seagullSoundInterval,
                      name = 'smiley2Seq')

## PARALLELS ##
# Intervals can be played in parallel (simultaneously) using Parallels.
# Make the dock lerp up so that it's up when the smiley reaches the end of
# its mopath
par  = Parallel(donaldSeq, smiley1Seq, smiley2Seq,
                name = 'par')

# Like Intervals and Sequences you can play, loop, setT, popupControls on a
# parallel
par.play()


##################################################
# TASKS
##################################################

# Tasks are useful for executing operations you want run every frame - that is,
# every simulation step.  Code can be found in $DIRECT/src/task/Tasks.py To
# spawn a task, you must first have a Python function with specific
```

```
# properties. It must take one argument, which is its own Task object.
# This task object is useful for querying state of the task, like the time
# that has passed since it was first run (task.time)

# The function must also return a code that tells the task manager if it is
# done (Task.done) or should continue until next frame (Task.cont)

import Task

# This task runs for 2 seconds, then prints done
def exampleTask(task):
    if task.time < 2.0:
        return Task.cont
    print 'done'
    return Task.done

# taskMgr is the single global TaskManager. To make a task start running,
# you tell the taskMgr to add it, and give it a name:
taskMgr.add(exampleTask, 'myTaskName')

# To remove a task from the task manager:
taskMgr.remove('myTaskName')

# To see what tasks are running, print the taskMgr. It also shows how long
# it took to execute each task on the list for profiling. You will see a few
# tasks that are created for you:
# dataloop     - processes the data graph each frame
# tkloop       - processes Tk gui events each frame (if tk is running)
# eventManager - processes all Panda and Direct events
# igloop       - draws the scene
print taskMgr


##################################################
# FINITE STATE MACHINES (FSM)
##################################################

# Much of our state keeping is done in finite state machines.  Code for our
# FSM is in $DIRECT/src/fsm/FSM.py and $DIRECT/src/fsm/State.py Below is a
# simple FSM for a light switch with two states (on and off)

import FSM
import State

# First define callbacks for the states you will have
def enterOn():
    print 'enterOn'

def exitOn():
    print 'exitOn'

def enterOff():
    print 'enterOff'

def exitOff():
    print 'exitOff'
```

```
# Each state gets a name, a function to call when entering the state, a
# function to call when exiting the state, and a list of state names it is
# allowed to transition to.
fsm = FSM.FSM('lightSwitch',
              [ State.State('on', enterOn, exitOn, ['off']),
                State.State('off', enterOff, exitOff, ['on'])
                ],
              'on', # Initial state
              'off' # Final state
              )

# Tell the fsm to enter its initial state:
fsm.enterInitialState()

# To request the fsm to transition to a new state: A warning will be
# printed if the fsm is not allowed to make the transition
fsm.request('off')

# To see a graphical view of the fsm:
fsm.view()

# Tell the fsm to go to its final state:
fsm.requestFinalState()

# NOTE: Finite state machines can also be nested. A state can have any
# number of child FSMs inside it (using the command addChild).  When the
# parent state is entered, all child FSMs transition to their initial
# state. When the parent state is exited, all child FSMs transition to
# their final state.


##################################################
# DIRECT GUI
##################################################

# The Direct Gui system is used to create buttons, labels, text entries,
# and frames within the graphics window.  All direct gui items can be
# decorated with text, images, and/or 3D Graphics objects.  Commands can be
# associated with button clicks, widget entry/exit, and clicking of the
# button.
# A detailed set of examples can be found in $DIRECT/src/gui/DirectGuiTest.py

# To get access to Direct Gui classes and methods:
from DirectGui import *

## DIRECT BUTTON
# Here we specify the button's command
def dummyCmd():
    print 'Executing Button Command!!!!'

b1 = DirectButton(text = 'My First Button',
                  command = dummyCmd)

# Buttons are first-class graphical objects (they inherit from NodePath).
# Thus buttons can be positioned, oriented, scaled, colored, etc. just like
# any node path
b1.setScale(.1)
```

```
# Buttons are all descendents of the node aspect2D.  The coordinate system
# of aspect2D is X (left-right), Y (into the screen), Z (up-down)
b1.setZ(0.8)

# To see what kind of options can be set on a button type:
b1.configure()

# You can change attributes on a button using the key index operator
b1['text'] = "New Text"
b1['relief'] = RAISED
b1['pad'] = (0.1, 0.1)
# If you wish to resize the button's background to fit around the text, call:
b1.resetFrameSize()

# Callbacks can be defined for various button events
def enterCallback(event):
    print 'Welcome to the button!'

def exitCallback(event):
    print 'Goodbye!'

b1.bind(ENTER, enterCallback)
b1.bind(EXIT, exitCallback)

## DIRECT LABELS
# Labels are like buttons, without commands
dl = DirectLabel(text = 'Peaceful', text_scale = (0.5, 0.5),
                 image = 'phase_3.5/maps/middayskyB.jpg')
dl.setScale(.2)

# DIRECT FRAMES
# Frames can be used as containers to hold other buttons, labels, and frames.
df = DirectFrame(image = 'phase_4/maps/garden_bricks2.jpg')
df.setScale(0.7)

# Existing buttons can be directly parented to the frame
# The button will now be positioned relative to the frame
b1.reparentTo(df)
# Same with the label
dl.reparentTo(df)

# The parent child relationship can be specified explicitly during widget
# construction.  df is passed in as the first argument, indicating that it
# is the parent of b2
# the text_fg takes (Red, Green, Blue, Opacity)
b2 = DirectButton(df, text = 'Child of df', text_fg = (1,0,0,1))
b2.setScale(.1)
b2.setPos(0,0,-.8)

# Note in the above constructor that you can modify characteristics of the
# widget's decorations using the component_characteristic notation,
# e.g. text_fg = (1,0,0,1).  These characteristics can also be modified
# after the fact:
b2['text_fg'] = (0,0,1,1)

# Direct Gui widgets can also contain 3D geometry:
```

```
happy = loader.loadModel('models/misc/smiley')
l2 = DirectButton(df, geom = happy, relief = RAISED, pad = (0.1, 0.1))

# Now add a command
def printHello():
    print 'Hello!'

l2['command'] = printHello

# If you have a handle to an interval for example (see Interval section)
# You could define a command to play the interval
def startPar():
    par.play()

l2['command'] = startPar

# NOTE: If you set the Configrc variable direct-gui-edit to #t you can move
# direct gui elements around with the middle mouse button.  Holding down
# the Control key during middle mouse motion allows one to adjust a widgets
# scale.  To find out the final attributes of a widget type:
l2.printConfig()


##################################################
# MATH
##################################################

# Panda has efficient vector and matrix operations built-in.

pos = Point3(1,2,3)
# To see what a point can do:
pdir(pos)

# Colors are 4 component vectors
colorVec = Vec4(1,1,1,1)
# To see vector operations:
pdir(colorVec)

mat = Mat4()
# To see what a mat (matrix) can do:
pdir(mat)


##################################################
# SOUND
##################################################

# We consider sampled sounds (wav,mp3) to be sound effects (sfx) and midi
# to be music. This way we can control the volume of the two types of audio
# independently.

s = base.loadSfx('phase_3.5/audio/dial/AV_duck_question.mp3')
base.playSfx(s)

m = base.loadMusic('phase_4/audio/bgm/trolley_song.mid')
base.playMusic(m)
```

```
################################################
# SHOWBASE
################################################

base

# base is the catchall global that creates and holds useful global
# methods for running Panda
# Code is in $DIRECT/src/showbase/ShowBase.py

base.loadSfx
base.playSfx
base.loadMusic
base.playMusic
base.toggleTexture
base.toggleWireframe
base.useDrive
base.useTrackball
base.screenshot

# It also creates these useful globals:
NAME       TYPE         DESCRIPTION
render     NodePath     Top of the rendered hierarchy
aspect2d   NodePath     Top of the 2D Gui hierarchy
hidden     NodePath     Top of the offscreen (not drawn) hierarchy
camera     NodePath     The camera
loader     Loader       loads models and textures
messenger  Messenger    handles event processing
taskMgr    TaskManager  Runs the task loop
ostream    Ostream      Panda C++ output stream
config     Config       Used to read Configrc variables
```